Table of Contents

Spin Transfer Torque

Version: 2016.3

Downloads & Links

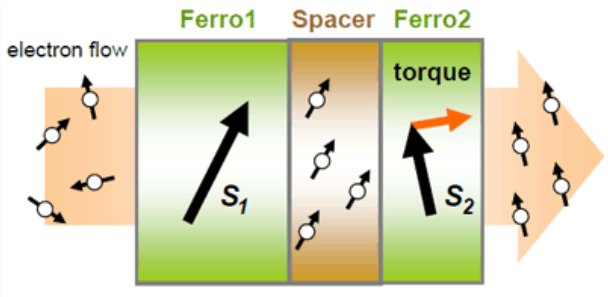
PDF version
para.py
theta-90.py
stt.py
angles.py
finite-bias.py
Basic QuantumATK Tutorial
ATK Reference Manual

In this tutorial you will learn about the spin transfer torque (STT) and how to calculate the STT in magnetic tunnel junctions.

We will use a simple model of such a tunnel juntion; a magnetic carbon-chain device. The QuantumATK package offers a convenient analysis object for calculating STT linear-response coefficients at zero bias, but you will also calculate the STT using finite-bias calculations and compare results to those obtained using linear response.

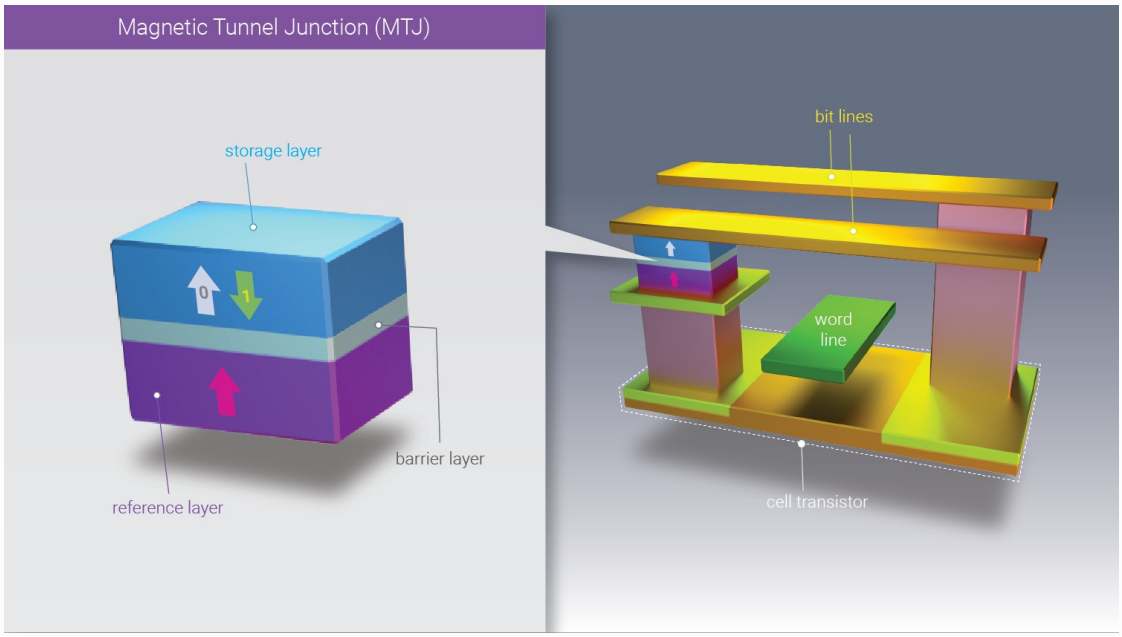
Introduction

Spin transfer torque occurs in situations where a current of spin-polarized carriers from the left part of a device with a particular polarization (given by the unit vector S_1) enters the right part of the device with a different magnetization direction (given by the unit vector S_2). When the electrons incoming from the left side enter the right magnetic part, they will eventually be polarized along S_2, meaning that the right magnetic domain has **exerted a torque** on the electrons in order to rotate their spin angular momentum. However, due to conservation of angular momentum, the electrons exert an **equal but opposite torque** on the right magnetic domain – the spin transfer torque. Note that a noncollinear description of electron spin is needed to study this effect!



Application in Random Access Memory (RAM)

The spin transfer torque can be used to modify the orientation of a magnetic layer in a magnetic tunnel junction (MTJ) by passing a spin-polarized current through it, and can therefore be used to flip the active elements in magnetic random-access memory (MRAM).



Theory

There are two principally different ways to calculate STT from atomic-scale models:

1. The STT can be found from the divergence of the spin current density, $\nabla \cdot I_s$, which in QuantumATK can be directly calculated using Green's function methods.
2. Another way to calculate the STT, here denoted T, is from the expression $\mathrm{T} = \mathrm{Tr}(\delta\rho_{\mathrm{neq}}\sigma \times \mathrm{B}_{\mathrm{xc}})$, where $\delta\rho_{\mathrm{neq}}$ is the non-equilibrium contribution to the density matrix, σ is a vector of Pauli matrices, and B_{xc} is the exchange-correlation magnetic field.

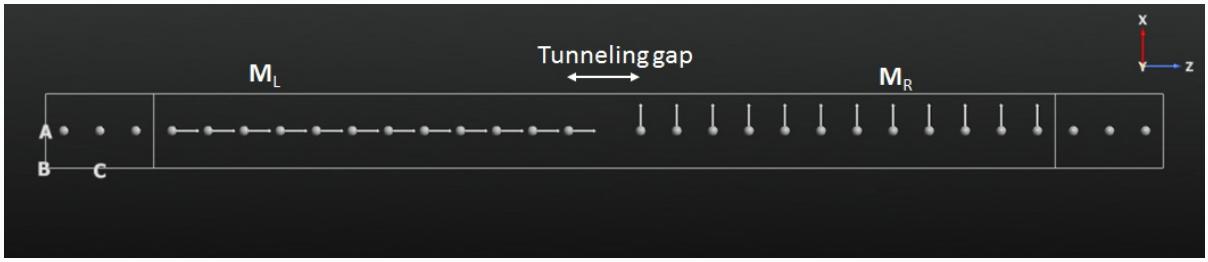
Method 2 will be used in this tutorial. We can either calculate $\delta\rho_{\mathrm{neq}}$ from the difference between finite-bias and zero-bias calculations, $\delta\rho_{\mathrm{neq}} = \rho_{\mathrm{neq}} - \rho_{\mathrm{eq}}$, or we can estimate it using linear response theory, $\delta\rho_{\mathrm{neq}} = (G^{\mathrm{r}}\Gamma_{\mathrm{L}}G^{\mathrm{a}})qU$, where $G^{\mathrm{r/a}}$ is the retarded/advanced Green's function, Γ_{L} is the left-electrode coupling operator, U is the bias voltage, and

q is the electron charge. The expression for the STT given in method 2 above is then evaluated in real space. See the related technical note for more information: TechNote.pdf.

ATK implements a convenient **analysis object** for calculating the spin transfer torque using method 2 in the linear-response approximation, where the non-equilibrium density response depends linearly on the bias.

Getting Started

You will here consider a device configuration consisting of a carbon chain with a gap in the middle, which the electrons have to tunnel through. The system is highly artificial, but serves as a simple model of a magnetic tunnelling junction.



Note

There is no need for k-point sampling in the A- and B-directions in this 1D device. However, for commonly studied magnetic tunnelling junctions, like Fe|MgO|Fe, a very fine k-point sampling along A and B is often needed.

Collinear Initial State

The 1D carbon-chain device illustrated above is similar to the one used in the tutorial Introduction to noncollinear spin, but not identical to it. Download the device configuration as an QuantumATK Python script: device.py.

As already mentioned, a noncollinear representation of the electron spin is needed when calculating STT. We will use a collinear, spin-parallel ground state as initial guess for the noncollinear state. First step is therefore a collinear calculation for the 1D device. Send the geometry to the **Script Generator** and set up the ATK-DFT calculation:

- Set the **default output file** to `para.nc`.
- Add a New Calculator block and set the **Spin** to *Polarized*. The **exchange-correlation** functional will automatically switch to *LSDA*. Change the carbon basis set to *SingleZetaPolarized* in order to speed up calculations.
- Add an Initial State block and select **User spin** in order to initialize the calculation with all carbon atoms maximally polarized.

Save the script as `para.py` (it should look roughly like this: para.py), and run the calculation using the **Job Manager**.

Noncollinear State: 90° Spin Rotation

Next step is a noncollinear calculation where the spins on the left side of the tunnelling gap point in a different direction than the spins on the right side of the gap. The basic recipe to do this is fairly simple:

1. The ATK-DFT calculator settings used in the collinear calculation, saved in `para.nc`, are loaded from

file and slightly adapted for the noncollinear calculation.

2. The spins in the right-hand part of the device are rotated 90° around the polar angle θ.
3. The spin-parallel collinear ground state is used as initial state for the noncollinear calculation.

The QuantumATK Python script shown below implements the basic steps given above, but also runs a Mulliken population analysis for visualizing the resulting spin directions throughout the device, and computes the electrostatic difference potential (EDP) and the transmission spectrum.

```
# Read in the collinear calculation
device_configuration = nlread('para.nc', DeviceConfiguration)[0]

# Use the special noncollinear mixing scheme
iteration_control_parameters = IterationControlParameters(
    algorithm=PulayMixer(noncollinear_mixing=True),
)

# Get the calculator and modify it for noncollinear LDA
calculator = device_configuration.calculator()
calculator = calculator(
    exchange_correlation=NCLDA.PZ,
    iteration_control_parameters=iteration_control_parameters,
)

# Define the spin rotation in polar coordinates
theta = 90*Degrees
left_spins  = [(i, 1, 0*Degrees, 0*Degrees) for i in range(12)]
right_spins = [(i, 1, theta,     0*Degrees) for i in range(12,24)]
spin_list   = left_spins + right_spins
initial_spin = InitialSpin(scaled_spins=spin_list)

# Setup the initial state as a rotated collinear state
device_configuration.setCalculator(
    calculator,
    initial_spin=initial_spin,
    initial_state=device_configuration,
)

# Calculate and save
device_configuration.update()
nlsave('theta-90.nc', device_configuration)

# ------------------------------------------------------------
# Mulliken Population
# ------------------------------------------------------------
mulliken_population = MullikenPopulation(device_configuration)
nlsave('theta-90.nc', mulliken_population)
nlprint(mulliken_population)

# ------------------------------------------------------------
# Electrostatic Difference Potential
# ------------------------------------------------------------
electrostatic_difference_potential = ElectrostaticDifferencePotential(device_configuration)
nlsave('theta-90.nc', electrostatic_difference_potential)

# ------------------------------------------------------------
# Transmission Spectrum
# ------------------------------------------------------------
kpoint_grid = MonkhorstPackGrid(
    force_timereversal=False
)

transmission_spectrum = TransmissionSpectrum(
    configuration=device_configuration,
    energies=numpy.linspace(-2,2,101)*eV,
    kpoints=kpoint_grid,
    energy_zero_parameter=AverageFermiLevel,
    infinitesimal=1e-06*eV,
    self_energy_calculator=RecursionSelfEnergy(),
    )
nlsave('theta-90.nc', transmission_spectrum)
```

A special density mixer for noncollinear calculations is employed, and the exchange-correlation is changed to `NCLDA.PZ`. Note how the `InitialSpin` object is used to define the initial spin directions on the

calculator.

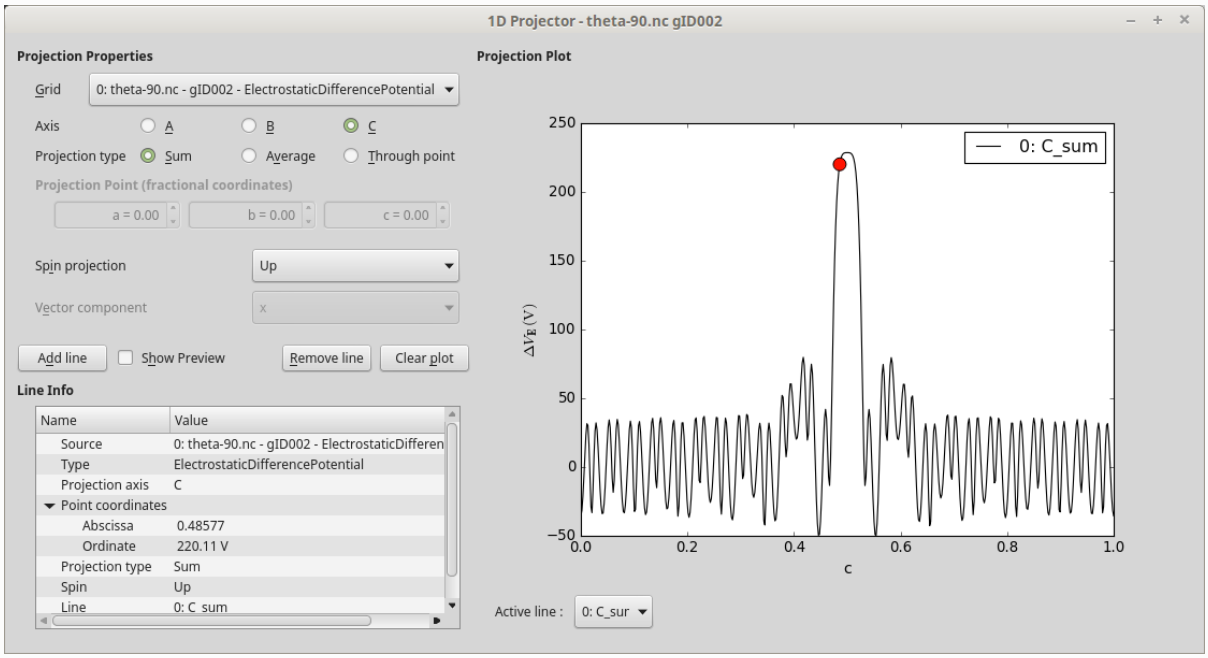
Save the script as `theta-90.py` and run it using the **Job Manager** – it should not take long to finish.

Visualizations

When the calculation finishes, the file `theta-90.nc` should appear in the QuantumATK Project Files list and the contents of it should be available on the LabFloor:

- DeviceConfiguration
- $\mathrm{Tr}(\rho\mathbf{S})$ MullikenPopulation
- $\delta V_{\mathbf{E}}(\mathbf{r})$ ElectrostaticDifferencePotential
- $T(\mathbf{k},\epsilon)$ TransmissionSpectrum

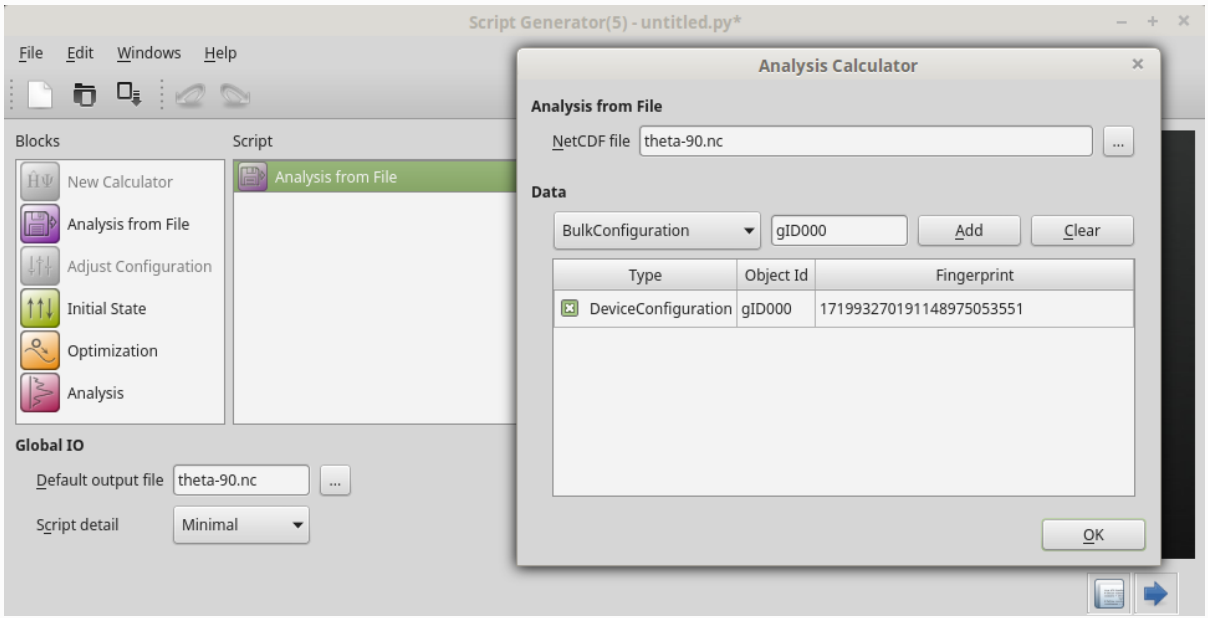
First of all, select the **electrostatic difference potential** and plot it using the **1D Projector** in order to check that the NEGF calculation converged to a well-behaved ground state: Project the average EDP onto the C axis, and observe that the ground state electrostatic potential is nicely periodic in both ends of the device toward the electrodes, as it should be, and has the expected non-periodic feature around the gap in the middle of the device.



Next, select the **Mulliken population** analysis item and open it in the **Viewer**. Use the *Camera* options (or click) to select the **ZX plane** for a clear view of the spin orientations.

Calculate the STT

You are now ready to use linear response (LR) theory to calculate the spin transfer torque. Open a new **Script Generator** window and add the **Analysis from File** block. Open the block and select the device configuration saved in `theta-90.nc` for post-SCF analysis. Then change the Script Generator default output file to `theta-90.nc` in order to save all calculated quantities in that file.

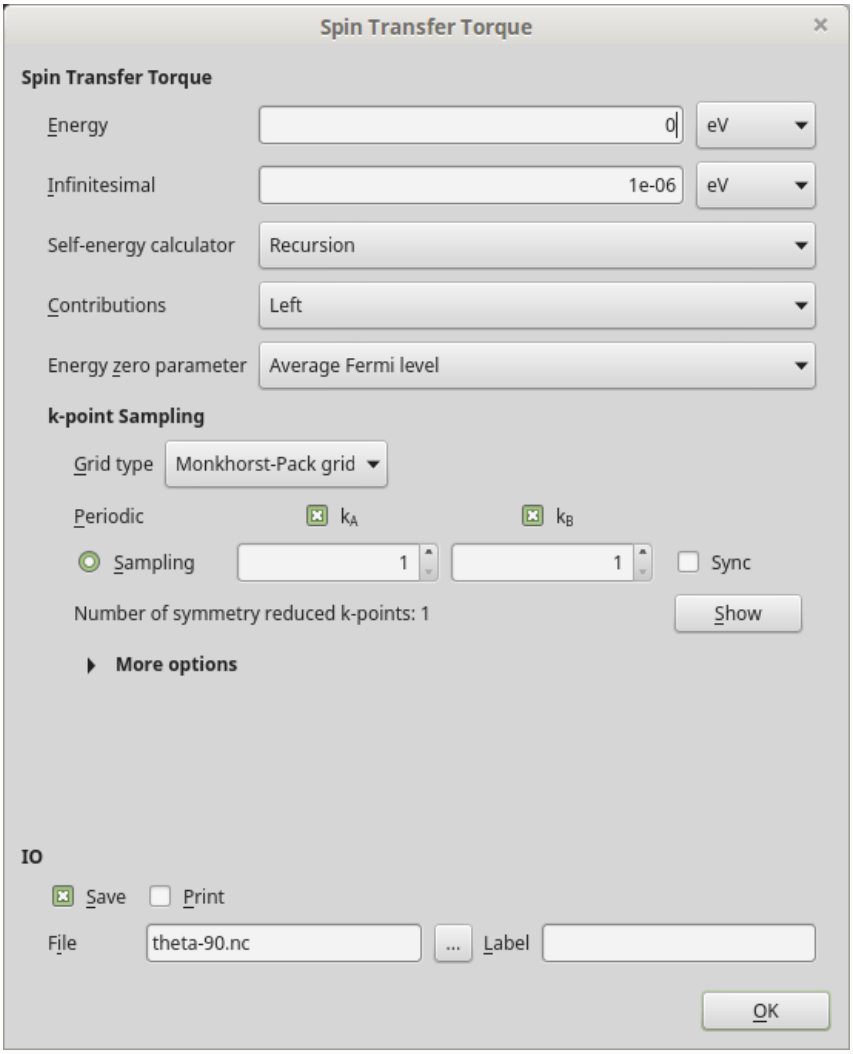


Add also the Analysis ▸ SpinTransferTorque analysis block, and open it to see the available options. Note in particular 3 important settings:

Energy: The electron energy, with respect to the Fermi level, for which the STT will be calculated.

Contributions: The STT is by default calculated for the left –> right current.

k-points: Fairly dense k-point grids are sometimes needed along periodic directions orthogonal to the transport direction.



Tip

See the QuantumATK Manual entry `SpinTransferTorque` for a full description of all input parameters for the STT calculation and all QuantumATK Python methods available on the object.

Leave all the STT options at defaults and save the script as `stt.py`. The script should look roughly like this: stt.py. Run the calculation using the **Job Manager** – it should finish in less than a minute.

The STT analysis object should now have been added to the file `theta-90.nc` and be available on the QuantumATK LabFloor:

- $\tau_i(\mathbf{r})$ SpinTransferTorque

Important

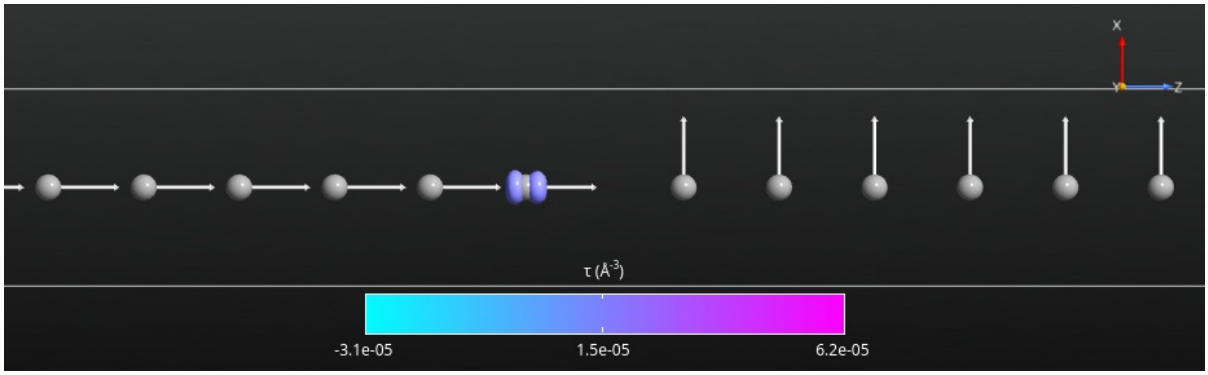
The spin transfer torque depends on the bias voltage across the electrodes, and is of course zero at zero bias, since no current flows. The linear-response spin transfer torque (LR-STT) assumes a linear relationship between the STT and the bias voltage
V,

$$\mathrm{T}(V) = V \cdot \left.\frac{\partial \mathrm{T}}{\partial V}\right|_{V=0} = V \cdot \tau,$$

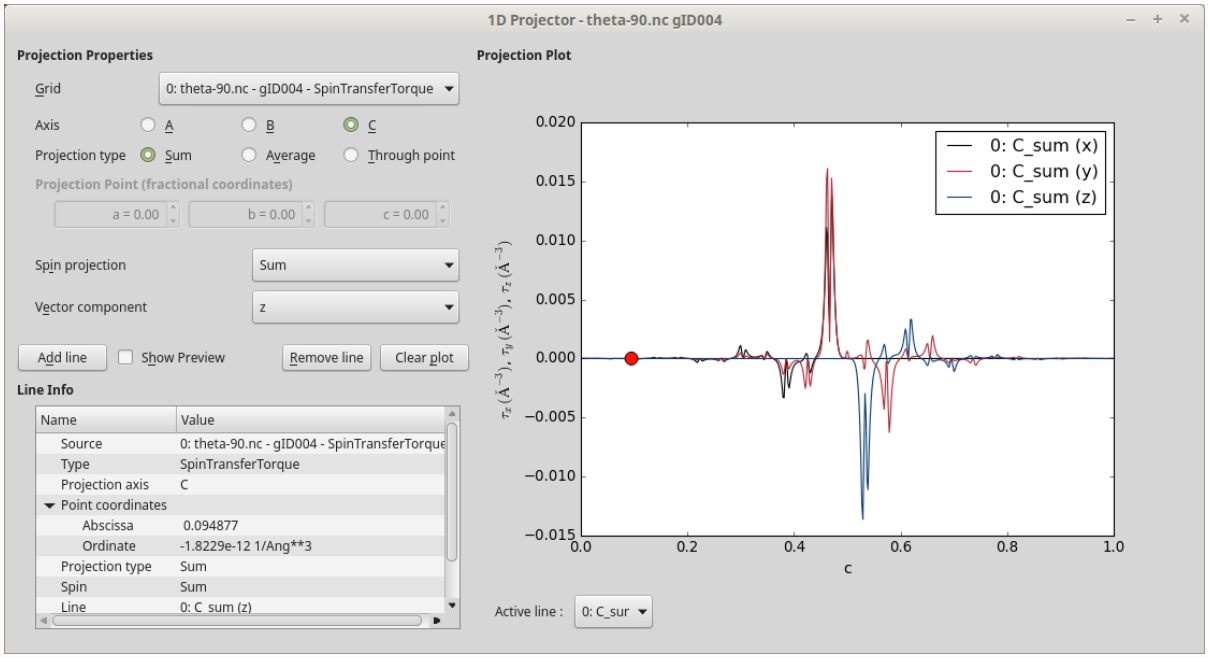
where
τ is a **linear-response coefficient**.

The QuantumATK SpinTransferTorque analysis object computes the local components of the LR-STT coefficient
τ, from which the small-bias STT may be calculated.

Use first the **Viewer** to get a 3D visualization of the values of the LR-STT coefficient for θ=90°: Open the Mulliken population in the Viewer, and then drag-and-drop the STT item onto the Viewer window. The coefficient
τ is represented on a **3D vector grid**, each vector consisting of the local x-, y-, and z-components, so you need to choose which component to visualize. Choose in this case to visualize the y-component as an isosurface:



You can also use the **1D Projector** to visualize the STT components. Select the STT item on the LabFloor, and click the 1D Projector plugin. Plot the x, y, and z vector components in the same plot, projected on the C axis.



It is common to divide the STT into **out-of-plane** and **in-plane** contributions. In the present case, the magnetizations in the left and right electrodes point in the z- and x-directions, respectively, thus defining the in-plane contribution as being in the XZ-plane, while the y-component gives the out-of-plane

contribution.

The two figures below show the out-of-plane and in-plane components of the LR-STT coefficient. The spin of the current is oriented along z when entering the right part of the device, and is then turned such as to be aligned along x. The x-component of
τ is therefore non-zero only in the left-hand part of the device, while the z-component is non-zero only in the right-hand part. The two components are in this case mirror images of each other.

Tip

Both figures above were generated using the 1D Projector. If you right-click the plot and select **Customize**, a dialog appears where you can customize plot details such as title, legend, colors, etc.

Angle Dependence

Let us next investigate how the LR-STT coefficient depends on the angle
θ in the interval 0° to 180°. What is needed is a range of range of zero-bias NEGF calculations for different values of
θ, each followed by calculation of the SpinTransferTorque analysis object for that particular spin configuration. This is most easily achieved by combining the main parts of the scripts `theta-90.py` and `stt.py` used above:

```
thetas = [0,10,20,30,40,50,60,70,80,100,110,120,130,140,150,160,170,180]

for theta in thetas:
    # Output data file
    ncfile = 'theta-%i.nc' % theta

```

```
    # Read in the collinear calculation
    device_configuration = nlread('para.nc', DeviceConfiguration)[0]

    # Use the special noncollinear mixing scheme
    iteration_control_parameters = IterationControlParameters(
        algorithm=PulayMixer(noncollinear_mixing=True),
    )

    # Get the calculator and modify it for noncollinear LDA
    calculator = device_configuration.calculator()
    calculator = calculator(
        exchange_correlation=NCLDA.PZ,
        iteration_control_parameters=iteration_control_parameters,
    )

    # Define the spin rotation in polar coordinates
    left_spins  = [(i, 1, 0*Degrees,    0*Degrees) for i in range(12)]
    right_spins = [(i, 1, theta*Degrees, 0*Degrees) for i in range(12,24)]
    spin_list   = left_spins + right_spins
    initial_spin = InitialSpin(scaled_spins=spin_list)

    # Setup the initial state as a rotated collinear state
    device_configuration.setCalculator(
        calculator,
        initial_spin=initial_spin,
        initial_state=device_configuration,
    )

    # Calculate and save
    device_configuration.update()
    nlsave(ncfile, device_configuration)

    # ------------------------------------------------------------
    # Mulliken Population
    # ------------------------------------------------------------
    mulliken_population = MullikenPopulation(device_configuration)
    nlsave(ncfile, mulliken_population)
    nlprint(mulliken_population)

    # ------------------------------------------------------------
    # Transmission Spectrum
    # ------------------------------------------------------------
    kpoint_grid = MonkhorstPackGrid(
        force_timereversal=False
    )

    transmission_spectrum = TransmissionSpectrum(
        configuration=device_configuration,
        energies=numpy.linspace(-2,2,101)*eV,
        kpoints=kpoint_grid,
        energy_zero_parameter=AverageFermiLevel,
        infinitesimal=1e-06*eV,
        self_energy_calculator=RecursionSelfEnergy(),
        )
    nlsave(ncfile, transmission_spectrum)

    # ------------------------------------------------------------
    # Spin Transfer Torque
    # ------------------------------------------------------------
    kpoint_grid = MonkhorstPackGrid(
        force_timereversal=False,
        )

    spin_transfer_torque = SpinTransferTorque(
        configuration=device_configuration,
        energy=0*eV,
        kpoints=kpoint_grid,
```

```
        contributions=Left,
        energy_zero_parameter=AverageFermiLevel,
        infinitesimal=1e-06*eV,
        self_energy_calculator=RecursionSelfEnergy(),
        )
    nlsave(ncfile, spin_transfer_torque)
```

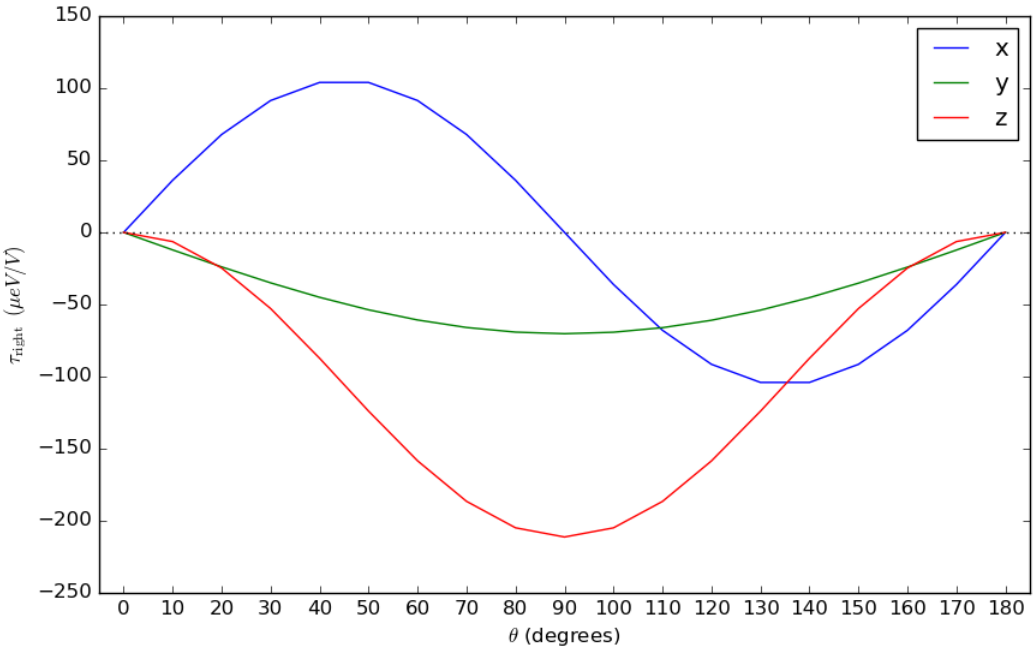
The Mulliken population and transmission spectrum are not strictly needed, but will prove useful for further analysis later on. Notice that the case
θ=90° is skipped because that calculation was already done above.

Download the script, angles.py, and run it using the **Job Manager** – it may take up to 30 minutes to complete.

When the job finishes you have a number of new nc-files on the QuantumATK LabFloor, each for a particular spin angle. You may visualize the Mulliken populations to check for yourself that the spin angle varies correctly.

Use the script angles_plot.py to analyze and plot the results. For each angle, it sums the x-, y- and z-components of
τ in the right-hand part of the device, and plots them against the angle:

```
# Prepare lists
thetas = [0,10,20,30,40,50,60,70,80,90,100,110,120,130,140,150,160,170,180]
x = []
y = []
z = []

# Read data
for theta in thetas:
    # Data file
    ncfile = 'theta-%i.nc' % theta

    # Read in the STT analysis object
    stt = nlread(ncfile, SpinTransferTorque)[0]

    # Get the STT 3D vector grid (units Bohr**-3)
    array = stt.toArray()

    # Get the index of middle position along C
    sh = numpy.shape(array)
    k = sh[2]/2

    # Get the volume element of the STT grid.
    dX, dY, dZ = stt.volumeElement()
    volume = numpy.abs(numpy.dot(dX, numpy.cross(dY, dZ)))

    # Integrate the vector components in the right-hand part of the device
    stt_x = numpy.sum(array[:,:,k:,:,0])*volume*stt.unit()
    stt_y = numpy.sum(array[:,:,k:,:,1])*volume*stt.unit()
    stt_z = numpy.sum(array[:,:,k:,:,2])*volume*stt.unit()

    # append to lists
    x.append(stt_x)
    y.append(stt_y)
    z.append(stt_z)

# Convert lists to arrays
x = numpy.array(x)
y = numpy.array(y)
z = numpy.array(z)

# Save data for future convenience
import pickle
f = open('angles_plot.pckl', 'w')
pickle.dump((thetas,x,y,z), f)
f.close()

# Plot results
import pylab
pylab.figure(figsize=(10,6))
pylab.plot(thetas, x*1e6, label='x')
pylab.plot(thetas, y*1e6, label='y')
pylab.plot(thetas, z*1e6, label='z')
pylab.axhline(0, color='k', linestyle=':')
pylab.legend()
pylab.xlabel(r'$\theta$ (degrees)')
pylab.ylabel(r'$\tau_\mathrm{right} \, \, \, (\mu eV/V)$')
ax = pylab.gca()
ax.set_xlim((-5,185))
ax.set_xticks(thetas)
pylab.savefig('angles_plot.png')
pylab.show()
```



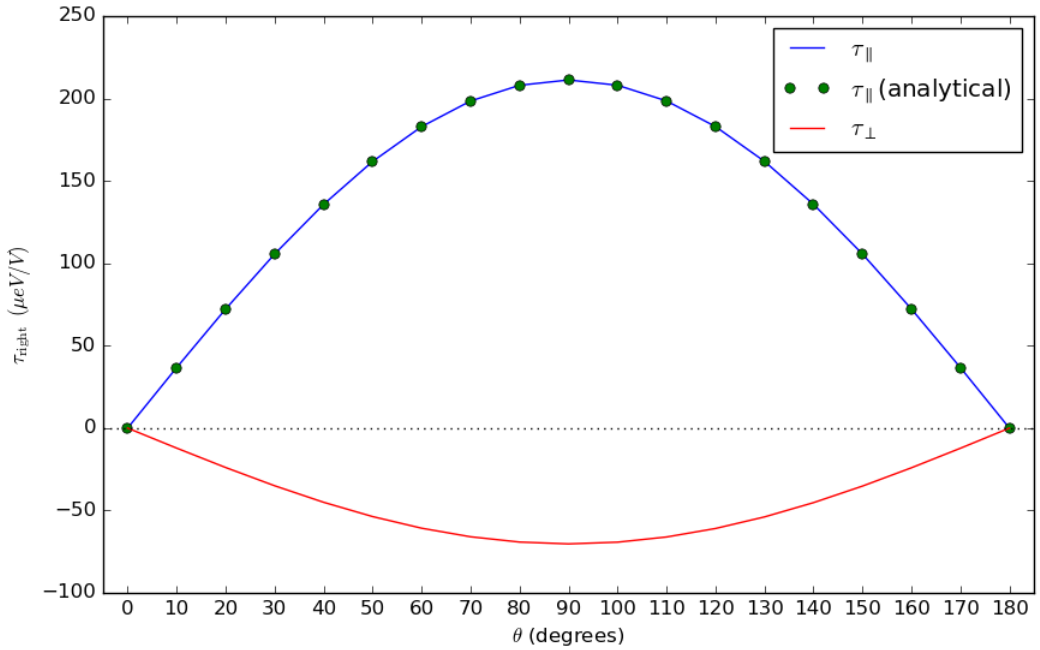
The in-plane STT may also be compared to an analytic expression from Ref. [TKK+06]:

$$\tau_{\parallel}(\theta) = 0.5 \cdot [T_z(180^\circ) - T_z(0^\circ)]\, \mathrm{M}_L \times (\mathrm{M}_R \times \mathrm{M}_L),$$

where
$T_z(180^\circ) = h/(4\pi e)[T_\uparrow(180^\circ) - T_\downarrow(180^\circ)]$ is the spin transmission for the anti-parallel configuration, and likewise for
$T_z(0^\circ)$. The vector
$\mathrm{M}_{L/R}$ is the direction of magnetization in the left/right part of the carbon chain.

Use the script analytical.py to do the analysis. It computes the in-plane (
$\tau_{\parallel} = \sqrt{\tau_x^2 + \tau_z^2}$) and out-of-plane (
$\tau_\perp = \tau_y$) components in the right-hand part of the device, and also computes
$\tau_{\parallel}$ from the analytical expression using the obtained transmission spectrum at each angle:

```
# Load STT data
import pickle
f = open('angles_plot.pckl', 'r')
(thetas,x,y,z) = pickle.load(f)
f.close()

# Process STT data
inplane = (x**2 + z**2)**0.5
outplane = y

# Analytical in-plane STT
transmissions = numpy.zeros(2)
for j,theta in enumerate([0,180]):
    # Data file
    ncfile = 'theta-%i.nc' % theta
    # Read the transmission spectrum
    transmission = nlread(ncfile, TransmissionSpectrum)[0]
    # Get the energies
    energies = transmission.energies().inUnitsOf(eV)
    # Find the index of the Fermi level
    i_Ef = numpy.argmin(abs(energies))
    # Calculate the spin-transmission (Up - Down)
    T = transmission.evaluate(spin=Spin.Up)[i_Ef]-transmission.evaluate(spin=Spin.Down)[i_Ef]
    # Append to list
    transmissions[j] = T
analytical = abs(transmissions[0]-transmissions[1])/2*numpy.sin(numpy.array(thetas)*numpy.pi/180)

# Plot results
import pylab
pylab.figure(figsize=(10,6))
pylab.plot(thetas, inplane*1e6,         label=r'$\tau_\parallel$')
pylab.plot(thetas, analytical*1e6, 'o', label=r'$\tau_\parallel$'+'(analytical)')
pylab.plot(thetas, outplane*1e6,        label=r'$\tau_\perp$')
pylab.axhline(0, color='k', linestyle=':')
pylab.legend()
pylab.xlabel(r'$\theta$ (degrees)')
pylab.ylabel(r'$\tau_\mathrm{right} \, \, \, (\mu eV/V)$')
ax = pylab.gca()
ax.set_xlim((-5,185))
ax.set_xticks(thetas)
pylab.savefig('analytical.png')
pylab.show()
```



The agreement between the angle-dependent calculation (blue line) and the analytical results (green dots), obtained only from the spin transmissions in the parallel and anti-parallel configurations, is excellent.

Finite-Bias Calculations

You will now go beyond the linear response approximation and explicitly evaluate the STT by performing finite-bias NEGF calculations. We shall focus on the θ=90° spin configuration, where the STT is largest.

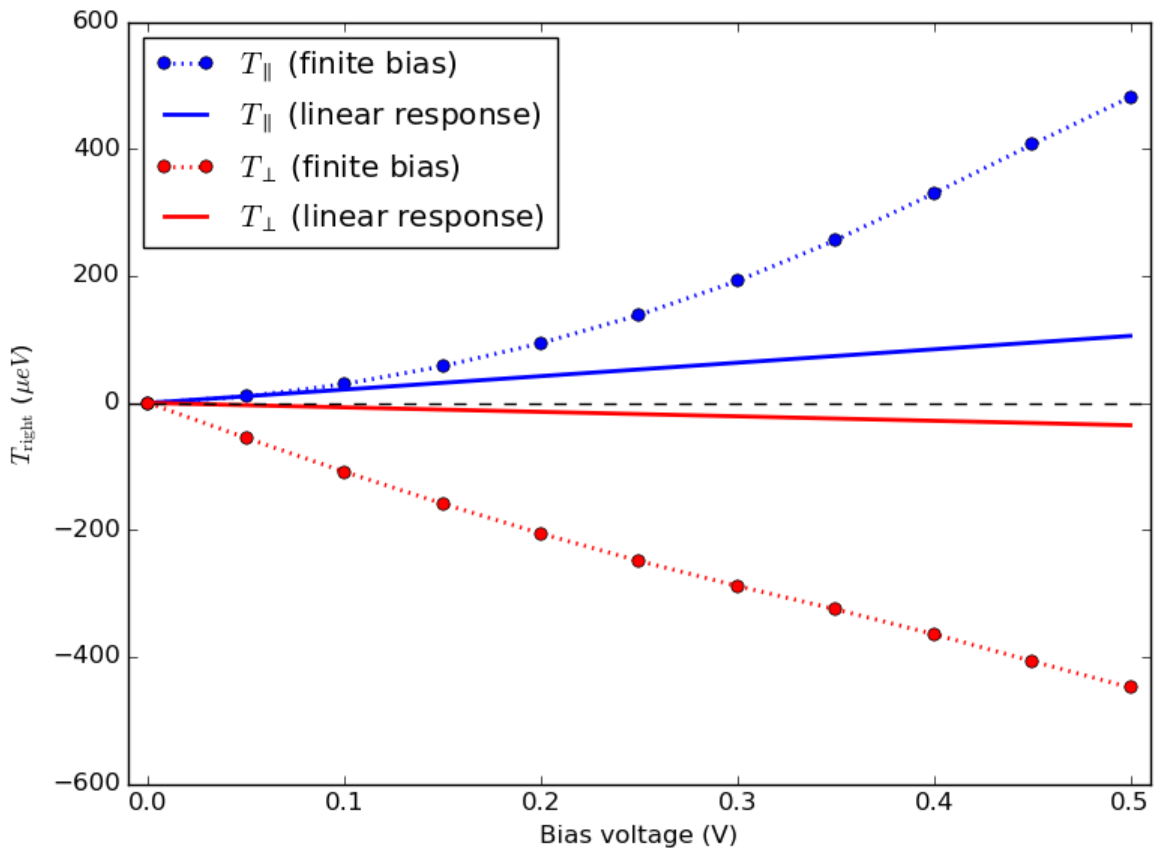
The script below sets up and runs a series of finite-bias NEGF calculations for bias voltages in the range 0–0.5 Volt. Each finite-bias NEGF calculation uses the ground state from the previous calculation as an initial guess for the new ground state. Notice that the script uses a finer than default calculator resolution for evaluating the `non_equilibrium_contour`, which improves convergence as well as accuracy of the results. At each bias point, the script also calculates a few QuantumATK analysis objects needed for explicitly computing the spin transfer torque. Notice also that the amount of information written to the QuantumATK log file is reduced by using the `setVerbosity` functionality.

```
# Reduce the QuantumATK log output
setVerbosity(MinimalLog)

# Range of bias points
biases = numpy.linspace(0.0, 0.5, 11)*Volt

# Read in the zero-bias calculation
device_configuration = nlread('theta-90.nc', DeviceConfiguration)[0]

# Loop over bias points
for bias in biases:
    # Filename for saving results
    ncfile = 'theta-90_bias_%2.2f.nc' % bias.inUnitsOf(Volt)

    # Setup accurate non-equilibrium contour integral for finite-bias
    non_equilibrium_contour = RealAxisContour(
        real_axis_point_density=0.0001*Hartree,
        real_axis_infinitesimal=1e-08*Hartree,
        )
    contour_parameters = ContourParameters(
```

```
        non_equilibrium_contour=non_equilibrium_contour,
        )

    # Get the calculator and modify it for finite-bias
    calculator = device_configuration.calculator()
    calculator = calculator(
        contour_parameters=contour_parameters,
        electrode_voltages=(0.0*Volt, bias),
    )

    # Use the previous converged ground state as initial guess.
    device_configuration.setCalculator(
        calculator,
        initial_state=device_configuration,
    )

    # Calculate and save
    device_configuration.update()
    nlsave(ncfile, device_configuration)

    # ------------------------------------------------------------
    # Mulliken Population
    # ------------------------------------------------------------
    mulliken_population = MullikenPopulation(device_configuration)
    nlsave(ncfile, mulliken_population)

    # ------------------------------------------------------------
    # Transmission Spectrum
    # ------------------------------------------------------------
    kpoint_grid = MonkhorstPackGrid(
        force_timereversal=False
    )

    transmission_spectrum = TransmissionSpectrum(
        configuration=device_configuration,
        energies=numpy.linspace(-2,2,101)*eV,
        kpoints=kpoint_grid,
        energy_zero_parameter=AverageFermiLevel,
        infinitesimal=1e-06*eV,
        self_energy_calculator=RecursionSelfEnergy(),
        )
    nlsave(ncfile, transmission_spectrum)

    # ------------------------------------------------------------
    # Exchange correlation potential
    # ------------------------------------------------------------
    exchange_correlation_potential = ExchangeCorrelationPotential(device_configuration)
    nlsave(ncfile, exchange_correlation_potential)

    # ------------------------------------------------------------
    # Electron density
    # ------------------------------------------------------------
    electron_density = ElectronDensity(device_configuration)
    nlsave(ncfile, electron_density)
```

Download the script, finite-bias.py, and run it using the **Job Manager**. The calculations may take quite a while (30–60 minutes), so be patient!

The nc-files containing the finite-bias results should now be available on the QuantumATK LabFloor. Use a custom script to evaluate and plot the total STT in the right-hand side of the device, and compare the finite-bias results to the linear-response ones previously calculated: finite-bias_plot.py.



The figure above is produced – it shows the in-plane (blue) and out-of-plane (red) components of the STT in the right part of the carbon chain. Results from finite-bias calculations at each bias point (circles) are compared to the linear responce results (full lines) obtained from applying the linear-response approximation
$\mathrm{T} = V\tau$ for each bias.

It is evident that the linear response results are only in agreement with the finite-bias values for very low bias voltages; below ~0.1 Volt for the in-plane component, and even lower voltages for the out-of-plane component.

References

[TKK+06] I. Theodonis, N. Kioussis, A. Kalitsov, M. Chshiev, and W. H. Butler. Anomalous bias dependence of spin torque in magnetic tunnel junctions. *Phys. Rev. Lett.*, 97:237205, Dec 2006. doi:10.1103/PhysRevLett.97.237205.