BORIS Computational Spintronics

User manual, *version 3.80*

Dr. Serban Lepadatu, 22nd *February 2023*

Abstract

This manual describes BORIS Computational Spintronics, a multi-physics and multi-scale research software designed to solve three-dimensional magnetization dynamics problems, coupled with a self-consistent charge and spin transport solver, heat flow solver with temperature-dependent material parameters, and elastodynamics solver including thermoelastic and magnetoelastic/magnetostriction effects, in arbitrary multi-layered structures and shapes. The computational routines run both on central processors and graphics processors using the CUDA platform. In addition to simple user control, advanced simulation configurations are made possible using Python scripts. The software is open source and currently runs on Windows 7, Windows 10, and Linux-based 64-bit operating systems, and was programmed using C++17 and Python.

boris-spintronics.uk/download

https://github.com/SerbanL/Boris2

https://groups.google.com/forum/#!forum/boris-computational-spintronics

Disclaimer

Contents

Installation - Windows

An installer has been provided with the program and instructions therein should be followed. The program is designed to run on Windows 7, Windows 10, and Windows 11, 64 bit versions, and requires Microsoft Visual C++ 2017 Redistributable (x64) – included with the installer. On Windows 10 the executable (Boris.exe) must be run in compatibility mode – the installer sets this.

CUDA

To enable CUDA computations Boris requires a CUDA-enabled graphics card with CUDA compute capability 5.0 or greater. The installer detects the CUDA compute capability of the graphics card and launches the required program version. **You should always run the installed Boris.exe program**, not the separate CUDA versions found in the same directory.

The following architectures are supported by the installer package: Maxwell (sm_50, sm_52, sm_53), Pascal (sm_60, sm_61, sm_62), Volta (sm_70, sm_72), Turing (sm_75), Ampere (sm_80, sm_86), Ada and Hopper (sm_90). CUDA Toolkit 12.0 has been used to compile.

Directories

BORIS is installed in C:\Program Files (x86)\Boris

BORIS user data is contained in C:\Users\(UserName)\Documents\Boris Data

Boris Data folder contains:

- BorisPythonScripts folder, which contains required Python files for simulation scripting, in particular NetSocks.py which is required for all Python scripts.
- Examples folder, with worked solutions for the Tutorials in this Manual.
- Simulations folder, which is the default folder for saving and loading simulation files.
- Manual, BorisMDB.txt materials database, errorlog.txt showing the errors log, startup.txt with BORIS startup configuration options.

Installation – Linux-based OS

Extract the archive. On Linux-based OS the program needs to be compiled from source using the provided *makefile* in the extracted BorisLin directory.

Make sure you have all the required updates and dependencies:

<u>Step 0: Updates.</u>

1. Get latest g++ compiler: *$ sudo apt install build-essential*
2. Get OpenMP: *$ sudo apt-get install libomp-dev*
3. Get LibTBB: *$ sudo apt install libtbb-dev*
4. Get latest CUDA Toolkit by following instructions at: CUDA Toolkit Downloads

   For example using the deb(network) method, the following instructions may be used:

```
$ wget https://developer.download.nvidia.com/compute/cuda/repos/ubuntu2004/x86_64/cuda-keyring_1.0-1_all.deb
$ sudo dpkg -i cuda-keyring_1.0-1_all.deb
$ sudo apt-get update
$ sudo apt-get -y install cuda
```

   NOTES:

   If at any point it is necessary to re-install the CUDA toolkit, it is recommended any existing CUDA installation is removed. This may be done using the following:

```
$ sudo apt-get purge nvidia*
$ sudo apt-get autoremove
$ sudo apt-get autoclean
$ sudo rm -rf /usr/local/cuda*
```

   FURTHER NOTES (IMPORTANT):

   It may be necessary to set the export path for the CUDA dynamically linked library. This may be done by editing the bash script:

$ nano /home/$USER/.bashrc

When bash file opens, add there the following lines (e.g. at the end), but replace 12.0 with correct CUDA version you installed:

export PATH="/usr/local/cuda-12.0/bin:$PATH"

export LD_LIBRARY_PATH="/usr/local/cuda-12.0/lib64:$LD_LIBRARY_PATH"

Ctrl + o : Save

Enter or Return key : Accept changes

Ctrl + x : Close editor

Close terminal and start another one.

Now nvcc --version should show correct version.

The include and library directories for CUDA are as follows, and these are expected by *makefile*:

Include directory for compilation:

-I/usr/local/cuda-12.0/targets/x86_64-linux/include

Library directory for linking:

-L/usr/local/cuda-12.0/targets/x86_64-linux/lib

TROUBLESHOOTING:

There are known incompatibility issues between certain CUDA Toolkit versions and *gcc* compiler versions. For example the combination *gcc* 11.3 and CUDA 11.5 will not compile. In this case it is necessary to upgrade to latest CUDA version.

Please report any incompatibilities and issues here:

https://groups.google.com/forum/#!forum/boris-computational-spintronics

5. Get and install FFTW3: Instructions at http://www.fftw.org/fftw2_doc/fftw_6.html
6. Get Python3 development version, required for running Python scripts in embedded mode. To get Python3 development version:

   $ sudo apt-get install python-dev

NOTES:

The following directories are expected by *makefile*, e.g. for Python3.8 version:

Include directory for compilation:

-I/usr/include/python3.8

Open terminal and go to extracted BorisLin directory.

Step 1: Configuration.

$ make configure (arch=xx) (sprec=0/1) (python=x.x) (cuda=x.x)

Before compiling you need to set the correct CUDA architecture for your NVidia GPU.

For a list of architectures and more details see: https://en.wikipedia.org/wiki/CUDA.
Possible values for **arch** are:

- *arch=50* is required for **Maxwell** architecture; translates to *-arch=sm_50* in nvcc compilation.
- *arch=60* is required for **Pascal** architecture; translates to *-arch=sm_60* in nvcc compilation.
- *arch=70* is required for **Volta** (and **Turing**) architecture; translates to *-arch=sm_70* in nvcc compilation.
- *arch=80* is required for **Ampere** architecture; translates to *-arch=sm_80* in nvcc compilation.
- *arch=90* is required for **Ada** (and **Hopper**) architecture; translates to *-arch=sm_90* in nvcc compilation.

sprec sets either single precision (1) or double precision (0) for CUDA computations.

python is the Python version installed, e.g. 3.8

cuda is the CUDA Toolkit version installed, e.g. 12.0.

Example: *$ make configure arch=80 sprec=1 python=3.8 cuda=12.0*

If *arch* is not specified a default value of 50 is used.

You can also compile CUDA code with single or double precision floating point. The default value, if not specified, is *sprec=1* (single precision – recommended for most users). If you have a GPU capable of handling double precision floating point efficiently you can configure with *sprec=0*.

Step 2: Compilation.

$ make compile -j N

(replace N with the number of logical cores on your CPU for multi-processor compilation, e.g. ***$ make compile -j 16***)

Step 3: Installation.

$ make install

Step4: Run.

$ ./BorisLin

Notes:

In the current version Boris runs in text mode only on Linux-based OS, thus the GRAPHICS 0 value needs to be kept in *CompileFlags.h* file. In a future version the graphical interface will also be available under Linux-based OS.

Directories

BORIS is installed in the extracted BorisLin directory. This folder contains the compiled executable, makefile, required Python files for simulation scripting, in particular NetSocks.py which is required for all Python scripts. It also contains a number of configuration files.

The code is contained in Boris folder, with additional library files in BorisLib and BorisCUDALib folders.

BORIS user data is contained in: …\BorisLin\Boris_Data
Boris_Data folder contains:

- Examples folder, with worked solutions for the Tutorials in this Manual.
- Simulations folder, which is the default folder for saving and loading simulation files.

Manual, BorisMDB.txt materials database, errorlog.txt showing the errors log, startup.txt with BORIS startup configuration options.

CompileFlags.h file:

There are a number of compilation options, described there, found in the Boris folder. In most cases this shouldn't be changed. Some special cases include:

- Set PYTHON_EMBEDDING to 0 if embedded Python scripting is not required. With this setting Python3 headers and libraries will not be required for compilation and linking. This could be useful for troubleshooting, but otherwise leave this value to 1.

- Set COMPILECUDA to 0 if CUDA computations are not required. In this mode only CPU computations will be available. This could be useful for troubleshooting, but otherwise leave this value to 1.

- GRAPHICS is set to 0 for compiling under Linux-based OS (this should already be set, so does not need to be changed).

Version Updates

The program version may be updated from the console using the **versionupdate** command, without having to download the entire installer again. Instead, incremental patches are downloaded and applied. Currently this only works for Windows. This is intended mainly for small updates, e.g. bug fix patches. The latest version 3.8 requires full download and install.

1) Check if incremental updates are available, but take no other action:

```
versionupdate check
```

2) Download any available and required incremental updates, but do not install:

```
versionupdate download
```

3) Download any available and required incremental updates, and install the latest version:

```
versionupdate install
```

Using this command automatically downloads, installs all required incremental patches, then automatically restarts the program which should now be the latest available version.

4) If you want to only update to a given version, which may not be the latest version then specify it as a parameter, e.g. v3.5 would be 350, etc:

```
versionupdate install 350
```

5) If you want to roll back to an earlier version, e.g. back to v3.4 then specify it as:

```
versionupdate rollback 340
```

Note this command assumes v3.4 is present on the working machine, which typically means the software was previously updated from this version to a newer one.

6) If you want to simply check if your version is the latest version then use the **checkupdates** command. To have the program automatically check for updates whenever it is started then use (from v3.4 this is disabled by default, but if you’ve previously installed an earlier version this might still be enabled in the local *startup.txt* configuration file):

```
startupupdatecheck 1
```

Overview

This manual contains a set of self-teaching tutorials that guide the user through most of its functionality. The tutorials contain a number of exercises designed for users without a background in micromagnetics, and may be skipped by more advanced users. A number of examples that accompany the tutorials have also been provided in the accompanying Examples folder.

You can use Tutorial 0 as a quick-start. This tutorial contains a number of Python scripts as examples but doesn't contain in-depth explanations.

Tutorials 1 to 7 cover the basics and it is recommended all users read through them. After these you can skip to the required tutorials as needed. Tutorial 9 covers the basics of automating simulations using Python and should be used as a starting point if required. For the transport solver Tutorials 8 and 10 should be used as a starting point. Tutorials 17 to 23 cover the spin transport solver. Tutorials 14 to 16 cover the heat solver.

The equations solved are given in the Differential Equations and Modules sections. All material parameters used in these equations have been given in the Material Parameters section. All the important parts of the program have been documented in the sections after the Tutorials, including output data and advanced features.

A full list of commands has been provided in the Commands section in alphabetical order. The most commonly used commands have also been outlined.

Tutorial 0 – Quick-Start

This tutorial contains a set of Python scripts for a selected number of micromagnetics problems, and are intended as a crash-course for experienced users who want to get things going very quickly. No detailed comments are provided, except for comments in the Python scripts – if you want in-depth explanations you need to read the other tutorials. All scripts are found in the Examples/Tutorial 0 folder.

The Python scripts below can be run either locally (connect through NSClient as 'localhost' – this is the default), or remotely (connect using the IP address of machine running Boris). Alternatively Python scripts may be executed in embedded mode, where Boris loads the script and calls the Python interpreter, not requiring communication through network sockets.

For the Python scripts below you need the NetSocks.py module. Before executing the script a Boris instance should be running.

The default program start-up state is a *permalloy* rectangle with dimensions 80 nm × 80 nm × 10 nm, and cubic 5 nm cellsize. The *demag*, *exchange*, and *Zeeman* modules are enabled. The LLG equation is set with the RKF45 evaluation method. To restore the program to default state at any time use the **default** console command (or alternatively send it as ns.default() from a Python script).

The default simulation stage is a *Relax* stage with $|mxh| < 10^{-4}$ stopping condition and no data saving condition.

Hysteresis Loop

```
import matplotlib.pyplot as plt
import numpy as np
from NetSocks import NSClient
ns = NSClient(); ns.configure(True)

########################################

#Make a ferromagnetic mesh with given rectangle, cubic cellsize, and set modules
Py = ns.Ferromagnet([160e-9, 80e-9, 10e-9], [5e-9])
Py.modules(['demag', 'exchange', 'Zeeman'])

#configure output data : applied field and average magnetisation
ns.setsavedata('hysteresis.txt', ['Ha', Py], ['<M>', Py])

########################################

#run two stages to sweep field up and down between -100 kA/m and +100kA/m in 100
steps, slightly off-axis
#each field step relaxed to mxh < 1e-5, and data saved every field step
ns.setode('LLGStatic', 'SDesc')
ns.Hpolar_seq([Py, [-100e3, 90, 1, +100e3, 90, 1, 100], 'mxh', 1e-5, 'step'])
ns.Hpolar_seq([Py, [100e3, 90, 1, -100e3, 90, 1, 100], 'mxh', 1e-5, 'step'])

########################################

#output file has field (x, y, z components) in columns 0, 1, 2, and average
magnetisation (x, y, z components) in columns 3, 4, 5
hysteresis_data = ns.Get_Data_Columns('hysteresis.txt', [0, 3])
#plot Mx vs Hx
plt.axes(xlabel = 'H (kA/m)', ylabel = '<M> (kA/m)')
plt.plot(np.array(hysteresis_data[0])/1e3, np.array(hysteresis_data[1])/1e3)
plt.show()
```

Hysteresis Loop (embedded script version)

```
import matplotlib.pyplot as plt
import numpy as np
from NetSocks import NSClient
ns = NSClient(embedded = True); ns.configure(True)

########################################

#Make a ferromagnetic mesh with given rectangle, cubic cellsize, and set modules
Py = ns.Ferromagnet([160e-9, 80e-9, 10e-9], [5e-9])
Py.modules(['demag', 'exchange', 'Zeeman'])

#configure output data : applied field and average magnetisation
ns.setsavedata('hysteresis.txt', ['Ha', Py], ['<M>', Py])

########################################

#This method is executed every simulation iteration, called by BORIS
def save_coercivity_image(Mx_previous):
    M = Py.showdata('<M>')

    if M[0] * Mx_previous < 0.0:
        if M[0] > 0.0: ns.savemeshimage('Coercive_up.png')
        else: ns.savemeshimage('Coercive_dn.png')

    return 1, M[0]

#run two stages to sweep field up and down between -100 kA/m and +100kA/m in 100
steps, slightly off-axis
#each field step relaxed to mxh < 1e-5, and data saved every field step
ns.setode('LLGStatic', 'SDesc')
ns.Hpolar_seq([Py,  [-100e3,  90,  1,  +100e3,  90,  1,  100],  'mxh',  1e-5,  'step'],
save_coercivity_image, -800e3)
ns.Hpolar_seq([Py,  [100e3,  90,  1,  -100e3,  90,  1,  100],  'mxh',  1e-5,  'step'],
save_coercivity_image, 800e3)

########################################

#output  file  has  field  (x,  y,  z  components)  in  columns  0,  1,  2,  and  average
magnetisation (x, y, z components) in columns 3, 4, 5
hysteresis_data = ns.Get_Data_Columns('hysteresis.txt', [0, 3])
#plot Mx vs Hx
plt.axes(xlabel = 'H (kA/m)', ylabel = '<M> (kA/m)')
plt.plot(np.array(hysteresis_data[0])/1e3, np.array(hysteresis_data[1])/1e3)
plt.show()
```

Domain Wall Movement

```
from NetSocks import NSClient
import matplotlib.pyplot as plt
import numpy as np

ns = NSClient(); ns.configure(True)

########################################

#This is based on Exercise 5.1, done entirely using a Python script

Py = ns.Ferromagnet([320e-9, 80e-9, 20e-9], [5e-9, 5e-9, 5e-9])
Py.modules(['demag', 'exchange', 'Zeeman'])

#setup the moving mesh algorithm for a transverse domain wall along the x axis:
#1. remove end magnetic charges using two dipole meshes (enables strayfield
module)
#2. freeze x-axis ends spins
#3. set a transverse domain wall
#4. enable moving mesh algorithm which keeps the dw centered
ns.preparemovingmesh()

#relax dw in zero field to |mxh| < 10^-5
ns.setode('LLGStatic', 'SDesc')
ns.Relax(['mxh', 1e-5])
ns.reset()

#set fixed time-step RK4 method with 500fs time step
ns.setode('LLG', 'RK4')
ns.setdt(500e-15)

#save time (s) and dw shift (m) data
ns.setsavedata('dwmovement_temp.txt', ['stime'], ['dwshift'])

Hrange = np.arange(100, 2400, 200)
dwvelocity = np.array([])

for H in Hrange:

    #first stage achieves steady state movement
    ns.Hxyz([Py, [H, 0, 0], 'time', 5e-9])
    #second stage captures data
    ns.Hxyz([Py, [H, 0, 0], 'time', 10e-9, 'time', 1e-12])
    ns.reset()

    #process data to extract domain wall velocity
    ns.dp_load('dwmovement_temp.txt', [0, 1, 0, 1])
    ns.dp_replacerepeats(1)
    dwdata = ns.dp_linreg(0, 1)
    dwvelocity = np.append(dwvelocity, dwdata[0])

    print('H (A/m) = %f, DW velocity (m/s) = %0.4f' % (H, dwdata[0]))

plt.axes(xlabel = 'H (A/m)', ylabel = 'DW Velocity (m/s)', title = 'DW Velocity
and Walker Breakdown')
plt.plot(Hrange, dwvelocity, 'o-')
plt.show()
```

Anisotropic Magnetoresistance

```
from NetSocks import NSClient
import matplotlib.pyplot as plt
import numpy as np

ns = NSClient(); ns.configure(True)

########################################
#This is based on Exercise 8.3, done entirely using a Python script

Py = ns.Ferromagnet([160e-9, 80e-9, 10e-9], [5e-9, 5e-9, 5e-9])
Py.modules(['Zeeman', 'demag', 'exchange', 'transport'])

#set amr percentage of 2%
Py.param.amr = 2.0

#amr loop angle (deg.)
direction_deg = 5
ns.setangle(90, 180.0 + direction_deg)

#set electrodes at x-axis ends with a 1 mV potential drop
ns.setdefaultelectrodes()
ns.setpotential(1e-3)

#save applied field (A/m) and resistance (Ohms)
ns.setsavedata('amr_rawdata.txt', ['Ha', Py], ['R'])

#Run hysteresis loop
ns.setode('LLGStatic', 'SDesc')
ns.Hpolar_seq(
    [Py, [-100e3, 90, direction_deg, 100e3, 90, direction_deg, 200],
     'mxh', 1e-6, 'step'])
ns.Hpolar_seq(
    [Py, [100e3, 90, direction_deg, -100e3, 90, direction_deg, 200],
     'mxh', 1e-6, 'step'])

########################################
#Plot loop

#load all columns from file (0, 1, 2, 3) into internal arrays (0, 1, 2, 4)
ns.dp_load('amr_rawdata.txt', [0, 1, 2, 3, 0, 1, 2, 4])
#get field strength along loop direction and save it in internal array 3
ns.dp_dotprod(0, np.cos(np.radians(direction_deg)),
np.sin(np.radians(direction_deg)), 0, 3)
#save field strength and resistance in processed file, then plot it here
ns.dp_save('amr_loop.txt', [3, 4])
amr_data = ns.Get_Data_Columns('amr_loop.txt', [0, 1])
plt.axes(xlabel = 'H (kA/m)', ylabel = 'R (Ohms)', title = 'AMR Loop')
plt.plot(np.array(amr_data[0])/1e3, amr_data[1])
plt.show()
```

RKKY Simulation (multi-mesh demonstration)

```
from NetSocks import NSClient
import matplotlib.pyplot as plt
import numpy as np

ns = NSClient(); ns.configure(True)

########################################

direction_deg = 1.0

FM1 = ns.Ferromagnet([320e-9, 160e-9, 10e-9], [5e-9, 5e-9, 5e-9])
FM1.modules(['Zeeman', 'demag', 'exchange', 'surfexchange'])
#shape mesh as an ellipse (mask file is stretched to mesh aspect ratios)
FM1.loadmaskfile('Circle')

#add a new ferromagnetic mesh (permalloy by default) above the first one with 1 nm
separation
FM2 = ns.Ferromagnet([0.0, 0.0, 11e-9, 320e-9, 160e-9, 31e-9], [5e-9, 5e-9, 5e-9])
FM2.modules(['Zeeman', 'demag', 'exchange', 'surfexchange'])
FM2.loadmaskfile('Circle')

#enable multilayered demag field calculation allowing exact and efficient
calculation of demag fields, even though the separation between meshes is 1 nm
ns.addmodule('supermesh', 'sdemag')

ns.setsavedata('rkky_hysteresis.txt', ['Ha', FM1], ['<M>', FM1], ['<M>', FM2])
ns.setode('LLGStatic', 'SDesc')
ns.cuda(1)
ns.Hpolar_seq(['supermesh', [-300e3, 90, direction_deg, 300e3, 90, direction_deg,
300], 'mxh', 1e-5, 'step'])
ns.Hpolar_seq(['supermesh', [300e3, 90, direction_deg, -300e3, 90, direction_deg,
300], 'mxh', 1e-5, 'step'])

########################################

u = [np.cos(np.radians(direction_deg)), np.sin(np.radians(direction_deg)), 0]
ns.dp_load('rkky_hysteresis', [0, 1, 2, 3, 4, 5, 6, 7, 8, 0, 1, 2, 3, 4, 5, 6, 7,
8])

ns.dp_dotprod(0, u[0], u[1], u[2], 10)
ns.dp_dotprod(3, u[0], u[1], u[2], 11)
ns.dp_dotprod(6, u[0], u[1], u[2], 12)
ns.dp_mul(11, 1.0/3)
ns.dp_mul(12, 2.0/3)
ns.dp_adddp(11, 12, 13)
ns.dp_save('rkky_hysteresis_loop.txt', [10, 13])

loop = ns.Get_Data_Columns('rkky_hysteresis_loop.txt', [0, 1])
plt.axes(xlabel = 'H (kA/m)', ylabel = 'M (kA/m)', title = 'RKKY Hysteresis Loop')
plt.plot(np.array(loop[0])/1e3, np.array(loop[1])/1e3)
plt.show()
```

Setting/Editing Parameter Variations Programatically

Parameter spatial variations can be set using ovf2 files in the most general case. For many cases it suffices to use the Shape class, imported from NetSocks – see chapter on Shapes and Regions. This allows defining regions for setting material parameter values easily.

```
from NetSocks import NSClient

ns = NSClient(); ns.configure(True)

########################################

ns.meshrect([800e-9, 800e-9, 5e-9])
#setup a grain structure, marking the grains with values in uniform random number
distribution between 0 and 1
ns.setparamvar('permalloy', 'J1', 'vor2D', [0.0, 1.0, 20e-9, 1])
#save it to a file and load it here so we can edit it
ns.saveovf2param(paramname = 'J1', filename = 'grains.ovf')
vec, n, rect = ns.Read_OVF2('grains.ovf')

#snap grain values to 0 or 1 : on average half will be 0, the other half 1
for idx in range(len(vec)):
    if vec[idx] < 0.5: vec[idx] = 0.0
    else: vec[idx] = 1.0

#write it to a file, then load the new grain values
ns.Write_OVF2('grains_snapped.ovf', vec, n, rect)
ns.setparamvar('permalloy', 'J1', 'ovf2', 'grains_snapped.ovf')

#setup display so we can see the grain structure
ns.display('ParamVar')
ns.setdisplayedparamsvar('permalloy', 'J1')
ns.displaydetail(5e-9)
```

Exchange Bias

```
from NetSocks import NSClient
import matplotlib.pyplot as plt
import numpy as np

ns = NSClient(); ns.configure(True)

########################################

AFM = ns.AntiFerromagnet([320e-9, 320e-9, 10e-9], [5e-9, 5e-9, 5e-9])
AFM.modules(['Zeeman', 'exchange', 'aniuni', 'surfexchange'])

#set sub-lattice A magnetisation to result in biasing towards +ve side
AFM.setangle(90, 180)

#Add Fe mesh on top of the antiferromagnet
FM = ns.Material('Fe', [0, 0, 10e-9, 320e-9, 320e-9, 12e-9], [2.5e-9, 2.5e-9, 2e-
9])
FM.modules(['Zeeman', 'demag', 'exchange', 'anicubi', 'surfexchange'])
FM.pbc('x', 10)
FM.pbc('y', 10)
#set bilinear surface exchange coupling value - exchange bias is proportional to
this
FM.param.J1 = 0.2e-3

########################################

ns.setsavedata('exchangebias.txt', ['Ha', FM], ['<M>', FM])
ns.setode('LLGStatic', 'SDesc')
ns.cuda(1)
ns.Hpolar_seq(['supermesh', [-50e3, 90, 5, 100e3, 90, 5, 50], 'mxh', 1e-5,
'step'])
ns.Hpolar_seq(['supermesh', [100e3, 90, 5, -50e3, 90, 5, 50], 'mxh', 1e-5,
'step'])

########################################

#we should really project along the 5 degree direction, but will keep this simple
data = ns.Get_Data_Columns('exchangebias.txt', [0, 3])
plt.axes(xlabel = 'H (kA/m)', ylabel = 'M (kA/m)', title = 'Exchange bias')
plt.plot(np.array(data[0])/1e3, np.array(data[1])/1e3)
plt.show()
```

Ultrafast Demagnetization and Skyrmion Creation in a Co/Pt/SiO_2 Trilayer (Advanced)

```
from NetSocks import NSClient
import matplotlib.pyplot as plt
import numpy as np

ns = NSClient(); ns.configure(True)

########################################
#Setup trilayer

#Co layer
FM = ns.Material('Co/Pt', [512e-9, 512e-9, 2e-9], [1e-9, 1e-9, 2e-9])
FM.modules(['Zeeman', 'demag', 'iDMexchange', 'aniuni', 'heat'])
FM.scellsize([4e-9, 4e-9, 2e-9])
FM.tcellsize([2e-9, 2e-9, 1e-9])
FM.tmodel(2)
FM.curietemperature(500)
FM.pbc('x', 10)
FM.pbc('y', 10)
FM.setangle(0, 0)
FM.setfield(100e3, 0, 0)

#Pt layer
HM = ns.Material('Pt', [0.0, 0.0, -8e-9, 512e-9, 512e-9, 0.0], [4e-9, 4e-9, 4e-9])
HM.modules(['heat'])
HM.tmodel(2)

#SiO2 layer
Substrate = ns.Material('SiO2', [0.0, 0.0, -48e-9, 512e-9, 512e-9, -8e-9], [8e-9,
8e-9, 8e-9])
Substrate.modules(['heat'])

ns.temperature(300)

########################################
#Solver

#RK4 with quartic polynomial extrapolation of demag field
ns.setode('sLLB', 'RK4')
ns.evalspeedup(5)
ns.setdtstoch(20e-15)

########################################
#Set-up display
FM.display('M')
FM.vecrep(3)
ns.displaydetail(1e-9)

########################################
#Ouput data and heat source

ns.setsavedata('ufsky.txt', ['time'], ['Q_topo', FM], ['<T>', FM, [255e-9, 255e-9,
0.0, 256e-9, 256e-9, 2e-9]])

#heat source due to laser spot in Co layer
FM.param.Q = 2e21
FM.param.Q.setparamvar('equation', 'exp(-sqrt((x/Lx - 0.5)^2 + (y/Ly - 0.5)^2) /
((d0/Lx)^2/(4*ln(2)))) * exp(-(t-2*tau)^2/(tau^2/(4*ln(2))))')
```

```
ns.equationconstants('d0', 400e-9)
ns.equationconstants('tau', 100e-15)

########################################
#Simulate

ns.cuda(1)

#0 to 10ps
ns.setdt(1e-15)
ns.setheatdt(1e-15)
ns.Relax(['time', 10e-12, 'time', 10e-15])

#10ps to 20ps
ns.setdt(4e-15)
ns.setheatdt(2e-15)
ns.Relax(['time', 20e-12, 'time', 10e-15])

#20ps to 60ps
#mid-simulation re-meshing! 1nm cellsize only needed around the Curie temperature.
FM.cellsize([2e-9, 2e-9, 2e-9])
ns.setdt(20e-15)
ns.setheatdt(2e-15)
ns.Relax(['time', 60e-12, 'time', 10e-15])

#60ps to 800ps
FM.tcellsize([4e-9, 4e-9, 1e-9])
ns.setdt(100e-15)
ns.setheatdt(5e-15)
ns.Relax(['time', 800e-12, 'time', 250e-15])

########################################
#Plotting

ns.savemeshimage('ufsky_final')

#now plot |Q| as a function of time
data = ns.Get_Data_Columns('ufsky.txt', [0, 1])
time_ps = [t/1e-12 for t in data[0]]
Qmod = [np.abs(Qval) for Qval in data[1]]

plt.axes(xlabel = 'Time (ps)', ylabel = '|Q|')
plt.xscale('log')
plt.yscale('log')
plt.plot(time_ps, Qmod)
plt.xlim(0.1)
plt.ylim(1e-3)
plt.savefig('ufsky_plot.png', dpi = 600)
plt.show()
```

Tutorial 1 – Introduction

Basics

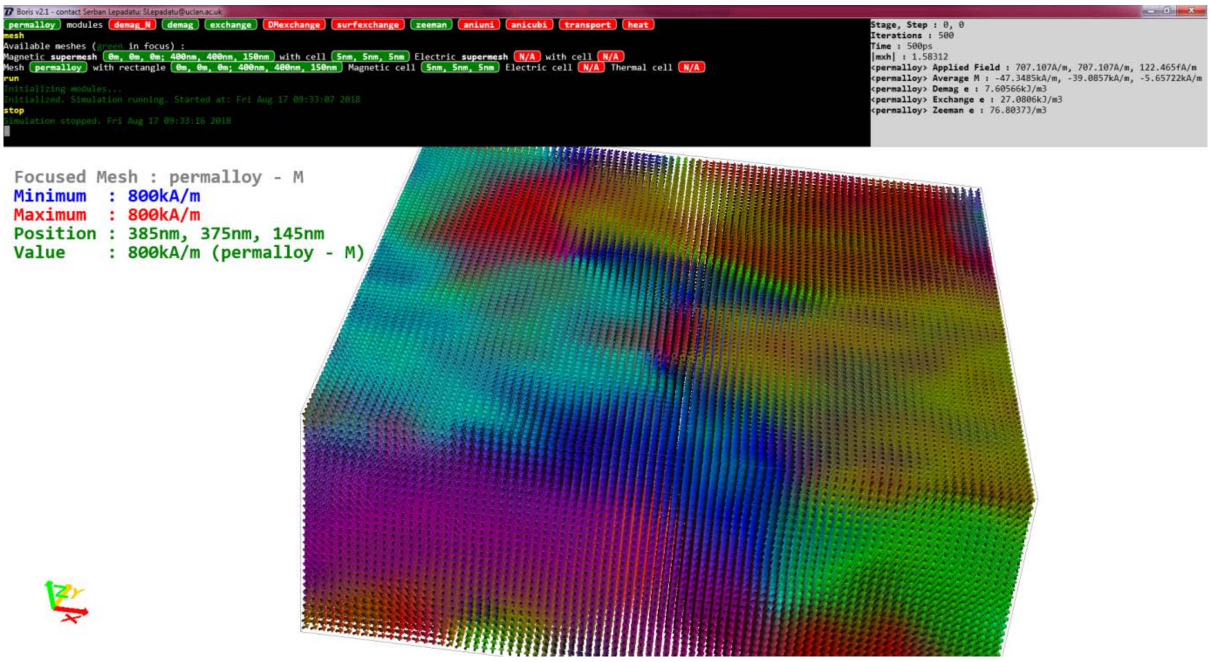
All commands are entered using the console (top-left black box) in Figure 1.1. Each console command has a case-sensitive syntax and may have a number of parameters separated by spaces. If a command is entered with wrong parameters a help prompt will be displayed explaining the command and full syntax. Alternatively a command may be immediately preceded by a question mark in order to display the command help. For example try it for the **run** command:

?run

Note, the program auto-completes commands entered out of the list of possible inputs – and equally stops any wrong inputs from being entered.

The main display shows the magnetization configuration (other vector and scalar quantities may be displayed, but magnetization is the default setting).

Figure 1.1 – Boris interface



```
Boris v2.1 - contact Serban Lepadatu: SLepadatu@uclan.ac.uk
permalloy modules demag_N demag exchange DMexchange surfexchange zeeman aniuni anicubi transport heat
mesh
Available meshes (green in focus) :
Magnetic supermesh 0m, 0m, 0m; 400nm, 400nm, 150nm with cell 5nm, 5nm, 5nm Electric supermesh N/A with cell N/A
Mesh permalloy with rectangle 0m, 0m, 0m; 400nm, 400nm, 150nm Magnetic cell 5nm, 5nm, 5nm Electric cell N/A Thermal cell N/A
run
Initializing modules...
Initialized. Simulation running. Started at: Fri Aug 17 09:33:07 2018
stop
Simulation stopped. Fri Aug 17 09:33:16 2018
```

```
Stage, Step : 0, 0
Iterations : 500
Time : 500ps
|mxh| : 1.58312
<permalloy> Applied Field : 707.107A/m, 707.107A/m, 122.465fA/m
<permalloy> Average M : -47.3485kA/m, -39.0857kA/m, -5.65722kA/m
<permalloy> Demag e : 7.60566kJ/m3
<permalloy> Exchange e : 27.0806kJ/m3
<permalloy> Zeeman e : 76.8037J/m3
```

```
Focused Mesh : permalloy - M
Minimum  : 800kA/m
Maximum  : 800kA/m
Position : 385nm, 375nm, 145nm
Value    : 800kA/m (permalloy - M)
```

The magnetization display may be controlled using the mouse: left-click and drag to re-position the display mesh, middle-click and move mouse to rotate the camera view about the center of the displayed (focused) mesh, right-click and move mouse up/down to zoom in/out, or left/right to rotate the camera about its axis; finally the wheel may be used to set the coarseness of magnetization representation: for large mesh dimensions, each arrow represents an average of the magnetization in that area.
Various simulation data may be displayed in the data box (top-right box) for convenience – more on these later. The displayed windows may be resized by dragging their outlines – the outline appears if you hover the mouse over the edge of a window.

The magnetization display can be reset to the default view using:

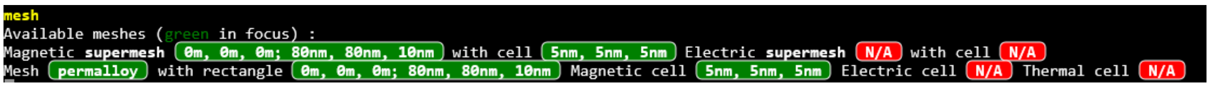
center

Simulation Mesh Control

To display the current problem size enter the following command (see Figure 1.2 for expected output):

mesh

Figure 1.2 – Default mesh configuration



```
mesh
Available meshes (green in focus) :
Magnetic supermesh 0m, 0m, 0m; 80nm, 80nm, 10nm with cell 5nm, 5nm, 5nm Electric supermesh N/A with cell N/A
Mesh permalloy with rectangle 0m, 0m, 0m; 80nm, 80nm, 10nm Magnetic cell 5nm, 5nm, 5nm Electric cell N/A Thermal cell N/A
```

The simulation space consists of one or more named meshes – the default configuration consists of a single ferromagnetic mesh named *permalloy*. This has a rectangle with lower-left corner coordinates of (0, 0, 0) and upper-right corner coordinates of (80 nm, 80 nm, 10 nm). The magnetic discretization cellsize is a rectangular prism with dimensions (d_x, d_y, d_z) = (5 nm, 5 nm, 5 nm) – a cubic cellsize by default. Thus the *permalloy* mesh is discretized with the integer number of cells (16, 16, 2).
To adjust the mesh dimensions you can use the **meshrect** command. An easy way to bring up this command, with current fields already entered, is to double-click on the outlined text containing the mesh rectangle dimensions: `0m, 0m, 0m; 80nm, 80nm, 10nm`

This type of outlined text is a special console text called an interactive console object, allowing a number of user interactions depending on the particular object, including left or right click, double-click, or drag. The text is also automatically updated to display currently set values. You can find out what an interactive object does by using shift-click.

Try to resize the *permalloy* mesh so it has the dimensions (300 nm, 100 nm, 15 nm). Values may be entered without specifying the units, in which case the applicable S.I. unit is assumed, or the applicable unit may be entered together with its magnitude specifier (e.g. for a meter the currently available units are designated as *am, fm, pm, nm, um, mm, m, km, Mm, Gm, Tm, Pm*). If entering the unit, do not leave a space between the number and unit.

The magnetic cellsize may be adjusted by double-clicking on the magnetic cell interactive object, which brings up the **cellsize** command. Try to adjust the cellsize so it has dimensions (6 nm, 6 nm, 5 nm). After changing the cellsize its dimensions are automatically adjusted in order to satisfy the requirement of integer number of discretization cells in each dimension.

Similarly the *permalloy* mesh may be renamed by double-clicking on the mesh name interactive object, which brings up the **renamemesh** command.

In Figure 1.2 you can also see an entry for the *supermesh*. Its rectangle is not controlled directly, but depends on the currently set meshes (however you can adjust the supermesh cellsize). The magnetic supermesh is the smallest simulation space containing all the currently set magnetic meshes and is useful to compute long-range interactions over several independently discretized meshes (e.g. supermesh demagnetizing field) – more on this in a dedicated tutorial.

Basic Simulation Control

The simulation is started and stopped using:

run
stop

The stop command simply pauses the simulation without resetting it. To continue from the stop point simply type **run** again. To reset the simulation use:

reset

The display refresh frequency can be set using (*iter* is the number of iterations – remember you can query to command for full details: **?iterupdate**):

iterupdate *iter*

The simulation may be saved at any point using (do not use a termination, the .bsm termination is added by default):

savesim *filename*

If a directory path is not specified, the default directory path is used. To set a default directory use:

chdir *directory*

To load a previously saved simulation use:

loadsim *filename*

Alternatively a simulation file may be dragged into the console area. At any point you can return to the default program state by using:

default

A uniform magnetization configuration can be set using the following (theta is the polar angle, phi is the azimuthal angle in spherical polar coordinates):

setangle theta phi

A uniform magnetic field can be set using (again use spherical polar coordinates):

setfield Hmag Htheta Hphi

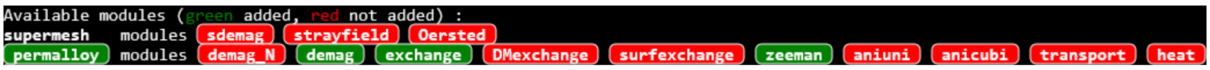
<u>Simulation Modules</u>

Simulation modules typically correspond to effective field terms. These can be managed using interactive objects by typing the following command:

modules

The default configuration includes the demagnetizing field (*demag*), direct exchange interaction (*exchange*), and applied field (*zeeman*) – see Figure 1.3.

Figure 1.3 – Default simulation modules



```
Available modules (green added, red not added) :
supermesh   modules  sdemag   strayfield   Oersted
 permalloy  modules  demag_N   demag   exchange   DMexchange   surfexchange   zeeman   aniuni   anicubi   transport   heat
```

Currently added modules are displayed in green. To add or remove a module left or right-click on the respective interactive object. Individual modules will be explored in future tutorials.

<u>Material Parameters</u>

Default simulation parameters are set for permalloy ($Ni_{80}Fe_{20}$), as M_s = 8e5 A/m, A = 1.3e-11 J/m, and α = 0.02. To see a list of currently set parameter values use the command:

params

Values may be modified by double-clicking on the respective interactive objects. By default parameters are constant for each mesh, however they can be assigned temperature and spatial dependence for advanced simulations (more on this later).

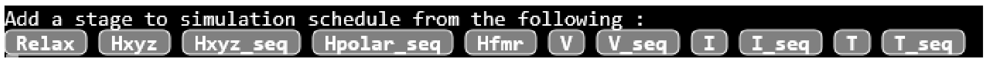
Simulation Flow

A basic simulation flow can be programmed by setting a number of *stages*. Each stage has an identifier, parameters depending on the identifier, a stopping condition and data save condition. To show the currently set simulation stages use:

stages

By default the *Relax* stage type is used (no simulation values changed), with a stopping condition based on the normalized torque |*mxh*| < 10^{-4} (*mxh: 0.0001*), and no data saving configured. New stages may be added by double-clicking on the interactive objects at the bottom – see Figure 1.4.

Figure 1.4 – Simulation stage types

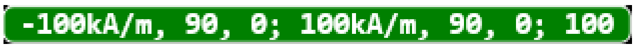

Add a stage to simulation schedule from the following :

You can delete added stages by right-clicking on them, change the stage type by double-clicking and editing, and re-arrange the stage order by dragging.

Available stopping conditions include: *nostop, iter* (stop after a number of iterations), *mxh* (stop when |*mxh*| falls below the set threshold), *dmdt* (stop when the normalized $|\partial \mathbf{m}/\partial t|$ value falls below the set threshold), or *time* (stop after an elapsed simulation time). When a stage reaches the stopping condition the next stage starts or the simulation finishes.

Some stage types are broken down into several sub-stages (referred to as *steps*), for example a field sequence using *Hpolar_seq*. Try to add a field sequence stage by double-clicking on the *Hpolar_seq* interactive object. The default parameters for a field sequence are shown in Figure 1.5. This consists of a field sequence starting from a field of -100 kA/m along the (90, 0) direction, and stopping at +100 kA/m along the (90, 0) direction – this is given by the polar and azimuthal angles in degrees, thus (90, 0) is along the x axis. The sequence consists of 100 field steps, thus the field step is 2 kA/m. The simulation proceeds to the next *step* when the |*mxh*| < 10^{-4} stopping condition is satisfied.

Figure 1.5 – Default parameters for the *Hpolar_seq* stage type



```
-100kA/m, 90, 0; 100kA/m, 90, 0; 100
```

Exercise 1.1

Set a 160×80×5 nm permalloy mesh (with cubic cellsize of 5 nm) starting from a saturated magnetization state along the negative x-axis direction. Set a field sequence from -60 kA/m to +60 kA/m along the x-axis using a step of 2 kA/m and *mxh* stopping condition of 10^{-5}. Note: additionally you should configure a static magnetization problem – more on this in future tutorials. Type **ode** in the console, then select the *LLGStatic* micromagnetic equation by clicking on it, and make sure the *SDesc* evaluation method is enabled.

Display the applied field and average magnetization in the data box. To do this use the command:

data

You will see a list of interactive console objects representing possible output data which can be displayed in the data box or saved to a file. For now just display the applied field (*Ha*) and average magnetization (<*M*>) by right-clicking on the interactive objects (or dragging them to the data box).

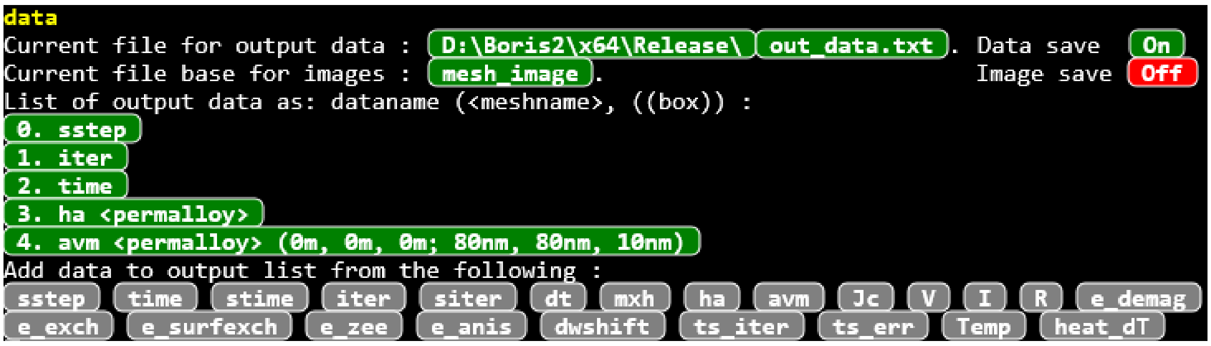
Run the simulation - the magnetization should switch during this field sequence.

Tutorial 2 – Data Output

Saving Numerical Data

In order to save numerical simulation data (automated saving of images will be explored in a future tutorial) you need to set a data saving file, a list of output data, and a saving schedule. To set output data and a save file use the **data** command.

Figure 2.1 – Default output data



```
data
Current file for output data : D:\Boris2\x64\Release\ out_data.txt . Data save  On
Current file base for images : mesh_image .                             Image save Off
List of output data as: dataname (<meshname>, ((box)) :
0. sstep
1. iter
2. time
3. ha <permalloy>
4. avm <permalloy> (0m, 0m, 0m; 80nm, 80nm, 10nm)
Add data to output list from the following :
sstep  time  stime  iter  siter  dt  mxh  ha  avm  Jc  V  I  R  e_demag
e_exch  e_surfexch  e_zee  e_anis  dwshift  ts_iter  ts_err  Temp  heat_dT
```

The default output data file is called *out_data.txt* and its name may be modified by double-clicking on the interactive object. The default saving directory is the path to the program executable file and may be modified by double-clicking on the interactive object.

The default output data includes *sstep* (stage and step), *iter* (iteration), *time* (simulation time), *Ha* (applied field), and <*M*> (average magnetization). This is the order the output data will appear in the output file as numerical columns. The order may be modified by dragging the respective interactive objects in the list of output data. New output data may be added by double-clicking on the interactive objects at the bottom, and set output data may be deleted by right-clicking on the respective interactive objects in the output list.

Some output data (such as *ha* and <*M*>) may be saved in a particular mesh – in this case the *permalloy* mesh which is specified using the notation <permalloy>, whilst other output data do not depend on any particular mesh. Some output data (such as <*M*>) may also be saved in a particular rectangle of the named mesh (the rectangle is relative to the named mesh) – by default the entire mesh rectangle is saved, but this can be modified by double-clicking on the respective interactive object and editing.

Finally, a saving schedule may be set in the simulation stages: use the **stages** command. Each stage has a list of possible saving conditions: *none* (default – do not save), *stage* (save at the end of the stage), *step* (save at the end of each step in the current stage), *iter* (save every n iterations), and *time* (save every t simulation seconds); for *iter* and *time* the parameters may be edited by double-clicking the respective interactive objects.

Exercise 2.1

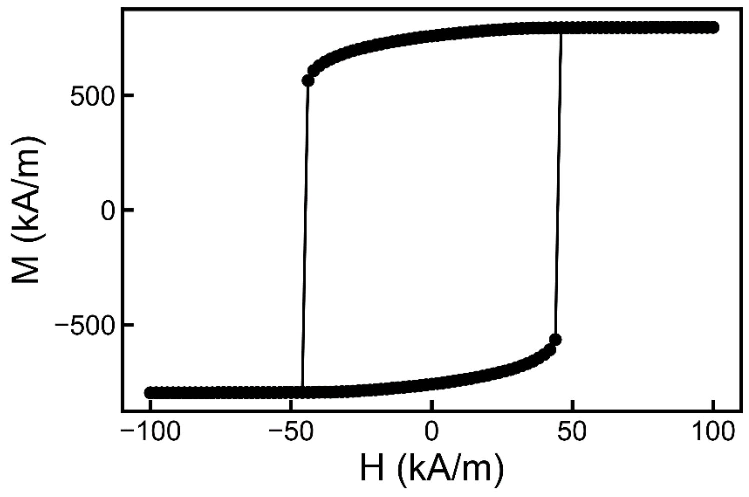
Set a 160×80×10 nm permalloy mesh (with cubic cellsize of 5 nm) starting from a saturated magnetization state along the negative x-axis direction. Set a field sequence from -100 kA/m to +100 kA/m and then back to -100 kA/m in the xy plane, and at 1 degree from the x-axis. Use a step of 2 kA/m and *mxh* stopping condition of 10^{-5}. Note: additionally you should configure a static magnetization problem as in exercise 1.1.

Configure the simulation so it saves output data for a hysteresis loop (applied field and average magnetization saved after every step).

Before running the simulation save the simulation file. Once the simulation file is saved using a specified name (and directory path if needed), the next time you don't need to specify the file name – simply use **savesim** without a file name and the previously used file name will be saved. Note, correct commands previously entered in the console can be recalled using the arrow keys (invalid commands are not saved). You can also use Ctrl^v to paste text in the console.

Run the simulation and plot the hysteresis loop (magnetization along the applied field direction) at the end.

Figure 2.2 – Hysteresis loop obtained in exercise 2.1



Further Data Box Control

As introduced in the previous tutorial, output data may also be displayed in the data box for convenience. The possible output data may be listed as interactive objects by using the **data** command, and the listed interactive objects may be displayed in the data box by dragging them there, or right-clicking on them. Data box entries may be removed by right-clicking on them in the data box, and they may be re-arranged by dragging them.

If you just want to quickly see the current values of particular data without displaying them in the data box, bring up an interactive object list using the **showdata** command and double-click on the respective interactive objects.

Exercise 2.2

In this exercise you will run the μMAG standard problem #4: https://www.ctcms.nist.gov/~rdm/std4/spec4.html

a) For this problem we need a permalloy mesh with dimensions 500x125x3 nm. First initialize the magnetization configuration to a so-called s-state: this may be obtained by reducing a large applied field to zero along the [1,1,1] direction. For example set a field sequence starting from 1 MA/m along the [1,1,1] direction, reducing to zero in

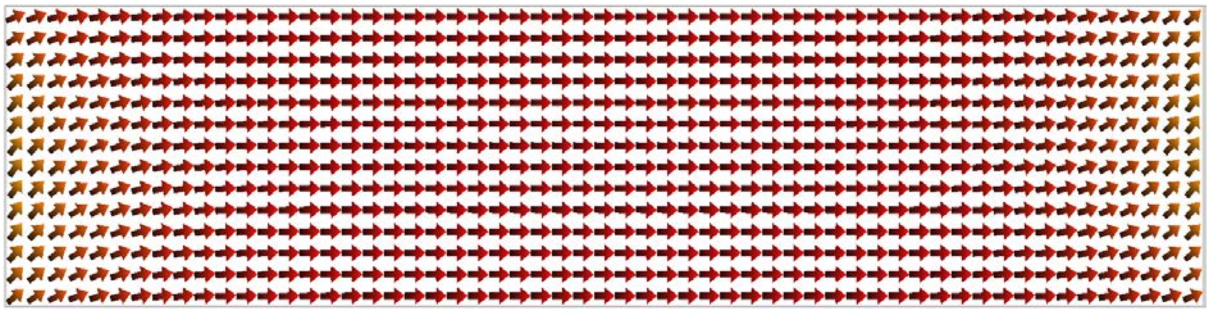
20 steps – you should use the stricter |mxh| < 10^{-5} condition this time. Save an image of the obtained magnetization configuration – see Figure 2.3.

The easiest way to setup Exercise 2.2a is to use a polar field sequence: *Hpolar_seq*. This specifies the starting and ending field values using polar coordinates: magnitude, polar angle and azimuthal angle – thus a starting field of 1 MA/m along the [1,1,1] direction would be specified (roughly) as *1MA/m, 55, 45*. Make sure to specify the ending field value as *0, 55, 45* to keep the field values in the sequence along the same direction. You should also set the starting magnetization state along the [1,1,1] direction: **setangle** *55 45*.

To save an image of the currently displayed mesh, use the command:

savemeshimage *(directory\)filename*

Figure 2.3 – Starting s-state for micromagnetics standard problem #4



Exercise 2.2 continued

b) Starting from the s-state, apply a fixed field with magnetic flux density (B-field) of 25 mT directed 170° counterclockwise from the x-axis in the x-y plane. Simulate the switching event for a duration of 5 ns, saving output data (in particular the average magnetization and simulation time are required) every 5 ps. Plot the 3 components of magnetization against time. Remember to save the simulation before starting it. How do these results compare with published solutions ?

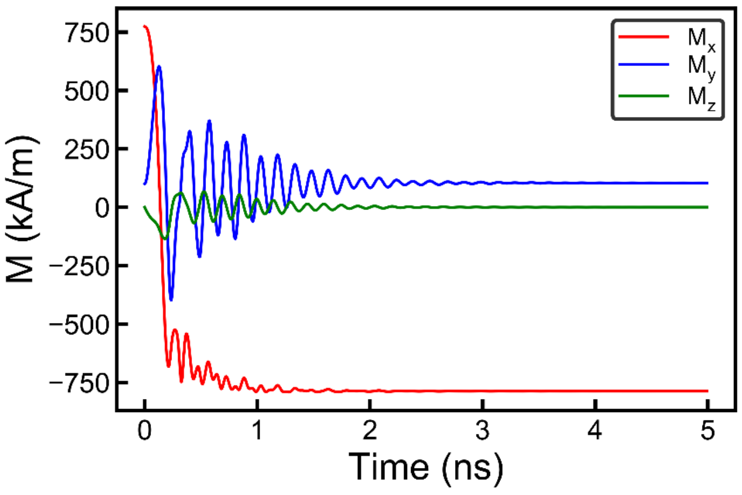
   (see https://www.ctcms.nist.gov/~rdm/std4/results.html)

c) Repeat part b) but this time for a B-field of 36 mT directed 190 degress counterclockwise from the x-axis in the x-y plane.

d) For parts b) and c) obtain images of the magnetization configuration when the average magnetization (x component) first crosses zero – use the output data to determine the time when this occurs, then run the simulation to stop at this particular time.

Figure 2.4 – a) Results obtained after running the μMAG standard problem #4 with field 1, and b) magnetization configuration when the average magnetization (x component) first crosses zero.

a)



b)

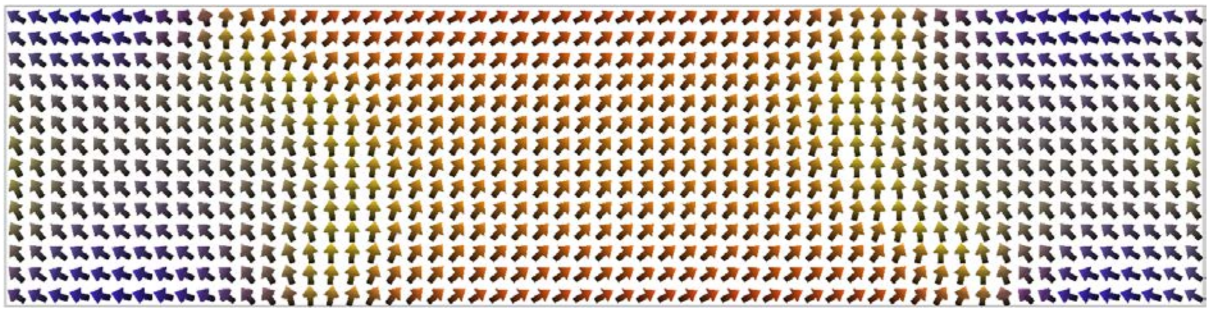
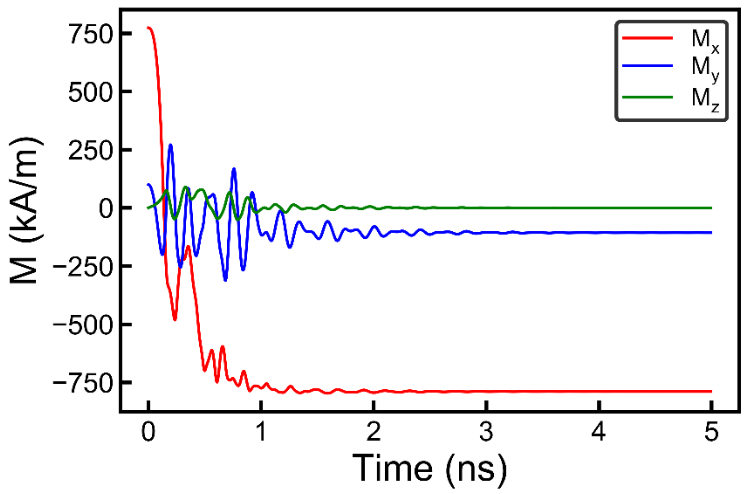
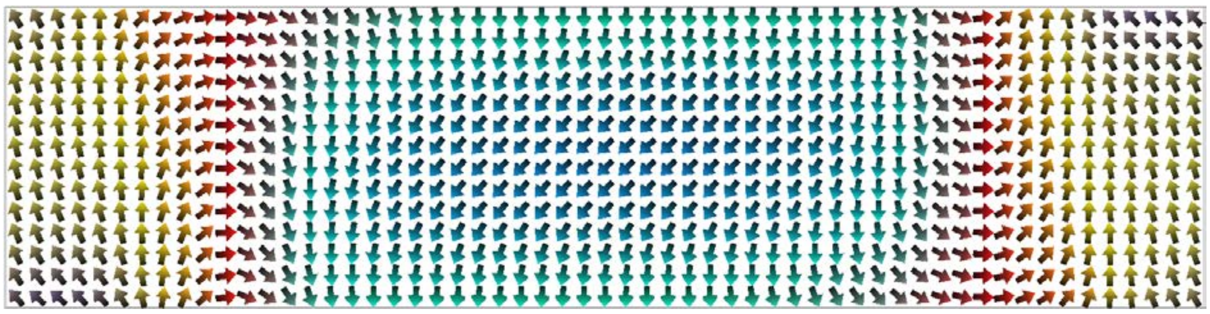


Figure 2.5 – a) Results obtained after running the μMAG standard problem #4 with field 2, and b) magnetization configuration when the average magnetization (x component) first crosses zero.

a)



b)



Making a video from an image sequence

A video may be encoded from a sequence of .png files (e.g. as produced from a simulation with an image saving schedule) – this functionality is built into the program for convenience; for advanced image processing you should use an external program. To produce a video file from an image sequence, use:

makevideo (directory\)*filebase fps quality*

This makes a video from all .png files which start with the *filebase* name, including the directory, at the given *fps* (frames per second). The *quality* parameter sets the bit-rate of the output video: 0 for worst quality but smallest size, 5 for best quality but largest size. For the **makevideo** command, the files are sorted by their creation time, not alphabetically.

To enable mesh image saving, use the **data** command then click on the respective interactive object. You can also edit the mesh image filebase name. The mesh images are saved using the same save conditions as output data.

Exercise 2.3

For the switching event in exercise 2.2b, set the problem to save mesh images every 10 ps; also reduce the simulation time to 3 ns and disable data saving. Make a video of the switching event (40 fps and quality level 3 works well).

If you want to capture only a part of the mesh display window you can set cropping factors. These are specified as normalized values and are applied whenever an image is saved (either manually or during a simulation). This is set using:

imagecropping *left bottom right top*

The left-bottom of the mesh display is (0, 0), whilst the right-top of the mesh display is (1,1).

Tutorial 3 – Further Data Output

Exercise 3.1

In this exercise you will compare coercive fields obtained using the full micromagnetics model with the predictions of the simpler Stoner-Wohlfarth model.

a) Simulate an in-plane hysteresis loop along a 10° direction in a permalloy rectangle with dimensions 250x50x5 nm using the full micromagnetics model (*demag*, *exch*, and *zeeman* modules for permalloy) and obtain the coercive field. In order to speed up the simulation you only need to simulate one branch of the hysteresis loop, e.g. negative to positive field only, and you should also set a coarse field step up to zero field (e.g. 10 kA/m), then a fine field step in order to obtain a more accurate switching field value (use a 500 A/m fine field step or less); use the *mxh* stopping condition with a 10^{-5} threshold. As before you should also configure a static micromagnetics problem (*LLGStatic* equation with *SDesc* evaluation – type **ode** command in console).

   Solution: use two polar field sequences (Hpolar_seq) along the (polar, azimuthal) = (90°, 10°) direction. Start at -200 kA/m and finish at 60 kA/m. This range is just enough to start from a saturated state and capture the switching field.

   i.e. : 1) Hpolar_seq -200kA/m 90 10 0kA/m 90 10 20, 2) Hpolar_seq 0kA/m 90 10 60kA/m 90 10 120

b) Compute the anisotropy energy density (shape anisotropy) and hence obtain the switching field predicted by the Stoner-Wohlfarth model.

Note, the Stoner-Wohlfarth model predicts the switching field:

$$H_S = \frac{2K_u}{\mu_0 M_S}\frac{\sqrt{1-t^2+t^4}}{1+t^2}, \text{ where } t = \sqrt[3]{\tan(\theta)}.$$

Here θ is the angle between the applied field and anisotropy easy axis (formula applicable for θ between 0 and 45°).

To compute the energy density for a given magnetization orientation, set the magnetization orientation (**setangle**) and calculate the energy density terms using the command:

computefields

This command runs the simulation for a single iteration and does not advance the simulation time – only the currently set simulation modules are refreshed. After running this command the required energy density term (*e_demag*) will be available.

c) For the same geometry and applied field direction obtain the hysteresis loop using the Stoner-Wohlfarth model. Does the coercive field agree with that obtained in part b) ?

   To run this you will need to use the *demag_N* module instead of the full *demag* module; the exchange module (*exch*) is not needed.

   The *demag_N* module computes the demagnetizing field using the simple approximation $H_{d,\,i} = -N_i M_i$ ($i = x, y, z$). You will need to enter correct values for the demagnetizing factors N_x and N_y (remembering that $N_x + N_y + N_z = 1$). These can be entered using the command **params**, then editing the values under the N_{xy} interactive object.

   Calculate N_x, N_y and N_z directly from the demagnetizing field (obtained using the full micromagnetics model) and also using the demagnetizing energy density values obtained in part b). Do the values agree, and does the relationship $N_x + N_y + N_z = 1$ hold?

To obtain the demagnetizing field you will need to update the field using the **computefields** command with only the *demag* module enabled. After this you can display the demagnetizing field using the command:

display

Using this command brings up a list of interactive objects with display options. Click on the *Heff* option under the *permalloy* mesh. This will display the computed effective field. Using the average effective field value you can obtain a value for the demagnetizing factor along the set magnetization direction using the expression H_d = -N M.

In order to obtain the average value of the demagnetizing field you can use the command:

averagemeshrect (*sx sy sz ex ey ez*)

This command returns the average value for the displayed quantity in the currently focused mesh (the *permalloy* mesh in this case) when used without parameters. The parameters specify a mesh rectangle (start and end Cartesian coordinates) which is relative to the currently focused mesh.

d) Repeat this exercise using the 100x25x5 nm.
e) Repeat this exercise using the 50x25x3 nm.

Another way to calculate the demagnetizing factors is to use the formula:

$$\varepsilon_d = -\frac{\mu_0}{2}\mathbf{M}.\mathbf{H}_d$$

Thus for **M** along *i* = x, y, z, you can obtain *e_demag*, then use:

$$\varepsilon_{d,i} = \frac{\mu_0}{2} N_i M_S^2 \quad (i = x, y, z)$$

Exercise 3.2

Here you will obtain the magnetization dynamics during a switching event and investigate the effect of the cellsize value on the simulation.

a) Set a 320x160x10 nm permalloy rectangle with magnetization along the length of the rectangle (set the magnetization towards the left, thus blue coloured). Obtain the stable magnetization configuration at zero field by reducing the magnetic field from a large saturation value along x to zero in a number of steps. (e.g. from -50 kA/m to 0). For now use a cellsize of 5 nm.

b) Starting from the magnetization configuration set-up in part a) set a single stage where you apply a large field along the x direction, opposing the magnetization – use 50 kA/m. As stopping condition using a time interval of 4 ns. Set a saving schedule to save the simulation time and average magnetization components every 10 ps. For now use a cellsize of 5 nm. From the saved data plot $<M_x>$ and $<M_y>$ versus time.

c) Repeat the simulation in b) with cellsize values of 10 nm and 2.5 nm. Plot $<M_x>$ and $<M_y>$ versus stage time for the 3 cellsize values. How do the results compare ? Which cellsize would you recommend to use ?

Tutorial 4 – Domain Walls

Generating Domain Walls

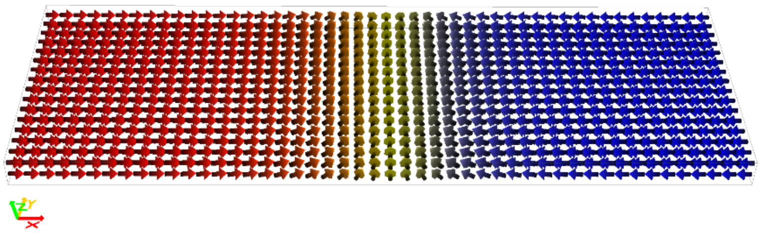
An idealized domain wall (using a tanh profile) along the x direction can be generated using:

dwall *longitudinal transverse width position*

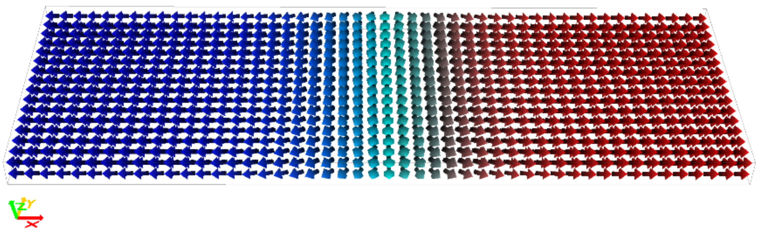
For an in-plane domain wall the longitudinal parameter determines if the wall is head-to-head (*longitudinal* = *x*) or tail-to-tail (*longitudinal* = -*x*); Bloch walls may be generated using *z* or –*z* for the longitudinal component. The transverse parameter determines the rotation direction through the wall – see Figure 4.1 for examples. The w*idth* value is the total domain wall width and *position* is the starting left-hand-side coordinate of the wall (along the x axis), relative to the focused mesh rectangle.

Figure 4.1 – Domain walls generated using the **dwall** command for a mesh with 240nm length, and:

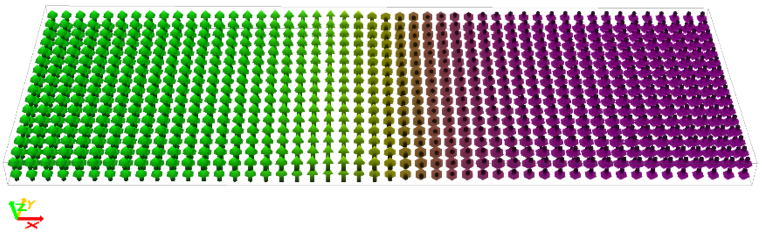
a) in-plane head-to-head transverse domain wall as **dwall** x y 240nm 0



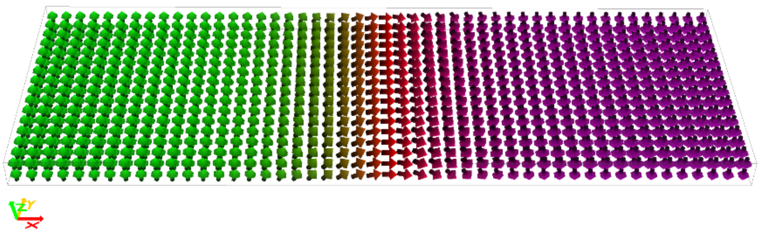
b) in-plane tail-to-tail transverse domain wall as **dwall** -x -y 240nm 0



c) Bloch domain wall as **dwall** z y 240nm 0



d) Néel domain wall as **dwall** z x 240nm 0



Exercise 4.1

Set a mesh for a permalloy rectangle of dimensions 640x80x5 nm with 5 nm cellsize. Generate a head-to-head domain wall over this wire. Run the simulation without a stopping condition and observe how the domain wall is relaxed. You will observe the domain wall does not remain in the center, but eventually drifts towards one side until it is expelled at one of the edges. Can you explain why this happens?

Setting Stray Fields

For domain wall mobility calculations and domain wall configuration relaxation problems in very long wires, it is possible to extend the wires outside of the ferromagnetic mesh by using external uniformly magnetized magnetic bodies, and to calculate the stray field inside the mesh, thereby allowing a smaller mesh size – this eliminates the domain wall drift problem noted in Exercise 4.1. This is done by adding dipole meshes at the left and right-hand-side of the *permalloy* mesh with magnetization direction set as a continuation of the magnetization inside the *permalloy* mesh, and enabling the *strayfield* module.

To add a dipole mesh use the following:

adddipole *name rectangle*

A dipole mesh has a uniform magnetization orientation which is not evolved by the ODE solver but may be handled in a similar manner to ferromagnetic meshes (such as the *permalloy* mesh). Thus modules may be added for computation (**modules**), mesh parameters edited (**params**), quantities displayed (**display**), etc.

You should also exchange couple the ends of the wires to the dipole meshes, thus completing the approximation of a long wire with a domain wall in the center. To enable this use the command:

coupletodipoles

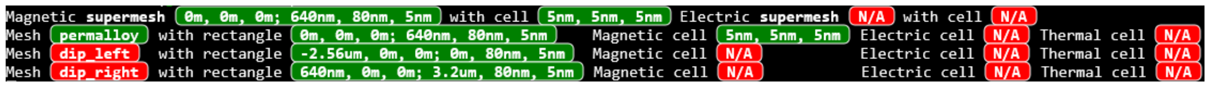
Click on the interactive object to enable exchange coupling to the On state – with this flag turned on all magnetic cells in a ferromagnetic meshes, at the interface with a dipole mesh, are exchange coupled to the fixed dipole magnetization direction.

Exercise 4.2

a) Set two dipole meshes to the left and right of the *permalloy* mesh from the previous exercise, with lengths of 2.56 μm (but same width and thickness). These dipole meshes should now be visible when using the **mesh** command – see Figure 4.2. Set their magnetization orientation in order to extend the head-to-head domain wall configuration from Exercise 4.1 (use **setangle** remembering to specify the mesh name – see **?setangle** for details) – note, the dipole meshes do not display anything by default, you will need to use the **display** command and click the *M* interactive object in order to see their magnetization orientation.

   Run the simulation to relax the domain wall configuration to |mxh| < 10^{-5}. What does the stray field from the dipole meshes look like? (use the **display** command)

Figure 4.2 – Configuration of dipole meshes for exercise 4.2


```
Magnetic supermesh 0m, 0m, 0m; 640nm, 80nm, 5nm with cell 5nm, 5nm, 5nm Electric supermesh N/A with cell N/A
Mesh permalloy with rectangle 0m, 0m, 0m; 640nm, 80nm, 5nm    Magnetic cell 5nm, 5nm, 5nm Electric cell N/A Thermal cell N/A
Mesh dip_left  with rectangle -2.56um, 0m, 0m; 0m, 80nm, 5nm   Magnetic cell N/A           Electric cell N/A Thermal cell N/A
Mesh dip_right with rectangle 640nm, 0m, 0m; 3.2um, 80nm, 5nm  Magnetic cell N/A           Electric cell N/A Thermal cell N/A
```

b) Change the *permalloy* mesh from part a) to a new size of 640x160x30 nm, making sure the dipole meshes are also scaled accordingly (to 160nm width and 30 nm thickness). For the purposes of this exercise you may use a 2D approximation by setting a cellsize of 5x5x30 nm.

   Run the simulation to relax the domain wall configuration to $|mxh| < 10^{-5}$. What type of domain wall results?

When changing mesh dimensions with multiple meshes added to the simulation, there are two options available: i) change just the mesh rectangle required, ii) change the mesh rectangle required and resize/translate all other meshes in proportion. To change the behaviour of the program use the following command and click the interactive object to the On state:

scalemeshrects

With multiple meshes, clicking on the mesh name interactive object (e.g. as displayed using the **mesh** command) changes the display focus to that mesh and resets camera orientation – try it. To quickly focus on a mesh without changing the camera orientation you can double-click on a mesh in the display window.

Exercise 4.3

In this exercise you will calculate the domain wall width for a symmetric transverse domain wall as a function of wire width and compare it to the values obtained using the domain wall formula (A is the exchange stiffness – see the **params** command – and K_u is the anisotropy energy density).

$$\Delta_{dw} = \pi \sqrt{\frac{A}{K_u}}$$

a) Relax domain walls as a function of wire width for a permalloy mesh with dimensions 640 x Width x 5nm, where Width ranges from 40 nm up to 160 nm (simulate at least 4 different values of width). Obtain the domain wall width defined as the half-M_s width value – i.e. the distance it takes for the longitudinal component to change from $+M_S/2$ to $–M_S/2$ for a head-to-head domain wall – and compare it to the value obtained using the formula above (when calculating K_u remember the longitudinal demagnetizing energy is assumed to be negligible as for a very long wire)

To obtain the domain wall profile you can use the following command:

dp_getprofile *start end dp_index*

The above command saves numerical data from the currently displayed mesh quantities (magnetization in this case) along a line starting from the *start* up to the *end* Cartesian coordinates (absolute position values, i.e. not relative to any mesh). The data is saved in internal data processing arrays – more on these in a separate tutorial. For now just obtain a magnetization profile through the middle of the wire as (e.g. for the 80nm wire width): **dp_getprofile** *0 40nm 0 640nm 40nm 0 0.*

The magnetization components are saved in the data processing arrays with indexes starting at 1 (so 1, 2, and 3), whilst data processing array 0 contains the position value. These can be saved to a file as numerical columns using the **dp_save** command as **dp_save** *(directory\)filename.txt 0 1 2 3*. The file will contain the 4 columns as position along the profile (so x coordinate), Mx, My, Mz.

b) Repeat the exercise for a thickness of 10nm using both a 3D simulation (cubic cellsize of 5x5x5 nm) and a 2D approximation (cellsize of 5x5x10 nm). How do the width values compare to those predicted by the formula and are the results obtained using the 2D and 3D model similar?

Tutorial 5 – Domain Wall Movement and Data Processing

In this tutorial you will learn how to obtain a domain wall field-driven mobility curve.
In order to simulate domain wall movement, in addition to setting up a domain wall and dipole meshes as in the previous exercise, the moving mesh algorithm must be enabled by setting a "triggering mesh" as:

movingmesh *mesh_name*

When moving mesh is enabled the magnetization is shifted either to the left or to the right by one notch at a time in order to keep the average x component of magnetization in the triggering mesh, <Mx>, within set boundaries. This keeps the domain wall roughly in the centre of the mesh. When the mesh is shifted to the left or to the right, the data parameter *dwshift* is changed. This data parameter is available for saving to file – see list of data parameters using the **data** command, as discussed previously. Note, this can also be displayed in the console using **showdata** *dwshift*, or displayed in the data box. By saving the simulation time and domain wall shift, the domain wall velocity can be calculated using linear regression.

Since this type of computation is common, there is a shortcut command which sets-up everything required (adding dipole meshes with exchange coupling to the ferromagnetic mesh, enabling stray field computation, setting a domain wall and a triggering mesh):

preparemovingmesh (meshname)

Exercise 5.1

a) Set a 320 × 80 × 20 nm permalloy rectangle with 5x5x20 nm cellsize (2D problem). Enable the moving mesh algorithm and let the domain wall relax in zero field.

b) Set a simulation stage with a field sequence starting from 500 A/m to 2000 A/m in 500 A/m steps, keeping each field step for exactly 2 ns. Set a data save file, and make sure you save the stage time and dwshift parameters every 50 ps.

c) For each field step extract the gradient of the dwshift vs time and plot the wall velocity as a function of field. How does the velocity compare with that predicted by the formula below? (K_u is the in-plane anisotropy energy density)

$$v_{dw} = H \frac{\gamma}{\alpha}\sqrt{\frac{A}{K_u}}, \ \ where\, \gamma = \mu_0 |\gamma_e| \cong 221276 \ (m/As)$$

<u>Console Data Processing</u>

There are a number of built-in commands which allow for a number of operations to be performed on data processing arrays.

First of all, it is possible to load tab-spaced data from a file (such as the data files produced by a Boris simulation) into the internal data processing arrays. This is done using the following command:

dp_load *filename filecol1 … dp_arr1 …*

The above command loads entire columns from the specified file. Thus if the file has a number of tab-spaced data columns, we can load the column with number *filecol1* from the file into the internal data processing array with number dp_*arr1* (these indexes are numbered from 0 up). Multiple columns can be loaded in one command.

Some common data processing commands are listed below.

To multiply a data processing array by a constant value use the following command:

dp_mul *dp_source value dp_dest*

The above command multiplies the data processing array with index *dp_source* by the specified value and stores the result in the *dp_dest* data processing array (this can be the same as *dp_source*). This can be used to normalize data (e.g. a hysteresis loop).

We can use the built-in linear regression command to extract the velocity values:

dp_linreg *dp_x dp_y (dp_z dp_out)*

The above command performs linear regression on the data stored in *dp_x* and *dp_y* data processing arrays and outputs the extracted gradient values and intercepts together with their uncertainties. If *dp_z* is specified multiple linear regressions are performed by using the values in the *dp_z* array to identify adjacent points to be included in a single linear regression; e.g. *dp_z* would contain the applied field values. In this case the outputs are placed in 5 data processing arrays starting at *dp_out* as follows: 1) unique *dp_z* values, 2) gradient, 3) gradient error, 4) intercept, 5) intercept error.

We can also save our processed data, e.g. the domain wall velocity curve, using:

dp_save *filename dp_arr1 …*

Exercise 5.2

Use the console data processing commands to process the output data from Exercise 5.1 and save a domain wall velocity curve.

Other notable commands include:

dp_coercivity *dp_x dp_y*
dp_remanence *dp_x dp_y*

These commands can be used on data from a simulated hysteresis loop in order to extract coercivity and remanence values.

The data processing arrays may be cleared using:

dp_clear *dp_arr1 …*

This clears data in the specified data processing arrays. If no parameters are included all data processing arrays are cleared.

Exercise 5.3

Continuing Exercise 1, find the Walker breakdown threshold with a resolution of 100 A/m starting at 100 A/m. Compare the velocity values with that predicted by the formula in Exercise 5.1c for the steady domain wall movement regime. What is the Walker breakdown threshold?

Exercise 5.4

Calculate the field-driven domain wall mobility curve for permalloy, with a resolution of 100 A/m, as a function of Gilbert damping, for values of damping 0.005, 0.01 and 0.015. How does the Walker breakdown threshold compare with the value predicted by the formula below? ($K_{u,op}$ is the out-of-plane anisotropy energy density)

$$H_W = \frac{\alpha}{2}\frac{K_{u,op}}{\mu_0 M_S} \quad (A/m)$$

Tutorial 6 – ODE Control and Setting Shapes

Setting an ODE solver

The differential equation to solve and its evaluation method is configured using the following command:

ode

The default equation is the Landau-Lifshitz-Gilbert (LLG) equation which you have been using so far. Other equations may be set, e.g. LLB for temperature-dependent simulations, which will be covered in other tutorials.

There are a number of evaluation methods which you can select. The fixed-step methods available are: *Euler* (1st order), trapezoidal Euler (*TEuler* – 2nd order) and Runge-Kuta (*RK4* - 4th order). The adaptive time-step methods are the adaptive Heun (*AHeun* – 2nd order), the multi-step Adams-Bashforth-Moulton (*ABM* – 2nd order), Runge-Kutta-Bogacki-Shampine (*RK23* – 3rd order with embedded 2nd order error estimator), Runge-Kutta-Fehlberg (*RKF45* – 4th order with embedded 5th order error estimator), Runge-Kutta-Fehlberg (*RKF56* – 5th order with embedded 6th order error estimator), Runge-Kutta-Cash-Karp (*RKCK45* – 4th order with embedded 5th order error estimator), and Runge-Kutta-Dormand-Prince (*RKDP54* – 5th order with embedded 4th order error estimator).

There is also a Steepest Descent solver (*SDesc*) used for static problems where we don't need the magnetization dynamics, but are merely interested in calculating the ground state (e.g. relaxing a magnetization configuration and hysteresis loops). The steepest descent solver is only enabled for the *LLGStatic* equation, which is the LLG equation with the precession term disabled and damping value set to 1. In this case the *SDesc* solver is typically at least an order of magnitude faster in computing the ground state compared to the other methods.

For most methods the calculated *mxh* and *dmdt* stopping condition values are the maximum |**m**x**h**| value (normalized torque) in any given iteration. The exceptions are the *Euler*, *TEuler*, and *AHeun* methods, which are only ever used in practice for stochastic equations when including a thermal field (more on this later); additionally *RK4* may also be used for stochastic equations. For this reason the |**m**x**h**| values for these are the mesh averages in any given iteration. This allows using these methods with an *mxh* stopping condition for stochastic equations.

As an exercise we will briefly investigate here the stability of the fixed-step methods as the time step is changed. The time step may be set using:

setdt *dt*

Here *dt* is the time value in seconds. For the adaptive time step methods this command sets the starting time step.

Exercise 6.1

a) Set a 320 × 160 x 20 nm permalloy mesh with a 5 nm cubic cell and relax the magnetization to |mxh| < 10^{-5}.

b) Set a stage with a magnetic field with components (40 kA/m, 5 kA/m, 0) for 5 ns and save data every 10 ps. Use the *RKF45* method. Record the actual computation time required to complete the simulations. Plot the magnetization switching dynamics.

c) Repeat the simulation using the *RK4* method for fixed time steps of 0.5 ps, 0.7 ps, 0.9 ps and 1.1 ps. Record the actual computation time required to complete the simulations. Compare the results with the reference results from the *RKF45* method. How do the results change and why?

d) Compare the computation times. Which method is more efficient whilst still maintaining accuracy?

e) Investigate the computation time required to complete the same problem with *TEuler* with a time step of 50fs and *Euler* with a time step of 30as.

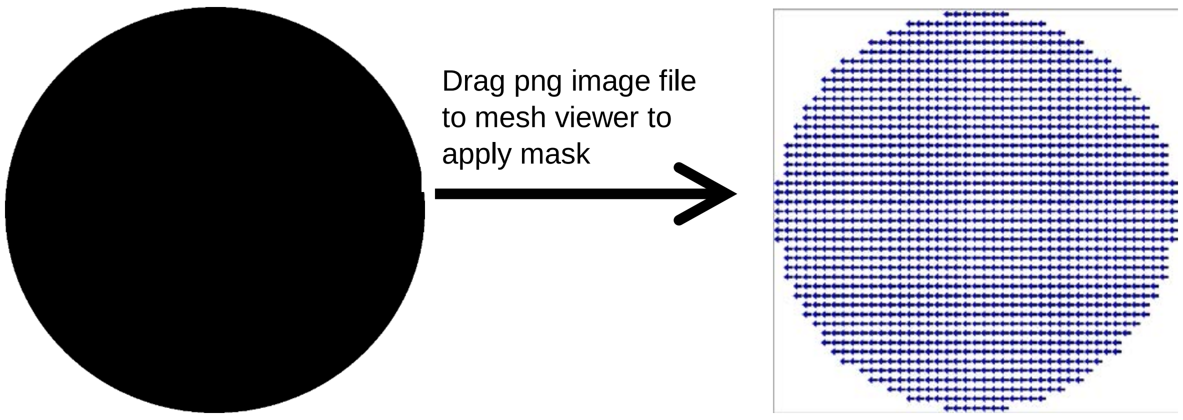
Setting shapes

Until now we've mostly considered meshes which are filled with magnetic cells. In general, complex shapes may be generated by masking the mesh using a shape in an image file. A more advanced method of setting shapes using numerical ovf2 files is covered in a later tutorial. This is achieved using the following command:

loadmaskfile *(zDepth) (directory\)maskfile*

In the simplest case the image file defines a 2D shape in black and the void cells in white – see Figure 6.1 for an example. Instead of using the command you may also drag and drop the image in the mesh viewer window. The *zDepth* value defines the depth the mesh is voided to from top down (if *zDepth* > 0), or the height the mesh is voided to from bottom up (if *zDepth* < 0); this may be used to define 3D shapes with the maskfile being a grayscale image.

Figure 6.1 – Setting a mesh shape using a mask from a png file



Exercise 6.2

a) Load an ellipse into a 320×160×10 nm mesh. Try not to leave void cells at the sides. You should use a circle mask as this will be stretched over the defined mesh rectangle.
b) Obtain hysteresis loops between -50 kA/m and 50 kA/m for the ellipse along the x axis and along the y axis separately using a cellsize of 5x5x10 nm (2D simulations).

You may use the relaxation condition |mxh| < 10^{-4} in order to speed up the exercise. Use the *RKF45* evaluation method. How do the two hysteresis loops compare ?

c) Obtain further hysteresis loops for ellipses with dimensions 260x160x10 nm and 200x160x10 nm. Compare results for all the simulated ellipses and explain the changes in the hysteresis loops.

There is a modifier for how shapes are applied to a mesh, accessed using the **individualshape** command. By default this flag is Off, and any shape applied to the mesh is set for all relevant computational quantities. Thus **M** (magnetization) is a computational quantity which may be shaped, but also *T* (temperature), and σ (electrical conductivity) may be shaped for the relevant solvers (micromagnetics, heat equation, and transport solver respectively). Note that in order to shape *T* and σ you need the relevant solvers enabled. If instead you only want to apply the shape to one of these quantities, or even have different shapes for all of them, you need to enabled the **individualshape** flag, then apply the mask. This is useful for example if you want to include non-magnetic components (e.g. contact leads) in the magnetic mesh.

You may reset the mesh back to its solid shape using:

resetmesh

This command is also useful to recover the mesh following a wrongly-posed computation – e.g. if too large a time-step is used the magnetization values will become *NaN* (not a number) and must be reset. Other methods to shape a mesh include setting and deleting rectangles using:

delrect *rectangle (meshname)*
addrect *rectangle (meshname)*

Tutorial 7 – Magnetocrystalline Anisotropy

In this tutorial you will learn how to use the anisotropy module and simulate hysteresis loops for different magneto-crystalline anisotropy configurations. You need to be familiar with all the basic tutorials.

There are two options available for adding magneto-crystalline anisotropy to the computations: uniaxial or cubic. These are enabled by choosing the *aniuni* or *anicubi* modules from the list displayed using the **modules** command. The modules are mutually exclusive, thus enabling one will delete the other one from the list of active modules.

The strength of the anisotropy is controlled using the *K1* and *K2* parameters (K1 and K2 are the anisotropy energy density constants) from the list displayed using the **params** command. These are the constants that appear in the anisotropy energy formulas:

Uniaxial anisotropy:

$$\varepsilon = K_1\left[1-(\mathbf{m}.\mathbf{e}_a)^2\right] + K_2\left[1-(\mathbf{m}.\mathbf{e}_a)^2\right]^2$$

Cubic anisotropy:

$$\varepsilon = K_1\left[\alpha^2\beta^2 + \alpha^2\gamma^2 + \beta^2\gamma^2\right] + K_2\left[\alpha^2\beta^2\gamma^2\right], \text{ where}$$
$$\alpha = \mathbf{m}.\mathbf{e}_1,\ \beta = \mathbf{m}.\mathbf{e}_2,\ \gamma = \mathbf{m}.(\mathbf{e}_1 \times \mathbf{e}_2).$$

We also need to define the anisotropy symmetry axes. For uniaxial anisotropy we only have one symmetry axis and this is set using the *ea1* parameter by giving the Cartesian components of the unit vector $\mathbf{e}_a$ (e.g. default is 1, 0, 0 for easy axis along the x-axis). For cubic anisotropy we need two symmetry axes directions, *ea1* and *ea2* which should normally be orthogonal.

In the following you will investigate the effect of magnetocrystalline anisotropy on hysteresis loops in circular dots.

Exercise 7.1

a) Set a 160 × 160 × 5 nm permalloy circle with 5 nm cellsize using a mask file. Set a uniform magnetization along the x direction towards the left (blue state). Set uniaxial anisotropy with K1 = 10 kJ/m^3, K2 = 0 J/m^3 and easy axis along x direction.
b) Simulate hysteresis loops along the x-axis (easy axis), y-axis (hard axis) and in between along a 45° in-plane direction (remember you will need to use a polar field sequence for this). You will need to determine appropriate field sweep ranges so the loops start from a saturated magnetization state.

c) Plot the hysteresis loops using the normalized magnetization (divide by M$_S$ value – the saturation magnetization constant). What are the coercivity and normalized remanence values? Explain the difference between the loops.

Exercise 7.2

Repeat the simulations in Exercise 7.1, but this time set cubic anisotropy with K1 = 20 kJ/m^3 and K2 = 0 J/m^3, with two perpendicular easy axes in the plane (e.g. x-axis and y-axis).

Plot the resulting hysteresis loops and compare them with the previous results.

Hints:

For the 45° direction you will need to project the magnetization along the applied field direction. You can do this by taking the dot product of **M** with the applied field direction unit vector:

$$M_H = \hat{\mathbf{h}}.\mathbf{M}$$

You can do this using console data processing with the command:

dp_dotprod *dp_vector ux uy uz dp_out*

Here *dp_vector* is the dp array index such that the dp arrays *dp_vector, dp_vector + 1, dp_vector + 2* hold the x, y, and z components of the magnetization, (*ux, uy, uz*) are the components of a vector (dot product taken with this vector), and *dp_out* is the dp array where the output is placed.

Finally, you can normalize the magnetization using the **dp_div** command where you will need to divide by M_S. To see the value of M_S you can look it up in the list displayed using the **params** command.

Remember you will first need to load into dp arrays the appropriate columns for the saved hysteresis loop data file using the **dp_load** command. You can save the contents of dp arrays after processing data using the **dp_save** command. As always you can find more details about a command by preceding it with the ? symbol, e.g. **?dp_load**.

Tutorial 8 – Anisotropic Magneto-Resistance

In this tutorial you will learn how to simulate magneto-resistance loops and calculate charge current densities using the *transport* module.

Transport Module Basics

The *transport* module is a complex spin and charge current solver (electron transport), allowing for a number of physical effects to be included in the magnetization dynamics problem, including Zhang-Li spin-transfer torques based on calculated charge currents, spin torques based on computed spin accumulations in multilayers, direct and inverse spin Hall effects (SHE and ISHE), spin pumping torques, anisotropic magneto-resistance (AMR), current-perpendicular-to-plane giant magneto-resistance (CPP-GMR), Oersted fields and Joule heating.

Here we will look at how a simple charge current density may be computed and AMR included in the simulation.

You will first need to enable the *transport* module from the list displayed using the **modules** command for the meshes where you want a charge current density to be computed. In the simplest case the computation is reduced to obtaining **J**, the charge current density, using Ohm's law : $\mathbf{J} = \sigma\mathbf{E}$, where σ is the electrical conductivity and $\mathbf{E} = -\nabla V$ is the electrical field with V being the electrical potential. If σ is constant this reduces to a Laplace equation for V.

The electrical conductivity, potential and charge current density are available as display outputs under the **display** command (*elC, V* and *Jc*).

The base electrical conductivity value may be changed by editing the *elC* mesh parameter displayed using the **params** command.

Similarly AMR may be enabled by editing the *amr* mesh parameter (0% by default which disables it). A typical value for permalloy is 2%. Enabling AMR results in a non-uniform electrical conductivity and now the equation for V becomes a Poisson equation.

Before starting a computation you will need to define at least 2 electrodes – these set Dirichlet boundary conditions for V (fixed potential values) in the Laplace/Poisson solvers. The most common electrode configuration is to define two electrodes at the x-axis ends of the mesh (so in the y-z plane). You can do this using the command:

setdefaultelectrodes

To see which electrodes have been defined use the command:

electrodes

You will see two electrode rectangles. The electrode rectangles are in absolute values, so not relative to any particular mesh. You can add new electrodes but they must always be placed at the edges of a mesh rectangle – when initializing the simulation Dirichlet boundary conditions will be flagged for the boundary cells of the mesh intersecting the electrode rectangle. Each electrode has a fixed potential which may be edited. Exactly one of the electrodes has to be designated as the ground – this is the electrode where the outgoing total electrical current is calculated.

You can edit the individual electrode potential values, however a more common scenario is the set a single electrical potential drop from the ground to the other electrodes using:

setpotential *potential*

This sets a single inversely-symmetrical potential drop (i.e. +*potential/2 to* -*potential/2*). The inversely-symmetrical potential drop minimizes floating point errors (as opposed to setting a potential drop of *potential* to 0).

Simulations may use the constant-voltage or the constant-current mode (the interactive object displayed when using the **electrodes** command may be toggled between these two states). Normally you would use the constant-voltage mode; with constant-current the electrode potentials are adjusted during the simulation to maintain a constant current (which may be set using the **setcurrent** command). In this tutorial we will be using the constant-voltage mode.

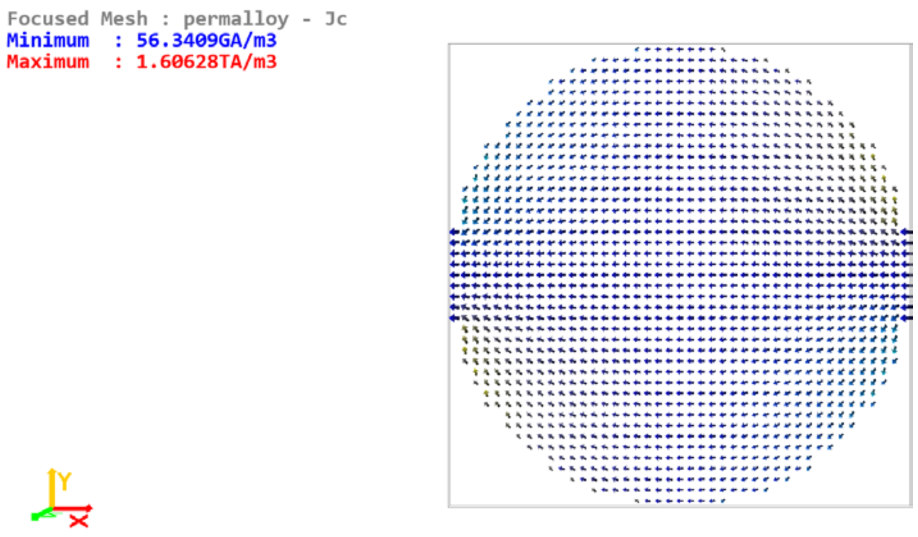
Exercise 8.1

Set a permalloy mesh with dimensions 320x320x10 nm and mask it using a circle shape (in the image file make sure to not leave any white spaces at the left and right sides).

Enable the transport module, set the default electrode configuration and a potential of 10 mV.

In the data box display the average current density (*Jc*), set potential (*V*), total current (*I*) and resistance (*R*). In the mesh display the current density. Run the simulation. The calculated current density should look similar to that in Figure 8.1.

Figure 8.1 – Computed charge current density for Exercise 8.1


```
Focused Mesh : permalloy - Jc
Minimum  : 56.3409GA/m3
Maximum  : 1.60628TA/m3
```

For more advanced simulations you can add electrodes using the **addelectrode** command. Electodes may be deleted using the **delelectrode** command (or more simply by right-clicking on an existing electrode in the list displayed by the **electrodes** command. All electrodes may be deleted by using the **clearelectrodes** command.

Further info:

When displaying the meshes (use the **mesh** command) you will now notice a value for the electric cell. This is the discretization cellsize used by the transport solvers. Normally this should be equal to the magnetic cellsize (default setting) but can be controlled separately for more advanced simulations (decrease computation time or increase computation accuracy as required). Multiple meshes with transport modules enabled may be configured. If the meshes are touching, composite media boundary conditions will automatically be inserted in the computation, however we still typically require 2 electrodes for a well-posed problem.

Exercise 8.2

Set a permalloy rectangle with dimensions 300x100x20 nm. Calculate the current density for the default electrodes setting by using a potential drop of 1 V.

What is the computed sample resistance and does it agree with that predicted by the formula:

$$R = \frac{\rho l}{A},$$

where l is the length, A the cross-sectional area and ρ is the resistivity.

What is the total current, and does it agree with the expected value for a 1 V potential drop?

What is the average current density and does it agree with the expected value for the total current?

Further info:

For advanced simulations the accuracy of the transport solver may be controlled by using the command:

tsolverconfig

This command displays the set convergence error (a value around 10^{-6}) is normally a good compromise between accuracy and computational speed. In this case the Laplace/Poisson solvers stop iterating when the maximum change in V from one iteration to another, normalized to the set potential drop, drops below this set convergence value. You can display the *ts_iter* (current number of transport solver iterations) and *ts_err* (current transport solver error) data in the data box. If the convergence error threshold is too low the transport solver will take a large number of iterations during computations – if AMR, GMR, ISHE or temperature-dependent transport parameters are enabled the transport solver must update after every magnetization and/or heat solver time step.

The transport solver may be used to calculate magneto-resistance loops when an AMR (anisotropic magneto-resistance) value is set. Since this is a static problem it is best to set the *LLGStatic* equation with the *SDesc* solver. When using the *SDesc* solver the stopping condition (*mxh* or *dmdt*) should be much lower than usual, typically at least 10^{-6} or even lower. You should also set the static transport solver flag to On (see **tsolverconfig** command output). Setting this flag to On will only iterate the transport solver at the end of a schedule step, thus resulting in faster computation. In this case you should also increase the iterations timeout to at least 5000 or more to ensure the transport solver converge threshold is met.

For problems where the current density is uniform a transport solver converge threshold of 10^{-6} is sufficient, however this may have to be decreased to 10^{-8} or even 10^{-9} if the current density is highly non-uniform.

In the following problem you should use both the static LLG equation and static transport solver as explained above.

Exercise 8.3

Here you will obtain longitudinal and transverse magneto-resistance loops.

a) Set a permalloy rectangle with dimensions 160x80x10 nm. You can use a 2D simulation with magnetic cellsize 5x5x10 nm, but the transport solver should be left with cubic electric cellsize of 5 nm.

   Sweep the field from -100 kA/m to +100 kA/m strength and back, using a field step of at most 1 kA/m, along a nearly longitudinal direction (use 5° from the x-axis). You should use a *Hpolar_seq* sequence, and an *mxh* stopping condition of 10^{-7}.

   Set the transport solver with default electrodes, a non-zero potential drop (e.g. 1 mV), and *amr* = 2% (see **params**).

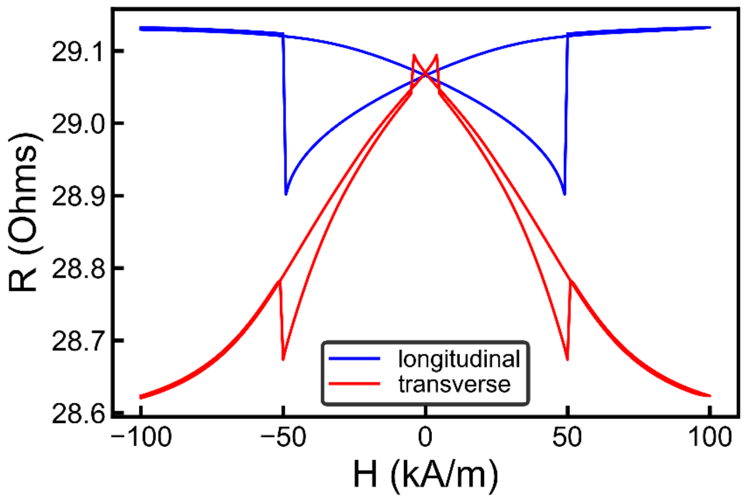
In the output data make sure to save the applied field and sample resistance every step. You should also save the transport solver error to check the converge threshold has been met at the end of each step (*ts_err*).

Run the simulation and plot the obtained MR loop. Explain your results.

b) Repeat the simulation along the in-plane nearly transverse direction (use 85° from the x-axis) by sweeping the field between -100 kA/m and +100 kA/m strength with a step of 1 kA/m.

   Estimate the AMR percentage from the obtained H vs R loops from parts a) and b) – does it agree with the set 2% value?

Figure 8.2 – Longitudinal and transverse MR loops due to AMR as calculated in Exercise 8.3.



Tutorial 9 – Scripting using Python

In this tutorial you will learn how to use scripts to automate multiple simulations. Boris can communicate with an external program via network sockets, so both local and remote script-based control is possible. For this purpose a Python module (NetSocks) is provided, allowing use of Python scripts to communicate with Boris. This is contained in the NetSocks.py file and requires **Python 3.7** or above.

It is strongly recommended Python scripts are run using a Python IDE and not directly from the terminal (although see below for embedded Python script execution). This should be configured with a search path to find NetSocks.py (on Windows this is located in *Documents/Boris Data/BorisPythonScripts* folder, on Linux-based operating systems NetSocks.py is located in the executable folder).

Alternatively Boris can load a Python script directly in emebbed mode and invoke the Python interpreter. Whilst this is an advanced user mode, it has a number of advantages: 1) Scripts may be launcher from the terminal directly, 2) Python methods may be injected into the simulation, which are called every simulation iteration efficiently, allowing the user to define custom data saving and simulation flow control algorithms, 3) script execution in the setup phase is much faster (runtime execution speed not affected), 4) when multiple GPUs are present, Python for loops may be parallelized easily when running in CUDA mode, which is useful for parameter-scan simulations.

The scripts work by invoking console commands, with syntax identical to that you would use when typing commands directly in the console. Some console commands also return values, which can be read by Python scripts. You can find out if a console command is set to return values by looking at the USAGE help for it. For example type the following in the console:

dp_coercivity

In the USAGE help you can see this command will return the calculated coercivity values and uncertainties.

NetSocks Python Commands Usage

Look through the examples in Tutorial 0. To create a new simulation script:

1. Place the NetSocks.py file in the Python include path and import NSClient from NetSocks.

2. Next a NSClient object, *ns*, must be created as:

   ns = NSClient()

   The default configuration is for a local host ('localhost'), i.e. the script must be on the same computer as the Boris running instance. For remote clients and other options see below. Now all communication is done through methods in the *ns* object.

3. After creating the NSClient object, it should normally be configured as:

   ns.configure(True)

   This function sets the working directory the same as the Python script directory, so any simulation outputs are saved in this directory. If calling the function with *True*, the program also resets back to the default starting state. If you don't want to reset to default call with *False*.

See *BorisScriptTemplate_Simple.py* script in this tutorial folder. All commands can be sent through the ns object as ns.command(parameters…), for example *ns.setangle(90, 0)* sets magnetization direction along the x axis, etc.

You can obtain returned parameters as data = ns.command(parameters…), for example *Mav = ns.showdata('<M>')* returns the average magnetization as a list with 3 elements.

In many cases the script must wait for a simulation to finish before proceeding. This is achieved by using the Run function:

- *ns.Run()*

This is a blocking function call, which sends the **run** command, then waits for the simulation to finish by listening to messages from the running program.

NSClient also has a useful method for writing data to a file:

- *ns.SaveDataToFile('Results.txt', [data1, data2, …])*

This command appends a new line in the Results.txt file (in the script directory) which contains two numbers: the x and y components of magnetization.

Another useful built-in command is *Get_Data_Columns*, which loads tab-spaced values from a file into a list (not numpy array), as they would be saved by a simulation. You can also specify a set of column indexes to load if you don't want to load all the output data. This is used as:

- *data = ns.Get_Data_Columns('OutputData.txt', [0, 1, …])*

There are 2 simple plotting helpers, which are just wrappers for matplotlib code designed to generate simple plots in just one line of code (which may be enough for many cases):

- *ns.Plot_Data(x, y, xlabel, ylabel, title, label, imageFile)*
- *ns.PlotPolar_Data(r, theta_deg, xlabel, ylabel, title, label, imageFile)*

Finally for working with OVF2 files 2 functions are provided, which work with numpy arrays:

- *ns.Write_OVF2(filename, vec, nodes, rect_m)*
- *vec, nodes, rect_m = ns.Read_OVF2(filename)*

In the above vec is a numpy 1-dimensional array, nodes = [Nx, Ny, Nz] contains the number of cells along x, y, z, and rect_m = [sx, sy, sz, ex, ey, ez] contains the mesh rectangle start and end coordinates in metres.

Look through the examples in Tutorial 0 and run them. Examples of other scripted simulations will be contained in further tutorials.

Multiple Remote Clients and Advanced Options

You can control a remote Boris instance by specifying the correct IP address and port as:

- *ns = NSClient(scriptserverip, scriptserverport, cudaDevice)*

The server port can be set in the required Boris instance (**serverport** command). This allows controlling multiple Boris instances from the same script, where each instance is assigned a different server port – for example, if you have multiple GPUs on the same workstation you can connect in localhost mode, but have each Boris instance configured to listen on a different port. You can also control multiple Boris instances in a cluster in the same way, or even multiple instances on a multi-GPU workstation by specifying the required CUDA device (*cudaDevice* = 0, 1, 2, …).

NetSocks includes functionality for executing simulations in parallel, which is useful for parameter-scan type of simulations when multiple GPUs are available. In this case NSMultiClient should be used instead of NSClient:

- *nsm = NSMultiClient(scriptserverports, cudaDevices)*

NSMultiClient is a wrapper class for a list of NSClient objects, each with a different server port and cuda device assigned through the *scriptserverports* and *cudaDevices* lists respectively. A simulation method should be defined, which must take an NSClient object as first argument, followed by any number of arguments. To launch multiple simulations use the *Run* method in NSMultiClient, by passing it the simulation method name, followed by a number of arguments. These arguments must be lists of same length – the length determines the number of simulations to run – and the number of arguments must match that expected by the simulation method. As an example see the *BorisScriptTemplate_Multiple.py* example script below. Before running it, there must two Boris instances running, each configured to listen to the respective ports (1000, and 1001), assuming at least 2 GPUs are present.

BorisScriptTemplate_Multiple.py:

```
from NetSocks import NSMultiClient

nsm = NSMultiClient(scriptserverports = range(1000, 1002), cudaDevices = range(0,
2))
#reset to default, but turn off verbosity so messages do not clash due to parallel
execution
nsm.configure(True, False)

####################################

#example simulation method
#this must take an NSClient object as first argument, and any other number of
arguments after
def simulate(ns, H, Ms, angle):

    FM = ns.Ferromagnet([200e-9, 100e-9, 20e-9], [5e-9])
    FM.param.Ms = Ms
    FM.setfield(H, 90, angle)
    FM.addpinneddata('Ha')
    ns.cuda(1)
    ns.Relax(['iter', 10000])

####################################

#example parameters passed to simulate function
H = [10e3, 20e3, 30e3, 40e3]
Ms = [600e3, 700e3, 800e3, 900e3]
angle = 45

#execute simulations (4 in total) in parallel using all configured devices (2
devices, thus 2 simulations each)
nsm.Run(simulate, H, Ms, [angle]*len(H))
```

Note that we used *nsm.configure(True, False)*. The second parameter sets the script verbosity to False, which is recommended with multiple clients to avoid a flood of messages.

Server Configuration

You can set the server port using: **serverport** *port*

You can set the server password using: **serverpassword** *password*

You can set the server response time using: **serversleepms** *time_ms*
The server polls the configured network socket for messages every *time_ms* ms. The default value is 5 ms. Lower values will increase server responsivity, but will increase the CPU load which could impact the execution time of some CPU-based simulations.

Embedded Python Scripts

In embedded mode Boris invokes the Python interpreter, thus communication through network sockets is not required. This presents a number of advantages as shown below. To run a script in embedded mode, then configure the NSClient object:

- *ns = NSClient(embedded = True)*

In emebbed mode the output from *print* commands will appear in the Boris console. Also *matplotlib* plots will be redirected to sequentially numbered png files, saved in the simulation script directory, instead of being shown. This requires the custom *matplotlib* backend module, *boris_matplotlib_backend.py*, to be in the same directory as *NetSocks.py*.

Embedded Python scripts may still be launched from a Python IDE, or alternatively directly from a terminal through the Boris executable. Command line arguments are shown in the table below.

Command Line Arguments	**Description**
-help / -h	Show help. Doesn't launch program.
-version	Show program version. Doesn't launch program.
-p *serverport (password)*	Set script server port and optional password. Default port 1542, and no password.
-g *gpu_select*	Set gpu selection: gpu_select = -1: automatic gpu selection (default). gpu_select = 0, 1, 2, ...: select indicated gpu if available.
-s *python_script_file*	Launch program and execute indicated Python script file (doesn't have to include termination). If path not given then use the default Simulations directory. Other flags shown below modify execution behaviour of Python script.
-sp *stride offset1 (offset2, (…))*	When running Python script from console (specified with -s flag), specify how for loops should be executed in parallel. Embedded Python scripts will attempt to run for loops in parallel, if they are immediately preceded by the directive: #pragma parallel for Default behaviour is to split the for loop iterations evenly between all available GPUs. This setting allows specifying the stride and offsets to be executed by this program instance, in this order: *stride offset1 offset2* …. Stride is the total number of GPUs or machines used to execute the script, offset is any number between 0 and stride - 1. Example scenario: 2 machines are available, first with 2 GPUs and a second with 1 GPU. In this case 2/3rd of the work should be executed on the first machine, and 1/3rd on the second. Thus use a stride of 3 and give offsets 0 and 1 to the first machine, and offset 2 to the second. Thus on the first machine launch: *Boris.exe -s script.py -sp 3 0 1*, and on the second machine launch: *Boris.exe -s script.py -sp 3 2*. Note, the first machine will automatically split the two offsets between the two GPUs. Do not assign more offsets than there are GPUs on any machine.
-sd *0/1*	After execution of Python script from the command line on a multi-GPU machine, delete any temporary Python script files generated : 1 (default), or keep: 0.

When running in embedded mode it is possible to inject Python code into the simulation loop. This takes the form of a Python function which is executed by Boris after every simulation iteration, and enables the user to implement custom advanced execution algorithms and data saving. The function, together with any parameters, should be passed to the *Run* method. The function must return the input parameters (which can be updated during the function exection), preceded by an integer value 0 or 1. This first return parameter determines if the simulation will keep iterating (1) or stop (0). As an example see *BorisScriptTemplate_Embedded.py*, where the simulation stops as soon as a magnetization switching event is detected, and saves data when the change in magnetization along the applied field direction, between data save events, exceeds 5 kA/m. Whilst there is a small cost to executing the Python function every iteration, this is negligible except for very small simulations with fast iteration.

BorisScriptTemplate_Embedded.py:

```python
from NetSocks import NSClient

ns = NSClient(embedded = True); ns.configure(True)

####################################

Py = ns.Ferromagnet([160e-9, 80e-9, 10e-9], [5e-9])

def check_switching(Mx_previous, Mx_previous_save):
    M = Py.showdata('<M>')

    if M[0] * Mx_previous < 0.0:
        print("Switching detected. Stopping simulation.")
        return 0, M[0], Mx_previous_save

    #trigger a save when Mx has changed by more than 5 kA/m
    if M[0] - Mx_previous_save > 5e3:
        Mx_previous_save = M[0]
        ns.savedata()

    return 1, M[0], Mx_previous_save

Py.setfield(4e4, 90, 0)
Py.setangle(90, 170)
ns.setsavedata('Mswitching.txt', ['time'], ['<M>', Py])
Mx = Py.showdata('<M>')[0]
ns.Run(check_switching, Mx, Mx)
```

Another possibility with embedded Python scripts, is to parallelize for loops for multi-GPU workstations. This is useful for parameter-scan type of simulations. The advantage over the un-embedded method of multi-GPU parameter scans is the simplicity of use. Only one Boris instance is required, and a for loop may be parallelized simply by preceding it with the *#pragma parallel for* comment. When this is encountered, all available GPUs will be used, with for loop iterations shared between them. As an example, which is an alternative to the un-embedded method discussed above, see the *BorisScriptTemplate_Multiple_Embedded.py* script.

BorisScriptTemplate_Multiple_Embedded.py:

```
from NetSocks import NSClient

ns = NSClient(embedded = True)

####################################

#example simulation method
def simulate(H, Ms, angle):
    ns.configure(True)

    FM = ns.Ferromagnet([200e-9, 100e-9, 20e-9], [5e-9])
    FM.param.Ms = Ms
    FM.setfield(H, 90, angle)
    FM.addpinneddata('Ha')
    ns.cuda(1)
    ns.Relax(['iter', 10000])

####################################

angle = 45

#pragma parallel for
for (H, Ms) in zip([10e3, 20e3, 30e3, 40e3], [600e3, 700e3, 800e3, 900e3]):
    simulate(H, Ms, angle)
```

Multiple for loops may be parallelized in the same script. By default all available GPUs will be used, however if you want to use only a subset of avbailable GPUs then you should launch the script from a terminal by specifying the required GPUs using the -sp flag. For example, if 4 GPUs are available, to use only the first 2 GPUs the script should be launched as:

Launching the *BorisScriptTemplate_Multiple_Embedded.py* script with 2 GPUs only, on Windows:

```
Boris.exe –s BorisScriptTemplate_Multiple_Embedded.py –sp 2 0 1
```

Launching the *BorisScriptTemplate_Multiple_Embedded.py* script with 2 GPUs only, on Linux-based OS:

```
./BorisLin –s BorisScriptTemplate_Multiple_Embedded.py –sp 2 0 1
```

If all GPUs are required, then you could simply launch the Python script from inside a Python IDE.

Tutorial 10 – Current-Induced Domain Wall Movement

In this tutorial you will learn how to obtain current-induced domain wall velocity curves. First we will simulate these using only console commands, including processing of output data, then we will use a Python script to more accurately determine the domain wall velocity.

The simplest available method for enabling spin torques is to use the Zhang-Li spin-transfer-torque (STT) formulation. In this formulation the calculated charge current density is used to calculate the following spin torques on the magnetization (included as additive terms in the normalized LLG equation):

$$T_{STT} = (\mathbf{u}.\nabla)\mathbf{m} - \beta\mathbf{m}\times[(\mathbf{u}.\nabla)\mathbf{m}]$$

where

$$\mathbf{u} = \mathbf{J_C}\frac{P}{M_S}\frac{\mu_B}{e}\frac{1}{1+\beta^2}$$

To enable this use the **ode** command and select the *LLG-STT* equation with the *RK4* evaluation method. You will also need to enable the *transport* module (use the **modules** command and select the *transport* module). As before you need to set two electrodes (**setdefaultelectrodes**) and enable the moving mesh algorithm (**preparemovingmesh**).

In the above equation we have two new parameters: P, the charge current spin polarisation, and β, the STT non-adiabaticity parameter. These can be edited by using the **params** command and double-clicking on the respective interactive console objects; the default values are set for permalloy.

Exercise 10.1

a) Set a 320 × 80 × 10 nm permalloy rectangle with 5x5x5 nm cellsize (3D problem). Enable the moving mesh algorithm and let the domain wall relax in zero field. Enable the transport module, set the default electrode configuration and enable the LLG-STT equation with the RK4 evaluation method.

b) Set a simulation schedule to vary the current density from 10^{11} A/m^2 up to 10^{12} A/m^2 in 10 steps, saving the domain wall shift (*dwshift*), current (*I*), voltage (*V*), and charge current density (*Jc*) output data. Each current density value should be maintained for 5 ns with a data saving schedule every 10 ps.

Hint: You should use the V_seq simulation schedule stage. This sets a sequence of voltage values with given start and stop values in a number of steps. Since we have a simple geometry you can calculate the required voltage values using the formula:

$$V = \frac{l|\mathbf{J}_\mathrm{C}|}{\sigma}$$

where l is the distance between electrodes (length of the magnetic mesh) and σ is the electrical conductivity (see the set value using the **params** command). *Solution: use V_seq with -4.57mV; -45.7mV; 10.*

Note: You can also vary the current density using the I_seq schedule stage – this defines a sequence of current values. Setting current values directly enables the *constant current* mode, i.e. the voltage drop between electrodes is adjusted during the simulation to keep the current constant (the opposite of this is the *constant voltage* mode).
You can see the current settings using the **electrodes** command.
You can also set individual values in the console using the **setpotential** or **setcurrent** commands.

c) Obtain and plot the domain wall velocity, *v*, using the method introduced in Tutorial 5 with **dp_linreg**, as a function of current density. Convert the current density to spin drift velocity using the formula:

$$u = |\mathbf{J}_\mathrm{C}| \frac{P}{M_S} \frac{\mu_B}{e} \frac{1}{1+\beta^2}$$

Verify that the following formula holds:

$$\frac{v}{u} = \frac{\beta}{\alpha}$$

Domain wall velocity curves may also be obtained using a Python script, allowing more accurate determination of the velocity. The problem with the **dp_linreg** method, it also includes in the regression domain wall displacement points at the start of each step. Since the domain wall requires some time to reach a steady state velocity (the acceleration is not zero at the start of each step) these initial displacement values should ideally be discarded.
With a Python script you can set a single voltage stage without saving any data (e.g. for 3 ns); this is followed by another voltage stage (e.g. again for 3 ns) during which data is saved, then a linear regression is performed to extract the velocity value for the set voltage value. The script then proceeds to set the next voltage value and so on.

Exercise 10.2

Repeat the simulation in Exercise 10.1 but using a Python script to control the simulation flow and data output.
(*Solution: see the attached Python script*)
Note, in general you may need to adjust the stage duration times (1 – to achieve steady state motion and 2 – to generate enough data to obtain a representative velocity value using linear regression) for best results, but the suggested 3 ns, 3 ns breakdown will be sufficient for this exercise.

When setting up the Python script you will need to use the appropriate commands to edit the stage stopping and saving conditions, as well as the stage value. These commands are:

edistagevalue
editstagestop
editdatasave

Note: an alternative and more elegant higher level method of setting up simulation stages and configuring data saving in Python scripts is shown in sections **Simulation Schedules** and **Output Data**, which the solution script uses.

Tutorial 11 – Oersted Fields

In addition to spin-transfer torques, electrical currents may also interact with magnetization via the generated Oersted fields. In this tutorial we will set-up a simple bilayer mesh structure, consisting of a magnetic wire and a non-magnetic metallic capping layer, then repeat the domain wall velocity simulations from the previous tutorial but also taking into account the generated Oersted field. Since the bilayer structure has a broken mirror symmetry in the z direction we might expect the domain wall speed to be different depending on the current direction.

To enable the Oersted field you need to enable the *Oersted* module (use the **modules** command). The *Oersted* module is an electric super-mesh module, i.e. it is calculated on the electric super-mesh with a separately controlled cell-size. After enabling the *Oersted* and *transport* modules bring up the configured meshes using the **mesh** command. You should now notice the electric super-mesh rectangle with its cellsize; the electric super-mesh is simply the smallest rectangle containing all meshes with the *transport* module enabled.

To set a non-magnetic metallic capping layer you will need to add a new mesh, in particular an electrical conductor mesh. This is done using the command:

addconductor *name rectangle*

Exercise 11.1

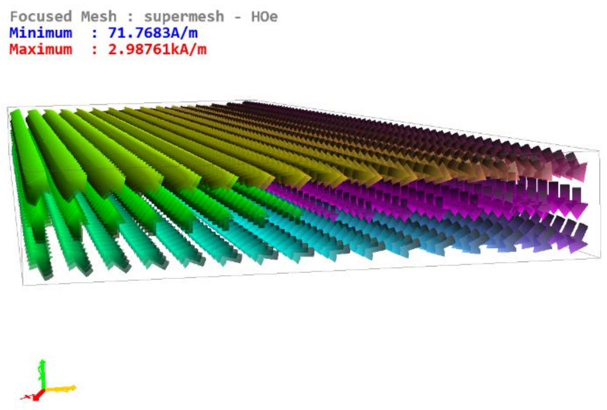
Set-up a simulation space as for Exercise 10.1. Add a metallic capping layer on top of the magnetic layer with a 5 nm thickness and enable its *transport* module (solution: use **addconductor** cap 0 0 10nm 320nm 80nm 15nm).

Set the electrodes to contact both meshes (use **setdefaultelectrodes** after both meshes had their *transport* module enabled).

Enable the Oersted field module and set a potential of 10mV (**setpotential** 10mV).

Compute a single iteration (**computefields**) and display the Oersted field (use the **display** command and select the *Oersted* display option on the super-mesh display line).

Figure 11.1 – Computed Oersted field for Exercise 11.1.



```
Focused Mesh : supermesh - HOe
Minimum  : 71.7683A/m
Maximum  : 2.98761kA/m
```

Exercise 11.2

Continuing from Exercise 11.1, use a Python script to simulate the domain wall velocity for both positive and negative currents in the magnitude range 10^{11} A/m^2 to 10^{12} A/m^2 in 10 steps. Plot the two ve

locity curves and compare them to the expected v = (β/α)u relation, as well as the simulations without an Oersted field. (Solution: see the Python script in the tutorial resources).

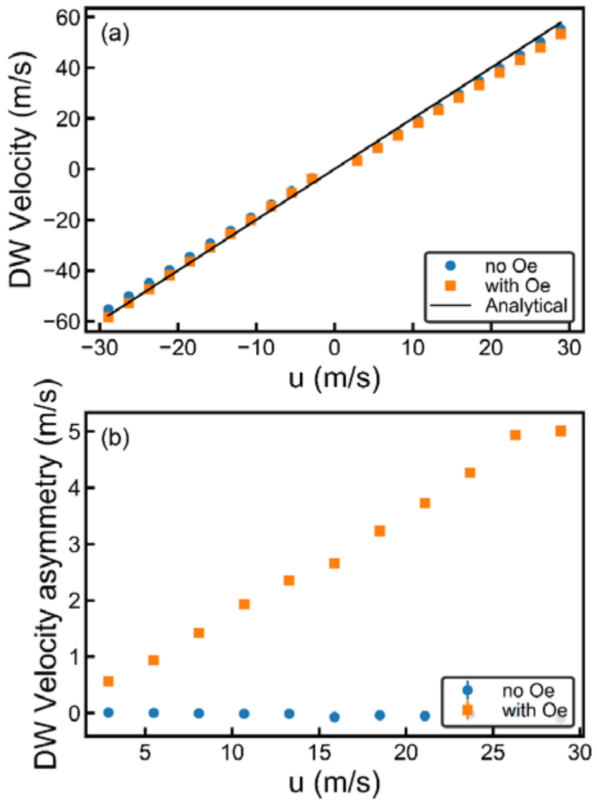
Note: when plotting *v* vs *u* you need to set the sign of *u* to be in the direction of electron drift, i.e. opposite sign to that of the charge current density.

In the above exercise the default electrical conductivity value of the capping layer is the same as for permalloy (this can be edited using the **params** command). In this case the current densities are the same in both layers. If you want to save output data for a particular mesh (e.g. Jc, the charge current density, which is mesh dependent) then you must focus on the required mesh first before adding that particular data to the output list.

You can focus on a particular mesh by clicking on the mesh name (bring up the list of meshes with **mesh** then click on a mesh name). Alternatively you can double click in the mesh graphical viewer on the required mesh.

Results from Exercise 11.2 are shown in Figure 11.2. The data obtained in Figure 11.2 could be improved further. The problem with using linear regression directly on the raw *dwshift* data, it contains steps due to the mesh discretisation. This is particularly problematic at low domain wall velocities where only a few steps are contained in the raw data, which can make the extracted velocity inaccurate.

Figure 11.2 – Simulation results from Exercise 11.2 showing a) domain wall velocity plotted against spin drift velocity for cases with and without Oersted fields and also compared with the analytical formula, and b) domain wall speed difference plotted against spin drift speed for cases with and without Oersted fields.



One possibility is to collect data for a longer time, but this is inefficient. Another possibility is to assume the domain wall displacement is linear with time (in this exercise this is a valid assumption). You can then either get rid of the repeated points before using linear regression, or replace the repeated points using linear interpolation. There is a built-in command for this, and is covered in a later tutorial on skyrmion movement (**dp_replacerepeats**).

Tutorial 12 – Surface Exchange, Multi-Layered Demagnetization and CUDA

Surface Exchange

Using the *surfexchange* module, two or more ferromagnetic meshes can be surface exchange coupled, allowing simulations of magnetic multilayers with RKKY interaction. The strength of the surface exchange coupling is controlled using the *J1* and *J2* material parameters: negative values result in anti-ferromagnetic coupling, positive values in ferromagnetic coupling. The *J1* parameter controls the strength of bilinear surface exchange, and *J2* controls the strength of biquadratic surface exchange.

Type **params** and have a look at *J1* and *J2*. For two meshes in surface exchange coupling, it is the top mesh *J1* and *J2* values that are used. This allows setting different coupling strength and types for the bottom and top of a mesh in a multi-layered structure. Boris allows surface exchange coupling only for xy planes, thus a multi-layered structure should be designed with the layers stacking along the z direction. Two ferromagnetic meshes will be surface exchange coupled if they both have the *surfexchange* module enabled and there’s no other ferromagnetic mesh with the *surfexchange* module enabled in between them along the z direction. The coupling will only be calculated for cells which overlap in the xy plane.

Multi-Layered Demagnetization

To add another ferromagnetic mesh use the **addmesh** command. The *demag* module for each mesh only calculates the demagnetizing field for that ferromagnetic mesh alone. When 2 or more ferromagnetic meshes are being used, if you want to compute the overall demagnetizing field you need to use the supermesh *sdemag* demagnetizing field module. In this case the individual *demag* modules are disabled and the overall demagnetizing field is computed for the collection of individual ferromagnetic meshes, including all stray field contributions. There are two ways to do this. The default method is called multi-layered convolution, and you can see the settings for this by using the command:

multiconvolution

This is an exact method of computing demagnetizing fields for a collection of computational meshes, which is able to handle arbitrary spacing and relative positioning between the layers without sacrificing accuracy or computational performance – see S. Lepadatu, Journal of Applied Physics 126, 103903 (2019) for details. In many cases you can rely on the default settings for multi-layered convolution, but for more advanced control you should read the information below.

Further info:

You can force the demagnetizing field to be computed in a 2D approximation in each mesh by clicking on the respective interactive console object (see output of the **multiconvolution** command, Force 2D : 2D meshes). This allows each mesh to have an arbitrary thickness, and is appropriate if the meshes are thin enough for the 2D approximation to hold. This option is turned off by default (Force 2D : Off), resulting in a 2D or 3D convolution in each mesh depending on their cellsizes. If the 2D approximation for each mesh is not appropriate you should use this option as long as the z cellsizes are the same in all meshes. If the z cellsizes differ the computation with Force 2D : Off may not be accurate, and in this case you should use the 2D layering option (Force 2D : 2D layered). This is similar to the 2D meshes option, however instead of forcing each mesh to be 2D, it is instead split into multiple layers, with each layer thickness set by the z cellsize in each respective mesh. This option is thus the most general case and allows exact computation of demagnetizing fields without any approximations, however it is in general more computationally expensive and should only be selected if the other 2 options are not appropriate.

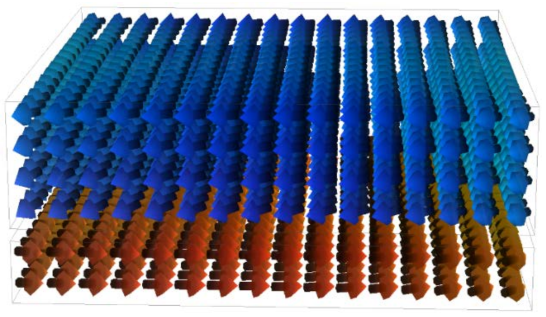
For multiple computational meshes with unequal sizes, the algorithm works by first transferring the magnetization values to scratch spaces with a common discretization. You can specify what this discretization should be, but by default it is calculated for you (see output of the **multiconvolution** command).

Another method of calculating demagnetizing fields for a collection of computational meshes is to use the so-called supermesh demagnetization. This is achieved by disabling the multi-layered convolution algorithm (see output of the **multiconvolution** command). This method calculates the demagnetizing field on the ferromagnetic supermesh by transferring magnetization and demagnetizing field values to and from the ferromagnetic supermesh using a local averaging smoother. Remember the ferromagnetic supermesh is the smallest rectangle containing all the ferromagnetic meshes and can be viewed using the **mesh** command. Computations on the ferromagnetic supermesh are done using its independent cellsize as can be seen in the console output of the **mesh** command. This cellsize will need to be carefully determined in each case to ensure accuracy of results. As a rule you should set the cellsize to be the minimum value out of the individual mesh cellsize values required to compute the demagnetizing field accurately separately. In most cases of interest, if full accuracy is required, this method is much slower than multi-layered convolution. It is also far less flexible, as it cannot accurately handle spacing between layers which cannot be exactly discretized.

Another use for the *sdemag* module without multi-layered convolution, is to calculate the demagnetizing field in an individual mesh with a different cellsize to that used for the exchange interaction. The exchange interaction typically requires a smaller cellsize to ensure accuracy, thus this method can be used to improve computational speed for larger simulations.

Starting from the **default** state, add another ferromagnetic mesh with dimensions of 80 nm × 80 nm × 20 nm, separated from the first mesh by 2 nm along the z direction. Enable the *surfexchange* module for both meshes, as well as the *sdemag* module. Now **run** the simulation and observe the result – see Figure 12.1.

Figure 12.1 – Anti-ferromagnetic surface exchange coupled ferromagnetic meshes



You can also add a metallic spacer layer in between, using the **addconductor** command, however this will not affect magnetic computations directly; it will be needed however if charge or spin transport computations are also enabled.

Exercise 12.1

a) Set-up a synthetic ferrimagnetic (SyF) Ni80Fe20 bilayer with elliptical shape of 320 nm × 160 nm, thickness values of 20 nm and 10 nm respectively, and with a separation of 2 nm between layers. Simulate a hysteresis loop for this SyF structure along 1° to the x-axis direction, plotting the average magnetization for the entire bilayer against field (you will need to calculate this from the magnetization saved for the two layers separately – see notes below).

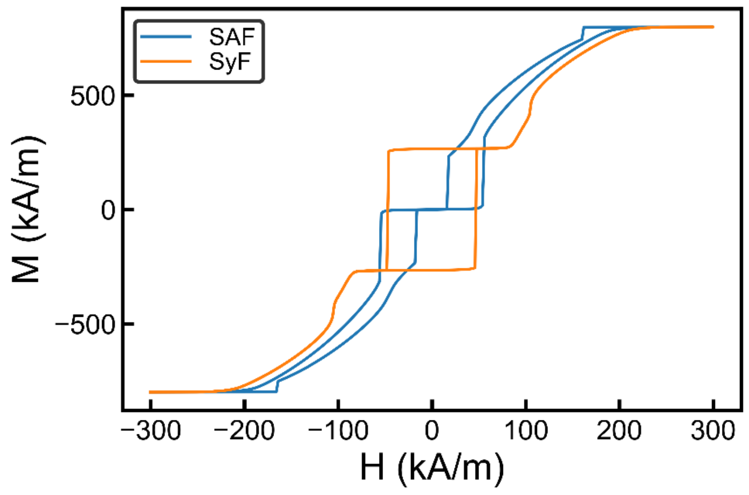
   You should use the *SDesc* with *LLGStatic* solver (use **ode** command) and set an *mxh* threshold value not higher than 10^{-5}. The field step should not be greater than 1 kA/m.

b) Repeat the same exercise but this time set-up a synthetic antiferromagnetic bilayer (SAF) with the same overall thickness as above – i.e. 15nm thick layers with 2 nm spacing.

When adding data to the output list, you can select the mesh for which it applies, if applicable, e.g. magnetization output. You can do this either by adding data when the required mesh is in focus, or editing that data entry later to change the applicable mesh name. For the exercise above you will need to add to the output the magnetization for both meshes as two separate entries (use **data** command and follow instructions therein).

You will need to apply the field for the hysteresis loop to both meshes, not just the first *permalloy* mesh. Use the **stages** command and add a *Hpolar_seq* stage. You will see the field is set to be applied only to the current mesh in focus, e.g. *Hpolar_seq <permalloy>*. Instead, you will need to edit this by double-clicking on the added stage entry, and changing the name from *permalloy* to *supermesh*. After this the entry should read *Hpolar_seq <supermesh>*. The field sequence will now be applied to both ferromagnetic meshes.

Figure 12.2 – Hysteresis loops obtained for the SyF and SAF bilayers in Exercise 12.1



When working with multiple ferromagnetic meshes, all the commands that affect changes in a ferromagnetic mesh have an optional parameter which specifies which mesh to use. If only one ferromagnetic mesh is created the mesh name doesn't need to be specified explicitly. If multiple meshes are used, unless the name is specified the settings are applied either to the current mesh in focus, or to the supermesh, depending on the command.

For example the **setangle** and **setfield** commands will make changes to all ferromagnetic meshes unless a specific name is specified – see the help for these commands (**?setangle**, **?setfield**). On the other hand the **loadmaskfile** command (remember you can just drag a .png file to the mesh viewer instead of typing this command) only applies the shape to the current mesh in focus.

CUDA Computations

If you have a CUDA-enabled graphics card you can enable GPU computations using the **cuda** command:

cuda 1

If CUDA computations are not available for your computer, typing the **cuda** command will show an N/A status. You can also see how much CPU and GPU-addressable memory you have by using the **memory** command.

When using CUDA computations, for optimum efficiency you will want to limit the display update frequency (use the **iterupdate** command). Boris is designed to limit memory transfers between GPU and CPU-addressable memory to an absolute minimum, as this is a critical bottleneck in performance. To display mesh data in the viewer, an average display data is computed on the GPU, then transferred to CPU-addressable memory so it can be used by graphics routines. This transfer can slow down computations, especially if the display viewing coarseness is small (remember you can use the mouse wheel to change viewing coarseness).

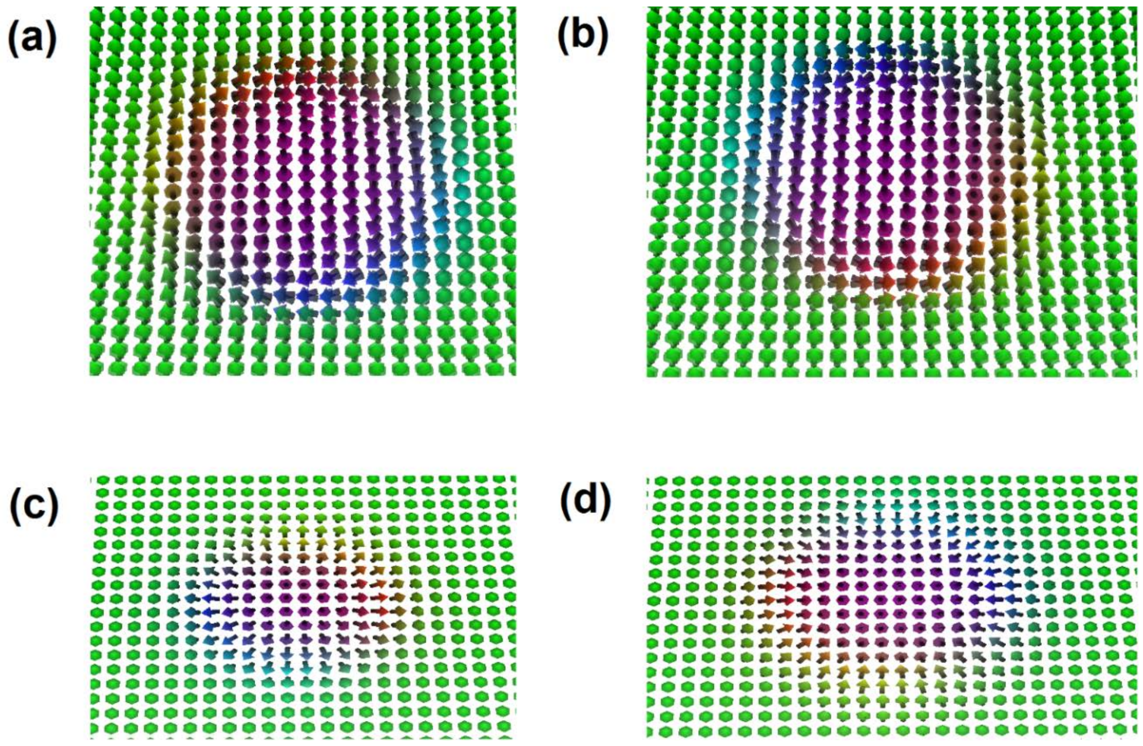
Tutorial 13 – Dzyaloshinskii-Moriya Exchange

Dzyaloshinskii-Moriya interaction (DMI) may be included in the simulation by enabling the *DMExchange*, or the *iDMExchange* module. The former is used for bulk DMI, whilst the latter is used for interfacial DMI. The strength of the DMI interaction is controlled using the *D* material parameter (use **params** command). Néel skyrmions may be generated in the xy plane using the **skyrmion** command:

skyrmion *core chirality diameter position*

In the above command the *core* parameter sets the z direction of the skyrmion core (-1 or 1), the *chirality* parameter sets the radial direction rotation (-1 for away from core, 1 for towards core). The *diameter* and *position* may be specified using metric units, with the *position* requiring 2 components. **Figure 13.1** shows examples of relaxed Néel (*iDMExchange*) and Bloch (*DMExchange*) skyrmions for both D > 0 and D < 0.

Figure 13.1 – a) Bloch skyrmion for D < 0, b) Bloch skyrmion for D > 0, c) Néel skyrmion for D < 0, and d) Néel skyrmion for D > 0.



For the following exercises, when computing a relaxed magnetization state you should use the *SDesc* with *LLGStatic* solver, remembering to set a low *mxh* convergence value (at least 10^{-6}).

Exercise 13.1

a) Obtain relaxed Néel skyrmions for both D > 0 and D < 0 in an ultrathin (1 nm) Co layer with perpendicular magnetization. You will need to use the *iDMExchange* module. Use material parameters as Ms = 600 kA/m, A = 10 pJ/m, |D| = 1.5 mJ/m^2, K1 = 380 kJ/m^3 for uniaxial anisotropy with easy axis along z direction. To reduce the skyrmion diameter apply an out-of-plane magnetic field opposing the skyrmion core, e.g. 15 kA/m along the 0, 0 (polar coordinates) direction.

b) Obtain Bloch skyrmions for both D > 0 and D < 0. You may use the same parameters as above, but this time set a thickness of 10 nm and use the *DMExchange* module. You will need to set a larger out-of-plane magnetic field to control the skyrmion diameter.

c) For part a) compute the skyrmion diameter and topological charge.

To complete Exercise 13.1 c), you will need to extract a profile along the skyrmion diameter, then fit it using an analytical model for a skyrmion. Boris provides this functionality through the **dp_fitskyrmion** command. The z component of **M** is fitted using the following formula (see X.S. Wang et al., Commun. Phys. 1, 31 (2018)):

$$M_Z(x) = M_S \cos\left(2\arctan\left(\frac{\sinh(R/w)}{\sinh((x-x_0)/w)}\right)\right), \quad w = \frac{\pi D}{4K}, \quad K = K_u - \mu_0 M_S^2/2$$

Here M_S (saturation magnetization), R (skyrmion radius), x_0 (skyrmion center position), and w are used as fitting parameters, and in particular the M_s and w values after the fit should match the expected values from the material parameters set.

Thus to obtain the skyrmion radius first use **dp_getprofile** to extract a profile through the center of the skyrmion, then use **dp_fitskyrmion** on the z component of **M**. The fitting works for both skyrmion topological charges, but the fitted value of M_S will change sign depending on the sign of the topological charge.

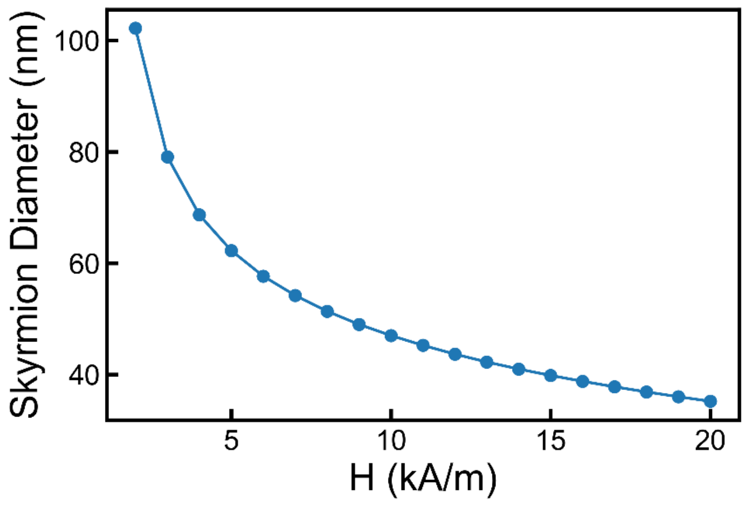
To calculate the topological charge, there is a built-in command, **dp_topocharge**. This command solves the following formula over the xy plane S:

$$Q = \frac{1}{4\pi}\int_S \mathbf{m}.\left(\frac{d\mathbf{m}}{dx} \times \frac{d\mathbf{m}}{dy}\right) dxdy$$

Exercise 13.2

For the same parameters as in Exercise 13.1, compute the skyrmion diameter as a function of out-of-plane field strength from 2 kA/m to 20 kA/m in 1 kA/m steps. You should set-up a Python script to automate this simulation.

Figure 13.2 – Skyrmion diameter as a function of out-of-plane field strength obtained in Exercise 13.2.



Tutorial 14 – Simulations with non-zero temperature

Landau-Lifshitz-Bloch equation and Curie temperature

Non-zero temperature simulations may be performed by using the Landau-Lifshitz-Bloch (LLB) equation – use the **ode** command then select the appropriate equation to solve.

With a non-zero temperature the magnetization length is no longer a constant, but depends on the applied field strength. This is modelled via a longitudinal susceptibility included in the LLB equation. Instead, we talk about the *equilibrium magnetization*, which is the stable magnetization length at a given field. Thus at zero temperature the equilibrium magnetization coincides with the saturation magnetization. With a non-zero temperature the equilibrium magnetization gradually decreases, reaching zero at the Curie temperature. Other parameters which change with temperature include the exchange stiffness and magnetization damping. With a non-zero temperature the damping is now divided into two terms: transverse damping (coincides with the Gilbert damping at zero temperature) and longitudinal damping.

In the simplest case the temperature inside the mesh is uniform, and is controlled using the **temperature** command, which sets the mesh *base temperature*:

temperature *value (meshname)*

When enabling the LLB equation you will also need to set appropriate temperature dependences for some material parameters, including *Ms*, *damping*, *A*, and *susrel* (the relative longitudinal susceptibility). The longitudinal damping used in the LLB equation is not available as a separate material parameter, but is automatically calculated based on the transverse damping parameter (*damping*). Default temperature dependences for these parameters may be generated based on the Curie temperature of the material – for details see S. Lepadatu, Journal of Applied Physics 120, 163908 (2016). You can do this using the **curietemperature** command:

curietemperature *value (meshname)*

Setting material parameters temperature dependences

Exercise 14.1

Set a Curie temperature of 870 K (appropriate for Ni80Fe20) and obtain plots of the temperature dependences of the *Ms*, *damping*, *A*, and *susrel* material parameters (see below).

Almost all material parameters available in Boris can be assigned a temperature dependence. This is achieved by specifying a scaling law, *t*. The value of a parameter at a temperature T is then obtained as value_at_T_K = value_at_0_K × *t*(T) – any computational routine in Boris for a which a parameter is used, obtains an updated value in this way where appropriate, where T is either the base temperature (uniform temperature mode) or the local temperature (non-uniform temperature mode). To see the currently set temperature dependences use the **paramstemp** command. You can set a temperature dependence by supplying a text equation, where *T* is the temperature value (see console output for **paramstemp**), or by loading an array using the **setparamtemparray** command. Further details on using text equations are given in a dedicated tutorial. Once a temperature dependence array has been set (e.g. after using the **curietemperature** command), you can load the set temperature dependence into an internal data processing array using the **dp_dumptdep** command:

dp_dumptdep *meshname paramname max_temperature dp_index*

For example to see the set temperature dependence for *Ms* use the following:

dp_dumptdep *permalloy Ms 870 0*
dp_save Ms_scaling 0

In the file Ms_scaling.txt (saved under the current working directory – see **data** command) you will see a single column with the scaling coefficients. These are saved in increments of 1 K, from 0 K up to 870 K. Internally the scaling coefficients are obtained from the user loaded array at 1 K increments, irrespective of how the user specified the temperature dependence – missing temperature points are filled in using interpolation. During computations the scaling coefficients are obtained by interpolating the nearest 2 temperature scaling points. To reset all parameters temperature dependences you can use the **clearparamstemp** command.

Field dependence of material parameters temperature dependences

With non-zero temperature simulations, the equilibrium magnetization also depends on the strength of the applied magnetic field. This dependence is enabled by setting the strength of the net atomic moment of the material, specified in Boris as multiples of the Bohr magneton. This is done using the command:

atomicmoment *(value)*

If this value is not zero, whenever the applied magnetic field changes, the temperature dependences of all the parameters affected by the Curie temperature setting (see above) are recalculated. Moreover, the longitudinal susceptibility is directly proportional to this value so must be set correctly whenever the LLB equation is used.

Non-zero temperature simulations

Exercise 14.2

Simulate the hysteresis loops in a 160 × 160 × 5 nm permalloy circle at zero temperature (using the LLG equation), as well as at room temperature (297 K, using the LLB equation) and compare the two loops. With the LLB equation the time step for numerical stability is usually lower – you might need to investigate this.

Tutorial 15 – Thermal Fields

When non-zero temperature modelling is considered, an additional effect that can be included is lattice thermal agitation. This gives rise to fluctuations in magnetic moments, and may be modelled by introducing appropriate stochastic fields and torques. In Boris thermal fields may be enabled by selecting a stochastic magnetization dynamics equation, e.g. sLLB – use the **ode** command then select the appropriate equation to solve. For further information see S. Lepadatu & M.M. Vopson, Materials 10, 991 (2017).

When solving stochastic equations, the choice of available ODE evaluation methods is more limited since they must be able to handle the stochasticity introduced. Currently the best fixed time-step method available in Boris is RK4. Since this method is a fixed time step method you will need to investigate the time step required for numerical stability. You can use the default time step as a starting point.

There is also an adaptive time-step method called *AHeun* which you can use.

Exercise 15.1

Simulate out-of-plane hysteresis loops at room temperature in a 256 nm × 256 nm Co rectangle with perpendicular magnetization, with 4 nm thickness, and cubic 4 nm cellsize. Use material parameters as Ms = 600 kA/m, A = 10 pJ/m, and K1 = 380 kJ/m^3 for uniaxial anisotropy with easy axis along z direction. You should simulate hysteresis loops with and without thermal fields for comparison.

Tutorial 16 – Heat Flow Solver and Joule Heating

Heat equation

Non-uniform temperature simulations may be enabled by selecting the *heat* module. Any mesh with this module enabled will solve the heat equation as a function of time. If any two meshes with the *heat* module enabled are in contact, then heat flow across the interface (also referred to as a composite media boundary) is automatically calculated based on the continuity of heat flux and temperature perpendicular to the composite media boundary.

There is a special type of mesh, referred to as an *insulator* mesh in Boris, which can be used to model substrates. You can add an insulator mesh using:

addinsulator *name rectangle*

When the *heat* module is enabled, the *thermal cell* discretisation cellsize becomes available in the mesh descriptions (use the **mesh** command). This can be controlled independently of the magnetic and electric cellsize (if enabled), and can also be set independently of other cellsize values in other meshes.

The heat equation time step may be set using:

setheatdt *value*

This value shouldn't be larger than the magnetization dynamics equation time step (**setdt**), since during computations the heat equation time is incremented only up to the current magnetization equation time (the global time, or total time – see the *time* output data). If this value is lower, the heat equation will be iterated multiple times until it catches up to the magnetization equation time.

The mesh temperature may be set as before using the **temperature** command. This sets a uniform mesh temperature as a starting point, but depending on the simulation configuration the mesh temperature can change. This is true particularly if the mesh ambient temperature is different. For the heat equation, boundary conditions for cells not at a composite media boundary are set based on Newton's law of cooling – i.e. Robin boundary conditions are used. These require an ambient temperature (the surrounding temperature) and a heat transfer coefficient (the Robin coefficient). To adjust these values you may use the **ambient** command, then double click on the respective interactive objects to modify their values. Note, when the **temperature** command is used, setting a mesh temperature automatically sets the ambient temperature to the same value too. You may also choose to have thermally insulating boundary conditions by selecting the appropriate options displayed by the **ambient** command.

Parameters for heat transport

A few parameters are used to specify the thermal properties of the material, in particular the *thermK* (thermal conductivity) and the *shc* (specific heat capacity) material parameters – see these by using the **params** command. Additionally the *density* (mass density) parameter also enters the heat equation.

Note, all material parameters with a temperature dependence enabled will now also vary non-uniformly throughout the mesh (if *heat* module enabled), taking on the value set by the local cell temperature value.

You may obtain the mesh average temperature through the <*Temp*> output data – use **data** command. The mesh temperature may also be displayed (*Temp*) – use the **display** command.

Joule heating

If the *transport* module is also enabled in the same mesh as the *heat* module, Joule heating is taken into account. This results in a heat source term in the heat equation due to the charge current density, **J** as:

$$C\rho \frac{\partial T(\mathbf{r},t)}{\partial t} = \nabla . K \nabla T(\mathbf{r},t) + \frac{\mathbf{J}^2}{\sigma}$$

In the next exercise you will investigate the effect of a voltage pulse on a $Ni_{80}Fe_{20}$ nanowire placed on a SiO_2 substrate, similar to the work in S. Lepadatu, Journal of Applied Physics 120, 163908 (2016). To model a very long wire on a substrate (say the wire is oriented along the x-axis) you should set the x-axis ends of both the magnetic wire and substrate as insulating since in this case the heat flux is oriented only along the y and z directions. Similar considerations apply to the substrate if you want it to be effectively infinite in the x-y plane and depth – set insulating boundary conditions in the required directions. Note in this latter case the modelled substrate must still be large enough for the temperature evolution to be correct for the required duration – for details see S. Lepadatu, Journal of Applied Physics 120, 163908 (2016). The x-axis ends of the magnetic wire should have electrodes so a uniform current density is achieved – remember you can use the **setdefaultelectrodes** command. You can leave the Robin heat transfer coefficient (see **ambient** command output) to the default value as this is appropriate for ventilated air surrounding.

The SiO_2 substrate may be added by using the **addinsulator** command and enabling its *heat* module. You will also need to enter appropriate values for thermal conductivity, specific heat capacity and density, and similarly for the $Ni_{80}Fe_{20}$ magnetic wire.

Note, typically the *thermal cellsize* can be greater than the magnetic (or electric) cellsize, and again can be set independently in different meshes (composite media boundary conditions do **not** require the discretizations to match on the two contacting meshes, for any computational routines used in Boris; generic composite media boundary computational routines are used which are second order accurate in space for all meshes). For the purposes of the next exercise you can use a simple cubic cellsize with a 10 nm side for all the thermal discretization lengths (note, if the mesh thickness is 10 nm then the z cellsize will be adjusted so there are at least 2 computational cells along the z direction – 3D solver used).

Exercise 16.1

Create a $Ni_{80}Fe_{20}$ nanowire with 160 nm width, 10 nm thickness and 640 nm length, centered on a SiO_2 substrate with 800 nm width, 640 nm length and 150 nm depth.

Enable heat equation computations in both meshes and transport module in the $Ni_{80}Fe_{20}$ nanowire. By setting appropriate insulating boundary conditions for heat conduction, define the nanowire to be effectively infinite along the x axis, and the substrate elongated in the x-y plane and depth. Set the ambient temperature (as well as the base temperature – the starting temperature) to be the room temperature value (297 K). The default thermal conductivity, specific heat capacity and mass density values are appropriate for $Ni_{80}Fe_{20}$. For SiO_2 you should edit these as K = 1.4 W/mK (*thermK*), C = 730 J/kgK (*shc*), and ρ = 2200 kg/m^3.

a) Set a voltage step with 50 ns duration which results in a current density of 10^{12} A/m^2 at T = 297 K. Use a temperature dependence for the electrical conductivity σ such that:

$$\sigma = \frac{\sigma_0}{1+0.025T}$$

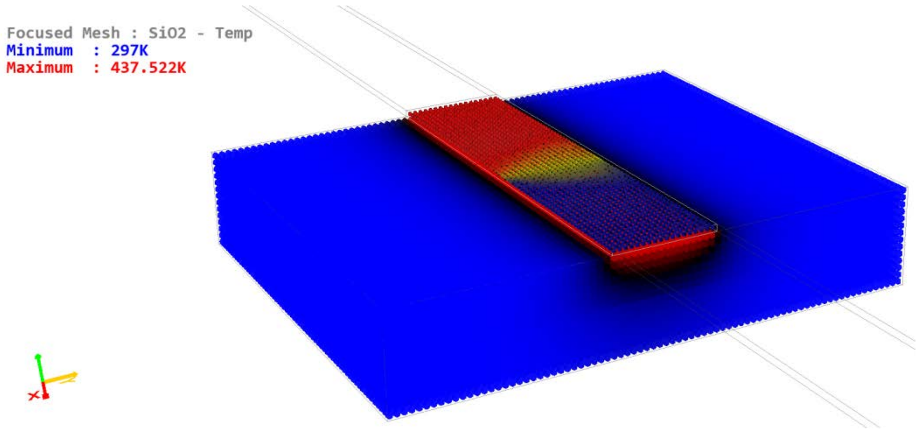
The above formula represents the default temperature dependence set for electrical conductivity - see **paramstemp** command. Obtain the average temperature and current density as a function of time in the $Ni_{80}Fe_{20}$ mesh both for the heating cycle (first 50 ns), as well as the next 50 ns of the cooling cycle when the voltage is set to zero. (Note, just for this part, to speed up the computations you may want to disable any magnetic computations in the $Ni_{80}Fe_{20}$ nanowire by disabling the *demag*, *exchange*, and *zeeman* modules).

b) Set a transverse domain wall in the center of the nanowire through the **preparemovingmesh** command and relax it. Redo the simulation in part a), but this time also obtain the domain wall displacement as a function of time when using the LLG-STT equation (use the **ode** command).

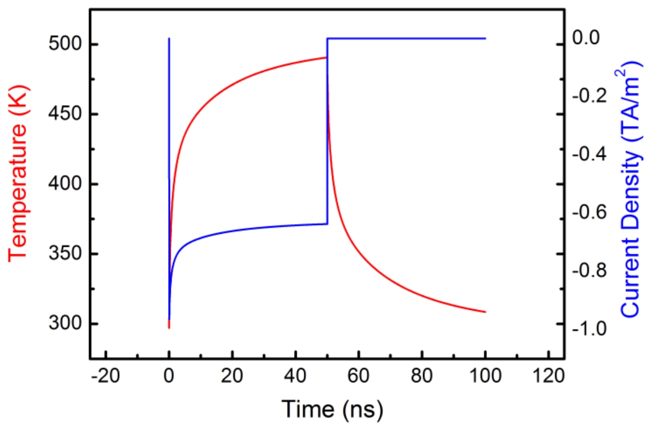
c) Repeat part b) but this time use the LLB-STT equation with a Curie temperature of 870 K. Note, you will need to reduce the time step significantly for the LLB equation for numerical stability.

Figure 16.1 – a) Geometry used for Exercise 16.1, showing a magnetic wire on a SiO_2 substrate with Joule heating computations enabled – heat is generated in the magnetic wire due to an applied charge current density. b) Average temperature in the permalloy nanowire, also showing the current density during and after the applied voltage pulse. c) Domain wall displacement simulated using the LLG-STT and LLB-STT equations. For the latter a Curie temperature of 870 K was set.

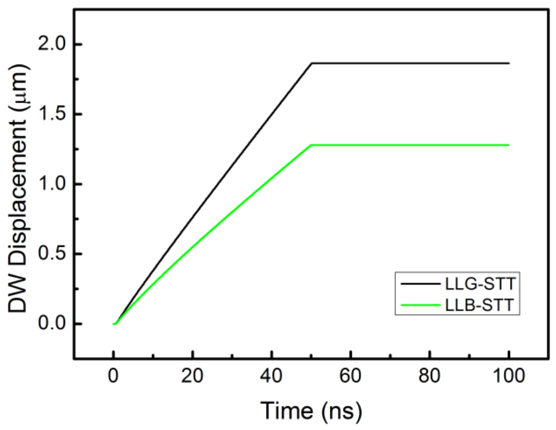
a)



b)



c)



Tutorial 17 – Spin Transport Solver

In addition to the simple Ohm's law used to obtain the charge current density with the *transport* module, Boris also integrates a 3D spin current solver based on the spin drift-diffusion equations – see S. Lepadatu, Scientific Reports 7, 12937 (2017). This solver allows for a number of effects to be computed self-consistently in arbitrary multi-layered geometries and integrated with the magnetization dynamics solver. These include the spin Hall effect (SHE), inverse SHE, CPP-GMR, spin diffusion and non-local spin transport effects, spin pumping, as well as bulk and interfacial spin torques calculated from the spin accumulation and composite media boundary conditions.

The spin transport solver computes both the charge and spin polarisation current densities, $\mathbf{J}_C$ and $\mathbf{J}_S$ respectively, together with the charge potential, V, and spin accumulation, **S**:

$$\mathbf{J}_C = \sigma\mathbf{E} + \beta_D D_e \frac{e}{\mu_B}(\nabla\mathbf{S})\mathbf{m} + \theta_{SHA} D_e \frac{e}{\mu_B}\nabla\times\mathbf{S} + P\sigma\frac{\hbar}{2e}\mathbf{E}^{\sigma} - P\frac{\sigma^2\hbar}{e^2 n}\mathbf{E}\times\mathbf{B}^{\sigma}$$

$$\mathbf{J}_S = -\frac{\mu_B}{e}P\sigma\mathbf{E}\otimes\mathbf{m} - D_e\nabla\mathbf{S} + \theta_{SHA}\frac{\mu_B}{e}\boldsymbol{\varepsilon}\sigma\mathbf{E}$$
$$+\frac{\hbar\mu_B\sigma}{2e^2}\sum_i \mathbf{e}_i \otimes (\dot{\mathbf{m}}\times\partial_i\mathbf{m}) + \frac{\hbar\mu_B\sigma^2}{e^3 n}(\mathbf{z}\times\mathbf{E})\otimes(\partial_x\mathbf{m}\times\partial_y\mathbf{m})$$

where:

$$E_i^{\sigma} = (\dot{\mathbf{m}}\times\partial_i\mathbf{m}).\mathbf{m}$$
$$\mathbf{B}^{\sigma} = \mathbf{z}(\partial_x\mathbf{m}\times\partial_y\mathbf{m}).\mathbf{m}$$

Here $\mathbf{E}^{\sigma}$ and $\mathbf{B}^{\sigma}$ are the directions of the emergent electric field due to charge pumping, and emergent magnetic field due to topological Hall effect respectively. $\mathbf{J}_S$ is a rank-2 tensor such that $\mathbf{J}_{Sij}$ signifies the flow of the *j* component of spin polarisation in the direction *i*. The electric field is given by **E** = -∇V and **S** satisfies the equation of motion:

$$\frac{\partial\mathbf{S}}{\partial t} = -\nabla.\mathbf{J}_S - D_e\left(\frac{\mathbf{S}}{\lambda_{sf}^2} + \frac{\mathbf{S}\times\mathbf{m}}{\lambda_J^2} + \frac{\mathbf{m}\times(\mathbf{S}\times\mathbf{m})}{\lambda_{\varphi}^2}\right)$$

In the above equations we have a number of material constants which can be controlled via the **params** command:

- D_e is the electron diffusion constant
- β_D is the diffusion spin polarisation - this term leads to CPP-GMR
- P is the charge current spin polarisation – this terms leads to Zhang-Li spin transfer torques, among other effects
- θ_{SHA} is the spin Hall angle (unitless) – the term in the equation for $\mathbf{J}_S$ leads to SHE, whilst the term in the equation for $\mathbf{J}_C$ leads to the inverse SHE; Note there are two related parameters available in Boris: *SHA* and *iSHA*. These represent the spin Hall angle, but may be set to different values, allowing the SHE or inverse SHE to be turned on or off in the computations by setting one or the other to zero.
- λ_{sf} is the spin flip length
- λ_J and λ_φ are the exchange rotation and spin dephasing lengths respectively, describing the absorption of transverse spin components (transverse to **m**, the magnetization direction) within a ferromagnetic material
- n is the carrier density (m^{-3})
- Charge pumping and topological Hall effect may be turned on or off by setting the *pump_eff* and *the_eff* values to 1 or 0 (disabled by default).

Bulk spin torques are included in the computations as:

$$\mathbf{T}_S = -\frac{D_e}{\lambda_J^2}\mathbf{m}\times\mathbf{S} - \frac{D_e}{\lambda_\varphi^2}\mathbf{m}\times(\mathbf{m}\times\mathbf{S})$$

This term is included in the implicit LLG (or LLB) equation as (in practice this term results in an effective field which is added to $\mathbf{H}_{eff}$):

$$\frac{\partial\mathbf{m}}{\partial t} = -\gamma\mathbf{m}\times\mathbf{H}_{eff} + \alpha\mathbf{m}\times\frac{\partial\mathbf{m}}{\partial t} + \frac{1}{M_S}\mathbf{T}_S$$

There is a parameter in Boris, called *ts_eff* (see **params**): This is a unitless constant which multiplies $\mathbf{T}_S$ and is termed the spin torque efficiency, allowing bulk spin torques to be turned off (*ts_eff* = 0) or fully on (*ts_eff* = 1).

There are two possibilities for treating composite media boundaries. The simplest approach is to assume continuity of a flux and potential – for the spin transport solver these are $\mathbf{J}_C$ and V for charge transport and $\mathbf{J}_S$ and **S** for spin transport. The continuity conditions are used when modelling interfaces between two normal metals (N) or two ferromagnets (F); they may also be used to model interfaces between a normal metal and ferromagnet (N/F) but in this case typically the second approach is more appropriate, based on interface spin conductances:

$$\mathbf{J}_C.\mathbf{n}|_N = \mathbf{J}_C.\mathbf{n}|_F = -\left(G^{\uparrow} + G^{\downarrow}\right)\Delta V + \left(G^{\uparrow} - G^{\downarrow}\right)\Delta \mathbf{V}_S.\mathbf{m}$$
$$\mathbf{J}_S.\mathbf{n}|_N - \mathbf{J}_S.\mathbf{n}|_F = \frac{2\mu_B}{e}\left[\mathrm{Re}\left\{G^{\uparrow\downarrow}\right\}\mathbf{m}\times\left(\mathbf{m}\times\Delta\mathbf{V}_S\right) + \mathrm{Im}\left\{G^{\uparrow\downarrow}\right\}\mathbf{m}\times\Delta\mathbf{V}_S\right]$$
$$\mathbf{J}_S.\mathbf{n}|_F = \frac{\mu_B}{e}\left[-\left(G^{\uparrow} + G^{\downarrow}\right)\left(\Delta\mathbf{V}_S.\mathbf{m}\right)\mathbf{m} + \left(G^{\uparrow} - G^{\downarrow}\right)\Delta V\mathbf{m}\right]$$

In the above equations $G^{\uparrow}$, $G^{\downarrow}$ are interface conductances for the majority and minority spin carriers respectively, and $G^{\uparrow\downarrow}$ is the complex spin mixing conductance. Also ΔV is the potential drop across the N/F interface ($\Delta V = V_F - V_N$) and $\Delta\mathbf{V}_S$ is the spin chemical potential drop, where $\mathbf{V}_S = \left(D_e/\sigma\right)\left(e/\mu_B\right)\mathbf{S}$. These interface conditions describe the absorption of transverse spin components at the interface, giving rise to interfacial spin torques:

$$\mathbf{T}_S^{interface} = \frac{g\mu_B}{ed_h}\left[\mathrm{Re}\left\{G^{\uparrow\downarrow}\right\}\mathbf{m}\times\left(\mathbf{m}\times\Delta\mathbf{V}_S\right) + \mathrm{Im}\left\{G^{\uparrow\downarrow}\right\}\mathbf{m}\times\Delta\mathbf{V}_S\right]$$

There is also an associated interfacial spin torque efficiency constant – *tsi_eff*. The above term is also included in the magnetization dynamics equation, much in the same way as $\mathbf{T}_S$ is. The main difference is this torque is only included in the computational cells at the interface, where d_h is the cellsize normal to the interface – this allows correct computation of interfacial spin torques for a ferromagnetic layer of given thickness *t*, independent of its computational discretization, since the effect on magnetization of the interfacial spin torque is averaged over its thickness.

Spin pumping is generated at an N/F interface as:

$$\mathbf{J}_S^{pump} = \frac{\mu_B}{2\pi}\left[\mathrm{Re}\left\{g^{\uparrow\downarrow}\right\}\mathbf{m}\times\frac{\partial\mathbf{m}}{\partial t} + \mathrm{Im}\left\{g^{\uparrow\downarrow}\right\}\frac{\partial\mathbf{m}}{\partial t}\right]$$

Here $g^{\uparrow\downarrow} = (h/e^2)G^{\uparrow\downarrow}$ and the pumped spin current is used in the calculation of composite media boundary conditions by including it on the normal metal side of the equations. As with the spin torques, there's an associated spin pumping efficiency parameter – *pump_eff* – which allows spin pumping to be turned on or off in the computations.

When modelling N/F interfaces you may need to have different interface conductances on different sides of a ferromagnetic layer (e.g. a Pt/Co/Ta multilayer, where the Pt/Co and Co/Ta interfaces may need different spin mixing conductances). For this reason, the interface conductances ($G^{\uparrow}$, $G^{\downarrow}$, and $G^{\uparrow\downarrow}$) are not associated just with a ferromagnetic mesh, but also appear in the list of parameters for normal metal meshes (*conductor* meshes - **addconductor**). When an N/F interface is defined by the contact of two meshes, the interface conductances stored in the *upper* mesh are used – e.g. if the meshes are arranged in a multilayer structure along the z direction, the upper mesh is that with a higher z coordinate. To turn off the interface conductance approach to modelling composite media boundaries you need to set the $G^{\uparrow\downarrow}$ values to zero for the appropriate mesh. In this case the computations revert to using the continuity approach described above.

To enable the spin transport solver you need to have the *transport* module active in the mesh you want spin transport computations and you must also select a magnetization dynamics equation with spin accumulation (e.g. LLG-SA, LLB-SA, etc.) – see the **ode** command. Using the **display** command you can select to display a number of associated quantities in the mesh viewer, including **S**, bulk and interfacial spin torques, x, y, and z directions for the spin current.

With the spin transport solver enabled you will need to pay attention to the convergence constant for the spin accumulation solver. Similarly to the charge potential solver, which solves a Poisson equation to obtain V within the set convergence constant, the spin accumulation **S** is obtained by solving a vector Poisson equation – this equation is obtained from the equation of motion for **S** in the "steady state", i.e. when $\partial\mathbf{S}/\partial t = 0$. The response time-scales of **m** and **S** are separated typically by 3 orders of magnitude (ps vs fs time-scales respectively) thus we only require to obtain the "steady state" values for **S** for a given magnetization configuration. The vector Poisson equation also uses a convergence constant and a timeout for the maximum number of allowed sequential iterations, and these values may be changed by using the **tsolverconfig** command.

Further info:

Whilst each iteration taken for the Poisson equations for V and **S** is relatively cheap, typical problems may require a large number of iterations to reach convergence, which significantly slows down computations. This is especially true in the initialization stage when the timeout number of iterations may be reached for the first few iterations; after this, small steps in **m** should result in relatively few steps in the solution of **S** (and where appropriate V). A recommended general approach is to solve for the steady state V and **S** values with all spin torques turned off and for a relaxed starting magnetization configuration. After this, save the simulation (which also saves the computed V and **S**), and re-enable the spin torques as required. From this point initialization should be quicker, with any further iterations in V and **S** triggered by changes in **m** (changing set electrode potential values can also trigger the Poisson solvers).

Setting the convergence factors too low may result in very slow simulations as the solvers will require a large number of iterations. You will need to determine the best compromise between computational speed and accuracy. The default normalized convergence values of 10^{-6} for V and 10^{-5} for **S** Poisson equations are set on the side of accuracy, having been found to give accurate results in all test cases, but you should still verify this for your particular simulation.

Tutorial 18 – Spin Hall Effect

Exercise 18.1

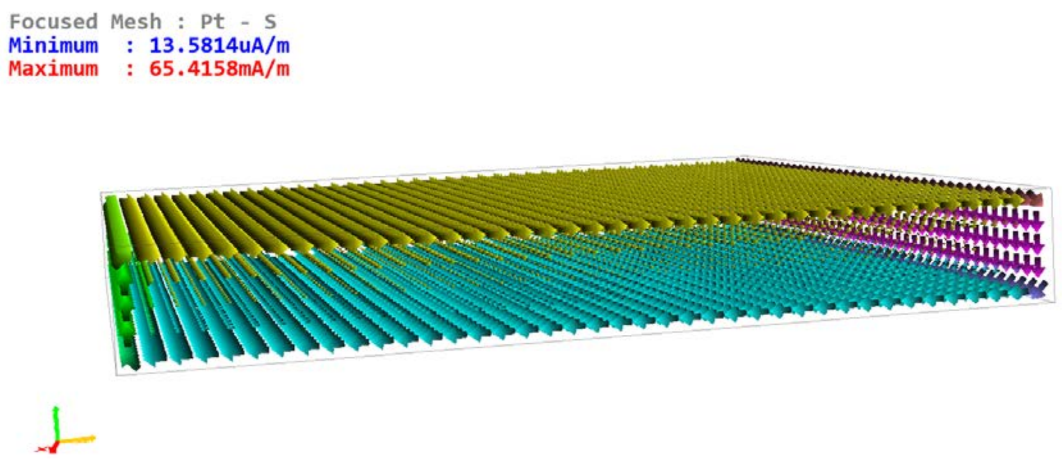
Consider a single Pt mesh with dimensions 320 nm x 320 nm and 40 nm thickness. Compute the spin accumulation and z-direction spin current density in response to a set potential of 10 mV with electrodes placed at the x-axis ends. Verify that **S** obeys the right-hand-rule with respect to the charge current direction.

For Pt you may use $\sigma = 7\times10^6$ S/m, $\lambda_{sf} = 1.4$ nm and $\theta_{SHA} = 0.1$. You may use a cubic cellsize with 5 nm side.

Plot the y components of the z-direction spin current density and spin accumulation along the z-axis, through the center of the Pt slab (remember the **dp_getprofile** command). Verify that the following relation holds, using the plotted value of the spin current at the center of the Pt slab:

$$\theta_{SHA} = -\frac{J_{sz,y}}{J_{cx}} / \frac{\mu_B}{e}$$

Figure 18.1 – Computed spin accumulation for Exercise 18.1, where the charge current density is along the negative x direction.



```
Focused Mesh : Pt - S
Minimum  : 13.5814uA/m
Maximum  : 65.4158mA/m
```

Exercise 18.2

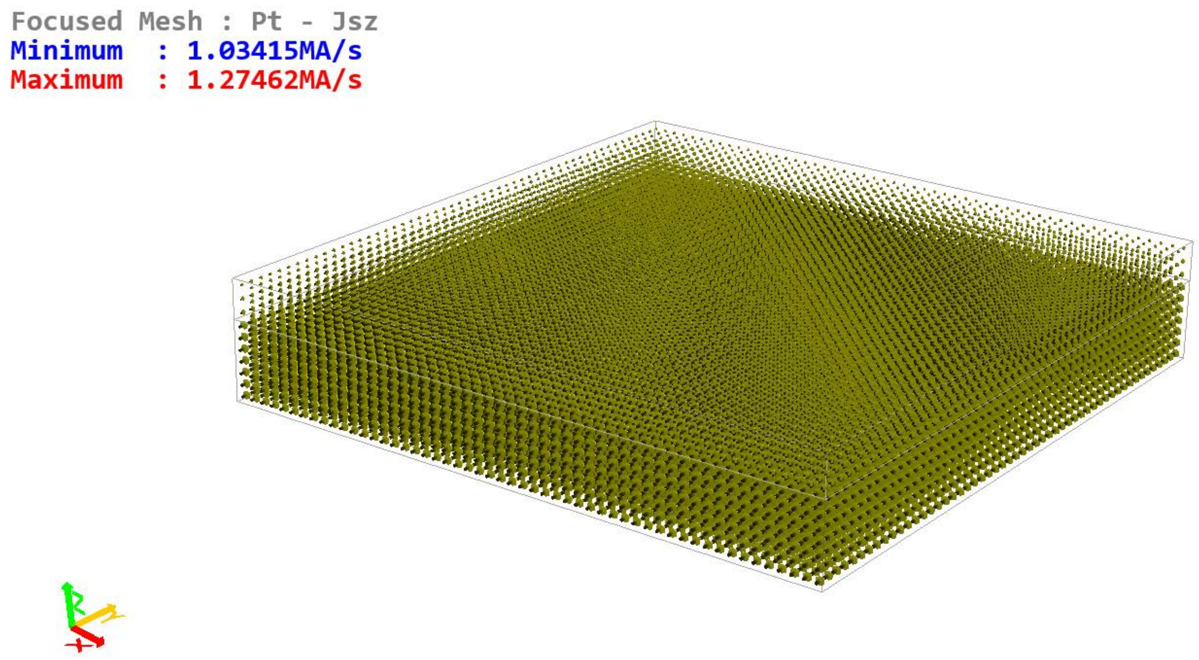
a) Continuing from Exercise 18.1, now add a $Ni_{80}Fe_{20}$ layer with 20 nm thickness on top of the Pt layer. Make sure to reset to default electrodes (**setdefaultelectrodes**) so a uniform charge current density is obtained in each layer. Plot the y components of the z-direction spin current density along the z axis for both the continuous and spin-mixing conductance interface models for a) magnetization direction along the injected spin current, i.e. along the y axis, and b) magnetization direction transverse to the injected spin current, i.e. along the x axis. Explain the differences between these cases.

Note, for this exercise you will have to use a smaller cellsize along the z direction. This is due to the large gradients involved, and is normally the case when N / F multilayers are used. You should use a cellsize of (5 nm, 5 nm, 1 nm).

Remember you can use the **computefields** command for this exercise, instead of using **run**, since you don't want to relax the magnetization configuration. It helps to monitor the transport solver number of iterations and convergence error (in the data box display the following using the **data** command and righ-clicking on the respective interactive objects: *v_iter*, *s_iter*, *ts_err*).

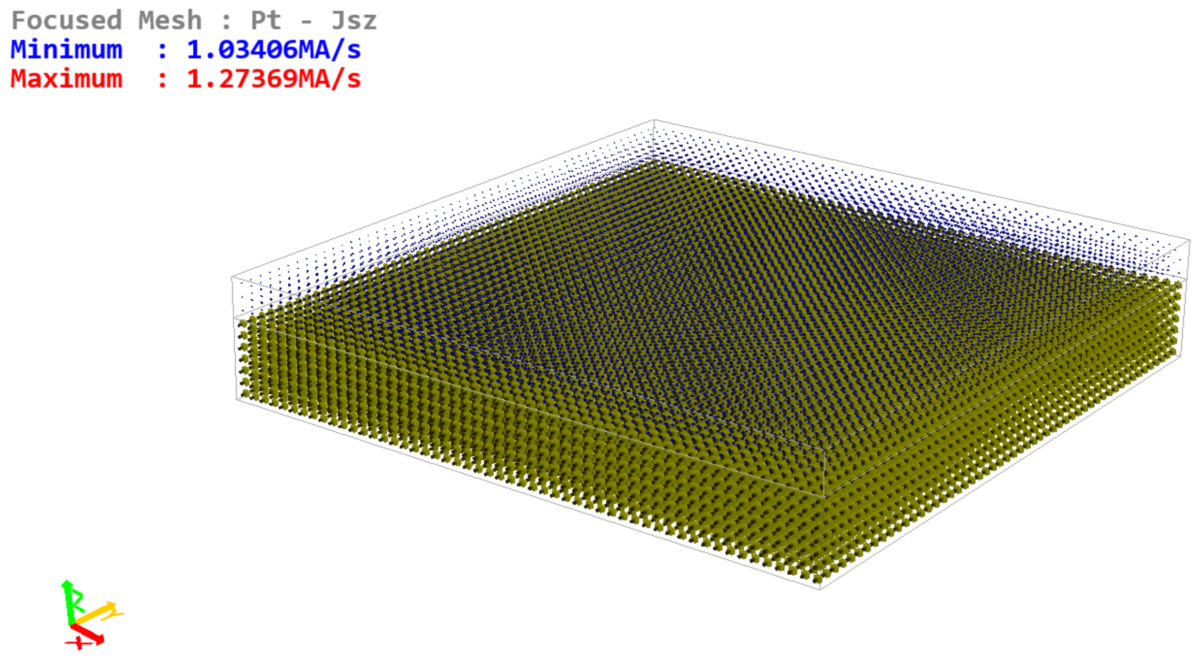
b) Does the relation in Exercise 18.1 hold at the N/F interface, and why not? Investigate this again with a spin flip length in Pt of 8 nm, checking the relation both at the center of the Pt layer and at the interface.

Figure 18.2 – Spin current density in the z direction for a Pt/$Ni_{80}Fe_{20}$ bilayer, where the magnetization in the permalloy mesh is along the y axis (longitudinal).



```
Focused Mesh : Pt - Jsz
Minimum  : 1.03415MA/s
Maximum  : 1.27462MA/s
```

Figure 18.3 – Spin current density in the z direction for a Pt/$Ni_{80}Fe_{20}$ bilayer, where the magnetization in the permalloy mesh is along the x axis (transverse).



```
Focused Mesh : Pt - Jsz
Minimum  : 1.03406MA/s
Maximum  : 1.27369MA/s
```

Tutorial 19 – Spin Pumping and Inverse Spin Hall Effect

In this tutorial you will set-up a ferromagnetic resonance in a magnetic dot, then investigate the generated spin Hall voltage in a Pt underlayer. Due to the motion of magnetic moments in the ferromagnetic layer a spin current is pumped in the Pt underlayer, where an electrical current is generated due to the inverse SHE. This leads to charge accumulation at opposing sides of the Pt underlayer, and thus an electrical potential is generated.

Exercise 19.1

Setup a ferromagnetic resonance (FMR) at 20 GHz excitation frequency in a $Ni_{80}Fe_{20}$ circle with 80 nm diameter and 10 nm thickness, with a bias field applied in the plane of the circle along the y axis.

First, find the demagnetizing factor in the plane of the circle and set the *demag_N* module with demagnetizing factors $N_x = N_y = N$. Remember you can calculate the demagnetizing energy for uniform magnetization (*e_demag*), and the demagnetizing factor is then related to it by:

$$\varepsilon_{demag} = \frac{\mu_0}{2} N M_S^2$$

For this exercise, since you are effectively using the Stoner-Wohlfarth model you can turn off the *exchange* module. Calculate the FMR bias field required for resonance at an r.f. frequency of 20 GHz. You can use Kittel's formula applicable for elliptical shapes for a bias field H_0 along the y direction:

$$f = \frac{\mu_0 |\gamma_e|}{2\pi} \sqrt{(H_0 + (N_x - N_y)M_s)(H_0 + (N_z - N_y)M_s)}$$

Using the *Hfmr* stage type, apply the excitation r.f. field (together with the calculated orthogonal bias field) in the plane of the circle for a number of cycles, and record the average magnetization. An r.f. field amplitude of 100 A/m is normally sufficient. If the r.f. field is applied for a sufficient number of cycles, the magnetization will achieve a steady state precession at resonance. Determine the number of cycles required by examining the output average magnetization data, then **reset** and save the simulation – the next time you load the simulation the FMR precession will start directly in the steady state.

The *Hfmr* stage consists of the following parameters: H_{0x}, H_{0y}, H_{0z}; H_{rfx}, H_{rfy}, H_{rfz}; *r.f. steps; r.f. cycles*.

The bias field (H_0) and r.f. field amplitude (H_{rf}) are specified using Cartesian coordinates. The *r.f. steps* is the number of discretisation steps in each r.f. cycle, and the *r.f. cycles* is the number of sinusoidal oscillations the r.f. field will be applied for. To set the required 20 GHz frequency you will need to set the correct combination of *r.f. steps* and time stopping condition for each step. For example, since at 20 GHz each period takes 50 ps, if you use 20 r.f. steps per cycle, the time stopping condition for each step should be 2.5 ps (the default time stopping condition of 50 ps results in a 1 GHz frequency with 20 r.f. steps per cycle, so you will need to edit this).

Exercise 19.2

Using the prepared simulation from Exercise 19.1, add a Pt underlayer with dimensions 160 nm × 160 nm with the magnetic dot centered, and 20 nm depth.
Enable spin pumping (*pump_eff* = 1 in the permalloy mesh), and inverse SHE (*SHA* = *iSHA* = 0.1) in the Pt mesh. Do not set any electrodes but make sure the transport module is enabled in both the permalloy and Pt meshes, and the LLG-SA equation is selected so the spin transport solver is enabled. You will need to refine the electric cellsize in both meshes along the z direction to 1 nm. You should also relax the transport solver convergence criteria to 10^{-4} for both the charge and spin solvers.

Obtain the induced spin Hall voltage at the opposing y-axis sides of the Pt mesh and plot them as a function of time for a few FMR precessions. Note, in the output data (**data**) you will have to add <*V*> (the average calculated voltage) for the Pt mesh two times, editing the respective rectangles to correspond to the required two sides of the Pt mesh.

Figure 19.1 – Inverse spin-Hall effect voltage in a Pt underlayer generated through spin pumping from a ferromagnetic dot at ferromagnetic resonance.

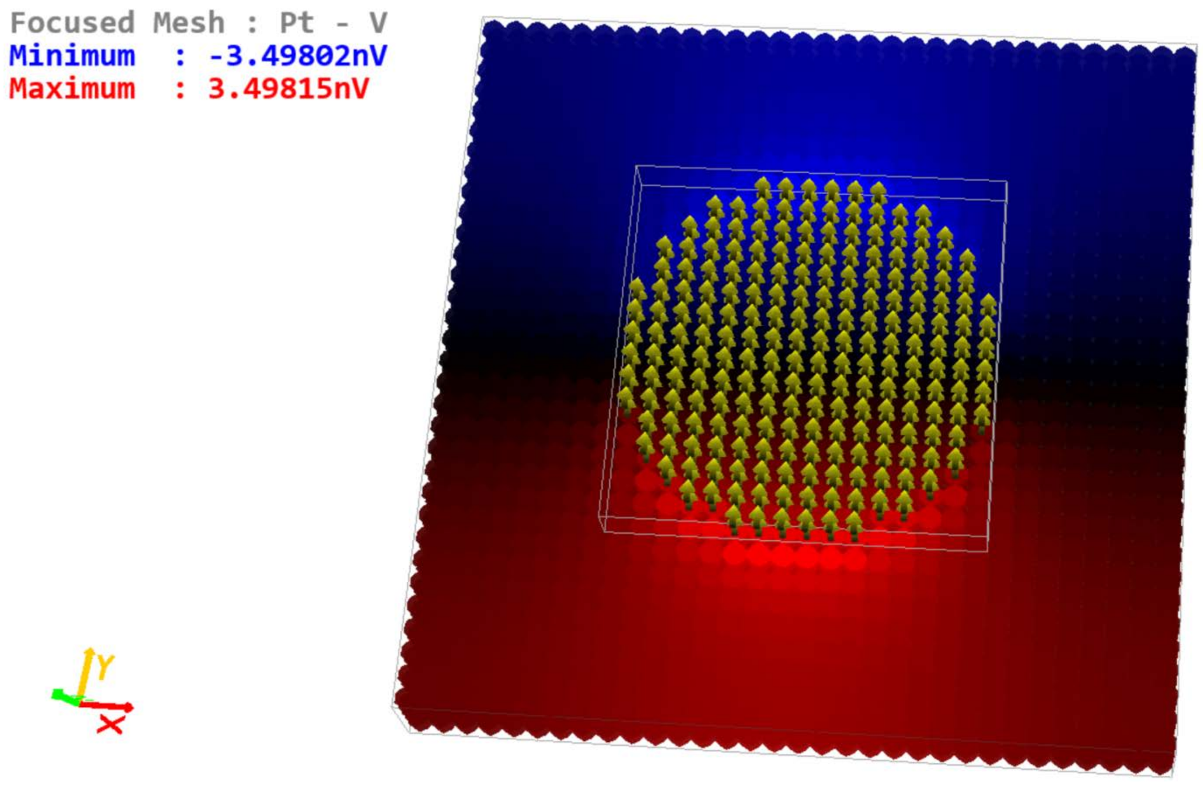
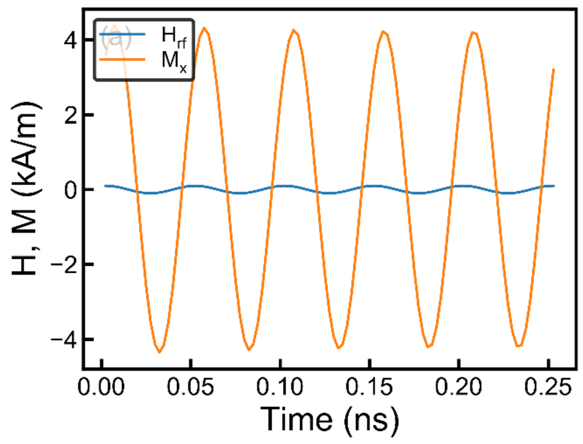
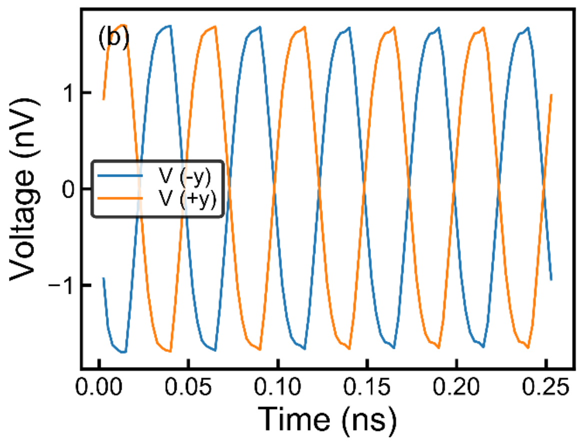


Figure 19.2 – a) Magnetization precession at ferromagnetic resonance with a 20 GHz r.f. field, b) inverse spin-Hall effect voltage at resonance on opposing sides of the Pt mesh – see Figure 19.1.

a)



b)



Tutorial 20 – Ferromagnetic Resonance

In this tutorial you will learn how to simulate ferromagnetic resonance (FMR), and re-produce input material parameters. Following this, ino the next tutorial you will investigate how the spin torques due to spin pumping and the SHE affect the effective damping observed. An introduction to FMR simulations was given in the preceding Tutorial, and you must complete it before proceeding. Here we will investigate both field-swept FMR (fixed excitation frequency) and frequency-swept FMR (fixed bias field).

Field-Swept FMR

In Boris, FMR simulations are best done using a Python script. As you will note from the previous tutorial, applying an r.f. field excitation requires a number of cycles for the magnetization precession to reach steady state. For a field-swept FMR simulation, after changing the bias field you must ensure the magnetization precession is stable before obtaining output data. The simulation procedure is as follows:

1) Set bias field value and run the simulation for a fixed number of r.f. cycles (the "chunk" – e.g. 20 or more), but do not save any output data.
2) After the chunk has completed, run the simulation for a single r.f. cycle and save the output data (<*M*>).
3) From the saved data obtain the magnetization oscillation amplitude along the r.f. field direction.
4) Compare the oscillation amplitude against the previous oscillation amplitude (which is zero if this is the first chunk). If the change exceeds a set threshold (e.g. 0.1%) then repeat from step 1), otherwise proceed.
5) Record the oscillation amplitude and bias field. Increase bias field value and start again from step 1) until the field sweep range is completed.

A general-purpose FMR simulation Python script has been prepared and saved in the examples folder for this Tutorial.

Exercise 20.1

Using a $Ni_{80}Fe_{20}$ square of 80 nm side and 10 nm thickness simulate an FMR peak around the resonance bias field and plot the resulting magnetization oscillation amplitude against bias field data. Set the bias field along the –y direction, i.e. at 270° azimuthal angle. Use a Python script to simulate this as described above. (*You may use $N_x = N_y = 0.12$, with the predicted resonance field of $H_0 \simeq 367$ kA/m; aim for at least 50 kA/m either side of resonance*).

From the simulated oscillation amplitude versus bias field data, you will need to obtain a quantity proportional to the absorbed FMR power. The simplest way to do this is to square the oscillation amplitude data. The FMR power absorption peak is described by a Lorentz peak function, and you will need to fit this to your squared amplitude data.

Boris has built-in data processing command to help with processing FMR simulation data. You will need the following commands:

First load bias field and oscillation amplitude from the raw output data file (e.g. named ‘fmr_fieldsweepFMR_data.*txt*’):

dp_load *fmr_fieldsweepFMR_data 0 1 0 1*

Next square the magnetization oscillation amplitude data:

dp_muldp *1 1 1*

Finally fit a Lorentz peak function to the data:

dp_fitlorentz *0 1*

The Lorentz peak function is given as:

$$f(x) = y_0 + S\frac{w}{4(x - x_0)^2 + w^2}$$

In the above equation *w* is the full-width half-maximum (FWHM), and x_0 is the peak center. You can obtain these values from the **dp_fitlorentz** command, including fitting uncertainties (Boris has a built-in generic Levenberg-Marquardt algorithm for curve fitting).

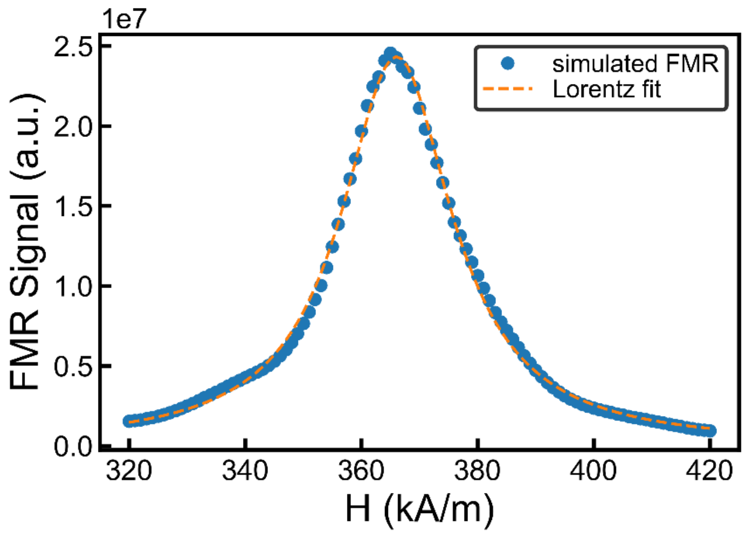
The magnetization damping value is related to the full-width half-maximum (ΔH) by:

$$\alpha = \frac{\mu_0 |\gamma_e|}{4\pi} \frac{\Delta H}{f}$$

Exercise 20.1 continued

Process the output FMR data and verify the damping obtained from the FWHM matches the set damping value (*damping* = 0.02).

Figure 20.1 – Simulated FMR peak with Lorentz peak function fit for Exercise 20.1.



Frequency-Swept FMR

Frequency-swept FMR simulations are also possible and are typically much more efficient to compute than field-swept FMR. This type of simulation is closely related to the method used to investigate spin-wave dispersion. Here we apply a sinc pulse in the time domain and capture the magnetization response, which we then transform to the frequency domain using a Fourier transform. The reason for using a sinc pulse is its Fourier transform is a symmetric hat function with a defined frequency cut-off. Thus in order to capture the required resonance (and also any required higher resonance modes) we simply need to set the cut-off to a large enough value.

The sinc pulse excitation is given as:

$$H(t) = H_e \sin\left(2\pi f_c \left(t - t_0\right)\right) / \left(2\pi f_c \left(t - t_0\right)\right)$$

Here f_c (Hz) is the cut-off frequency, H_e is the excitation amplitude, with $H(t)$ taking on the value H_e at $t = t_0$, the sinc pulse centre. We also need a fixed bias field, H_0, which must be orthogonal to the excitation field, and in general we must capture the average magnetization (for now we'll assume the magnetization is uniform) and use the magnetization component along the excitation field. We need to capture the magnetization at a time step set by the Nyquist criterion:

$$t_S = \frac{1}{2 f_C} \quad (s)$$

Capturing magnetization data at other time steps is sub-optimal and will result in lost information if larger than the above formula, or increased noise in the Fourier transform spectrum if smaller than the above formula. It is recommended the sinc pulse is simulated for a total time of $2t_0$. If you want to increase the spectrum resolution (number of points) then you must increase the t_0 value, but still simulate for a total time of $2t_0$, saving data at a time step of t_s.

We can set this type of excitation using a text equation stage, namely *Hequation*. To set the above equation, edit the stage value for the added *Hequation* stage to: *He*sinc(2*PI*fc*(t-t0)), H0, 0*. This will set the excitation field along the x axis, and bias field along the y axis, with zero z axis field. A dedicated chapter in the manual is given for details on using text equations. In the above equation *t* is a reserved parameter, namely the stage time, and the equation provided is evaluated internally every iteration. The remaining parameters, *He, fc, t0*, *H0* must be defined in order for the equation evaluation to function. Equation constants may be given as:

equationconstants *name value,*

e.g. **equationconstants** fc 200e9.

Other related commands are **delequationconstant**, and **clearequationconstants**.

For frequency-swept FMR after taking the Fourier transform, similarly to field-swept FMR, the output data must be squared. The resulting FMR peak in the frequency domain is also described by a Lorentz peak, where the FWHM, Δf, is related to the resonance frequency f_0, and Gilbert damping as:

$$\alpha = \frac{\Delta f}{2 f_0}$$

By capturing a set of frequency-domain FMR peaks as a function of bias field H_0 the following Kittel formula can be verified:

$$f = \frac{\mu_0 |\gamma_e|}{2\pi} \sqrt{\left(H_0 + \left(N_y - N_z + k\right) M_S\right)\left(H_0 + \left(N_x - N_z + k\right) M_S\right)}$$

Here N_x, N_y, and N_z are demagnetizing factors such that the bias field is applied along the z direction. The above formula also includes uniaxial anisotropy, with easy axis along the z direction, where:

$$k = \frac{2K_1}{\mu_0 M_S^2}$$

For the following exercise you'll simulate a thin film Co material interfaced with Pt (this is stored in the materials database and can be loaded as **setmaterial** *Co/Pt* – more on the materials database in a dedicated chapter. You will also need to simulate a thin film, thus must set periodic boundary conditions in the xy plane as: **pbc** x 10, **pbc** y 10 – more on periodic boundary conditions in a dedicated tutorial.

Exercise 20.2 (Advanced)

Simulate the frequency-swept FMR response of a thin-film Co/Pt material with 2 nm thickness using the method described above.

You can use a cut-off frequency of 200 GHz, with excitation field amplitude of 1000 A/m, and vary the bias field between 100 kA/m and 1 MA/m. Simulate both out-of-plane FMR (anisotropy easy axis and bias field out of plane), and in-plane FMR (anisotropy easy axis and bias field in the plane, e.g. both along the y direction).

In both cases use the Kittel relation and damping formula to verify the input simulation parameters using fitting procedures (damping and magneto-crystalline anisotropy).

Tutorial 21 – Ferromagnetic Resonance with Spin Torques

Continuing from the previous tutorial, you will now investigate the effect of spin torques due to the spin-Hall effect on the magnetization damping using a Pt/$Ni_{80}Fe_{20}$ bilayer. The spin current generated in the Pt underlayer is absorbed by the $Ni_{80}Fe_{20}$ layer, resulting in a combination of damping-like and field-like torques. Depending on the current direction a decrease or increase of the effective damping is obtained. First the full spin transport solver is used, and following this a simpler method using an analytical form for the spin-orbit torques is introduced. Using the full spin transport solver we can also consider the effect of spin pumping on the effective damping, and this is investigated at the end of this tutorial.

The interfacial spin orbit torque added to the implicit LLG equation, as explained in Tutorial 17, is given by:

$$\mathbf{T}_S^{interface} = \frac{g\mu_B}{ed_h}\left[\mathrm{Re}\left\{G^{\uparrow\downarrow}\right\}\mathbf{m}\times\left(\mathbf{m}\times\Delta\mathbf{V}_S\right) + \mathrm{Im}\left\{G^{\uparrow\downarrow}\right\}\mathbf{m}\times\Delta\mathbf{V}_S\right]$$

For an N/F interface in the x-y plane with uniform current densities we can obtain an analytical expression for this interfacial torque as (see S. Lepadatu, Scientific Reports 7, 12937 (2017)):

$$\mathbf{T}_S^{interface} = \theta_{SHA,eff}\,\frac{\mu_B}{e}\frac{\left|J_c\right|}{d_h}\left[\mathbf{m}\times\left(\mathbf{m}\times\mathbf{p}\right) + r_G\mathbf{m}\times\mathbf{p}\right]$$

Here **p** = **z** × $\mathbf{e}_{\mathbf{Jc}}$, where $\mathbf{e}_{\mathbf{Jc}}$ is the charge current direction, and:

$$\theta_{SHAeff} = \theta_{SHA}\left(1 - \frac{1}{cosh(d_N/\lambda_{sf}^N)}\right)\frac{\mathrm{Re}\left\{\tilde{G}\right\}^2 - \mathrm{Im}\left\{\tilde{G}\right\}^2 + N_\lambda Re\left\{\tilde{G}\right\}}{\left(N_\lambda + Re\left\{\tilde{G}\right\}\right)^2 + Im\left\{\tilde{G}\right\}^2},$$

$$r_G = \frac{N_\lambda\,\mathrm{Im}\{\tilde{G}\} + 2\,\mathrm{Re}\{\tilde{G}\}\,\mathrm{Im}\{\tilde{G}\}}{N_\lambda\,\mathrm{Re}\{\tilde{G}\} + \mathrm{Re}\{\tilde{G}\}^2 - \mathrm{Im}\{\tilde{G}\}^2}$$

In the above, $N_\lambda = tanh\left(d_N / \lambda_{sf}^N\right) / \lambda_{sf}^N$, $F_\lambda = tanh\left(d_F / \lambda_{sf}^F\right) / \lambda_{sf}^F$, and $\tilde{G} = 2G^{\uparrow\downarrow} / \sigma_N$. Thus the interfacial torque has a damping-like and a field-like component. In the limit of abrupt interface ($\lambda_\varphi \rightarrow 0$ or equivalently Re{$G^{\uparrow\downarrow}$} $\rightarrow \infty$) the field-like component tends to zero and we obtain the following expression for the torque:

$$\mathbf{T}_{SOT} = \theta_{SHA,eff} \frac{\mu_B}{e} \frac{|J_c|}{d_h} \mathbf{m} \times (\mathbf{m} \times \mathbf{p})$$

and

$$\theta_{SHA,eff} = \theta_{SHA} \left(1 - \frac{1}{cosh(d_N / \lambda_{sf}^N)} \right)$$

This approximation can also be used when the damping-like torque is much larger than the field-like torque. This expression is commonly used in the literature to model the spin-orbit torque resulting from the spin-Hall effect. Note however the spin-Hall angle in this expression is not the bulk (or intrinsic) spin Hall angle, but an effective spin Hall angle, scaled by transport parameters; if further the N layer thickness is many times larger than its spin flip length, we can use the approximation $\theta_{SHA,eff} \cong \theta_{SHA}$. In many cases this may not be true, and moreover the abrupt interface approximation may not be good either, thus to model the effect of the damping-like torque with the analytical form of the spin-orbit torque, in the expression for $\mathbf{T}_{SOT}$ you should use the full expression for the effective spin-Hall angle given above.

In Boris you can include this analytical spin-orbit torque using the *SOTField* module. Enabling this module in a ferromagnetic mesh introduces an additional effective field into the LLG equation which results in the $\mathbf{T}_{SOT}$ torque given above (as it appears in the implicit LLG equation). To use it you still need to have the *transport* module enabled in order to calculate the charge current density, but instead of selecting LLG-SA (enabling the full spin transport solver) you should select just the LLG equation (**ode** command). In the material parameters for the ferromagnetic mesh (**params** command) you need to enter the correct effective spin-Hall angle (*SHA*) to use with the *SOTField* module.

Exercise 21.1

Using the $Ni_{80}Fe_{20}$ layer from the previous tutorial, now add a Pt underlayer with the same dimensions (80 nm × 80 nm × 10 nm), using the Pt parameters from Tutorial 18. Set default electrodes (resulting in current flow along the x direction), enabling the spin transport solver both in the $Ni_{80}Fe_{20}$ and Pt meshes (add *transport* modules and set the **ode** solver to LLG-SA). Make sure to disable spin pumping (*pump_eff* = 0) and the inverse SHE (*iSHA* = 0). You should also disable bulk spin torques (*ts_eff* = 0), only leaving interfacial spin torques enabled (*tsi_eff* = 1). As before you may need to decrease the z-direction electrical cellsize to ensure accuracy (and numerical convergence!).

Obtain FMR peaks for charge current densities in the Pt layer of Jc = $\pm 10^{12}$ A/m^2. How does the damping change with current density direction?

In this case, even though you are using the Stoner-Wohlfarth model (*demag_N* module), you should still enabled the *exchange* module. The reason for this, the spin torques may not be perfectly uniform (e.g. if the permalloy and Pt layers have the same width, the spin torques will not be uniform since the spin accumulation has gradients at the sample edges), thus you do need to take the exchange interaction into consideration.

The change in damping due to a damping-like spin-orbit torque may be roughly approximated by:

$$\Delta\alpha_{SHE} \cong \theta_{SHA,eff} \frac{\mu_B}{e} \frac{J_c}{2\pi f M_S d_F}$$

Verify the change in damping obtained from simulations with the above formula.

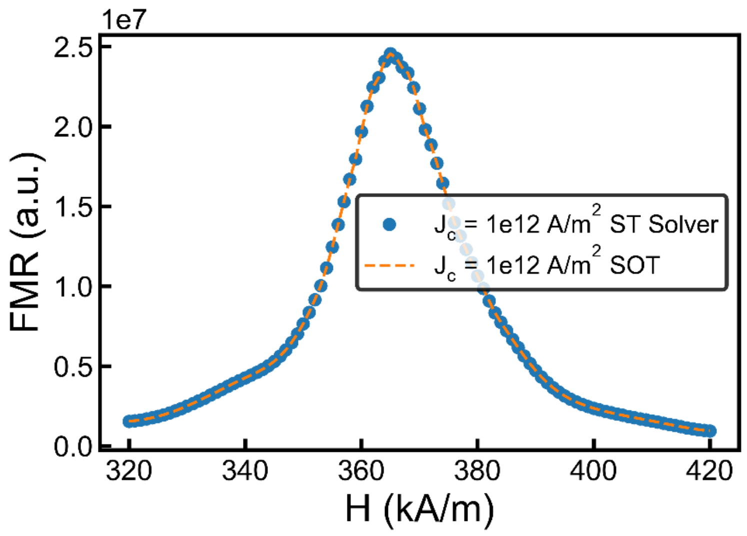
Exercise 21.2

Repeat Exercise 21.1 but this time without the spin transport solver, only using the analytical form for the spin-orbit torque (*SOTField* module). You should delete the Pt mesh and reset the electrodes and potential to give you the correct current density. Calculate an appropriate effective spin-Hall angle to use. Compare the results with the previous exercise.

Exercise 21.3

Repeat Exercise 21.1, using the full spin-transport solver, but now enable spin pumping (set *pump_eff* = 1). What is the increase in damping?

Figure 21.1 – FMR simulations with spin orbit torques for both the full spin transport solver (ST Solver) and effective field obtained from the analytical spin-orbit torque (SOTField).

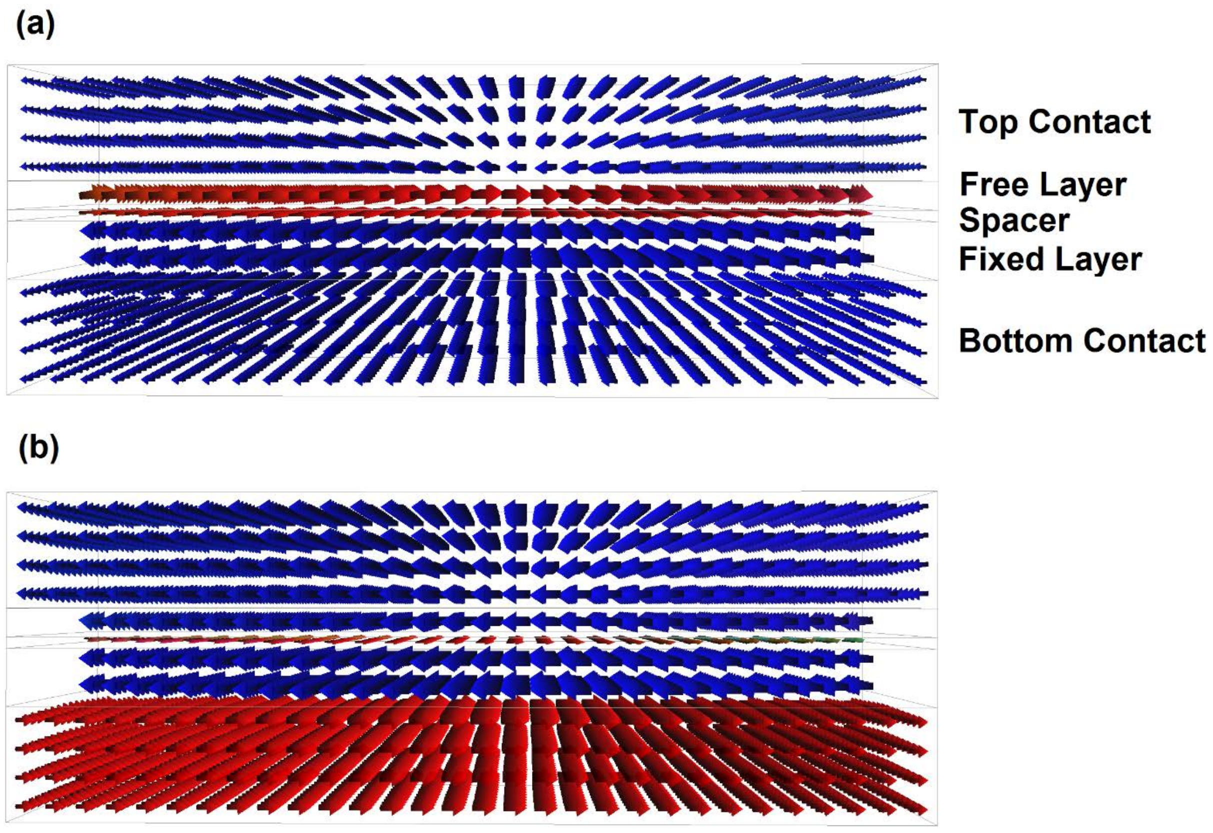


Tutorial 22 – CPP-GMR

The spin transport solver is also able to reproduce the spin torques in a current-perpendicular to plane (CPP) giant magneto-resistance (GMR) spin valve, in addition to its magneto-resistance. Here we will investigate the current-induced switching in a simple generic spin valve between the parallel and anti-parallel states, see Figure 22.1, and plot the resistance during these switching events.

A spin valve, in its simplest form, consists of a fixed magnetic layer, a free magnetic layer which can be switched between an anti-parallel and parallel orientation with respect to the fixed layer, and a thin metallic spacer layer. The spacer layer thickness can be adjusted to give either a ferromagnetic or anti-ferromagnetic surface exchange coupling between the two magnetic layers. In the following simulation we will also add two metallic contacts, top and bottom.

Figure 22.1 – CPP-GMR spin valve showing the spin accumulation in the spacer layer, top, and bottom contacts, and the magnetization in the elliptically shaped fixed and free layers for a) anti-parallel state, and b) parallel state.



Exercise 22.1

Setup a generic spin valve structure (i.e. just use the default mesh parameters unless indicated otherwise) similar to that shown in Figure 22.1. This consists of:

- Bottom and top contacts (**addconductor**) with dimensions 160 nm × 80 nm × 20 nm. Disable spin-Hall effects in both (*SHA* = *iSHA* = 0).
- Spacer layer with dimensions 160 nm × 80 nm × 2 nm and set it to an elliptical shape (drag a .png file with a circle shape to the mesh viewer when the spacer layer mesh is in focus). Disable spin-Hall effects.
- Fixed layer (**addmesh**) with dimensions 160 nm × 80 nm × 10 nm and elliptical shape. Disable spin torques and spin pumping in this mesh (*ts_eff* = *tsi_eff* = *ts_pump* = 0). You should also disable magnetization dynamics in this mesh so the magnetization is fixed. You can do this by setting the relative gyromagnetic factor to zero (*grel* = 0 in **params**).
- Free layer with dimensions 160 nm × 80 nm × 5 nm and elliptical shape. Disable spin pumping and bulk spin torques only, in this mesh (i.e. keep *tsi_eff* = 1).

You will need to add the following modules:

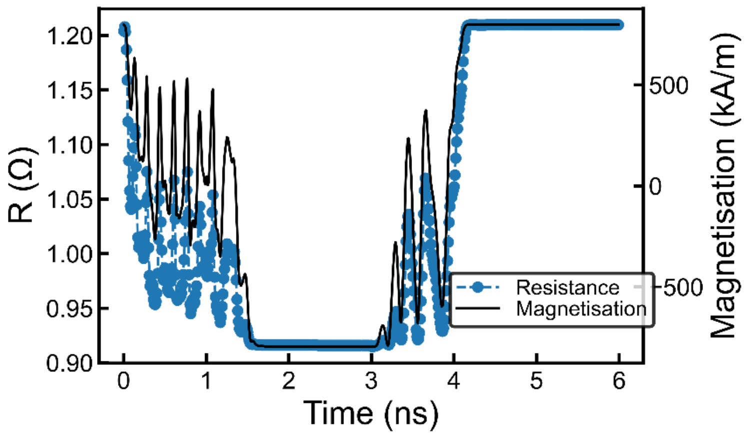
- super-mesh multi-layered demagnetization (*sdemag*).
- *surfexchange* modules in both magnetic meshes. Edit the *J1* (bilinear surface exchange energy density) value in the free layer to give you a weak ferromagnetic coupling; set J1 = 0.1 mJ/m^2.
- *transport* modules in all meshes. Set the electrical cellsize to 5 nm × 5 nm × 1 nm everywhere except in the spacer layer where you should set it to 5 nm × 5 nm × 0.5 nm.

For the **ode** solver you should set the LLG-SA equation (thus enabling the spin-transport solver) with RKF45 evaluation. For output **data** you should have *time*, *R* (resistance), and *<M>* (average magnetization) in the free layer. Set electrodes top and bottom (**addelectrode**), designating the bottom electrode to be the ground electrode (**electrodes**). You need to simulate switching starting from the anti-parallel state (see Figure 22.1) using a +15 mV pulse for 3 ns, then back to this state with a further -15 mV pulse for 3 ns. Save data every 10 ps for both stages. Before starting the simulation you should relax the starting state as follows:

1) Set all spin torques to zero and insert a *Relax* stage with *nostop* condition at the start.
2) First relax the magnetization in the anti-parallel state without the spin-transport solver (set **ode** to LLG).
3) Next enable the spin-transport solver and relax it (run it until the solver no longer iterates, monitoring *v_iter* and *s_iter* **data**).
4) Re-enable the appropriate spin torques (*tsi_eff* = 1 in the free layer only), **reset**, delete the *Relax* stage, then save the simulation (**savesim**).

Explain the resistance change observed by comparing it with the magnetization in the free layer as a function of time – see Figure 22.2 for expected results.

Figure 22.2 – Change in resistance for the CPP-GMR spin valve of Exercise 22.1, together with the magnetization along the longitudinal direction in the free layer.



Tutorial 23 – Skyrmion Movement with Spin Currents

Skyrmions may be displaced using charge and spin currents. To study their movement a skyrmion tracking window can be used in Boris. There are two methods available for tracking a skyrmion, discussed below, both available as data outputs: *skyshift* and *skypos* (use **data** command).

The *skyshift* entry needs a rectangle defined, which should be set around the initial position of a skyrmion, making sure to fully contain it, but don't leave excessive space around it; the thickness of this rectangle should be set to the thickness of the ferromagnetic mesh containing the skyrmion. During a simulation a x-y shift is recorded and saved in the output data file. This shift is determined by comparing the average magnetization magnitude in the 4 quadrants of the skyrmion tracking window – e.g. if the skyrmion shifts to the right, the average magnetization magnitude in the 2 right-hand-side quadrants will decrease compared to the left, thus a right single-cell shift is recorded. Multiple *skyshift* entries can be defined, with different rectangles, to track multiple skyrmions. Note, the *skyshift* entry only works with data file output, and not in the data box or with the **showdata** command.

The raw output *skyshift* data will contain staircase steps due to mesh discretisation. It is possible to obtain a more natural skyrmion movement path by assuming linear displacement in between the staircase steps. To remove the stair steps and replace them using linear interpolation you can use the **dp_replacerepeats** command on both the x and y *skyshift* data columns. The x, y data can then be plotted directly in Cartesian coordinates. This path normally doesn't start from (0, 0). To display the skyrmion displacement path relative to its starting position you should remove this offset using the **dp_removeoffset** command on both the x and y data. Finally, if you want to plot this path using polar coordinates you can use the **dp_cartesiantopolar** command, included in Boris for convenience. Note, especially when converting to polar coordinates you should check the processed data correctly represents the raw data. Problems may occur due to blips in the raw data, especially if the tracking window was not defined well or the starting state is not sufficiently relaxed, thus the results from this procedure must be carefully compared with the raw data.

The second method uses the *skypos* data output. Again this needs an initial rectangle defined around the skyrmion as for the *skyshift* data output. *Skypos* uses a slightly more computationally expensive algorithm to track the skyrmion, but is able to obtain the exact skyrmion center position, as well as skyrmion diameters along the x and y axes, without being affected by staircase discretisation artifacts. It is also able to adjust the tracking window size if the skyrmion diameter changes significantly. The algorithm works by fitting the skyrmion using an analytical function (the same one used for the **dp_fitskyrmion** command) in three steps: 1) Skyrmion is fitted along the x axis through the center of the skyrmion tracking window in order to find the center position. The tracking window x position is adjusted to center it. 2) Skyrmion is fitted along the y axis through the center of the skyrmion tracking window in order to find the center position. The tracking window y position is adjusted to center it. The y diameter is also recorded at this step. 3) The skyrmion is again fitted along the x axis to obtain its x diameter.

For the following exercises you should first use the *skyshift* method of tracking the skyrmion, the repeat them using the *skypos* method. Note, you should not use both methods simultaneously on the same skyrmion.

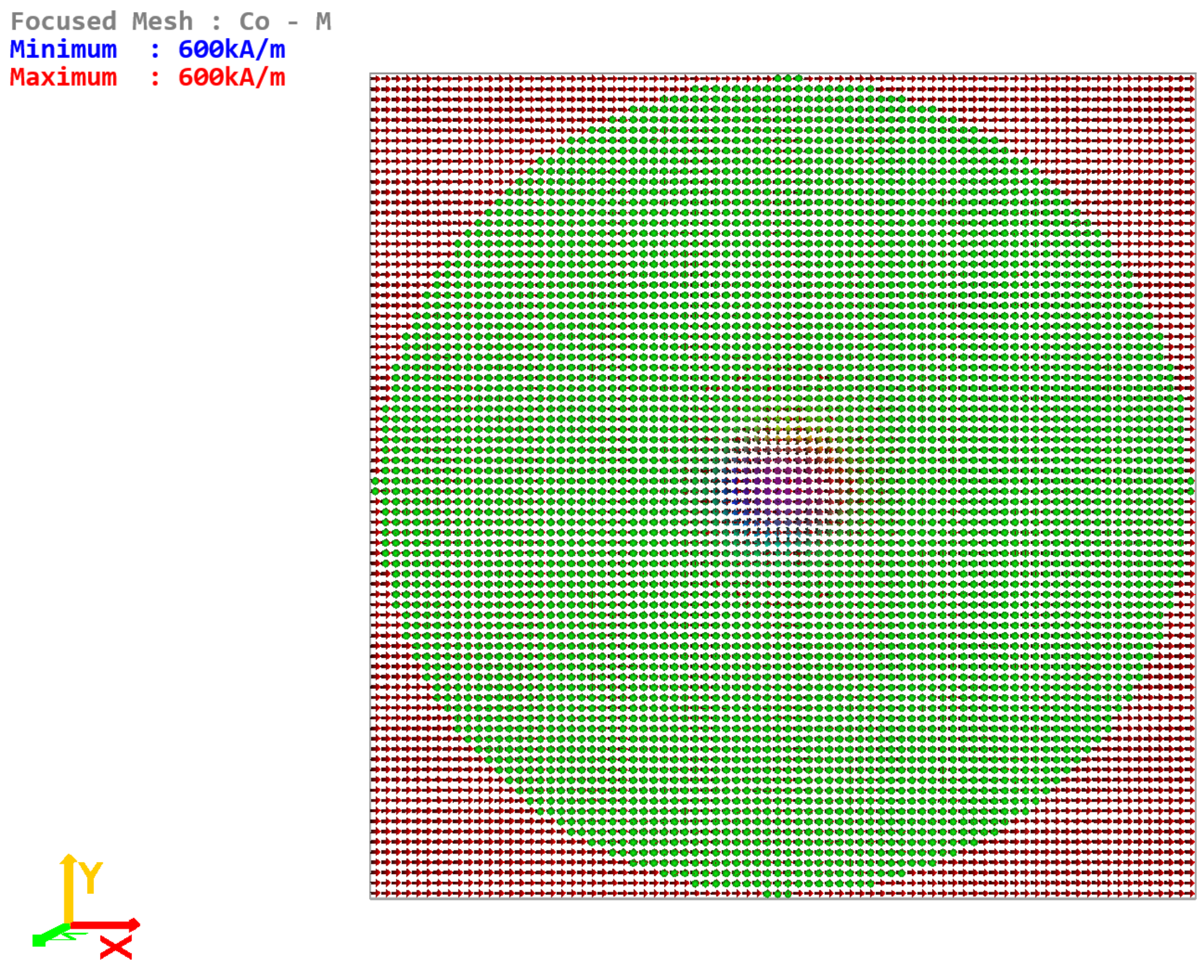
Exercise 23.1

a) Setup a Pt/Co bilayer with a skyrmion relaxed at the center of the Co layer under a 15 kA/m out-of-plane magnetic field as shown in Figure 23.1. The Pt layer should be a 320 nm × 320 nm × 3 nm rectangle, whilst the Co layer should be a 320 nm diameter disk with a 1 nm thickness. Relax this magnetization configuration.

   For Pt you should use σ = 7×10^6 S/m, λ_{sf} = 1.4 nm and θ_{SHA} = 0.19. Use a discretisation cellsize of (4 nm, 4 nm, 0.5 nm). Set *iSHA* to zero.

   For Co you should use σ = 5×10^6 S/m, λ_{sf} = 38 nm, λ_J = 2 nm, λ_φ = 4 nm, G_{mix} = 1.5 PS/m^2, g_{rel} = 1.3, α = 0.03, M_S = 600 kA/m, A = 10 pJ/m, D = -1.5 mJ/m^2, K_1 = 380 kJ/m^3 with uniaxial anisotropy perpendicular to the plane. You should also enable the interfacial DM exchange module. Use a discretisation cellsize of (4 nm, 4 nm, 1 nm) for magnetic computations and (4 nm, 4 nm, 0.25 nm) for spin transport computations. In the Co mesh only enable the interfacial spin torques, not the bulk spin torques or spin pumping.

Figure 23.1 – Skyrmion in a Co disk on a Pt underlayer.

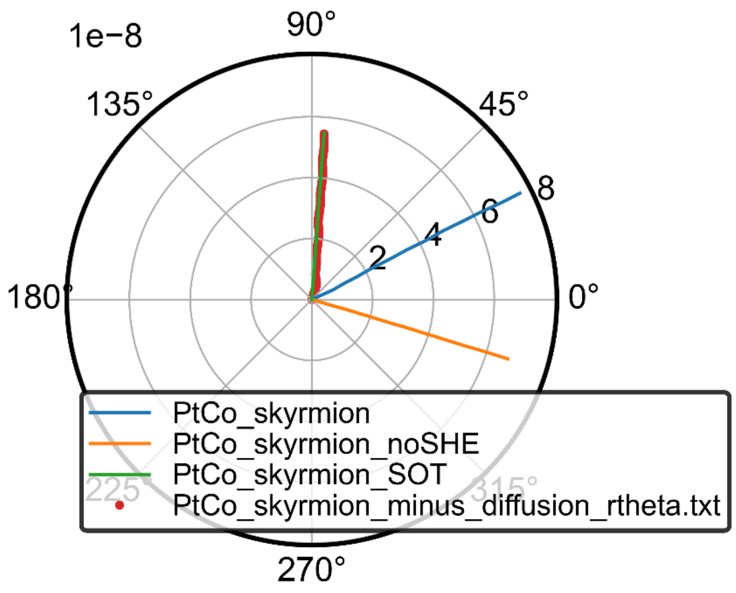


b) Enable the spin transport solver in both meshes and set electrodes at the x-axis ends of the Pt mesh only. Set a -20 mV potential for 3 ns and save the *time* and *skyshift* (or *skypos*) data every 10 ps. (for *skyshift* and *skypos* define a rectangle around the initial position of the skyrmion). Simulate the skyrmion movement path with SHE enabled (*SHA* = 0.19 in the Pt mesh), as well as without SHE (*SHA* = 0 in the Pt mesh), and plot them in polar coordinates.

c) Simulate the skyrmion movement path without the spin transport solver but with the *SOTfield* module enabled. Still keep the *transport* module enabled to calculate the charge current density. For the *SOTfield* module set a suitable effective spin Hall angle in the Co mesh (*SHA*). You can use the effective spin Hall angle formula from Tutorial 21, but note this is only strictly applicable for uniform magnetization and spin currents. You can use this as a starting point, but will need to adjust the effective spin Hall angle.

   Plot the skyrmion path in polar coordinates and compare it with the path obtained as the difference between the SHE and no SHE simulations above – see Figure 23.2 for expected results.

Figure 23.2 – Skyrmion movement paths obtained in Exercise 23.1.



Tutorial 24 – Roughness and Staircase Corrections

Staircase Corrections

With finite difference discretisation, errors can arise due to a staircase effect when discretising curved boundaries. In micromagnetics the largest errors arise in the demagnetizing field and may be reduced by decreasing the discretisation cellsize. This method is inefficient however since for most problems the results converge when the discretisation cellsize is close to the exchange length of the material – thus to further reduce this everywhere just to improve the discretisation accuracy at a boundary is very inefficient. Note, for materials where the demagnetizing energy dominates the exchange length may be defined as:

$$l_{ex} = \sqrt{\frac{2A}{\mu_0 M_S^2}}$$

For systems where the anisotropy energy dominates ($K_u > \mu_0 M_S^2 / 2$), the exchange length may be defined as:

$$l_{ex} = \sqrt{\frac{A}{K_u}}$$

Instead of refining this, a good approximation may be achieved by computing a correction field using a finely discretised demagnetization kernel, but applying it at run-time to the coarsely discretised mesh, as described in S. Lepadatu, Journal of Applied Physics 118, 243908 (2015). This correction field is typically similar to an uniaxial anisotropy field when averaged.

To enable staircase corrections in a particular magnetic mesh you must enable its *Roughness* module. When applying a mask shape to the mesh, staircase corrections will now automatically be taken into account. You must enable the *Roughness* module and reset the mesh shape before applying the mask to correctly enable staircase corrections. You must also set the required refinement using the **refineroughness** command:

refineroughness m_x m_y m_z

When calculating the correction field factors, the shape is first discretised on a fine mesh as set by the **refineroughness** parameters. Thus if the coarse mesh has cellsize (h_x, h_y, h_z), the fine mesh used for correction field initialization has cellsize (h_x / m_x, h_y / m_y, h_z / m_z). Typically the improvement in accuracy is small above $m > 10$, so the refinement set should not be excessive. Since a demagnetizing kernel must be computed for the fine mesh, the initialisation time may become very long, and the available memory may be exceeded if the *m* factors are set too large. You must also set them before applying the mask shape.

With the *Roughness* module enabled, setting a mask shape may result in a slightly different shape than without. This is because a fine shape is internally obtained first, then the coarse mesh shape is calculated to be the smallest shape which includes the fine shape on the coarse mesh – this is a requirement of the corrections calculation method. To clear the staircase corrections, effectively setting the fine mesh shape to the coarse mesh shape you can use:

clearroughness

There's an energy density term associated with the correction fields, and this is available as a **data** parameter: *e_rough*. The demagnetizing energy density, *e_demag*, still corresponds to the coarse mesh shape; the sum of *e_rough* and *e_demag* is the approximated demagnetizing energy for the fine mesh shape.

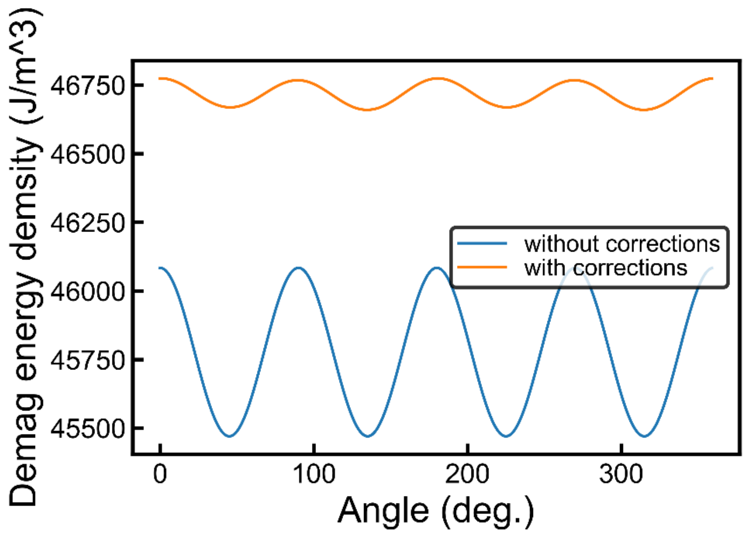
Exercise 24.1

Set a 80 nm diameter $Ni_{80}Fe_{20}$ disk with 10 nm thickness. Calculate the demagnetizing energy density as a function of in-plane uniform magnetization orientation from 0° through 360° by saturating in a strong magnetic field (10^6 A/m). Repeat this computation but now set the shape with the *Roughness* module enabled and a refinement of (10, 10, 1) –

refineroughness. Compare the demagnetizing energies for the coarse mesh and the approximated demagnetizing energy for the fine mesh (*e_rough* + *e_demag*).

For the above exercise, in theory the demagnetizing energy should be constant for a circle as the field rotates. In practice, due to discretisation errors a shape anisotropy effect is observed (the magnetization is not fully saturated even at 10^6 A/m, so some non-uniformity persists). With staircase corrections enabled this anisotropy should be significantly reduced, thus closer to the ideal uniform demagnetizing energy – see Figure 24.1. The refinement can be increased but further improvement is small.

Figure 24.1 – Demagnetizing energy computed for a circle with and without staircase corrections.



Edge and Surface Roughness

The same model used to reduce staircase corrections may be applied to compute the effect of topological roughness with variations below the exchange length of the material. As before, coefficients for a roughness field are computed at initialisation depending on the shape of a finely discretised mesh (the mesh with topological roughness applied), and that of the coarse mesh (the actual mesh used in computations but without roughness).

A roughness profile may be applied using a built-in algorithm, or alternatively a mask may be used. See Figures 24.2 – 4 for examples of real surface scans, processed into a grayscale image suitable for use as masks. These may be found in the Examples folder for this tutorial.

Figure 24.2 – Granular surface roughness profile

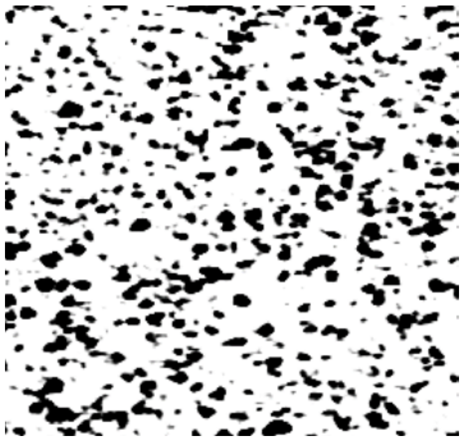


Figure 24.3 – Maze-like surface roughness profile.

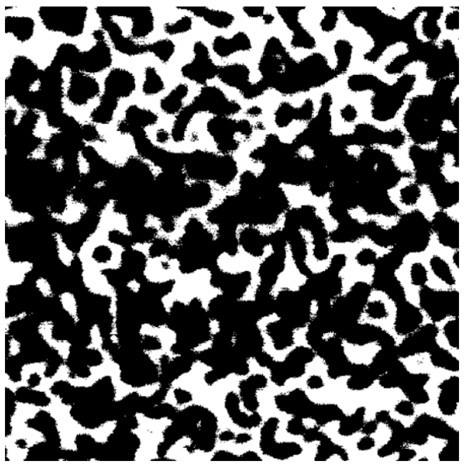
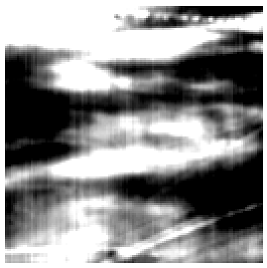


Figure 24.4 – Elongated defects, or stripes, surface roughness profile.



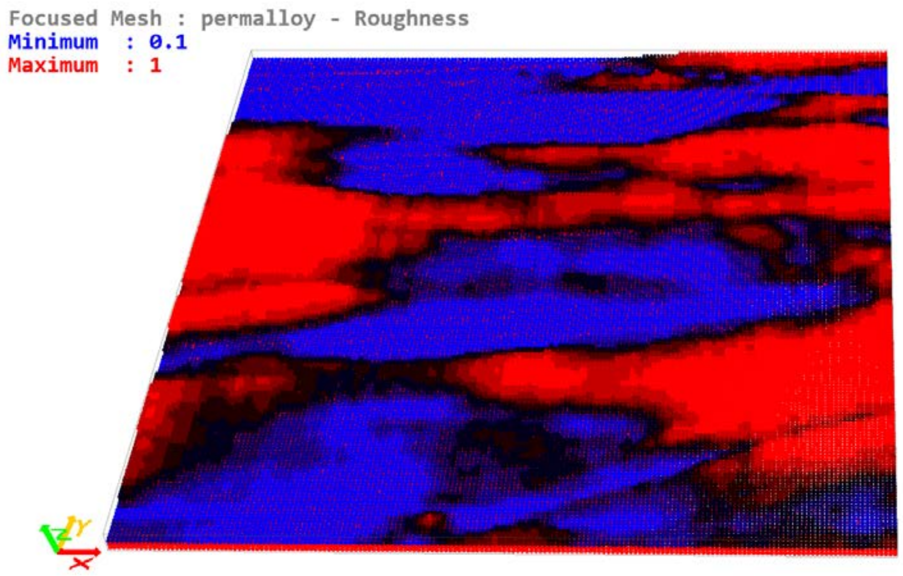
To apply a roughness profile using a mask, instead of simply dragging the file to the mesh viewer (as you would do when applying a shape), you should also specify the depth to which you want to apply the profile. For example, with the mask shown in Figure 24.4, to apply it to a 4.5 nm depth (the coarse discretisation cellsize is 5 nm so this keeps the coarse mesh shape intact) you need to use:

loadmaskfile *4.5nm (directory\)Stripes*

This will apply a surface roughness profile on the top face up to 4.5 nm depth – black results in 0 depth cut, whilst white results in full depth cut (i.e. 4.5 nm); values on the greyscale in between are correlated linearly with the depth cut. For the actual surface roughness profile obtained see Figure 24.5.

In this example the starting mesh has dimensions of 320 nm × 320 nm × 10 nm, and the roughness refinement was set to (4, 4, 10) – **refineroughness**. To view the set roughness, under **display** select the *Roughness* option for the respective mesh. To apply the surface roughness to the bottom face, negative values need to be set for the depth value – see help for **loadmaskfile** command.

Figure 24.5 – Applied surface roughness using the mask in Figure 24.4.



You can also apply edge and surface roughness using a built-in console command. Currently two methods are available: **roughenmesh**, **surfroughenjagged**. The **roughenmesh** command applies a completely random roughness profile to one of the 6 faces as indicated –

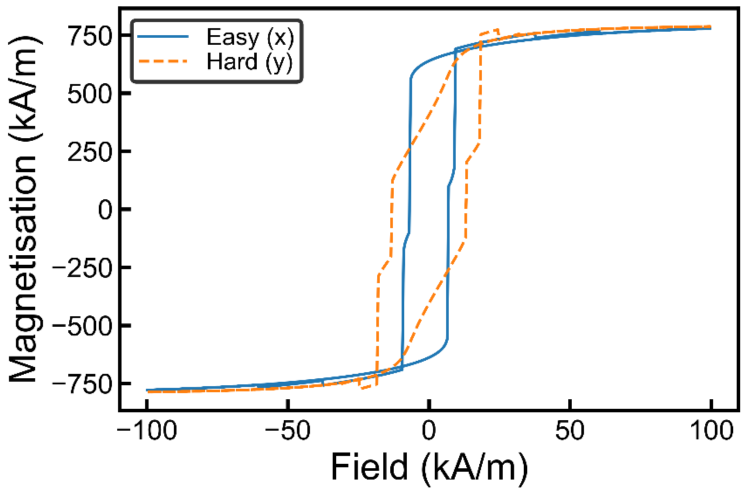
see help for this command. The **surfroughenjagged** command applies a jagged profile to either the top, bottom, or both, faces – see help for this command.

Exercise 24.2

Set a 160 nm × 160 nm × 10 nm permalloy rectangle and apply the Stripes roughness profile from Figure 24.4 (use file in Examples folder) to a depth of 4.5 nm. Use a roughness refinement of (4, 4, 10). Simulate the hysteresis loops along the x and y directions and compare them. (*You should apply the field at a slight angle to the x and y directions to avoid artifacts associated with a finite geometry – in particular for the easy axis you want to avoid the "U" shape configuration at zero field which can happen if the field is perfectly along the x axis.*)

Since permalloy does not have a magneto-crystalline anisotropy, without roughness it is expected the two hysteresis loops will be identical. With roughness applied an effective anisotropy is observed, due to the orientation of the surface roughness stripes as seen in Figure 24.5.

Figure 24.6 – Hysteresis loops for Exercise 24.2, showing a roughness-induced anisotropy effect.



Tutorial 25 – Defects and Impurities

Material parameters in Boris may also be assigned a spatial variation, in addition to a temperature dependence. This spatial variation will be taken into account in all routines where the material parameters appear, allowing inclusion of material defects and impurities in simulations as appropriate.

To see the currently set parameters spatial variation use the command:

paramsvar

You can use a pre-defined method of generating defects by following the instructions displayed after using the **paramsvar** command. Currently these include: *random*, *jagged*, *defects*, *faults*. To see the spatial variation generated, under **display** select the *ParamVar* option, making sure to select the required parameter under the **paramvar** list. The generated spatial variation is stored as an array of coefficients, multiplying the base parameter value. For examples of these profiles see Figures 25.1 – 4.

Figure 25.1 – *Random* parameter variation between 0.9 and 1.1 with generator seed 1.

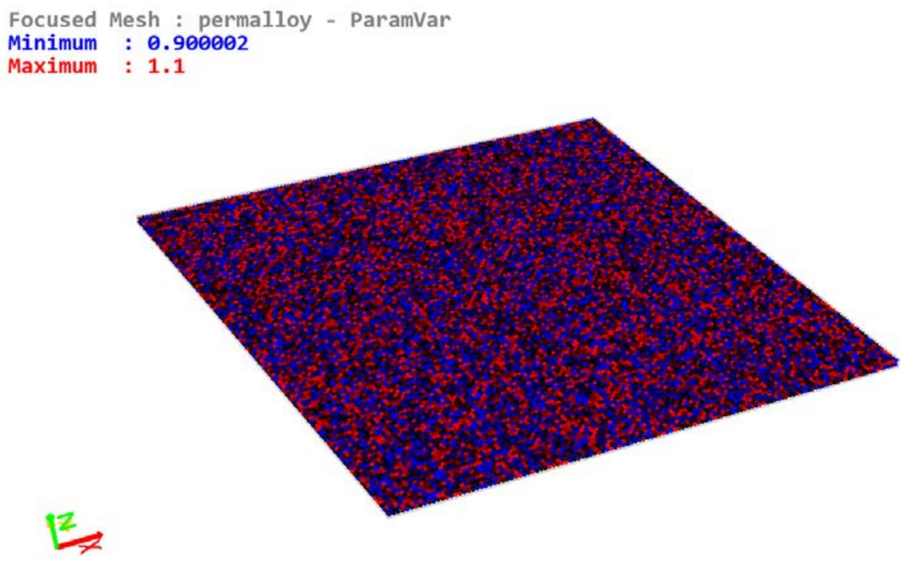

```
Focused Mesh : permalloy - ParamVar
Minimum  : 0.900002
Maximum  : 1.1
```

Figure 25.2 – *Jagged* parameter variation between 0.9 and 1.1 with 30 nm average spacing and generator seed 1.

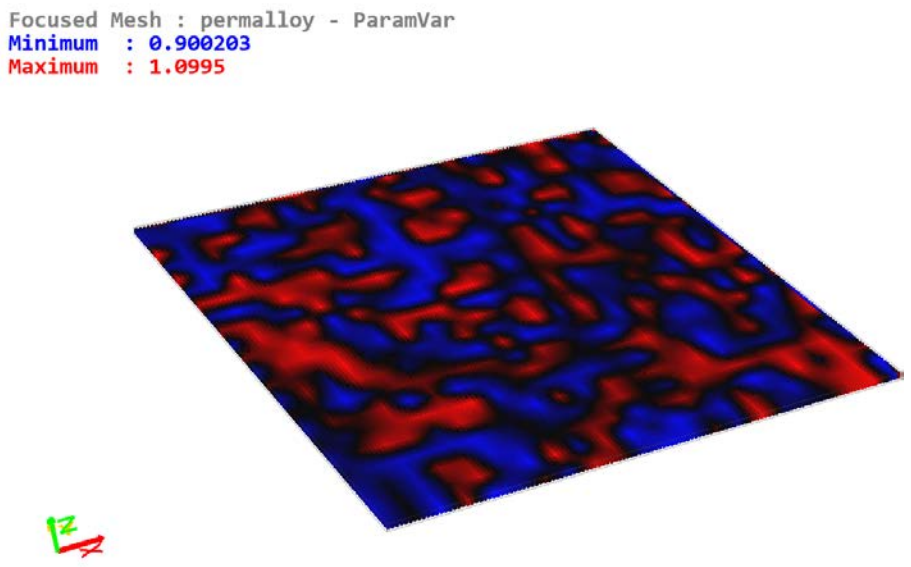


Figure 25.3 – *Defects* parameter variation between 0.9 and 1.1 with diameters in the range 20 nm to 50 nm, and 40 nm average spacing with generator seed 1.

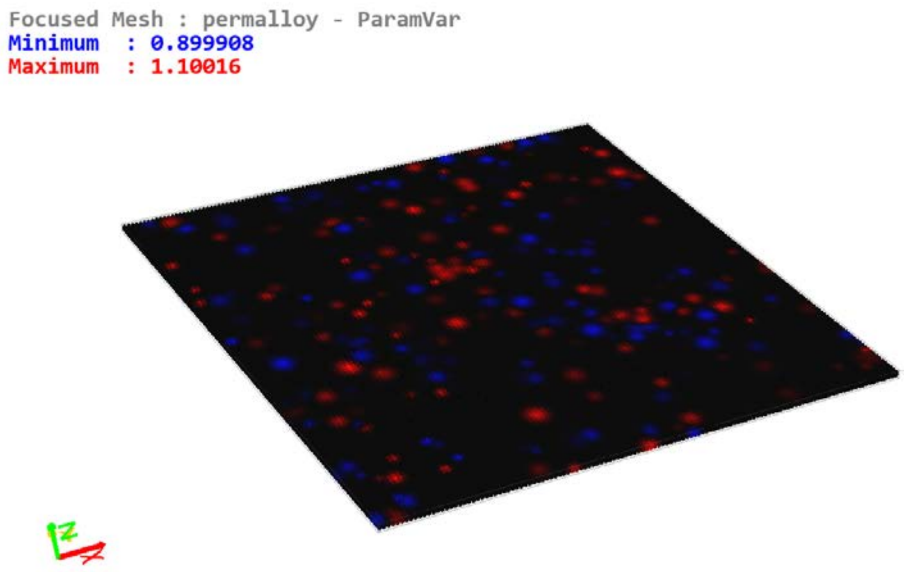
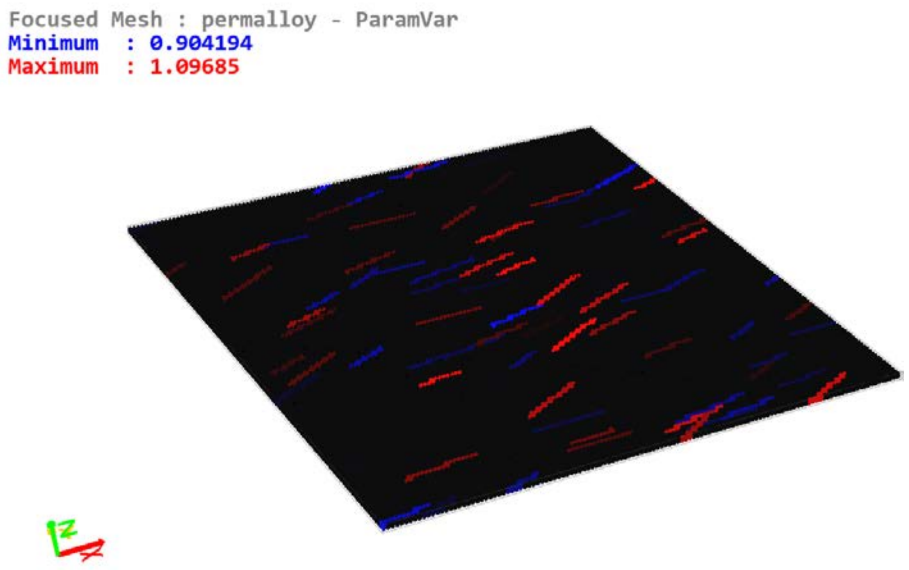


Figure 25.4 – *Faults* parameter variation between 0.9 and 1.1 with 20 nm to 50 nm fault length, -30° to 30° fault orientation and 50 nm average spacing with generator seed 1.


```
Focused Mesh : permalloy - ParamVar
Minimum  : 0.904194
Maximum  : 1.09685
```

<u>*Exercise 25.1*</u>

Generate M_S defects in a 640 nm × 640 nm × 10 nm mesh as shown in Figures 25.1 – 4.

You can also set a custom parameter variation using an image as a mask file, although this is now a legacy option and documented in a previous version (v2.4). Instead if you want to set an arbitrary parameter variation you should use the ovf2 file option, which can be programmatically generated – a routine is included in the NetSocks module, allowing for easy generation of parameter spatial variation. This is covered in a separate chapter in the manual on working with ovf2 files. You can also set parameter variation using a text equation, which simultaneously allows for both spatial and temporal dependence. This is covered in a separate chapter in the manual on working with text equations.

Tutorial 26 – Polycrystalline and Granular Films

Boris includes a Voronoi tessellation generator, both 2D and 3D, which can be used to generate polycrystalline and granular films.

Figure 26.1 – Polycrystalline film showing *K1* parameter variation generated using **vor2D** generator between 0.9 and 1.1 with 40 nm spacing and generator seed 1.

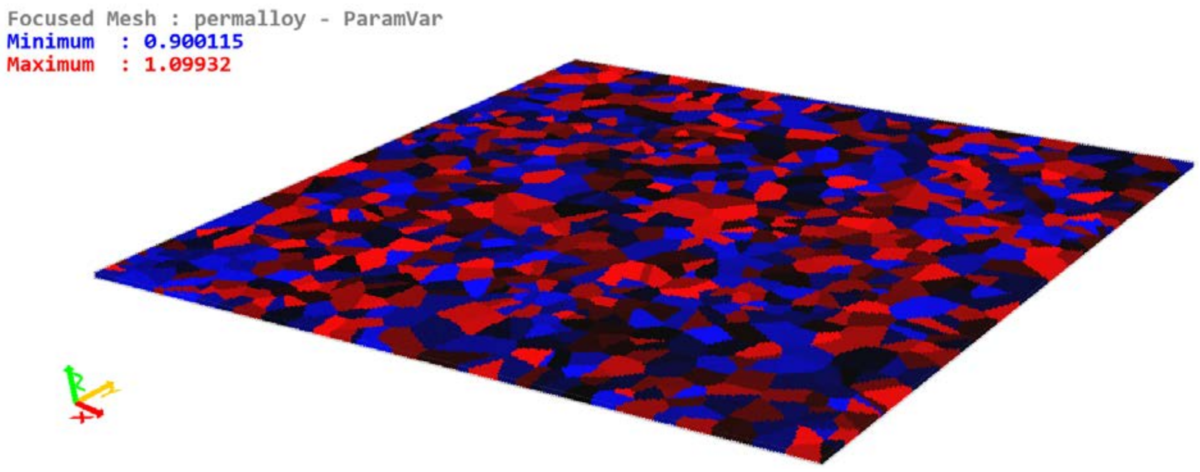
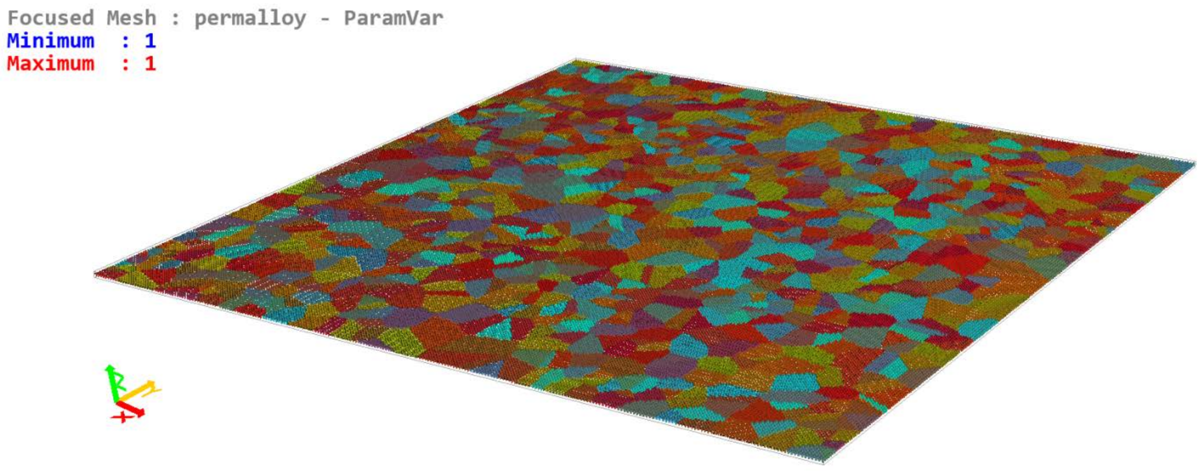

```
Focused Mesh : permalloy - ParamVar
Minimum  : 0.900115
Maximum  : 1.09932
```

Figure 26.2 – Polycrystalline film showing easy axis (*ea1*) parameter variation generated using **vorrot2D** generator with polar angle range 70° to 110°, azimuthal angle range -90° to 90°, with 40 nm spacing and generator seed 1.


```
Focused Mesh : permalloy - ParamVar
Minimum  : 1
Maximum  : 1
```

Polycrystalline films may be simulated by generating parameter variations using one of the Voronoi tessellation generators under the **paramsvar** command. These include:

vor2d *min, max; spacing; seed* – Used for 2d crystallites in the xy plane.
vor3d *min, max; spacing; seed* – Used for 3d crystallites.

vorbnd2d *min, max; spacing; seed* – Used for 2d crystallites in the xy plane, but parameter variation generated randomly only at Voronoi cell boundaries.
vorbnd3d *min, max; spacing; seed* – Used for 3d crystallites, but parameter variation generated randomly only at Voronoi cell boundaries.

vorrot2d *min_polar, max_polar; min_azimuthal, max_azimuthal, spacing; seed* – Used for 2d crystallites in the xy plane, specifically magneto-crystalline anisotropy easy axes.
vorrot3d *min_polar, max_polar; min_azimuthal, max_azimuthal, spacing; seed* – Used for 3d crystallites, specifically magneto-crystalline anisotropy easy axes.

Both 2D and 3D crystallites may be generated using the generators listed above. For an example of a 2D polycrystalline film with *K1* (magneto-crystalline anisotropy) variation see Figure 26.1. In order to generate crystallites with a varying magneto-crystalline anisotropy easy axis orientation you can use either the **vorrot2d** or **vorrot3d** generator – for example see Figure 26.2 for the *ea1* parameter having the same polycrystalline structure as in Figure 26.1. Figure 26.2 shows the rotation to be applied, as a vector quantity. For an easy axis base value set along the x-axis this coincides with the resulting easy axis orientation.

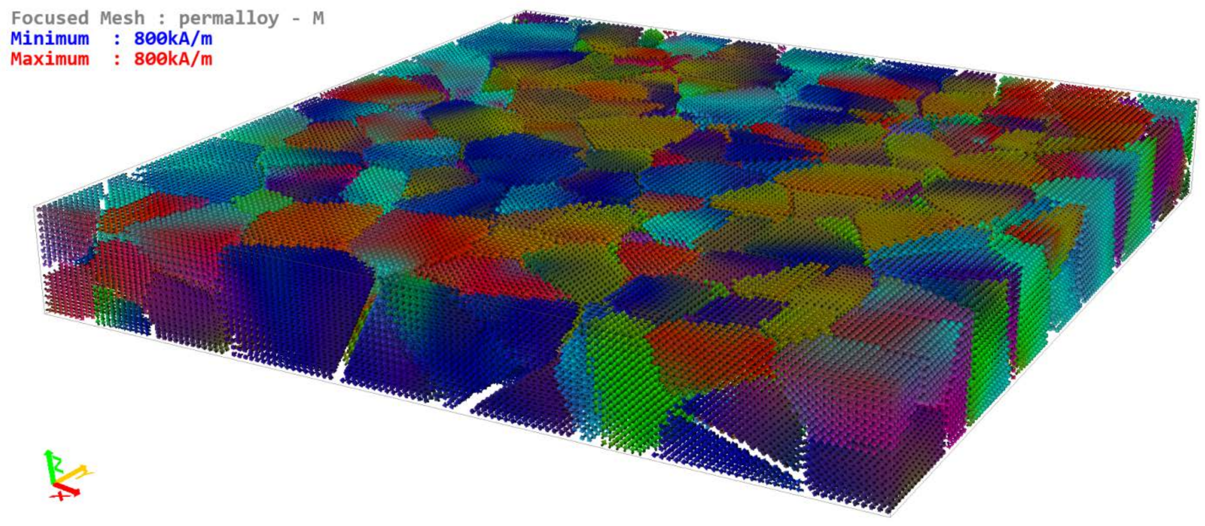
You can also generate a parameter variation at the Voronoi cell boundaries rather than in the cells themselves. This can be done using the **vorbnd2d** and **vorbnd3d** generators. This could be useful for example to modify the electrical conductivity at the grain boundaries only.

In order to generate a granular film with non-magnetic phase separation you can use one of the following commands:
generate2dgrains *spacing (seed)*
generate3dgrains *spacing (seed)*

These commands generate granular films directly on the magnetization mesh – see for example Figure 26.3. You can also combine this with parameter variations generated using the same generator seed and sizes (e.g. grains with varying M_S values).

When generating grains for the magnetic mesh you can choose to generate grains for the electrical conductivity mesh also. To carry the grain structure over you should generate the grains first without the *Transport* module enabled. After enabling the *Transport* module the granular structure is also applied to the electrical conductivity mesh. This could be useful for example in a multi-layered structure. If you apply the grain structure with the *Transport* module already enabled, the grains are not generated for the electrical conductivity also. You could combine this with a Voronoi generator for the *elC* material parameter however (e.g. **vorbnd2d** or **vorbnd3d**, or even **vor2d** or **vor3d**) as mentioned above.

Figure 26.3 – Granular film (80 nm thick) with non-magnetic phase separation, generated using **generate3dgrains** command with 50 nm spacing and generator seed 1. The image shows a magnetization configuration at zero field.



Tutorial 27 – Periodic Boundary Conditions

Periodic boundary conditions (PBC) may be applied when computing demagnetizing fields. Enabling this setting also affects exchange coupling (e.g. *Exchange*, *DMExchange*, *iDMExchange*), resulting in a wrap-around effect. PBCs are useful to simulate periodic arrays, or to approximate effectively infinite thin films or tracks using only a finite simulation window. PBCs are applicable to both single mesh demagnetization (*Demag*), as well as supermesh demagnetization (*SDemag*), including multi-layered demagnetization. Moreover, when enabling the *Roughness* module, PBCs are also taken into account when calculating an effective roughness field.

To enable PBCs use the command:

pbc

The configuration for all meshes, or the supermesh if applicable, will be shown. You can enable PBCs in any direction by setting a number of images using the interactive console objects (10 images are set by default when enabled, which can be edited if required; the default setting is for no PBCs). If the *SDemag* module is enabled you can edit the PBC settings on the supermesh, otherwise you need to edit them for the individual meshes.

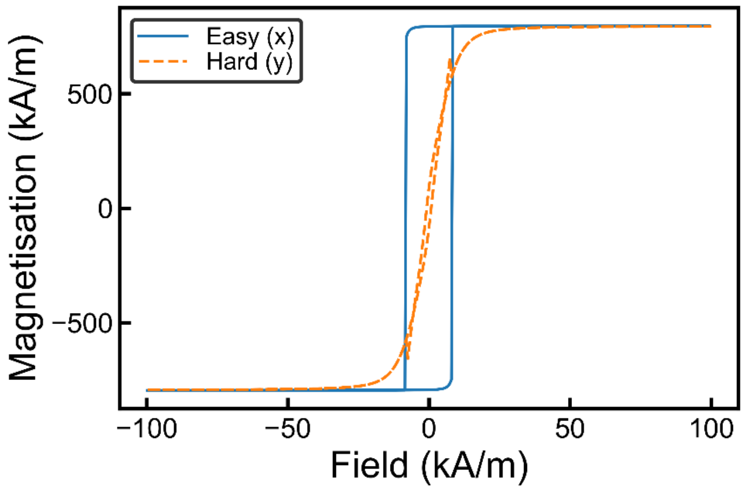
When using PBCs with multi-layered convolution you need to ensure the problem is physically meaningful. For example the recommended approach is to use a z direction stacking of layers, then either x or y PBCs are fine, but z PBCs will not lead to physically meaningful results in this case.

Exercise 27.1

Repeat Exercise 24.2, but this time set default PBCs along both x and y. Compare the hysteresis loops.

When using PBCs with the *Roughness* module enabled, you should be careful about setting a combination of a large number of PBC images and fine roughness refinement (**refineroughness**). This can result in excessive initialization time due to the large demag kernel calculated by the *Roughness* module.

Figure 27.1 – Hysteresis loops obtained for Exercise 27.1. The small peaks in the hard axis loop are artifacts resulting from the finite simulation mesh size, and could be decreased by increasing the simulation mesh rectangle.



Tutorial 28 – Ultrafast Demagnetization

Ultrafast demagnetization processes may be studied in Boris using the LLB equation (or sLLB) coupled to a two-temperature model. The default heat equation doesn't differentiate between lattice and electron temperature, i.e. it is a 1-temperature model. With the two-temperature model, two coupled equations for the electron and lattice temperatures are given as:

$$C_e \rho \frac{\partial T_e(\mathbf{r},t)}{\partial t} = \nabla . K \nabla T_e(\mathbf{r},t) - G_e(T_e - T_l) + S$$
$$C_l \rho \frac{\partial T_l(\mathbf{r},t)}{\partial t} = G_e(T_e - T_l)$$

Here C_e and C_l are the electron and lattice specific heat capacities, ρ is the mass density, K is the thermal conductivity, and G_e is the electron-lattice coupling constant, typically of the order 10^{18} W/m^{3}K.

To change the temperature model use the following command (or use the interactive console output from **tmodel** command):

tmodel *num_temperatures (meshname)*

e.g. **tmodel** 2 sets the two-temperature model for the currently focused mesh (nnot applicable to *insulator* meshes). The material parameters in the above equations are available as usual under the **params** command console output (also see list of parameters under the Material Parameters section in this manual).

In the above equation *S* is a heat source; a heat source may be specified in simulation using one of the following stage types: *Q*, *Q_seq*, *Qequation*, *Qfile*. These stages specify a heat source (W/m^3) in the heat equation: *Q* sets a constant value, *Q_seq* sets a sequence of values (similar to Hxyz_seq for external fields), *Qequation* sets a heat source using a text equation, thus allowing both spatial and temperature dependence, and finally *Qfile* sets a spatially uniform heat source, but with arbitrary time dependence as specified using a text file (see description of stage in console help). Moreover a spatial variation may be assigned to the parameter *Q* under the **paramsvar** command.

A typical heat source from a focused laser pulse is given as:

$$S = P_0 \exp\left(\frac{-|\mathbf{r}-\mathbf{r}_0|}{d^2/4\ln(2)}\right)\exp\left(\frac{-(t-t_0)^2}{t_R{}^2/4\ln(2)}\right) \quad (W/m^3)$$

Here d and t_R are full-width at half-maximum (FWHM) values (pulse diameter and duration respectively) and P_0 is the maximum power density. This heat pulse may be simulated using a *Qequation* stage set to (pulse centre coincides with simulation mesh centre):

*Q0 * exp(-sqrt((x/Lx - 0.5)^2 + (y/Ly - 0.5)^2) / ((d0/Lx)^2/(4*ln(2)))) * exp(-(t-2*tau)^2/(tau^2/(4*ln(2))))*

For this equation we need to define the user constants (**equationconstants**): i) Q0, ii) d0, iii) tau.

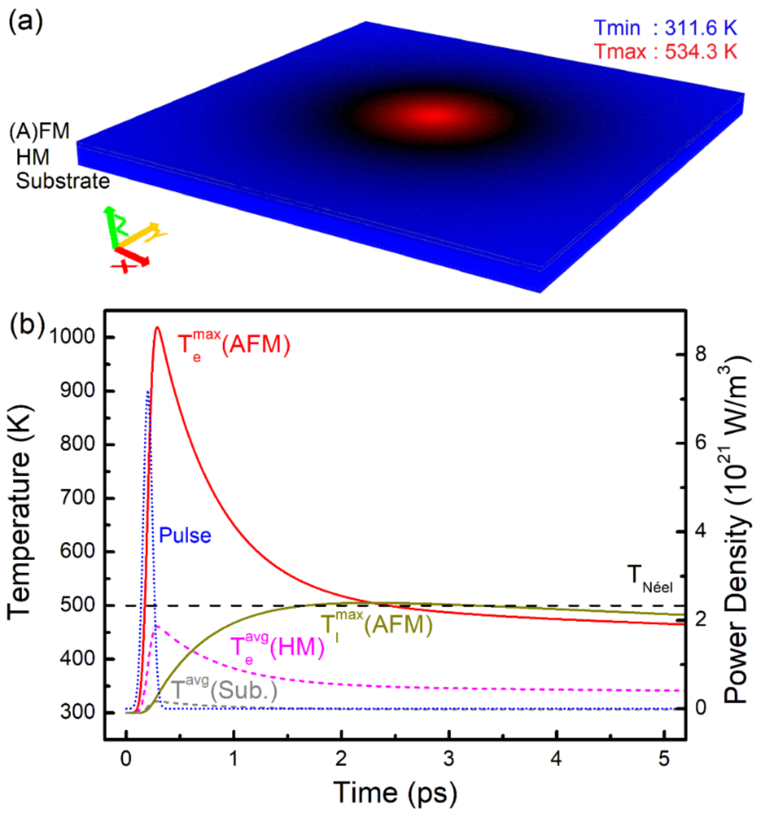
Exercise 28.1

In this exercise you will simulate the effect of a single laser pulse on a Co/Pt/SiO$_2$ trilayer, similar to that used in [Phys. Rev. B 102, 094402 (2020)], when the electron temperature rises above the Curie temperature, and observe creation of Néel skyrmions, plotting the variation in topological charge magnitude as a function of time. The structure to simulate is shown in Figure 28.1. The topological charge is computed as:

$$Q = \frac{1}{4\pi}\int_A \mathbf{m}.\left(\frac{\partial \mathbf{m}}{\partial x}\times\frac{\partial \mathbf{m}}{\partial y}\right)dxdy$$

The topological charge takes on unit values for skyrmions, and may be computed in Boris using the **dp_topocharge** command.

Figure 28.1 – Trilayer structure similar to that used for Exercise 28.1, consisting of Co (2 nm) / Pt (8 nm) / SiO_2 (40 nm), showing (a) temperature during a Gaussian profile laser pulse, and (b) typical ultrafast laser pulse and temperature time dependence in the three layers: Co layer maximum electron and lattice temperatures, Pt layer average electron temperature and SiO_2 average temperature. Reproduced from [Phys. Rev. B 102, 094402 (2020)].

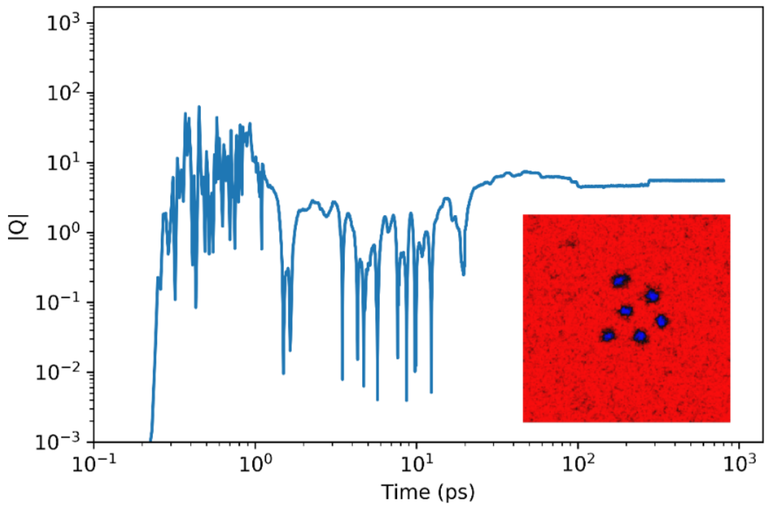


See the ufsky_creation.py script in the Examples/Tutorial 0 folder. This script sets up the simulation for this exercise from scratch. Study the script to understand what all the commands are used for (if needed look up the command help in the console or manual). Run this script several times and build a probability distribution of number of skyrmions created in each run. Is the computed probability distribution described by a Poisson counting distribution? What is the mean number of skyrmions created?

Hint: to obtain a reasonable probability distribution you will need to run the script at least 50 times – this skyrmion counting process can be automated, and will require up to 1h or 2h of simulation time depending on your workstation.

Further hint: the computed Q value with dp_topocharge will not be an integer for two reasons: i) stochasticity, ii) cellsize could be too large. However, rounding the Q value will result in a correct integer Q value. The other possibility is to turn off the stochasticity at the end of the simulation and allow the magnetization to quickly relax before obtaining the Q value with dp_topocharge. With an in-plane cellsize of 1 nm this will result in a value very close to an integer (e.g. -3.98 for 4 skyrmions etc.).

Figure 28.2 – Topological charge magnitude as a function of time, computed for a run of Exercise 28.1. The resulting skyrmion state after 800 ps is shown in the inset, showing 6 skyrmions created (Q ≅ -6).



Tutorial 29 – Magneto-Optical Effect

With circularly polarised pulses the polarisation can be clockwise or anti-clockwise. Due to the circular polarisation of the laser pulse a strong perpendicular magneto-optical field is present, given by $H_{MO} = \sigma^{\pm} H_{MO}^{0} f_{MO}(\mathbf{r},t)\hat{\mathbf{z}}$. Here f_{MO} gives the spatial and temporal dependence of the laser pulse, H_{MO} (A/m) gives the strength of the magneto-optical field, and $\sigma^{\pm} = \pm 1$ its helicity. Thus for the Gaussian laser pulse from the previous Tutorial, the spatial and temporal dependence of the magneto-optical field is given by:

$$f_{MO}(\mathbf{r},t) = \exp\left(\frac{-|\mathbf{r}-\mathbf{r}_0|}{d^2/4\ln(2)}\right)\exp\left(\frac{-(t-t_0)^2}{t_R{}^2/4\ln(2)}\right)$$

In Boris you can enable the magneto-optical effect with the *moptical* module (**addmodule** *meshname moptical*).

The H_{MO} parameter appears as a material parameter (see **params**), named *Hmo*. Thus you can set the strength of the magneto-optical effect by setting the parameter value. You can also set the f_{MO} dependence as above by setting a material parameter variation for *Hmo* (**paramsvar**). Thus for the above example you need to set the spatial variation using a text equation set to:

*exp(-sqrt((x/Lx - 0.5)^2 + (y/Ly - 0.5)^2) / ((d0/Lx)^2/(4*ln(2)))) * exp(-(t-2*tau)^2/(tau^2/(4*ln(2))))*

Exercise 29.1

Based on the simulation file from Exercise 28.1, study the effect of a train of circularly polarised pulses with positive or negative helicities on skyrmion creation. Use a reduced laser power density which does not raise the temperature above the Curie temperature (set Q0 = 9e20 W/m^3). Write a Python script to apply 20 laser pulses with a 6 ps repetition period and strength of 40 MA/m. Compare the results after 30 pulses for the two helicities.

Tutorial 30 – Spin-Wave Dispersion

Similarly to the method of simulating frequency-swept FMR (see tutorial on FMR first), we can simulate spin-wave dispersion by applying a sinc pulse which has not only a time dependence, but also a spatial dependence. In the xy plane the excitation field has the form:

$$H(t) = H_e sinc(k_c(x - x_0))sinc(k_c(y - y_0))sinc(2\pi f_c(t - t_0))$$

As before f_c is the frequency cut-off (Hz) with t_0 the temporal sinc pulse centre. To excite spin-waves with non-zero wave-vector we also need a spatial dependence for the sinc pulse. Here k_c is the wave-vector cut-off (rad/m), and (x_0, y_0) is the spatial sinc pulse centre. When obtaining the spin-wave dispersion, we need to consider the direction of the wave-vector, **k**. Instead of sampling the average magnetization, we need to obtain a magentisation profile along a given direction. The direction in which we sample is the wave-vector direction **k**. You can obtain a magentisation profile using the **dp_getexactprofile** command. Also we need to sample the magnetisaiton at fixed steps, determined by the Nyquist criterion (as for temporal samping):

$$\Delta_S = \frac{\pi}{k_C} \quad (m)$$

We only need to sample one component of the magnetization, and the spin wave dispersion is obtained by performing a 2D Fourier transform on the spatial-temporal 2D data, transforming it to wave-vector-frequency 2D space.

Consider a spin-wave waveguide as an elongated magnetic track. There are three possible configurations, determined by the direction of the bias field.

Bias field along length – this is called the backward volume configuration.
Bias field along thickness – this is called the forward volume configuration.
Bias field along width – this is called the surface spin wave configuration.

In all three cases the excitation and analysed magnetization component need to be perpendicular to the bias field; for simplicity the analysed magnetization component can be chosen to be along the excitation direction. When simulating the spin-wave dispersion it is important to choose the cell-size correctly, as the computed spin-wave dispersion will be inaccurate at larger wave-vector values if the cellsize is not small enough.

Exercise 30.1

Read through the reference IEEE Trans. Mag. 49, 524 (2013). Simulate the spin-wave dispersion as described in this reference for the three spin-wave configurations described above. Use periodic boundary conditions along the length only.

You should make the following modifications to the proposed problem:
Cellsize along length should be 1 nm; the specified 2 nm cellsize results in a large discrepancy between computations and analytical formulas at larger k values.
You should simulate for a total time of 2×t_0 as explained in the tutorial on FMR. For this exercise it is suggested you use a t_0 value of 200 ps, although 100 ps is still fine. A value of 50 ps results in a coarse frequency spectrum.
The spatial sampling interval should be 4 nm due to the Nyquist criterion.

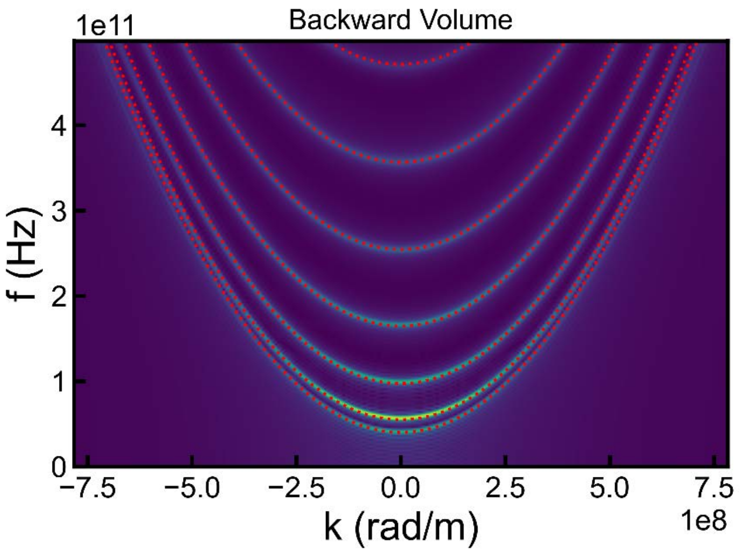
*Hint: use an Hequation stage set to 'H0, He * sinc(kc*(x-Lx/2))*sinc(kc*(y-Ly/2))*sinc(2*PI*fc*(t-t0)), 0' for the backward volume configuration.*

When analysing the spin-wave dispersion you can use the following formula:

$$w = w_n + w_M \frac{2A}{\mu_0 M_S^2} k^2 \quad (rad/s)$$

Here w_n is the resonance frequency (rad/s) for the nth spin-wave mode at k = 0, and may be extracted from the computed spectrum at k = 0, or alternatively you can use an analytical formula to predict it (not given here). The value w_M is given as γM_S, where $\gamma = \mu_0|\gamma_e|$ (2.212761569×10^5 mA/s).

Figure 30.1 – Spin-wave dispersion computed in Exercise 30.1 together with analytical predictions (even spin-wave modes excited n = 0, 2, 4, 6, 8, 10, 12). A damping value of 0.01 was used.



Tutorial 31 – Two-Sublattice Model

In a micromagnetics formulation we can study antiferromagnetic, ferrimagnetic, as well as binary ferromagnetic alloys using a two-sublattice model, where we consider magnetic orders of two sub-lattices A, B, and couple them using inter-lattice exchange stiffness terms. The two-sublattice stochastic LLB equation (sLLB) is given below:

$$\frac{\partial \mathbf{M}_i}{\partial t} = -\tilde{\gamma}_i \mathbf{M}_i \times \mathbf{H}_{eff,i} - \tilde{\gamma}_i \frac{\tilde{\alpha}_{\perp,i}}{M_i} \mathbf{M}_i \times \left(\mathbf{M}_i \times \left(\mathbf{H}_{eff,i} + \mathbf{H}_{th,i}\right)\right)$$
$$+ \gamma_i \frac{\tilde{\alpha}_{\parallel,i}}{M_i} \left(\mathbf{M}_i . \mathbf{H}_{\parallel,i}\right) \mathbf{M}_i + \boldsymbol{\eta}_{th,i} \quad (i = A, B)$$

The reduced gyromagnetic ratio is given by $\tilde{\gamma}_i = \gamma_i / \left(1 + \alpha_{\perp,i}^2\right)$, and the reduced transverse and longitudinal damping parameters by $\tilde{\alpha}_{\perp(\parallel),i} = \alpha_{\perp(\parallel),i} / m_i$, where $m_i(T) = M_i(T) / M_{S,i}^0$, with $M_{S,i}^0$ denoting the zero-temperature saturation magnetization, and $M_i \equiv |\mathbf{M}_i|$. The damping parameters are continuous at T_N – the phase transition temperature – and given by:

$$\alpha_{\perp,i} = \alpha_i \left(1 - \frac{T}{3\left(\tau_i + \tau_{ij} m_{e,j} / m_{e,i}\right)\tilde{T}_N} \right), \quad T < T_N$$

$$\alpha_{\parallel,i} = \alpha_i \left(\frac{2T}{3\left(\tau_i + \tau_{ij} m_{e,j} / m_{e,i}\right)\tilde{T}_N} \right), \quad T < T_N$$

$$\alpha_{\perp,i} = \alpha_{\parallel,i} = \frac{2T}{3T_N}, \quad T \geq T_N$$

We denote $\tilde{T}_N$ the re-normalized transition temperature, given by:

$$\tilde{T}_N = \frac{2T_N}{\tau_A + \tau_B + \sqrt{\left(\tau_A - \tau_B\right)^2 + 4\tau_{AB}\tau_{BA}}}$$

The micromagnetic parameters τ_i and $\tau_{ij} \in [0, 1]$, are coupling parameters between exchange integrals and the phase transition temperature, such that $\tau_A + \tau_B = 1$ and $|J| = 3\tau k_B T_N$. Here J is the exchange integral for intra-lattice ($i = A,B$) and inter-lattice ($i,j = A,B,\ i \neq j$) coupling respectively. For a simple antiferromagnet we have $\tau_A = \tau_B = \tau_{AB} = \tau_{BA} = 0.5$. The normalized equilibrium magnetization functions $m_{e,i}$ are obtained from the Curie-Weiss law as:

$$m_{e,i} = B\left[\left(m_{e,i}\tau_i + m_{e,j}\tau_{ij}\right)3\tilde{T}_N / T + \mu_i \mu_0 H_{ext} / k_B T\right],$$

where $B(x) = \coth(x) - 1/x$, and μ_i is the atomic magnetic moment. The magnetization length is not constant, and can differ from the equilibrium magnetization length, giving rise to a longitudinal relaxation field which includes both intra-lattice and inter-lattice contributions:

$$\mathbf{H}_{\|,i} = \left\{ \frac{1}{2\mu_0 \tilde{\chi}_{\|,i}} \left(1 - \frac{m_i^2}{m_{e,i}^2}\right) + \frac{3\tau_{ij} k_B T_N}{2\mu_0 \mu_i} \left[\frac{\tilde{\chi}_{\|,j}}{\tilde{\chi}_{\|,i}} \left(1 - \frac{m_i^2}{m_{e,i}^2}\right) - \frac{m_{e,j}}{m_{e,i}} \left(\hat{\mathbf{m}}_i . \hat{\mathbf{m}}_j\right) \left(1 - \frac{m_j^2}{m_{e,j}^2}\right) \right] \right\} \mathbf{m}_i, \quad T < T_N$$

$$\mathbf{H}_{\|,i} = -\left\{ \frac{1}{\mu_0 \tilde{\chi}_{\|,i}} + \frac{3\tau_{ij} k_B T_N}{\mu_0 \mu_i} \left[\frac{\tilde{\chi}_{\|,j}}{\tilde{\chi}_{\|,i}} - \frac{m_{e,j}}{m_{e,i}} \left(\hat{\mathbf{m}}_i . \hat{\mathbf{m}}_j\right) \right] \right\} \mathbf{m}_i, \quad T > T_N$$

Here $\hat{\mathbf{m}}_i = \mathbf{m}_i / m_i$, and the relative longitudinal susceptibility is $\tilde{\chi}_{\|,i} = \chi_{\|,i} / \mu_0 M_{S,i}^0$, where:

$$k_B T \tilde{\chi}_{\|,i} = \frac{\mu_i B_i' \left(1 - 3\tau_j \tilde{T}_N B_j' / T\right) + \mu_j 3\tau_{ij} \tilde{T}_N B_i' B_j' / T}{\left(1 - 3\tau_i \tilde{T}_N B_i' / T\right)\left(1 - 3\tau_j \tilde{T}_N B_j' / T\right) - \tau_{ij}\tau_{ji} B_i' B_j' \left(3\tilde{T}_N / T\right)^2}$$

and $B_i' \equiv B_{m_{e,i}}' \left[\left(m_{e,i}\tau_i + m_{e,j}\tau_{ij}\right)3\tilde{T}_N / T\right]$.

The direct exchange term includes the usual intra-lattice contribution, as well as homogeneous and non-homogeneous inter-lattice contributions, and is given by:

$$\mathbf{H}_{ex,i} = \frac{2A_i}{\mu_0 M_{e,i}^2} \nabla^2 \mathbf{M}_i + \frac{4A_{h,i}}{\mu_0 M_{e,i} M_{e,j}} \mathbf{M}_j + \frac{A_{nh,i}}{\mu_0 M_{e,i} M_{e,j}} \nabla^2 \mathbf{M}_j$$

The intra-lattice exchange stiffness A_i has the temperature dependence $A_i = A_i^0 m_{e,i}^2$, whilst the inter-lattice exchange stiffnesses have the temperature dependences $A_{h(nh),i} = A_{h(nh),i}^0 m_{e,i} m_{e,j}$. Finally, the terms $\mathbf{H}_{th,i}$ and $\boldsymbol{\eta}_{th,i}$ are stochastic quantities with zero spatial, vector components, and inter-lattice correlations, and whose components follow Gaussian distributions with zero mean and standard deviations given respectively by:

$$H_{th,i}^{std.} = \frac{1}{\alpha_{\perp,i}} \sqrt{\frac{2k_B T\left(\alpha_{\perp,i} - \alpha_{\parallel,i}\right)}{\gamma_i \mu_0 M_{S,i}^0 V\Delta t}}$$

$$\eta_{th,i}^{std.} = \sqrt{\frac{2k_B T \alpha_{\parallel,i} \gamma_i M_{S,i}^0}{\mu_0 V\Delta t}}$$

Here *V* is the stochastic computational cellsize volume, and Δ*t* is the stochastic time-step.

In Boris a mesh with the two-sublattice model may be added using the command:

addafmesh *meshname rectangle*

Default temperature dependences may be generated as for a ferromagnetic mesh with the command (same command as for a ferromagnetic mesh, hence the naming Curie):

curietemperature *value (meshname)*

Control of two-sublattice model meshes is exactly the same as for a ferromagnetic mesh, but the list of parameters and available modules is different (see modules and params). For a description of how the modules handle the two-sublattice model meshes see the chapter on Modules in the manual.

Some important parameters you need to control are:

1) Micromagnetic τ coupling factors. Set these using the tau command as:

   tau *tau11 tau22 tau12 tau21*

2) Atomic moments μ_A, μ_B. Set these using the atomicmoment command as:
   atomicmoment *mu1 mu2 meshname*

   Here *mu1* and *mu2* are the atomic moment values in units of the Bohr magneton, and meshname must be the name of an antiferromagnetic mesh. Note this command is also used to set the atomic moment for the LLB equation for a ferromagnetic mesh, which only takes a single *mu* value – this is the default behaviour, so giving the meshname parameter is important here.

3) Most magnetic parameters now have 2-sublattice values, so you can control them separately if needed. The two inter-lattice exchange stiffness coupling terms, *Ah* and *Anh*, only appear in two-sublattice model meshes.

When using the sLLB equation, by default the stochastic field generation time-step is the same as the time-step using for the equation evaluation. You may need to control this separately (set it to a larger value), and this is possible using the **setdtstoch** command – see the output of the **stochastic** command for an interactive display.
You can also set the stochastic cellsize value to be different than the magnetic cellsize value using the **scellsize** command.

Exercise 31.1 (Advanced)

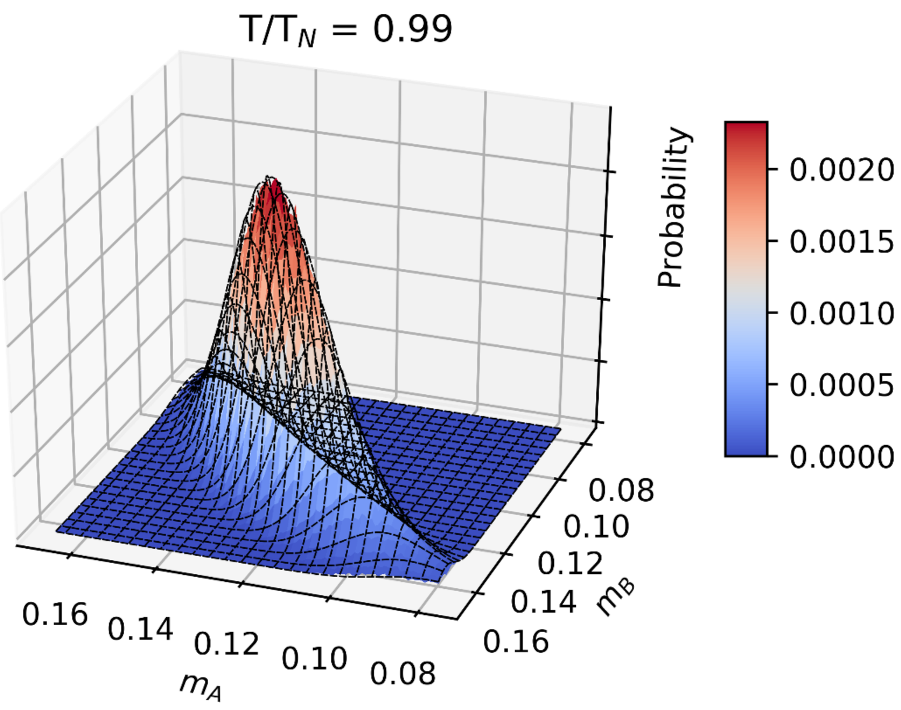
The two-sublattice sLLB equation produces a distribution of magnetization lengths on the two sub-lattices, m_A, m_B, which is expected to obey the following bi-variate Boltzmann probability distribution:

$$P_i(m_A, m_B) \propto m_i^2 \exp\left\{ -\frac{M_{S,i}^0 V}{4\mu_i m_{e,i} k_B T} \left[\frac{\left(m_i^2 - m_{e,i}^2\right)^2}{m_{e,i}} \frac{\left(\mu_i + 3\tau_{ij} k_B T_N \tilde{\chi}_{\parallel,j}\right)}{2\tilde{\chi}_{\parallel,i}} + \frac{\left(m_j^2 - m_{e,j}^2\right)}{m_{e,j}} 3\tau_{ij} k_B T_N m_i^2 \right] \right\}$$

$$(i, j = A, B, i \neq j)$$

Write a general-purpose Python script which tests the above equation against computed bi-variate probability distributions for any possible combination of T, T_N, τ_i, τ_{ij}, μ_i, and $M_{S,i}^0$ parameters. Test it for particular values (e.g. antiferromagnetic case).
Hint: there is a useful command built into Boris which computes a histogram for the two-sublattice model, namely ***dp_histogram2*** *– see help for this command (there's also a* ***dp_histogram*** *command which works for ferromagnetic meshes). See Figure 31.1 for a typical output you should obtain from your script.*

Figure 31.1 – Computed two-sublattice normalized magnetization length probability distribution at T/T$_N$ = 0.99 (colored surface) for an antiferromagnet in Exercise 31.1, compared with the bi-variate Boltzmann probability distribution prediction (wire-frame).



Exercise 31.2

Using the default antiferromagnetic mesh (**addafmesh**) in Boris, verify the Kittel formula for antiferromagnetic resonance is reproduced (see F. Keefer and C. Kittel, Phys. Rev. 85, 329 (1952)):

$$\frac{w}{\gamma} = H_0 + \sqrt{H_A(2H_E + H_A)}$$

Here H_A is the uniaxial anisotropy field along the bias field, $H_A = 2K_1/\mu_0 M_S$, where M_S is the magnetization length on a sub-lattice, and H_E is the Weiss exchange field, given by $H_E = 4A_h/\mu_0 M_S$.

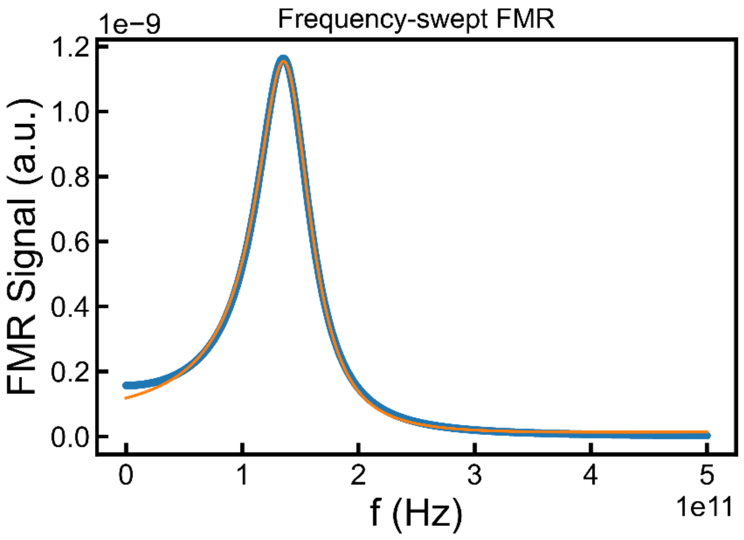
Hint: use the LLG equation, and adapt a previous exercise on frequency-swept FMR to simulate a frequency-swept FMR peak and obtain the resonance frequency. You should either use H_0 set to zero, or small bias field values as the above formula becomes inaccurate at large bias field values.

For the above exercise you should fit a Lorentz peak function with both symmetric and asymmetric components for a more accurate result:

$$f(x) = y_0 + S\frac{w + A(x - x_0)}{4(x - x_0)^2 + w^2}$$

Such a fitting procedure has already been built into Boris and can be accessed using the dp_**fitlorentz2** command.

Figure 31.2 – Antiferromagnetic resonance peak with fitted symmetric and asymmetric Lorentz peak function, verifying Kittel's formula in Exercise 31.2.



Tutorial 32 – Exchange Bias

The exchange bias field on a ferromagnetic layer from an antiferromagnet is given as:

$$\mathbf{H}_{ex} = \frac{J}{\mu_0 M_S t_F}\mathbf{m}_j$$

Here M_S and t_F are the saturation magnetization and thickness of the ferromagnetic layer, and $\mathbf{m}_j$ is the exchange bias field direction from the antiferromagnet. This effect may be modelled in Boris using the *surfexchange* module, since the bilinear surface exchange field is given as:

$$\mathbf{H}_i = \frac{J_1}{\mu_0 M_S t_F}\mathbf{m}_j$$

Here $\mathbf{m}_j$ is the magnetization direction on sub-lattice A of an interfacing antiferromagnetic mesh. Thus in order to model exchange bias in Boris you need an antiferromagnetic mesh (**addafmesh**), and a ferromagnetic mesh (**addmesh**) in contact with it, and they both need the *surfexchange* module enabled, remembering it is the top mesh (in order of z axis direction) which sets the *J1* value.

Exercise 32.1

Simulate the exchange bias effect in a Fe 2nm thin film using a generic antiferromagnetic material, 10 nm thick (use the **addafmesh** command to create a default antiferromagnetic mesh). Enable in-plane uniaxial anisotropy for the antiferromagnetic material (x axis), and set the antiferromagnetic sub-lattice A magnetization direction to result in a bias effect towards the +ve side.

You can add the Fe material from the materials database, and you should enable cubic anisotropy for it. You can use periodic boundary conditions in the xy plane, simulating an area of 320 × 320 nm^2. Set the J_1 surface exchange constant to 0.2 mJ/m^2. *Note: avoid using SDesc for this problem as it can be problematic, instead use LLGStatic with RKF45 and mxh of* 3×10^{-5}.

Tutorial 33 – Magneto-Elastic Effect and Elastodynamics Solver

The magneto-elastic effect may be simulated in Boris using the *melastic* module. The magneto-elastic effect can be included for a cubic crystal using a strain tensor. The strain tensor is given as:

$$\mathbf{S} = \begin{pmatrix} \varepsilon_{xx} & \varepsilon_{xy} & \varepsilon_{xz} \\ \varepsilon_{xy} & \varepsilon_{yy} & \varepsilon_{yz} \\ \varepsilon_{xz} & \varepsilon_{yz} & \varepsilon_{zz} \end{pmatrix}.$$

Here we define the diagonal strain vector as $\mathbf{S}_d = (\varepsilon_{xx}, \varepsilon_{yy}, \varepsilon_{zz})$, and off-diagonal strain vector as $\mathbf{S}_{od} = (\varepsilon_{yz}, \varepsilon_{xz}, \varepsilon_{xy})$. The strain tensor can have a spatial dependence, and currently needs to either be loaded from ovf2 files (strain computed with an external package), or alternatively a displacement vector field can be loaded (using ovf2 files, computed externally), and the strain tensor computed as:

$$\mathbf{S}_d = \left(\frac{\partial u_x}{\partial x}, \frac{\partial u_y}{\partial y}, \frac{\partial u_z}{\partial z} \right)$$
$$\mathbf{S}_{od} = \frac{1}{2}\left(\frac{\partial u_y}{\partial z} + \frac{\partial u_z}{\partial y}, \frac{\partial u_x}{\partial z} + \frac{\partial u_z}{\partial x}, \frac{\partial u_x}{\partial y} + \frac{\partial u_y}{\partial x} \right)$$

In the simplest case a uniform stress may be applied which results in a constant strain with zero off-diagonal terms. From the strain tensor, for a cubic crystal with orthogonal axes $\mathbf{e}_1$, $\mathbf{e}_2$, $\mathbf{e}_3$, and magneto-elastic constants B1, B2, we have the following diagonal and off-diagonal energy density terms:

$$\varepsilon_{mel,d} = B_1\left[(\mathbf{m}.\mathbf{e}_1)^2(\mathbf{S}_d.\mathbf{e}_1) + (\mathbf{m}.\mathbf{e}_2)^2(\mathbf{S}_d.\mathbf{e}_2) + (\mathbf{m}.\mathbf{e}_3)^2(\mathbf{S}_d.\mathbf{e}_3)\right]$$

$$\varepsilon_{mel,od} = 2B_2\left[(\mathbf{m}.\mathbf{e}_1)(\mathbf{m}.\mathbf{e}_2)(\mathbf{S}_{od}.\mathbf{e}_3) + (\mathbf{m}.\mathbf{e}_1)(\mathbf{m}.\mathbf{e}_3)(\mathbf{S}_{od}.\mathbf{e}_2) + (\mathbf{m}.\mathbf{e}_2)(\mathbf{m}.\mathbf{e}_3)(\mathbf{S}_d.\mathbf{e}_1)\right]$$

To set a uniform stress use the command (similar to the **setfield** command):

setstress *magnitude polar azimuthal (meshname)*

A uniform stress may also be set using the *Sunif* stage. When applying a uniform stress, **T**, the strain tensor is generated based on the material Young's modules and Poisson ratio as:

$$\mathbf{S} = \frac{1}{Y_m}\begin{pmatrix} 1 & -\nu & -\nu \\ -\nu & 1 & -\nu \\ -\nu & -\nu & 1 \end{pmatrix}\mathbf{T}$$

Young's moduls and Poisson ratio are available as material parameters as *Ym* and *Pr* respectively. The magneto-elastic constants are available as material parameters as *MEc* (2-component parameter for B1 and B2 respectively). The orthogonal axes $\mathbf{e}_1$, $\mathbf{e}_2$, $\mathbf{e}_3$ are set by the magento-crystalline anisotropy axes (ea1 is **e1**, ea2 is **e2** and $\mathbf{e}_3 = \mathbf{e}_1 \times \mathbf{e}_2$.

When applying a uniform stress the mesh origin (0, 0, 0) is a fixed point, and no shear strain or physical displacement is allowed. This means a positive stress value along the x axis results in elongation, whilst a negative stress value along the x axis results in compression.

You can run computations with a non-uniform strain, but in the current version this must be computed externally. There are two ways of setting a non-uniform strain. The simplest method is to compute the displacement vector map **u** externally and save it into a ovf2 file. This can then be loaded into the currently focused mesh using the command (must have *melastic* module enabled):

loadovf2disp *filename*

After the displacement map is loaded the strain tensor used for computations is obtained using the equations above.
You can also load the strain tensor directly, but this requires saving the diagonal and off-diagonal components in two separate ovf2 files in vector data format. The diagonal file will then contain vector data with strain components xx, yy, zz, and the off-diagonal file will contain vector data with strain components yz, xz, xy (in this order). The ovf2 files can then be loaded as:

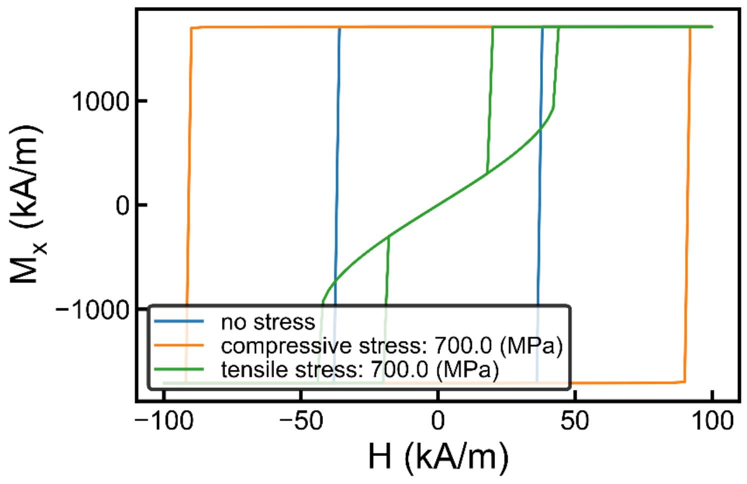
loadovf2strain *filename_diag filename_odiag*

With the *melastic* module enabled, there is a separate cellsize for the strain tensor, controlled using the **mcellsize** command, and this should be set before loading the externally computed strain or displacement.

Exercise 33.1

Simulate hysteresis loops for a 5 nm thick Fe thin film (found in materials database) with cubic anisotropy and *melastic* module enabled, along the x axis. Use a magneto-elastic coefficient B_1 = 8 MJ/m^3. Simulate three cases: i) no strain, ii) compressive stress along the x axis of 0.7 GPa, iii) tensile stress along the x axis of 0.7 GPa. Explain the differences between the 3 curves.

Figure 33.1 – Hysteresis loops computed in Exercise 33.1 for a Fe thin film, with and without mechanical stress.



It is also possible to compute strains in response to 1) external forces, 2) magnetostriction, 3) thermoelastic effect, as well as use of elastodynamics solver. In this case at least one fixed surface should be defined, where the mechanical displacement is zero:

surfacefix *(meshname) rect_or_face*

The easiest way is to specify an entire fixed face for a given mesh (typical use case), using string literals -*x*, *x*, -*y*, *y*, -*z*, *z*. For example **surfacefix** -*z* designates the entire bottom face as fixed. Alternatively a rectangle may be specified in absolute coordinates, but this should intersect with at least one mesh face. Multiple fixed surfaces may be specified.

An elastodynamics solver time-step also needs to be specified using the **seteldt** command. For stability this should satisfy the Courant, Friedrichs, Lewy (CFL) condition, and should also not exceed the magnetisation solver time-step:

$$\Delta t < 1/\left(v_{\mathrm{p}}\sqrt{\Delta x^{-2} + \Delta y^{-2} + \Delta z^{-2}}\right)$$

Here v_{P} is the elastic compressional wave velocity (v_{P} is typically 4000 to 6000 m/s), and Δx, Δy, Δz are the mechanical cell-size dimensions. Setting the elastodynamics solver time-step to zero disables it and keeps the currently calculated strain fixed. Elastic stiffness coefficients (*cC* material parameter) and mass density (*density*) should also be specified. There's also a mechanical damping coefficient, which results in mechanical energy dissipation (*mdamping* material parameter).

In addition to magneto-elastic coefficient, *MEc*, there are also corresponding coefficients for magnetostriction effects (*mMEc*) – in theory these are identical to *MEc*, but can be set to zero in order to disable magnetostriction. A linear thermoelastic expansion coefficient may also be specified (*thalpha*), which should be used in conjuction with the heat flow solver. For full details see the *melastic* module description in Modules.

Depending on the use case, it may also be necessary to reset the elastodynamics solver to initial conditions, and this can be achieved using the **resetelsolver** command.

Finally, external forces may be specified using the **surfacestress** command:

surfacestress *(meshname) rect_or_face vector_equation*

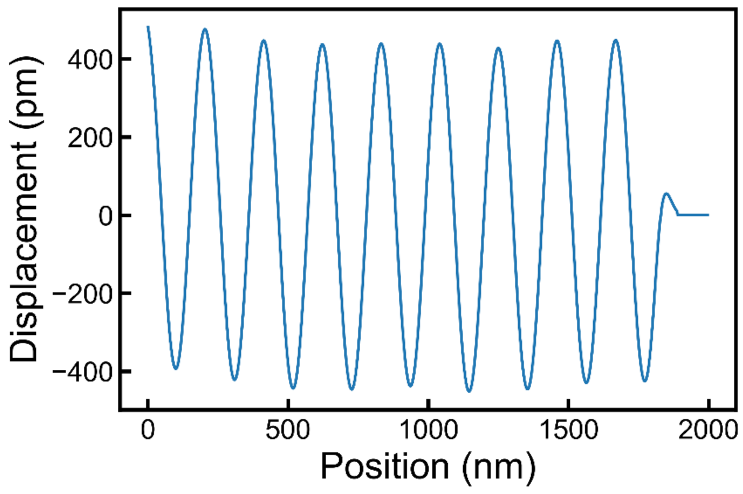
The same format as for the **surfacefix** command is used here, but additionally the external stimulus (units of N/m^2, i.e. a pressure) is specified using a vector text equation (3 components). The equation depends on *x* and *y* spatial coordinates, which are relative to the defined surface rectangle, and also on time *t*. See the User-Defined Text Equations section for details on text equations. Any number of external stimuli may be specified.

Exercise 33.2

Setup a surface acoustic wave (SAW) with 200 nm wavelength, in a Fe track, 2 μm long, 50 nm wide, and 40 nm thick. Assume the Fe track is on a stiff substrate, and generate the SAW using i) an external pressure of 1 GPa amplitude on the -*x* face, and ii) using interdigitated electrodes on the top surface with a 0.5 GPa shear pressure amplitude. Assume the compressional wave velocity is v_p = 4500 m/s for the default elastic coefficients (c_{11} = 300 GPa, c_{12} = 200 GPa, c_{44} = 50 GPa). Obtain the mechanical displacement profile after 5 ns. You may disable magnetostriction.

Notes: the SAW condition is $\lambda = v_p / f$, thus the external stimulus should have the form $F_0 \sin(2\pi v_{\|} t / \lambda)$. For interdigitated electrodes the electrodes spacing and width should be $\lambda/4$ with a π radians phase difference between them. It is also necessary to allow for energy dissipation otherwise waves will be reflected from the opposite end of the stimulus. This may be achieved by setting the mechanical damping parameter with an absorbing boundary spatial variation: use the *abl_tanh* generator so the mechanical damping reaches a value of 10^{16} kg/m^{3}s at one end, but has a relatively small value elsewhere (e.g. 10^{10} kg/m^{3}s), so SAW attenuation is confined to one end only.

Figure 33.2 – Longitudinal mechanical displacement profile for the 200 nm SAW generated in Exercise 33.2. The SAW is absorbed at the far end to prevent reflections.



Shapes and Regions

There are a number of built-in elementary shapes (rectangle, disk, triangle, ellipsoid, torus, pyramid, tetrahedron, cone) which can be combined to form composite shapes. These shapes can then be used as 1) physical objects in meshes, 2) as masks to set magnetization directions, and 3) as regions to set material parameter values. The shapes also have transformation functions, including scaling, translation, rotation, array generation.

Elementary shapes can be generated using the console commands: **shape_rect**, **shape_disk**, **shape_triangle**, **shape_tetrahedron**, **shape_pyramid**, **shape_cone**, **shape_ellipsoid**, **shape_torus**. These functions use modifiers, including rotation, array generation, and method (add, subtract), set using the commands: **shape_rotation**, **shape_repetitions**, **shape_displacement**, **shape_method**.

A more powerful approach to working with shapes consists in using Python scripted simulations. The NetSocks.py module defines a class called Shape. This class can be used to define the above elementary shapes, which can then be combined using arithmetic operators (+,-) to form composite shapes. The shapes can then be physically set in a mesh, used to set a material parameter spatial variation by effectively defining a region, or used as a mask to set magnetization directions. They can also be transformed using methods defined in the Shape class, as detailed in the tables below.

Table SR.I – Elementary shapes methods with given dimensions (m) and center position (m). Overloaded operators are + and -, and the result is a composite shape.

Shape class: elementary shapes methods	
disk(dimensions, position)	*pyramid(dimensions, position)*
rect(dimensions, position)	*cone(dimensions, position)*
triangle(dimensions, position)	*ellipsoid(dimensions, position)*
tetrahedron(dimensions, position)	*torus(dimensions, position)*

Table SR.II – Shape transformation methods. Rotations are specified using YXZ Tait-Bryan angles in degrees.

Shape class: transformation methods for both composite and elementary shapes	
setdimensions(dimensions)	Set dimensions of first elementary shape contained, and scale everything else in proportion.
scale(scalefactors)	Scale dimensions of shape.
setposition(position)	Set position of shape, defined by the position of the first elementary shape contained.
move(positionshift)	Translate position of shape.
setrotation(rotation)	Set rotation of shape, around shape position as defined by the first elementary shape contained.
rotate(rotation)	Rotate shape around shape position as defined by the first elementary shape contained.
setrepetitions(repetitions, displacement)	Set number of repetitions and displacement of shape for generating array.

Once a shape has been defined, we can use 1) **shape_set** command to set a physical shape in the focused mesh, 2) **shape_setangle** command to set magnetization angle in the defined shape, and 3) **shape_setparam** command to set the material parameter spatial variation scaling in the region defined by the shape.

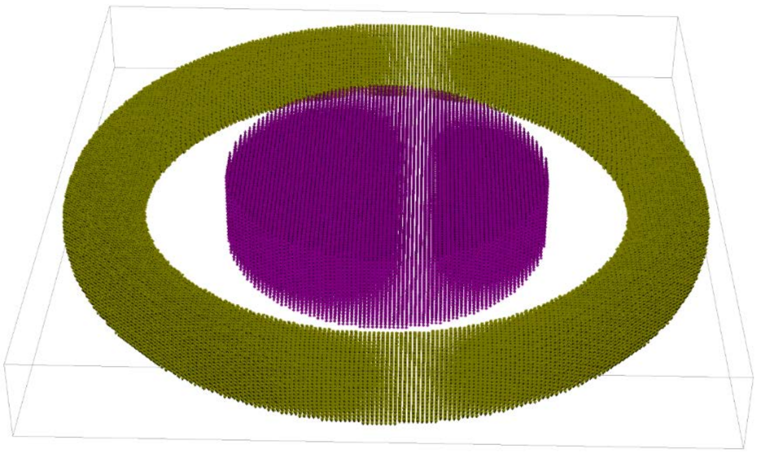
The following contains examples, with Python scripts included in the Examples/Shapes and Regions directory.

```
from NetSocks import NSClient, Shape

ns = NSClient(); ns.configure(True)

########################################

l, w, t = 800e-9, 800e-9, 100e-9

FM = ns.Ferromagnet([l, w, t], [5e-9])
ns.delrect()

tor = Shape.torus([l, w, t])
disk = Shape.disk([l/2, w/2, t])

#set disk centred
FM.shape_set(disk.move([l/2, w/2, t/2]))
```

```
#set torus centred
FM.shape_set(tor.move([l/2, w/2, t/2]))

#set magnetization angles of defined shapes
FM.shape_setangle(disk, [180, 0])
FM.shape_setangle(tor, [90, 90])
```

The above example generates a torus and a disk, shown in Figure SR.1.

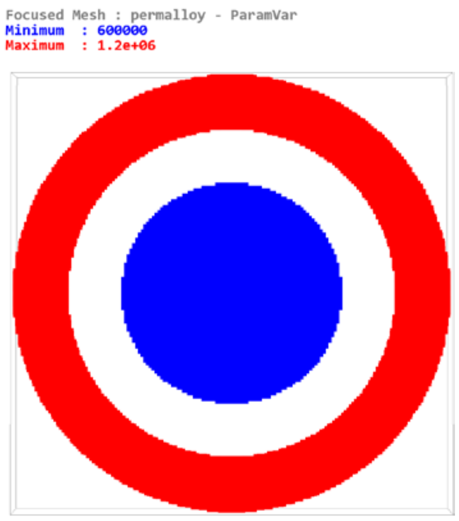
Figure SR.1 – Physical shapes generated in a magnetic mesh.



The following sets the M_S material parameter value in the region defined by the torus and disk shapes, with the result shown in Figure SR.2.

```
#set material parameter values of regions defined by shapes
Ms = FM.param.Ms.setparam()
FM.param.Ms.shape_setparam(disk, 600e3 / Ms)
FM.param.Ms.shape_setparam(tor, 1200e3 / Ms)
```

Figure SR.2 – Material parameter (M_S) spatial variation set in defined regions.

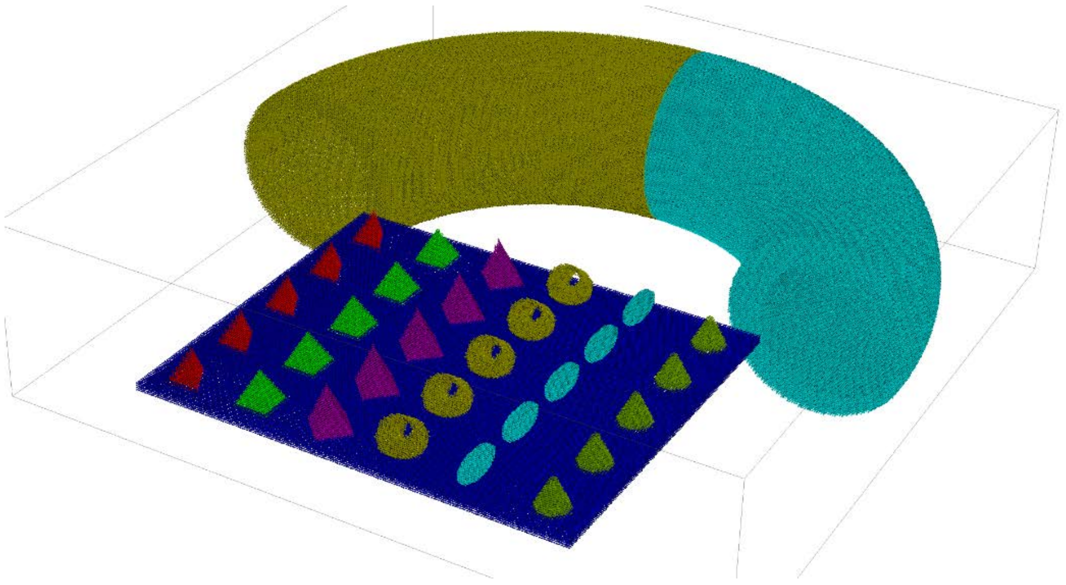


The following exemplifies the use of composite shapes and transformation functions, with the result shown in Figure SR.3.

```
from NetSocks import NSClient, Shape

ns = NSClient(); ns.configure(True)

########################################

l, w, t = 800e-9, 800e-9, 200e-9
FM = ns.Ferromagnet([l, w, t], [4e-9])
ns.delrect()

#define hollow half-torus
tor1 = Shape.torus([l, w, t])
tor2 = Shape.torus([l - t/2 + t*0.75/2, w - t/2 + t*0.75/2, t*0.75])
htor = tor1 - tor2 - Shape.rect([l, w/2, t]).move([0, -w/4, 0])

#define masks for left and right sides of half-torus
left = htor - Shape.rect([l/2, w/2, t]).move([l/4, w/4, 0])
right = htor - Shape.rect([l/2, w/2, t]).move([-l/4, w/4, 0])

#set shape and magnetization directions for left and right sides
FM.shape_set(htor.move([l/2, w/2, t/2]))
FM.shape_setangle(left.move([l/2, w/2, t/2]), [90, 90])
FM.shape_setangle(right.move([l/2, w/2, t/2]), [90, 270])

#rectangular base
base = Shape.rect([l * 0.6, w*0.5, 10e-9])
FM.shape_set(base.move([l/2, w/4, t/3]))

#repeated tetrahedra
d = 40e-9
tetra = Shape.tetrahedron([d, d, d])
tetra.setrepetitions([1, w / (4*d), 1], [0, d * 2, 0])
FM.shape_set(tetra.move([l * 0.2 + d, d, t/3 + d/2]))
FM.shape_setangle(tetra, [90, 0])

#repeated pyramids
pyramid = Shape.pyramid([d, d, d])
pyramid.setrepetitions([1, w / (4*d), 1], [0, d * 2, 0])
FM.shape_set(pyramid.move([l * 0.2 + 3*d, d, t/3 + d/2]))
FM.shape_setangle(pyramid, [0, 0])

#repeated rotated triangles
triangle = Shape.triangle([d, d, d])
triangle.setrepetitions([1, w / (4*d), 1], [0, d * 2, 0])
triangle.rotate([0, 90, 45])
FM.shape_set(triangle.move([l * 0.2 + 5*d, d, t/3 + d/2]))
FM.shape_setangle(triangle, [180, 0])

#repeated rotated excentric tubes
tube = Shape.disk([d, d, d]) - Shape.disk([d/2, d/2, d]).move([d/5, 0, 0])
tube.setrepetitions([1, w / (4*d), 1], [0, d * 2, 0])
tube.rotate([-30, 45, 45])
FM.shape_set(tube.move([l * 0.2 + 7*d, d, t/3 + d/2]))
FM.shape_setangle(tube, [90, 90])
```

```
#repeated ellipsoids
ellipsoid = Shape.ellipsoid([d/2, d*1.5, d*0.75])
ellipsoid.setrepetitions([1, w / (4*d), 1], [0, d * 2, 0])
FM.shape_set(ellipsoid.move([l * 0.2 + 9*d, d, t/3 + d/2]))
FM.shape_setangle(ellipsoid, [90, 270])

#repeated cones
cone = Shape.cone([d, d, d])
cone.setrepetitions([1, w / (4*d), 1], [0, d * 2, 0])
FM.shape_set(cone.move([l * 0.2 + 11*d, d, t/3 + d/2]))
FM.shape_setangle(cone, [45, 45])

ns.displaydetail(2e-9)
```

Figure SR.3 – Examples of elementary and composite physical shapes generated in a magnetic mesh.

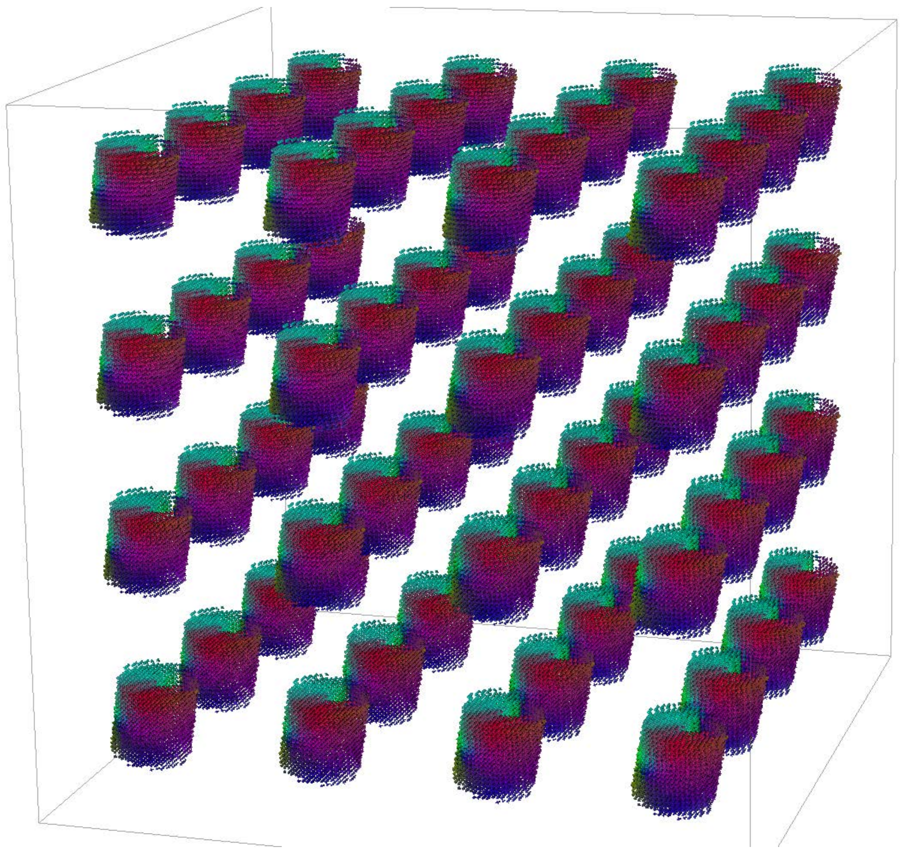


As a final example we generate a 3D array of a composite, rotated and scaled shape, with the result shown in Figure SR.4.

```
from NetSocks import NSClient, Shape

ns = NSClient(); ns.configure(True)

########################################

l, w, t = 800e-9, 800e-9, 800e-9
FM = ns.Ferromagnet([l, w, t], [5e-9])
ns.delrect()

disk = Shape.disk([300e-9, 300e-9, 300e-9])
triangle = Shape.triangle([150e-9, 150e-9, 300e-9])
shape = disk - triangle.move([60e-9, 60e-9, 0])
```

```
shape.rotate([10, 10, 10])
shape.scale([0.3, 0.3, 0.3])
shape.setrepetitions([4, 4, 4], [200e-9, 200e-9, 200e-9])

FM.shape_set(shape.setposition([100e-9, 100e-9, 100e-9]))
```

Figure SR.4 – Example 3D array of composite and rotated physical shapes.



User-Defined Text Equations

Arbitrary text equations may be supplied by the user using fundamental functions, a number of special functions, and mathematical operators as detailed below. These equations are evaluated efficiently at run-time every iteration. Text equations use a number of reserved variables depending on the context, including *x*, *y*, *z* to define spatial variation, *t* to define temporal dependence, *T* to define temperature dependence, and also use a number of reserved constants as listed below. Both scalar and vector text equations may be supplied as appropriate. The contexts in which text equations may be used are:

1. Setting stage values

The available stage types are: i) *Hequation* (set external field using a vector text equation), ii) *Vequation* (set electrode potential drop using a scalar text equation), iii) *Iequation* (set ground electrode current using a scalar text equation), iv) *Tequation* (set mesh base temperature using a scalar text equation – applicable when heat module disabled), v) *Qequation* (set heat source in heat equation using a scalar equation).

Reserved variables:

x, *y*, *z* (spatial coordinates in meters, relative to mesh where used), *t* (stage time in seconds).

Defined constants:

Lx, *Ly*, *Lz* (mesh dimensions in meters), *Tb* (mesh base temperature), *Ss* (stage step).

Example: *0, 0, He * sinc(kc*(x-Lx/2))*sinc(kc*(y-Ly/2))*sinc(2*PI*fc*(t-t0)),*

The above example is used with *Hequation* (vector equation), and applies a spatial and temporal sinc pulse at the centre of the mesh with time centre at t0, for the z field component. The constants *He*, *kc*, *fc*, *t0* are defined by the user – see below.

2. Parameter temperature dependence

Any material parameter for which a temperature dependence is allowed (see output of **paramstemp** command), may be assigned a temperature dependence defined using a text equation. To set this either use the interactive console, or use the following command:

```
setparamtempequation (meshname) paramname text_equation
```

In Python scripts you can use *ns.setparamtempequation(paramname, text_equation)*, or for a specific mesh as *ns.setparamtempequation(meshname, paramname, text_equation)*.

Reserved variables:

T (temperature, either mesh base temperature if *heat* module not enabled, or the local temperature if *heat* module is enabled – in the latter case the temperature can be non-uniform and the parameter value is evaluated according to the local computational cell temperature).

Defined constants:

Tb (mesh base temperature), *Tc* (set mesh Curie (or Néel) temperature).

Example: *me(T/Tc)^2*

The above example applies a squared Curie-Weiss scaling relation (me), for temperatures ranging from 0 to *Tc*.

3. Parameter spatial variation

Any material parameter for which a spatial variation dependence is allowed (see output of **paramsvar** command), may be assigned a spatial and temporal dependence defined using a text equation. To set this either use the interactive console, or use the following command:

```
setparamvar (meshname) paramname equation text_equation
```

Note in the above command the field "*equation*" specifies the type of generator used to generate a spatial variation and must be inputted literally as above.

Reserved variables:

x, *y*, *z* (spatial coordinates in meters, relative to mesh where used), *t* (stage time in seconds).

Defined constants:

Lx, *Ly*, *Lz* (mesh dimensions in meters).

Example: *exp(-(x-Lx/2)^2/Sx) * exp(-(y-Ly/2)^2/Sy) / (exp(0)^2)*

The above example applies a Gaussian spatial variation scaling in the xy plane at the centre of the mesh.

Functions

Allowed functions in text equations are given below.

Function Name	**Description**
sin	Fundamental sin function.
sinc	sinc(x) = sin(x) / x
cos	Fundamental cos function.
tan	tan(x) = sin(x) / cos(x)
sinh	sinh(x) = [exp(x) - exp(-x)] / 2
cosh	cosh(x) = [exp(x) + exp(-x)] / 2
tanh	tanh(x) = sinh(x) / cosh(x)
sqrt	Square root.
exp	Fundamental exp function.
asin	Inverse sin function.
acos	Inverse cos function.
atan	Inverse tan function.
asinh	Inverse hyperbolic sin function.
acosh	Inverse hyperbolic cos function.
atanh	Inverse hyperbolic tan function.
ln	Natural logarithm.
log	Logarithm base 10.
abs	Modulus function.
sgn	Sign function.
floor	Floor function (round down).
ceil	Ceil function (round up).

round	Round function (round to nearest integer).
step	Step function: step(x) = 0 for x < 0, = 1 for x>= 0.
swav	Square wave function with period 2π, s.t. swav(0+) = +1, swav(π+) = -1.
twav	Triangular wave function with period 2π, s.t. twav(0) = +1, twav(π) = -1.
me	Normalized Curie-Weiss law.
chi	Normalized relative longitudinal susceptibility.
me1	Normalized Curie-Weiss law for sub-lattice A.
me2	Normalized Curie-Weiss law for sub-lattice B.
chi1	Normalized relative longitudinal susceptibility for sub-lattice A.
chi2	Normalized relative longitudinal susceptibility for sub-lattice B.
alpha1	Transverse damping temperature dependence.
alpha2	Longitudinal damping temperature dependence.

Any nested combination of functions is allowed.

Special equation expanders

sum

This is the sum function, with format given as: *sum(i;low;high;func(<i>))*.
Thus *func*(*i*) is summed for *i* ranging from integers *low* to *high* inclusive; note the format *i* must appear in *func*, namely <*i*>. The variable named *i* can be changed to a different string literal but must not clash with any equation variables, or reserved names. For example the expression:

*sum(j;0;10;sin(2*PI*<j>*x))*, evaluates to: $\sum_{j=0}^{10} \sin(2\pi j x)$

Nested *sum* functions are also allowed, and may be combined with any of the reserved function names in the table above. Note that using excessive limits (e.g. integer range over 1000) in the sum function will result in very slow evaluation times and should be avoided, especially if the evaluation is to be performed in every computational mesh cell.

Allowed mathematical operators

Operator Symbol (in precedence order)	**Description**
^	Exponentiation operator.
/	Division operator.
*	Multiplication operator.
-	Subtraction operator.
+	Addition operator.

All functions, variables, and constants must be linked by an operator, assumed multiplication is not allowed, e.g. *2*sin(x)* is a valid equation, *2sin(x)* is not a valid equation in the current program version.

Bracketing

Only round brackets are allowed, (,), which must appear in equal numbers.

Numerical constants

Numbers may be specified in floating point or scientific format.

Reserved constants

A number of pre-defined numerical constants are available. Since the names in the table below are reserved, they must not clash with any user defined constants.

Symbol	**Description**
PI	3.14159265358979323846264338 33
mu0	Free space permeability: 4*PI*10^{-7} (N/A^2)
muB	Bohr magneton: 9.27400968e-24 (Am^2)
ec	Electron charge: 1.60217662e-19 (C)
hbar	Reduced Planck constant: 1.054571817e-34 (m^2kg/s)
kB	Boltzmann constant: 1.3806488e-23 (m^2kg/s^2K)
gamma	Gyromagnetic ratio modulus: 2.212761569e5 (m/As)

User defined constants

Any number of user defined constants, named using alphanumeric strings, may be given, with the restriction they must not clash with any of the reserved names in the tables above, or equation variables. Equation constants may be given as:

```
equationconstants name value
```

Other related commands are **delequationconstant**, and **clearequationconstants**.

Vector and scalar equations

Scalar equation have a single component which must conform to the above rules. Vector equations with 3 components are allowed where appropriate, and must be specified using comma separators as: *component1*, *component2*, *component3*. Here the three components are scalar equations.

Working with OVF2 Files

Data may be loaded and saved using the OVF2 file format introduced in OOMMF. This allows setting magnetic shapes programmatically, as well as importing and exporting data to other software which support this file format.

OVF2 files may also be used to save mesh data numerically, not only for magnetization, but for any mesh quantity which may be displayed on screen, including material parameter spatial variation. The latter may also be set programmatically using OVF2 files, via the ovf2 spatial variation generator (see below).

OVF2 files allow scalar or vector formats, and both are handled in Boris. Data may be saved in natural text format (*text*), single precision binary (*bin4*), or double precision binary (*bin8*).

Commands which handle OVF2 files

```
loadovf2mag (meshname) (renormalize_value) (directory/)filename
```

Load an OOMMF-style OVF 2.0 file containing magnetization data, into the currently focused mesh or given *meshname* (which must be ferromagnetic), mapping the data to the current mesh dimensions. By default the loaded data will not be renormalized: renormalize_value = 0. If a value is specified for renormalize_value, the loaded data will be renormalized to it (e.g. this would be an Ms value).

```
saveovf2mag (meshname) (n) (data_type) (directory/)filename
```

Save an OOMMF-style OVF 2.0 file containing magnetization data from the currently focused mesh or given *meshname* (which must be ferromagnetic). You can normalize the data to Ms0 value by specifying the *n* flag (e.g. **saveovf2mag** *n filename*) - by default the data is not normalized. You can specify the data type as *data_type* = *bin4* (single precision 4 bytes per float), *data_type* = *bin8* (double precision 8 bytes per float), or *data_type* = *text*. By default *bin8* is used.

```
saveovf2 (meshname) (data_type) (directory/)filename
```

Save an OOMMF-style OVF 2.0 file containing data from the currently focused mesh or given *meshname*. You can specify the data type as *data_type* = *bin4* (single precision 4 bytes per float), *data_type* = *bin8* (double precision 8 bytes per float), or *data_type* = *text*. By default *bin8* is used.

```
loadovf2temp (meshname) (directory/)filename
```

Load a temperature distribution into the currently focused mesh or given *meshname*, which must have the *heat* module enabled. You can disable the heat solver by setting **setheatdt** 0.

```
loadovf2curr (meshname) (directory/)filename
```

Load a current density distribution into the currently focused mesh or given *meshname*, which must have the transport module enabled. This should normally be used with transport solver iteration disabled (but *transport* module enabled) so the loaded current distribution remains constant: **disabletransportsolver** 1.

```
loadovf2disp (meshname) (directory/)filename
```

Load mechanical displacement data into the currently focused mesh or given *meshname*, which should have the *melastic* module enabled. A strain will be computed internally.

```
loadovf2strain (meshname) (directory/)filename_diag filename_odiag
```

Load strain tensor data into the currently focused mesh or given *meshname*, which should have the *melastic* module enabled. Two vector files are required, one for the diagonal strain tensor components, and the other for the off-diagonal components, where a symmetric strain tensor for a cubic crystal is assumed.

```
loadovf2field (meshname) (directory/)filename
```

Load magnetic field into the currently focused mesh or given *meshname*, which should have the *Zeeman* module enabled. Alternatively. if meshname is *supermesh*, load a global field with absolute rectangle coordinates and its own discretization.

```
setparamvar (meshname) paramname ovf2 filename
```

Set the named parameter, *paramname*, spatial dependence for the named mesh, *meshname*, using an ovf2 file with *filename*, located in current working directory. The type of data in the OVF2 file must match the expected format of the parameter (scalar or vector). Once you've loaded the ovf2 file you can display the set spatial variation by selecting the *ParamVar* option under **display**, and clicking on the required parameter under **paramsvar**.

The provided NetSocks.py module used for Python scripts, provides a convenient method to handle OVF2 files: *Write_OVF2*, which allows writing a Python list into an OVF2 file. An example of programmatically setting a magnetic shape using an OVF2 file generated in a Python script is given in Tutorial 0.

```
saveovf2param (meshname) (data_type) paramname (directory/)filename
```

Save an OOMMF-style OVF 2.0 file containing the named parameter spatial variation data from the named mesh (currently focused mesh if not specified). You can specify the data type as *data_type* = *bin4* (single precision 4 bytes per float), *data_type* = *bin8* (double precision 8 bytes per float), or *data_type* = *text*. By default *bin8* is used.

Materials Database

Material definitions can be saved in a materials database. This includes base material parameter values. The default database is called BorisMDB.txt. You can see this using the command:

materialsdatabase

The default materials database can be updated from a shared database stored on a server. The shared database can be seen at: https://boris-spintronics.uk/online-materials-database. To update your local BorisMDB database using the latest material parameter definitions, use the command:

updatemdb

You can also switch to an alternative custom database using the **materialsdatabase** command. To add a new computational mesh with given material parameters you can use the **addmaterial** command. The type of material will determine the type of computational mesh generated. For example the *ferromagnetic* type will generate a computational mesh with LLG/LLB solvers enabled. The *conductor* mesh type will generate a computational mesh with only the transport and heat solvers enabled, while the *insulator* mesh will only have the heat solver enabled (e.g. a substrate material).

Users can also send in entries to be added to the centrally stored database. This can be done using the **requestmdbsync** command. Before sending entries, you must properly format the material entry. The procedure is described as follows.

1. Add a new entry to your local BorisMDB file.

   Suppose you want to enter a new ferromagnetic material. First create a ferromagnetic mesh in Boris (**addmesh**), and set as many parameter values as possible. Next, add the entry to your local BorisMDB file using:
   addmdbentry *meshname (materialname)*

Here meshname is the name of the mesh as it appears in Boris (the one you created using **addmesh**), and *materialname* is the name of the material, or entry, you want to create, if different.

2. Edit the material description fields in BorisMDB.txt

   There are 5 description fields for the entry:

   Name, Formula, Type, Description, Contributor

   Name is the name of your new material entry, which should not already be in the shared online database. This should already be filled.

   Formula is the symbolic formula for the material. Make sure to fill this.

   Type is already filled for you, and is the type of computational mesh for which the material applies.

   Description should have a very brief description for the material entry, with any useful information. Make sure to fill this.

   Contributor is the name of the entry contributor; leave as N/A if you don't want to specify this.

   There is another field called *State*. This specifies if the entry was taken from the online database (SHARED) or if it's a new user-created entry (LOCAL). You don't need to change this.

3. Set references for the parameters

   After each parameter there is a column called DOI. This must hold a DOI reference for where the material parameter value was taken from, or derived. You can leave it as N/A only if not applicable, but in most cases should be properly referenced.

4. Unspecified material parameters

   If you cannot reasonably give an entry for a material parameter value then override it as “N/A”.

5. Send in the entry

   After properly formatting the entry, you can upload it to a holding database using the **requestmdbsync** command:

   requestmdbsync *materialname (email)*

   Materialname is the name of the material you’ve just created. If you specify an email address you will receive feedback about whether the entry was added to the shared database or not – the entry will be verified for validity before being added to the online materials database. Once entered there, it will be visible at https://boris-spintronics.uk/online-materials-database, and other users can update their databases with it.

Meshes

Boris uses cell-centred finite difference discretization. In the simplest case a single magnetic mesh may be used, which occupies a rectangular space. Object shapes may be defined inside the mesh, however these must always be contained within the reserved mesh rectangular space. The size of the mesh rectangle is set using the **meshrect** command as:

```
meshrect (meshname) sx sy sz ex ey ez
```

The coordinates *sx*, *sy*, *sz*, *ex*, *ey*, *ez* are the Cartesian mesh rectangle start and end coordinates (units m). In the simplest case where a single mesh is used, only the end coordinates need be provided, as the start is assumed to be (0, 0, 0):

```
meshrect ex ey ez
```

Boris also allows multiple meshes to be used, each with its own independent discretization and independent mesh rectangle. In this case meshes are addressed using their individual names, i.e. *meshname*. This is a common pattern with most commands: for single-mesh simulations the *meshname* parameter doesn't need to be provided as it's optional; but should be specified for multi-mesh simulations.

The magnetic discretization, or cellsize, can be set using:

```
cellsize (meshname) cx cy cz
```

Here *cx*, *cy*, *cz* are cellsize values (unit m). The mesh rectangle must be subdivided by the cellsize into an integer number of cells. If the set cellsize does not conform to this requirement it will be automatically adjusted to the nearest values which result in a correct discretization.

There are several types of meshes: ferromagnetic meshes (**FM**), 2-sublattice magnetic meshes, e.g. ferrimagnetic, antiferromagnetic, or binary alloys (**AFM**), atomistic spin cubic meshes (**ASC**), electrical conductor meshes (**C**), electrical insulator meshes (**I**), and fixed magnetic dipole meshes (**D**).

Micromagnetic meshes (**FM**, **AFM**) have a set micromagnetic dynamics equation – see Differential Equations. Atomistic meshes (**ASC**) have a set atomistic dynamics equation –

see Differential Equations. Other types of meshes do not have a magnetic dynamics equations set. Electrical conductor meshes (**C**) allow electron charge and spin transport computations (*transport* module), and heat flow computations (*heat* module), while electrical insulator meshes (**I**) only allow heat flow computations. Magnetic dipole meshes (**D**) define a fixed magnetic dipole for the purposes of generating a magnetic stray field. Advanced simulations allows any combinations of these mesh types, e.g. multilayered devices which might have ferromagnetic, antiferromagnetic, non-magnetic spacer layers, and substrate materials included.

The different mesh types also differ in their available computational modules – see Modules.

When multiple meshes are defined, they are contained in a *supermesh* (**S**). The *supermesh* has its own modules which work across multiple meshes – see Modules. For example these are the *sdemag* (supermesh magnetostatic field module, which allows computing the demagnetizing field from any number of meshes), *strayfield* (computes the stray field from dipole meshes, to be included in magnetic meshes), and *Oersted* modules (computes the Oersted field from all meshes with a current density defined, and includes it in all magnetic meshes). The *supermesh* rectangle is automatically set as the smallest rectangle which encompasses all magnetic meshes.

Ferromagnetic meshes (**FM**):

To erase all meshes and set a single **FM** mesh:

```
setmesh name rectangle
```

To add another **FM** mesh:

```
addmesh name rectangle
```

Two-sublattice meshes (**AFM**):

To erase all meshes and set a single **AFM** mesh:

```
setafmesh name rectangle
```

To add another **AFM** mesh:

```
addafmesh name rectangle
```

Atomistic simple-cubic meshes (**ASC**):

To erase all meshes and set a single **ASC** mesh:

```
setameshcubic name rectangle
```

To add another **ASC** mesh:

```
addameshcubic name rectangle
```

Electrical conductor meshes (**C**):

To add a **C** mesh:

```
addconductor name rectangle
```

Electrical insulator meshes (**I**):

To add a **I** mesh:

```
addinsulator name rectangle
```

Magnetic dipole meshes (**D**):

To add a **D** mesh:

```
adddipole name rectangle
```

Materials

A material may be added from the materials database, which contains a number of defined materials parameters, and also defines the type of mesh to generate. See Materials Database for further details.

To erase all meshes and set a single material from the materials database (*name* must match a material name in the materials database):

```
addmaterial name rectangle
```

To add another mesh from the materials database (*name* must match a material name in the materials database):

```
setmaterial name rectangle
```

Cellsizes

There are different mesh discretizations for each of their physical properties (magnetic, electric, thermal, mechanical), even within the same mesh.

Magnetic

```
cellsize (meshname) cx cy cz
```

This affects the discretization of all magnetic quantities.

Electric

```
ecellsize (meshname) cx cy cz
```

This affects the discretization of quantities related to charge and spin transport in the *transport* module.

Thermal

```
tcellsize (meshname) cx cy cz
```

This affects the discretization of quantities related to heat flow in the *heat* module.

Elastic

```
mcellsize (meshname) cx cy cz
```

This affects the discretization of quantities related to mechanical elastic properties in the *melastic* module.

Supermesh magnetic cellsize

```
fmscellsize cx cy cz
```

This is used by the *sdemag* module with **multiconvolution** 0 (not recommended in general).

Supermesh electric cellsize

```
escellsize cx cy cz
```

This is used by the *Oersted* module – when computing the Oersted field the current density from individual meshes is transferred to the electric supermesh at this discretization. The Oersted field is then also computed at this discretization, and finally transferred back to individual magnetic meshes.

Mesh Objects in Python Scripts

A higher level, and recommend, approach to working with meshes in Python scripts, is to create them as objects using respective classes. Any commands which take a *meshname* as a parameter can then be accessed directly from the mesh object created. Also see Material Parameters section on accessing parameters through a mesh object, and Output Data section on using mesh objects for setting output data.

When creating a mesh object, a rectangle and a primary quantity cellsize must be given. For magnetic meshes the primary cellsize is for magnetic properties discretization, for conductor meshes the primary cellsize is for electrical conduction discretization, and for insulator meshes the primary cellsize is for temperature discretization.

The examples below show how different types of meshes may be created in a Python script. You can create any number of mesh objects in this way, and the first one in a code context erases all existing meshes (e.g. erases the existing default *permalloy* mesh).

Ferromagnetic meshes (**FM**):

```
FM = ns.Ferromagnet(rect, cellsize)
```

Use the created object to access commands which take *meshname* as a parameter, e.g. to set the electrical cellsize also:

```
FM.ecellsize(ecellsize)
```

Two-sublattice meshes (**AFM**):

```
AFM = ns.AntiFerromagnet(rect, cellsize)
```

Atomistic simple-cubic meshes (**ASC**):

```
ASC = ns.Atomistic(rect, cellsize)
```

Electrical conductor meshes (**C**):

```
NM = ns.Conductor(rect, cellsize)
```

Electrical insulator meshes (**I**):

```
I = ns.Insulator(rect, cellsize)
```

Magnetic dipole meshes (**D**):

```
Dipole = ns.Dipole(rect, cellsize)
```

Materials

Depending on the mesh object type, different options will be visible (e.g. material parameters, output data, display options, etc.). You can also use the materials database to create a mesh, and the mesh object type will be determined directly by the type of material created.

```
#This creates a Ferromagnet mesh with material definitions of Co
Co1 = ns.Material('Co', rect, cellsize)

#This creates a Conductor mesh with material definitions of Pt
Pt1 = ns.Material('Pt', rect, cellsize)
```

Differential Equations

This section outlines the magnetization dynamics equations solved – to see a list in the console use the **ode** command.

To set an equation to solve and evaluation method use the **setode** command as **setode** *equation evaluation*, e.g. **setode** *LLG-STT RK4*:

```
setode equation evaluation
```

For fixed time step evaluation methods you can set the time step as **setdt** *value*. For adaptive time step evaluation methods this sets the starting value:

```
setdt value
```

To fine tune the adaptive time step controller the **astepctrl** command may be used, but should not normally be necessary.

For a list of available equations and evaluation methods see below. Parameters entering the equations are:

Symbol (MM units) (A units)	g_{rel}	α	M_s (A/m)	P	β	$g_{rel,i}$	α_i	$M_{S,i}$ (A/m)	μ_S (μ_B)
Param. (dim.)	grel (1)	damping (1)	Ms (1)	P (1)	beta (1)	grel_AFM (2)	damping_AFM (2)	Ms_AFM (2)	mu_s (1)

A number of evaluation methods are available for the magnetization dynamics equations. These are fixed step methods Euler (*Euler* – 1st order), trapezoidal Euler (*TEuler* – 2nd order) and Runge-Kuta (*RK4* – 4th order). Adaptive time-step methods are the adaptive Heun (*AHeun* – 2nd order), the multi-step Adams-Bashforth-Moulton (*ABM* – 2nd order), Runge-Kutta-Bogacki-Shampine (*RK23* – 3rd order with embedded 2nd order error estimator), Runge-Kutta-Fehlberg (*RKF45* – 4th order with embedded 5th order error estimator), Runge-Kutta-Cash-Karp (*RKCK45* – 4th order with embedded 5th order error estimator), Runge-Kutta-Dormand-Prince (*RKDP54* – 5th order with embedded 4th order error estimator), and Runge-Kutta-Fehlberg (*RKF56* – 5th order with embedded 6th order error estimator) For static problems a steepest descent solver is available, *SDesc*, using Barzilai-Borwein stepsize selection formulas. For further details see [S. Lepadatu, IEEE Trans. Mag. 58, 1 (2022)].

LLG – Landau-Lifshitz-Gilbert (**FM**)

The normalized LLG equation (**m** = **M** / M$_S$) in implicit form is given by:

$$\frac{\partial \mathbf{m}}{\partial t} = -\gamma \mathbf{m} \times \mathbf{H} + \alpha \mathbf{m} \times \frac{\partial \mathbf{m}}{\partial t}$$

Here $\gamma = \mu_0 g_{rel} |\gamma_e|$, where $\gamma_e = -g\mu_B / \hbar$ is the electron gyromagnetic ratio and g_{rel} is a relative gyromagnetic factor (g_{rel} = 1 by default giving γ = 2.212761569×10^5 m/As). Damping constant is α. In explicit form the normalized LLG equation is given by:

$$\frac{\partial \mathbf{m}}{\partial t} = -\frac{\gamma}{1+\alpha^2} \mathbf{m} \times \mathbf{H} - \frac{\alpha\gamma}{1+\alpha^2} \mathbf{m} \times (\mathbf{m} \times \mathbf{H})$$

LLG (**AFM**)

In antiferromagnetic (or ferrimagnetic, or binary ferromagnetic alloys) meshes, a two-sublattice model is used, where the two sub-lattices are labelled *A*, *B*, and each have an LLG equation set. Coupling between the equations is done through the effective fields, in particular the exchange field. Each sub-lattice has its own set of simulation parameters. In implicit and normalized form this is given by:

$$\frac{\partial \mathbf{m}_i}{\partial t} = -\gamma_i \mathbf{m}_i \times \mathbf{H}_i + \alpha_i \mathbf{m}_i \times \frac{\partial \mathbf{m}_i}{\partial t}, \quad (i = A, B)$$

LLG (**ASC**)

Atomistic equation for magnetic dipole spin **S** with magnetic moment μ_S is given in normalized form:

$$\frac{\partial \mathbf{S}}{\partial t} = -\gamma \mathbf{S} \times \mathbf{H} + \frac{\alpha}{\mu_S} \mathbf{S} \times \frac{\partial \mathbf{S}}{\partial t}$$

LLG-STT – Landau-Lifshitz-Gilbert with Spin-Transfer Torques (**FM**)

The LLG equation can be complemented by Zhang-Li spin-transfer torques. In implicit form this becomes:

$$\frac{\partial \mathbf{m}}{\partial t} = -\gamma \mathbf{m} \times \mathbf{H} + \alpha \mathbf{m} \times \frac{\partial \mathbf{m}}{\partial t} + (\mathbf{u}.\nabla)\mathbf{m} - \beta \mathbf{m} \times (\mathbf{u}.\nabla)\mathbf{m}$$

The spin-drift velocity **u** is given by:

$$\mathbf{u} = \mathbf{J}\frac{Pg\mu_B}{2eM_s}\frac{1}{1+\beta^2}$$

The LLG-STT equation in explicit form is given by:

$$\frac{\partial \mathbf{m}}{\partial t} = -\frac{\gamma}{1+\alpha^2}\mathbf{m} \times \mathbf{H} - \frac{\alpha\gamma}{1+\alpha^2}\mathbf{m} \times (\mathbf{m} \times \mathbf{H}) +$$
$$\frac{1}{1+\alpha^2}\left[(1+\alpha\beta)(\mathbf{u}.\nabla)\mathbf{m} - (\beta-\alpha)\mathbf{m} \times (\mathbf{u}.\nabla)\mathbf{m} - \alpha(\beta-\alpha)(\mathbf{m}.(\mathbf{u}.\nabla)\mathbf{m})\mathbf{m}\right]$$

LLG-STT (**AFM**)

In implicit and normalized form this is given by:

$$\frac{\partial \mathbf{m}_i}{\partial t} = -\gamma_i \mathbf{m}_i \times \mathbf{H}_i + \alpha_i \mathbf{m}_i \times \frac{\partial \mathbf{m}_i}{\partial t} + (\mathbf{u}_i.\nabla)\mathbf{m}_i - \beta \mathbf{m}_i \times (\mathbf{u}_i.\nabla)\mathbf{m}_i, \quad (i = A, B)$$

Here:

$$\mathbf{u}_i = \mathbf{J}\frac{Pg\mu_B}{2eM_{s,i}}\frac{1}{1+\beta^2}, \quad (i = A, B)$$

LLG-STT (**ASC**)

In implicit and normalized form this is given by (magnetic dipole spin **S** with direction $\hat{\mathbf{S}}$):

$$\frac{\partial \hat{\mathbf{S}}}{\partial t} = -\gamma \hat{\mathbf{S}} \times \mathbf{H} + \alpha \hat{\mathbf{S}} \times \frac{\partial \hat{\mathbf{S}}}{\partial t} + (\mathbf{u}.\nabla)\hat{\mathbf{S}} - \beta \hat{\mathbf{S}} \times (\mathbf{u}.\nabla)\hat{\mathbf{S}}$$

The spin-drift velocity **u** is given by (μ_S is the magnetic moment, and *V* the lattice unit cell volume):

$$\mathbf{u} = \mathbf{J}\frac{Pg\mu_B V}{2e\mu_S}\frac{1}{1+\beta^2}$$

LLB – Landau-Lifshitz-Bloch (**FM**)

For non-zero temperature simulations the LLB equation should be used and in implicit form is given by (un-normalized):

$$\frac{\partial \mathbf{M}}{\partial t} = -\gamma \mathbf{M}\times\mathbf{H} + \frac{\tilde{\alpha}_\perp}{|\mathbf{M}|}\mathbf{M}\times\frac{\partial \mathbf{M}}{\partial t} + \frac{\gamma\tilde{\alpha}_\parallel}{|\mathbf{M}|}(\mathbf{M}.\mathbf{H})\mathbf{M}$$

Here for $T < T_C$ (T_C is the Curie temperature) $\alpha_\perp = \alpha(1 - T/3T_C)$, $\alpha_\parallel = \alpha 2T/3T_C$ and $\tilde{\alpha}_\perp = \alpha_\perp / m$, $\tilde{\alpha}_\parallel = \alpha_\parallel / m$, where m is the magnetization length normalized to its zero temperature value, i.e. $m = |\mathbf{M}|/M_S^0$. For T > T$_C$ $\alpha_\perp = \alpha_\parallel = 2T/3T_C$.

The effective field **H** must be complemented by a longitudinal susceptibility field given by (which due to vector cross products only affects the longitudinal torque term):

$$\mathbf{H}_l = \begin{cases} \left(1 - \frac{m^2}{m_e^2}\right)\frac{\mathbf{m}}{2\mu_0\tilde{\chi}_\parallel} & , \quad T \le T_C \\ -\frac{\mathbf{m}}{\mu_0\tilde{\chi}_\parallel}\left(1 + \frac{3T_C m^2}{5(T - T_C)}\right) & , \quad T > T_C \end{cases}$$

The field and temperature-dependent equilibrium magnetization, m_e, is given by:

$m_e(T) = B\left[m_e\frac{3T_C}{T} + \frac{\mu\mu_0 H_{ext}}{k_B T}\right]$, where $B(x) = \coth(x) - 1/x$ is the Langevin function, μ is the atomic moment.

The relative longitudinal susceptibility is given by (units 1/T), where $\chi_\parallel$ is the longitudinal susceptibility (unitless):

$\tilde{\chi}_\parallel(T) = \frac{\mu}{k_B T}\frac{B'(x)}{1 - B'(x)(3T_C/T)} = \chi_\parallel(T)/\mu_0 M_S^0$, with $x = m_e 3T_C/T$.

Further, we have the temperature dependence $M_S(T) = M_S^0 m_e(T)$. Temperature dependences for other magnetic parameters may also be set. To use the LLB equation a Curie temperature needs to be set, which automatically also calculates the above default temperature dependences: **curietemperature** *Tc*.

In explicit form the LLB equation becomes:

$$\frac{\partial \mathbf{M}}{\partial t} = -\frac{\gamma}{1+\tilde{\alpha}_\perp^2}\mathbf{M}\times\mathbf{H} - \frac{\tilde{\alpha}_\perp \gamma}{1+\tilde{\alpha}_\perp^2}\frac{1}{|\mathbf{M}|}\mathbf{M}\times(\mathbf{M}\times\mathbf{H}) + \frac{\gamma\tilde{\alpha}_\parallel}{|\mathbf{M}|}(\mathbf{M}.\mathbf{H})\mathbf{M}$$

LLB (**AFM**)

In explicit and un-normalized form this is given by:

$$\frac{\partial \mathbf{M}_i}{\partial t} = -\tilde{\gamma}_i \mathbf{M}_i \times \mathbf{H}_{eff,i} - \tilde{\gamma}_i \frac{\tilde{\alpha}_{\perp,i}}{M_i}\mathbf{M}_i \times (\mathbf{M}_i \times \mathbf{H}_{eff,i}) + \gamma_i \frac{\tilde{\alpha}_{\parallel,i}}{M_i}(\mathbf{M}_i . \mathbf{H}_{\parallel,i})\mathbf{M}_i, \quad (i = A, B)$$

Here $\tilde{\gamma}_i = \gamma_i / (1+\tilde{\alpha}_{\perp,i}^2)$ and $M_i \equiv |\mathbf{M}_i|$. As before $\tilde{\alpha}_{\perp(\parallel),i} = \alpha_{\perp(\parallel),i}/m_i$, where $m_i(T) = M_i(T)/M_{S,i}^0$ with $M_{S,i}^0$ denoting the zero-temperature saturation magnetization. The damping parameters are continuous at T_N – the phase transition temperature also set using the **curietemperature** command – and given by:

$$\alpha_{\perp,i} = \alpha_i\left(1 - \frac{T}{3(\tau_i + \tau_{ij} m_{e,j}/m_{e,i})\tilde{T}_N}\right), \quad T < T_N$$

$$\alpha_{\parallel,i} = \alpha_i\left(\frac{2T}{3(\tau_i + \tau_{ij} m_{e,j}/m_{e,i})\tilde{T}_N}\right), \quad T < T_N$$

$$\alpha_{\perp,i} = \alpha_{\parallel,i} = \frac{2T}{3T_N}, \quad T \geq T_N$$

Here we denote $\tilde{T}_N$ the re-normalized transition temperature, given by:

$$\tilde{T}_N = \frac{2T_N}{\tau_A + \tau_B + \sqrt{(\tau_A - \tau_B)^2 + 4\tau_{AB}\tau_{BA}}}$$

The micromagnetic parameters τ_i and $\tau_{ij} \in [0, 1]$, are coupling parameters between exchange constants and the phase transition temperature, such that $\tau_A + \tau_B = 1$ and $|J| = 3\tau k_B T_N$. Here J

is the exchange constant for intra-lattice (*i* = *A*,*B*) and inter-lattice (*i*,*j* = *A*,*B*, *i* ≠ *j*) coupling respectively. These are set using the **tau** command.

The normalized equilibrium magnetization functions $m_{e,i}$ are:

$$m_{e,i} = B\left[\left(m_{e,i}\tau_i + m_{e,j}\tau_{ij}\right)3\tilde{T}_N / T + \mu_i \mu_0 H_{ext} / k_B T\right]$$

The longitudinal relaxation field includes both intra-lattice and inter-lattice contributions as:

$$\mathbf{H}_{\|,i} = \left\{ \frac{1}{2\mu_0 \tilde{\chi}_{\|,i}} \left(1 - \frac{m_i^2}{m_{e,i}^2}\right) + \frac{3\tau_{ij} k_B T_N}{2\mu_0 \mu_i} \left[\frac{\tilde{\chi}_{\|,j}}{\tilde{\chi}_{\|,i}} \left(1 - \frac{m_i^2}{m_{e,i}^2}\right) - \frac{m_{e,j}}{m_{e,i}} \left(\hat{\mathbf{m}}_i . \hat{\mathbf{m}}_j\right) \left(1 - \frac{m_j^2}{m_{e,j}^2}\right) \right] \right\} \mathbf{m}_i, \quad T < T_N$$

$$\mathbf{H}_{\|,i} = -\left\{ \frac{1}{\mu_0 \tilde{\chi}_{\|,i}} + \frac{3\tau_{ij} k_B T_N}{\mu_0 \mu_i} \left[\frac{\tilde{\chi}_{\|,j}}{\tilde{\chi}_{\|,i}} - \frac{m_{e,j}}{m_{e,i}} \left(\hat{\mathbf{m}}_i . \hat{\mathbf{m}}_j\right) \right] \right\} \mathbf{m}_i \quad , \quad T > T_N$$

Here $\hat{\mathbf{m}}_i = \mathbf{m}_i / m_i$, and the relative longitudinal susceptibility is $\tilde{\chi}_{\|,i} = \chi_{\|,i} / \mu_0 M_{S,i}^0$, where:

$$k_B T \tilde{\chi}_{\|,i} = \frac{\mu_i B_i' \left(1 - 3\tau_j \tilde{T}_N B_j' / T\right) + \mu_j 3\tau_{ij} \tilde{T}_N B_i' B_j' / T}{\left(1 - 3\tau_i \tilde{T}_N B_i' / T\right)\left(1 - 3\tau_j \tilde{T}_N B_j' / T\right) - \tau_{ij}\tau_{ji} B_i' B_j' \left(3\tilde{T}_N / T\right)^2},$$

and $B_i' \equiv B_{m_{e,i}}' \left[\left(m_{e,i}\tau_i + m_{e,j}\tau_{ij}\right)3\tilde{T}_N / T\right]$.

LLB-STT – Landau-Lifshitz-Bloch with Spin-Transfer Torques (**FM**)

In implicit form we have the LLB-STT equation as:

$$\frac{\partial \mathbf{M}}{\partial t} = -\gamma \mathbf{M} \times \mathbf{H} + \frac{\tilde{\alpha}_\perp}{|\mathbf{M}|} \mathbf{M} \times \frac{\partial \mathbf{M}}{\partial t} + \frac{\gamma \tilde{\alpha}_\parallel}{|\mathbf{M}|} (\mathbf{M}.\mathbf{H})\mathbf{M} + (\mathbf{u}.\nabla)\mathbf{M} - \frac{\beta}{|\mathbf{M}|} \mathbf{M} \times (\mathbf{u}.\nabla)\mathbf{M}$$

In this case the spin-drift velocity is given by:

$$\mathbf{u} = \mathbf{J} \frac{P g \mu_B}{2 e M_S} \frac{1}{1+\beta^2}$$

In explicit form the LLB-STT equation becomes:

$$\frac{\partial \mathbf{M}}{\partial t} = -\frac{\gamma}{1+\tilde{\alpha}_\perp^2} \mathbf{M} \times \mathbf{H} - \frac{\tilde{\alpha}_\perp \gamma}{1+\tilde{\alpha}_\perp^2} \frac{1}{|\mathbf{M}|} \mathbf{M} \times (\mathbf{M} \times \mathbf{H}) + \frac{\gamma \tilde{\alpha}_\parallel}{|\mathbf{M}|} (\mathbf{M}.\mathbf{H})\mathbf{M} +$$
$$\frac{1}{(1+\tilde{\alpha}_\perp^2)} \left[(1+\tilde{\alpha}_\perp \beta)(\mathbf{u}.\nabla)\mathbf{M} - \frac{(\beta - \tilde{\alpha}_\perp)}{|\mathbf{M}|} \mathbf{M} \times (\mathbf{u}.\nabla)\mathbf{M} - \frac{\tilde{\alpha}_\perp (\beta - \tilde{\alpha}_\perp)}{|\mathbf{M}|^2} (\mathbf{M}.(\mathbf{u}.\nabla)\mathbf{M})\mathbf{M} \right]$$

LLB-STT (**AFM**)

In explicit and un-normalized form this is given by:

$$\frac{\partial \mathbf{M}_i}{\partial t} = -\frac{\gamma_i}{1+\tilde{\alpha}_{\perp,i}^2} \mathbf{M}_i \times \mathbf{H}_i - \frac{\tilde{\alpha}_{\perp,i} \gamma_i}{1+\tilde{\alpha}_{\perp,i}^2} \frac{1}{|\mathbf{M}_i|} \mathbf{M}_i \times (\mathbf{M}_i \times \mathbf{H}_i) + \frac{\gamma_i \tilde{\alpha}_{\parallel,i}}{|\mathbf{M}_i|} (\mathbf{M}_i.\mathbf{H}_i)\mathbf{M}_i +$$
$$\frac{1}{(1+\tilde{\alpha}_{\perp,i}^2)} \left[(1+\tilde{\alpha}_{\perp,i} \beta_i)(\mathbf{u}_i.\nabla)\mathbf{M}_i - \frac{(\beta - \tilde{\alpha}_{\perp,i})}{|\mathbf{M}_i|} \mathbf{M}_i \times (\mathbf{u}_i.\nabla)\mathbf{M}_i - \frac{\tilde{\alpha}_{\perp,i} (\beta - \tilde{\alpha}_{\perp,i})}{|\mathbf{M}_i|^2} (\mathbf{M}_i.(\mathbf{u}_i.\nabla)\mathbf{M}_i)\mathbf{M}_i \right]$$

Here:

$$\mathbf{u}_i = \mathbf{J} \frac{P g \mu_B}{2 e M_{S,i}} \frac{1}{1+\beta^2}, \quad (i = A, B)$$

sLLG – Stochastic Landau-Lifshitz-Gilbert (**FM**)

The explicit and normalized sLLG equation is given as:

$$\frac{\partial \mathbf{m}}{\partial t} = -\frac{\gamma}{1+\alpha^2}\mathbf{m}\times(\mathbf{H}+\mathbf{H}_{thermal}) - \frac{\alpha\gamma}{1+\alpha^2}\mathbf{m}\times(\mathbf{m}\times(\mathbf{H}+\mathbf{H}_{thermal}))$$

Each vector component of the thermal field follows a Gaussian distribution with zero mean and standard deviation given by:

$$H_\sigma = \sqrt{\frac{2\alpha k_B T}{\gamma \mu_0 M_s^0 V \Delta t}}$$

V is the volume of the stochastic computational cell, and Δt is the time step used to update the stochastic field. The stochastic field has zero spatial and vector component correlations. By default V is the mesh cellsize, and Δt is the evaluation method time-step, however different values may be set using **scellsize** and **setdtstoch** commands.

sLLG (**AFM**)

In explicit and normalized form this is given by:

$$\frac{\partial \mathbf{m}_i}{\partial t} = -\frac{\gamma_i}{1+\alpha_i^2}\mathbf{m}_i\times\left(\mathbf{H}_i+\mathbf{H}_{th,i}\right) - \frac{\alpha_i\gamma_i}{1+\alpha_i^2}\mathbf{m}_i\times\mathbf{m}_i\times\left(\mathbf{H}_i+\mathbf{H}_{th,i}\right), \quad (i = A, B)$$

As with the sLLG equation the thermal field standard deviation is given by:

$$H_{th,i}^{std.} = \sqrt{\frac{2\alpha_i k_B T}{\gamma_i \mu_0 M_{S,i}^0 V \Delta t}}$$

sLLG (**ASC**)

$$\frac{\partial \mathbf{S}}{\partial t} = -\gamma \mathbf{S}\times(\mathbf{H}+\mathbf{H}_{thermal}) + \frac{\alpha}{\mu_S}\mathbf{S}\times\frac{\partial \mathbf{S}}{\partial t}$$

Here $H_\sigma = \sqrt{\frac{2\alpha k_B T}{\gamma \mu_0 \mu_B \mu_S \Delta t}}$.

sLLB – Stochastic Landau-Lifshitz-Bloch (**FM**)

For the stochastic LLB equation we have both a thermal field and thermal torque, and is given by:

$$\frac{\partial \mathbf{M}}{\partial t} = -\frac{\gamma}{1+\tilde{\alpha}_\perp^2}\mathbf{M}\times\mathbf{H} - \frac{\tilde{\alpha}_\perp \gamma}{1+\tilde{\alpha}_\perp^2}\frac{1}{|\mathbf{M}|}\mathbf{M}\times\left(\mathbf{M}\times\left(\mathbf{H}+\mathbf{H}_{thermal}\right)\right) + \frac{\gamma\tilde{\alpha}_\parallel}{|\mathbf{M}|}\left(\mathbf{M}.\mathbf{H}\right)\mathbf{M} + \boldsymbol{\eta}_{thermal}$$

The components of the thermal field and torque follow Gaussian distributions with no correlations, zero mean and standard deviation given respectively by:

$$H_\sigma = \frac{1}{\alpha_\perp}\sqrt{\frac{2k_B T(\alpha_\perp - \alpha_\parallel)}{\gamma\mu_0 M_s^0 V\Delta t}}$$

$$\eta_\sigma = \sqrt{\frac{2k_B T\alpha_\parallel \gamma M_s^0}{\mu_0 V\Delta t}}$$

sLLB (**AFM**)

This is given by:

$$\frac{\partial \mathbf{M}_i}{\partial t} = -\tilde{\gamma}_i \mathbf{M}_i \times \mathbf{H}_{eff,i} - \tilde{\gamma}_i \frac{\tilde{\alpha}_{\perp,i}}{M_i}\mathbf{M}_i\times\left(\mathbf{M}_i\times\left(\mathbf{H}_{eff,i}+\mathbf{H}_{th,i}\right)\right) + \gamma_i\frac{\tilde{\alpha}_{\parallel,i}}{M_i}\left(\mathbf{M}_i\cdot\mathbf{H}_{\parallel,i}\right)\mathbf{M}_i + \boldsymbol{\eta}_{th,i} \quad (i = A, B)$$

As with sLLB we have ($i = A, B$):

$$H_{th,i}^{std.} = \frac{1}{\alpha_{\perp,i}}\sqrt{\frac{2k_B T\left(\alpha_{\perp,i} - \alpha_{\parallel,i}\right)}{\gamma_i \mu_0 M_{S,i}^0 V\Delta t}}$$

$$\eta_{th,i}^{std.} = \sqrt{\frac{2k_B T\alpha_{\parallel,i}\gamma_i M_{S,i}^0}{\mu_0 V\Delta t}}$$

sLLG-STT – Stochastic Landau-Lifshitz-Gilbert with Spin-Transfer Torques (**FM**, **AFM**, **ASC**)

This is similar to the LLG-STT equation, but also has the thermal field from the sLLG equation added.

sLLB-STT – Stochastic Landau-Lifshitz-Bloch with Spin-Transfer Torques (**FM**, **AFM**)

This is similar to the LLB-STT equation, but also has the thermal field from the sLLB equation added to the damping torque term, as well as the additional thermal torque term.

LLG-SA, sLLG-SA, LLB-SA, sLLB-SA – Equations with Spin Accumulation (**FM**)

The LLG, LLB, sLLG, and sLLB equations also appear in the forms LLG-SA, LLB-SA, sLLG-SA, and sLLB-SA. When using these equations the spin transport solver is enabled and a spin accumulation **S** is calculated. This gives rise to bulk and interfacial torques which are added to the respective equation. For example for the LLG equation we obtain the LLG-SA equation as:

$$\frac{\partial \mathbf{M}}{\partial t} = -\gamma \mathbf{M} \times \mathbf{H} + \frac{\alpha}{|\mathbf{M}|} \mathbf{M} \times \frac{\partial \mathbf{M}}{\partial t} + \mathbf{T}_S$$

The bulk spin-accumulation torque is given by:

$$\mathbf{T}_S = -\frac{D_e}{\lambda_J^2} \mathbf{m} \times \mathbf{S} - \frac{D_e}{\lambda_\varphi^2} \mathbf{m} \times (\mathbf{m} \times \mathbf{S})$$

This is included as an additional effective field in the explicit forms of the equations:

$$\mathbf{H}_S = \frac{D_e}{\gamma |\mathbf{M}|} \left(\frac{\mathbf{S}}{\lambda_J^2} + \frac{\mathbf{m} \times \mathbf{S}}{\lambda_\varphi^2} \right)$$

Note there are no SA version for the STT equations. This is because the Zhang-Li STTs result from the bulk $\mathbf{T}_S$ torque as a special case (see e.g. S. Lepadatu, Scientific Reports 7, 12937 (2017)).

Interfacial spin-accumulation torques are also present when N/F interfaces are used:

$$\mathbf{T}_S^{interface} = \frac{g\mu_B}{ed_h}\left[\mathrm{Re}\left\{G^{\uparrow\downarrow}\right\}\mathbf{m}\times\left(\mathbf{m}\times\Delta\mathbf{V}_S\right) + \mathrm{Im}\left\{G^{\uparrow\downarrow}\right\}\mathbf{m}\times\Delta\mathbf{V}_S\right],$$

where $\Delta\mathbf{V}_S = \mathbf{V}_{S,F} - \mathbf{V}_{S,N}$ and $\mathbf{V}_S = \left(D_e/\sigma\right)\left(e/\mu_B\right)\mathbf{S}$.

This is included as an additional effective field in the explicit forms of the equations:

$$\mathbf{H}_S = \frac{-1}{\gamma\,|\mathbf{M}|}\frac{g\mu_B}{ed_h}\left(\mathrm{Re}\left\{G^{\uparrow\downarrow}\right\}\mathbf{m}\times\Delta\mathbf{V}_S + \mathrm{Im}\left\{G^{\uparrow\downarrow}\right\}\Delta\mathbf{V}_S\right)$$

Currently antiferromagnetic meshes do not have a drift-diffusion model included, so the SA versions are not active with two-sublattice equations. This will be included in a future version.

<u>LLGStatic</u> – Static Landau-Lifshitz-Gilbert

This equation is used for static problems, i.e. where only the relaxed magnetization state is required. It is the explicit LLG equation without the precession term, and with the damping factor set to 1. This is given by:

$$\frac{\partial\mathbf{m}}{\partial t} = -\frac{\gamma}{2}\mathbf{m}\times\mathbf{m}\times\mathbf{H}$$

Modules

Modules typically correspond to an additive field in the total effective field **H** appearing in the equations shown in the Differential Equations section:

$$\mathbf{H} = \mathbf{H}_{eff} = \mathbf{H}_1 + \mathbf{H}_2 + \ldots$$

To enable a module you need to add it using the **addmodule** command. To disable a module you need to delete it using the **delmodule** command. Most modules also have an energy density term associated with their effective field contributions, available as an output data parameter. All contributions are evaluated on a cell-centered uniform finite difference mesh, with all differential operators evaluated to second order accuracy.

To see a list of available modules in the console use the **modules** command. To add a module to simulations use the **addmodule** command as **addmodule** *modulename*, e.g. **addmodule** *iDMexchange*. For multi-mesh simulations you can specify the mesh the module should be added to, e.g. **addmodule** *meshname iDMexchange*.

In Python scripts you can use *ns.addmodule('modulename')*, or for a specific mesh *ns.addmodule('meshname', 'modulename')*. To add a supermesh module you need to use *meshname* as *supermesh*, e.g. **addmodule** *supermesh sdemag*. To remove a module from simulations use the **delmodule** command, with the same parameters as **addmodule**. For a list of available module names see below.

When adding a new mesh, or resetting to default, only the *demag*, *exchange*, and *Zeeman* modules are enabled by default. In Python scripts you can also specify directly which modules should be used in a single command, after mesh creation (recommened approach), e.g.:

```
FM = ns.Ferromagnet(rect, cellsize)
FM.modules([‘Zeeman’, ‘demag’, ‘exchange’, ‘aniuni’])
```

Applicable meshes are also indicated for each module below: ferromagnetic (**FM**), 2 sub-lattice (**AFM**), atomistic simple cubic (**ASC**), conductor (**C**), insulator (**I**), dipole (**D**), supermesh (**S**).

Modules Table of Contents:

anibi – Biaxial Magneto-Crystalline Anisotropy (**FM**, **AFM**, **ASC**)

```
addmodule anibi
```

Symbol (MM units) (A units)	K_1 (J/m^3) (J)	K_2 (J/m^3) (J)	$\mathbf{e}_A$	$\mathbf{e}_{b1}$	$\mathbf{e}_{b2}$	M_S (A/m)	$K_{1,i}$ (J/m^3)	$K_{2,i}$ (J/m^3)	$M_{S,i}$ (A/m)	μ_S (μ_B)
`Param.` `(dim.)`	`K1` `(1)`	`K2` `(1)`	`ea1` `(3)`	`ea2` `(3)`	`ea3` `(3)`	`Ms` `(1)`	`K1_AFM` `(2)`	`K2_AFM` `(2)`	`Ms_AFM` `(2)`	`mu_s` `(1)`

FM:

$$\mathbf{H} = \frac{2K_1}{\mu_0 M_S}(\mathbf{m}.\mathbf{e}_A)\mathbf{e}_A - \frac{2K_2}{\mu_0 M_S}[(\mathbf{m}.\mathbf{e}_{b1})(\mathbf{m}.\mathbf{e}_{b2})^2\mathbf{e}_{b1} + (\mathbf{m}.\mathbf{e}_{b1})^2(\mathbf{m}.\mathbf{e}_{b2})\mathbf{e}_{b2}]$$

Here **m** = **M** / M_S.

Output data parameter *e_anis*:

$$\varepsilon = K_1[1-(\mathbf{m}.\mathbf{e}_A)^2] + K_2(\mathbf{m}.\mathbf{e}_{b1})^2(\mathbf{m}.\mathbf{e}_{b2})^2$$

AFM:

$$\mathbf{H}_i = \frac{2K_{1,i}}{\mu_0 M_{S,i}}(\mathbf{m}_i.\mathbf{e}_A)\mathbf{e}_A - \frac{2K_{2,i}}{\mu_0 M_{S,i}}[(\mathbf{m}_i.\mathbf{e}_{b1})(\mathbf{m}_i.\mathbf{e}_{b2})^2\mathbf{e}_{b1} + (\mathbf{m}_i.\mathbf{e}_{b1})^2(\mathbf{m}_i.\mathbf{e}_{b2})\mathbf{e}_{b2}], \quad (i = A, B)$$

Here $\mathbf{m}_i$ = $\mathbf{M}_i$ / $M_{S,i}$.

Output data parameter *e_anis*:

$\varepsilon = (\varepsilon_A + \varepsilon_B)/2$, where

$$\varepsilon_i = K_{1,i}[1-(\mathbf{m}_i.\mathbf{e}_A)^2] + K_{2,i}(\mathbf{m}_i.\mathbf{e}_{b1})^2(\mathbf{m}_i.\mathbf{e}_{b2})^2, \quad (i = A, B)$$

ASC:

$$\mathbf{H} = \frac{2K_1}{\mu_0 \mu_B \mu_S}(\hat{\mathbf{S}}.\mathbf{e}_A)\mathbf{e}_A + \frac{2K_2}{\mu_0 \mu_B \mu_S}[(\hat{\mathbf{S}}.\mathbf{e}_{b1})(\hat{\mathbf{S}}.\mathbf{e}_{b2})^2\mathbf{e}_{b1} + (\hat{\mathbf{S}}.\mathbf{e}_{b1})(\hat{\mathbf{S}}.\mathbf{e}_{b2})\mathbf{e}_{b2}].$$

Here $\mathbf{S} = \mu_S \hat{\mathbf{S}} \quad (\mu_B)$.

Output data parameter *e_anis*:

$$\varepsilon = K_1[1-(\hat{\mathbf{S}}.\mathbf{e}_A)^2] + K_2(\hat{\mathbf{S}}.\mathbf{e}_{b1})^2(\hat{\mathbf{S}}.\mathbf{e}_{b2})^2$$

anitens – 'Tensorial' Magneto-Crystalline Anisotropy (**FM**, **AFM**, **ASC**)

```
addmodule anitens
```

Symbol (MM units) (A units)	K_1 (J/m^3) (J)	K_2 (J/m^3) (J)	K_3 (J/m^3) (J)	$\mathbf{e}_1$	$\mathbf{e}_2$	$\mathbf{e}_3$	$K_{1,i}$ (J/m^3)	$K_{2,i}$ (J/m^3)	$K_{3,i}$ (J/m^3)
Param. (dim.)	K1 (1)	K2 (1)	K3 (1)	ea1 (3)	ea2 (3)	ea3 (3)	K1_AFM (2)	K2_AFM (2)	K3_AFM (2)

This module allows any magneto-crystalline anisotropy to be specified by giving the coefficients in the energy expansion:

$$\varepsilon = \sum_{o \geq 0} \sum_{\substack{i,j,k \geq 0 \\ i+j+k=o}} d_{ijk}^{(o)} \alpha^i \beta^j \gamma^k$$

Here $\alpha = \mathbf{m}.\mathbf{e}_1$, $\beta = \mathbf{m}.\mathbf{e}_2$, and $\gamma = \mathbf{m}.\mathbf{e}_3$. Also $\mathbf{m} = \mathbf{M} / M_S$.

To specify which energy terms are to be included in computations, use the **setktens** command as follows: **setktens** *dxn1yn2zn3* …

Here *d* is the coefficient for the term, *n1*, *n2*, *n3* are respective integer powers for the direction cosines, and *x*, *y*, *z* are string literals. Powers of 0 can simply be omitted, powers of 1 don't have to be written explicitly. All other modules (*aniuni*, *anibi*, *anicubi*) are special cases, so for example cubic anisotropy may be implemented in anitens to 6th order as:

setktens *x2y2 x2z2 y2z2 x2y2z2*

By convention the existing material parameters *K1*, *K2*, *K3* (and *K1_AFM*, *K2_AFM*, *K3_AFM*) now apply to 2nd, 4th, and 6th order terms in the energy expansion. For any other orders you have to specify the energy density value directly in the string set through **setktens**, as a prefactor for the respective term.

aniuni – Uniaxial Magneto-Crystalline Anisotropy (**FM**, **AFM**, **ASC**)

```
addmodule aniuni
```

Symbol (MM units) (A units)	K_1 (J/m^3) (J)	K_2 (J/m^3) (J)	$\mathbf{e}_A$	M_S (A/m)	$K_{1,i}$ (J/m^3)	$K_{2,i}$ (J/m^3)	$M_{S,i}$ (A/m)	μ_S (μ_B)
`Param.` `(dim.)`	`K1` `(1)`	`K2` `(1)`	`ea1` `(3)`	`Ms` `(1)`	`K1_AFM` `(2)`	`K2_AFM` `(2)`	`Ms_AFM` `(2)`	`mu_s` `(1)`

FM:

$$\mathbf{H} = \frac{2K_1}{\mu_0 M_S}(\mathbf{m}.\mathbf{e}_A)\mathbf{e}_A + \frac{4K_2}{\mu_0 M_S}[1-(\mathbf{m}.\mathbf{e}_A)^2](\mathbf{m}.\mathbf{e}_A)\mathbf{e}_A$$

Here **m** = **M** / M_S.

Output data parameter *e_anis*:

$$\varepsilon = K_1[1-(\mathbf{m}.\mathbf{e}_A)^2] + K_2[1-(\mathbf{m}.\mathbf{e}_A)^2]^2$$

AFM:

$$\mathbf{H}_i = \frac{2K_{1,i}}{\mu_0 M_{S,i}}(\mathbf{m}_i.\mathbf{e}_A)\mathbf{e}_A + \frac{4K_{2,i}}{\mu_0 M_{S,i}}[1-(\mathbf{m}_i.\mathbf{e}_A)^2](\mathbf{m}_i.\mathbf{e}_A)\mathbf{e}_A, \quad (i = A, B)\,.$$

Here $\mathbf{m}_i$ = $\mathbf{M}_i$ / $M_{S,i}$.

Output data parameter *e_anis*:

$\varepsilon = (\varepsilon_A + \varepsilon_B)/2$, where

$$\varepsilon_i = K_{1,i}[1-(\mathbf{m}_i.\mathbf{e}_A)^2] + K_{2,i}[1-(\mathbf{m}_i.\mathbf{e}_A)^2]^2, \quad (i = A, B)$$

ASC:

$$\mathbf{H} = \frac{2K_1}{\mu_0 \mu_B \mu_S}(\hat{\mathbf{S}}.\mathbf{e}_A)\mathbf{e}_A + \frac{4K_2}{\mu_0 \mu_B \mu_S}[1-(\hat{\mathbf{S}}.\mathbf{e}_A)^2](\hat{\mathbf{S}}.\mathbf{e}_A)\mathbf{e}_A\,.$$

Here $\mathbf{S} = \mu_S \hat{\mathbf{S}} \quad (\mu_B)$.

Output data parameter *e_anis*:

$$\varepsilon = K_1[1-(\hat{\mathbf{S}}.\mathbf{e}_A)^2] + K_2[1-(\hat{\mathbf{S}}.\mathbf{e}_A)^2]^2$$

anicubi – Cubic Magneto-Crystalline Anisotropy (**FM**, **AFM**, **ASC**)

```
addmodule anicubi
```

Symbol (MM units) (A units)	K_1 (J/m^3) (J)	K_2 (J/m^3) (J)	$\mathbf{e}_1$	$\mathbf{e}_2$	$\mathbf{e}_3$	M_S (A/m)	$K_{1,i}$ (J/m^3)	$K_{2,i}$ (J/m^3)	$M_{S,i}$ (A/m)	μ_S (μ_B)

`Param.`
`(dim.)`
	`K1`
	`(1)`
		`K2`
		`(1)`
			`ea1`
			`(3)`
				`ea2`
				`(3)`
					`ea3`
					`(3)`
						`Ms`
						`(1)`
							`K1_AFM`
							`(2)`
								`K2_AFM`
								`(2)`
									`Ms_AFM`
									`(2)`
										`mu_s`
										`(1)`

FM:

$$\mathbf{H} = -\frac{2K_1}{\mu_0 M_S}[\mathbf{e}_1\alpha(\beta^2+\gamma^2)+\mathbf{e}_2\beta(\alpha^2+\gamma^2)+\mathbf{e}_3\gamma(\alpha^2+\beta^2)] - \frac{2K_2}{\mu_0 M_S}[\mathbf{e}_1\alpha\beta^2\gamma^2+\mathbf{e}_2\alpha^2\beta\gamma^2+\mathbf{e}_3\alpha^2\beta^2\gamma]$$

Here $\alpha = \mathbf{m}.\mathbf{e}_1$, $\beta = \mathbf{m}.\mathbf{e}_2$, and $\gamma = \mathbf{m}.\mathbf{e}_3$. Also $\mathbf{m} = \mathbf{M} / M_S$.

Output data parameter *e_anis*:

$$\varepsilon = K_1[\alpha^2\beta^2+\alpha^2\gamma^2+\beta^2\gamma^2]+K_2\alpha^2\beta^2\gamma^2$$

AFM:

$$\mathbf{H}_i = -\frac{2K_{1,i}}{\mu_0 M_{S,i}}[\mathbf{e}_1\alpha_i(\beta_i^2+\gamma_i^2)+\mathbf{e}_2\beta_i(\alpha_i^2+\gamma_i^2)+\mathbf{e}_3\gamma_i(\alpha_i^2+\beta_i^2)]$$
$$-\frac{2K_{2,i}}{\mu_0 M_{S,i}}[\mathbf{e}_1\alpha_i\beta_i^2\gamma_i^2+\mathbf{e}_2\alpha_i^2\beta_i\gamma_i^2+\mathbf{e}_3\alpha_i^2\beta_i^2\gamma_i], \quad (i = A, B)$$

Here $\alpha_i = \mathbf{m}_i.\mathbf{e}_1$, $\beta_i = \mathbf{m}_i.\mathbf{e}_2$, and $\gamma_i = \mathbf{m}_i.\mathbf{e}_3$. Also $\mathbf{m}_i = \mathbf{M}_i / M_{S,i}$.

Output data parameter *e_anis*:

$\varepsilon = (\varepsilon_A + \varepsilon_B)/2$, where

$$\varepsilon_i = K_{1,i}[\alpha_i^2\beta_i^2+\alpha_i^2\gamma_i^2+\beta_i^2\gamma_i^2]+K_{2,i}\alpha_i^2\beta_i^2\gamma_i^2, \quad (i = A, B)$$

ASC:

$$\mathbf{H} = -\frac{2K_1}{\mu_0 \mu_B \mu_S}[\mathbf{e}_1\alpha(\beta^2+\gamma^2)+\mathbf{e}_2\beta(\alpha^2+\gamma^2)+\mathbf{e}_3\gamma(\alpha^2+\beta^2)] - \frac{2K_2}{\mu_0 \mu_B \mu_S}[\mathbf{e}_1\alpha\beta^2\gamma^2+\mathbf{e}_2\alpha^2\beta\gamma^2+\mathbf{e}_3\alpha^2\beta^2\gamma]$$

Here $\alpha = \hat{\mathbf{S}}.\mathbf{e}_1$, $\beta = \hat{\mathbf{S}}.\mathbf{e}_2$ and $\gamma = \hat{\mathbf{S}}.\mathbf{e}_3$. Also $\mathbf{S} = \mu_S \hat{\mathbf{S}} \quad (\mu_B)$.

Output data parameter *e_anis*:

$$\varepsilon = K_1[\alpha^2\beta^2+\alpha^2\gamma^2+\beta^2\gamma^2]+K_2\alpha^2\beta^2\gamma^2$$

demag_N – Stoner-Wohlfarth Magnetostatic Interaction (**FM**, **AFM**, **ASC**)

```
addmodule demag_N
```

Symbol (MM units) (A units)	M_S (A/m)	$M_{S,i}$ (A/m)	μ_S (μ_B)	N_x, N_y
`Param.` `(dim.)`	`Ms` `(1)`	`Ms_AFM` `(2)`	`mu_s` `(1)`	`Nxy` `(2)`

FM:

$$H_i = -N_i M_i \quad (i = x, y, z)$$

Here $N_z = 1 - N_x - N_y$.

Output data parameter *e_demag*:

$$\varepsilon = -\frac{\mu_0}{2}\mathbf{M}.\mathbf{H}$$

AFM:

$$H_{i,(A,B)} = -N_i (M_{i,A} + M_{i,B})/2 \quad (i = x, y, z)$$

Output data parameter *e_demag*:

$$\varepsilon = -\frac{\mu_0}{2}\mathbf{H}.(\mathbf{M}_A + \mathbf{M}_B)/2$$

ASC:

$$H_i = -N_i S_i \mu_B / a^3 \quad (i = x, y, z)$$

Here $N_z = 1 - N_x - N_y$, and *a* is the unit cell size.

Output data parameter *e_demag*:

$$\varepsilon = -\frac{\mu_0 \mu_B}{2a^3}\mathbf{S}.\mathbf{H}$$

demag - Magnetostatic Interaction (**FM**, **AFM**, **ASC**)

```
addmodule demag
```

Symbol (MM units) (A units)	M_s (A/m)	$M_{S,i}$ (A/m)	μ_S (μ_B)

`Param.`
`(dim.)`
	`Ms`
	`(1)`
		`Ms_AFM`
		`(2)`
			`mu_s`
			`(1)`

FM:

$$\mathbf{H}(\mathbf{r}_0) = -\int_{\mathbf{r}\in V} \mathbf{N}(\mathbf{r}-\mathbf{r}_0)\mathbf{M}(\mathbf{r})d\mathbf{r}$$

Here **N** is a rank-2 tensor with the following symmetry:

$$\mathbf{N} = \begin{pmatrix} N_{xx} & N_{xy} & N_{xz} \\ N_{xy} & N_{yy} & N_{yz} \\ N_{xz} & N_{yz} & N_{zz} \end{pmatrix}$$

N is computed using the formulas in A.J. Newell et al., “A Generalization of the Demagnetizing Tensor for Nonuniform Magnetization” J. Geophys. Res. **98**, 9551 (1993).

Output data parameter *e_demag*:

$$\varepsilon = -\frac{\mu_0}{2}\mathbf{M}.\mathbf{H}$$

AFM:

$$\mathbf{H}_{(A,B)}(\mathbf{r}_0) = -\int_{\mathbf{r}\in V} \mathbf{N}(\mathbf{r}-\mathbf{r}_0)\frac{\mathbf{M}_A(\mathbf{r})+\mathbf{M}_B(\mathbf{r})}{2}d\mathbf{r}$$

Output data parameter *e_demag*:

$$\varepsilon = -\frac{\mu_0}{2}\mathbf{H}.(\mathbf{M}_A + \mathbf{M}_B)/2$$

ASC:

Same formulas as for **FM**, where $\mathbf{M} = \frac{\mu_B}{h^3}\sum_i \mathbf{S}_i$. Here the sum runs over all spins in a macrocell with size *h*. The macrocell is set using **dmcellsize** command, and must be a multiple of the unit cell in all directions.

dipoledipole – Dipole-Dipole Interaction (ASC)

```
addmodule dipoledipole
```

Symbol (A units)	μ_S (μ_B)
Param. (dim.)	mu_s (1)

ASC:

Interaction field at spin *i* given by:

$$\mathbf{H}_i = \frac{\mu_B}{4\pi}\sum_{i \neq j}\frac{3(\mathbf{S}_j.\hat{\mathbf{r}}_{ij})\hat{\mathbf{r}}_{ij} - \mathbf{S}_j}{r_{ij}^3}$$

Here $\mathbf{r}_{ij} = r_{ij}\hat{\mathbf{r}}_{ij}$ is the distance vector from spin *i* to spin *j*.

Output data parameter *e_demag*, averaged over entire mesh from individual unit cell contributions, where *a* is the unit cell size:

$$\varepsilon_i = -\frac{\mu_0\mu_B}{2a^3}\mathbf{S}_i.\mathbf{H}_i$$

The dpin-dipole field may also be calculated using macrocells, where $\mathbf{M}_i = \sum_j \mathbf{S}_j$ is the total moment of the macrocell (units of μ_B). Here the sum runs over all spins in a macrocell *i* with size *h*. The macrocell is set using **dmcellsize** command, and must be a multiple of the unit cell in all directions. Then we have:

$$\mathbf{H}_i = \frac{\mu_B}{4\pi}\sum_{i \neq j}\frac{3(\mathbf{M}_j.\hat{\mathbf{r}}_{ij})\hat{\mathbf{r}}_{ij} - \mathbf{M}_j}{r_{ij}^3} - \frac{\mu_B}{3}\frac{\mathbf{M}_i}{h^3}$$

The output data parameter *e_demag* is not computed in macrocell mode.

<u>DMExchange</u> – Dzyaloshinskii-Moriya Bulk Exchange Interaction (**FM**, **AFM**, **ASC**)

```
addmodule DMExchange
```

Symbol (MM units) (A units)	D (J/m^2) (J)	A (J/m)	D_i (J/m^2)	D_h (J/m^3)	$\mathbf{d}_h$	M_s (A/m)	$M_{S,i}$ (A/m)	μ_S (μ_B)
Param. (dim.)	D (1)	A (1)	D_AFM (2)	Dh (1)	dh_dir (3)	Ms (1)	Ms_AFM (2)	mu_s (1)

FM:

$$\mathbf{H} = -\frac{2D}{\mu_0 M_S^2}\nabla\times\mathbf{M}$$

Non-homogeneous Neumann boundary conditions are used to evaluate the curl operator:

$\frac{\partial\mathbf{M}}{\partial\mathbf{n}} = \frac{D}{2A}\mathbf{n}\times\mathbf{M}$, where **n** is the surface normal.

The DM exchange field adds to the direct exchange field (see *exchange* module).

Output data parameter *e_exch*:

$$\varepsilon = -\frac{\mu_0}{2}\mathbf{M}.(\mathbf{H}+\mathbf{H}_{exch})$$

AFM:

$$\mathbf{H}_i = -\frac{2D_i}{\mu_0 M_{S,i}^2}\nabla\times\mathbf{M}_i + \eta_i\frac{D_h}{\mu_0 M_{S,A}M_{S,B}}\mathbf{d}_h\times\mathbf{M}_j \quad (i, j = A, B; i\neq j)\text{, where } \eta_A = +1,\ \eta_B = -1.$$

Output data parameter *e_exch*:

$\varepsilon = (\varepsilon_A + \varepsilon_B)/2$, where

$$\varepsilon_i = -\frac{\mu_0}{2}\mathbf{M}_i.(\mathbf{H}_i + \mathbf{H}_{exch,i})$$

ASC:

$\mathbf{H} = \frac{D}{\mu_0\mu_B\mu_S}\sum_{j\in N}\hat{\mathbf{r}}_{ij}\times\hat{\mathbf{S}}_j$, where sum runs over all neighbors *j*, such that $\hat{\mathbf{r}}_{ij}$ is the unit vector from this spin (*i*) to neighbour spin *j*. Here $\mathbf{S} = \mu_S\hat{\mathbf{S}} \quad (\mu_B)$.

Output data parameter *e_exch*:

$\varepsilon = -\frac{\mu_0\mu_B}{2a^3}\mathbf{S}.(\mathbf{H}+\mathbf{H}_{exch})$, where *a* is the unit cell size.

exchange – Direct Exchange Interaction (**FM**, **AFM**, ASC)

```
addmodule exchange
```

Symbol (MM units) (A units)	A (J/m)	A_i (J/m)	J (J)	$A_{h,i}$ (J/m^3)	$A_{nh,i}$ (J/m)	M_s (A/m)	$M_{S,i}$ (A/m)	μ_S (μ_B)
Param. (dim.)	A (1)	A_AFM (2)	J (1)	Ah (2)	Anh (2)	Ms (1)	Ms_AFM (2)	mu_s (1)

FM:

$$\mathbf{H} = \frac{2A}{\mu_0 M_S^2}\left(\nabla^2 \mathbf{M} - \frac{\mathbf{M}}{2|M|^2}\nabla^2(|M|^2)\right)$$

Homogeneous Neumann boundary conditions are used to evaluate the Laplacian operator.

Output data parameter *e_exch*:

$$\varepsilon = -\frac{\mu_0}{2}\mathbf{M}.\mathbf{H}$$

This is equivalent to:

$$\varepsilon = \frac{A}{M_S^2}\left[\left(\frac{\partial \mathbf{M}}{\partial x}\right)^2 + \left(\frac{\partial \mathbf{M}}{\partial y}\right)^2 + \left(\frac{\partial \mathbf{M}}{\partial z}\right)^2\right]$$

AFM:

$$\mathbf{H}_i = \frac{2A_i}{\mu_0 M_{S,i}^2}\nabla^2 \mathbf{M}_i - \frac{4A_{h,i}}{\mu_0 M_{S,A} M_{S,B}}\frac{\mathbf{M}_i \times (\mathbf{M}_i \times \mathbf{M}_j)}{M_i^2} + \frac{A_{nh,i}}{\mu_0 M_{S,A} M_{S,B}}\nabla^2 \mathbf{M}_j, \quad (i, j = A, B, \ i \neq j)$$

Output data parameter *e_exch*:

$\varepsilon = (\varepsilon_A + \varepsilon_B)/2$, where

$$\varepsilon_i = -\frac{\mu_0}{2}\mathbf{M}_i.\mathbf{H}_i$$

ASC:

$\mathbf{H} = \dfrac{J}{\mu_0 \mu_B \mu_S}\sum_{j \in N}\hat{\mathbf{S}}_j$, where sum runs over all neighbors j.

Here $\mathbf{S} = \mu_S \hat{\mathbf{S}} \quad (\mu_B)$.

Output data parameter *e_exch*:

$\varepsilon = -\dfrac{\mu_0 \mu_B}{2a^3}\mathbf{S}.\mathbf{H}$, where a is the unit cell size.

heat – Heat Equation Solver (**FM, AFM, ASC, C, I, D**)

```
addmodule heat
```

Symbol	ρ (kg/m^3)	C, C_l (J/kgK)	C_e (J/kgK)	K (W/mK)	G_e (W/m^{3}K)	Q (W/m^3)	σ (S/m)
`Param.` `(dim.)`	`density` `(1)`	`shc` `(1)`	`shc_e` `(1)`	`thermK` `(1)`	`G_e` `(1)`	`Q` `(1)`	`elC` `(1)`

The heat equation in the 1-temperature model with Joule heating and any other additional heat sources (S) is given by:

$$C(\mathbf{r})\rho(\mathbf{r})\frac{\partial T(\mathbf{r},t)}{\partial t} = \nabla.K(\mathbf{r})\nabla T(\mathbf{r},t) + Q(\mathbf{r},t) + \frac{\mathbf{J}(\mathbf{r},t)^2}{\sigma(\mathbf{r})}$$

Robin boundary conditions are used to evaluate the differential operators.

The heat equation is evaluated using the simple forward-time centered-space method. The heat equation time-step required is normally (though not always) comparable to the magnetization equation time-step thus a more time-efficient method (e.g. Crank-Nicolson) is not normally required.

In the two-temperature model, the heat equation is given as:

$$C_e(\mathbf{r})\rho(\mathbf{r})\frac{\partial T_e(\mathbf{r},t)}{\partial t} = \nabla.K(\mathbf{r})\nabla T_e(\mathbf{r},t) - G_e(\mathbf{r})[T_e(\mathbf{r},t) - T_l(\mathbf{r},t)] + Q(\mathbf{r},t)$$
$$C_l(\mathbf{r})\rho(\mathbf{r})\frac{\partial T_l(\mathbf{r},t)}{\partial t} = G_e(\mathbf{r})[T_e(\mathbf{r},t) - T_l(\mathbf{r},t)]$$

Here C_e and C_l are the electron and lattice specific heat capacities, ρ is the mass density, K is the thermal conductivity, and G_e is the electron-lattice coupling constant, typically of the order 10^{18} W/m^{3}K.

iDMExchange – Dzyaloshinskii-Moriya Interfacial Exchange Interaction (**FM**, **AFM**, **ASC**)

```
addmodule iDMExchange
```

Symbol (MM units) (A units)	D (J/m^2) (J)	A (J/m)	A_i (J/m)	D_i (J/m^2)	$A_{nh,i}$ (J/m)	M_s (A/m)	$M_{S,i}$ (A/m)	μ_S (μ_B)
`Param.` `(dim.)`	`D` `(1)`	`A` `(1)`	`A_AFM` `(2)`	`D_AFM` `(2)`	`Anh` `(2)`	`Ms` `(1)`	`Ms_AFM` `(2)`	`mu_s` `(1)`

FM:

$$\mathbf{H} = \frac{2D}{\mu_0 M_S^2}((\nabla.\mathbf{M})\hat{\mathbf{z}} - \nabla M_z)$$

Non-homogeneous Neumann boundary conditions are used to evaluate the differential operators:

$\frac{\partial \mathbf{M}}{\partial \mathbf{n}} = \frac{D}{2A}(\hat{\mathbf{z}} \times \mathbf{n}) \times \mathbf{M}$, where **n** is the surface normal.

The DM exchange field adds to the direct exchange field (see *exchange* module).

Output data parameter *e_exch*:

$$\varepsilon = -\frac{\mu_0}{2}\mathbf{M}.\mathbf{H}$$

AFM:

$$\mathbf{H}_i = \frac{2D_i}{\mu_0 M_{S,i}^2}((\nabla.\mathbf{M}_i)\hat{\mathbf{z}} - \nabla M_{z,i}), \quad (i = A, B)$$

Non-homogeneous Neumann boundary conditions are used to evaluate the differential operators:

$$\frac{\partial \mathbf{M}_i}{\partial \mathbf{n}} = \frac{D_i}{2A_i(1-c_i^2)}[(\hat{\mathbf{z}} \times \mathbf{n}) \times (\mathbf{M}_i - c_i \mathbf{M}_j)], \quad (i, j = A, B, \; i \neq j),$$

where $c_i = A_{nh,i} / 2A_i$ and **n** is the surface normal.

Output data parameter *e_exch*:

$\varepsilon = (\varepsilon_A + \varepsilon_B)/2$, where

$$\varepsilon_i = -\frac{\mu_0}{2}\mathbf{M}_i.\mathbf{H}_i$$

ASC:

$\mathbf{H} = \frac{D}{\mu_0 \mu_B \mu_S} \sum_{j \in N} (\hat{\mathbf{r}}_{ij} \times \hat{\mathbf{z}}) \times \hat{\mathbf{S}}_j$, where sum runs over all neighbors *j*, such that $\hat{\mathbf{r}}_{ij}$ is the unit vector from this spin (*i*) to neighbour spin *j*. Here $\mathbf{S} = \mu_S \hat{\mathbf{S}} \quad (\mu_B)$.

Output data parameter *e_exch*:

$\varepsilon = -\frac{\mu_0 \mu_B}{2a^3} \mathbf{S}.(\mathbf{H} + \mathbf{H}_{exch})$, where *a* is the unit cell size.

melastic – Magneto-elastic effect (**FM**, **AFM**, **C**, **I**)

```
addmodule melastic
```

Symbol	ρ (kg/m^3)	B_1, B_2 (J/m^3)	B_1, B_2 (J/m^3)	$\mathbf{e}_1$	$\mathbf{e}_2$	$\mathbf{e}_3$	Y (Pa)	ν	c_{11}, c_{12}, c_{44} (N/m^2)	η (kg/m^{3}s)	α_T (K^{-1})
`Param. (dim.)`	*density* (1)	*MEc* (2)	*mMEc* (2)	*ea1* (3)	*ea2* (3)	*ea3* (3)	*Ym* (1)	*Pr* (1)	*cC* (3)	*mdamping* (1)	*thalpha* (1)

Uniform Stress or Imported Files Mode

The magneto-elastic effect can be included for a cubic crystal using a strain tensor. The strain tensor is given as:

$$\boldsymbol{\varepsilon} = \begin{pmatrix} \varepsilon_{xx} & \varepsilon_{xy} & \varepsilon_{xz} \\ \varepsilon_{xy} & \varepsilon_{yy} & \varepsilon_{yz} \\ \varepsilon_{xz} & \varepsilon_{yz} & \varepsilon_{zz} \end{pmatrix}.$$

Here we define the diagonal strain vector as $\mathbf{S}_\mathrm{d} = (\varepsilon_{xx}, \varepsilon_{yy}, \varepsilon_{zz})$, and off-diagonal, or shear, strain vector as $\mathbf{S}_\mathrm{od} = (\varepsilon_{yz}, \varepsilon_{xz}, \varepsilon_{xy})$. The strain tensor can have a spatial dependence, and can be loaded from two ovf2 files (diagonal and shear strain components computed with an external package – **loadovf2strain**), or alternatively a displacement vector field can be loaded (using ovf2 files, computed externally – **loadovf2disp**). From the displacement **u** = (u_x, u_y, u_z), the strain tensor is computed as:

$$\varepsilon_{pq} = \frac{1}{2}\left(\frac{\partial u_p}{\partial q} + \frac{\partial u_q}{\partial p}\right), \quad (p, q = x, y, z)$$

In the simplest case a uniform stress may be applied which results in a constant strain with zero off-diagonal terms – **setstress**. This is given below for uniform external pressure **F** = (F_x, F_y, F_z), where Y is Young's modulus and ν is Poisson's ratio:

$$\mathbf{S}_d = \frac{1}{Y}\begin{pmatrix} 1 & -\nu & -\nu \\ -\nu & 1 & -\nu \\ -\nu & -\nu & 1 \end{pmatrix}\begin{pmatrix} F_x \\ F_y \\ F_z \end{pmatrix}$$

From the strain tensor, for a cubic crystal with orthogonal axes $\mathbf{e}_1$, $\mathbf{e}_2$, $\mathbf{e}_3$, and magneto-elastic constants *B_1*, *B_2*, we have the following diagonal and off-diagonal energy density terms (*e_mel* output data is the sum of the two contributions*)*:

$$\varepsilon_{mel,d} = B_1\left[(\mathbf{m}.\mathbf{e}_1)^2(\mathbf{S}_d.\mathbf{e}_1) + (\mathbf{m}.\mathbf{e}_2)^2(\mathbf{S}_d.\mathbf{e}_2) + (\mathbf{m}.\mathbf{e}_3)^2(\mathbf{S}_d.\mathbf{e}_3)\right]$$

$$\varepsilon_{mel,od} = 2B_2\left[(\mathbf{m}.\mathbf{e}_1)(\mathbf{m}.\mathbf{e}_2)(\mathbf{S}_{od}.\mathbf{e}_3) + (\mathbf{m}.\mathbf{e}_1)(\mathbf{m}.\mathbf{e}_3)(\mathbf{S}_{od}.\mathbf{e}_2) + (\mathbf{m}.\mathbf{e}_2)(\mathbf{m}.\mathbf{e}_3)(\mathbf{S}_d.\mathbf{e}_1)\right]$$

For example, for axes coninciding with the *x*-*y*-*z* axes (default setting), this gives the total energy density:

$$\varepsilon_{mel} = B_1[m_x{}^2\varepsilon_{xx} + m_y{}^2\varepsilon_{yy} + m_z{}^2\varepsilon_{zz}] + 2B_2[m_x m_y\varepsilon_{xy} + m_x m_z\varepsilon_{xz} + m_y m_z\varepsilon_{yz}]$$

Here $\mathbf{m} = \mathbf{M} / M_S$.

The effective field can be computed using the usual formula: $\mathbf{H}_{mel} = -\frac{1}{\mu_0 M_S}\frac{\partial\varepsilon_{mel}}{\partial\mathbf{m}}$.

Elastodynamics Solver

The elastodynamics equations are solved in the velocity-stress representation, included in a magneto-thermo-elastic model, based on three coupled dynamics equations for magnetization, temperature, and elasticity properties:

$$\begin{aligned}
&\frac{\partial\mathbf{m}}{\partial t} = -\gamma\mathbf{m}\times\mathbf{H}_{eff} + \alpha\mathbf{m}\times\frac{\partial\mathbf{m}}{\partial t}\\
&C\rho\frac{\partial T}{\partial t} = K\nabla^2 T + Q\\
&\rho\frac{\partial v_p}{\partial t} = \sum_{q=x,y,z}\frac{\partial\sigma_{pq}}{\partial q} - \eta v_p, \quad (p = x, y, z)
\end{aligned}$$

The first equation is the LLG equation, although any of the implemented magnetization dynamics equations may be used, and the second expression is the heat equation (see **heat** module description). The third expression is the elastodynamics equation, written in the velocity-stress representation, where **v** is the velocity, **σ** is the symmetric stress tensor, and η is a mechanical damping factor resulting in energy dissipation. This is solved using the finite-difference time-domain (FDTD) scheme. The stress and strain for a cubic crystal are related using:

$$\sigma_{pp} = c_{11}\varepsilon_{pp} + 2c_{12}(\varepsilon_{qq} + \varepsilon_{rr}) - (c_{11} + 2c_{12})\alpha_T(T - T_0) - B_1 m_p^2, \quad (p,q,r = x,y,z,\ p \neq q \neq r)$$

$$\sigma_{pq} = c_{44}\varepsilon_{pq} - 2B_2 m_p m_q, \quad (p,q = x,y,z,\ p \neq q)$$

Here c_{11}, c_{12}, c_{44} are elastic stiffness coefficients, α_T is the linear thermal expansion coefficient, and T_0 is the ambient temperature (**ambient** command). The terms involving B_1 and B_2 give the magnetostriction contribution. Note, the inverse thermo-elastic effect (change in temperature due to lattice strains) is not currently included in the solver.

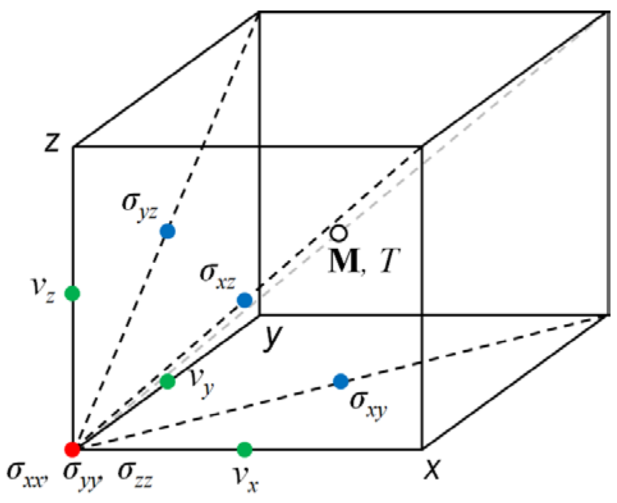
Taking the time derivative of strain and stress expressions allows replacing the displacement with velocity, and we obtain a system of first-order differential equations for elastodynamics, written in the velocity-stress representation as:

$$\rho \frac{\partial v_p}{\partial t} = \sum_{q=x,y,z} \frac{\partial \sigma_{pq}}{\partial q} - \eta v_p, \quad (p = x,y,z)$$

$$\frac{\partial \sigma_{pp}}{\partial t} = c_{11}\frac{\partial v_p}{\partial p} + c_{12}\left(\frac{\partial v_q}{\partial q} + \frac{\partial v_r}{\partial r}\right) - (c_{11} + 2c_{12})\alpha_T \frac{\partial T}{\partial t} - 2B_1 m_p \frac{\partial m_p}{\partial t}, \quad (p,q,r = x,y,z,\ p \neq q \neq r)$$

$$\frac{\partial \sigma_{pq}}{\partial t} = \frac{c_{44}}{2}\left(\frac{\partial v_p}{\partial q} + \frac{\partial v_q}{\partial p}\right) - 2B_2\left(m_p \frac{\partial m_q}{\partial t} + m_q \frac{\partial m_p}{\partial t}\right), \quad (p,q = x,y,z,\ p \neq q)$$

Time derivatives of temperature and magnetization are evaluated directly in the dynamical magneto-thermo-elastic model. The equations are solved using the FDTD scheme, with staggered velocity and stress components as indicated in the figure below for a given computational cell:



Staggered velocity and stress components – Velocity, stress, magnetization and temperature components for finite difference discretization of the magneto-thermo-elastic model. Velocity components are edge-centred, diagonal stress components are at vertices, shear stress components are face-centred, and magnetization and temperature values are cell-centred.

The temperature and magnetization values are cell-centred; their values and that of their time derivative at vertices, edge and face centres, are obtained to second order accuracy in space using interpolation. Spatial derivatives at inner points are evaluated using standard finite differences to second order accuracy. There must be at least one fixed surface where the mechanical displacement is zero, since no translational motion is allowed, and thus the velocity is also zero (add/delete a fixed surface using the **surfacefix** and **delsurfacefix** commands). Velocity components on a fixed surface are set directly as zero, whilst derivatives of velocity components perpendicular to a fixed surface are evaluated using standard expressions with Dirichlet boundary condition of zero. As an example, for a fixed *xz* surface, $v_x = v_z = 0$, whilst for $\partial v_y / \partial y$ at the boundary the Dirichlet condition $v_y = 0$ applies. For all other surfaces (free surfaces) boundary stress values are prescribed from external forces (add/delete an external force surface using the **surfacestress** and **delsurfacestress** commands) as (δ is the Kronecker delta):

$$\sum_{q=x,y,z} \sigma_{pq}\delta_{rq} = F_r, \quad (p, r = x, y, z)$$

Here p is normal to the free surface. The boundary stress values are either set directly for components on free surfaces, or else are used as Dirichlet boundary conditions to evaluate derivatives of stress perpendicular to a free surface. As an example, for a fixed *xz* surface, $\sigma_{yy} = F_y$ is set directly, whilst $\sigma_{xy} = F_x$ and $\sigma_{yz} = F_z$ are used as Dirichlet boundary conditions to evaluate $\partial \sigma_{xy} / \partial y$ and $\partial \sigma_{yz} / \partial y$ respectively, at the boundary. Finally, required derivatives of velocity perpendicular to a free surface are obtained as:

$$\frac{\partial v_p}{\partial p} = \frac{1}{c_{11}} \frac{\partial F_p}{\partial t} - \frac{c_{12}}{c_{11}} \left(\frac{\partial v_q}{\partial q} + \frac{\partial v_r}{\partial r} \right) + \left(1 + 2\frac{c_{12}}{c_{11}} \right) \alpha_T \frac{\partial T}{\partial t} + \frac{2B_1}{c_{11}} m_p \frac{\partial m_p}{\partial t}, \quad (p, q, r = x, y, z,\ p \neq q \neq r)$$

For multi-layered structures, composite media boundary (CMB) conditions are also required. These consist of enforcing continuity of velocity and stress components located on the CMB interface, achieved by interpolating respective values either side of the interface. These values are set after the elastodynamics equation is iterated in all layers. It is also necessary to specify initial values for stress and velocity. The velocity initial value is zero throughout, whilst the initial values of stress components, which are not on a free surface, are computed by taking the initial strain to be zero. To reset the solver to initial conditions use the **resetelsolver** command.

The time-step for the elastodynamics equation (set using **seteldt** command) must satisfy the Courant, Friedrichs, Lewy condition, $\Delta t < 1/\left(v_\mathrm{p}\sqrt{\Delta x^{-2} + \Delta y^{-2} + \Delta z^{-2}}\right)$, where v_p is the elastic compressional wave velocity (typically v_p = 6000 m/s), and Δx, Δy, Δz are the cell-size dimensions. The mechanical cellsize is set using the **mcellsize** command.

<u>moptical</u> – Magneto-optical effect (**FM**, **AFM**, **ASC**)

```
addmodule moptical
```

This module applies a *z*-axis field as:

$$H_{MO} = \sigma^{\pm} H_{MO}^{0} f_{MO}(\mathbf{r}, t)\hat{\mathbf{z}}$$

Here $\sigma^{\pm} H_{MO}^{0}$ is set using the *Hmo* parameter, and f_{MO} is its spatial (and temporal) variation.

Energy density term (output data parameter: *e_mo*):

$$\varepsilon = -\mu_0 \mathbf{M}.\mathbf{H}$$

For two-sublattice models the same field is applied to both sublattices.

<u>mstrayfield</u> – Stray field from fixed magnetic dipoles (**FM**, **AFM**, **ASC**)

```
addmodule mstrayfield
```

This is similar to the **<u>strayfield</u>** module, but instead of computing stray field on the supermesh and then transferring them to an individual mesh using interpolation, the stray field is computed directly in the given mesh with its magnetization discretization.

Oersted – Oersted Field (**S**)

```
addmodule supermesh Oersted
```

Effective field contribution:

$$\mathbf{H}(\mathbf{r}_0) = \int_{\mathbf{r}\in V} \mathbf{K}(\mathbf{r}-\mathbf{r}_0)\mathbf{J}_C(\mathbf{r})d\mathbf{r}$$

Here **K** is a rank-2 tensor with the following symmetry:

$$\mathbf{K} = \begin{pmatrix} 0 & K_{xy} & K_{xz} \\ -K_{xy} & 0 & K_{yz} \\ -K_{xz} & -K_{yz} & 0 \end{pmatrix}$$

K is computed using the formulas in B. Krüger, "Current-Driven Magnetization Dynamics: Analytical Modeling and Numerical Simulation", PhD Dissertation, University of Hamburg (2011) – Appendix D, page 118. Also $\mathbf{J}_C$ is the current density, and the Oersted field is computed from all meshes with *transport* module enabled (thus on the *electric supermesh*).

For two-sublattice models the Oersted field is applied equally to both sublattices.

roughness – Roughness Field and Staircase Magnetostatic Corrections (**FM, AFM**)

```
addmodule roughness
```

Effective field contribution computed on the coarse mesh (i.e. the actual mesh discretisation used at run-time with N_V number of discretisation cells):

$$\mathbf{H}(\mathbf{r}_0) = -\left[\sum_{\mathbf{r}\in V} \mathbf{N}(\mathbf{r}-\mathbf{r}_0) G(\mathbf{r},\mathbf{r}_0)\right] \mathbf{M}(\mathbf{r}_0) \quad (\mathbf{r}_0 \in V)$$

Here **N** is the demagnetizing tensor computed on the fine mesh with N_{Vr} number of discretisation cells, and:

$$G(\mathbf{r},\mathbf{r}_0) = \begin{cases} \dfrac{N_V}{N_{Vr}} - 1 & \mathbf{r} \wedge \mathbf{r}_0 \in V_R \\ -1 & \mathbf{r} \vee \mathbf{r}_0 \in V - V_R \end{cases}$$

V is the smooth body without roughness and V_R is the mesh with roughness, and we require $V_R \subseteq V$. If the coarse cellsize has dimensions (h_x, h_y, h_z), the fine cellsize must have dimensions (h_x / m_x, h_y / m_y, h_z / m_z), where the m factors are integers. The function $\sum_{\mathbf{r}\in V} \mathbf{N}(\mathbf{r}-\mathbf{r}_0) G(\mathbf{r},\mathbf{r}_0)$ is computed at initialisation on the finely discretised mesh then averaged up to the coarse mesh (each coarse cellsize value is obtained as an average of its contained fine cellsize values).

Energy density term (output data parameter: *e_rough*):

$$\varepsilon = -\frac{\mu_0}{2} \mathbf{M}.\mathbf{H}$$

Details can be found in: S. Lepadatu, "Effective field model of roughness in magnetic nano-structures" J. Appl. Phys. **118**, 243908 (2015).

For two-sublattice models the roughness field is computed using the average magnetization, and applied equally to both sublattices.

sdemag – Supermesh Magnetostatic Interaction (**S**)

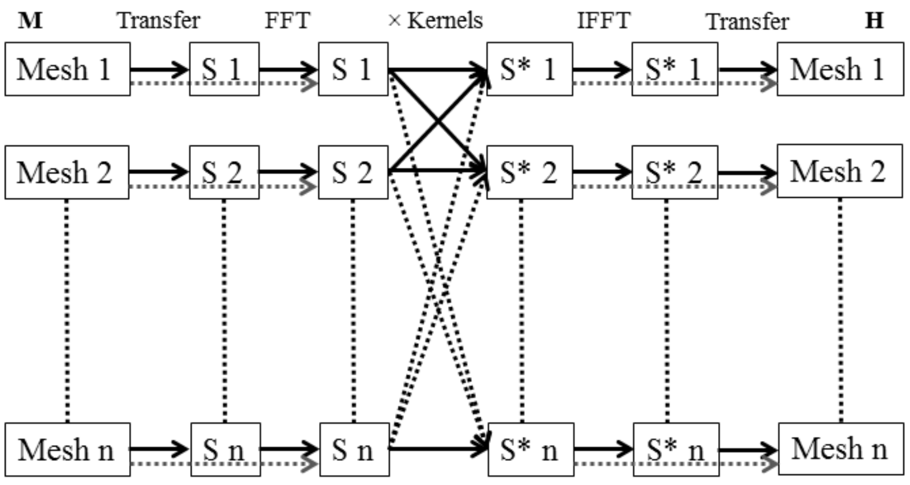
```
addmodule supermesh sdemag
```

I. Multi-Layered convolution (**multiconvolution** 1). This is the default. Can handle multiple meshes which do not fit onto a regular supermesh grid.

A generalisation of the single layer convolution algorithm is used here. We can write the convolution sum as:

$$\mathbf{H}(\mathbf{r}'_{kl}) = -\sum_{\substack{i=1,\dots,n \\ \mathbf{r}_{ij}\in V_i}} \mathbf{N}(\mathbf{r}'_{kl} - \mathbf{r}_{ij}, \mathbf{h}_k, \mathbf{h}_i)\mathbf{M}(\mathbf{r}_{ij}), \quad k = 1,\dots,n; \quad \mathbf{r}'_{kl} \in V_k$$

In the demagnetizing tensor for the equation above we explicitly specify the cellsize, **h**, of the two computational meshes the tensor relates. Since we have *n* terms of the form appearing in the single layer convolution sum, we can again apply the convolution theorem. This time for each output mesh (**H**) we have *n* input meshes (**M**), together with *n* kernels. Thus to calculate the outputs in all *n* meshes we require a total of n^2 sets of kernel multiplications in the transform space. This is illustrated in the figure below.



Multi-Layered convolution algorithm for *n* computational meshes. The magnetization input of each mesh is transformed separately using a FFT algorithm, either directly (dotted line), or by first transferring to a scratch space with a common discretisation cellsize, using a weighted average smoother (solid lines). In the transform space the inputs are multiplied with pre-computed kernels for a total of n^2 sets of point-by-point multiplications. Finally the output demagnetizing fields are obtained using an inverse FFT algorithm, which are set directly in the output meshes (dotted line), or transferred using a weighted average smoother if the discretisation cellsizes differ (solid lines).

II. Supermesh demagnetization (**multiconvolution** 0). Only use if multiple meshes fit onto a regular supermesh grid. Not recommended otherwise.

The same formulas as for the *demag* module are used when computing demagnetizing fields on the uniformly discretised super-mesh. The ferromagnetic super-mesh may have a cellsize which differs from that of the individual ferromagnetic meshes. In this case a weighted average smoother is used to transfer magnetization to the super-mesh and demagnetizing field values back from the super-mesh.

Consider a discrete distribution of magnetization values **M** at points $V = \{\mathbf{r}_i;\ i \in P\}$. Let **h** be the cellsize of the input mesh, with the set of cells $\{c_i;\ i \in P\}$ centered around the points $\mathbf{r}_i$. To obtain the magnetization value at a point $\mathbf{r}'$ in a cell c with dimensions $\mathbf{h}'$ we introduce the definitions $d_i = |\mathbf{r}' - \mathbf{r}_i|$, $d_V = |\mathbf{h}' + \mathbf{h}|/2$, and $\tilde{d}_i = d_V - d_i$. The weighted average is given as:

$$\mathbf{M}(\mathbf{r}') = \sum_{i \in P} w_i \mathbf{M}(\mathbf{r}_i)$$

where

$$w_i = \frac{\tilde{d}_i \delta_i}{\tilde{d}_T}$$

$$\delta_i = \begin{cases} 1, & c_i \cap c \neq \varnothing \\ 0, & otherwise \end{cases}$$

$$\tilde{d}_T = \sum_{i \in P} \tilde{d}_i \delta_i$$

<u>SOTfield</u> – Spin-Orbit Torque Field (**FM**, **AFM**)

```
addmodule SOTfield
```

Symbol	M_s (A/m)	$M_{S,i}$ (A/m)	$\theta_{SHA,eff}$	r_G	g_R	$g_{R,i}$	**d**
Param. (dim.)	*Ms* (1)	*Ms_AFM* (2)	*SHA* (1)	*flST* (1)	*grel* (1)	*grel_AFM* (2)	*STp* (3)

The spin-orbit torque is given by:

$$\mathbf{T}_{SOT} = \theta_{SHA,\,eff} \frac{\mu_B}{e} \frac{|J_c|}{d_h} \left(\mathbf{m} \times (\mathbf{m} \times \mathbf{p}) + r_G \mathbf{m} \times \mathbf{p} \right)$$

Here **p** = **d** × $\mathbf{e}_{Jc}$, where **d** is the spin current direction incident on the interface (default **z** corresponding to an HM/FM bilayer stacked along *z* axis), and $\mathbf{e}_{Jc}$ is the current density direction. d_h is the FM layer thickness. The *transport* module needs to be enabled and configured so a current density is available.

FM:

$$\mathbf{H}_{SOT} = -\frac{1}{\gamma M_S} \theta_{SHA,\,eff} \frac{\mu_B}{e} \frac{|J_c|}{d_h} \left(\mathbf{m} \times \mathbf{p} + r_G \mathbf{p} \right)$$

Here $\gamma = \mu_0 |\gamma_e| g_R$, where γ_e is the electron gyromagnetic ratio and g_R is a relative gyromagnetic factor. Also **m** = **M** / M_S.

AFM:

$$\mathbf{H}_{SOT,i} = -\frac{1}{\gamma_i M_{S,i}} \theta_{SHA,\,eff} \frac{\mu_B}{e} \frac{|J_c|}{d_h} \left(\mathbf{m}_i \times \mathbf{p} + r_G \mathbf{p} \right) \quad (i = A, B)$$

Here $\gamma_i = \mu_0 |\gamma_e| g_{R,i}$, where γ_e is the electron gyromagnetic ratio and $g_{R,i}$ is a relative gyromagnetic factor. Also $\mathbf{m}_i = \mathbf{M}_i / M_{S,i}$.

STfield – Slonczewski Spin Torques Field (**FM**)

```
addmodule STfield
```

Symbol	M_s (A/m)	q_+, q_-	*A*, *B*	r_G	g_R	**p**	q_+, q_- (top)	*A*, *B* (top)	r_G (top)
`Param. (dim.)`	`Ms (1)`	`STq (2)`	`STa (2)`	`flST (1)`	`grel (1)`	`STp (3)`	`STq2 (2)`	`STa2 (2)`	`flST2 (1)`

The spin torque is given by (including damping-like and field-like components):

$$\mathbf{T}_{ST} = \frac{\mu_B}{e}\frac{J_z}{d}\eta\left[\mathbf{m}\times(\mathbf{m}\times\mathbf{p}) + r_G\mathbf{m}\times\mathbf{p}\right]$$

$$\eta = \frac{q_+}{A + B(\mathbf{m}.\mathbf{p})} + \frac{q_-}{A - B(\mathbf{m}.\mathbf{p})}$$

Here **p** is the spin polarization from the fixed layer, and J_z is the current density in the z direction (this module is applicable for layers stacked along the z direction, i.e. CPP-GMR geometry), and *d* is the thickness of the free layer. The angular dependence is specified through the η function, which is parmetrized by q_+, q_-, *A*, and *B*. The field-like coefficient is r_G. The *transport* module needs to be enabled and configured so a current density is available.

This results in an effective field in the magnetization dynamics equation given by:

$$\mathbf{H}_{ST} = -\frac{1}{\gamma M_S}\eta(\theta)\frac{\mu_B}{e}\frac{J_z}{d}(\mathbf{m}\times\mathbf{p} + r_G\mathbf{p})$$

Here $\gamma = \mu_0 |\gamma_e| g_R$, where γ_e is the electron gyromagnetic ratio and g_R is a relative gyromagnetic factor. Also **m** = **M** / M_S. This effective field is applied equally to all cells along the z direction.

Non-constant polarization mode

In the above approach, the polarization **p** was set to a fixed direction using the the *STp* parameter. In a multi-layered simulation, it is possible to use the polarization value directly from neighboring magnetic layers in a stack along the z direction. This mode is selected simply by setting *STp* to be a zero vector. Then, the program will search for neighboring magnetic layers, top and bottom along the z direction, and in each cell next to the interface use the **p** value (normalized magnetization direction) obtained from the neighboring layer. Thus, e.g. for a bilayer F / F$_{bot}$, the effective field used for F at interface cells is obtained as:

$$\mathbf{H}_{ST} = -\frac{1}{\gamma M_S}\eta(\theta)\frac{\mu_B}{e}\frac{J_z}{h}\left(\mathbf{m}\times\mathbf{p} + r_G\mathbf{p}\right),\ \eta = \frac{q_+}{A + B(\mathbf{m}.\mathbf{p})} + \frac{q_-}{A - B(\mathbf{m}.\mathbf{p})}$$

Here *h* is the z-direction cellsize of the F layer. Thus, unlike in the fixed polarization mode, here the effective field is included at interface cells only. Over the entire thickness the total spin torque still averages to a 1 / *d* dependence, but this approach results in a more realistic description for thicker layers, since typically the spin torque from an adjacent layer is much stronger close to the interface.

Aditionally, the non-constant polarization mode allows inclusion of spin torques from both top and bottom layers independently, e.g. as in a trilayer F$_{top}$ / F / F$_{bot.}$ The spin torque from F$_{top}$ on F is still calculated using the above formula, however the spin torque parameters can be set independently using the *STq2, STa2,* and *flST2* parameters, i.e. *STq2* = (q_+, q_-), *STa2* = (*A*, *B*), an *flST2* = r_G for cells at the F$_{top}$ / F interface, with **p** taken from F$_{top}$ for each interface cell. In general these values may be obtained by fitting the angular dependence of spin torques computed self-consistently using the spin transport solver for any particular geometry.

strayfield – Stray field from fixed magnetic dipoles (**S**)

```
addmodule supermesh strayfield
```

If $\mathbf{M}_d$ is the magnetization of a uniformly magnetized prism with dimensions **d**, then the magnetic field (stray field) at a distance **r** from the centre of the prism is given by:

$$\mathbf{H} = -\mathbf{N}_d(\mathbf{r}, \mathbf{d})\mathbf{M}_d$$

Here $\mathbf{N}_d$ is a rank-2 tensor with the following symmetry:

$$\mathbf{N}_d = \begin{pmatrix} N_{xx} & N_{xy} & N_{xz} \\ N_{xy} & N_{yy} & N_{yz} \\ N_{xz} & N_{yz} & N_{zz} \end{pmatrix}$$

$\mathbf{N}_d$ is computed using the formulas in A. Andreev et al., “Universal Method for the Calculation of Magnetic Microelectronic Components: the Saturated Ferromagnetic Rectangular Prism and the Rectangular Coil.” ICSE2000 Proceedings, Nov. 2000, 187. Note these formulas are equivalent to the Newell formulas used to compute the demagnetizing tensor.

This module computes the stray field resulting from all dipole meshes and includes it in all magnetic meshes. For two-sublattice models the field is applied equally to both sublattices.

surfexchange – Surface Exchange Interaction (**FM**, **AFM**, **ASC**)

```
addmodule surfexchange
```

Symbol (MM units) (A units)	J_1, J_A (J/m^2)	J_2, J_B (J/m^2)	J_S (J)	M_s (A/m)	$M_{S,i}$ (A/m)	μ_S (μ_B)
`Param.` `(dim.)`	`J1` `(1)`	`J2` `(1)`	`Js` `(1)`	`Ms` `(1)`	`Ms_AFM` `(2)`	`mu_s` `(1)`

This module allows surface exchange coupling between different meshes along the *z* direction. In order for 2 meshes to be exchange coupled they both need to have the *surfexchange* module enabled, and must be stacked adjacently along the *z* direction. The coupling effective fields are computed between pairs of cells either side of the interface/separation. For 2 coupled meshes it is the top mesh which sets the material parameters to be used.

The different types of exchange coupling currently possible are given below.

RKKY (**FM** to **FM** coupling):

Let $\mathbf{m}_i$ and $\mathbf{m}_j$ be the normalized magnetization values of two cells (normalized to M_S), *i* and *j*, which are surface exchange coupled across a gap between two ferromagnetic meshes. Let Δ be the thickness of the ferromagnetic layer for which the surface exchange field is computed. The surface exchange field at cell *i*, from cell *j*, is given as:

$$\mathbf{H}_{ij} = \frac{J_1}{\mu_0 M_S \Delta}\mathbf{m}_j + \frac{2J_2}{\mu_0 M_S \Delta}(\mathbf{m}_i.\mathbf{m}_j)\mathbf{m}_j$$

The effective field is applied to all cells along the z direction if a 3D simulation mesh is used (surface exchange field applicable for thin films).

Output data parameter *e_surfexch*:

$$\varepsilon_{ij} = -\frac{J_1}{\Delta}\mathbf{m}_i.\mathbf{m}_j - \frac{J_2}{\Delta}(\mathbf{m}_i.\mathbf{m}_j)^2$$

RKKY (**ASC** to **ASC** coupling):

The surface exchange field at cell *i*, from cell *j*, is given as:

$$\mathbf{H}_{ij} = \frac{J_S}{\mu_0 \mu_B \mu_S} \hat{\mathbf{S}}_j$$

The effective field is applied only to cells at the interface.

Output data parameter *e_surfexch*:

$\varepsilon_{ij} = -\frac{J_S}{a^3} \hat{\mathbf{S}}_i . \hat{\mathbf{S}}_j$, where *a* is the unit cell size.

RKKY (**ASC** to **FM** coupling):

Micromagnetic and atomistic meshes may also be coupled using RKKY. This is a combination of the above equations, where magnetization values are obtained by averaging over atomistic spins and dividing by cell-size volume. Thus to compute effective field in **FM**, the **FM** to **FM** coupling formula is used, but magnetization on the **ASC** side is obtained by averaging over atomistic spins. To compute the effective field in **ASC**, the **ASC** to **ASC** coupling formula is used, but the magnetization direction in **FM** sets the spin direction.

Exchange Bias (**AFM** to **FM** coupling):

Let $\mathbf{m}_i$ be the normalized magnetization (normalized to M_S) of cell *i* in the ferromagnetic mesh, and $\mathbf{m}_{jA}$, $\mathbf{m}_{jB}$ be the normalized magnetization (normalized to $M_{S,A}$ respectively $M_{S,B}$) values in cell *j* for sub-lattices *A* and *B* of the 2-sublattice mesh (antiferromagnet). Let Δ be the thickness of the ferromagnetic layer for which the surface exchange field is computed. The surface exchange field at cell *i*, from cell *j*, is given as:

$$\mathbf{H}_{ij} = \frac{J_A}{\mu_0 M_S \Delta} \mathbf{m}_{j,A} + \frac{J_B}{\mu_0 M_S \Delta} \mathbf{m}_{j,B}$$

The effective field is applied to all cells along the z direction if a 3D simulation mesh is used (surface exchange field applicable for thin films).

Output data parameter *e_surfexch*:

$$\varepsilon_{ij} = -\frac{J_A}{\Delta}\mathbf{m}_i.\mathbf{m}_{j,A} - \frac{J_B}{\Delta}\mathbf{m}_i.\mathbf{m}_{j,B}$$

<u>Exchange Bias</u> (**FM** to **AFM** coupling):

Let $\mathbf{m}_j$ be the normalized magnetization (normalized to M_S) of cell j in the ferromagnetic mesh, and $\mathbf{m}_{iA}$, $\mathbf{m}_{iB}$ be the normalized magnetization (normalized to $M_{S,A}$ respectively $M_{S,B}$) values in cell i for sub-lattices A and B of the 2-sublattice mesh (antiferromagnet). Let Δ be the thickness of the 2-sublattice mesh for which the surface exchange field is computed. The surface exchange field at cell i, from cell j, is given as:

$$\mathbf{H}_{ij,A} = \frac{J_A}{\mu_0 M_{S,A}\Delta}\mathbf{m}_j,\ \mathbf{H}_{ij,B} = \frac{J_B}{\mu_0 M_{S,B}\Delta}\mathbf{m}_j$$

The effective field is applied to all cells along the z direction if a 3D simulation mesh is used (surface exchange field applicable for thin films).

Output data parameter *e_surfexch*:

$$\varepsilon_{ij} = -\frac{J_A}{2\Delta}\mathbf{m}_{i,A}.\mathbf{m}_j - \frac{J_B}{2\Delta}\mathbf{m}_{i,B}.\mathbf{m}_j$$

<u>Exchange Coupling</u> (**AFM** to **AFM** coupling):

Let $\mathbf{m}_{j,A}$, $\mathbf{m}_{j,B}$, $\mathbf{m}_{iA}$, $\mathbf{m}_{iB}$ be the normalized magnetization (normalized to $M_{S,A}$ respectively $M_{S,B}$) of cells i, j for sub-lattices A and B of the 2-sublattice meshes. Let Δ be the thickness of the 2-sublattice mesh for which the surface exchange field is computed. The surface exchange field at cell i, from cell j, is given as:

$$\mathbf{H}_{ij,A} = \frac{J_A}{\mu_0 M_{S,A}\Delta}\mathbf{m}_{j,A},\ \mathbf{H}_{ij,B} = \frac{J_B}{\mu_0 M_{S,B}\Delta}\mathbf{m}_{j,B}$$

The effective field is applied to all cells along the z direction if a 3D simulation mesh is used (surface exchange field applicable for thin films).

Output data parameter *e_surfexch*:

$$\varepsilon_{ij} = -\frac{J_A}{2\Delta}\mathbf{m}_{i,A}.\mathbf{m}_{j,A} - \frac{J_B}{2\Delta}\mathbf{m}_{i,B}.\mathbf{m}_{j,A}$$

Exchange Bias (**ASC** to **AFM** coupling):

Micromagnetic and atomistic meshes may also be coupled using exchange bias. This is a combination of the above equations, where magnetization values are obtained by averaging over atomistic spins and dividing by cell-size volume – see description of RKKY (**ASC** to **FM** coupling).

transport – Charge and Spin-Transport Solver (**FM, AFM, C, ASC**)

```
addmodule transport
```

Charge Transport (**FM, AFM, C, ASC**)

Symbol (MM units)	σ (S/m)	*AMR* (%)
Param. (dim.)	elC (1)	amr (1)

When solving only for the charge current density, a Poisson-type equation for V is solved as:

$$\nabla^2 V = -\frac{(\nabla V).(\nabla \sigma)}{\sigma}$$

For *V* Dirichlet boundary conditions are used at boundaries containing a fixed potential electrode, otherwise Neumann boundary conditions are used. The conductivity may have an AMR contribution (AMR given as a percentage value) calculated as (**FM** only):

$$\sigma = \frac{\sigma_0}{1+(AMR\ /100)d^2},\ \text{where}\ d = \frac{\mathbf{J}_C.\mathbf{M}}{|\mathbf{J}_C||\mathbf{M}|}.$$

From *V* the charge current density is obtained as $\mathbf{J}_c = -\sigma\nabla V$, and is used for Joule heating computations and to obtain spin torques (Zhang-Li STT and SOT).

The Poisson equation is evaluated using the successive over-relaxation (SOR) algorithm with black-red ordering for parallelization.

Charge and Spin Transport (**FM, C, ASC**)

Symbol (MM units)	*P*	D_e (m^2/s)	*n* (m^{-3})	β_D	λ_{sf} (m)	λ_J (m)	λ_φ (m)	$G^\uparrow, G^\downarrow$ (S/m^2)	$G^{\uparrow\downarrow}$ (S/m^2)
Param. (dim.)	P (1)	De (1)	n (1)	betaD (1)	l_sf (1)	l_sf (1)	l_phi (1)	Gi (2)	Gmix (2)

Charge and spin current densities are given as (see S. Lepadatu, "Unified treatment of spin torques using a coupled magnetization dynamics and three-dimensional spin current solver" Scientific Reports 7, 12937 (2017) for details):

$$\mathbf{J}_C = \sigma\mathbf{E} + \beta_D D_e \frac{e}{\mu_B}(\nabla\mathbf{S})\mathbf{m} + \theta_{SHA} D_e \frac{e}{\mu_B}\nabla\times\mathbf{S} + P\sigma\frac{\hbar}{2e}\mathbf{E}^{\sigma} - P\frac{\sigma^2\hbar}{e^2 n}\mathbf{E}\times\mathbf{B}^{\sigma}$$

$$\mathbf{J}_S = -\frac{\mu_B}{e}P\sigma\mathbf{E}\otimes\mathbf{m} - D_e\nabla\mathbf{S} + \theta_{SHA}\frac{\mu_B}{e}\boldsymbol{\varepsilon}\sigma\mathbf{E} + \frac{\hbar\mu_B\sigma}{2e^2}\sum_i \mathbf{e}_i\otimes(\dot{\mathbf{m}}\times\partial_i\mathbf{m}) + \frac{\hbar\mu_B\sigma^2}{e^3 n}(\mathbf{z}\times\mathbf{E})\otimes(\partial_x\mathbf{m}\times\partial_y\mathbf{m})$$

Here:

$$E_i^{\sigma} = (\dot{\mathbf{m}}\times\partial_i\mathbf{m}).\mathbf{m}$$

$$\mathbf{B}^{\sigma} = \mathbf{z}(\partial_x\mathbf{m}\times\partial_y\mathbf{m}).\mathbf{m}$$

Here $\mathbf{E}^{\sigma}$ and $\mathbf{B}^{\sigma}$ are the directions of the emergent electric field due to charge pumping, and emergent magnetic field due to topological Hall effect respectively.

With the full spin transport solver enabled both V and $\boldsymbol{S}$ are computed using Poisson-type equations as:

$$\nabla^2 V = -\frac{(\nabla V).(\nabla\sigma)}{\sigma} + \frac{\beta_D D_e}{\sigma}\frac{e}{\mu_B}\nabla.(\nabla\mathbf{S})\mathbf{m}$$
$$+\frac{P\hbar}{2e}\left[\partial_x\dot{\mathbf{m}}\times\partial_x\mathbf{m} + \partial_y\dot{\mathbf{m}}\times\partial_y\mathbf{m} + \dot{\mathbf{m}}\times(\partial_x^2\mathbf{m} + \partial_y^2\mathbf{m})\right].\mathbf{m}$$
$$-\frac{P\sigma\hbar}{e^2 n}\left[\mathbf{x}(\partial_{xy}^2\mathbf{m}\times\partial_y\mathbf{m} + \partial_x\mathbf{m}\times\partial_y^2\mathbf{m}).\mathbf{m} - y(\partial_x^2\mathbf{m}\times\partial_y\mathbf{m} + \partial_x\mathbf{m}\times\partial_{xy}^2\mathbf{m}).\mathbf{m}\right].\nabla V$$

and

$$\nabla^2\mathbf{S} = -\frac{\mu_B}{e}\frac{P\sigma}{D_e}(\mathbf{E}.\nabla)\mathbf{m} + \frac{\mu_B}{e}\frac{P\sigma}{D_e}(\nabla^2 V)\mathbf{m} + \frac{\theta_{SHA}}{D_e}\frac{\mu_B}{e}\sigma\nabla.(\boldsymbol{\varepsilon}\mathbf{E})$$
$$+\frac{\hbar\mu_B\sigma}{2e^2 D_e}\left[\partial_x\dot{\mathbf{m}}\times\partial_x\mathbf{m} + \partial_y\dot{\mathbf{m}}\times\partial_y\mathbf{m} + \dot{\mathbf{m}}\times(\partial_x^2\mathbf{m} + \partial_y^2\mathbf{m})\right]$$
$$+\frac{\hbar\mu_B\sigma^2}{e^3 n D_e}\left[E_x(\partial_{xy}^2\mathbf{m}\times\partial_y\mathbf{m} + \partial_x\mathbf{m}\times\partial_y^2\mathbf{m}) - E_y(\partial_x^2\mathbf{m}\times\partial_y\mathbf{m} + \partial_x\mathbf{m}\times\partial_{xy}^2\mathbf{m})\right]$$
$$+\frac{\mathbf{S}}{\lambda_{sf}^2} + \frac{\mathbf{S}\times\mathbf{m}}{\lambda_J^2} + \frac{\mathbf{m}\times(\mathbf{S}\times\mathbf{m})}{\lambda_{\varphi}^2}$$

For boundaries containing an electrode with a fixed potential, differential operators applied to V use a Dirichlet boundary condition. For other external boundaries the following non-homogeneous Neumann boundary conditions are used:

$$\nabla V.\mathbf{n} = \theta_{\mathrm{SHA}} \frac{D_e}{\sigma} \frac{e}{\mu_{\mathrm{B}}} (\nabla \times \mathbf{S}).\mathbf{n}$$

$$(\nabla \mathbf{S}).\mathbf{n} = \theta_{\mathrm{SHA}} \frac{\sigma}{D_e} \frac{\mu_{\mathrm{B}}}{e} (\boldsymbol{\varepsilon}\mathbf{E}).\mathbf{n}$$

At N/F composite media boundaries the following conditions are applied:

$$\mathbf{J}_C.\mathbf{n}\big|_N = \mathbf{J}_C.\mathbf{n}\big|_F = -\left(G^{\uparrow} + G^{\downarrow}\right)\Delta V + \left(G^{\uparrow} - G^{\downarrow}\right)\Delta \mathbf{V}_S.\mathbf{m}$$

$$\mathbf{J}_S.\mathbf{n}\big|_N - \mathbf{J}_S.\mathbf{n}\big|_F = \frac{2\mu_B}{e}\left[\mathrm{Re}\left\{G^{\uparrow\downarrow}\right\}\mathbf{m} \times (\mathbf{m} \times \Delta \mathbf{V}_S) + \mathrm{Im}\left\{G^{\uparrow\downarrow}\right\}\mathbf{m} \times \Delta \mathbf{V}_S\right]$$

$$\mathbf{J}_S.\mathbf{n}\big|_F = \frac{\mu_B}{e}\left[-\left(G^{\uparrow} + G^{\downarrow}\right)(\Delta \mathbf{V}_S.\mathbf{m})\mathbf{m} + \left(G^{\uparrow} - G^{\downarrow}\right)\Delta V \mathbf{m}\right]$$

Spin pumping is included on the N side of the above equations as:

$$\mathbf{J}_S^{pump} = \frac{\mu_B \hbar}{e^2}\left[\mathrm{Re}\left\{G^{\uparrow\downarrow}\right\}\mathbf{m} \times \frac{\partial \mathbf{m}}{\partial t} + \mathrm{Im}\left\{G^{\uparrow\downarrow}\right\}\frac{\partial \mathbf{m}}{\partial t}\right]$$

At N/F interfaces, interfacial spin torques are obtained as (h_F is the discretisation cellsize of the F layer in the direction normal to the composite media boundary) :

$$\mathbf{T}_S = \frac{g\mu_B}{eh_F}\left[\mathrm{Re}\left\{G^{\uparrow\downarrow}\right\}\mathbf{m} \times (\mathbf{m} \times \Delta \mathbf{V}_S) + \mathrm{Im}\left\{G^{\uparrow\downarrow}\right\}\mathbf{m} \times \Delta \mathbf{V}_S\right]$$

From **S**, bulk spin torques are obtained as:

$$\mathbf{T}_S = -\frac{D_e}{\lambda_J^2}\mathbf{m} \times \mathbf{S} - \frac{D_e}{\lambda_\varphi^2}\mathbf{m} \times (\mathbf{m} \times \mathbf{S})$$

Both Poisson equations are evaluated using the SOR algorithm with black-red ordering for parallelization. Note, whilst the SOR algorithm is robust in evaluating the spin transport equations in arbitrary multi-layers with composite media boundary conditions, it does suffer from slow convergence in particular for lower target solver errors. This algorithm is due to be replaced with a more efficient method in a future version.

Finally, different contributions in the spin transport solver may be turned on or off using the parameters *pump_eff* (spin pumping), *cpump_eff* (charge pumping), *the_eff* (topological Hall effect), *ts_eff* (bulk spin torques), *tsi_eff* (interfacial spin torques).

TMR (**I**)

Symbol (MM units)	R_pA (Ωm^2)	$R_{ap}A$ (Ωm^2)	σ (S/m)	D_e (m^2/s)	λ_{sf} (m)	$G^{\uparrow}, G^{\downarrow}$ (S/m^2)	$G^{\uparrow\downarrow}$ (S/m^2)
Param. (dim.)	RAtmr_p (1)	RAtmr_ap (1)	elC (1)	De (1)	l_sf (1)	Gi (2)	Gmix (2)

In tunnel barriers separating two ferromagnetic layers, such that $\cos\theta$ is the dot product of pairs of magnetization unit vectors either side of the barrier, we calculate the conductivity from the angular-dependent resistance, $R(\theta)$, obtained from Slonczewski's formula:

$$R(\theta) = \frac{R_0}{1 + \dfrac{R_{ap} - R_p}{R_{ap} + R_p} \cos\theta}$$

Here $R_0 = 2R_{ap}R_p / (R_{ap} + R_p)$ is the resistance for perpendicular orientation of ferromagnetic layers' magnetizations, with R_{ap} and R_p being resistances for antiparallel and parallel orientations respectively. We use the definition $TMR = (R_{ap} - R_p)/R_p$. The tunnel barrier's resistance area (*RA)* product is obtained for the parallel orientation of magnetization vectors, i.e. $RA = R_p A$, and it is used as a parameter in the model. The conductivity is then expressed as $\sigma = d_I / R(\theta)A$, where d_I is the insulator layer thickness and A is the barrier area.

Thus, within this model the insulator layer is treated as a poor conductor, and a charge current density is computed throughout a simulated structure by taking the electrical potential V (where $\mathbf{E} = -\nabla V$), to be continuous across all interfaces. Further, across the tunnel barrier we impose the requirement of net spin current conservation, which gives the following equations within the tunnel barrier:

$$\nabla^2 V = -\frac{(\nabla V).(\nabla \sigma)}{\sigma}$$

$$\nabla^2 \mathbf{S} = 0$$

Within this model spin currents are conserved across the tunnel barrier, but discontinuities in the spin accumulation can arise between the two sides of the barrier.

The $R(\theta)$ formula is set using the **tmrtype** command, where the default setting of 1 is the Slonczewski formula above. The other possibility is setting of 0, which sets the formula:

$$R(\theta) = R_p + (R_{ap} - R_p)(1 - \cos\theta) / 2 .$$

Metallic conduction channels may be introduced by setting the *elC* material parameter to have a spatial variation (non-zero values of *elC* will be treated using metallic conduction, using the parameters D_e, λ_{sf}, $G^{\uparrow}, G^{\downarrow}$, and $G^{\uparrow\downarrow}$).

TAMR (**FM**, **ASC**)

Symbol (MM units)	σ_0 (S/m)	*TAMR* (%)	$\mathbf{e}_1$	$\mathbf{e}_2$	$\mathbf{e}_3$
Param. (dim.)	*elC* (1)	*tamr* (1)	*ea1* (3)	*ea2* (3)	*ea3* (3)

Tunelling anisotropic magnetoresistance (TAMR) may be included in magnetic meshes, which gives rise to a dependence of the conductivity depending on the set material parameters and TAMR formula.

By default the following TAMR formula is used:

$$\sigma = \frac{\sigma_0}{1 + (TAMR \ / 100)(1 - d_1^2)}$$. Here $d_1 = \mathbf{m}.\mathbf{e}_1$, where **m** is the magnetization direction.

It is possible to specify a custom formula for TAMR, which takes into account the *tamr* parameter, as well as the three axes *ea1*, *ea2*, *ea3*. This is done using the **tamrequation** command (see Commands). This command takes a user text equation (see User-Defined Text Equations), where the text equation uses parameters *TAMR*, *d1*, *d2*, *d3*. Here TAMR = *tamr* / 100 is the ratio, where *tamr* is the material parameter set as a percentage value. Also $d_i = \mathbf{m}.\mathbf{e}_i, \quad (i = 1,2,3)$.

Thermoelectric Effect (**FM**, C, ASC)

Symbol (MM units)	σ_0 (S/m)	S_C (V/K)
Param. (dim.)	*elC* (1)	*S* (1)

The thermoelectric effect may be included by enabling both the *transport* and *heat* modules, and setting a non-zero Seebeck coefficient (S_C – this is the *S* material parameter). In this case any temperature gradients will generate a thermoelectric effect.

The electric field is now obtained as $\mathbf{E} = -\nabla V - S_C \nabla T$. From $\nabla.\mathbf{J} = 0$, where $\mathbf{J} = \sigma \mathbf{E}$, we now obtain the following contribution to the Poisson equation for *V*, which the solver uses for computations:

$$\nabla^2 V = -S_C \nabla^2 T$$

There are two computation modes: 1) open potential mode, and 2) externally set potential mode (this is the default mode). In the default mode, where the potential on electrodes is fixed, then no net thermoelectric current can be generated. In the open potential mode, an external resistance is defined (e.g. this would be the external resistance across defined electrodes, to which the device interfaces), R_0, which allows a net thermoelectric current to be generated by setting the potential drop across electrodes as $V_0 = R_0 I_T$. Here I_T is the total thermoelectric current flowing into the ground electrode, where the corresponding thermoelectric current density is $\mathbf{J}_C = \sigma S_C \nabla T$.

The thermoelectric effect computation mode is set using the **openpotentialresistance** command.
Set R_0 to zero to disable open potential mode:

```
openpotentialresistance 0
```

Set a non-zero R_0 value to enable net thermoelectric current generation (it is recommended R_0 is set to a matched value, i.e. set to the base device resistance between electrodes, *R*), e.g.:

```
openpotentialresistance 100
```

viDMExchange – Vectorial Dzyaloshinskii-Moriya interfacial exchange interaction (**FM**, **AFM**, **ASC**)

```
addmodule viDMExchange
```

Symbol (MM units) (A units)	D (J/m^2) (J)	A (J/m)	A_i (J/m)	D_i (J/m^2)	$A_{nh,i}$ (J/m)	M_s (A/m)	$M_{S,i}$ (A/m)	μ_S (μ_B)	**d**$_{DMI}$
`Param. (dim.)`	`D (1)`	`A (1)`	`A_AFM (2)`	`D_AFM (2)`	`Anh (2)`	`Ms (1)`	`Ms_AFM (2)`	`mu_s (1)`	`Ddir (3)`

FM:

$$\mathbf{H}_x = \frac{2D}{\mu_0 M_S^2}((\nabla.\mathbf{M})\hat{\mathbf{x}} - \nabla M_x),\ \mathbf{H}_y = \frac{2D}{\mu_0 M_S^2}((\nabla.\mathbf{M})\hat{\mathbf{y}} - \nabla M_y),\ \mathbf{H}_z = \frac{2D}{\mu_0 M_S^2}((\nabla.\mathbf{M})\hat{\mathbf{z}} - \nabla M_z)$$

Then: $\mathbf{H} = \alpha\mathbf{H}_x + \beta\mathbf{H}_y + \gamma\mathbf{H}_z$, where $\mathbf{d}_{DMI} = (\alpha, \beta, \gamma)$ is a unit vector.

Non-homogeneous Neumann boundary conditions are used to evaluate the differential operators:

$\frac{\partial \mathbf{M}}{\partial \mathbf{n}} = \frac{D}{2A}[\alpha(\hat{\mathbf{x}} \times \mathbf{n}) + \beta(\hat{\mathbf{y}} \times \mathbf{n}) + \gamma(\hat{\mathbf{z}} \times \mathbf{n})] \times \mathbf{M}$, where **n** is the surface normal.

The DM exchange field adds to the direct exchange field (see *exchange* module).

Output data parameter *e_exch*:

$$\varepsilon = -\frac{\mu_0}{2}\mathbf{M}.\mathbf{H}$$

AFM:

$$\mathbf{H}_{x,i} = \frac{2D_i}{\mu_0 M_{S,i}^2}((\nabla.\mathbf{M}_i)\hat{\mathbf{x}} - \nabla M_{y,i}),\quad (i = A, B)$$

$$\mathbf{H}_{y,i} = \frac{2D_i}{\mu_0 M_{S,i}^2}((\nabla.\mathbf{M}_i)\hat{\mathbf{y}} - \nabla M_{y,i}),\quad (i = A, B)$$

$$\mathbf{H}_{z,i} = \frac{2D_i}{\mu_0 M_{S,i}^2}((\nabla.\mathbf{M}_i)\hat{\mathbf{z}} - \nabla M_{z,i}),\quad (i = A, B)$$

Then: $\mathbf{H}_i = \alpha\mathbf{H}_{x,i} + \beta\mathbf{H}_{y,i} + \gamma\mathbf{H}_{z,i},\quad (i = A, B)$, where $\mathbf{d}_{DMI} = (\alpha, \beta, \gamma)$ is a unit vector.

Non-homogeneous Neumann boundary conditions are used to evaluate the differential operators:

$$\frac{\partial \mathbf{M}_i}{\partial \mathbf{n}} = \frac{D_i}{2A_i(1-c_i^2)}[\alpha(\hat{\mathbf{x}} \times \mathbf{n}) + \beta(\hat{\mathbf{y}} \times \mathbf{n}) + \gamma(\hat{\mathbf{z}} \times \mathbf{n})] \times (\mathbf{M}_i - c_i\mathbf{M}_j),\quad (i, j = A, B,\ i \neq j),$$

where $c_i = A_{nh,i} / 2A_i$, and **n** is the surface normal.

Output data parameter *e_exch*:

$\varepsilon = (\varepsilon_A + \varepsilon_B)/2$, where

$$\varepsilon_i = -\frac{\mu_0}{2}\mathbf{M}_i.\mathbf{H}_i$$

ASC:

$$\mathbf{H}_x = \frac{D}{\mu_0 \mu_B \mu_S}\sum_{j \in N}(\hat{\mathbf{r}}_{ij} \times \hat{\mathbf{x}}) \times \hat{\mathbf{S}}_j, \ \mathbf{H}_y = \frac{D}{\mu_0 \mu_B \mu_S}\sum_{j \in N}(\hat{\mathbf{r}}_{ij} \times \hat{\mathbf{y}}) \times \hat{\mathbf{S}}_j, \ \mathbf{H}_z = \frac{D}{\mu_0 \mu_B \mu_S}\sum_{j \in N}(\hat{\mathbf{r}}_{ij} \times \hat{\mathbf{z}}) \times \hat{\mathbf{S}}_j$$

Then: $\mathbf{H} = \alpha\mathbf{H}_x + \beta\mathbf{H}_y + \gamma\mathbf{H}_z$, where $\mathbf{d}_{DMI} = (\alpha, \beta, \gamma)$ is a unit vector.

Here the sums run over all neighbors *j*, such that $\hat{\mathbf{r}}_{ij}$ is the unit vector from this spin (*i*) to neighbour spin *j*. Here $\mathbf{S} = \mu_S \hat{\mathbf{S}} \quad (\mu_B)$.

Output data parameter *e_exch*:

$\varepsilon = -\frac{\mu_0 \mu_B}{2a^3}\mathbf{S}.(\mathbf{H} + \mathbf{H}_{exch})$, where *a* is the unit cell size.

<u>Zeeman</u> – Applied Magnetic Field (**FM**, **AFM**, **ASC**)

```
addmodule Zeeman
```

Symbol	c_H
`Param. (dim.)`	`cHa (1)`

Effective field contribution:

$$\mathbf{H} = \mathbf{H}_{ext} c_H(\mathbf{r})$$

Output data parameter *e_zee*:

$$\varepsilon = -\mu_0 \mathbf{M}.\mathbf{H}$$

For two-sublattice models the field is applied equally to both sublattices.

Material Parameters

To set a material parameter value use the **setparam** command as **setparam** *meshname paramname value*, e.g. **setparam** *permalloy A* 1e-11, or even **setparam** *permalloy A* 10pJ/m.

```
setparam paramname value
```

To set the parameter in a specific mesh:

```
setparam meshname paramname value
```

In Python scripts you can set a parameter value as *ns.setparam('paramname', value)*, or for a specific mesh as *ns.setparam('meshname', 'paramname', value).* A higher level approach to working with material parameters in Python is through a mesh object, e.g.:

```
#create a ferromagnetic mesh with given rectangle and discretization
FM = ns.Ferromagnet(rect, cellsize)

#now access material parameters through the param class in FM
#e.g. the line below sets the exchange stiffness value
FM.param.A = 1e-11

#to read a parameter value:
A = FM.param.A.setparam()
```

For list of available parameter names see below. To see a list of available parameters in the console use the **params** command.

Format:

paramname: *name in equations* (units)

Description.

A: *A* (J/m)

Exchange stiffness.

A_AFM: A_i (J/m)

Exchange stiffness.

Ah: $A_{h,i}$ (J/m^3)

Homogeneous inter-lattice exchange coupling per lattice constant.

Anh: $A_{nh,i}$ (J/m^3)
Non-homogeneous inter-lattice exchange stiffness.

amr: *AMR* (%)
Anisotropic magneto-resistance as a percentage of base resistance.

beta: β (unitless)
Spin-transfer torque non-adiabaticity parameter.

betaD: β_D (unitless)
Diffusion spin polarisation.

cC: (c_{11}, c_{12}, c_{44}) (N/m^2)
Stiffness constants for a cubic crystal.

cHA: *cHA* (unitless)
Applied field spatial variation parameter, which multiplies the applied field value.

cpump_eff: *cpump_eff* (unitless)
Charge-pumping efficiency.

cT: *cT* (unitless)
Set temperature spatial variation parameter, which multiplies the set temperature value. To enable spatial variation of temperature you need to have the *heat* module enabled. If you want the set temperature to remain constant you need to disable the heat equation by using **setheatdt** 0.

D: *D* (J/m^2)
Dzyaloshinskii-Moriya exchange constant.

D: *D* (J)
Atomistic Dzyaloshinskii-Moriya exchange constant.

Ddir: $\mathbf{d_D}$ (unitless)

Interfacial DMI symmetry axis.

D_AFM: D_i (J/m^2)

Dzyaloshinskii-Moriya exchange constant.

damping: α (unitless)

Gilbert magnetization damping.

damping_AFM: α_i (unitless)

Gilbert magnetization damping.

Ddir: $\mathbf{d_{DMI}}$ (unitless)

Interfacial Dzyaloshinskii-Moriya interaction direction unit vector.

De: D_e (m^2/s)

Electron diffusion constant.

density: ρ (kg/m^3)

Mass density.

Dh: D_h (J/m^3)

Homogeneous Dzyaloshinskii-Moriya exchange constant for 2-sublattice model.

dh_dir: $\mathbf{d_h}$ (unitless)

Homogeneous Dzyaloshinskii-Moriya interaction unit vector for 2-sublattice model.

ea1: $\boldsymbol{e}_{a1}$ (unit vector)

Uniaxial magneto-crystalline anisotropy symmetry axis, or first cubic magneto-crystalline anisotropy symmetry axis.

ea2: $\boldsymbol{e}_{a2}$ (unit vector)

Second cubic magneto-crystalline anisotropy symmetry axis.

ea3: $\boldsymbol{e}_{a3}$ (unit vector)
Third cubic magneto-crystalline anisotropy symmetry axis.

elC: σ (S/m)
Base electrical conductivity.

flST: r_G (unitless)
Field-like spin orbit torque coefficient.

G_e: G_e (W/m^{3}K)
Electron-lattice coupling constant (two temperature model).

Gi: $G^{\uparrow}$, $G^{\downarrow}$ (S/m^2)
Interface spin-dependent conductivity (for majority and minority carriers). The top contacting mesh sets the interface value, thus *Gi* is available in both magnetic and non-magnetic meshes.

Gmix: $\mathrm{G}^{\uparrow\downarrow} = Re\{G^{\uparrow\downarrow}\} + i\ Im\{G^{\uparrow\downarrow}\}$ (S/m^2)
Interface spin-mixing conductivity (real and imaginary parts). The top contacting mesh sets the interface value, thus *Gmix* is available in both magnetic and non-magnetic meshes.

grel: g_{rel} (unitless)
Relative electron gyromagnetic ratio.

grel_AFM: $g_{rel,i}$ (unitless)
Relative electron gyromagnetic ratio.

Hmo: *Hmo* (A/m)
Magneto-optical field strength.

iSHA: θ_{SHA} (unitless)
Spin Hall angle used for the inverse spin Hall effect.

J: J (J)

Atomistic direct exchange constant.

J1: J_1 (J/m^2)

Bilinear surface exchange coupling. For coupled meshes it is the top mesh that sets the J values.

J2: J_2 (J/m^2)

Biquadratic surface exchange coupling. For coupled meshes it is the top mesh that sets the J values.

joule_eff: η_J (unitless)

Joule heating effect efficiency.

Js: J_S (J)

Atomistic surface exchange constant.

K1: K_1 (J/m^3)

Magneto-crystalline anisotropy energy density.

K1: K_1 (J)

Atomistic magneto-crystalline anisotropy energy.

K1_AFM: $K_{1,i}$ (J/m^3)

Magneto-crystalline anisotropy energy density.

K2: K_2 (J/m^3)

Magneto-crystalline anisotropy energy density, higher order.

K2: K_2 (J)

Atomistic magneto-crystalline anisotropy energy, higher order.

K2_AFM: $K_{2,i}$ (J/m^3)

Magneto-crystalline anisotropy energy density, higher order.

K3: K_3 (J/m^3)

Magneto-crystalline anisotropy energy density, higher order.

K3: K_3 (J)

Atomistic magneto-crystalline anisotropy energy, higher order.

K3_AFM: $K_{3,i}$ (J/m^3)

Magneto-crystalline anisotropy energy density, higher order.

l_J: λ_J (m)

Spin exchange rotation length.

l_phi: λ_φ (m)

Spin dephasing length.

l_sf: λ_{sf} (m)

Spin-flip length.

MEc: B_1, B_2 (J/m^3)

Magneto-elastic coefficients.

mMEc: B_1, B_2 (J/m^3)

Magneto-elastic coefficients used for magnetostriction (should be same as *MEc*).

Ms: Ms (A/m)

Saturation magnetization.

Ms_AFM: Ms_i (A/m)

Saturation magnetization.

mdamping: η (kg/m^{3}s)

Mechanical damping.

mu_s: μ_S (μ_B)

Atomistic magnetic moment (in units of μ_B).

n: *n* (m^{-3})

Conduction electrons density.

Nxy: N_x, N_y (unitless)

In-plane demagnetizing factors (used by *demag_N* module)

P: *P* or β_σ (unitless)

Charge current spin polarization.

Pr: ν (unitless)

Poisson's ratio.

pump_eff: η_{pump} (unitless)

Spin pumping efficiency.

Q: *Q* (W/m^3)

Heat source added to the heat equation. Can be non-uniform by setting a spatial variation.

RAtmr_ap: $R_{ap}\times A$ (Ωm^2)

TMR RA product for antiparallel state.

RAtmr_p: $R_p\times A$ (Ωm^2)

TMR RA product for parallel state.

SHA: θ_{SHA} (unitless)

Spin Hall angle used for the spin Hall effect.

S: S (V/K)

Seebeck coefficient.

shc: C, C_l (J/kgK).

Specific heat capacity (total – one temperature model, lattice – two temperature model).

shc_e: C_e (J/kgK).

Electronic specific heat capacity (two temperature model).

STa: A, B (unitless).

Parameters in Slonczewski spin torques angular dependence (denominator constants).

STq: q_+, q_- (unitless).

Parameters in Slonczewski spin torques angular dependence (symmetric and asymmetric proportional components).

STp: $\mathbf{p}$ (unitless).

Spin polarization from fixed layer in Slonczewski spin torques (unit vector), or spin current direction for SOT.

s_eff: η_s (unitless)

Stochasticity efficiency parameter.

susrel: $\chi_{||}$ (As^2/kg)

Longitudinal (parallel) susceptibility divided by $\mu_0 M_S$.

susrel_AFM: $\chi_{||,i}$ (As^2/kg)

Longitudinal (parallel) susceptibility divided by $\mu_0 M_{S,i}$.

tamr: *tamr* (%)

Tunnelling anisotropic magnetoresistance.

thalpha: α_T (K^{-1})

Linear coefficient of thermal expansion.

the_eff: η_{the} (unitless)

Topological Hall effect efficiency.

thermK: K (W/mK)

Thermal conductivity.

ts_eff: η_{ts} (unitless)

Spin accumulation torque efficiency in the bulk.

tsi_eff: η_{tsi} (unitless)

Spin accumulation torque efficiency at interfaces.

Ym: Y (Pa)

Young’s modulus.

Parameters Temperature Dependence

Material parameters may be assigned a temperature dependence where appropriate. There are two available methods: using an *array*, or using a *text equation*. In all cases the temperature dependence is the scaling to be applied to the base parameter value. Thus if A_0 is the set material parameter base value, then $a(T)$ is its set temperature dependence, such that the material parameter as a function of temperature is $A(T) = A_0\ a(T)$.

1) To set a temperature dependence using a text equation:

```
setparamtempequation (meshname) paramname text_equation
```

For details see User-Defined Text Equations. When using a text equation both a temperature and time dependence may be set.

2) To set a temperature dependence using an array loaded from a file:

```
setparamtemparray (meshname) paramname filename
```

The file must contain 2 tab-spaced columns. The first gives the temperature value, and the second gives the scaling value at the respective temperature.

To remove a material parameter temperature dependence use:

```
clearparamstemp (meshname) (paramname)
```

The above commands may be used in Python scripts through the param class in mesh objects, e.g.:

```
#create a ferromagnetic mesh with given rectangle and discretization
FM = ns.Ferromagnet(rect, cellsize)

#Now access temperature dependences commands through the param class
#e.g. the lines below work for the exchange stiffness
FM.param.A.setparamtemparray(filename)
FM.param.A.setparamtempequation(equation_string)
FM.param.A.clearparamstemp()
```

Parameters Spatial Variation

Material parameters may be assigned a spatial variation where appropriate. There are four types of methods used to set a spatial variation: using *files*, *shapes*, *text equation*, or a built-in *generator*.

For scalar material parameters the spatial variation is the scaling to be applied to the base parameter value. Thus if A_0 is the set material parameter base value, then $a(x,y,z)$ is its set spatial variation, such that the material parameter as a function of position is $A(x,y,z) = A_0 a(x,y,z)$.

For vector material parameters (e.g. anisotropy axes) the spatial variation also requires 3 components, and rotates the base unit vector value. Thus if $\mathbf{e_0} = (e_{0x}, e_{0y}, e_{0z})$ is the base unit vector, and $\mathbf{n} = (n_x, n_y, n_z)$ is the set spatial variation, then the resultant vector used in computations is:

$$\begin{pmatrix} e_x \\ e_y \\ e_z \end{pmatrix} = \begin{pmatrix} n_x & -n_y / n_{xy} & -n_x n_z / n_{xy} \\ n_y & n_x / n_{xy} & -n_y n_z / n_{xy} \\ n_z & 0 & n_{xy} \end{pmatrix} \begin{pmatrix} e_{0x} \\ e_{0y} \\ e_{0z} \end{pmatrix}$$

Here $n_{xy} = (n_x^2 + n_y^2)^{0.5}$. Thus if the base unit vector is set to [1, 0, 0], the spatial variation gives directly the resultant vector direction.

1) To set a spatial variation using a text equation:

```
setparamvar (meshname) paramname equation text_equation
```

For details see User-Defined Text Equations. When using a text equation both a spatial variation and time dependence may be set.

2) To set a spatial variation using a file you can use either a *mask* file (png file) or an *ovf2* file:

```
setparamvar (meshname) paramname mask filename
```

or

```
setparamvar (meshname) paramname ovf2 filename
```

The *mask* file method uses a grayscale png image file where black means scaling value of 0 and white means scaling value of 1. This method only assigns an *x*, *y* dependence.

The *ovf2* file method specifies the scaling coefficients directly. For details see Working with OVF2 Files.

3) To set a spatial variation using a shape:

```
setparamvar (meshname) paramname shape shape_definition value
```

This method is mainly intended to be used from Python scripts since shape definitions are very long to type in manually, and can comprise single or composite shapes. Here a single multiplier value is set for the given shape. For details see Shapes and Regions.

4) To set a spatial variation using a generator:

```
setparamvar (meshname) paramname generator_name (arguments…)
```

This method uses built-in generators, where *generator_name* is one of the generators listed below. Each has a different set of arguments. If no arguments are given then default values are set.

random

```
setparamvar (meshname) paramname random min max seed
```

Uniform random variation between *min* and *max* with generator *seed* (≥ 0).

jagged

```
setparamvar (meshname) paramname jagged min max spacing seed
```

A set of uniform random values between *min* and *max* are generated on a square grid in the *xy* plane with given spacing (units m). A generator *seed* can also be given (≥ 0). The remaining points are obtained using bi-linear interpolation. This is useful for generating random jagged variations with a given in-plane correlation length (the spacing).

defects

```
setparamvar (meshname) paramname defects min max min_dia max_dia spacing seed
```

Circular defects are generated in the *xy* plane, at an average *spacing* (units m), and with diameters (units m) in the range *min_dia* to *max_dia*. A generator *seed* can also be given (≥ 0). The circular defects take on a value at the centre randomly between *min* and *max*. Within the defect the value changes linearly along the radius from value of 1 at the extremity to the centre value.

faults

```
setparamvar (meshname) paramname faults min_val max_val min_len max_len min_deg
max_deg spacing seed
```

Fault lines are generated in the *xy* plane, at an average *spacing* (units m), and with length (units m) in the range *min_len* to *max_len*. The fault lines orientation can vary in the plane between *min_deg* and *max_deg* (azimuthal angle in degrees). A generator *seed* can also be given (≥ 0).

vor2D

```
setparamvar (meshname) paramname vor2D min max spacing seed
```

Voronoi 2D tessellation in the *xy* plane, with a Voronoi cell size given by *spacing* (units m). Each Voronoi cell takes on a fixed value randomly between *min* and *max*. A generator *seed* can also be given (≥ 0).

vor3D

```
setparamvar (meshname) paramname vor3D min max spacing seed
```

Voronoi 3D tessellation, with a Voronoi cell size given by *spacing* (units m). Each Voronoi cell takes on a fixed value randomly between *min* and *max*. A generator *seed* can also be given (≥ 0).

vorbnd2D

```
setparamvar (meshname) paramname vorbnd2D min max spacing seed
```

Voronoi 2D tessellation in the *xy* plane, with a Voronoi cell size given by *spacing* (units m). Values vary randomly between *min* and *max* only at the boundaries of the Voronoi cells. A generator *seed* can also be given (≥ 0).

vorbnd3D

```
setparamvar (meshname) paramname vorbnd3D min max spacing seed
```

Voronoi 3D tessellation, with a Voronoi cell size given by *spacing* (units m). Values vary randomly between *min* and *max* only at the boundaries of the Voronoi cells. A generator *seed* can also be given (≥ 0).

vorrot2D

```
setparamvar  (meshname)  paramname  vorrot2D  minpol_deg  maxpol_deg  minazim_deg
maxazim_deg spacing seed
```

Voronoi 2D tessellation in the *xy* plane, with a Voronoi cell size given by *spacing* (units m). This is intended to be used for vectorial material parameters only (e.g. anisotropy axes), and in each Voronoi cell a fixed rotation is generated using polar and azimuthal angles, randomly in the range *minpol_deg* to *maxpol_deg* for polar angle in degrees, and *minazim_deg* to *maxazim_deg* for azimuthal angle in degrees. A generator *seed* can also be given (≥ 0).

vorrot3D

```
setparamvar  (meshname)  paramname  vorrot3D  minpol_deg  maxpol_deg  minazim_deg
maxazim_deg spacing seed
```

Voronoi 3D tessellation, with a Voronoi cell size given by *spacing* (units m). This is intended to be used for vectorial material parameters only (e.g. anisotropy axes), and in each Voronoi cell a fixed rotation is generated using polar and azimuthal angles, randomly in the range *minpol_deg* to *maxpol_deg* for polar angle in degrees, and *minazim_deg* to *maxazim_deg* for azimuthal angle in degrees. A generator *seed* can also be given (≥ 0).

abl_pol

```
setparamvar (meshname) paramname abl_pol ratio_-x ratio_+x ratio_-y ratio_+y
ratio_-z ratio_+z min max exponent
```

Absorbing boundary layer with polynomial slopes of given *exponent*. The slopes vary between *max* value at the mesh exterior boundary and *min* value in the interior. The slopes may be applied to –*x*, +*x*, –*y*, +*y*, –*z*, +z boundaries as specified by the *ratio* values. These are ratios of the mesh dimensions along the respective axes, with a value of zero meaning no slope is applied to the respective boundary. For example *ratio_-x* value of 0.1 means apply slope to the left for the first 10% of the mesh length *L*, so *max* value is taken at $x = 0$, and *min* value is taken at $x = l = 0.1 \times L$. Specifically, the following formula is used:

$$\begin{aligned} s(x) &= \frac{M-m}{(-l)^n}(x-l)^n + m, \quad 0 \le x \le l \\ s(x) &= m \qquad\qquad\qquad\qquad , \quad x \ge l \end{aligned}.$$

Here *M* is *max*, *m* is *min*, l = *ratio_-x* × *L*, *n* = *exponent*, etc.

abl_tanh

```
setparamvar (meshname) paramname abl_tanh ratio_-x ratio_+x ratio_-y ratio_+y
ratio_-z ratio_+z min max sigma_nm
```

Absorbing boundary layer with tanh slopes of given width specified by *sigma_nm*. The slopes vary between *max* value at the mesh exterior boundary and *min* value in the interior. The slopes may be applied to –*x*, +*x*, –*y*, +*y*, –*z*, +z boundaries as specified by the *ratio* values. These are ratios of the mesh dimensions along the respective axes, with a value of zero meaning no slope is applied to the respective boundary. For example *ratio_-x* value of 0.1 means apply slope to the left for the first 10% of the mesh length *L*, so *max* value is taken at $x = 0$, and *min* value is taken at $x = l = 0.1 \times L$. Specifically, the following formula is used:

$$\begin{aligned} s(x) &= \frac{M-m}{2\tanh(l/2\sigma)}\left[\tanh\left(-\frac{x-l/2}{\sigma}\right) + \tanh\left(\frac{l}{2\sigma}\right)\right] + m, \quad 0 \le x \le l \\ s(x) &= m \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad , \quad x \ge l \end{aligned}.$$

Here *M* is *max*, *m* is *min*, l = *ratio_-x* × *L*, σ = *sigma_nm* × 10^{-9}, etc.

abl_exp

```
setparamvar (meshname) paramname abl_exp ratio_-x ratio_+x ratio_-y ratio_+y
ratio_-z ratio_+z min max sigma_nm
```

Absorbing boundary layer with exponential slopes of given width specified by *sigma_nm*. The slopes vary between *max* value at the mesh exterior boundary and *min* value in the interior. The slopes may be applied to –*x*, +*x*, –*y*, +*y*, –*z*, +z boundaries as specified by the *ratio* values. These are ratios of the mesh dimensions along the respective axes, with a value of zero meaning no slope is applied to the respective boundary. For example *ratio_-x* value of 0.1 means apply slope to the left for the first 10% of the mesh length *L*, so *max* value is taken at *x* = 0, and *min* value is taken at *x* = *l* = 0.1 × *L*. Specifically, the following formula is used:

$$\begin{array}{ll} s(x) = \dfrac{M - m}{\exp(l/\sigma) - 1}\left[\exp\left(-\dfrac{x-l}{\sigma}\right) - 1\right] + m, & 0 \le x \le l \\ s(x) = m & , \quad x \ge l \end{array}.$$

Here *M* is *max*, *m* is *min*, *l* = *ratio_-x* × *L*, σ = *sigma_nm* × 10^{-9}, etc.

To remove a material parameter spatial variation use:

```
clearparamsvar (meshname) (paramname)
```

The above commands may be used in Python scripts through the param class in mesh objects, e.g.:

```
#create a ferromagnetic mesh with given rectangle and discretization
FM = ns.Ferromagnet(rect, cellsize)

#Now access spatial variation commands through the param class
#e.g. the lines below work for the exchange stiffness
FM.param.A.setparamvar(generator_name, parameters_list)
FM.param.A.clearparamsvar()
```

Simulation Schedules

A simulation may be started using the **run** command:

```
run
```

This will execute the currently configured simulation schedule. A schedule is defined by a number of *stages*. There are a number of different stages available, and each has a set *stopping* condition, and a set *data saving* condition. Stages are numbered sequentially starting from 0, and there must always be at least one stage set.

To set the stopping condition for a stage:

```
editstagestop index stoptype (stopvalue)
```

Here *index* is the stage number, *stoptype* is one of the available stopping conditions (see below) and *stopvalue* is the associated numerical value for that condition.

Available Stopping Conditions

nostop: stage will never finish.

iter: stage will finish when the number of iterations specified through *stopvalue* have completed.

mxh: stage will finish when the normalized maximum torque, calculated as $|\mathbf{M}\times\mathbf{H}|/M_S^2$, falls below that specified through *stopvalue* (for simulations with stochasticity the average normalized torque is used instead).

dmdt: stage will finish when the normalized maximum torque, calculated as $|\mathrm{d}\mathbf{M}/\mathrm{d}t|/M_S$, falls below that specified through *stopvalue* (for simulations with stochasticity the average normalized torque is used instead).

time: stage will finish when the set simulation time specified through *stopvalue* has elapsed.

mxh_iter: combination of *mxh* and *iter* conditions – stop when either condition is satisfied.

dmdt_iter: combination of *dmdt* and *iter* conditions – stop when either condition is satisfied.

Stage Types

There are several different types of stages (see below), and some may be associated with a particular mesh. Some stages also accept a parameter.

To set a single stage and erase all others:

```
setstage (meshname) stagetype
```

To append a stage to the simulation schedule:

```
addstage (meshname) stagetype
```

To delete a stage:

```
delstage index
```

When a new stage is defined, which requires some value to be set (e.g. a stage which sets a magnetic field), then a default value is first assumed. This can then be edited as:

```
editstagevalue index value
```

Relax: this is the default starting stage, is not associated with any mesh and does not set any values.

Hxyz: set a magnetic field in the named mesh using Cartesian coordinates (or focused mesh if name not given).
For example the following defines a stage with a 1 kA/m field set along the *x* direction:

```
setstage Hxyz
editstagevalue 0 1e3 0 0
```

In multi-mesh simulations the field may be applied to all meshes by specifying the *meshname* as *supermesh*:

```
setstage supermesh Hxyz
```

Hxyz_seq: set a field sequence from a start to an end magnetic field value, given in Cartesian coordinates, in a number of steps.

For example the following defines a field sequence from -1 kA/m to +1 kA/m in 10 steps (so field step is 2 kA/m / 10):

```
setstage Hxyz_seq
editstagevalue 0 -1e3 0 0 1e3 0 0 10
```

For such multi-step stages, the stopping condition applies to each step.

Hpolar_seq: set a field sequence from a start to an end magnetic field value, given in polar coordinates with angles in degrees, in a number of steps.

For example the following defines a field sequence from -1 kA/m to +1 kA/m in 10 steps along the 30 degrees *xy*-plane azimuthal angle:

```
setstage Hpolar_seq
editstagevalue 0 -1e3 90 30 1e3 90 30 10
```

Hfmr: set a bias field and r.f. field for FMR simulations. Note: this is deprecated now, and whilst it may still be used the recommended approach is to use the *Hequation* stage instead. The r.f. field is discretized in a number of steps, s_d, and a number of r.f. cycles may be set. This stage type should be used with a time stopping condition. Since this is a multi-step stage, the stopping condition applies to each step. Thus if an r.f. frequency f is required, then the time stopping condition should be: $t_S = (1 / f) / s_d$.

For example the following defines a bias field of 1 MA/m along the *y* axis, with an r.f. excitation of 100 A/m amplitude along the *x* axis, lasting for 1000 cycles with a frequency of 10 GHz, and each cycle discretized in 20 steps:

```
setstage Hfmr
editstagevalue 0 0 1e6 0 100 0 0 20 1000
editstagestop 0 time 5e-12
```

Hequation: field specified using a vector text equation.

For example the FMR example above would be specified as:

```
setstage Hequation
editstagevalue 0 100*sin(2*PI*10e9*t), 1e6, 0
editstagestop 0 time 100e-9
```

Hfile: field specified using an input text file. The file must contain tab-separated columns as 1) first column specifies time values in seconds, increasing order, 2) following columns specify values to set, so requires 3 columns for the 3 field components. This stage should use a *time* stopping condition. The number of steps in this stage is determined by the maximum time value in the file (so last row entry) divided by the *time* stopping condition – i.e. each step lasts for the set *time* stopping condition which determines the resolution. Every new step sets a new field value; if the time values specified in the file are coarser, then linear interpolation is used.

V, V_seq, Vequation, Vfile: as for *H* but sets electrode potential difference. Mesh name not applicable.

I, I_seq, Iequation, Ifile: as for *H* but sets electrode electrical current. Mesh name not applicable.

T, T_seq, Tequation, Tfile: as for *H* sets mesh base temperature. As for *H* can use *supermesh* to set base temperature for all meshes.

Q, Q_seq, Qequation, Qfile: as for *T* but sets heat source.

Sunif: set uniform stress.

MonteCarlo: run a Mone-Carlo computation, either in an atomistic mesh, or micromagnetic mesh – for details see Monte Carlo Algorithms.

Finally, stages may also be configured to save data – see Output Data for details.

At any point to stop the simulation from the console use the **stop** command. Another useful command is **reset**, which sets the stage-step (*sstep*) counter to zero, iterations (*iter*) counter to zero, and also the time (*time*).

Stages in Python Scripts

Rather than using separate commands to set, add, and edit stage stop and data saving conditions, stage objects are available which make the Python scripts simpler. These are shown below. When called, these stages are executed immediately. In the general form below round brackets indicate optional arguments.

Relax:

General form:

```
ns.Relax([(stop_condition, (stop_value)), (save_condition, (save_value))])
```

Example:

```
ns.Relax([‘mxh_iter’, [1e-5, 10000], ‘iter’, 100])
```

Hxyz:

General form:

```
ns.Hxyz([Mesh, stage_value, (stop_condition, (stop_value)), (save_condition,
(save_value))])
```

Example:

```
FM = ns.Ferromagnet(rectangle, cellsize)
ns.Hxyz([FM, [1e4, 1e4, 0], ‘mxh_iter’, [1e-5, 10000], ‘iter’, 100])
```

or:

```
ns.Hxyz([‘supermesh’, [1e4, 1e4, 0], ‘mxh_iter’, [1e-5, 10000], ‘iter’, 100])
```

Hxyz_seq:

General form:

```
ns.Hxyz_seq([Mesh, stage_value, (stop_condition, (stop_value)), (save_condition,
(save_value))])
```

Hpolar_seq:

General form:

```
ns.Hpolar_seq([Mesh, stage_value, (stop_condition, (stop_value)), (save_condition,
(save_value))])
```

Hfmr:

General form:

```
ns.Hfmr([Mesh, stage_value, (stop_condition, (stop_value)), (save_condition,
(save_value))])
```

Hequation:

General form:
```
ns.Hequation([Mesh, stage_value, (stop_condition, (stop_value)), (save_condition,
(save_value))])
```

Hfile:

General form:
```
ns.Hfile([Mesh, stage_value, (stop_condition, (stop_value)), (save_condition,
(save_value))])
```

V, V_seq, Vequation, Vfile:

General form:
```
ns.V([stage_value, (stop_condition, (stop_value)), (save_condition,
(save_value))])
```

I, I_seq, Iequation, Ifile:

General form:
```
ns.I([stage_value, (stop_condition, (stop_value)), (save_condition,
(save_value))])
```

T, T_seq, Tequation, Tfile:

General form:
```
ns.T([Mesh, stage_value, (stop_condition, (stop_value)), (save_condition,
(save_value))])
```

Q, Q_seq, Qequation, Qfile:

General form:
```
ns.Q([Mesh, stage_value, (stop_condition, (stop_value)), (save_condition,
(save_value))])
```

Sunif:

General form:
```
ns.Sunif([Mesh, stage_value, (stop_condition, (stop_value)), (save_condition,
(save_value))])
```

MonteCarlo:.

General form:
```
ns.MonteCarlo([acceptance_rate, (stop_condition, (stop_value)), (save_condition,
(save_value))])
```

Monte Carlo Algorithms

Boris uses parallelized Monte Carlo algorithms in atomistic meshes (see [S. Lepadatu et al., JMMM 540, 168460 (2021)]), and a micromagnetic Monte Carlo algorithm implemented for FM and AFM meshes [S. Lepadatu, J. Appl. Phys. 130, 163902 (2021)]. Note, the demagnetizing interaction and dipole-dipole interactions can be included in the parallelized Monte Carlo algorithms. The Monte Carlo algorithms may also be used at 0 K as energy minimizers.

Atomistic Monte Carlo

To use Monte Carlo then set a *MonteCarlo* stage. For micromagnetic meshes this requires a Curie temperature to be set (**curietemperature**), and in all cases a non-zero temperature should be set (**temperature**).

There are serial and parallel versions for the atomistic meshes, selected using the **mcserial** command. By default the parallel version is enabled, and the serial version should only be used for testing. To switch between the classical and constrained version in atomistic meshes use the **mcconstrain** command in the required mesh (different meshes can use different types of Monte Carlo algorithm, including with different constraining directions):

To set the classical Monte Carlo algorithm (the default):

```
mcconstrain (meshname) 0
```

To set the constrained Monte Carlo algorithm specify the constraining direction as a unit vector, e.g. along the *x* axis:

```
mcconstrain (meshname) 1 0 0
```

You can also disable the Monte Carlo algorithm in some meshes if needed (*status* = 1):

```
mcdisable (meshname) status
```

When running a *MonteCarlo* stage, iteration of the dynamics equation is disable so effective field are no longer computed. This means energy density output data is no longer available. If you want to have these data computed then use (*status* = 1):

```
mccomputefields status
```

When enabled this slows down the iteration; when using micromagnetic meshes with the *demag* module set this is automatically enabled.

The stage value for *MonteCarlo* is the target acceptance rate, set to 0.5 by default. This can be adjusted, e.g. the following sets a 0.6 target acceptance rate:

```
setstage MonteCarlo
editstagevalue 0 0.6
```

During iteration the trial move cone angle is adjusted in order to reach the set target acceptance rate. The minimum and maximum allowed cone angles can be controlled as:

```
mcconeangle min_angle_deg max_angle_deg
```

Thus in order to set a fixed cone angle, set the minimum and maximum allowed values to be equal.

The set cone angle and actual acceptance rate are available in the *MCparams* output data – see Output Data.

Micromagnetic Monte Carlo

In **FM** and **AFM** meshes, the MonteCarlo stage will use the micromagnetic Monte Carlo algorithm introduced in [S. Lepadatu, J. Appl. Phys. 130, 163902 (2021)]. This differs from the usual atomistic Monte Carlo algorithm, in that in addition to a rotation trial move, a magnetization length compound trial move is also introduced. This is based on the Maxwell-Boltzmann distribution of the micromagnetic free energy *F*:

$f(\mathbf{m}) = m^2 \exp\left(-F(\mathbf{m})/k_B T\right)/Z$, where $Z = \sum_i m_i^2 \exp\left(-F(\mathbf{m}_i)/k_B T\right)$

Here $\mathbf{m} = \mathbf{M}/M_{S0}$. In particular the magnetization length probability distribution is given by (Z_l is a renormalization factor):

$$f_l(m) = \frac{m^2}{Z_l}\begin{cases} \exp\left(-\dfrac{VM_{S0}}{8\tilde{\chi}_{\parallel} m_e^2 k_B T}\left(m^2 - m_e^2\right)^2\right) & ,T < T_C \\ \exp\left(-\dfrac{VM_{S0} m^2}{2\tilde{\chi}_{\parallel} k_B T}\left(1 + \dfrac{3T_C m^2}{10(T - T_C)}\right)\right) & ,T > T_C \end{cases}$$

For symbol defitions see the LLB equation in Differential Equations section.

This leads to a modified trial move acceptance formula for **FM** as (see the publication for details):

$$P_{accept}(A \to B) = min\left\{1, \frac{m_B^4}{m_A^4}\exp\left(-\Delta F / k_B T\right)\right\}, \text{ where } \Delta F = F(\mathbf{m}_B) - F(\mathbf{m}_A).$$

For **AFM** meshes, trial moves are considered on the two sub-lattices in pairs, with acceptance formulas and procedure detailed below.

1) Move $\mathbf{M}_A$ to $\tilde{\mathbf{M}}_A$:

   $\Delta F_A = F\left(\tilde{\mathbf{M}}_A, \mathbf{M}_B\right) - F\left(\mathbf{M}_A, \mathbf{M}_B\right)$

2) Accept/reject:

   $P_{accept,A} = min\left\{1, \frac{\tilde{M}_A^4}{M_A^4}\exp\left(-\Delta F_A / k_B T\right)\right\}$

3) Move $\mathbf{M}_B$ to $\tilde{\mathbf{M}}_B$:

   $\Delta F_B = F\left(\mathbf{M}_A, \tilde{\mathbf{M}}_B\right) - F\left(\mathbf{M}_A, \mathbf{M}_B\right)$

4) Accept/reject:

   $P_{accept,B} = min\left\{1, \frac{\tilde{M}_B^4}{M_B^4}\exp\left(-\Delta F_B / k_B T\right)\right\}$

Here the micromagnetic free energy is a function of both $\mathbf{M}_A$ and $\mathbf{M}_B$, and in particular the magnetization length probability distribution is proportional to:

$$P_i\left(m_A, m_B\right) \propto m_i^2 \exp\left\{-\frac{M_{S,i}^0 V}{4\mu_i m_{e,i} k_B T}\left[\frac{\left(m_i^2 - m_{e,i}^2\right)^2}{m_{e,i}} \frac{\left(\mu_i + 3\tau_{ij} k_B T_N \tilde{\chi}_{\parallel,j}\right)}{2\tilde{\chi}_{\parallel,i}} + \frac{\left(m_j^2 - m_{e,j}^2\right)}{m_{e,j}} 3\tau_{ij} k_B T_N m_i^2\right]\right\}$$

$(i, j = A, B, i \neq j)$

For symbol defitions see the two-sublattice LLB equation in Differential Equations section.

Simulation Algorithms

Domain Wall Movement

The domain wall movement algorithm has been described in detail in Tutorial 5 – Domain Wall Movement and Data Processing section.

Dipole Shifting

Dipole meshes may be shifted during a simulation, which allows computations with a shifting stray field pattern. The supermesh **strayfield** module should be enabled, or the **mstrayfield** module enabled in each mesh where a stray field is required.

Dipole shifting can be achieved efficiently at runtime using 1 of 2 methods (for details see description of respective commands in the Commands section).

1) **shiftdipole** command

   This allows a single Dipole mesh to be shifted by the required x, y, z amount. For efficiency this method should only be used with embedded Python scripts.

2) **dipolevelocity** command

   It is possible to assign velocities for Dipole meshes at the set-up stage, which will allow them to be shifted accordingly during a simulation.

Global Field Shifting

A global magnetic field may be set using the **loadovf2field** command (set on the *supermesh*). This may later be cleared using the **clearglobalfield** command.
The loaded global field pattern may be shifted using the **shiftglobalfield** command (for efficiency, this should only be used with embedded Python scripts).

Data Extraction Algorithms

Skyshift Algorithm

This is available through the *skyshift* output data (see Output Data section). This algorithm allows tracking of a skyrmion in a given mesh, by defining a rectangle relative to that mesh, such that the skyrmion is roughly in the centre of this rectangle. The output of this algorithm are the *x* and *y* shifts of the skyrmion, relative to the starting position.

Any number of skyrmions may be tracked in this way, by adding multiple *skyshift* output data. This requires the different *skyshift* output data to have different initial rectangles.

The algorithm works by dividing the defined rectangle into quadrants. The *z* direction average magnetization is computed separately in these quadrants, and the rectangle position is adjusted in order to keep these values the same (which occurs when the skyrmion is centred). Thus the rectangle shift becomes the skyrmion shift output data.

Skypos Algorithm

This is an alternative skyrmion tracking algorithm, which additionally allows obtaining skyrmion diameters, as well as tracking skyrmions with changing size. As with the *skyshift* data, the *skypos* output data (see Output Data section) requires a given mesh, and an initial rectangle relative to that mesh, such that the skyrmion is roughly in the centre of this rectangle. Any number of skyrmions may be tracked in this way, by adding multiple *skypos* output data. This requires the different *skypos* output data to have different initial rectangles.

The output data are the skyrmion *x* and *y* positions relative to the containing mesh, as well as the skyrmion *x* and *y* diameters.

The algorithm works by analysing horizontal (*x*) and vertical (*y*) profiles through the centre of the defined rectangle. Magnetization *z* component crossing points are detected, which allows extraction of skyrmion centre positions along *x* and *y* directions, as well as respective diameters. The tracking rectangle position is adjusted in order to keep the skyrmion in the centre of the rectangle. The tracking rectangle size is also adjusted depending on the size of the skyrmion. This is determined by a multiplicative factor, set using the **skyposdmul** command. The default value is 2.0, which means the tracking rectangle sizes along *x* and *y* are set to twice the skyrmion diameters along *x* and *y* respectively. If multiple densly packed skyrmions are being tracked, it may be necessary to reduce this value, but it is recommended this should exceed a multiplier of 1.2 otherwise loss of tracking can result.

It is recommended the *skypos* algorithm be used only ay 0 K, since thermal fluctuations will result in loss of skyrmion tracking due to multiple crossing points. If thermal fluctuations are included, either the *skyshift* algorithm should be used, or a custom *skypos*-type algorithm may be implemented which includes averaging over many iterations. Such an algorithm may best be implemented using embedded Python scripts.

Dwpos algorithm

It is also possible to track individual domain walls using the *dwpos_x*, *dwpos_y*, and *dwpos_z* output data (see Output Data section), which obtain the domain wall centre position and width.

A rectangle is required, and a domain wall profile will be extracted through the rectangle centre (along *x* for *dwpos_x*, along *y* for *dwpos_y*, and along *z* for *dwpos_z*), where each profile point is obtained as the average in the transverse rectangle cross-section. The appropriate magnetization component in the extracted profile is then fitted using a *tanh* function, namely $\tanh(-\pi(x-x_0)/\Delta)$, where x_0 is the domain wall centre position and Δ its width.

It is possible to specify which magnetization component is used for *tanh* fitting, using the **dwposcomponent** command. By default the *tanh* component is automatically detected, but it is also possible to specify directly the *x*, *y*, or *z* components.

Demagnetizing Field Polynomial Extrapolation

In order to speedup dynamical simulations when using higher order explicit evaluation methods, it is possible to approximate the demagnetizing fields using polynomial extrapolation at sub-steps of the evaluation method. For details see [S. Lepadatu, IEEE Trans. Mag. 58, 1 (2022)]. Thus, the demagnetizing field need only be computed once per iteration, and with the correct polynomial approximation no, or negligible, solution error should arise.

This approach is disabled by default, with full computation of the demagnetizing field at all sub-steps of the evaluation method. To enable it use the following command to set the required approximating polynomial order (in general the polynomial order should match the evaluation method order – see [S. Lepadatu, IEEE Trans. Mag. 58, 1 (2022)]):

```
evalspeedup level
```

level 0: no speedup (default)

level 1: step, polynomial order 0.

level 2: linear, polynomial order 1.

level 3: quadratic, polynomial order 2.

level 4: cubic, polynomial order 3.

level 5: quartic, polynomial order 4.

level 6: quintic, polynomial order 5.

It’s also possible, although not recommended in general, to manually control the time step at which the demagnetizing field is computed (**setdtspeedup** command). In general this is linked to the differential equation evaluation method time step (**linkdtspeedup** command).

Output Data

Output data may be obtained in the following ways. 1) The simulation schedule may be configured to extract output data automatically during a simulation. 2) Output data may also be obtained in interactive mode from the console. 3) Python scripts may be used to issue commands which output data to files, or as return parameters.

1) Saving data during a simulation schedule.

Data saving conditions may be set for stages (see Simulation Schedules) as:

```
editdatasave index savetype (savevalue)
```

During simulations a data save is triggered when the set condition is met. There are a few data save types available (*savetype*), detailed below:

none: no data save set.

stage: save at the end of every stage.

step: save at the end of every step in this stage.

iter: save every given number of iterations. For example the following saves every 100 iterations in the first stage:

```
editdatasave 0 iter 100
```

time: save at given time interval. For example the following saves every 100ps in the first stage:

```
editdatasave 0 time 100ps
```

When data saving is triggered an output data file is used (*out_data.txt* by default), which is set using:

```
savedatafile (directory/)filename
```

The data to be saved is contained in a list, which may be configured as follows.

To set a single output data entry and erase all others:

```
setdata (meshname) dataname (rectangle)
```

Here, *dataname* is the name of a possible output data field. Possible names are given below. The different data types may be applicable to a given mesh, and have a rectangle set (e.g. averaging operations).

To append a new output data entry:

```
adddata (meshname) dataname (rectangle)
```

To delete an existing output data entry:

```
deldata index
```

Here, output data entries are contained sequentially with *index* number starting from zero. The default configuration contains, in this order, 0: *sstep*, 1: *iter*, 2: *time*, 3: *Ha* for the default *permalloy* mesh, 4: <*M*> for the default *permalloy* mesh with the entire mesh rectangle.

During a data saving event, the configured list is saved to the set output data file in a tab-spaced row. Different output data types have different number of component, e.g. *iter* is the iteration number and has just one component, *sstep* is the stage-step value and has 2 components, <*M*> is the average magnetization in the configured rectangle relative to the given mesh and has 3 components.

The different output data types are listed below.

sstep: (2 components); stage-step.

time: (1 component); simulation time.

stime: (1 component); stage time.

iter: (1 component); simulation iteration number.

siter: (1 component); stage iteration number.

dt: (1 component); evaluation time-step.

heat_dT: (1 component); heat equation time-step.

mxh: (1 component); normalized torque as $|\mathbf{M}\times\mathbf{H}|/M_S^2$.

dmdt: (1 component); normalized torque as $|\mathrm{d}\mathbf{M}/\mathrm{d}t|/M_S$.

Ha: (3 components, needs mesh name); applied field.

<M>: (3 components, needs mesh name and rectangle relative to mesh); average magnetization in given rectangle.

<M>th: (3 components, needs mesh name and rectangle relative to mesh); thermodynamic average magnetization in given rectangle. (*experimental, do not use*)

<M2>: (3 components, needs mesh name and rectangle relative to mesh); average magnetization in given rectangle, from sub-lattice B for 2-sublattice meshes.

|M|mm: (2 components, needs mesh name and rectangle relative to mesh); minimum-maximum magnetization length in given rectangle.

Mx_mm: (2 components, needs mesh name and rectangle relative to mesh); minimum-maximum magnetization *x* component in given rectangle.

My_mm: (2 components, needs mesh name and rectangle relative to mesh); minimum-maximum magnetization *y* component in given rectangle.

Mz_mm: (2 components, needs mesh name and rectangle relative to mesh); minimum-maximum magnetization *z* component in given rectangle.

MCparams: (2 components, needs mesh name); Monte Carlo algorithm cone angle and acceptance rate.

<Jc>: (3 components, needs mesh name and rectangle relative to mesh); average current density in given rectangle.

<Jsx>: (3 components, needs mesh name and rectangle relative to mesh); average *x*-direction spin current in given rectangle.

<Jsy>: (3 components, needs mesh name and rectangle relative to mesh); average *y*-direction spin current in given rectangle.

<Jsz>: (3 components, needs mesh name and rectangle relative to mesh); average *z*-direction spin current in given rectangle.

<mxdmdt>: (3 components, needs mesh name and rectangle relative to mesh); average $\mathbf{m} \times \mathrm{d}\mathbf{m}/\mathrm{dt}$, where $\mathbf{m} = \mathbf{M} / M_s$, in given rectangle.

<dmdt>: (3 components, needs mesh name and rectangle relative to mesh); average $\mathrm{d}\mathbf{m}/\mathrm{dt}$, where $\mathbf{m} = \mathbf{M} / M_s$, in given rectangle.

<mxdmdt2>: (3 components, needs mesh name and rectangle relative to mesh); average $\mathbf{m}_B \times \mathrm{d}\mathbf{m}_B/\mathrm{dt}$, where $\mathbf{m}_B = \mathbf{M}_B / M_{s,B}$, in given rectangle for sub-lattice B when using 2-sublattice mesh.

<dmdt2>: (3 components, needs mesh name and rectangle relative to mesh); average $\mathrm{d}\mathbf{m}_B/\mathrm{dt}$, where $\mathbf{m}_B = \mathbf{M}_B / M_{s,B}$, in given rectangle for sub-lattice B when using 2-sublattice mesh.

<mxdm2dt>: (3 components, needs mesh name and rectangle relative to mesh); average $\mathbf{m}_A \times \mathrm{d}\mathbf{m}_B/\mathrm{dt}$, where $\mathbf{m}_{A,B} = \mathbf{M}_{A,B} / M_{s,A,B}$, in given rectangle for sub-lattice B when using 2-sublattice mesh.

<*m2xdmdt*>: (3 components, needs mesh name and rectangle relative to mesh); average $\mathbf{m}_B \times d\mathbf{m}_A/dt$, where $\mathbf{m}_{A,B} = \mathbf{M}_{A,B} / M_{s,A,B}$, in given rectangle for sub-lattice B when using 2-sublattice mesh.

<*V*>: (1 component, needs mesh name and rectangle relative to mesh); average charge potential in given rectangle.

<*S*>: (3 components, needs mesh name and rectangle relative to mesh); average spin accumulation in given rectangle.

<*elC*>: (1 component, needs mesh name and rectangle relative to mesh); average electrical conductivity in given rectangle.

<*T*>: (1 component, needs mesh name and rectangle relative to mesh); average temperature in given rectangle.

<*T_l*>: (1 component, needs mesh name and rectangle relative to mesh); average lattice temperature in given rectangle, when using 2-temperature model.

V: (1 component); electrodes potential difference.

I: (1 component); ground electrode charge current.

R: (1 component); electrical resistance.

<*u*>: (3 components, needs mesh name and rectangle relative to mesh); average mechanical displacement in given rectangle.

<*Sd*>: (3 components, needs mesh name and rectangle relative to mesh); average linear strain in given rectangle (components as ε_{xx}, ε_{yy}, ε_{zz}).

<*Sod*>: (3 components, needs mesh name and rectangle relative to mesh); average shear strain in given rectangle (components as ε_{yz}, ε_{xz}, ε_{xy}).

e_demag: (1 component, needs mesh name and rectangle relative to mesh); average demagnetizing energy density in given rectangle.

e_exch: (1 component, needs mesh name and rectangle relative to mesh); average exchange energy density in given rectangle.

t_exch: (3 components, needs mesh name and rectangle relative to mesh); average exchange interaction torque in given rectangle, to be used in atomistic meshes only.

e_surfexch: (1 component, needs mesh name and rectangle relative to mesh); average surface exchange energy density in given rectangle.

t_surfexch: (3 components, needs mesh name and rectangle relative to mesh); average surface exchange interaction torque in given rectangle, to be used in atomistic meshes only.

e_zee: (1 component, needs mesh name and rectangle relative to mesh); average Zeeman energy density in given rectangle.

t_zee: (3 components, needs mesh name and rectangle relative to mesh); average applied field torque in given rectangle, to be used in atomistic meshes only.

e_stray: (1 component, needs mesh name and rectangle relative to mesh); average dipole stray field energy density in given rectangle.

t_stray: (3 components, needs mesh name and rectangle relative to mesh); average dipole stray field torque in given rectangle, to be used in atomistic meshes only.

e_mo: (1 component, needs mesh name and rectangle relative to mesh); average magneto-optical energy density in given rectangle.

e_mel: (1 component, needs mesh name and rectangle relative to mesh); average magneto-elastic energy density in given rectangle.

e_anis: (1 component, needs mesh name and rectangle relative to mesh); average magneto-crystalline anisotropy energy density in given rectangle.

t_anis: (3 components, needs mesh name and rectangle relative to mesh); average magneto-crystalline anisotropy interaction torque in given rectangle, to be used in atomistic meshes only.

e_rough: (1 component, needs mesh name and rectangle relative to mesh); average roughness module energy density in given rectangle.

e_total: (1 component); total energy density.

dwshift: (1 component); domain wall shift when using domain wall movement algorithm (see Tutorial 5 – Domain Wall Movement and Data Processing).

dwpos_x: (2 components, needs mesh name and rectangle relative to mesh); Domain wall position and width if applicable. A profile is extracted through the set rectangle along the *x* direction, then a *tanh* fitting function is used to obtain the domain wall position and width. The magnetization component fitted ia controlled using the **dwposcomponent** command.

dwpos_y, *dwpos_z*: As for *dwpos_x*, but the profile is extracted along the *y*, respectively *z* direction.

skyshift: (2 components, needs mesh name and rectangle relative to mesh); Skyrmion *xy* position is extracted using the skyshift algorithm in the given starting relative rectangle. The rectangle tracks the skyrmion, thus multiple skyrmions may be tracked by assigning multiple *skyshift* output data with different initial rectangles. For further details see Tutorial 23 – Skyrmion Movement with Spin Currents. Note: this is considered obsolete and you should use *skypos* instead.

skypos: (4 components, needs mesh name and rectangle relative to mesh); Skyrmion *xy* position and *xy* diameters are extracted using the skypos algorithm in the given starting relative rectangle. The rectangle tracks the skyrmion, thus multiple skyrmions may be tracked by assigning multiple *skypos* output data with different initial rectangles. For further details see Tutorial 23 – Skyrmion Movement with Spin Currents. To fine-tune this algorithm particularly for tracking multiple skyrmions which may be close together see the **skyposdmul** command.

Q_topo: (1 component, needs mesh name and rectangle relative to mesh); Topological charge computed in given rectangle. Also see the related **dp_topocharge** command.

v_iter: (1 component); Spin-transport solver, number of iterations for *V* convergence (potential).

s_iter: (1 component); Spin-transport solver, number of iterations for **S** convergence (spin accumulation).

ts_err: (1 component); Spin-transport solver achieved convergence error.

TMR: (1 component, needs mesh name and rectangle relative to mesh); Tunneling magneto-resistance in an insulator mesh separating two ferromagnetic meshes.

commbuf: (0 components); This is a special entry which doesn't save any data, but triggers executation of the command buffer which allows more complex and/or pre-processed data to be saved during a simulation schedule, e.g. profiles, ovf files, etc. – for details see Command Buffering.

It's also possible to configure the simulation schedule to save image files, e.g. for creating videos. You can set an image file base using the **saveimagefile** command. To enable automatic image saving then set the image save flag using **saveimageflag** command.

2) Interactive data output

With any of the data types above, the output may be obtained at any time using the **showdata** command:

```
showdata dataname (meshname, (rectangle))
```

There are a number of commands which extract output data. These include (see command descriptions for details): **savemeshimage**, **saveovf2mag**, **saveovf2**, **dp_getexactprofile**, **averagemeshrect**, **dp_topocharge**, **dp_calctopochargedensity**, **dp_histogram**, **dp_histogram2**, **dp_anghistogram**.

3) Data output in Python scripts

Any of the commands above may be using from a Python script. Some have return parameters which may be read (see command descriptions). For example with **showdata**:

```
data = ns.showdata(dataname, meshname, rectangle)
```

As a specific example the following obtains the average magnetization in a 20 nm cube:

```
data = ns.showdata('<M>', [0, 0, 0, 20e-9, 20e-9, 20e-9])
```

There is also an easy method of configuring output data during a simulation using the *setsavedata* method in NSClient:

```
ns.setsavedata(filename, [data_name, (Mesh, (rectangle))], ...)
```

For example the following configures to save the time and average magnetization in part of a mesh:

```
FM = ns.Ferromagnet(rectangle, cellsize)
ns.setsavedata(fileName, [‘time’], [‘<M>’, FM, sub_rectangle])
```

It is also possible to extract data from physical mesh quantities using a number of built-in commands. For details see the Mesh Display section, Mesh quantities output in Python scripts subsection.

Command Buffering

NOTE: The command buffering technique is considered largely obsolete since v3.8. Whilst it is still available, the recommended approach is to use embedded Python scripts when more complex data types need to be extracted during a simulation. See Tutorial 9 – Scripting using Python for help with embedded Python scripts.

When using Python scripts to control simulations, sometimes more complex data types need to be extracted, which is not normally possible with a simulation schedule. For example you may want to obtain magnetization profiles at fixed time intervals, Δt, or save the entire magnetization configuration to ovf files periodically. This is possible with a Python script by setting a single stage which lasts for time Δt, saving any required data from the script, then looping until finished. The problem with this approach, if Δt is small so data needs to be saved often, it can be slow due to communication time between the script and Boris. If the same set of commands needs to be executed each time data is saved, then it's possible to buffer these commands in Boris, so whenever a data save is triggered the command buffer is executed.

When using NSClient in a Python script, all commands have a parameter called *bufferCommand*. By default this is *False*, which means the command is executed straight away. If *bufferCommand* = *True*, then instead of executing the command, it is appended to the execution buffer (from the console this is the **buffercommand** command). The command buffer may be executed at any point by calling the **runcommbuffer** command. Alternatively the *commbuf* output data may be set so the command buffer is executed during a simulation schedule whenever a data save is triggered. The buffer may be cleared by using the **clearcommbuffer** command.

As an example the following script saves the entire magnetization configuration to ovf files every 10 iterations using the *commbuf* output data, as well as the iteration, time, and average magnetization to an output text file:

```
from NetSocks import NSClient

ns = NSClient(); ns.configure(True)

ns.setdata('iter')
ns.adddata('time')
ns.adddata('<M>')
ns.adddata('commbuf')
ns.savedatafile('simdata.txt')

ns.editstagestop(0, 'time', 10e-9)
ns.editdatasave(0, 'iter', 10)

ns.saveovf2mag('mag_%iter%.ovf', bufferCommand = True)

ns.Run()
```

Another thing to notice here is the name of the output ovf file : ‘mag_%iter%.ovf’. Here ‘%*iter*%’ means replace ‘%*iter*%’ by the actual value of the *iter* data. Thus the above command will save the magnetization to files numbered sequentially as mag_0.ovf, mag_10.ovf, mag_20.ovf, etc. This works in general, and ‘%*dataname*%’ may be used with most commands which work with input/output files, where *dataname* is any of the output data names as listed in Output Data.

As a further example see the solutions to Tutorial 30 – Spin-Wave Dispersion, where command buffering is used to extract magnetization profiles and append them to an output file in a single simulation schedule run.

Effective Fields and Energies Spatially Resolved Output Data

It is possible to obtain the energy density and effective fields from individual modules with full spatial resolution, rather than just average values. Each module typically computes a different contribution to the total effective field. The total effective field is available under the *Heff* display (see Mesh Display). Once displayed it can be saved for later analysis using the **saveovf2** command, or profiles extracted from the display (**dp_getexactprofile**), etc.

In order to obtain information from invididual modules this needs to be configured using:

```
displaymodule (meshname) modulename
```

Here *modulename* is the name of the required module (see Modules). Once configured, then the module effective field will be displayed when *Heff* is selected for display (**display** (*meshname*) *Heff*). At this point data may be extracted as explained above. In order to get the energy density with spatial resolution then *Ed* needs to be displayed instead (**display** (*meshname*) *Ed*).

Once a module is selected in this way, the module data needs to be recomputed so the effective fields are refreshed:

```
computefields
```

Only one module may be configured in this way at any one time, thus if the effective field or energy density is required from multiple modules this procedure needs to be applied to each in turn.

The module effective field display may be turned off by using:

```
displaymodule (meshname) none
```

For example the following saves the demagnetizing field and demagnetizing energy density to ovf files:

```
displaymodule demag
computefields
display Heff
saveovf2 Hdemag.ovf
display Ed
saveovf2 Edemag.ovf
```

Mesh Display

The mesh viewer may be configured using the **display** command as:

```
display (meshname) name
```

This sets the physical quantity to be displayed in the mesh viewer for the given mesh (different meshes can be configured to display different things). The possible entries that can be specified with *name* are:

Nothing: don’t display anything for this mesh.

M: magnetization.

M2: magnetization of sub-lattice B for 2-sublattice meshes.

M12: magnetization, showing simultaneously sub-lattices A and B for 2-sublattice meshes.

mu: magnetic moments for atomistic meshes.

Heff: total or module effective field (see Effective Fields and Energies Spatially Resolved Output Data)

Heff2: total or module effective field of sub-lattice B for 2-sublattice meshes (see Effective Fields and Energies Spatially Resolved Output Data)

Heff12: total or module effective field showing simultaneously sub-lattices A and B for 2-sublattice meshes (see Effective Fields and Energies Spatially Resolved Output Data)

Ed: module energy density spatial variation (see Effective Fields and Energies Spatially Resolved Output Data)

Ed2: module energy density spatial variation of sub-lattice B for 2-sublattice meshes (see Effective Fields and Energies Spatially Resolved Output Data)

Jc: current density.

V: charge potential.

elC: electrical conductivity.

S: spin accumulation.

Jsx: x-direction spin current.

Jsy: y-direction spin current.

Jsz: z-direction spin current.

Ts: bulk spin torque obtained when running the spin-transport solver.
Tsi: interfacial spin torque obtained when running the spin-transport solver.

Temp: temperature.

u: mechanical displacement.

S_d: strain tensor, diagonal components.

S_od: strain tensor, off-diagonal components.

*ParamVa*r: parameter spatial variation. To select which parameter spatial variation is to be displayed use the **setdisplayedparamsvar** command.

Roughness: applied mesh roughness (see Tutorial 24 – Roughness and Staircase Corrections).

Cust_V: specialcustom display of vectorial quantities, used with some commands.

Cust_S: special display of scalar quantities, used with some commands.

The mesh display handles both scalar and vectorial quantites. For vectorial quantities it is possible to display only one of the components. To do this use the **vecrep** command:

```
vecrep (meshname) vecreptype
```

vecreptype = 0, full.

vecreptype = 1, x component only.

vecreptype = 2, y component only.

vecreptype = 3, z component only.

vecreptype = 4, vector direction only.

vecreptype = 5, vector magnitude only.

It's also possible to display quantities computed on the supermesh by setting *meshname* to *supermesh* in the **display** command. The possible entries that can be specified with *name* on the *supermesh* are:

Nothing: don't display anything on supermesh.

Hdemag: demagnetizing field from *sdemag* module when **multiconvolution** 0.

HOe: Oersted field.

Hstray: stray field from dipole meshes.

Finally, the mesh viewer rotation, scaling etc., possible by using the mouse, may also be controlled using the commands: **rotcamaboutorigin**, **rotcamaboutaxis**, **adjustcamdistance**, **shiftcamorigin**.

Mesh quantities output in Python scripts

The above physical quantities are associated with meshes, and they can be accessed using the *quant* class in mesh objects. There are a number of commands available for each to allow data extraction (see Commands section for details on usage):

averagemeshrect, **dp_getexactprofile**, **getvalue**, **saveovf2**, **shape_get**.
As an example, the following saves the effective field to an ovf2 file:

```
FM = ns.Ferromagnet(rectangle, cellsize)

FM.quant.Heff.saveovf2(filename)
```

All mesh display quantities listed above may be accessed in this way, when available for the given mesh type.

Commands

2dmulticonvolution *status*
Switch to multi-layered convolution and force it to 2D layering in each mesh (2), or 2D convolution for each mesh (1), or allow 3D (0).

addafmesh *name rectangle*
Add antiferromagnetic mesh with given name and rectangle (m). The rectangle can be specified as: sx sy sz ex ey ez for the start and end points in Cartesian coordinates, or as: ex ey ez with the start point as the origin.

addameshcubic *name rectangle*
Add an atomistic mesh with simple cubic structure, with given name and rectangle (m). The rectangle can be specified as: sx sy sz ex ey ez for the start and end points in Cartesian coordinates, or as: ex ey ez with the start point as the origin.

addconductor *name rectangle*
Add a normal metal mesh with given name and rectangle (m). The rectangle can be specified as: sx sy sz ex ey ez for the start and end points in Cartesian coordinates, or as: ex ey ez with the start point as the origin.

adddata *(meshname) dataname (rectangle)*
Add dataname to list of output data. If applicable specify meshname and rectangle (m) in mesh. If not specified and required, focused mesh is used with entire mesh rectangle.

adddipole *name rectangle*
Add a rectangular dipole with given name and rectangle (m). The rectangle can be specified as: sx sy sz ex ey ez for the start and end points in Cartesian coordinates, or as: ex ey ez with the start point as the origin.

addelectrode *electrode_rect*
Add an electrode in given rectangle (m).

addinsulator *name rectangle*
Add an insulator mesh with given name and rectangle (m). The rectangle can be specified as: sx sy sz ex ey ez for the start and end points in Cartesian coordinates, or as: ex ey ez with the start point as the origin.

addmaterial *name rectangle*
Add a new mesh with material parameters loaded from the materials database. The name is the material name as found in the mdb file (see materialsdatabase command); this also determines the type of mesh to create, as well as the created mesh name. The rectangle (m) can be specified as: sx sy sz ex ey ez for the start and end points in Cartesian coordinates, or as: ex ey ez with the start point as the origin.

Script return values: meshname - return name of mesh just added (can differ from the material name).

addmdbentry *meshname (materialname)*

Add new entry in the local materials database from parameters in the given mesh. The name of the new entry is set to materialname if specified, else set to meshname. For a complete entry you should then edit the mdb file manually with all the appropriate fields shown there.

addmesh *name rectangle*

Add a ferromagnetic mesh with given name and rectangle (m). The rectangle can be specified as: sx sy sz ex ey ez for the start and end points in Cartesian coordinates, or as: ex ey ez with the start point as the origin.

addmodule *(meshname) handle*

Add module with given handle to named mesh. If mesh name not specified, this is set for all applicable meshes.

addpinneddata *(meshname) dataname (rectangle))*

Add new entry in data box (at the end) with given dataname and meshname if applicable. A rectangle may also be specified if applicable, however this will not be shown in the data box.

addrect *(meshname) rectangle*

Fill rectangle (m) within given mesh (focused mesh if not specified). The rectangle coordinates are relative to mesh.

addstage *(meshname) stagetype*

Add a generic stage type to the simulation schedule with name stagetype, specifying a meshname if needed (if not specified and required, focused mesh is used).

adjustcamdistance *dZ*

Adjust camera distance from origin using dZ. This is the same as right mouse hold and up-down drag on mesh viewer.

ambient *(meshname) ambient_temperature*

Set mesh ambient temperature (all meshes if meshname not given) for Robin boundary conditions : flux normal = alpha * (T_boundary - T_ambient).

Script return values: ambient_temperature - ambient temperature for named mesh.

astepctrl *err_fail dT_incr dT_min dT_max*

Set parameters for adaptive time step control: err_fail - repeat step above this (the tolerance), dT_incr - limit multiplicative increase in dT using this, dT_min, dT_max - dT bounds.

Script return values: err_fail dT_incr dT_min dT_max

atomicmoment *(meshname) ub_multiple*

Set atomic moment as a multiple of Bohr magnetons for given mesh (all ferromagnetic meshes if not specified). This affects the temperature dependence of 'me' (see curietemperature command). A non-zero value will result in me(T) being dependent on the applied field.

Script return values: ub_multiple - atomic moment multiple of Bohr magneton for named mesh.

averagemeshrect *(meshname (quantity)) (rectangle (dp_index))*

Calculate the average value depending on currently displayed quantities, or the named quantity if given (see output of display command for possible quantities). The rectangle is specified in relative coordinates to the named mesh (focused mesh if not specified); if not specified average the entire focused mesh. If dp_index is specified then also append value to dp array.

Script return values: value

benchtime

Show the last simulation duration time in ms, between start and stop; used for performance becnhmarking.

Script return values: value

blochpreparemovingmesh *(meshname)*

Setup the named mesh (or focused mesh) for moving Bloch domain wall simulations: 1) set movingmesh trigger, 2) set domain wall structure, 3) set dipoles left and right to remove end magnetic charges, 4) enable strayfield module.

bprint *message*

Print message in console.

buffercommand *command (params...)*

Append command with parameters to command buffer.

cellsize *(meshname) value*

Change cellsize of meshname (m). The cellsize can be specified as: hx hy hz, or as: hxyz. If meshname not given use the focused mesh.

Script return values: cellsize - return cellsize of focused mesh, or of meshname if given.

center

Center mesh view and scale to fit window size.

chdir *directory*

Change working directory.

Script return values: directory

checkupdates

Connect to boris-spintronics.uk to check if updates to program or materials database are available.

clearcommbuffer

Clear commands stored in command buffer.

clearelectrodes

Delete all currently set electrodes.

clearequationconstants

Clear all user-defined constants for text equations.

clearglobalfield

Clear a set global field (previously loaded with loadovf2field on supermesh).

clearmovingmesh

Clear moving mesh settings made by a prepare command.

clearparamstemp *(meshname) (paramname)*

Clear material parameter temperature dependence in given mesh. If meshname not given clear temperature dependences in all meshes. If paramname not given clear all parameters temperature dependences.

clearparamsvar *(meshname) (paramname)*

Clear parameter spatial dependence in given mesh. If meshname not given clear spatial dependence in all meshes. If paramname not given clear all parameters spatial dependence.

clearroughness *(meshname)*

Clear roughness in given mesh (focused mesh if not specified) by setting the fine shape same as the coarse M shape.

clearscreen

Clear all console text.

clearstrainequations

Clear strain equations from all meshes.

computefields

Run simulation from current state for a single iteration without advancing the simulation time.

copymeshdata *meshname_from meshname_to (...)*

Copy all primary mesh data (e.g. magnetization values and shape) from first mesh to all other meshes given - all meshes must be of same type.

copyparams *meshname_from meshname_to (...)*

Copy all mesh parameters from first mesh to all other meshes given - all meshes must be of same type.

coupletodipoles *status*

Set/unset coupling to dipoles : if magnetic meshes touch a dipole mesh then interface magnetic cells are coupled to the dipole magnetization direction.

crosstie *(meshname) (direction (radius thickness centre))*

Set magnetization in a crosstie state with given direction (-1 or +1) starting at given centre, over given radius and thickness. If not specified then entire mesh is used, centred. If mesh name not specified, the focused mesh is used.

cuda *status*

Switch CUDA GPU computations on/off.

Script return values: status

curietemperature *(meshname) curie_temperature*

Set Curie temperature for ferromagnetic mesh (all ferromagnetic meshes if not specified). This will set default temperature dependencies as: Ms = Ms0*me, A = Ah*me^2, D = D0*me^2, K = K0*me^3 (K1 and K2), damping = damping0*(1-T/3Tc) T < Tc, damping = damping0*2T/3Tc T >= Tc, susrel = dme/d(mu0Hext). Setting the Curie temperature to zero will disable temperature dependence for these parameters.

Script return values: curie_temperature - Curie temperature for named mesh.

data

Shows list of currently set output data and available data.

Script return values: number of set output data fields

dataprecision *precision*

Set number of significant figures used for saving data to file as well as data display. The default (and minimum) is 6.

Script return values: precision

default

Reset program to default state.

deldata *index*

Delete data from list of output data at index number. If index number is -1 then delete all data fields, leaving just a default time data field - there must always be at least 1 output data field.

delelectrode *index*

Delete electrode with given index.

delequationconstant *name*

Delete named user constant used in text equations.

delmdbentry *materialname*

Delete entry in the local materials database (see materialsdatabase for current selection).

delmesh *meshname*

Delete mesh with given name.

delmodule *(meshname) handle*

Delete module with given handle from named mesh (focused mesh if not specified).

delpinneddata *index*

Delete entry in data box at given index (index in order of appearance in data box from 0 up).

delrect *((meshname) (rectangle))*

Void rectangle (m) within given mesh (focused mesh if not specified). The rectangle coordinates are relative to mesh. If rectangle not given void entire mesh.

delstage *index*

Delete stage from simulation schedule at index number. If index number is -1 then delete all stages, leaving just a default Relax stage - there must always be at least 1 stage set.

delsurfacefix *index*

Delete a fixed surface with given index (previously added with surfacefix). Use index -1 to delete all fixed surfaces.

delsurfacestress *index*

Delete an external stress surface with given index (previously added with surfacestress). Use index -1 to delete all external stress surfaces.

designateground *electrode_index*

Change ground designation for electrode with given index.

dipolevelocity *meshname velocity (clipping)*

Set velocity for dipole shifting algorithm (x, y, z, components). Optionally specify a clipping distance (x, y, z components) - i.e. dipole shift clipped. Default is 0.5 nm; to disable clipping set to zero.

Script return values: velocity clipping

disabletransportsolver *status*

Disable iteration of transport solver so any set current density remains constant.

Script return values: status

diskbufferlines *lines*

Set output disk buffer number of lines - default is 100. You shouldn't need to adjust this.

Script return values: bufferlines

display *(meshname) name*

Change quantity to display for given mesh (focused mesh if not specified).

displaybackground *(meshname) name*

Change background quantity to display for given mesh (focused mesh if not specified).

displaydetail *size*

Change displayed detail level to given displayed cell size (m).

Script return values: size

displaymodule *(meshname) modulename*

Select module for effective field and energy density display (focused mesh if not specified). If modulename is none then display total effective field.

displayrenderthresholds *thresh1 thresh2 thresh3*

Set display render thresholds of number of displayed cells for faster rendering as: 1) switch to using simpler elements (vectors only), 2) don't display surrounded cells, 3) display on checkerboard pattern even if not surrounded (vectors only). Set to zero to disable.

Script return values: thresh1 thresh2 thresh3

displaythresholds *minimum maximum*

Set thresholds for foreground mesh display : magnitude values outside this range are not rendered. If both set to 0 then thresholds are ignored.

displaythresholdtrigger *trigtype*

For vector quantities, set component to trigger thresholds on. trigtype = 1 (x component), trigtype = 2 (y component), trigtype = 3 (z component), trigtype = 5 (magnitude only)

displaytransparency *foreground background*

Set alpha transparency for display. Values range from 0 (fully transparent) to 1 (opaque). This is applicable in dual display mode when we have a background and foreground for the same mesh.

dmcellsize *(meshname) value*

Change demagnetizing field macrocell size of meshname, for atomistic meshes (m). The cellsize can be specified as: hx hy hz, or as: hxyz. If meshname not given use the focused mesh.

Script return values: cellsize - return demagnetizing field macrocell size of focused mesh, or of meshname if given.

dp_add *dp_source value (dp_dest)*

Add value to dp array and place it in destination (or at same position if destination not specified).

dp_adddp *dp_x1 dp_x2 dp_dest*

Add dp arrays : dp_dest = dp_x1 + dp_x2

dp_anghistogram *(meshname) dp_x dp_y (cx cy cz (nx ny nz (numbins min max)))*

Calculate an angular deviation histogram with given number of bins, minimum and maximum bin values, from the magnetization of the given mesh (must be magnetic; focused mesh if not specified). Save histogram in dp arrays at dp_x, dp_y. If cx cy cz values given, then first average in macrocells containing (cx, cy, cz) individual cells. If unit vector direction nx ny nz not given, then angular deviation is calculated from the average magnetization direction. If histogram parameters not given use 100 bins with minimum and maximum magnetization magnitude values.

dp_append *dp_original dp_new*

Append data from dp_new to the end of dp_original.

dp_calcsot *hm_mesh fm_mesh*

For the given heavy metal and ferromagnetic meshes calculate the expected effective spin Hall angle and field-like torque coefficient according to analytical equations (see manual).

Script return values: SHAeff, flST.

dp_calctopochargedensity *(meshname)*

Calculate topological charge density spatial dependence for the given mesh (must be magnetic; focused mesh if not specified). Output available in Cust_S.

dp_cartesiantopolar *dp_in_x dp_in_y (dp_out_r dp_out_theta)*

Convert from Cartesian coordinates (x,y) to polar (r, theta).

dp_chunkedstd *dp_index chunk*

Obtain stadard deviation from every chunk number of elements in dp array, then average all the chunk std value to obtain final std. In the special case where chunk is the number of elements in the dp array (set chunk = 0 for this case), the chunked std is the same as the usual std on the mean.

Script return values: std.

dp_clear *indexes...*

Clear dp arrays with specified indexes.

dp_clearall

Clear all dp arrays.

dp_coercivity *dp_index_x dp_index_y*

Obtain coercivity from x-y data: find first crossings of x axis in the two possible directions, with uncertainty obtained from step size.

Script return values: Hc_up Hc_up_err- Hc_up_err+ Hc_dn Hc_dn_err- Hc_dn_err+.

dp_completehysteresis *dp_index_x dp_index_y*

For a hysteresis loop with only one branch continue it by constructing the other direction branch (invert both x and y data and add it in continuation) - use only with hysteresis loops which are expected to be symmetric.

dp_countskyrmions *(meshname) (x y radius)*

Calculate the number of skyrmions for the given mesh (must be magnetic; focused mesh if not specified), optionally in the given circle with radius and centered at x y (relative values). Use Qmag = Integral(|m.(dm/dx x dm/dy)| dxdy) / 4PI.

Script return values: Q - the calculated topological charge.

dp_crossingsfrequency *dp_in_x dp_in_y dp_level dp_freq_up dp_freq_dn (steps)*

From input x-y data build a histogram of average frequency the x-y data crosses a given line (up and down, separated). The line varies between minimum and maximum of y data in given number of steps (100 by default). Output the line values in dp_level with corresponding crossings frequencies in dp_freq_up and dp_freq_dn.

dp_crossingshistogram *dp_in_x dp_in_y dp_level dp_counts (steps)*

From input x-y data build a histogram of number of times x-y data crosses a given line (up or down). The line varies between minimum and maximum of y data in given number of steps (100 by default). Output the line values in dp_level with corresponding number of crossings in dp_counts.

dp_div *dp_source value (dp_dest)*

Divide dp array by value and place it in destination (or at same position if destination not specified).

dp_divdp *dp_x1 dp_x2 dp_dest*

Divide dp arrays : dp_dest = dp_x1 / dp_x2

dp_dotprod *dp_vector ux uy uz dp_out*

Take dot product of (ux, uy, uz) with vectors in dp arrays dp_vector, dp_vector + 1, dp_vector + 2 and place result in dp_out.

dp_dotproddp *dp_x1 dp_x2*

Take dot product of dp arrays : value = dp_x1.dp_x2

dp_dumptdep *meshname paramname max_temperature dp_index*

Get temperature dependence of named parameter from named mesh up to max_temperature, at dp_index - temperature scaling values obtained.

dp_erase *dp_index start_index length*

From dp_index array erase a number of points - length - starting at start_index.

dp_extract *dp_in dp_out start_index (length)*

From dp_in array extract a number of points - length - starting at start_index, and place them in dp_out.

dp_fitadiabatic *(meshname) (abs_err Rsq T_ratio (stencil))*
Fit the computed self-consistent spin torque (see below) using Zhang-Li STT with fitting parameters P and beta (non-adiabaticity) using a given square in-plane stencil (default size 3) in order to extract the spatial variation of P. Cut-off values for absolute fitting error (default 0.1), Rsq measure (default 0.9), and normalized torque magnitude (default 0.1) can be set - value of zero disables cutoff. The focused mesh must be ferromagnetic, have the transport module set with spin solver enabled, and we also require Jc and either Ts or Tsi to have been computed. The fitting is done on Ts, Tsi, or on their sum depending if they've been enabled or not. Output available in Cust_S.

dp_fitdw *dp_x dp_y*
Fit domain wall tanh component to obtain width and center position : M(x) = A * tanh(PI * (x - x0) / D).

Script return values: D, x0, A, std_D, std_x0, std_A.

dp_fitlorentz *dp_x dp_y*
Fit Lorentz peak function to x y data : f(x) = y0 + S dH / (4(x-H0)^2 + dH^2).

Script return values: S, H0, dH, y0, std_S, std_H0, std_dH, std_y0.

dp_fitlorentz2 *dp_x dp_y*
Fit Lorentz peak function with both symmetric and asymmetric parts to x y data : f(x) = y0 + S (dH + A * (x - H0)) / (4(x-H0)^2 + dH^2).

Script return values: S, A, H0, dH, y0, std_S, std_A, std_H0, std_dH, std_y0.

dp_fitnonadiabatic *(meshname) (abs_err Rsq T_ratio (stencil))*
Fit the computed self-consistent spin torque (see below) using Zhang-Li STT with fitting parameters P and beta (non-adiabaticity) using a given square in-plane stencil (default size 3) in order to extract the spatial variation of beta. Cut-off values for absolute fitting error (default 0.1), Rsq measure (default 0.9), and normalized torque magnitude (default 0.1) can be set - value of zero disables cutoff. The focused mesh must be ferromagnetic, have the transport module set with spin solver enabled, and we also require Jc and either Ts or Tsi to have been computed. The fitting is done on Ts, Tsi, or on their sum depending if they've been enabled or not. Output available in Cust_S.

dp_fitskyrmion *dp_x dp_y*
Fit skyrmion z component to obtain radius and center position : Mz(x) = Ms * cos(2*arctan(sinh(R/w)/sinh((x-x0)/w))).

Script return values: R, x0, Ms, w, std_R, std_x0, std_Ms, std_w.

dp_fitsot *(meshname) (hm_mesh) (rectangle)*

Fit the computed self-consistent interfacial spin torque using SOT with fitting parameters SHAeff and flST (field-like torque coefficient). hm_mesh specifies the heavy metal mesh from which to obtain the current density - if not specified, then the current density in meshname is used instead. The fitting is done inside the specified rectangle for the given mesh (focused mesh if not specified), with the rectangle specified using relative coordinates as sx sy sz ex ey ez (entire mesh if not specified). This mesh must be ferromagnetic, have the transport module set with spin solver enabled, and we also require Jc and Tsi to have been computed.

Script return values: SHAeff, flST, std_SHAeff, std_flST, Rsq.

dp_fitsotstt *(meshname) (hm_mesh) (rectangle)*

Fit the computed self-consistent spin torque (see below) using Zhang-Li STT with fitting parameters P and beta (non-adiabaticity), and simultaneously also using SOT with fitting parameters SHAeff and flST (field-like torque coefficient). hm_mesh specifies the heavy metal mesh from which to obtain the current density for SOT. The fitting is done inside the specified rectangle for the given mesh (focused mesh if not specified), with the rectangle specified using relative coordinates as sx sy sz ex ey ez (entire mesh if not specified). The focused mesh must be ferromagnetic, have the transport module set with spin solver enabled, and we also require Jc and either Ts or Tsi to have been computed to have been computed. The fitting is done on Ts, Tsi, or on their sum depending if they’ve been enabled or not.

Script return values: SHAeff, flST, P, beta, std_SHAeff, std_flST, std_P, std_beta, Rsq.

dp_fitstt *(meshname) (rectangle)*

Fit the computed self-consistent spin torque (see below) using Zhang-Li STT with fitting parameters P and beta (non-adiabaticity). The fitting is done inside the specified rectangle for the given mesh (focused mesh if not specified), with the rectangle specified using relative coordinates as sx sy sz ex ey ez (entire mesh if not specified). The focused mesh must be ferromagnetic, have the transport module set with spin solver enabled, and we also require Jc and either Ts or Tsi to have been computed. The fitting is done on Ts, Tsi, or on their sum depending if they’ve been enabled or not.

Script return values: P, beta, std_P, std_beta, Rsq.

dp_get *dp_arr index*

Show value in dp_arr at given index - the index must be within the dp_arr size.

Script return values: value

dp_getampli *dp_source pointsPeriod*

Obtain maximum amplitude obtained every pointsPeriod points.

Script return values: amplitude.

dp_getaveragedprofile *(meshname) start end step dp_index*

This is used in conjuction with dp_getexactprofile. Get in dp arays starting at dp_index the averaged profile built with dp_getexactprofile. start, end and step are the same parameters used by dp_getexactprofile. Calling this command causes the average counter to be reset to zero. Thus a sequence consists of 1) dp_getexactprofile executed as many times as needed (e.g. to average stochasticity), called with dp_index = -1, 2) dp_getaveragedprofile to read out the profile and reset averaging counter, ready for next batch of dp_getexactprofile calls.

Script return values: number of averages

dp_getexactprofile *(meshname (quantity)) start end step dp_index (stencil)*

Extract profile of physical quantity displayed on screen, or the named quantity if given (see output of display command for possible quantities), for named mesh (focused mesh if not specified), directly from the mesh (so using the exact mesh resolution not the displayed resolution), along the line specified with given start and end relative cartesian coordinates (m) and with the given step size (m). If stencil specified - as x y z (m) - then obtain profile values using weighted averaging with stencil centered on profile point. If dp_index >= 0 then place profile in given dp arrays: up to 4 consecutive dp arrays are used, first for distance along line, the next 3 for physical quantity components (e.g. Mx, My, Mz) so allow space for these starting at dp_index. If dp_index = -1 then just average profile in internal memory, to be read out with dp_getaveragedprofile. Profile will wrap-around if exceeding mesh size.

dp_getprofile *start end dp_index*

Extract profile of physical quantities displayed on screen, at the current display resolution, along the line specified with given start and end cartesian absolute coordinates (m). Place profile in given dp arrays: up to 4 consecutive dp arrays are used, first for distance along line, the next 3 for physical quantity components (e.g. Mx, My, Mz) so allow space for these starting at dp_index. NOTE : if you only want to extract the profile from a single mesh, then use dp_getexactprofile instead; this command should only be used to extract profiles across multiple meshes.

dp_histogram *(meshname) dp_x dp_y (cx cy cz (numbins min max))*

Calculate a histogram with given number of bins, minimum and maximum bin values, from the magnetization magnitude of the given mesh (must be magnetic; focused mesh if not specified). Save histogram in dp arrays at dp_x, dp_y.If cx cy cz values given, then first average in macrocells containing (cx, cy, cz) individual cells. If histogram parameters not given use 100 bins with minimum and maximum magnetization magnitude values.

dp_histogram2 *(meshname) dp_x dp_y (numbins min max M2 deltaM2)*

Calculate a histogram for the given 2-sublattice mesh (focused mesh if not specified) with given number of bins, minimum and maximum bin values for sub-lattice A, if the corresponding magnetization magnitude in sublattice B equals M2 within the given deltaM2. Save histogram in dp arrays at dp_x, dp_y. If histogram parameters not given use 100 bins with minimum and maximum magnetization magnitude values, with M2 set to MeB and deltaM2 set 0.01*MeB respectively.

dp_linreg *dp_index_x dp_index_y (dp_index_z dp_index_out)*
Fit using linear regression to obtain gradient and intercept with their uncertainties. If dp_index_z is specified multiple linear regressions are performed on adjacent data points with same z value; output in 5 dp arrays starting at dp_index_out as: z g g_err c c_err.

Script return values: g g_err c c_err.

dp_load *(directory/)filename file_indexes... dp_indexes...*
Load data columns from filename into dp arrays. file_indexes are the column indexes in filename (.txt termination by default), dp_indexes are used for the dp arrays; count from 0. If directory not specified, the default one is used.

dp_mean *dp_index (exclusion_ratio)*
Obtain mean value with standard deviation. If exclusion_ratio (>0) included, then exclude any points which are greater than exclusion_ratio normalised distance away from the mean.

Script return values: mean stdev.

dp_minmax *dp_index*
Obtain absolute minimum and maximum values, together with their index position.

Script return values: min_value min_index max_value max_index.

dp_monotonic *dp_in_x dp_in_y dp_out_x dp_out_y*
From input x-y data extract monotonic sequence and place it in output x y arrays.

dp_mul *dp_source value (dp_dest)*
Multiply dp array with value and place it in destination (or at same position if destination not specified).

dp_muldp *dp_x1 dp_x2 dp_dest*
Multiply dp arrays : dp_dest = dp_x1 * dp_x2

dp_newfile *(directory/)filename*
Make new file, erasing any existing file with given name. If directory not specified, the default one is used.

dp_peaksfrequency *dp_in_x dp_in_y dp_level dp_freq (steps)*

From input x-y data build a histogram of average frequency of peaks in the x-y data in bands given by the number of steps. The bands vary between minimum and maximum of y data in given number of steps (100 by default). Output the line values in dp_level with corresponding peak frequencies in dp_freq.

dp_pow *dp_source exponent (dp_dest)*
Exponentiate source aray values and place it in destination (or at same position if destination not specified).

dp_rarefy *dp_in dp_out (skip)*
Pick elements from dp_in using the skip value (1 by default) and set them in dp_out; e.g. with skip = 2 every 3rd data point is picked. The default skip = 1 picks every other point.

dp_remanence *dp_index_x dp_index_y*
Obtain remanence from x-y data: find values at zero field in the two possible directions.

Script return values: Mr_up Mr_dn.

dp_removeoffset *dp_index (dp_index_out)*
Subtract the first point (the offset) from all the points in dp_index. If dp_index_out not specified then processed data overwrites dp_index.

dp_replacerepeats *dp_index (dp_index_out)*
Replace repeated points from data in dp_index using linear interpolation: if two adjacent sets of repeated points found, replace repeats between the mid-points of the sets. If dp_index_out not specified then processed data overwrites dp_index.

dp_save *(directory/)filename dp_indexes...*
Save specified dp arrays in filename (.txt termination by default). If directory not specified, the default one is used. dp_indexes are used for the dp arrays; count from 0.

dp_saveappend *(directory/)filename dp_indexes...*
Save specified dp arrays in filename (.txt termination by default) by appending at the end. If directory not specified, the default one is used. dp_indexes are used for the dp arrays; count from 0.

dp_saveappendasrow *(directory/)filename dp_index*
Save specified dp array in filename (.txt termination by default) as a single row with tab-spaced values, appending to end of file. If directory not specified, the default one is used.

dp_saveasrow *(directory/)filename dp_index*
Save specified dp array in filename (.txt termination by default) as a single row with tab-spaced values. If directory not specified, the default one is used.

dp_sequence *dp_index start_value increment points*

Generate a sequence of data points in dp_index from start_value using increment.

dp_set *dp_arr index value*

Set value in dp_arr at given index - the index must be within the dp_arr size.

dp_showsizes *(dp_arr)*

List sizes of all non-empty dp arrays, unless a specific dp_arr index is specified, in which case only show the size of dp_arr.

Script return values: dp_arr size if specified

dp_smooth *dp_in dp_out window_size*

Smooth data in dp_in using nearest-neighbor averaging with given window size, and place result in dp_out (must be different).

dp_sub *dp_source value (dp_dest)*

Subtract value from dp array and place it in destination (or at same position if destination not specified).

dp_subdp *dp_x1 dp_x2 dp_dest*

Subtract dp arrays : dp_dest = dp_x1 - dp_x2

dp_sum *dp_index*

Obtain sum of all elements in dp array.

Script return values: sum.

dp_thavanghistogram *(meshname) dp_x dp_y cx cy cz (nx ny nz (numbins min max))*

Calculate an angular deviation histogram with given number of bins, using thermal averaging in each macrocell containing (cx, cy, cz) individual cells, minimum and maximum bin values, from the magnetization of the given mesh (must be magnetic; focused mesh if not specified). Save histogram in dp arrays at dp_x, dp_y. If unit vector direction nx ny nz not given, then angular deviation is calculated from the average magnetization direction. If histogram parameters not given use 100 bins with minimum and maximum magnetization magnitude values.

dp_thavhistogram *(meshname) dp_x dp_y cx cy cz (numbins min max)*

Calculate a histogram with given number of bins, using thermal averaging in each macrocell containing (cx, cy, cz) individual cells, minimum and maximum bin values, from the magnetization magnitude of the focused mesh (must be magnetic). Save histogram in dp arrays at dp_x, dp_y. If histogram parameters not given use 100 bins with minimum and maximum magnetization magnitude values.

dp_topocharge *(meshname) (x y radius)*

Calculate the topological charge for the given mesh (must be magnetic; focused mesh if not specified), optionally in the given circle with radius and centered at x y (relative values). Q = Integral(m.(dm/dx x dm/dy) dxdy) / 4PI.

Script return values: Q - the calculated topological charge.

dwall *(meshname) longitudinal transverse width position*

Create an idealised domain wall (tanh profile for longitudinal component, 1/cosh profile for transverse component) along the x-axis direction in the given mesh (focused mesh if not specified). For longitudinal and transverse specify the components of magnetization as x, -x, y, -y, z, -z, i.e. specify using these std::string literals. For width and position use metric units.

dwposcomponent *value*

Set vector component used to extract dwpos_x, dwpos_y, dwpos_z data: this should be the component with a tanh profile. -1: automatic (detect tanh component), 0: x, 1: y, 2: z. Default is automatic, but you might want to specify component as the automatic detection can fail when very noisy (e.g. close to Tc for atomistic simulations).

Script return values: value

ecellsize *(meshname) value*

Change cellsize of meshname for electrical conduction (m). The cellsize can be specified as: hx hy hz, or as: hxyz. If meshname not given use the focused mesh.

Script return values: cellsize - return electrical conduction cellsize of focused mesh, or of meshname if given.

editdata *index (meshname) dataname (rectangle)*

Edit entry in list of output data at given index in list. If applicable specify meshname and rectangle (m) in mesh. If not specified and required, focused mesh is used with entire mesh rectangle.

editdatasave *index savetype (savevalue)*

Edit data saving condition in simulation schedule. Use index < 0 to set condition for all stages.

editstage *index (meshname) stagetype*

Edit stage type from simulation schedule at index number.

editstagestop *index stoptype (stopvalue)*

Edit stage/step stopping condition in simulation schedule. Use index < 0 to set condition for all stages.

editstagevalue *index value*

Edit stage setting value in simulation schedule. The value type depends on the stage type.

editsurfacefix *index rect*

Edit rectangle of fixed surface with given index.

editsurfacestress *index rect*

Edit rectangle of stress surface with given index.

editsurfacestressequation *index equation*

Edit equation of stress surface with given index.

electrodes

Show currently configured electrodes.

equationconstants *name value*

Create or edit user constant to be used in text equations.

errorlog *status*

Set error log status.

escellsize *value*

Change cellsize for electric super-mesh (m). The cellsize can be specified as: hx hy hz, or as: hxyz

Script return values: cellsize - return cellsize for electric super-mesh.

evalspeedup *level*

Enable/disable evaluation speedup by extrapolating demag field at evaluation substeps from previous field updates. Status levels: 0 (no speedup), 1 (step), 2 (linear), 3 (quadratic), 4 (cubic), 5 (quartic), 6 (quintic). If enabling speedup strongly recommended to always use quadratic type.

Script return values: level

exchangecoupledmeshes *(meshname) status*

Set/unset direct exchange coupling to neighboring meshes : if neighboring ferromagnetic meshes touch the named mesh (focused mesh if not specified) then interface magnetic cells are direct exchange coupled to them.

Script return values: status

excludemulticonvdemag *(meshname) status*

Set exclusion status (0 or 1) of named mesh from multi-layered demag convolution (focused mesh if not specified).

flower *(meshname) (direction (radius thickness centre))*

Set magnetization in a flower state with given direction (-1 or +1) starting at given centre, over given radius and thickness. If not specified then entire mesh is used, centred. If mesh name not specified, the focused mesh is used.

flusherrorlog

Clear error log.

fmscellsize *value*

Change cellsize for ferromagnetic super-mesh (m). The cellsize can be specified as: hx hy hz, or as: hxyz

Script return values: cellsize - return cellsize for ferromagnetic super-mesh.

generate2dgrains *(meshname) spacing (seed)*

Generate 2D Voronoi cells in the xy plane at given average spacing in given mesh (focused mesh if not specified). The seed is used for the pseudo-random number generator, 1 by default.

generate3dgrains *(meshname) spacing (seed)*

Generate 3D Voronoi cells at given average spacing in given mesh (focused mesh if not specified). The seed is used for the pseudo-random number generator, 1 by default.

getmeshtype *meshname*

Show mesh type of named mesh.

Script return values: meshtype.

getvalue *(meshname (quantity)) position*

Read value at position depending on currently displayed quantites, or the named quantity if given (see output of display command for possible quantities), in named mesh (focused mesh if not specified).

Script return values: value

gpukernels *status*

When in CUDA mode calculate demagnetization kernels initialization on the GPU (1) or on the CPU (0).

Script return values: status

imagecropping *left bottom right top*

Set cropping of saved mesh images using normalized left, bottom, right, top values: 0, 0 point is left, bottom of mesh window and 1, 1 is right, top of mesh window.

individualshape *status*

When changing the shape inside a mesh set this flag to true so the shape is applied only to the primary displayed physical quantity. If set to false then all relevant physical quantities are shaped.

Script return values: status

insulatingside *(meshname) side_literal status*

Set temperature insulation (Neumann boundary condition) for named mesh side (focused mesh if not specified). side_literal : x, -x, y, -y, z, -z.

Script return values: status_x status_-x status_y status_-y status_z status_-z - insulating sides status for named mesh.

invertmag *(meshname) (components)*

Invert magnetization direction. If mesh name not specified, the focused mesh is used. You can choose to invert just one or two components instead of the entire vector: specify components as x y z, e.g. invertmag x

isrunning

Checks if the simulation is running and sends state value to the calling script.

Script return values: state - return simulation running state.

iterupdate *iterations*

Update mesh display every given number of iterations during a simulation.

Script return values: iterations - return number of iterations for display update.

linkdtelastic *flag*

Links elastodynamics solver time-step to magnetic ODE time-step if set, else elastodynamics time-step is independently controlled.

linkdtspeedup *flag*

Links speedup time-step to ODE time-step if set, else speedup time-step is independently controlled.

linkdtstochastic *flag*

Links stochastic time-step to ODE time-step if set, else stochastic time-step is independently controlled.

linkstochastic *(meshname) flag*

Links stochastic cellsize to magnetic cellsize if flag set to 1 for given mesh, else stochastic cellsize is independently controlled. If meshname not given set for all meshes.

loadmaskfile *(meshname) (z_depth) (directory/)filename*

Apply .png mask file to magnetization in given mesh (focused mesh if not specified); i.e. transfer shape from .png file to mesh - white means empty cells. If image is in grayscale then void cells up to given depth top down (z_depth > 0) or down up (z_depth < 0). If z-depth = 0 then void top down up to all z cells.

loadovf2curr *(meshname) (directory/)filename*

Load an OOMMF-style OVF 2.0 file containing current density data, into the given mesh (which must have transport module enabled; focused mesh if not specified), mapping the data to the current mesh dimensions.

loadovf2disp *(meshname) (directory/)filename*

Load an OOMMF-style OVF 2.0 file containing mechanical displacement data, into the given mesh (which must be ferromagnetic and have the melastic module enabled; focused mesh if not specified), mapping the data to the current mesh dimensions. From the mechanical displacement the strain tensor is calculated.

loadovf2field *(meshname) (directory/)filename*

Load an OOMMF-style OVF 2.0 file containing applied field data, into the given mesh (which must be magnetic; focused mesh if not specified), mapping the data to the current mesh dimensions. Alternatively if meshname is supermesh, then load a global field with absolute rectangle coordinates and its own discretization.

loadovf2mag *(meshname) (renormalize_value) (directory/)filename*

Load an OOMMF-style OVF 2.0 file containing magnetization data, into the given mesh (which must be magnetic; focused mesh if not specified), mapping the data to the current mesh dimensions. By default the loaded data will not be renormalized: renormalize_value = 0. If a value is specified for renormalize_value, the loaded data will be renormalized to it (e.g. this would be an Ms value).

loadovf2strain *(meshname) (directory/)filename_diag filename_odiag*

Load an OOMMF-style OVF 2.0 file containing strain tensor data, into the given mesh (which must be ferromagnetic and have the melastic module enabled; focused mesh if not specified), mapping the data to the current mesh dimensions. The symmetric strain tensor is applicable for a cubic crystal, and has 3 diagonal component (specified in filename_diag with vector data as xx, yy, zz), and 3 off-diagonal components (specified in filename_odiag with vector data as yz, xz, xy).

loadovf2temp *(meshname) (directory/)filename*

Load an OOMMF-style OVF 2.0 file containing temperature data, into the given mesh (which must have heat module enabled; focused mesh if not specified), mapping the data to the current mesh dimensions.

loadsim *(directory/)filename*

Load simulation with given name.

makevideo *(directory/)filebase fps quality*

Make a video from .png files sharing the common filebase name. Make video at given fps and quality (0 to 5 worst to best).

manual

Opens Boris manual for current version.

matcurietemperature *(meshname) curie_temperature*

Set indicative material Curie temperature for ferromagnetic mesh (focused mesh if not specified). This is not used in calculations, but serves as an indicative value - set the actual Tc value with the curietemperature command.

Script return values: curie_temperature - Indicative material Curie temperature for named mesh.

materialsdatabase *(mdbname)*

Switch materials database in use. This setting is not saved by savesim, so using loadsim doesn't affect this setting; default mdb set on program start.

mccomputefields *status*

Enable/disable modules update during Monte Carlo stages. Disabled by default, but if you want to save energy density terms you need to enabled this - equivalent to running ComputeFields after every Monte Carlo step.

Script return values: status

mcconeangle *min_angle max_angle*

Set Monte Carlo cone angle limits (1 to 180 degrees). Setting min and max values the same results in a fixed cone angle Monte Carlo algorithm.

Script return values: min_angle max_angle

mcconstrain *(meshname) value*

Set value 0 to revert to classic Monte-Carlo Metropolis for ASD. Set a unit vector direction value (x y z) to switch to constrained Monte Carlo as described in PRB 82, 054415 (2010). If meshname not specified setting is applied to all atomistic meshes.

mcdisable *(meshname) status*

Disable or enable Monte Carlo algorithm for given mesh (focused mesh if not specified).

mcellsize *(meshname) value*

Change cellsize of meshname for mechanical solver (m). The cellsize can be specified as: hx hy hz, or as: hxyz. If meshname not given use the focused mesh.

Script return values: cellsize - return mechanical cellsize of focused mesh, or of meshname if given.

mcserial *(meshname) value*

Change Monte-Carlo algorithm type. 0: parallel (default); red-black ordering parallelization 1: serial (testing only); spins picked in random order. If meshname not specified setting is applied to all atomistic meshes.

memory

Show CPU and GPU-addressable memory information (total and free).

mesh

Display information for all meshes.

meshfocus *meshname*

Change mesh focus to given mesh name.

Script return values: meshname - return name of focused mesh.

meshfocus2 *meshname*

Change mesh focus to given mesh name but do not change camera orientation.

Script return values: meshname - return name of focused mesh.

meshrect *(meshname) rectangle*

Change rectangle of meshname (m). The rectangle can be specified as: sx sy sz ex ey ez for the start and end points in Cartesian coordinates, or as: ex ey ez with the start point as the origin. If meshname not given use the focused mesh.

Script return values: rectangle - return rectangle of focused mesh, or of meshname if given.

mirrormag *(meshname) axis*

Mirror magnetization in a given axis, specified as x, y, z, e.g. mirrormag x. If mesh name not specified, the focused mesh is used.

modules *(meshname (modules...))*

Show interactive list of available and currently set modules. If meshname specified then return all set modules names, and set given modules if specified.

Script return values: modulenames

movingmesh *status_or_meshname*

Set/unset trigger for movingmesh algorithm. If status_or_meshname = 0 then turn off, if status_or_meshname = 1 then turn on with trigger set on first ferromagnetic mesh, else status_or_meshname should specify the mesh name to use as trigger.

movingmeshasym *status*

Change symmetry type for moving mesh algorithm: 1 for antisymmetric (domain walls), 0 for symmetric (skyrmions).

Script return values: status

movingmeshthresh *value*

Set threshold used to trigger a mesh shift for moving mesh algorithm - normalised value between 0 and 1.

Script return values: threshold

multiconvolution *status*

Switch between multi-layered convolution (true) and supermesh convolution (false).

ncommon *sizes*

Switch to multi-layered convolution and force it to user-defined discretisation, specifying sizes as nx ny nz.

ncommonstatus *status*

Switch to multi-layered convolution and force it to user-defined discretisation (status = true), or default discretisation (status = false).

neelpreparemovingmesh *(meshname)*

Setup the named mesh (or focused mesh) for moving Neel domain wall simulations: 1) set movingmesh trigger, 2) set domain wall structure, 3) set dipoles left and right to remove end magnetic charges, 4) enable strayfield module.

newinstance *port (cudaDevice (password))*

Start a new local Boris instance with given server port, and optionally cuda device number (0/1/2/3/...); a value of -1 means automatically determine cuda device. If password not blank this should be specified otherwise server will not start.

nextstage

Increment stage value by one.

ode

Show interactive list of available and currently set ODEs and evaluation methods.

onion *(meshname) (direction (radius1 radius2 thickness centre))*

Set magnetization in an onion state with given direction (-1 clockwise, +1 anti-clockwise) starting at given centre, from radius1 to radius2 and thickness. If not specified then entire mesh is used, centred. If mesh name not specified, the focused mesh is used.

openpotentialresistance *resistance*

Set open potential mode for electrodes, by setting a resistance value - set zero to disable open potential mode. In this mode the electrode potential is determined during the simulation, e.g. if thermoelectric effect is enabled, then setting electrodes in open potential mode allows a thermoelectric current to flow through the electrodes, with potential determined as thermoelectric current generated times resistance set.

Script return values: resistance

params *(meshname)*

List all material parameters. If meshname not given use the focused mesh.

paramstemp *(meshname)*

List all material parameters temperature dependence. If meshname not given use the focused mesh.

paramsvar *(meshname)*

List all material parameters spatial variation. If meshname not given use the focused mesh.

pbc *(meshname) flag images*

Set periodic boundary conditions for magnetization in given mesh (must be magnetic, or supermesh; supermesh/all meshes if name not given). Flags specify types of perodic boundary conditions: x, y, or z; images specify the number of mesh images to use either side for the given direction when calculating the demagnetising kernel - a value of zero disables pbc. e.g. pbc x 10 sets x periodic boundary conditions with 10 images either side for all meshes; pbc x 0 clears pbc for the x axis.

preparemovingmesh *(meshname)*

Setup the named mesh (or focused mesh) for moving transverse (or vortex) domain wall simulations: 1) set movingmesh trigger, 2) set domain wall structure, 3) set dipoles left and right to remove end magnetic charges, 4) enable strayfield module.

prngseed *(meshname) seed*
Set PRNG seed, for computations with stochasticity, in given mesh (all meshes if name not given). If seed = 0 then system tick count is used as seed (default behavior), otherwise the fixed set value is used. NOTE: seed values > 0 only take effect for computations with cuda 1.

Script return values: seed

raapbiasequation *meshname text_equation*
Set equation for bias dependence of TMR anti-parallel RA value in given mesh (must be insulator with tmr module added) - the text equation uses V for the bias value. Call with empty equation string to clear any currently set equation.

random *(meshname (seed))*
Set random magnetization distribution in mesh, with pseud-random number generator seed (default 1 if not specified). If mesh name not specified, the focused mesh is used.

randomxy *(meshname (seed))*
Set random magnetization distribution in the xy plane in mesh, with pseud-random number generator seed (default 1 if not specified). If mesh name not specified, the focused mesh is used.

rapbiasequation *meshname text_equation*
Set equation for bias dependence of TMR parallel RA value in given mesh (must be insulator with tmr module added) - the text equation variables are RAp (parallel TMR RA), RAap (antiparallel TMR RA), and V (bias across insulator mesh). Call with empty equation string to clear any currently set equation.

refineroughness *(meshname) value*
Set roughness refinement cellsize divider in given mesh (focused mesh if not specified), i.e. cellsize used for roughness initialization is the magnetic cellsize divided by value (3 components, so divide component by component).

Script return values: value - roughness refinement.

refreshmdb
Reload the local materials database (see materialsdatabase for current selection). This is useful if you modify the values in the materials database file externally.

refreshscreen
Refreshes entire screen.

renamemesh *(old_name) new_name*

Rename mesh. If old_name not specified then the focused mesh is renamed.

requestmdbsync *materialname (email)*

Request the given entry in the local materials database is added to the online shared materials database. This must be a completed entry - see manual for instructions. The entry will be checked before being made available to all users through the online materials database. If you want to receive an update about the status of this request include an email address.

reset

Reset simulation state to the starting state.

resetelsolver

Reset the elastodynamics solver to unstrained state.

resetmesh *(meshname)*

Reset to constant magnetization in given mesh (focused mesh if not specified).

robinalpha *(meshname) robin_alpha*

Set alpha coefficient (all meshes if meshname not given) for Robin boundary conditions : flux normal = alpha * (T_boundary - T_ambient).

Script return values: robin_alpha - Robin alpha value for named mesh.

rotcamaboutaxis *dAngle*

Rotate camera about its axis using an angle change. This is the same as right mouse hold and left-right drag on mesh viewer.

rotcamaboutorigin *dAzim dPolar*

Rotate camera about origin using a delta azimuthal and delta polar angle change. This is the same as middle mouse hold and drag on mesh viewer.

roughenmesh *(meshname) depth (side (seed))*

Roughen given mesh (focused mesh if not specified) to given depth (m) on named side (use side = x, y, z, -x, -y, -z as literal, z by default). The seed is used for the pseudo-random number generator, 1 by default.

run

Run simulation from current state.

runcommbuffer

Execute commands stored in command buffer.

runscript *(directory/)filename*

Execute a Python-scripted simulation (filename must be a Python script). If directory not specified then default directory is used.

runscriptnewinstance *(directory/)filename*

Execute a Python-scripted simulation (filename must be a Python script) in a new program instance. If directory not specified then default directory is used.

runstage *stage*

Run given simulation stage only.

savecomment *(directory/)filename comment*

Save comment in given file by appending to it.

savedata *(append_option)*

Save data (as currently configured in output data - see setdata/adddata commands) to output file (see savedatafile command). append_option = -1 : use default append behaviour (i.e. new file with header created for first save, then append). append_option = 0 : make new file then save. append_option = 1 : append to file.

savedatafile *(directory/)filename*

Change output data file (and working directory if specified).

Script return values: filename

savedataflag *status*

Set data saving flag status.

Script return values: status

saveimage *((directory/)filename)*

Save currently displayed mesh image, with a centered view of displayed quantity only, to given file (as .png). If directory not specified then default directory is used. If filename not specified then default image save file name is used.

saveimagefile *(directory/)filename*

Change image file base (and working directory if specified).

Script return values: filename

saveimageflag *status*

Set image saving flag status.

Script return values: status

savemeshimage *((directory/)filename)*

Save currently displayed mesh image to given file (as .png). If directory not specified then default directory is used. If filename not specified then default image save file name is used.

saveovf2 *(meshname (quantity)) (data_type) (directory/)filename*

Save an OOMMF-style OVF 2.0 file containing data from the given mesh (focused mesh if not specified), depending on currently displayed quantites, or the named quantity if given (see output of display command for possible quantities). You can specify the data type as data_type = bin4 (single precision 4 bytes per float), data_type = bin8 (double precision 8 bytes per float), or data_type = text. By default bin8 is used.

saveovf2mag *(meshname) (n) (data_type) (directory/)filename*

Save an OOMMF-style OVF 2.0 file containing magnetization data from the given mesh (which must be magnetic; focused mesh if not specified). You can normalize the data to Ms0 value by specifying the n flag (e.g. saveovf2mag n filename) - by default the data is not normalized. You can specify the data type as data_type = bin4 (single precision 4 bytes per float), data_type = bin8 (double precision 8 bytes per float), or data_type = text. By default bin8 is used.

saveovf2param *(meshname) (data_type) paramname (directory/)filename*

Save an OOMMF-style OVF 2.0 file containing the named parameter spatial variation data from the given mesh (focused mesh if not specified). You can specify the data type as data_type = bin4 (single precision 4 bytes per float), data_type = bin8 (double precision 8 bytes per float), or data_type = text. By default bin8 is used.

savesim *(directory/)filename*

Save simulation with given name. If no name given, the last saved/loaded file name will be used.

scalemeshrects *status*

When changing a mesh rectangle scale and shift all other mesh rectangles in proportion if status set.

Script return values: status

scellsize *(meshname) value*

Change cellsize of meshname for stochastic properties (m). The cellsize can be specified as: hx hy hz, or as: hxyz. If meshname not given use the focused mesh.

Script return values: cellsize - return stochastic properties cellsize of focused mesh, or of meshname if given.

scriptserver *status*

Enable or disable the script communication server. When enabled the program will listen for commands received using network sockets on port 1542.

selectcudadevice *number*

Select CUDA device to use from available devices. The device CUDA Compute version must match Boris CUDA version. Devices numbered from 0 up, default selection at startup is device 0.

Script return values: number

serverpassword *password*

Set/change script server password - this is used to authenticate remote client messages. By default no password is set (blank).

serverport *port*

Set script server port.

Script return values: port

serversleepms *time_ms*

Set script server thread sleep time in ms: lower value makes server more responsive, but increases CPU load.

Script return values: time_ms

setafmesh *name rectangle*

Set a single antiferromagnetic mesh (deleting all other meshes) with given name and rectangle (m). The rectangle can be specified as: sx sy sz ex ey ez for the start and end points in Cartesian coordinates, or as: ex ey ez with the start point as the origin.

setameshcubic *name rectangle*

Set a single atomistic mesh (deleting all other meshes) with simple cubic structure, with given name and rectangle (m). The rectangle can be specified as: sx sy sz ex ey ez for the start and end points in Cartesian coordinates, or as: ex ey ez with the start point as the origin.

setangle *(meshname) polar azimuthal*

Set magnetization angle (deg.) in mesh uniformly using polar coordinates (all magnetic meshes if name not given).

setatomode *equation evaluation*

Set differential equation to solve in atomistic meshes, and method used to solve it (same method is applied to micromagnetic and atomistic meshes).

setconductor *name rectangle*

Set a single metal mesh (deleting all other meshes) with given name and rectangle (m). The rectangle can be specified as: sx sy sz ex ey ez for the start and end points in Cartesian coordinates, or as: ex ey ez with the start point as the origin.

setcurrent *current*

Set a constant current source with given value. The potential will be adjusted to keep this constant current.

Script return values: current

setcurrentdensity *(meshname) Jx Jy Jz*

Set a constant current density vector with given components in given mesh (focused mesh if not specified). Must have transport module added, but transport solver iteration will be disabled.

setdata *(meshname) dataname (rectangle)*

Delete all currently set output data and set dataname to list of output data. If applicable specify meshname and rectangle (m) in mesh. If not specified and required, focused mesh is used with entire mesh rectangle.

setdefaultelectrodes *(sides)*

Set electrodes at the x-axis ends of the given mesh, both set at 0V. If sides specified (literal x, y, or z) then set at respective axis ends instead of default x-axis ends. Set the left-side electrode as the ground. Delete all other electrodes.

setdipole *name rectangle*

Set a single dipole (deleting all other meshes) with given name and rectangle (m). The rectangle can be specified as: sx sy sz ex ey ez for the start and end points in Cartesian coordinates, or as: ex ey ez with the start point as the origin.

setdisplayedparamsvar *(meshname) paramname*

Set param to display for given mesh (focused mesh if not specified) when ParamVar display is enabled (to show spatial variation if any).

setdt *value*

Set differential equation time-step (only applicable to fixed time-step methods).

Script return values: dT

setdtspeedup *value*

Set time step for evaluation speedup.

Script return values: dTspeedup

setdtstoch *value*

Set time step for stochastic field generation.

Script return values: dTstoch

seteldt *value*

Set elastodynamics equation solver time step. Zero value disables solver.

Script return values: value - elastodynamics equation time step.

setelectrodepotential *electrode_index potential*

Set potential on electrode with given index.

Script return values: potential

setelectroderect *electrode_index electrode_rect*

Edit rectangle (m) for electrode with given index.

setfield *(meshname) magnitude polar azimuthal*

Set uniform magnetic field (A/m) using polar coordinates. If mesh name not specified, this is set for all magnetic meshes - must have Zeeman module added.

Script return values: <Ha_x, Ha_y, Ha_z> - applied field in Cartesian coordinates for, or of meshname if given.

setheatdt *value*

Set heat equation solver time step. Zero value disables solver.

Script return values: value - heat equation time step.

setinsulator *name rectangle*

Set a single insulator mesh (deleting all other meshes) with given name and rectangle (m). The rectangle can be specified as: sx sy sz ex ey ez for the start and end points in Cartesian coordinates, or as: ex ey ez with the start point as the origin.

setktens *term1 (term2 ...)*

Set the anisotropy energy terms for tensorial anisotropy. Each term must be of the form dxn1yn2zn3, where d is a number, n1, n2, n3 are integers >= 0. x, y, z are string literals.

setmaterial *name rectangle*

Set a single mesh with material parameters loaded from the materials database (deleting all other meshes). The name is the material name as found in the mdb file (see materialsdatabase command); this also determines the type of mesh to create, as well as the created mesh name. The rectangle (m) can be specified as: sx sy sz ex ey ez for the start and end points in Cartesian coordinates, or as: ex ey ez with the start point as the origin.

Script return values: meshname - return name of mesh just added (can differ from the material name).

setmesh *name rectangle*

Set a single ferromagnetic mesh (deleting all other meshes) with given name and rectangle (m). The rectangle can be specified as: sx sy sz ex ey ez for the start and end points in Cartesian coordinates, or as: ex ey ez with the start point as the origin.

setobjectangle *(meshname) polar azimuthal position*

Set magnetization angle in solid object only containing given relative position (x y z) uniformly using polar coordinates. If mesh name not specified, the focused mesh is used.

setode *equation evaluation*

Set differential equation to solve in both micromagnetic and atomistic meshes, and method used to solve it (same method is applied to micromagnetic and atomistic meshes).

setodeeval *evaluation*

Set differential equation method used to solve it (same method is applied to micromagnetic and atomistic meshes).

setparam *(meshname) paramname (value)*

Set the named parameter to given value. If meshname not given use the focused mesh.

Script return values: value - return value of named parameter in named mesh (focused mesh if not specified).

setparamtemparray *(meshname) paramname filename*

Set the named parameter temperature dependence using an array in the given mesh (all applicable meshes if not given). This must contain temperature values and scaling coefficients. Load directly from a file (tab spaced).

setparamtempequation *(meshname) paramname text_equation*
Set the named parameter temperature dependence equation for the named mesh (all applicable meshes if not given).

setparamvar *(meshname) paramname generatorname (arguments...)*
Set the named parameter spatial dependence for the named mesh (all applicable meshes if not specified) using the given generator (including any required arguments for the generator - if not given, default values are used).

setpotential *potential*
Set a symmetric potential drop : -potential/2 for ground electrode, +potential/2 on all other electrodes.

Script return values: potential

setrect *(meshname) polar azimuthal rectangle*
Set magnetization angle in given rectangle of mesh (relative coordinates) uniformly using polar coordinates. If mesh name not specified, the focused mesh is used.

setschedulestage *stage*
Set stage value, but do not run simulation. Must be a valid stage number.

Script return values: stage

setsordamping *damping_v damping_s*
Set fixed damping values for SOR algorithm used to solve the Poisson equation for V (electrical potential) and S (spin accumulation) respectively.

Script return values: damping_v damping_s

setstage *(meshname) stagetype*
Delete all currently set stages, and set a new generic stage type to the simulation schedule with name stagetype, specifying a meshname if needed (if not specified and required, focused mesh is used).

setstagevalue *index*
Set configured value for given stage index.

setstress *(meshname) magnitude polar azimuthal*
Set uniform mechanical stress (Pa) using polar coordinates. If mesh name not specified, this is set for all magnetic meshes - must have MElastic module added. Setting a uniform stress disables the elastodynamics solver.

Script return values: <Tsig_x, Tsig_y, Tsig_z> - applied mechanical stress in Cartesian coordinates for focused mesh, or of meshname if given.

shape_cone *(meshname) len_x len_y len_z cpos_x cpos_y cpos_z*
Set a conical shape in given mesh (focused mesh if not specified) with given lengths (x, y, z) at given centre position coordinates (x, y, z).

Script return values: shape object definition as name dim_x dim_y dim_z cpos_x cpos_y cpos_z rot_theta rot_phi repeat_x repeat_y repeat_z disp_x disp_y disp_z method

shape_disk *(meshname) dia_x dia_y cpos_x cpos_y (z_start z_end)*
Set a disk shape in given mesh (focused mesh if not specified) with given x and y diameters at given centre position with x and y coordinates. Optionally specify z start and end coordinates, otherwise entire mesh thickness used.

Script return values: shape object definition as name dim_x dim_y dim_z cpos_x cpos_y cpos_z

shape_displacement *x y z*
Set modifier for shape generator commands: displacements along x, y, z to use with shape repetitions (shape_repetitions).

Script return values: value

shape_ellipsoid *(meshname) dia_x dia_y dia_z cpos_x cpos_y cpos_z*
Set a prolate ellipsoid shape in given mesh (focused mesh if not specified) with given diameters (x, y, z) at given centre position coordinates (x, y, z).

Script return values: shape object definition as name dim_x dim_y dim_z cpos_x cpos_y cpos_z rot_theta rot_phi repeat_x repeat_y repeat_z disp_x disp_y disp_z method

shape_get *(meshname (quantity)) name dim_x dim_y dim_z cpos_x cpos_y cpos_z rot_psi rot_theta rot_phi repeat_x repeat_y repeat_z disp_x disp_y disp_z method ...*
Get average value from a shape in given mesh (focused mesh if not specified), depending on currently displayed quantites, or the named quantity if given (see output of display command for possible quantities), where shape definition is shown in shape_set command help. NOTE: this command does not work yet with CUDA enabled, will be finished in a future version.

shape_method *method*
Set modifier for shape generator commands: method to use when applying shape as string literal: add or sub.

Script return values: name

shape_pyramid *(meshname) len_x len_y len_z cpos_x cpos_y cpos_z*

Set a pyramid shape in given mesh (focused mesh if not specified) with given lengths (x, y, z) at given centre position coordinates (x, y, z).

Script return values: shape object definition as name dim_x dim_y dim_z cpos_x cpos_y cpos_z rot_theta rot_phi repeat_x repeat_y repeat_z disp_x disp_y disp_z method

shape_rect *(meshname) len_x len_y cpos_x cpos_y (z_start z_end)*

Set a rectangle shape in given mesh (focused mesh if not specified) with given x and y lengths at given centre position with x and y coordinates. Optionally specify z start and end coordinates, otherwise entire mesh thickness used.

Script return values: shape object definition as name dim_x dim_y dim_z cpos_x cpos_y cpos_z

shape_repetitions *x y z*

Set modifier for shape generator commands: number of shape repetitions along x, y, z.

Script return values: value

shape_rotation *psi theta phi*

Set modifier for shape generator commands: rotation using psi (around y), theta (around x) and phi (around z) in degrees.

shape_set *(meshname) name dim_x dim_y dim_z cpos_x cpos_y cpos_z rot_psi rot_theta rot_phi repeat_x repeat_y repeat_z disp_x disp_y disp_z method ...*

General command for setting a shape: meant to be used with scripted simulations, not directly from the console. Can be used with the return data from shape_... commands, and can take multiple elementary shapes to form a composite object. Set a named shape in given mesh (focused mesh if not specified) with given dimensions (x, y, z) at given centre position coordinates (x, y, z).

shape_setangle *(meshname) name dim_x dim_y dim_z cpos_x cpos_y cpos_z rot_psi rot_theta rot_phi repeat_x repeat_y repeat_z disp_x disp_y disp_z method ... theta polar*

Set magnetization angle (theta - polar, phi - azimuthal) in degrees for given shape identifier (or composite shape) in given mesh (focused mesh if not specified). The shape identifier is returned by one of the shape_... commands, also same parameters taken by the shape_set command. As with shape_set this command is intended for scripted simulations only.

shape_setparam *(meshname) paramname name dim_x dim_y dim_z cpos_x cpos_y cpos_z rot_psi rot_theta rot_phi repeat_x repeat_y repeat_z disp_x disp_y disp_z method ... scaling_value*
Set material parameter scaling value in given shape only, in given mesh (focused mesh if not specified). If you want to set effective parameter value directly, not as a scaling value, then set base parameter value (setparam) to 1 and the real parameter value through this command.

shape_tetrahedron *(meshname) len_x len_y len_z cpos_x cpos_y cpos_z*
Set a tetrahedron shape in given mesh (focused mesh if not specified) with given lengths (x, y, z) at given centre position coordinates (x, y, z).

Script return values: shape object definition as name dim_x dim_y dim_z cpos_x cpos_y cpos_z rot_theta rot_phi repeat_x repeat_y repeat_z disp_x disp_y disp_z method

shape_torus *(meshname) len_x len_y len_z cpos_x cpos_y cpos_z*
Set a torus shape in given mesh (focused mesh if not specified) with given lengths (x, y, z) at given centre position coordinates (x, y, z).

Script return values: shape object definition as name dim_x dim_y dim_z cpos_x cpos_y cpos_z rot_theta rot_phi repeat_x repeat_y repeat_z disp_x disp_y disp_z method

shape_triangle *(meshname) len_x len_y cpos_x cpos_y (z_start z_end)*
Set a triangle shape in given mesh (focused mesh if not specified) with given x and y lengths at given centre position with x and y coordinates. Optionally specify z start and end coordinates, otherwise entire mesh thickness used.

Script return values: shape object definition as name dim_x dim_y dim_z cpos_x cpos_y cpos_z rot_theta rot_phi repeat_x repeat_y repeat_z disp_x disp_y disp_z method

shearstrainequation *(meshname) text_equation*
Set equation for shear strain components, using a vector text equation (3 equations separated by commas) in given meshname (focused mesh if not specified). This will disable the elastodynamics solver in all meshes.

shiftcamorigin *dX dY*
Shift camera origin using dX dY. This is the same as left mouse hold and drag on mesh viewer.

meshrect *meshname shift*
Shift the dipole mesh by the given amount (will only take effect when called on a dipole mesh). shift is specified using x y z coordinates.

shiftglobalfield *shift*
Shift rectangle of global field (previously loaded with loadovf2field on supermesh).

Script return values: rect (global field rectangle)

showa *(meshname)*
Show predicted exchange stiffness (J/m) value for current focused mesh (must be atomistic; focused mesh if not specified), using formula A = J*n/2a, where n is the number of atomic moments per unit cell, and a is the atomic cell size.

Script return values: A

showdata *(meshname) dataname (rectangle)*
Show value(s) for dataname. If applicable specify meshname and rectangle (m) in mesh. If not specified and required, focused mesh is used with entire mesh rectangle.

Script return values: varies

showk *(meshname)*
Show predicted uniaxial anisotropy (J/m^3) constant value for current focused mesh (must be atomistic; focused mesh if not specified), using formula K = k*n/a^3, where n is the number of atomic moments per unit cell, and a is the atomic cell size.

Script return values: A

showlengths *(meshname)*
Calculate a number of critical lengths for the focused mesh (must be ferromagnetic; focused mesh if not specified) to inform magnetization cellsize selection. lex = sqrt(2 A / mu0 Ms^2) : exchange length, l_Bloch = sqrt(A / K1) : Bloch wall width, l_sky = PI D / 4 K1 : Neel skyrmion wall width.

showmcells *(meshname)*
Show number of discretisation cells for magnetization for focused mesh (must be ferromagnetic; focused mesh if not specified).

Script return values: n

showms *(meshname)*
Show predicted saturation magnetization (A/m) value for current focused mesh (must be atomistic; focused mesh if not specified), using formula Ms = mu_s*n/a^3, where n is the number of atomic moments per unit cell, and a is the atomic cell size.

Script return values: Ms

showtc *(meshname)*
Show predicted Tc value (K) for current focused mesh (must be atomistic; focused mesh if not specified), using formula Tc = J*e*z/3kB, where e is the spin-wave correction factor, and z is the coordination number.

Script return values: Tc

skyposdmul *(meshname) multiplier*
Set skyrmion diameter multiplier for given meshname (focused mesh if not specified) which determines skypos tracking rectangle size. Default is 2.0, i.e. tracking rectangle side is twice the diameter along each axis. Reduce if skyrmion density is too large, but at the risk of losing skyrmion tracking (recommended to keep above 1.2).

Script return values: multiplier

skyrmion *(meshname) core chirality diameter position*
Create an idealised Neel-type skyrmion with given diameter and centre position in the x-y plane (2 relative coordinates needed only) of the given mesh (focused mesh if not specified). Core specifies the skyrmion core direction: -1 for down, 1 for up. Chirality specifies the radial direction rotation: 1 for towards core, -1 away from core. For diameter and position use metric units.

skyrmionbloch *(meshname) core chirality diameter position*
Create an idealised Bloch-type skyrmion with given diameter and centre position in the x-y plane (2 relative coordinates needed only) of the given mesh (focused mesh if not specified). Core specifies the skyrmion core direction: -1 for down, 1 for up. Chirality specifies the radial direction rotation: 1 for clockwise, -1 for anti-clockwise. For diameter and position use metric units.

skyrmionpreparemovingmesh *(meshname)*
Setup the named mesh (or focused mesh) for moving skyrmion simulations: 1) set movingmesh trigger, 2) set domain wall structure, 3) set dipoles left and right to remove end magnetic charges, 4) enable strayfield module.

ssolverconfig *s_convergence_error (s_iters_timeout)*
Set spin-transport solver convergence error and iterations for timeout (if given, else use default).

Script return values: s_convergence_error s_iters_timeout

stages
Shows list of currently set simulation stages and available stage types.

Script return values: number of set stages

startupscriptserver *status*

Set startup script server flag.

startupupdatecheck *status*

Set startup update check flag.

statictransportsolver *status*

If static transport solver is set, the transport solver is only iterated at the end of a stage or step. You should set a high iterations timeout if using this mode.

Script return values: status

stochastic

Shows stochasticity settings : stochastic cellsize for each mesh and related settings.

stop

Stop simulation without resetting it.

stopscript

Stop any currently running Python script.

strainequation *(meshname) text_equation*

Set equation for diagonal strain components, using a vector text equation (3 equations separated by commas) in given meshname (focused mesh if not specified). This will disable the elastodynamics solver in all meshes.

surfacefix *(meshname) rect_or_face*

Add a fixed surface for the elastodynamics solver. A plane rectangle may be specified in absolute coordinates, which should intersect at least one mesh face, which will then be set as fixed. Alternatively an entire mesh face may be specified using a string literal: -x, x, -y, y, -z, or z. In this case a named mesh will also be required (focused mesh if not given).

surfacestress *(meshname) rect_or_face vector_equation*

Add an external stress surface for the elastodynamics solver. A plane rectangle may be specified in absolute coordinates, which should intersect at least one mesh face, which will then be set as fixed. Alternatively an entire mesh face may be specified using a string literal: -x, x, -y, y, -z, or z. In this case a named mesh will also be required (focused mesh if not given). The external stress is specified using a vector equation, with variables x, y, t. Coordinates x, y are relative to specified rectangle, and t is the time.

surfroughenjagged *(meshname) depth spacing (seed (sides))*

Roughen given mesh (focused mesh if not specified) surfaces using a jagged pattern to given depth (m) and peak spacing (m). Roughen both sides by default, unless sides is specified as -z or z (string literal). The seed is used for the pseudo-random number generator, 1 by default.

tamrequation *meshname text_equation*

Set conductivity equation for TAMR effect (must be a mesh with transport module added) - the text equation uses parameters TAMR, d1, d2, d3. Here TAMR = tamr / 100 is the ratio, where tamr is the material parameter set as a percentage value. di = eai.m, i = 1, 2, 3, where eai is the crystalline axis and m is the magnetization direction. Call with empty equation string to clear any currently set equation and use default conductivity formula: cond / elC = 1 / (1 + TAMR * (1 - d1*d1)), where elC is the base conductivity (material parameter). Note, the formula set always multiplies elC to obtain conductivity due to TAMR.

tau *(meshname) tau_11 tau_22 (tau_12 tau_21)*

Set ratio of exchange parameters to critical temperature (Neel) for antiferromagnetic mesh (all antiferromagnetic meshes if not specified). tau_11 and tau_22 are the intra-lattice contributions, tau_12 and tau_21 are the inter-lattice contributions.

Script return values: tau_11 tau_22 tau_12 tau_21

tcellsize *(meshname) value*

Change cellsize of meshname for thermal conduction (m). The cellsize can be specified as: hx hy hz, or as: hxyz. If meshname not given use the focused mesh.

Script return values: cellsize - return thermal conduction cellsize of focused mesh, or of meshname if given.

temperature *(meshname) value*

Set mesh base temperature (all meshes if meshname not given) and reset temperature. Also set ambient temperature if Heat module added. If the base temperature setting has a spatial dependence specified through cT, this command will take it into account but only if the Heat module is added. If you want the temperature to remain fixed you can still have the Heat module enabled but disable the heat equation by setting the heat dT to zero (setheatdt 0).

Script return values: value - temperature value for focused mesh.

threads *number*

Set number of threads to use for cuda 0 computations. Value of zero means set maximum available.

Script return values: num_threads

tmodel *(meshname) num_temperatures*

Set temperature model (determined by number of temperatures) in given meshname (focused mesh if not specified). Note insulating meshes only allow a 1-temperature model.

tmrtype *(meshname) setting*

Set formula used to calculate TMR angle dependence in named mesh (all meshes if not given). setting = 0: RA = (RAp + (RAap - RAp) * (1 - cos(theta))/2). setting = 1 (Slonczewski form, default): RA = 2*RAp/[(1 + RAp/RAap) + (1 - RAp/RAap)*cos(theta)]

Script return values: setting

tsolverconfig *convergence_error (iters_timeout)*

Set transport solver convergence error and iterations for timeout (if given, else use default).

Script return values: convergence_error iters_timeout

updatemdb

Switch to, and update the local materials database from the online shared materials database.

updatescreen

Updates all displayed values on screen and also refreshes.

vecrep *(meshname) vecreptype*

Set representation type for vectorial quantities in named mesh, or supermesh (focused mesh if not specified). vecreptype = 0 (full), vecreptype = 1 (x component), vecreptype = 2 (y component), vecreptype = 3 (z component), vecreptype = 4 (direction only), vecreptype = 5 (magnitude only).

versionupdate *action (target_version)*

Incremental version update command. action = check : find if incremental update is available for current program version. action = download : download available incremental updates if any (or up to target version if specified). action = install : install incremental updates if any (or up to target version if specified), downloading if needed. action = rollback : roll back to specified target version if possible.

vortex *(meshname) longitudinal rotation core (rectangle)*

Create a vortex domain wall with settings: longitudinal (-1: tail-to-tail, 1: head-to-head), rotation (-1: clockwise, 1: counter-clockwise), core (-1: down, 1: up). The vortex may be set in the given rectangle (entire mesh if not given), in the given mesh (focused mesh if not specified).

References

There are a number of articles which cover various parts of the software.

General

- S. Lepadatu, “Boris computational spintronics — High performance multi-mesh magnetic and spin transport modeling software”, *Journal of Applied Physics* **128**, 243902 (2020)

Differential equation solvers

- S. Lepadatu “Speeding Up Explicit Numerical Evaluation Methods for Micromagnetic Simulations Using Demagnetizing Field Polynomial Extrapolation” *IEEE Transactions on Magnetics* **58**, 1 (2022)

Multilayered convolution

- S. Lepadatu, “Efficient computation of demagnetizing fields for magnetic multilayers using multilayered convolution” Journal of Applied Physics **126**, 103903 (2019)

Parallel Monte Carlo algorithm

- S. Lepadatu, G. Mckenzie, T. Mercer, C.R. MacKinnon, P.R. Bissell, “Computation of magnetization, exchange stiffness, anisotropy, and susceptibilities in large-scale systems using GPU-accelerated atomistic parallel Monte Carlo algorithms” Journal of Magnetism and Magnetic Materials **540**, 168460 (2021)

Micromagnetic Monte Carlo algorithm (with demagnetizing field parallelization)

- S. Lepadatu “Micromagnetic Monte Carlo method with variable magnetization length based on the Landau–Lifshitz–Bloch equation for computation of large-scale thermodynamic equilibrium states” Journal of Applied Physics **130**, 163902 (2021)

Roughness effective field

- S. Lepadatu, "Effective field model of roughness in magnetic nano-structures" Journal of Applied Physics **118**, 243908 (2015)

Heat flow solver, LLB and 2-sublattice LLB

- S. Lepadatu, "Interaction of Magnetization and Heat Dynamics for Pulsed Domain Wall Movement with Joule Heating" Journal of Applied Physics **120**, 163908 (2016)
- S. Lepadatu "Emergence of transient domain wall skyrmions after ultrafast demagnetization" Physical Review B **102**, 094402 (2020)

Spin transport solver

- S. Lepadatu, "Unified treatment of spin torques using a coupled magnetisation dynamics and three-dimensional spin current solver" Scientific Reports **7**, 12937 (2017)
- S. Lepadatu, "Effect of inter-layer spin diffusion on skyrmion motion in magnetic multilayers" Scientific Reports **9**, 9592 (2019)
- C.R. MacKinnon, S. Lepadatu, T. Mercer, and P.R. Bissell "Role of an additional interfacial spin-transfer torque for current-driven skyrmion dynamics in chiral magnetic layers" Physical Review B **102**, 214408 (2020)
- C.R. MacKinnon, K. Zeissler, S. Finizio, J. Raabe, C.H. Marrows, T. Mercer, P.R. Bissell, and S. Lepadatu, "Collective skyrmion motion under the influence of an additional interfacial spin-transfer torque" Scientific Reports **12**, 10786 (2022)
- S. Lepadatu and A. Dobrynin, "Self-consistent computation of spin torques and magneto-resistance in tunnel junctions and magnetic read-heads with metallic pinhole defects" Journal of Physics: Condensed Matter **35**, 115801 (2023)

Elastodynamics solver with thermoelastic effect and magnetostriction

- S. Lepadatu, "All-optical magneto-thermo-elastic skyrmion motion" arXiv:2301.07034v2 (2023)