NMAG User Manual Documentation

Release 0.2.1

Hans Fangohr, Thomas Fischbacher, Matteo Franchin
Giuliano Bordignon, Jacek Generowicz, Andreas Knittel
Michael Walter, Maximilian Albert

January 13, 2012

CONTENTS

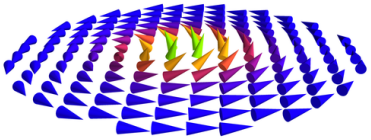
Authors Hans Fangohr, Thomas Fischbacher, Matteo Franchin, Giuliano Bordignon, Jacek Generowicz, Andreas Knittel, Michael Walter, Maximilian Albert

Licence GNU General Public License (GPL) version 2

Version 0.2.1 (467:d7c316deb156) released on Fri Jan 13 09:38:20 2012

Date Manual generated on 2012-01-13 at 09:39.

Home page http://nmag.soton.ac.uk



CHAPTER

ONE

INTRODUCTION

Nmag is a flexible finite element micromagnetic simulation package with an user interface based on the Python programming language.

If you use Nmag in your published work, please cite:

- Thomas Fischbacher, Matteo Franchin, Giuliano Bordignon, and Hans Fangohr. *A Systematic Approach to Multiphysics Extensions of Finite-Element-Based Micromagnetic Simulations: Nmag*, in IEEE Transactions on Magnetics, **43**, 6, 2896-2898 (2007). (Available online)

1.1 Nmag Philosophy

Many specialized simulation codes used in research today consist of a highly specialized core application which initially was written to simulate the behaviour of some very specific system. Often, the core application then evolved into a more broadly applicable tool through the introduction of additional parameters. Some simulation codes reach a point where it becomes evident that they need an amount of flexibility that can only be provided by including some script programming capabilities.

The approach underlying Nmag turns this very common pattern of software evolution (which we also have seen in web browsers, CAD software, word processors, etc) on its head: rather than gradually providing more and more flexibility in an ad-hoc manner through adding configuration parameters, slowly evolving into an extensive specialized programming language, Nmag starts out as an extension to a widely used programming language (Python) from which it gains all its flexibility and evolves towards more specialized notions to conveniently define and study the properties of very specific physical systems [1].

The main advantage of this approach is two-fold: first, we do not gradually evolve another ad-hoc (and potentially badly implemented) special purpose programming language. Second, by drawing upon the capabilities of a well supported existing framework for flexibility, we get a lot of additional power for free: the user can employ readily available and well supported Python libraries for tasks such as data post-processing and analysis, e.g. generating images for web pages etc. In addition to this, some users may benefit from the capability to use Nmag interactively from a command prompt, which can be very helpful during the development phase of an involved simulation script [2].

The disadvantage is of course that a novice user may be confronted with much more freedom than he can handle. We try to cope with this issue by providing a collection of example scripts (in the *Guided Tour*) for the most common applications that only need very slight modification for basic use (e.g. changing of the mesh filename or material parameters).

At present, Nmag is based on the Python programming language. This seems to be a somewhat reasonable choice at present, as Python is especially friendly towards casual users who do not want to be forced to first become expert programmers before they can produce any useful results. Furthermore, Python is quite widespread and widely supported these days.

[1] Thomas Fischbacher, Matteo Franchin, Giuliano Bordignon,, Hans Fangohr, *A Systematic Approach to Multiphysics Extensions of Finite-Element-Based Micromagnetic Simulations: Nmag*, IEEE Transactions on Magnetics **43**, 6, 2896-2898 (2007), online at http://eprints.soton.ac.uk/46725/

[2] Thomas Fischbacher, Matteo Franchin, Giuliano Bordignon, Andreas Knittel, Hans Fangohr, *Parallel execution and scriptability in micromagnetic simulations*, Journal of Applied Physics **105**, 07D527 (2009), online at http://link.aip.org/link/?JAPIAU/105/07D527/1

1.2 How to read this document

We suggest you follow the *Guided Tour* through a number of examples to get a quick overview of what nmag looks like in real use, and to see examples that can be used to carry out typical simulations. We provide a number of skeletons that are easily adapted to specific systems which show how to compute hysteresis loops, do energy minimisation, or compute time evolution.

The *Command reference* section explains the relevant commands provided by Nmag in full detail. This should be especially useful to advanced users who want to design sophisticated simulation scripts in Python.

If you are new to micromagnetic modelling, you may want to start with the *Mini tutorial micromagnetic modelling*.

1.3 Development status

The first Nmag release was late in 2007, and many bugs have been fixed since then. Having said that, without doubt there are bugs left in the system, and there is a long list of wishes for extra features, changes, improvements.

Currently, there is no significant amount of funding or man power available to support Nmag users or develop it further. The software should thus be seen to be provided as is.

Should you use Nmag for your work, please cite

- Thomas Fischbacher, Matteo Franchin, Giuliano Bordignon, and Hans Fangohr, *A Systematic Approach to Multiphysics Extensions of Finite-Element-Based Micromagnetic Simulations: Nmag*, IEEE Transactions on Magnetics **43**, 6, 2896-2898 (2007), online: preprint and http://dx.doi.org/10.1109/TMAG.2007.893843

to demonstrate the value of open-source infrastructure in the community. (We should also cite the corresponding recommended publications when using OOMMF, Magpar, Mumax, MicroMagnum etc).

1.4 Mailing list

If you are using nmag, we recommend that you subscribe to at least one of these following two lists. If you have a question about how to use the software, we suggest you subscribe to *nmag-users*, and post it there.

1.4.1 nmag-announce

nmag-announce@lists.soton.ac.uk is a low traffic read-only mailing list which will broadcast updates of nmag and any other relevant news.

To subscribe to this list, send an email to nmag-announce-request@lists.soton.ac.uk with an empty subject and the word `subscribe` in the body of the email.

The **archives** can be found and searched at http://groups.google.com/group/nmag-announce.

1.4.2 nmag-users

nmag-users@lists.soton.ac.uk is a mailing list to discuss the use of nmag, and for users to support users. Any announcements to `nmag-announce` will also be sent to this mailing list.

To subscribe to this list, send an email to nmag-users-request@lists.soton.ac.uk with an empty subject and the word `subscribe` in the body of the email.

Information about how to unsubscribe are provided with the welcome message once you have subscribed.

The **archives** can be found and searched at http://groups.google.com/group/nmag-users.

1.5 Support

Support will be provided within our limited resources (which may be None). After consulting the manual, please feel free to use the *Mailing list* `nmag-users@lists.soton.ac.uk` to seek advice, or contact the *nmag team* directly.

1.6 License and Disclaimer

CHAPTER

TWO

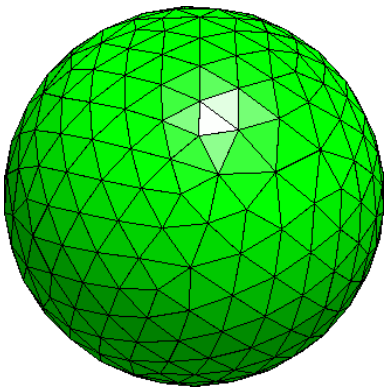
GUIDED TOUR

We present a number of worked out examples that are explained in detail and should cover most of the usual applications. (You may also want to check the *Frequently Asked Questions*.)

2.1 Example: Demag field in uniformly magnetised sphere

This is the most basic example that computes the demagnetisation field in an uniformly magnetised sphere. For this simple system, the exact result is known analytically: the demag field vector has to be equal to minus one-third of the magnetisation vector, everywhere.

When using finite element calculations, a crucial (and non-trivial) part of the work is the *finite element mesh generation*. We provide a very small mesh for this example (`sphere1.nmesh.h5`) which was generated with Netgen (from this `geometry file`). This gives us a sphere of radius 10nm.



We can then use the following nmag script `sphere1.py`:

```
import nmag
from nmag import SI

#create simulation object
sim = nmag.Simulation()

# define magnetic material
Py = nmag.MagMaterial(name = 'Py',
                      Ms = SI(1e6, 'A/m'),
                      exchange_coupling = SI(13.0e-12, 'J/m'))

# load mesh
sim.load_mesh('sphere1.nmesh.h5',
              [('sphere', Py)],
              unit_length = SI(1e-9, 'm'))

# set initial magnetisation
```

```
sim.set_m([1,0,0])

# set external field
sim.set_H_ext([0,0,0], SI('A/m'))

# Save and display data in a variety of ways
sim.save_data(fields='all') # save all fields spatially resolved
                            # together with average data

# sample demag field through sphere
for i in range(-10,11):
    x = i*1e-9                      #position in metres
    H_demag = sim.probe_subfield_siv('H_demag', [x,0,0])
    print "x =", x, ": H_demag = ", H_demag
```

To execute this script, we have to give its name to the *nsim* executable, for example (on linux):

```
$ nsim sphere1.py
```

Some simulations produce output files which nsim will refuse to overwrite when run for a second time. The rationale is that big simulations may have to run for a long time and so, there should be a safeguard against accidental destruction of data.

In order to re-run a simulation, removing all old output data files, the extra option `--clean` should be given, as in:

```
$ nsim sphere1.py --clean
```

Let us discuss the `sphere1.py` script step by step.

2.1.1 Importing nmag

First we need to import the nmag module, and any subpackages of nmag that we want to use. (In this basic example, this is just the SI module for dimensionful physical quantities).

```
import nmag
from nmag import SI
```

2.1.2 Creating the simulation object

Next, we need to create a simulation object. This will contain and provide information about our physical system.

```
sim = nmag.Simulation()
```

2.1.3 Defining (magnetic) materials

After importing the nmag module into Python's workspace and creating the simulation object `sim`, we need to define a material using `nmag.MagMaterial`. We give it a name (as a Python string) which in this case we choose to be `"Py"` (a common abbreviation for PermAlloy) and we assign a saturation magnetisation and an exchange coupling strength.

```
Py = nmag.MagMaterial(name = 'Py',
                      Ms = SI(1e6, 'A/m'),
                      exchange_coupling = SI(13.0e-12, 'J/m'))
```

The name of the material is important, as we may want to simulate systems made up of multiple different materials, and the material name will be used as a postfix to the name of some *Fields and subfields*. The output files will also use that name to label output data. Names must be alphanumeric (i.e. formed exclusively out of the characters in the set `0-9\_a-zA-Z`) here.

Rather than representing dimensionful physical quantities as numbers, nmag uses a special object class, the "SI object". The underlying rationale is that this allows automated detection of mismatches of physical dimensions. If some physical parameter is given to nmag in a dimension different from the expected one, nmag will detect this and report an error. Also, any nmag output [e.g. a three-dimensional VTK visualisation file] will provide a sufficient amount of contextual information to clarify the physical meaning (i.e. dimensions) of numerical data.

We thus express the saturation magnetisation in Ampere per meter (`Ms = SI(1e6,"A/m")`) and the exchange coupling constant (often called A in micromagnetism) in Joules per meter (`exchange_coupling = SI(13.0e-12, "J/m")`). (Note that these are not the true physical parameters of PermAlloy, but have been chosen ad hoc for the sake of providing a simple example!)

2.1.4 Loading the mesh

The next step is to load the mesh.

```
sim.load_mesh('sphere1.nmesh.h5',
              [('sphere', Py)],
              unit_length = SI(1e-9, 'm'))
```

The first argument is the file name (`"sphere1.nmesh.h5"`). The second argument is a list of tuples which describe the domains (also called regions) within the mesh. In this example we have a one-element list containing the 2-tuple `("sphere", Py)`. The left element of this pair, `"sphere"`, is a string (of the user's choice) and this is the name given to mesh region 1 (i.e. the space occupied by all simplices that have the region id 1 in the mesh file).

[This information is currently only used for debugging purposes (such as when printing the simulation object).]

The second part of the tuple is the `MagMaterial` object that has been created in *Defining (magnetic) materials* and bound to the variable `Py`. This object determines the material properties of the material in this domain; in this example, we have specified the properties of PermAlloy.

The third argument to *load_mesh* is an *SI object* which defines what physical distance should be associated with the length 1.0 as given in the mesh file. In this example, the mesh has been created in nanometers, i.e. the distance 1.0 in the mesh file should correspond to 1 nanometer in the real world. We thus use a SI object representing 1 nm.

2.1.5 Setting the initial magnetisation

To set the initial magnetisation, we use the *set_m* method.

```
sim.set_m([1,0,0])
```

The field `m` describes the direction of magnetisation (as a field of normalised vectors) whereas the field `M` contains the magnetisation with its proper magnitude. So, `|M|` is the saturation magnetisation (in Amperes per meter), whereas `m` is dimensionless with `|m|=1.0`. There are different ways to set a particular magnetisation, in the simplest case of a homogeneously magnetised body, it is sufficient to provide the magnetisation vector. So, in this example, we provide a unit vector pointing in positive x-direction. (We could provide a vector with non-normalised magnitude, which would be normalised automatically. This is convenient for, say, setting an initial magnetisation in the x-y-plane with a 45 degree angle towards the x axis by specifying `[1,1,0]`).

2.1.6 Setting the external field

We can set the external field using the *set_H_ext* command

```
sim.set_H_ext([0,0,0], SI('A/m'))
```

In contrast to *set_m*, this method takes two arguments. The first defines numerical values for the direction and magnitude of the external field. The second determines the meaning of these numerical values using an SI object. Suppose we would like an external field of 1e6 A/m acting in the y-direction, then the

command would read: `sim.set_H_ext([0,1e6,0],SI(1,"A/m"))`. However, we could also use `sim.set_H_ext([0,1,0],SI(1e6,"A/m"))`.

The default value for the external field is [0,0,0] A/m, so for this example, we could have omitted the *set_H_ext* command altogether.

2.1.7 Extracting and saving data

We have three different ways of extracting data from the simulation:

1. saving averaged values of fields (which can be analysed later)
2. saving spatially resolved fields (which can be analysed later)
3. extracting field values at arbitrary positions from within the running program

In this basic example, we demonstrate the use of all three methods:

Saving averaged data

```
sim.save_data()
```

The *save_data* method writes (spatial) averages of all fields (see *Fields and subfields*) into a text file (which will be named `sphere1_dat.ndt`, see below). This file is best analysed using the *ncol* tool but can also just be read with a text editor. The format follows OOMMF's `odt` file format: every row corresponds to one snapshot of the system (see *save_data*).

The function can also be called with parameters to save spatially resolved field data (see *Saving spatially resolved data*).

The first and second line in the data file are headers that explain (by column) the physical quantities (and their dimensions).

The *ncol* tool allows to extract particular columns easily so that these can be plotted later (useful for hysteresis loop studies). In this example we have only one "timestep": there only is one row of data in this file. We will therefore discuss this in more detail in a subsequent example.

Extracting arbitrary data from the running program

The line

```
H_demag = sim.probe_subfield_siv( 'H_demag', [x,0,0] )
```

obtains the demagnetisation field (see *Fields and Subfields in Nmag*) at position (x,0,0). The suffix "_siv" to this function means that both positions and return values will be given as SI values.

The for-loop in the program (which iterates `x` in the range from -10*1e-9 to 10*1e-9 in steps of 1e-9) produces the following output

```
x = -1e-08 : H_demag =  None
x = -9e-09 : H_demag =  [-329655.76203912671, 130.62999726469423, 194.84338557811344]
x = -8e-09 : H_demag =  [-329781.46587966662, 66.963624669268853, 137.47161381890737]
x = -7e-09 : H_demag =  [-329838.57852402801, 181.46249265908259, 160.61298054099865]
x = -6e-09 : H_demag =  [-329899.63327447395, 131.06488858715838, 71.383139326493094]
x = -5e-09 : H_demag =  [-329967.79622912291, 82.209856975234786, -16.893046828024836]
x = -4e-09 : H_demag =  [-329994.67306536058, 61.622521557150371, -34.433041910642359]
x = -3e-09 : H_demag =  [-329997.62759666931, 23.222244635691535, -65.991127111463769]
x = -2e-09 : H_demag =  [-330013.90370482224, 10.11035370824321, -61.358763616681067]
x = -1e-09 : H_demag =  [-330023.50844056415, -6.9714476825652287, -54.900260456937708]
x = 0.0   : H_demag =  [-330030.98847923806, -26.808832466764223, -48.465748009067141]
x = 1e-09 : H_demag =  [-330062.38479507214, -38.660812022013424, -42.83439139610747]
x = 2e-09 : H_demag =  [-330093.78111090627, -50.512791577262625, -37.20303478314781]
```

```
x = 3e-09 : H_demag =  [-330150.72580001026, -64.552170478617398, -23.120555702674721]
x = 4e-09 : H_demag =  [-330226.19050178828, -77.236085707456397, -5.5373829923226916]
x = 5e-09 : H_demag =  [-330304.59300913941, -90.584413821813229, 14.090609104026118]
x = 6e-09 : H_demag =  [-330380.1392610991, -115.83746059068679, 37.072085708324757]
x = 7e-09 : H_demag =  [-330418.85831447819, -122.47512022500726, 62.379121138009992]
x = 8e-09 : H_demag =  [-330476.40747455234, -110.84257225592108, 108.06217226524763]
x = 9e-09 : H_demag =  [-330500.20126762061, -68.175725285038382, 162.46166752217249]
x = 1e-08 : H_demag =  [-330517.86675206106, -24.351273685146875, 214.40344001233677]
```

At position -1e-8, there is no field defined (this point lies just outside our sphere-mesh) and therefore the value `None` is returned.

We can see how the demagnetisation field varies slightly throughout the sphere. The x-component is approximately a third of the magnetisation, and the y- and z-components are close to zero (as would be expected for a perfectly round sphere).

We mention for completeness that most fields (such as magnetisation, exchange field, anisotropy field etc) are only defined within the region(s) occupied by magnetic material. However, there is a special function *probe_H_demag_siv* to probe the demagnetisation field anywhere in space.

Saving spatially resolved data

The command

```
sim.save_data(fields='all')
```

will save full spatially resolved data on all fields (see *Fields and subfields*) for the current configuration into a file with name `sphere1_dat.h5`. (It will also save the spatially averaged values as described in *Saving averaged data*.) Whenever the *save_data* function is called, it will write the averaged field values into the *Data files (.ndt)* file. This name is, by default, based on the name of the simulation script, but can be overridden with an optional argument to the *Simulation* constructor. The data in this file are kept in some compressed binary format (built on the hdf5 standard) and can be extracted and converted later using the *nmagpp* tool.

For example, we can extract the magnetisation field from this file with the command:

```
$ nmagpp --dump sphere1
```

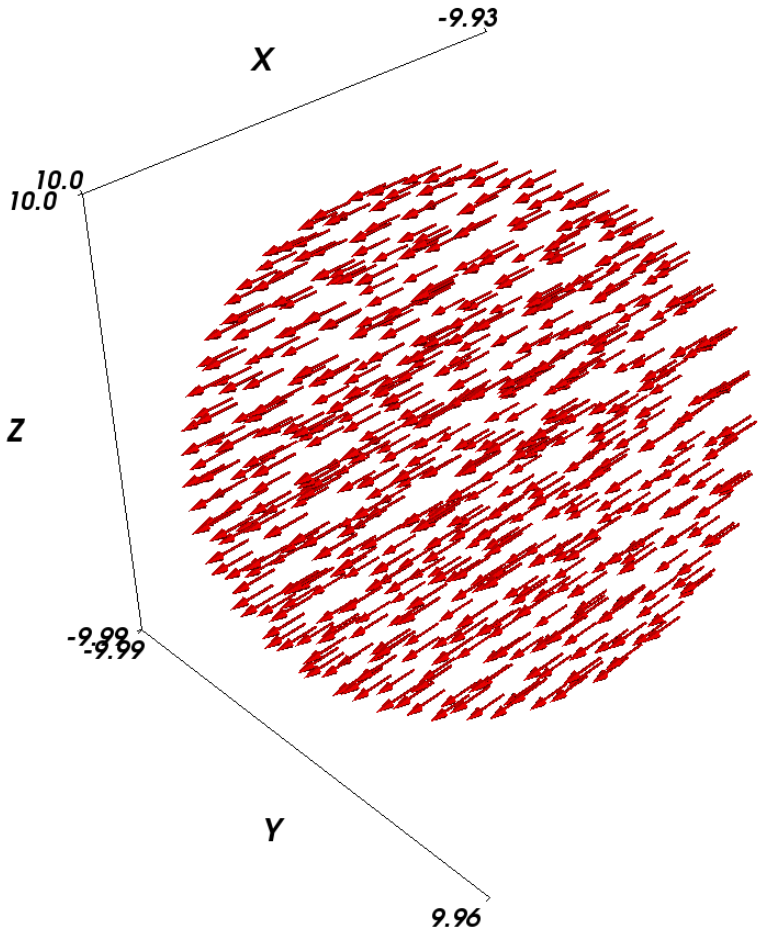
However, here we are interested in creating a *vtk* from the saved data file for visualisation. We use:

```
$ nmagpp --vtk sphere1.vtk sphere1
```

where `sphere1.vtk` is the base name of the vtk file that is to be generated.

In this manual, we use *MayaVi* as the visualisation tool for vtk files but there are others available (see *vtk*).

Starting MayaVi with the command `mayavi -d sphere1-000000.vtk` will load our simulation data. Using the pull-down menu `Visualize -> Modules -> VelocityVector` will then tell MayaVi to display the magnetisation vector field. (Likewise, we can use `Visualize -> Modules -> Axes` to add a 3d coordinate system to the visualization):



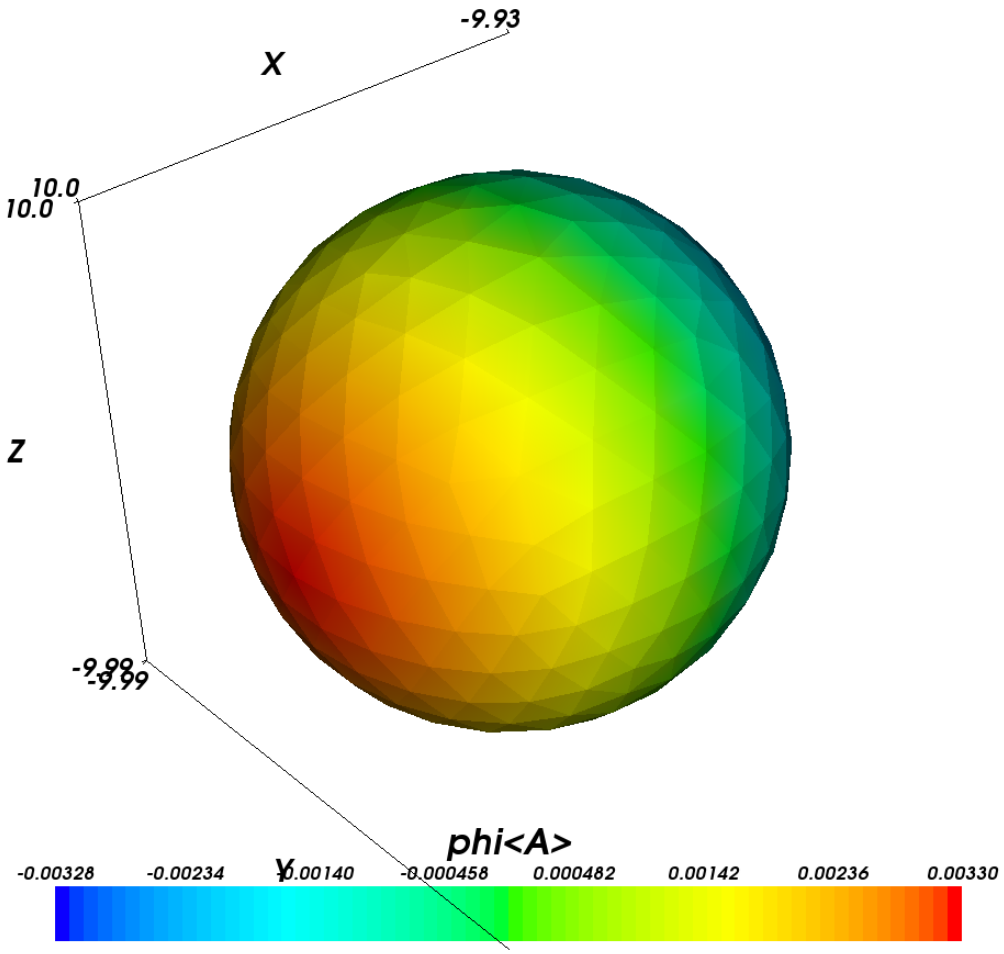
The magnetisation is pointing in positive x-direction because we initialised the magnetisation in this orientation by issuing the command `sim.set_m([1,0,0])`.

The `Configure Data` button in the DataVizManager section of MayaVi's user interface allows to select:

- a vector field and
- a scalar field

which provide the data that is used for subsequent visualisation modules. Above, we have used the `m_Py` vector field.

The demagnetisation field should point in the opposite direction of the magnetisation. However, let's first create a colour-coded plot of the scalar magnetic potential, phi, from which the demag field is computed (by taking its negative gradient):



We first need to select `phi` as the data source for 'scalar' visualisation modules: Through clicking on the `Configure Data` button in the DataVizManager section of MayaVi's user interface, we can select `phi<A>` as the data source for scalar visualisations. (The `<A>` simply indicates that the units of the potential are Ampere).

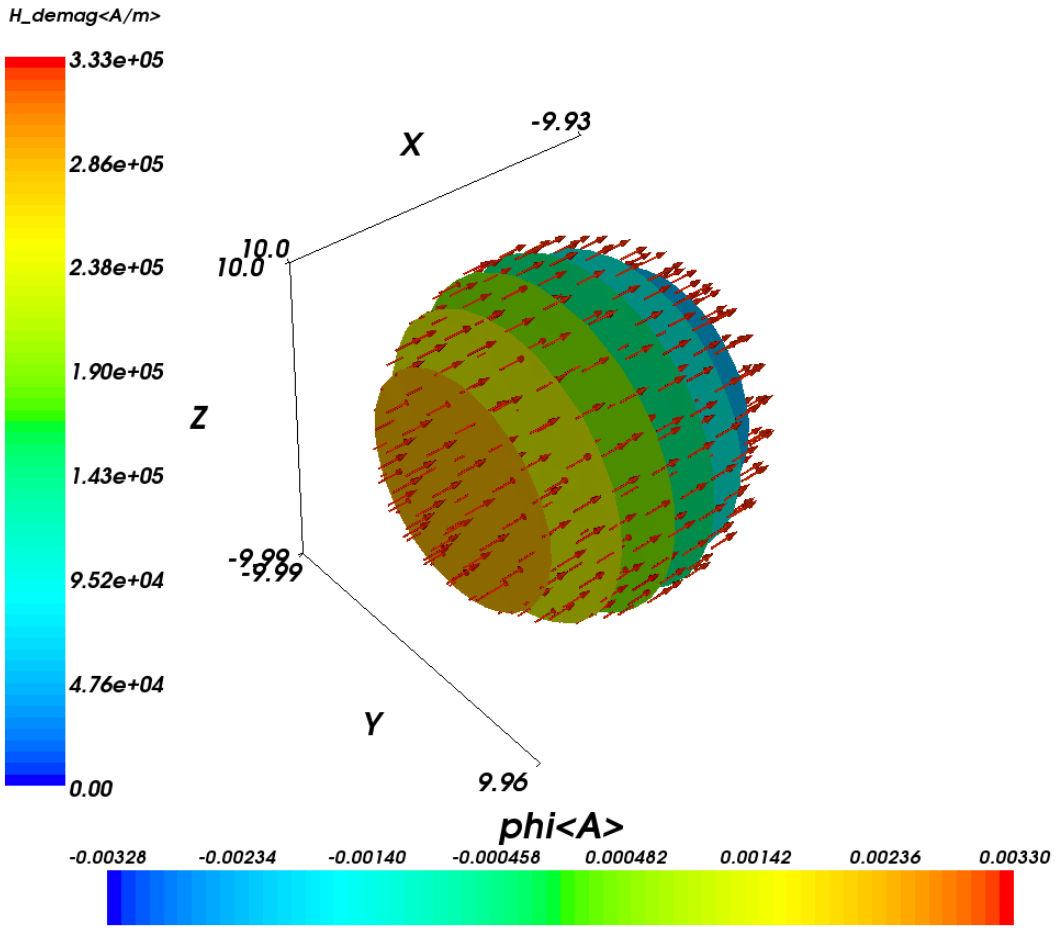
To show the scalar potential, we use the `Visualize->Module->SurfaceMap` module.

We can see that the potential varies along the x-direction. The legend at the bottom of the figure shows the colour code used. We can also see from the legend title that the physical dimension of the potential phi is Ampere (this is the `<A>`).

Unless the user specifies a particular request for physical dimensions, the following rules apply for vtk files:

- position are given in the same coordinates as the mesh coordinates (that is why in this example, the x, y and z axis have values going from -10 to 10).
- all field data are given in SI units.

The next plot shows the demag field (the vectors) together with isosurfaces of the magnetic potential:



It can be seen that the isosurfaces are completely flat planes (i.e. the potential is changing only along x) and the demagnetisation field is perpendicular to the isosurfaces. The color bar on the left refers to the magnitude of the demagnetisation field which is expressed in Ampere per meter, as can be seen from the label `<A/m>`. (Note that all the `H_demag` arrows are colored red as they have identical length.)

2.2 Example 2: Computing the time development of a system

This example computes the time development of the magnetisation in a bar with (x,y,z) dimensions 30 nm x 30 nm x 100 nm. The initial magnetisation is pointing in the [1,0,1] direction, i.e. 45 degrees away from the x axis in the direction of the (long) z-axis. We first show the simulation code and then discuss it in more detail.

2.2.1 Mesh generation

While it is down to the mesh generation software (see also *Finite element mesh generation*) to explain how to generate finite element meshes, we briefly summarize the steps necessary to create a mesh for this example in Netgen, and how to convert it into an `nmesh` mesh.

1. The finite element method requires the domain of interest to be broken down into small regions. Such a subdivision of space is known as a mesh or grid. We use Netgen to create this mesh. Netgen reads a `geometry file` describing the three-dimensional structure. To create the mesh used here, we can start Netgen and load the geometry file by using the menu: `File-> Load Geometry`. We then tell Netgen that we like the edge length to be shorter than 3 by going to `Mesh->Meshing Options->Mesh Size` and enter `3.0` in the `max mesh-size` box. Then a click on the `Generate Mesh` button will generate the mesh. Finally, using `File->Export` will save the mesh as a "*neutral*" file (this is the default)

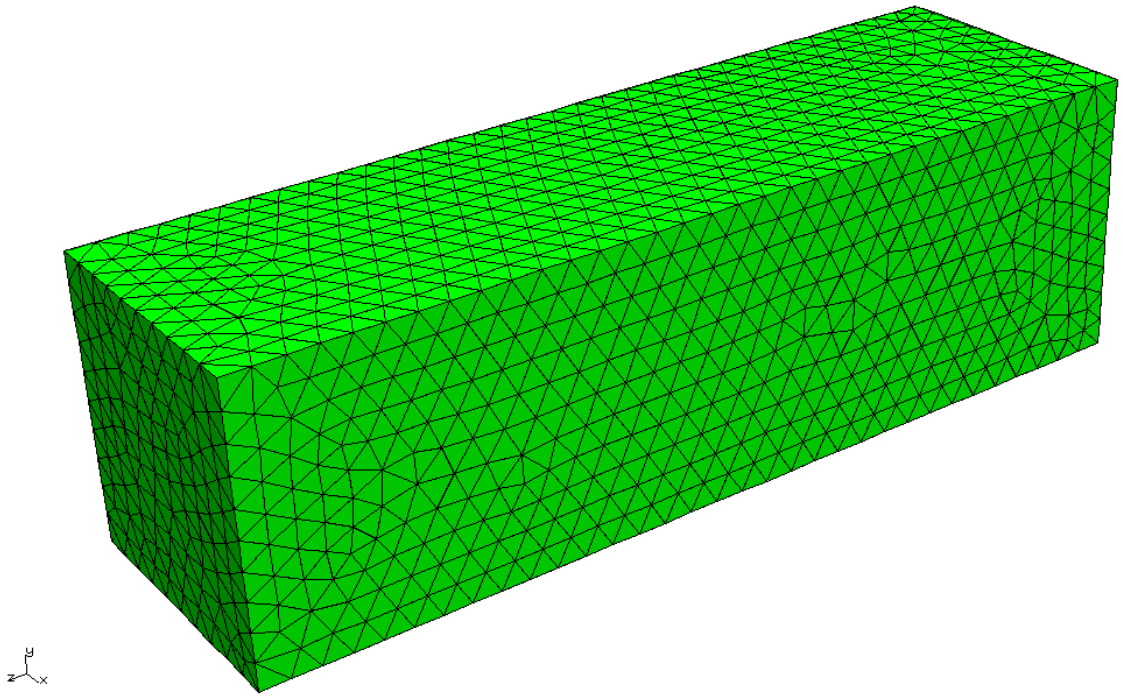
under the name `bar30_30_100.neutral`. (We provide a `gzipped version of this file` for completeness.)

2. This neutral file needs to be converted into a nmesh file. We do this using the command:

   ```
   $ nmeshimport --netgen bar30_30_100.neutral bar30_30_100.nmesh.h5
   ```

   By providing the `.h5` extension, we tell nmeshimport to write a compressed mesh file which is significantly smaller than an ascii file (see *mesh file size*).

The generated mesh looks like this:



We can examine the mesh using *nmeshpp* to obtain information about mesh quality, the statistical distribution of edge lengths, the overall number of points and elements etc.

If you like to script the mesh generation starting from a Netgen geometry file and ending with the nmesh file, you could use (for the example above), the following shell commands:

```
netgen -geofile=bar30_30_100.geo -meshfiletype="Neutral Format" -meshfile=bar30_30_100.neutral -ba
nmeshimport --netgen bar30_30_100.neutral bar30_30_100.nmesh.h5
```

2.2.2 The simulation

Having obtained the mesh file `bar30_30_100.nmesh.h5`, we can use the program `bar30_30_100.py` to run the simulation:

```
import nmag
from nmag import SI

mat_Py = nmag.MagMaterial(name="Py",
                          Ms=SI(0.86e6,"A/m"),
                          exchange_coupling=SI(13.0e-12, "J/m"),
                          llg_damping=0.5)

sim = nmag.Simulation("bar")

sim.load_mesh("bar30_30_100.nmesh.h5",
              [("Py", mat_Py)],
```

```
                    unit_length=SI(1e-9,"m"))

    sim.set_m([1,0,1])

    dt = SI(5e-12, "s")

    for i in range(0, 61):
        sim.advance_time(dt*i)                    #compute time development

        if i % 10 == 0:                           #every 10 loop iterations,
            sim.save_data(fields='all')           #save averages and all
                                                  #fields spatially resolved
        else:
            sim.save_data()                       #otherwise just save averages
```

As in *Example: demag field in uniformly magnetised sphere*, we start by importing nmag and creating the material object.

```
import nmag
from nmag import SI

mat_Py = nmag.MagMaterial(name="Py",
                          Ms=SI(0.86e6,"A/m"),
                          exchange_coupling=SI(13.0e-12, "J/m"),
                          llg_damping=0.5)
```

We set the `llg_damping` parameter to 0.5. As this is a dimensionless parameter, we can pass a number. Alternatively, we may give it as `SI(0.5)`. (Note that in this example, we give the appropriate physical value for the saturisation magnetisation of PermAlloy.)

The next line creates the simulation object:

```
sim = nmag.Simulation("bar")
```

Here, we provide a name for the simulation, which is `bar`. This will be used as the stem of the name of any data files that are being written. If this name is not specified (as in *Example: demag field in uniformly magnetised sphere*), it defaults to the name of the file that contains the script (but without the `.py` extension).

Next, we load the mesh file, and set the initial (normalised) magnetisation to point in the `[1,0,1]` direction, i.e. to have equal magnitude in the x- and z-direction and 0 in the y-direction.

```
sim.load_mesh("bar30_30_100.nmesh.h5",
              [("Py", mat_Py)],
              unit_length=SI(1e-9,"m"))

sim.set_m([1,0,1])
```

This vector will automatically be normalised within nmag, so that `[1,0,1]` is equivalent to the normalised vector `[0.70710678,0,0.70710678]`.

In this example, we would like to study a dynamic process and will ask nmag to compute the time development over a certain amount of time `dt`. The line:

```
dt = SI(5e-12, "s")
```

simply creates a *SI object* which represents our timescale.

We then have a Python `for`-loop in which `i` will take integer values ranging from 0 to 60 for subsequent iterations. All indented lines are the body of the for-loop. (In the Python programming language, scoping is expressed through indentation rather than braces or other types of parentheses. Text editors such as Emacs come with built-in support for properly indenting Python code [by pressing the `Tab` key on a line to be indented].)

```
for i in range(0, 61):
    sim.advance_time(dt*i)
```

```
    if i % 10 == 0:
        sim.save_data(fields='all')
    else:
        sim.save_data()
```

In each iteration, we first call `sim.advance_time(i*dt)` which instructs nmag to carry on time integration up to the time `i*dt`.

The call to *save_data* will save the average data into the `bar_dat.ndt` file.

The last four lines contain an `if` statement which is used to save spatially resolved data every ten time steps only, and averaged data every time step. The percent operator `%` computes `i` modulo 10. This will be 0 when `i` takes values 0, 10, 20, 30, ... In this case, we call:

```
sim.save_data(fields='all')
```

which will save the (spatial) averages of all *fields* (going into the `bar_dat.ndt` file), *and* the spatially resolved data for all fields (that are saved to `bar_dat.h5`).

If `i` is not an integer multiple of 10, then the command:

```
sim.save_data()
```

is called, which only saves spatially averaged data.

2.2.3 Analysing the data

Time dependent averages

We first plot the average magnetisation vector against time. To see what data is available, we call *ncol* with just the name of the simulation (which here is `bar`):

```
$ ncol bar
 0:          #time            #<s>              0
 1:             id              <>              1
 2:           step              <>              0
 3:     stage_time             <s>              0
 4:     stage_step              <>              0
 5:          stage              <>              0
 6:     E_total_Py      <kg/ms^2>  -0.2603465789714
 7:            phi             <A>  0.0002507410390772
 8:       E_ext_Py      <kg/ms^2>              0
 9:      H_demag_0          <A/m>  -263661.6680783
10:      H_demag_1          <A/m>  -8.218106743355
11:      H_demag_2          <A/m>  -77027.641984
12:      dmdt_Py_0         <A/ms>  -8.250904652583e+15
13:      dmdt_Py_1         <A/ms>  2.333344983225e+16
14:      dmdt_Py_2         <A/ms>  8.250904652583e+15
15:    H_anis_Py_0          <A/m>              0
16:    H_anis_Py_1          <A/m>              0
17:    H_anis_Py_2          <A/m>              0
18:         m_Py_0             <>  0.7071067811865
19:         m_Py_1             <>              0
20:         m_Py_2             <>  0.7071067811865
21:         M_Py_0          <A/m>  608111.8318204
22:         M_Py_1          <A/m>              0
23:         M_Py_2          <A/m>  608111.8318204
24:      E_anis_Py      <kg/ms^2>              0
25:      E_exch_Py      <kg/ms^2>  5.046530179037e-17
26:            rho        <A/m^2>  0.03469702141876
27:        H_ext_0          <A/m>              0
28:        H_ext_1          <A/m>              0
29:        H_ext_2          <A/m>              0
```

```
30:  H_total_Py_0          <A/m>  -263661.6680783
31:  H_total_Py_1          <A/m>  -8.218106743352
32:  H_total_Py_2          <A/m>  -77027.641984
33:    E_demag_Py      <kg/ms^2>  -0.2603465789714
34:   H_exch_Py_0          <A/m>  3.301942533099e-11
35:   H_exch_Py_1          <A/m>                  0
36:   H_exch_Py_2          <A/m>  3.301942533099e-11
37: maxangle_m_Py          <deg>                  0
38:     localtime             <>  2007/08/15-11:16:19
39:      unixtime            <s>   1187172979.6
```

The meaning of the various entries is discussed in detail in section *ncol*. Here, we simply note that the column indices (given by the number at the beginning of every line) we are most interested in are:

- 0 for the `time`,
- 21 for `M_Py_0` which is the x-component of the magnetisation of the Py material,
- 22 for `M_Py_1` which is the y-component of the magnetisation of the Py material, and
- 23 for `M_Py_2` which is the z-component of the magnetisation of the Py material,

We can use *ncol* to extract this data into a file `data_M.dat` which has the time for each time step in the first column and the x, y and z component of the magnetisation in columns 2, 3 and 4, respectively:

```
$ ncol bar 0 21 22 23 > data_M.txt
```

This creates a text file `data_M.txt` that can be read by other applications to create a plot.

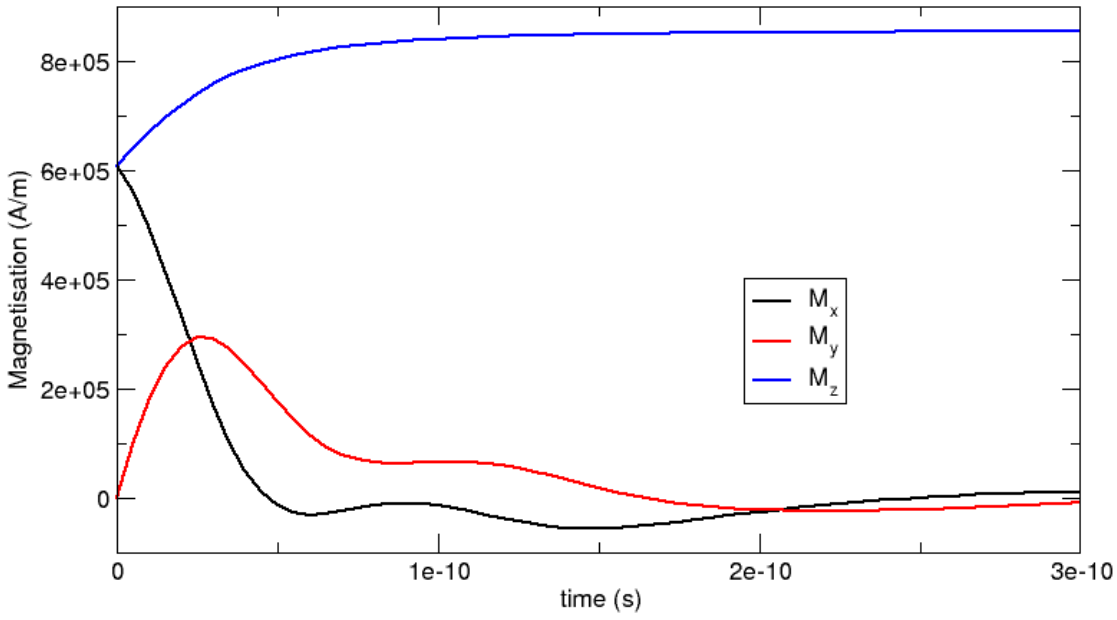
Note, however, that the order of the entries in the ndt file is not guaranteed, i.e. the numbers corresponding to fields may change with different versions of the software, or different simulations (for example, the user may add extra fields). Therefore, the recommended approach is to directly specify the names of the columns that are to be extracted (i.e. `time M_Py_0 M_Py_1 M_Py_2`):

```
$ ncol bar time M_Py_0 M_Py_1 M_Py_2 > data_M.txt
```

We use the xmgrace command:

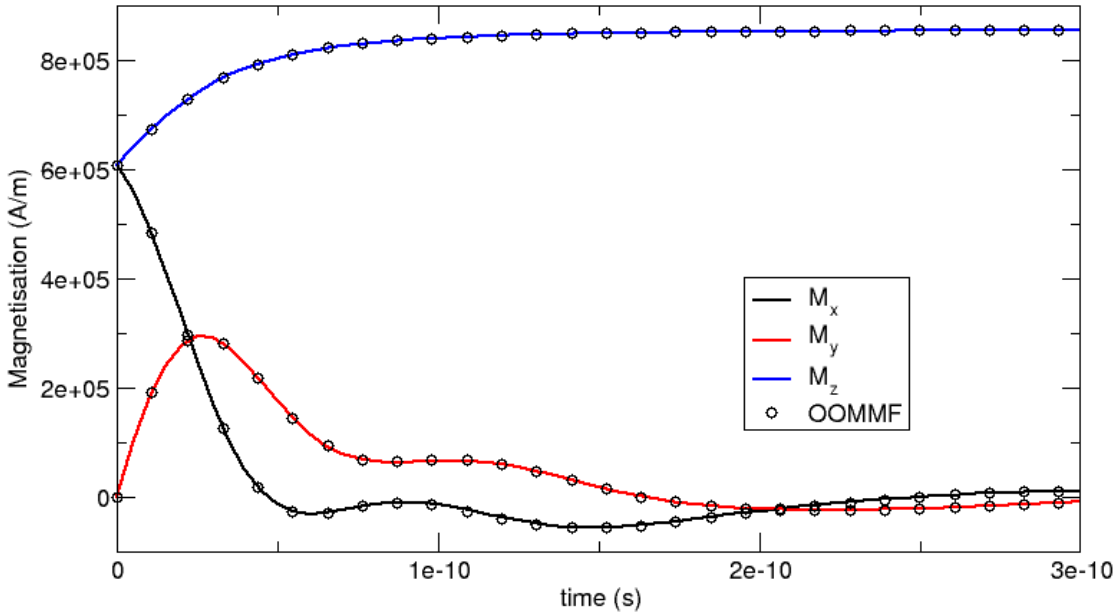
```
xmgrace -nxy data_M.txt
```

to create the following plot (manually adding the legend and axis labels):



Comparison with OOMMF and Magpar

We have carried out the same simulation with Magpar and OOMMF. The following plot shows the corresponding OOMMF-curves (as spheres) together with nmag's results. (The Magpar curve, which is not shown here, follows the nmag data very closely.)



Spatially resolved fields

The command `sim.save_data(fields='all')` saves all *fields* into the file `bar_dat.h5` (as explained, the filename is composed of the name of the simulation [here `bar`] and the extension `_dat.h5`). The code `bar30_30_100.py` above calls the *save_data* command every 10 iterations. As every `dt` corresponds to 0.5 picoseconds, the data hence is saved every 5 picoseconds.

We can confirm this by using the *nmagpp* command:

```
$ nmagpp --idlist bar
```

which produces the following output:

```
 id   stage   step      time  fields
 0->    1      0         0 E_anis E_demag E_exch E_ext E_total H_anis H_demag  ... phi pin rho
10->    1    312     5e-11 E_anis E_demag E_exch E_ext E_total H_anis H_demag  ... phi pin rho
20->    1    495     1e-10 E_anis E_demag E_exch E_ext E_total H_anis H_demag  ... phi pin rho
30->    1    603   1.5e-10 E_anis E_demag E_exch E_ext E_total H_anis H_demag  ... phi pin rho
40->    1    678     2e-10 E_anis E_demag E_exch E_ext E_total H_anis H_demag  ... phi pin rho
50->    1    726   2.5e-10 E_anis E_demag E_exch E_ext E_total H_anis H_demag  ... phi pin rho
60->    1    762     3e-10 E_anis E_demag E_exch E_ext E_total H_anis H_demag  ... phi pin rho
```

The first column is a *unique identifier id* for a configuration of the system. We can use the `--range` argument to select entries for further processing. The `stage` is only relevant for calculations of hysteresis curves (see *Example: Simple hysteresis loop*). The `step` is the time-stepper iteration counter for this calculation. The time is given in seconds (<s>). (Note the 5 pico-second interval between entries.) The stage, step and time data is provided for convenience. What follows is a list of fields that have been saved for each of these configurations.

We convert the first saved time step into a vtk file with base name `bar_initial.vtk` using

```
$ nmagpp --range 0 --vtk bar_initial.vtk bar
```

and we also convert the last saved time step at 300 picoseconds to a vtk file with base name `bar_final.vtk` using:

```
$ nmagpp --range 60 --vtk bar_final.vtk bar
```

The actual file names that are created by these two commands are `bar_initial-000000.vtk` and `bar_final-000060.vtk`. The appended number is the `id` of the saved configuration. This is useful if one wants to create vtk files for all saved configurations. For example:

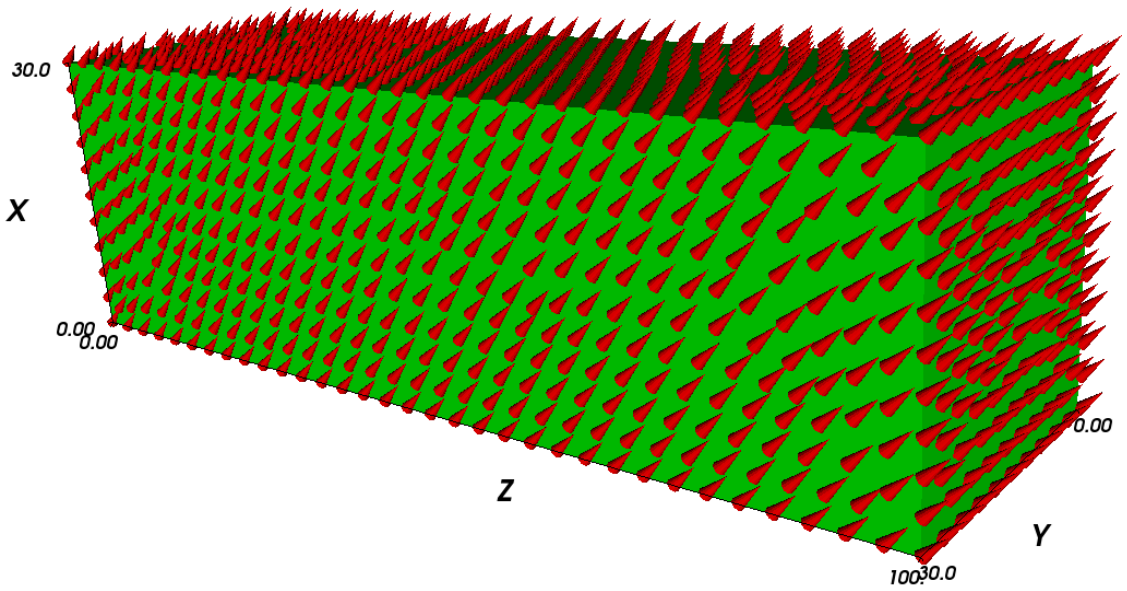
```
$ nmagpp --vtk bar.vtk bar
```

will create the files:

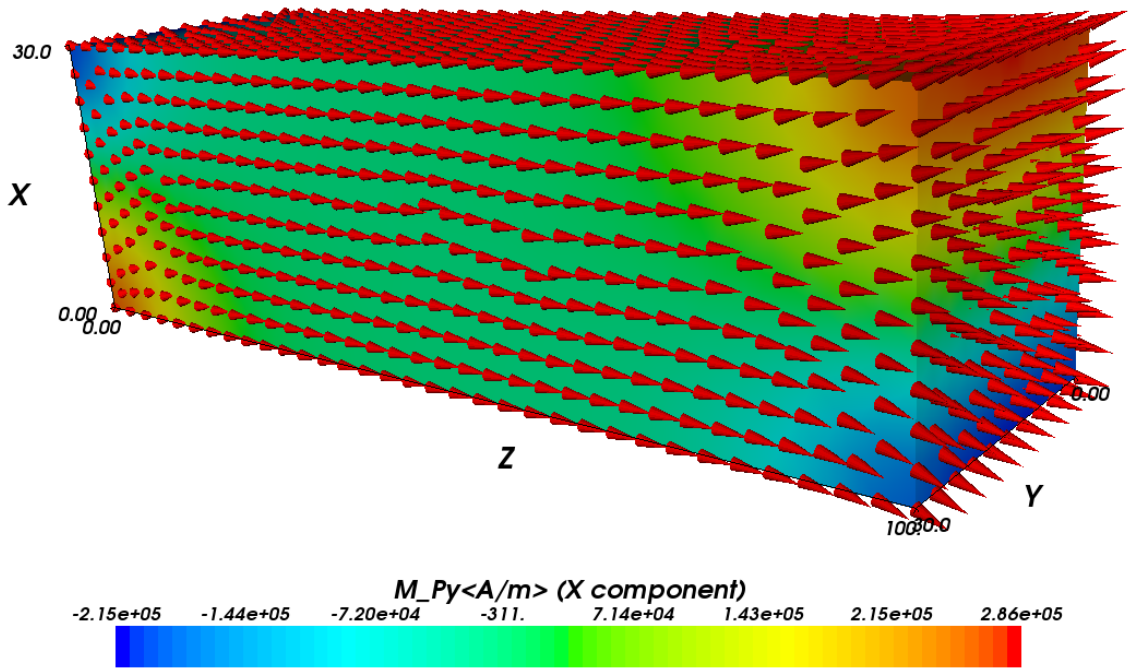
```
bar-000000.vtk
bar-000010.vtk
bar-000020.vtk
bar-000030.vtk
bar-000040.vtk
bar-000050.vtk
bar-000060.vtk
```

Using *MayaVi*, we can display this data in a variety of ways. Remember that all field values are shown in SI units by default (see *nmagpp*), and positions are as provided in the mesh file. In this case, positions are expressed in nanometers (this comes from the `unit_length=SI(1e-9,"m")` expression in the `sim.load_mesh()` command.

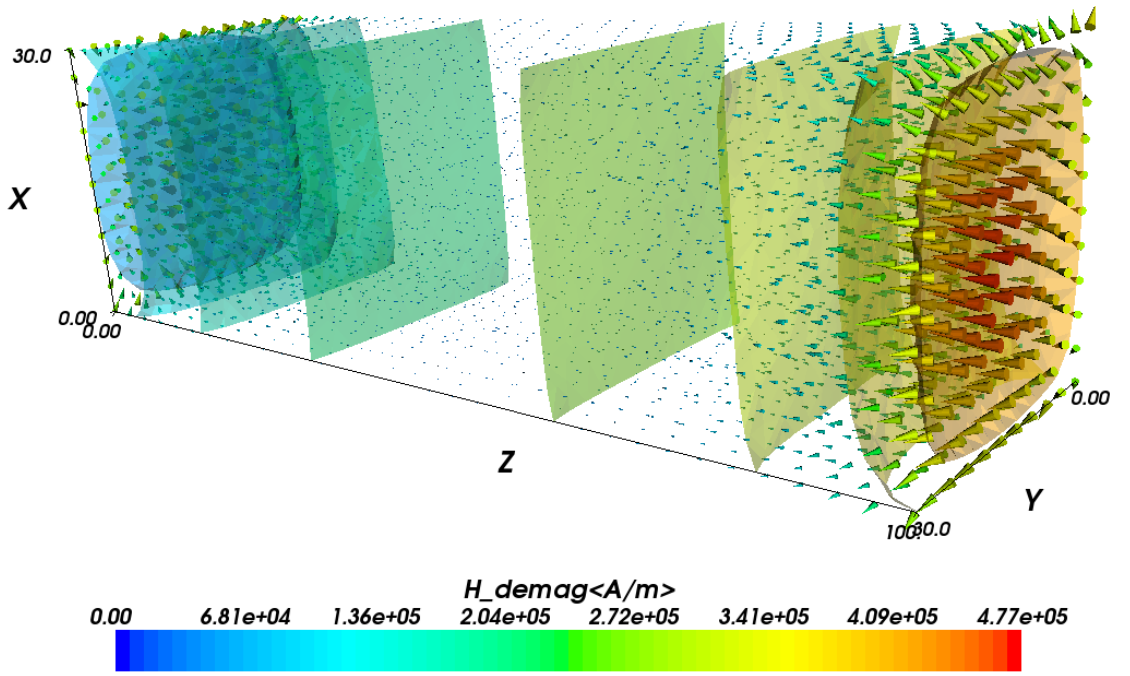
This is the initial configuration with magnetisation pointing in the [1,0,1] direction:



The "final" configuration shows that the magnetisation aligns along the z-direction. The coloured surface shows the x-component of the magnetisation (and the colorbar provides the scale). It can be seen that the magnetisation at position z=100 nm goes into a flower state to minimise the overall energy. (Note that, strictly speaking, this system is not yet in a meta-stable state after 300 ps – but already quite close.):



Because we have saved all fields (not just the magnetisation), we can also study other properties. For example, the following image shows the demagnetisation field as vectors (and the legend refers to the magnitude of the vectors), as well as the magnetic scalar potential (as a stack of isosurfaces). Because the demagnetisation field is the (negative) gradient of the scalar potential, the vectors are perpendicular on the isosurfaces:



2.2.4 Higher level functions

We now have seen an overview over the fundamental commands used to set up a micromagnetic simulation and demonstrate how to advance the configuration of the system through time. In principle, this is all one would need

to know to compute hysteresis loops and carry out most micromagnetic computations. However, there are more advanced functions that simplify and automatise the most frequent tasks, such as computing a hysteresis loop.

2.2.5 "Relaxing" the system

The *relax* command takes the current magnetisation configuration of a simulation and computes the time development until the torque on each mesh site is smaller than a certain threshold. This is useful for this particular example as we do not know for how long we need to integrate the system until it stops in a local energy minimum configuration. We can adjust the code of this example to make use of the `relax` command (`modified source code`):

```
import nmag
from nmag import SI, every, at

mat_Py = nmag.MagMaterial( name="Py",
                           Ms=SI(0.86e6, "A/m"),
                           exchange_coupling=SI(13.0e-12, "J/m"),
                           llg_damping=0.5)


sim = nmag.Simulation("bar_relax")

sim.load_mesh("bar30_30_100.nmesh.h5", [("Py", mat_Py)],
              unit_length=SI(1e-9, "m"))

sim.set_m([1, 0, 1])

ps = SI(1e-12,"s")
sim.relax(save = [('averages', every('time', 5*ps)),
                  ('fields', at('convergence'))])
```

(Note the additions to the `import` statement!)

The particular `relax` command employed here:

```
sim.relax(save = [('averages', every('time',5*ps)),
                  ('fields', at('convergence'))])
```

works as follows:

The argument `save = [ ]` tells `relax` to save data according to the instructions given in the form of a python list (i.e. enclosed by square brackets). The first relax instruction is this tuple:

```
('averages', every('time',5*ps)
```

and means that the *averages* should be saved *every 5 picoseconds*. The syntax used here breaks down into the following parts:

- `'averages'` is just the keyword (a string) to say that the average data should be saved.
- `every(...)` is a special object which takes two parameters. They are here:
  - `'time'` to indicate that something should be done every time a certain amount of simulated time has passed, and
  - `5*ps` which is the amount of time after which the data should be saved again. This has to be a *SI object*, which we here obtain by multiplying a number (`5`) with the SI object `ps` which has been defined earlier in our example program to represent a pico-second.
  - We can provide further keywords to the `every` object (for example to save the data every `10` iteration steps we can use `every('step', 10)`).

Internally, the *relax* command uses the *hysteresis* command, so the documentation of *hysteresis* should be consulted for a more detailed explanation of parameters.

The second relax instruction is:

```
('fields', at('convergence'))
```

which means that the *fields* should be saved *at convergence*, i.e. when the relaxation process has finished and the magnetisation has converged to its (meta)stable configuration:

- `'fields'` is a string that indicates that we would like to save all the defined fields.
- `at('convergence')` is a special object that indicates that this should happen exactly when the relaxation process has converged.

After running this program, we can use the *ncol* tool to look at the averages saved:

```
$ ncol bar_relax step time
```

gives output which starts like this:

```
  0              0
 82          5e-12
120          1e-11
146        1.5e-11
176          2e-11
201        2.5e-11
227          3e-11
248        3.5e-11
```

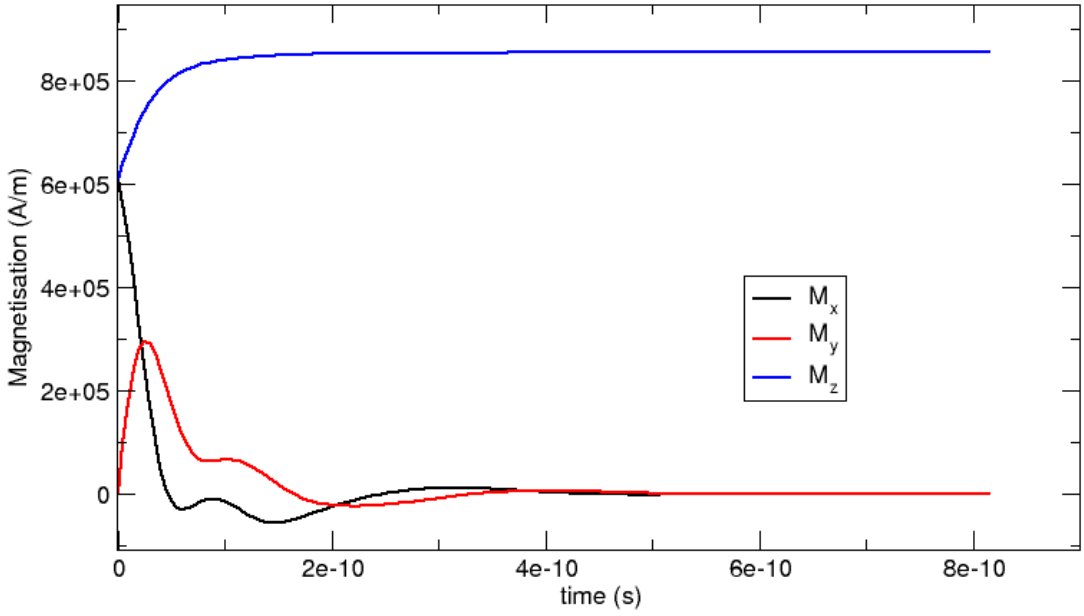
Here, we see the iterations on the left and the simulated time (in seconds) on the right. As requested, there is one data entry (i.e. line) every 5 picoseconds.

Note that it may happen that the system saves the data not exactly at the requested time, i.e.:

```
532        6.5e-11
580  7.047908066945e-11
620        7.5e-11
```

The middle line shows that the data has been saved when the simulated time was approximately 7.05e-11 seconds whereas we requested 7e-11 seconds. Such small deviations are tolerated by the system to improve performance [1].

From the data saved, we can obtain the following plot:



[1] The time integrator (here, Sundials CVODE) would have to do an extra step to get to the requested time. If the current time is very close to the requested time, it will simply report this value.

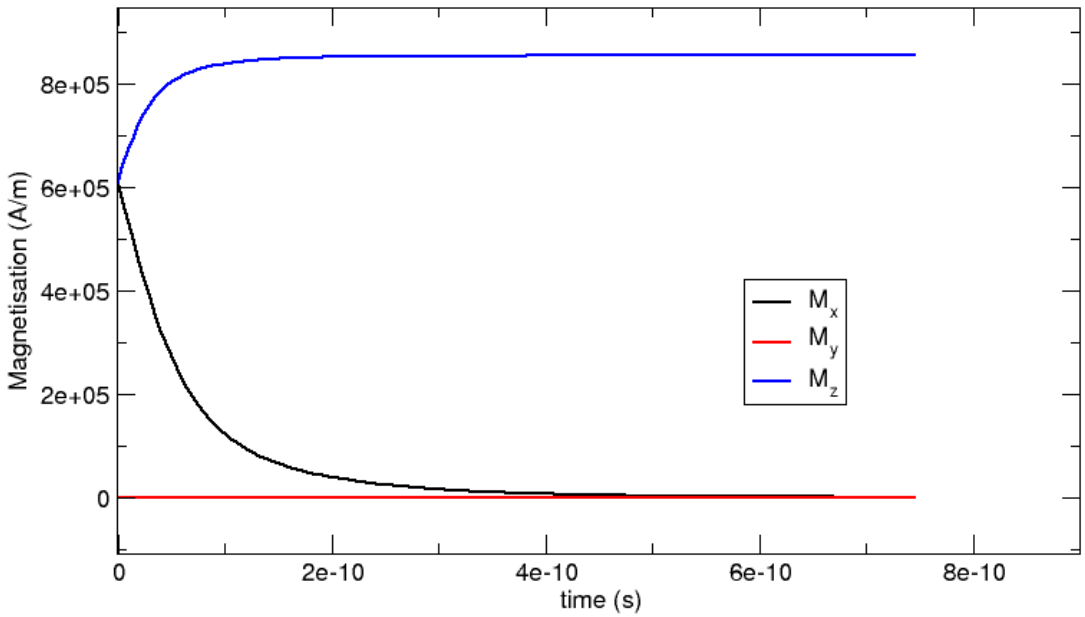
In summary, the *relax* function is useful to obtain a meta-stable configuration of the system. In particular, it will carry out the time integration until the remaining torque at any point in the system has dropped below a certain threshold.

2.2.6 "Relaxing" the system faster

If we are only interested in the final (meta-stable) configuration of a run, we can switch off the precession term in the Laundau Lifshitz and Gilbert equation. The MagMaterial definition in the following example shows how to do this:

```
import nmag
from nmag import SI, every, at

mat_Py = nmag.MagMaterial( name="Py",
                           Ms=SI(0.86e6, "A/m"),
                           exchange_coupling=SI(13.0e-12, "J/m"),
                           llg_damping=0.5,
                           do_precession=False )


sim = nmag.Simulation("bar_relax2")

sim.load_mesh("bar30_30_100.nmesh.h5", [("Py", mat_Py)],
              unit_length=SI(1e-9, "m"))

sim.set_m([1, 0, 1])

ps = SI(1e-12,"s")
sim.relax(save = [('averages', every('time', 5*ps)),
                  ('fields', at('convergence'))])
```

The new option is `do_precession=False` in the constructor of the PermAlloy material `mat_Py`. As a result, there will be no precession term in the equation of motion:



While the time-development of the system happens at the same time scale as for the system with the precession term (see "Relaxing" the system), the computation of the system without the precession is significantly faster (for this example, we needed about 3500 iterations with the precession term and 1500 without it, and the computation time scales similarly).

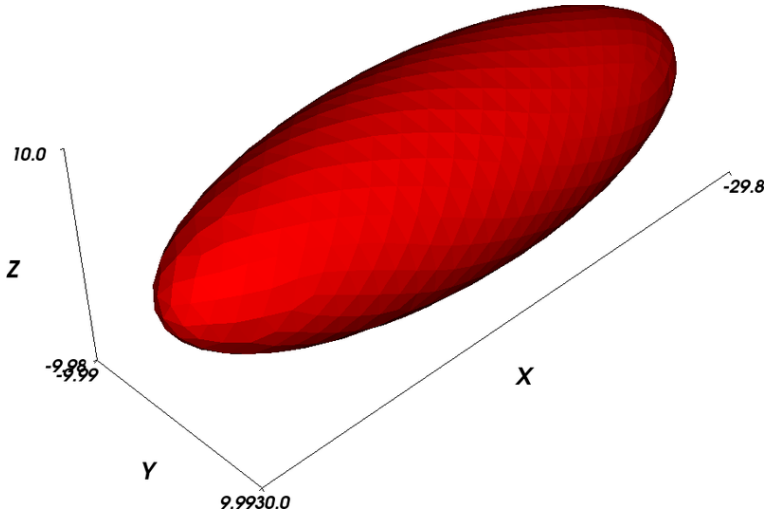
Note, that the ''dynamics" shown here are of course artificial and only used to obtain a meta-stable physical configuration more efficiently!

2.2.7 Decreasing execution time

Note that the execution time can generally be reduced significantly by decreasing the tolerances for the time integrator. In short, one has to use the *set_params* function (after *set_m* has been called). Decreasing the requested accuracy will of course make the simulation results less accurate but this is often acceptable. An example of how to use the *set_m* function and detailed discussion of the micromagnetic example shown in this section for a variety of tolerance values is given in the section *Example timestepper tolerances*.

2.3 Example: Simple hysteresis loop

This example computes the hysteresis loop of an ellipsoidal magnetic object. We use an ellipsoid whose x,y,z semi-axes have lengths 30 nm, 10 nm and 10 nm, respectively. (The mesh is contained in `ellipsoid.nmesh.h5` and produced with Netgen from `ellipsoid.geo`):



This picture has been obtained by converting the mesh to a *vtk* file using:

```
$ nmeshpp --vtk ellipsoid.nmesh.h5 mesh.vtk
```

and subsequent visualisation with *MayaVi*:

```
$ mayavi -d mesh.vtk -m SurfaceMap
```

We have further added the axes within MayaVi (Visualize->Modules->Axes), and changed the display color from blue to red (Double click on `SurfaceMap` in the selected Modules list, then uncheck the `Scalar Coloring` box, click on `Change Object Color` and select a suitable color).

We provide the mayavi file `mesh.mv` that shows the visulisation as in the figure above. (If you want to load this file into MayaVi, just use `$ mayavi mesh.mv` but make sure that `mesh.vtk` is in the same directory as mayavi will need to read this.)

2.3.1 Hysteresis simulation script

To compute the hysteresis loop for the ellipsoid, we use the script `ellipsoid.py`:

```
import nmag
from nmag import SI, at

#create simulation object
```

```
sim = nmag.Simulation()

# define magnetic material
Py = nmag.MagMaterial(name="Py",
                      Ms=SI(1e6,"A/m"),
                      exchange_coupling=SI(13.0e-12, "J/m"))

# load mesh: the mesh dimensions are scaled by 0.5 nm
sim.load_mesh("ellipsoid.nmesh.h5",
              [("ellipsoid", Py)],
              unit_length=SI(1e-9,"m"))

# set initial magnetisation
sim.set_m([1.,0.,0.])

Hs = nmag.vector_set(direction=[1.,0.01,0],
                     norm_list=[ 1.00,  0.95, [], -1.00,
                                -0.95, -0.90, [],  1.00],
                     units=1e6*SI('A/m'))

# loop over the applied fields Hs
sim.hysteresis(Hs, save=[('restart','fields', at('convergence'))])
```

As in the previous examples, we first need to import the modules necessary for the simulation. `at('convergence')` allows us to save the fields and the averages whenever convergence is reached. We then define the material of the magnetic object, load the mesh and set the initial configuration of the magnetisation as well as the external field.

2.3.2 Hysteresis loop computation

We apply the external magnetic fields in the x-direction with range of 1e6 A/m down to -1e6 A/m in steps of `0.05e6 A/m`.

To convey this information efficiently to nmag, we use:

1. a direction for the applied field (here just `[1,0.01,0]`), (note that we have a small y-component of 1% in the applied field to break the symmetry)
2. a list of magnitudes of the field that will be multiplied with the direction vector,
3. another multiplier that defines the physical dimension of the applied fields (here `1000kA/m`, given as `1e6*SI('A/m')`).

Putting all this together, we obtain this command:

```
Hs = nmag.vector_set(direction=[1., 0.01, 0],
                     norm_list=[ 1.00,  0.95, [], -1.00,
                                -0.95, -0.90, [],  1.00],
                     units=1e6*SI('A/m'))
```

which computes a list of vectors `Hs`. Each entry in the list corresponds to one applied field.

The *hysteresis* command takes this list of applied fields `Hs` as one input parameter, and computes the hysteresis loop for these fields:

```
sim.hysteresis(Hs, save=[('restart', 'fields', at('convergence'))])
```

The `save` parameter is used to tell the hysteresis command what data to save, and how often. We have come across this notation when explaining the `relax` command in the section *"Relaxing" the system* of the previous example. In the example shown here, we request that the *fields* and the *restart* data should be saved *at* the point in time where we reach *convergence*. (The spatially averaged data is saved automatically to the *Data files (.ndt)* file when the *fields* are saved.) This is done in a compact notation shown above which is equivalent to this more explicit version:

```
sim.hysteresis(Hs,
               save=[('restart', at('convergence')),
                     ('fields',   at('convergence'))])
```

The compact notation can be used because here we want to save *fields* and *restart* data at the same time.

The *hysteresis* command computes the time development of the system for one applied field until a convergence criterion is met. It then proceeds to the next external field value provided in `Hs`.

We run the simulation as usual using:

```
$ nsim ellipsoid.py
```

If you have run the simulation before, we need to use the `--clean` switch to enforce overriding of existing data files:

```
$ nsim ellipsoid.py --clean
```

The simulation should take only a few minutes (for example 3 minutes on an Athlon64 3800+), and needs about 75MB of RAM.

If the simulation has been interrupted, it can be continued using

    $ nsim ellipsoid.py –restart

2.3.3 Obtaining the hysteresis loop data

Once the calculation has finished, we can plot the graph of the magnetisation (projected along the direction of the applied field) as a function of the applied field.

We use the *ncol* command to extract the data into a text file named `plot.dat`:

```
$ ncol ellipsoid H_ext_0 m_Py_0 > plot.dat
```

2.3.4 Plotting the hysteresis loop with Gnuplot

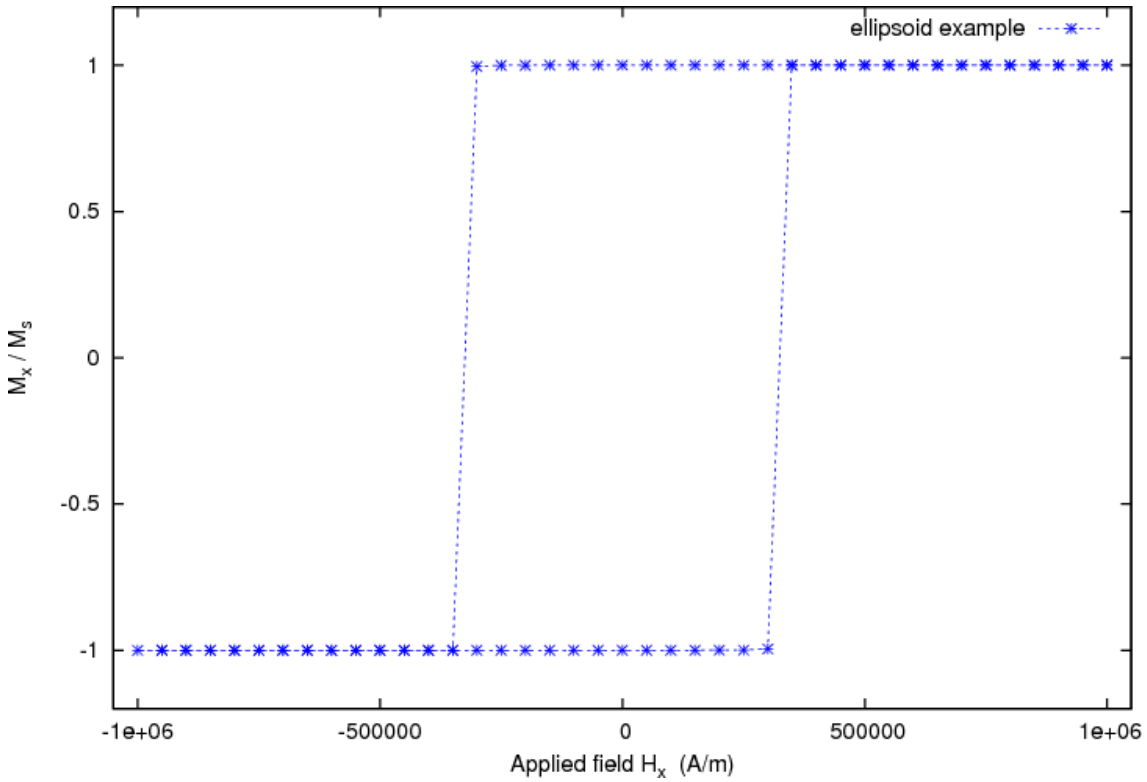
In this example, rather than using xmgrace, we show how to plot data using Gnuplot:

```
$ gnuplot make_plot.gnu
```

The contents of the gnuplot script `make_plot.gnu` are:

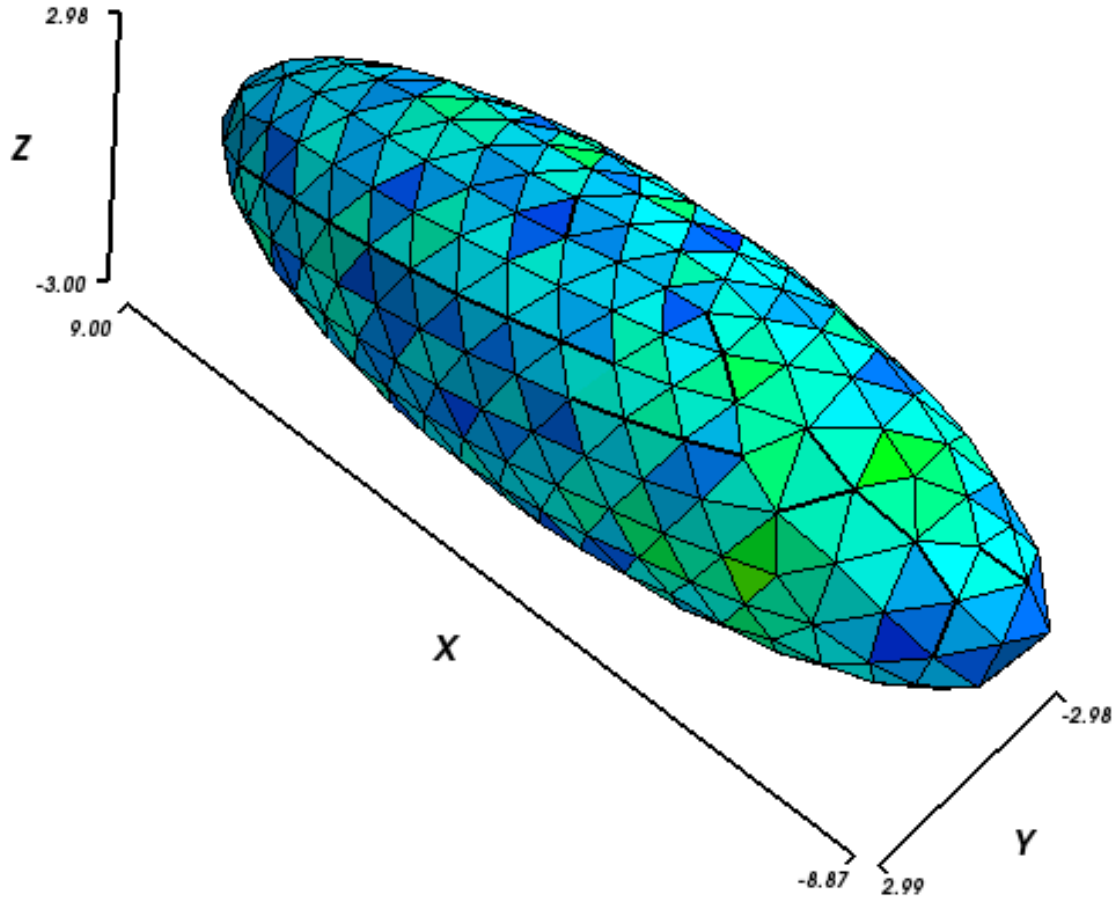
```
set term postscript eps enhanced color
set out 'hysteresis.eps'
set xlabel 'Applied field H_x   (A/m)'
set ylabel 'M_x / M_s'
set xrange [-1050000:1050000]
set yrange [-1.2:1.2]
plot 'plot.dat' u 1:2 ti 'ellipsoid example' w lp 3
```

which generates the following hysteresis loop graph:



2.4 Example: Hysteresis loop for Stoner-Wohlfarth particle

This example is very similar to *Example: Simple hysteresis loop* but computes the hysteresis loop of a smaller ellipsoidal magnetic object. This allows to compare the results with the analytical solution given by the Stoner-Wohlfarth model. We use an ellipsoid whose x,y,z semi-axes have lengths 9 nm, 3 nm and 3 nm, respectively. (The mesh is contained in `ellipsoid.nmesh.h5` and produced with Netgen from `ellipsoid.geo`):



To compute the hysteresis loop for the ellipsoid, we use the script `ellipsoid.py`:

```
import nmag
from nmag import SI, at

#create simulation object
sim = nmag.Simulation()

# define magnetic material
Py = nmag.MagMaterial(name="Py",
                      Ms=SI(1e6,"A/m"),
                      exchange_coupling=SI(13.0e-12, "J/m"))

# load mesh: the mesh dimensions are scaled by 0.5 nm
sim.load_mesh("ellipsoid.nmesh.h5",
              [("ellipsoid", Py)],
              unit_length=SI(1e-9,"m"))

# set initial magnetisation
sim.set_m([1.,1.,0.])

Hs = nmag.vector_set(direction=[1.,1.,0.],
                     norm_list=[1.0, 0.995, [], -1.0,
                                -0.995, -0.990, [], 1.0],
                     units=1e6*SI('A/m'))

# loop over the applied fields Hs
sim.hysteresis(Hs, save=[('averages', at('convergence'))])
```

We apply external magnetic fields in [110] direction (i.e. 45 degrees between the x and the y-axis) to this system, with strengths in the range of 1000 kA/m down to -1000 kA/m in steps of `5 kA/m`.

The `save` parameter is used to tell the hysteresis command what data to save, and how often. Here, we are only interested in saving the spatially averaged magnetisation values for every stage (i.e. meta-stable equilibrium before the applied field is changed).

2.4.1 Plotting the hysteresis loop

To extract the data needed for plotting the hysteresis loop we proceed as explained in the previous example *Example: Simple hysteresis loop*. We use the *ncol* command and extract the data into a text file named `plot.dat`:

```
$ ncol ellipsoid H_ext_0 H_ext_1 H_ext_2 m_Py_0 m_Py_1 m_Py_2 > plot.dat
```

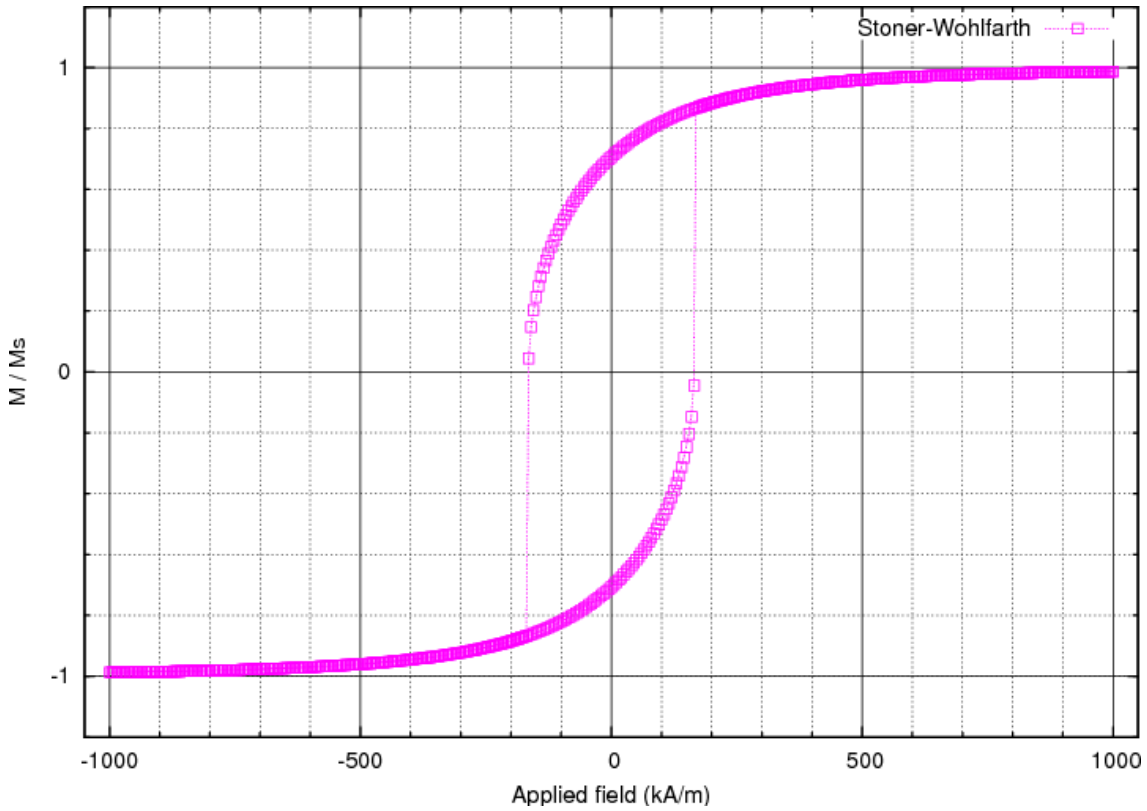
We then use Gnuplot to plot the loop:

```
$ gnuplot make_plot.gnu
```

The gnuplot script `make_plot.gnu` is:

```
set term postscript eps enhanced color
set out 'hysteresis.eps'
set xlabel 'Applied field (kA/m)'
set ylabel 'M / Ms'
versor_x = 1/sqrt(2)
versor_y = 1/sqrt(2)
versor_z = 0.0
scalar_prod(x1,x2,x3) = x1*versor_x + x2*versor_y + x3*versor_z

set mxtics 5            # minor tics and grid
set ytics 1
set mytics 5
set grid xtics ytics mxtics mytics lt -1 lw 0.5, lt 0
plot [-1050:1050] [-1.2:1.2] \
  'plot.dat' u (scalar_prod($1,$2,$3)/1000):(scalar_prod($4,$5,$6)) t 'Stoner-Wohlfarth' w lp 4
```

Note that within the gnuplot file, we project the magnetisation data in the `[1,1,0]` direction because the applied field was acting in this direction. We obtain this hysteresis loop:



The coercive field, which is located somewhere between 165 and 170 kA/m, can now be compared with the analytically known result for this particular system. To compute it, we need the demagnetizing factors Nx, Ny, Nz of the particle along the main axes. Since we deal with a prolate ellipsoid where two of the axes have the same dimension (y and z in this case), it is sufficient to compute the factor along the longest axis (x axis). The other two are easily derived from the relation Nx + Ny + Nz = 1. The expression to compute Nx is

$$N_x = \frac{1}{m^2-1} \cdot \left[\frac{m}{2\sqrt{m^2-1}} \cdot \ln\left(\frac{m+\sqrt{m^2-1}}{m-\sqrt{m^2-1}} \right) - 1 \right]$$

where we call the length of the x semi-axis `a`, the length of the y (or z) semi-axis `c`, and take `m` to be the ratio `m = a/c`. Here, the value of Nx is therefore 0.1087, so we have Ny = Nz = 0.4456. With these values the shape anisotropy is easily computed according to the expression:

$$H_a = M_s \cdot \Delta N = M_s \cdot (N_z - N_x)$$

This gives Ha = 337 kA/m in the case of Ms = 1000 kA/m. The final step is to compute the coercive field hc using this analytical (Stoner-Wohlfarth) result:

$$h_c = \frac{H_c}{H_a} = \sin\theta_0 \cdot \cos\theta_0$$

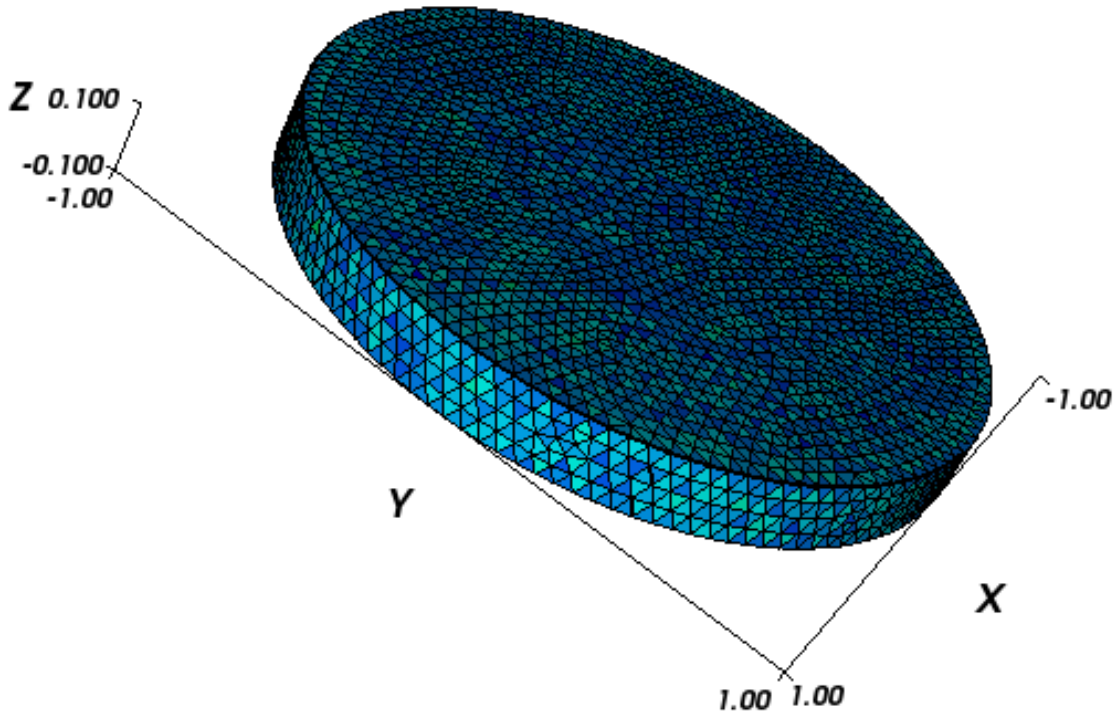
Here, theta_0 is the angle between the easy-axis of the particle (x-axis in our case) and the direction of the applied field. Substituting theta_0 = 45 (degrees) in the formula, we obtain hc = 0.5, that is Hc = 0.5 * Ha = 168 kA/m. As we have seen before, the simulated hysteresis loop gives a value between 165 and 170 kA/m, which is in agreement with the analytical solution.

Note that this simulation is relatively slow due to a number of constraints: to get good Stoner-Wolfarth behaviour, we need to describe the shape of the ellipsoid well, and thus need a small edgelength when we generate the mesh. We further need uniform behaviour of the magnetisation, which limits the overall size of the ellipsoid. A general property of micromagnetic simulations is that the associated differential equations get stiffer if the edge lengths (or more generally: distances between neighbouring degrees of freedom) become smaller. Stiffer systems of differential equations are harder to intergrate, and thus take more time.

2.5 Example: Hysteresis loop for thin disk

This example computes the hysteresis loop of a flat disc magnetised along a direction orthogonal to the main axis. In comparison to the previous *Example: Hysteresis loop for Stoner-Wohlfarth particle*, it demonstrates the use of a more complex sequence of applied fields.

We use a disc 20 nm thick and 200 nm in diameter for this example (the mesh is contained in `nanodot1.nmesh.h5` which is created from `the_nanodot.geo` with Netgen):



To compute the hysteresis loop for the disc, we use the script `nanodot1.py`:

```
import nmag
from nmag import SI, at

#create simulation object
sim = nmag.Simulation()

# define magnetic material
Py = nmag.MagMaterial( name="Py",
                       Ms=SI(795774,"A/m"),
                       exchange_coupling=SI(13.0e-12, "J/m")
                     )

# load mesh: the mesh dimensions are scaled by 100nm
sim.load_mesh( "nanodot1.nmesh.h5",
               [("cylinder", Py)],
               unit_length=SI(100e-9,"m")
             )

# set initial magnetisation
sim.set_m([1.,0.,0.])

Hs = nmag.vector_set( direction=[1.,0.,0.],
                      norm_list=[1000.0, 900.0, [],
                                 95.0, 90.0, [],
                                 -100.0, -200.0, [],
```

```
                                -1000.0, -900.0, [],
                                -95.0, -90.0, [],
                                100.0, 200.0, [], 1000.0],
                        units=1e3*SI('A/m')
                       )


# loop over the applied fields Hs
sim.hysteresis(Hs,
               save=[('averages', 'fields', 'restart', at('convergence'))]
               )
```

We assume that the previous example have been sufficiently instructive to explain the basic steps such as importing nmag, creating a simulation object, defining the material and leading the mesh. Here, we focus on the *hysteresis* command:

We would like to apply fields ranging from `[1e6, 0, 0] A/m` to `[100e3, 0, 0] A/m` in steps of `100e3 A/m`. Then, from `[95e3, 0, 0] A/m` to `[-95e3, 0, 0] A/m` we would like to use a smaller step size of `5e3 A/m` (to resolve this applied field range better).

This will take us through zero applied field (`[0,0,0] A/m`). Now, symmetrically to the positive field values, we would like to use a step size of `100e3 A/m` again to go from `[-100e3, 0, 0] A/m` to `[-1e6, 0, 0] A/m`. At this point, we would like to reverse the whole sequence (to sweep the field back to the initial value).

The information we need for the *hysteresis* command includes:

1. a direction for the applied field (here just `[1,0,0]`),
2. a list of magnitudes of the field (this is the `norm_list`) that will be interpreted, and then multiplied with the direction vector,

   As in the *Example: Simple hysteresis loop* and in the *Example: Hysteresis loop for Stoner-Wohlfarth particle*, we employ a special notation for ranges of field strengths understood by `nmag.vector_set`. The expression:

   ```
   [1000.0, 900.0, [], 95.0]
   ```

   means that we start with a magnitude of 1000, the next magnitude is 900. The empty brackets (`[]`) indicate that this sequence should be continued (i.e. 800, 700, 600, 500, 400, 300, 200, 100) up to but not beyond the next value given (i.e. 95).
3. another multiplier that defines the strength of the applied fields (here, `1e3*SI('A/m')`).

The corresponding command is:

```
Hs = nmag.vector_set( direction=[1,0,0],
                      norm_list=[1000.0, 900.0, [],
                                 95.0, 90.0, [],
                                 -100.0, -200.0, [],
                                 -1000.0, -900.0, [],
                                 -95.0, -90.0, [],
                                 100.0, 200.0, [], 1000.0],
                      units=1e6*SI('A/m')
                     )
```

which computes a list of vectors `Hs`. The *hysteresis* command takes this list of applied fields `Hs` as one input parameter, and will compute the hysteresis loop for these fields:

```
sim.hysteresis(Hs,
               save=[('averages', 'fields', 'restart', at('convergence'))]
              )
```

Again, the second parameter (`save`) is used to tell the hysteresis command what data to save, and how often. We request that the *averages* of the fields, the *fields* and the *restart* data should be saved *at* those points in time where we reach *convergence*. (See also *Restart example*).

2.5.1 Thin disk hysteresis loop

Once the calculation has finished, we can plot the hysteresis loop, i.e. the graph of the magnetisation computed along the direction of the applied field as a function of the applied field strength.

We use the *ncol* command to extract the data into a text file `plot.dat`:

```
$ ncol nanodot1 H_ext_0 m_Py_0 > plot.dat
```

This file starts as follows:

```
1000000  0.9995058139817
1000000  0.9995058139817
 900000  0.9994226410102
 900000  0.9994226410102
 800000  0.9993139080655
```

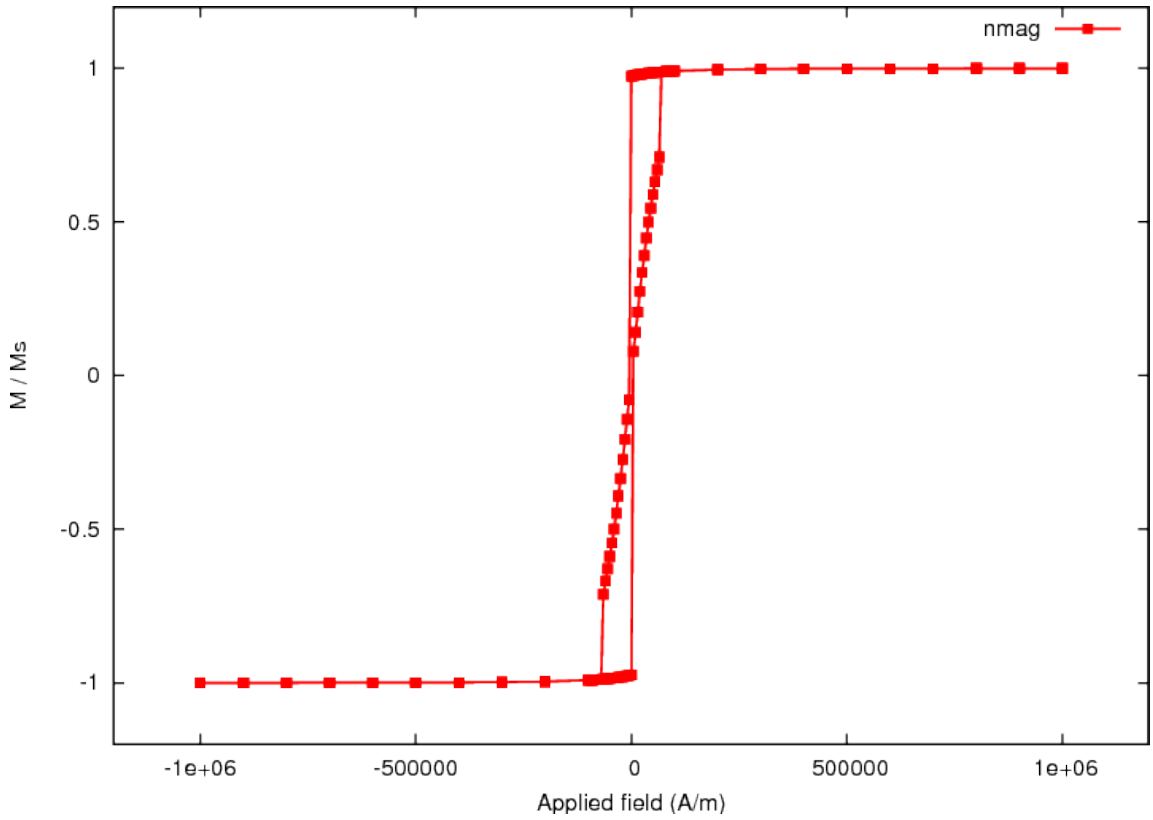
We use Gnuplot to plot the hysteresis loop:

```
$ gnuplot make_plot.gnu
```

using the gnuplot script `make_plot.gnu`:

```
set term postscript eps enhanced color
set out 'nanodot_hyst.eps'
set xlabel 'Applied field (A/m)'
set ylabel 'M / Ms'
set xrange [-1.2e6:1.2e6]
set yrange [-1.2:1.2]
plot 'plot.dat' u 1:2 ti 'nmag' with linespoints lw 3 pt 5
```

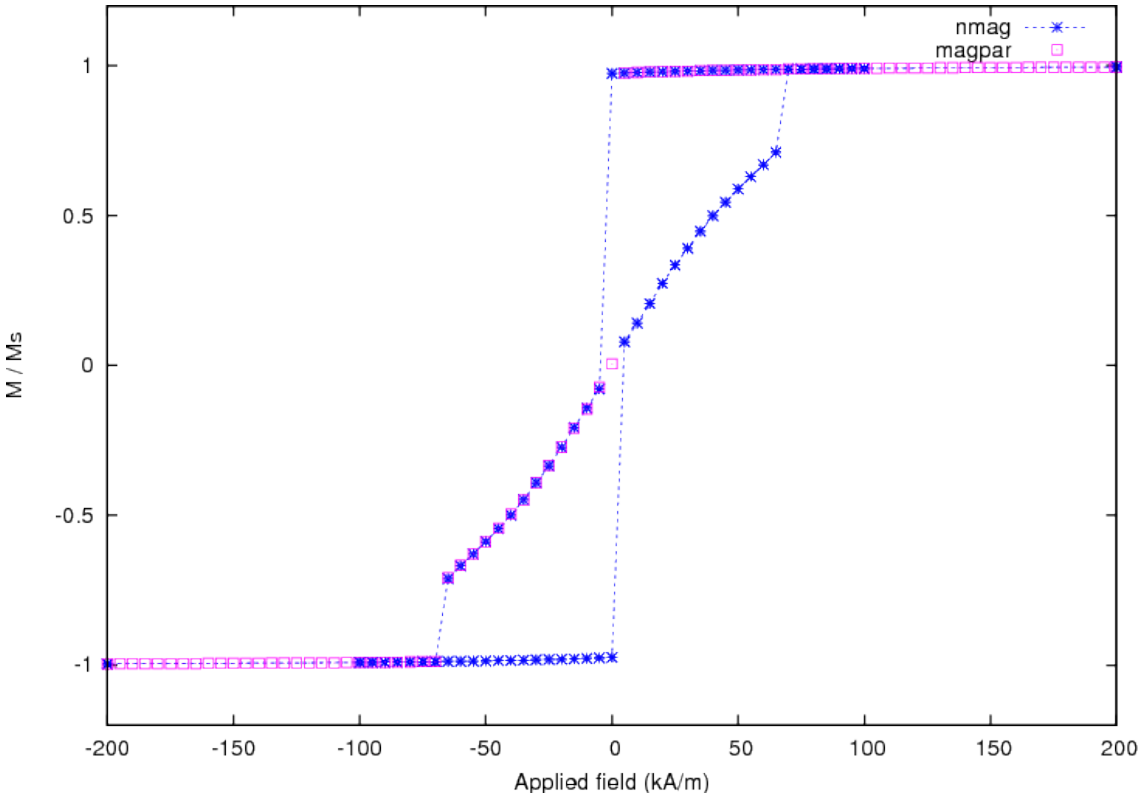
The resulting graph is:



and the comparison with the Magpar data, obtained with the script `make_comparison_plot.gnu`:

```
set term postscript eps enhanced color
set out 'nanodot_comparison_hyst.eps'
set xlabel 'Applied field (kA/m)'
set ylabel 'M / Ms'
set xrange [-0.2e3:0.2e3]
set yrange [-1.2:1.2]
plot 'plot.dat' u ($1/1000):2  ti 'nmag' w lp 3, 'magpar.dat' u 1:2  ti 'magpar' w p 4
```

is shown here (note that the Magpar computation only shows half of the hysteresis loop.):

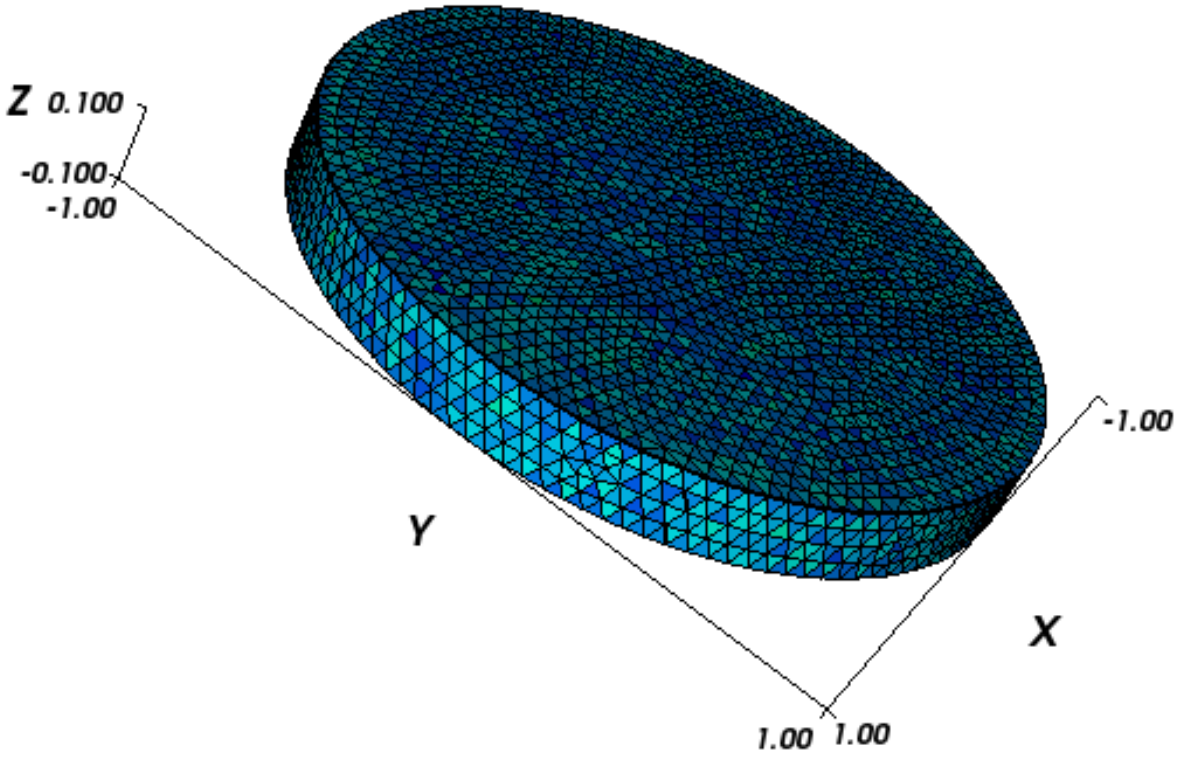


Here we can see a slight difference between nmag and Magpar in the location of the switching point, probably due to different tolerances in both programs when determining time integrator convergence.

2.6 Example: Vortex formation and propagation in disk

This example computes the evolution of a vortex in a flat cylinder magnetised along a direction orthogonal to the main axis.

We use the same geometry as in the *Hysteresis loop for thin disk example*: a flat cylinder, 20 nm thick and 200 nm in diameter (the mesh is contained in `nanodot.nmesh.h5`):



To simulate the magnetised disc, we use the following script (`nanodot.py`):

```
import nmag
from nmag import SI, at

#create simulation object
sim = nmag.Simulation()

# define magnetic material
Py = nmag.MagMaterial( name="Py",
                       Ms=SI(795774,"A/m"),
                       exchange_coupling=SI(13.0e-12, "J/m")
                     )

# load mesh: the mesh dimensions are scaled by 100nm
sim.load_mesh( "../example_vortex/nanodot.nmesh.h5",
               [("cylinder", Py)],
               unit_length=SI(100e-9,"m")
             )

# set initial magnetisation
sim.set_m([1.,0.,0.])

Hs = nmag.vector_set( direction=[1.,0.,0.],
                      norm_list=[12.0, 7.0, [], -200.0],
                      units=1e3*SI('A/m')
                    )


# loop over the applied fields Hs
sim.hysteresis(Hs,
               save=[('averages', at('convergence')),
                     ('fields',   at('convergence')),
                     ('restart',  at('convergence'))
                    ]
```

```
                )
```

We would like to compute the magnetisation behaviour in the applied fields ranging from `[12e3, 0, 0] A/m` to `[-200e3, 0, 0] A/m` in steps of `-5e3 A/m`. The command for this is:

```
Hs = nmag.vector_set( direction=[1,0,0],
                      norm_list=[12.0, 7.0, [], -200.0],
                      units=1e3*SI('A/m')
                    )

sim.hysteresis(Hs,
               save=[('averages', at('convergence')),
                     ('fields',   at('convergence')),
                     ('restart',  at('convergence'))
                    ]
              )
```

The *ncol* command allows us to extract the data and we redirect it to a text file with name `nmag.dat`:

```
$ ncol nanodot H_ext_0 m_Py_0 > nmag.dat
```

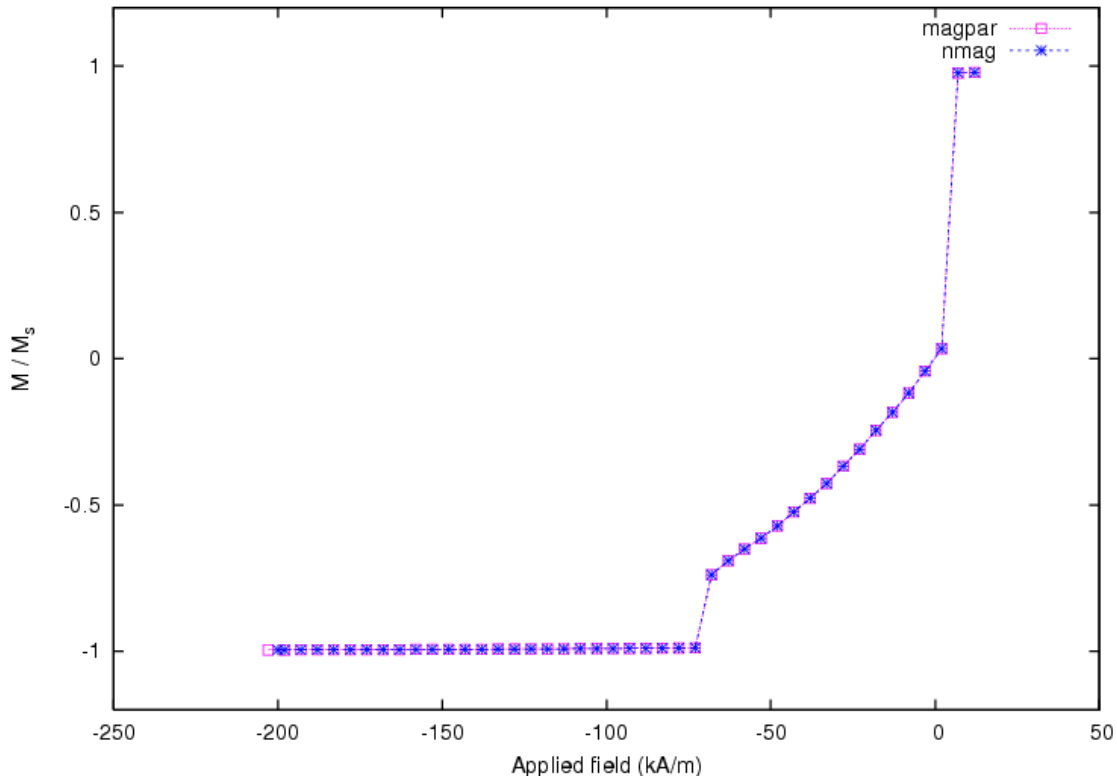
Plotting the data with Gnuplot:

```
$ gnuplot make_comparison_plot.gnu
```

which uses the script in `make_comparison_plot.gnu`:

```
set term postscript eps enhanced color
set out 'nanodot_evo.eps'
set xlabel 'Applied field (kA/m)'
set ylabel 'M / M_s'
plot [-250:50] [-1.2:1.2] 'magpar.dat' u 2:3 ti 'magpar' w lp 4 , 'nmag.dat' u ($1/1000):2  ti 'nm
```

The resulting graph is shown here:



and we can see that the results from *nsim* match those from magpar. The magnetisation configurations during the switching process are shown in the following snapshots:

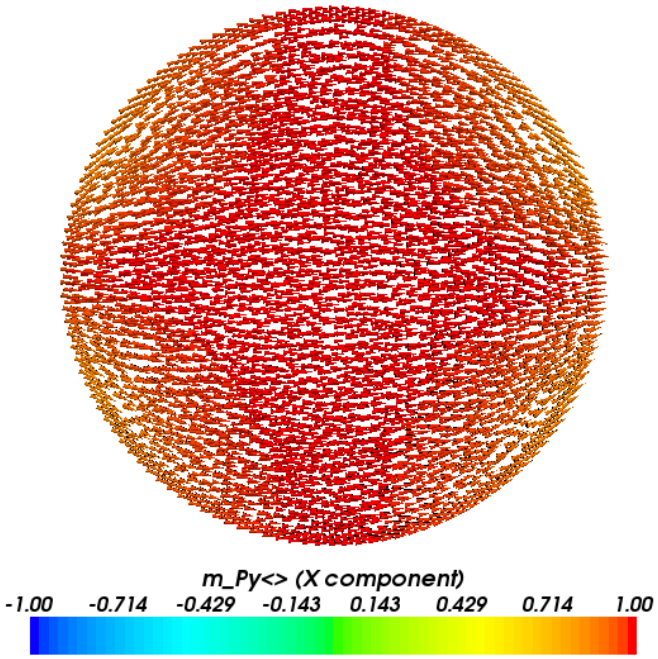


Figure 2.1: Magnetisation configuration for a decreasing applied field of 20 kA/m. The x-axis is increasing from left to right for this and the subsequent plots.

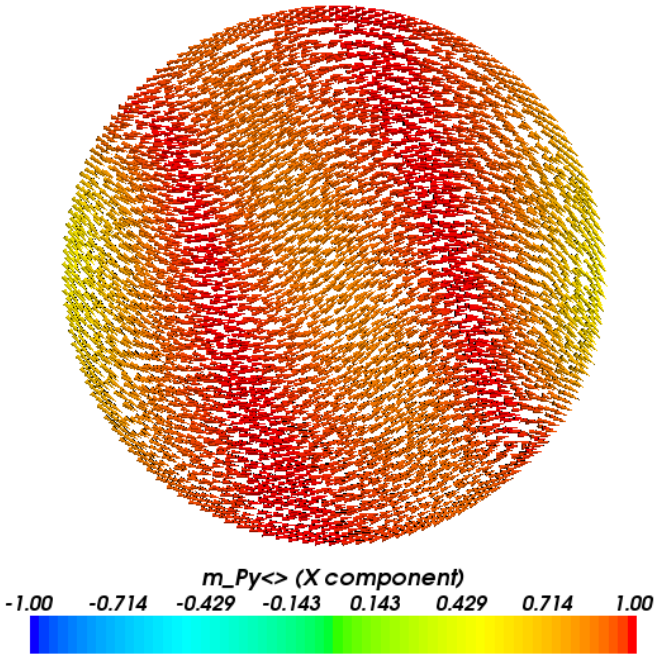


Figure 2.2: Magnetisation configuration for a decreasing applied field of 15 kA/m.

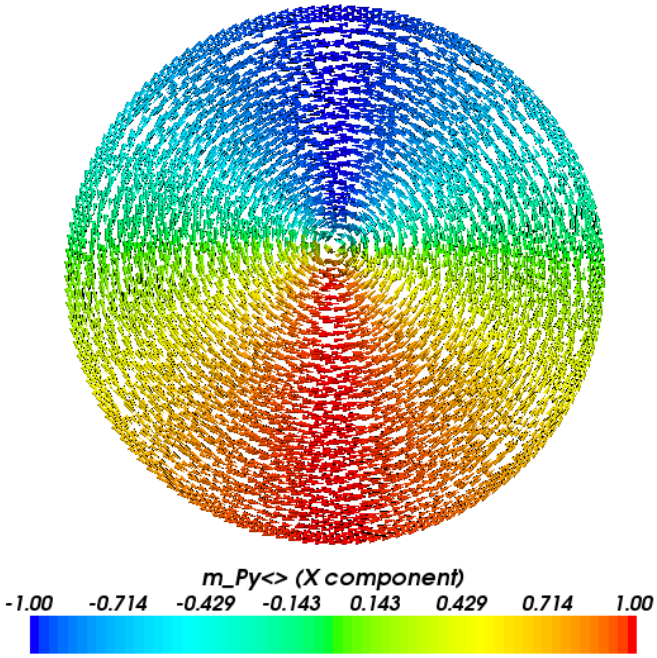


Figure 2.3: Magnetisation configuration for a decreasing applied field of 10 kA/m.

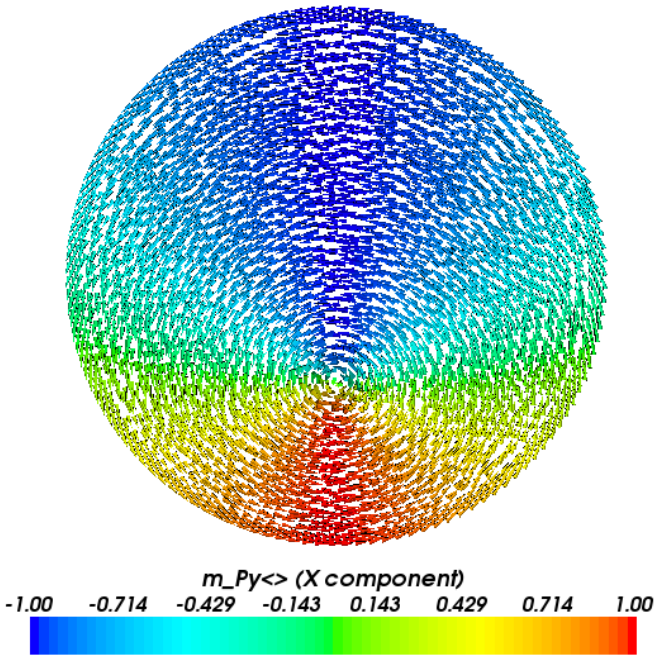


Figure 2.4: Magnetisation configuration for a decreasing applied field of -30 kA/m.

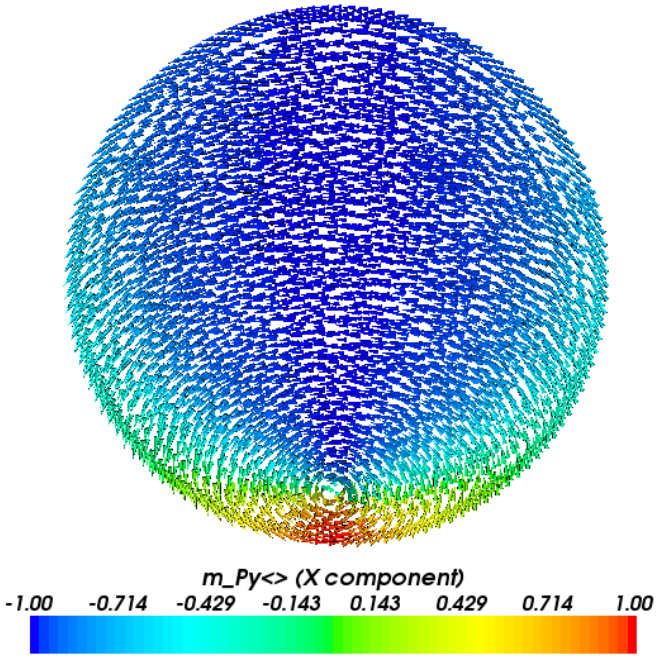


Figure 2.5: Magnetisation configuration for a decreasing applied field of -95 kA/m.

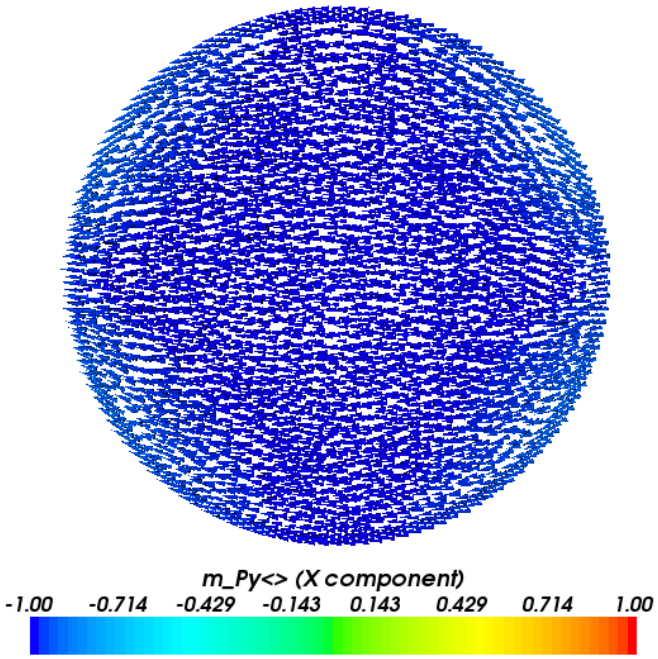


Figure 2.6: Magnetisation configuration for a decreasing applied field of -100 kA/m.

We see that during magnetisation reversal a vortex nucleates on the boundary of the disc when the field is sufficiently decreased from its saturation value. As the field direction is aligned with the x-axis, the vortex appears in the disc region with the largest y component, and it moves downwards towards the centre along the y-axis. With a further decrease of the applied field the vortex moves towards the opposite side of the disc with respect to the nucleation position, and it is eventually expelled when the magnetisation aligns with the field direction over all the disc.

2.7 Example: Manipulating magnetisation

There are two basic techniques to modify the magnetisation: on the one hand, we can use the *set_m* method to replace the current magnetisation configuration with a new one. We can use *set_m* to specify both homogeneous (see *Setting the initial magnetisation*) and non-homogeneous magnetisations (see the *Spin-waves example*). Alternatively, we can selectively change magnetic moments at individual mesh sites. This example demonstrates how to use the latter technique.

The basics of this system are as in *Example: Demag field in uniformly magnetised sphere*: we study a ferromagnetic sphere with initially homogeneous magnetisation. The corresponding script file is `sphere_manipulate.py`

```
import nmag
from nmag import SI

# Create simulation object
sim = nmag.Simulation()

# Define magnetic material
Py = nmag.MagMaterial(name="Py",
                      Ms=SI(1e6, "A/m"),
                      exchange_coupling=SI(13.0e-12, "J/m"))

# Load mesh
sim.load_mesh("sphere1.nmesh.h5", [("sphere", Py)], unit_length=SI(1e-9, "m"))

# Set initial magnetisation
sim.set_m([1, 0, 0])

# Set external field
sim.set_H_ext([0, 0, 0], SI("A/m"))

# Save and display data in a variety of ways
# Step 1: save all fields spatially resolved

sim.save_data(fields='all')

# Step 2: sample demag field through sphere
for i in range(-10, 11):
    x = i*1e-9                          # position in metres
    H_demag = sim.probe_subfield_siv('H_demag', [x, 0, 0])
    print "x =", x, ": H_demag = ", H_demag

# Step 3: sample exchange field through sphere
for i in range(-10, 11):
    x = i*1e-9                          # position in metres
    H_exch_Py = sim.probe_subfield_siv('H_exch_Py', [x, 0, 0])
    print "x =", x, ": H_exch_Py = ", H_exch_Py


# Now modify the magnetisation at position (0,0,0) (this happens to be
# node 0 in the mesh) in steps 4 to 6:

# Step 4: request a vector with the magnetisation of all sites in the mesh
```

```
myM = sim.get_subfield('m_Py')

# Step 5:  We modify the first entry:
myM[0] = [0, 1, 0]

# Step 6: Set the magnetisation to the new (modified) values
sim.set_m(myM)

# Step 7: saving the fields again (so that we can later plot the demag
# and exchange field)
sim.save_data(fields='all')

# Step 8: sample demag field through sphere (as step 2)
for i in range(-10, 11):
    x = i*1e-9                      # position in metres
    H_demag = sim.probe_subfield_siv('H_demag', [x, 0, 0])
    print "x =", x, ": H_demag = ", H_demag

# Step 9: sample exchange field through sphere (as step 3)
for i in range(-10, 11):
    x = i*1e-9                      # position in metres
    H_exch_Py = sim.probe_subfield_siv('H_exch_Py', [x, 0, 0])
    print "x =", x, ": H_exch_Py = ", H_exch_Py
```

To execute this script, we have to give its name to the *nsim* executable, for example (on linux):

```
$ nsim sphere_manipulate.py
```

After having created the simulation object, defined the material, loaded the mesh, set the initial magnetisation and the external field, we save the data the first time (Step 1).

We could visualise the magnetisation and all other fields as described in *Example: Demag field in uniformly magnetised sphere*, and would obtain the same figures as shown in section *Saving spatially resolved data*.

In step 2, we probe the demag field at positions along a line going from [-10,0,0]nm to [10,0,0]nm, and then print the values. This produces the following output:

```
x = -1e-08 : H_demag =  None
x = -9e-09 : H_demag =  [-329656.18892701436, 131.69946810517845, 197.13873034397167]
x = -8e-09 : H_demag =  [-329783.31649797881, 68.617197264295427, 140.00328871543459]
x = -7e-09 : H_demag =  [-329842.17628131888, 183.37401011699876, 163.01612229436262]
x = -6e-09 : H_demag =  [-329904.84956877632, 133.62473797637142, 74.090532749764847]
x = -5e-09 : H_demag =  [-329974.43178624194, 85.517390832982983, -13.956465964930704]
x = -4e-09 : H_demag =  [-330002.69224229571, 64.187663119270084, -30.832135394870004]
x = -3e-09 : H_demag =  [-330006.79488959321, 25.479055440690821, -61.958073893954818]
x = -2e-09 : H_demag =  [-330020.18327401817, 11.70722487517595, -58.143562276077219]
x = -1e-09 : H_demag =  [-330025.52325345919, -5.7120648683347452, -52.237341988696294]
x = 0.0    : H_demag =  [-330028.67095553532, -25.707310077918752, -46.346108473560378]
x = 1e-09  : H_demag =  [-330058.98559210222, -37.699378078580203, -41.167364094137213]
x = 2e-09  : H_demag =  [-330089.30022866925, -49.691446079241658, -35.988619714714041]
x = 3e-09  : H_demag =  [-330145.36618529289, -63.819285767062581, -22.213920341440794]
x = 4e-09  : H_demag =  [-330220.13307247689, -76.54950394725968, -5.0509172407556262]
x = 5e-09  : H_demag =  [-330298.69089200837, -90.534514175273259, 13.57279800234617]
x = 6e-09  : H_demag =  [-330375.34327985492, -117.01128011426778, 35.262477275758371]
x = 7e-09  : H_demag =  [-330415.38940687838, -123.68558207391983, 60.580352625726341]
x = 8e-09  : H_demag =  [-330474.37719032855, -112.22952205433305, 106.13032196062491]
x = 9e-09  : H_demag =  [-330499.64039893239, -69.97070465326442, 160.41688110297264]
x = 1e-08  : H_demag =  [-330518.649930441, -26.536490670368085, 212.32392103651733]
```

The data is approximately 1/3 Ms = 333333 (A/m) in the direction of the magnetisation, and approximately zero in the other directions, as we would expect in a homogeneously magnetised sphere. The deviations we see are due to (i) the shape of the sphere not being perfectly resolved (*ie* we actually look at the demag field of a polyhedron) and (ii) numerical errors.

In step 3, we probe the exchange field along the same line. The exchange field is effectively zero because the magnetisation is pointing everywhere in the same direction:

```
x = -1e-08 : H_exch_Py =  None
x = -9e-09 : H_exch_Py =  [-1.264324643856989e-09, 0.0, 0.0]
x = -8e-09 : H_exch_Py =  [-2.0419540595507732e-10, 0.0, 0.0]
x = -7e-09 : H_exch_Py =  [-1.4334754136843496e-09, 0.0, 0.0]
x = -6e-09 : H_exch_Py =  [-2.7214181426130964e-10, 0.0, 0.0]
x = -5e-09 : H_exch_Py =  [1.6323042074911775e-09, 0.0, 0.0]
x = -4e-09 : H_exch_Py =  [-1.6243345875473033e-09, 0.0, 0.0]
x = -3e-09 : H_exch_Py =  [-5.6526341264934703e-09, 0.0, 0.0]
x = -2e-09 : H_exch_Py =  [-6.1145979552370084e-09, 0.0, 0.0]
x = -1e-09 : H_exch_Py =  [-3.0929969691649876e-09, 0.0, 0.0]
x = 0.0    : H_exch_Py =  [9.2633407053741312e-10, 0.0, 0.0]
x = 1e-09  : H_exch_Py =  [1.9476821552904271e-09, 0.0, 0.0]
x = 2e-09  : H_exch_Py =  [2.9690302400434413e-09, 0.0, 0.0]
x = 3e-09  : H_exch_Py =  [2.6077357277001043e-09, 0.0, 0.0]
x = 4e-09  : H_exch_Py =  [1.5836815585162886e-09, 0.0, 0.0]
x = 5e-09  : H_exch_Py =  [1.6602158583197139e-09, 0.0, 0.0]
x = 6e-09  : H_exch_Py =  [1.8844573960991853e-09, 0.0, 0.0]
x = 7e-09  : H_exch_Py =  [-6.2460015649740799e-09, 0.0, 0.0]
x = 8e-09  : H_exch_Py =  [-1.1231714572170603e-08, 0.0, 0.0]
x = 9e-09  : H_exch_Py =  [-7.3643182171284044e-09, 0.0, 0.0]
x = 1e-08  : H_exch_Py =  [-3.4351784609779937e-09, 0.0, 0.0]
```

Note that the subfield name we are probing for the exchange field is `H_exch_Py` whereas the subfield name we used to probe the demag field is `H_demag` (without the extension `_Py`. The reason for this is that the exchange field is a something that is associated with a particular material (here Py) whereas there is only one demag field that is experienced by all materials (see also *Fields and subfields*).

2.7.1 Modifying the magnetisation

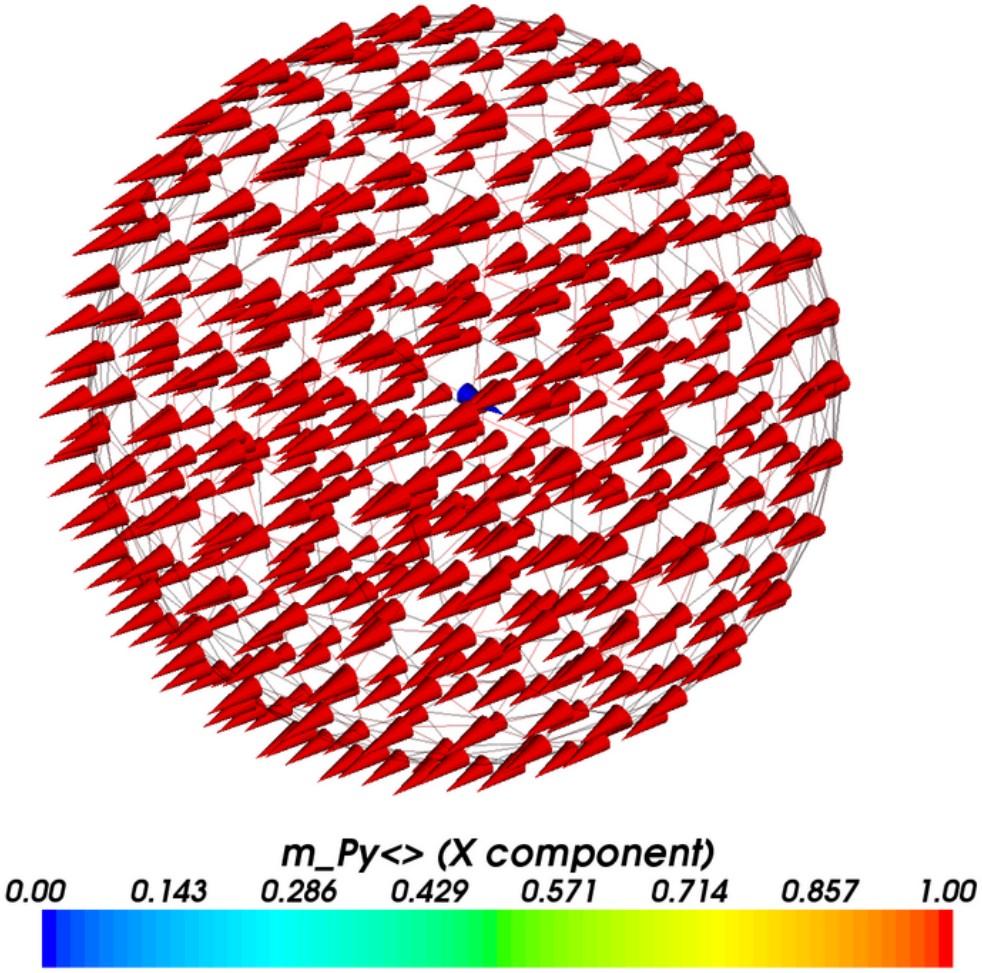
In step 4, we use the *get_subfield* command. This will return a (*NumPy*) array that contains one 3d vector for every *Site* of the finite element mesh.

In step 5, we modify the first entry in this array (which has index 0), and set its value to `[0,1,0]`. Whereas the magnetisation is pointing everywhere in [1,0,0] (because we have used the *set_m* command in the very beginning of the program, it is now pointing in the [0,1,0] at site 0.

The information, which site corresponds to which entry in the data vector, that we have obtained using *get_subfield*, can be retrieved from *get_subfield_sites*. Correspondingly, the position of the sites can be obtained using *get_subfield_positions*.

We now need to set this modified magnetisation vector (Step 6) using the *set_m* command.

If we save the data again to the file (Step 7), we can subsequently convert this to a vtk file (using, for example, `nmagpp --vtk data sphere_manipulate`) and visualise with *MayaVi*:



We can see one blue cone in the centre of the sphere - this is the one site that he have modified to point in the y-direction (whereas all other cones point in the x-direction).

As before, we can probe the fields along a line through the center of the sphere (Step 8). For the demag field we obtain:

```
x = -1e-08 : H_demag =  None
x = -9e-09 : H_demag =  [-333816.99138074159, -1884.643376396662, 16.665519199152595]
x = -8e-09 : H_demag =  [-334670.87148225965, -2293.608410913705, -102.38526828192296]
x = -7e-09 : H_demag =  [-335258.77403632947, -3061.1708540342884, -532.73877752122235]
x = -6e-09 : H_demag =  [-339506.72150998382, -5316.1506383768137, -969.36630578549921]
x = -5e-09 : H_demag =  [-344177.83909963415, -8732.9787600552572, -1610.433091871927]
x = -4e-09 : H_demag =  [-344725.75257842313, -16708.164927667149, -5224.2484897904633]
x = -3e-09 : H_demag =  [-337963.49070659198, -24567.078937669514, -3321.016613832679]
x = -2e-09 : H_demag =  [-321612.85117992124, -30613.873989917105, -1385.6383061516099]
x = -1e-09 : H_demag =  [-298312.3363571504, -41265.117003123923, 636.60703829516081]
x = 0.0    : H_demag =  [-273449.78240732534, -52534.176864875568, 2793.5027588779139]
x = 1e-09  : H_demag =  [-293644.21931918303, -39844.049389551074, 4310.6449471266505]
x = 2e-09  : H_demag =  [-313838.65623104072, -27153.921914226579, 5827.7871353753881]
x = 3e-09  : H_demag =  [-330296.09687372146, -21814.293451835449, 5525.7290665358933]
x = 4e-09  : H_demag =  [-343611.94111195666, -18185.932406317523, 4931.5464761658959]
x = 5e-09  : H_demag =  [-348062.40814087034, -11029.603829202088, 3781.8263522408147]
x = 6e-09  : H_demag =  [-342272.36888512014, -6604.210117819096, 50.151907623841332]
x = 7e-09  : H_demag =  [-338716.66400897497, -3860.7761876767272, 485.90273674867018]
x = 8e-09  : H_demag =  [-335656.89887674141, -2610.0345208853882, 586.74812908870092]
x = 9e-09  : H_demag =  [-334985.59512328985, -2169.9546280837162, 542.76746044672041]
x = 1e-08  : H_demag =  [-334441.59096545313, -1634.8337299563193, 627.17874011463311]
```

The change of the magnetisation at position [0,0,0] from [1,0,0] to [0,1,0] has reduced the x-component of the demag field somewhat around x=0, and has introduced a significant demag field in the -y direction around x=0.

Looking at the exchange field (Step 9):

```
x = -1e-08 : H_exch_Py =  None
x = -9e-09 : H_exch_Py =  [-1.264324643856989e-09, 0.0, 0.0]
x = -8e-09 : H_exch_Py =  [-2.0419540595507732e-10, 0.0, 0.0]
x = -7e-09 : H_exch_Py =  [-1.4334754136843496e-09, 0.0, 0.0]
x = -6e-09 : H_exch_Py =  [-2.7214181426130964e-10, 0.0, 0.0]
x = -5e-09 : H_exch_Py =  [1.6323042074911775e-09, 0.0, 0.0]
x = -4e-09 : H_exch_Py =  [-153858.81305452777, 153858.81305452611, 0.0]
x = -3e-09 : H_exch_Py =  [-972420.67935341748, 972420.67935341166, 0.0]
x = -2e-09 : H_exch_Py =  [-2445371.8369108676, 2445371.8369108611, 0.0]
x = -1e-09 : H_exch_Py =  [5283169.701234119, -5283169.7012341227, 0.0]
x = 0.0    : H_exch_Py =  [15888993.991894867, -15888993.991894867, 0.0]
x = 1e-09 :  H_exch_Py =  [8434471.7912872285, -8434471.7912872266, 0.0]
x = 2e-09 :  H_exch_Py =  [979949.59067958547, -979949.59067958279, 0.0]
x = 3e-09 :  H_exch_Py =  [-1112837.3087986181, 1112837.3087986207, 0.0]
x = 4e-09 :  H_exch_Py =  [-193877.66176242317, 193877.6617624248, 0.0]
x = 5e-09 :  H_exch_Py =  [1.6602158583197139e-09, 0.0, 0.0]
x = 6e-09 :  H_exch_Py =  [1.8844573960991853e-09, 0.0, 0.0]
x = 7e-09 :  H_exch_Py =  [-6.2460015649740799e-09, 0.0, 0.0]
x = 8e-09 :  H_exch_Py =  [-1.1231714572170603e-08, 0.0, 0.0]
x = 9e-09 :  H_exch_Py =  [-7.3643182171284044e-09, 0.0, 0.0]
x = 1e-08 :  H_exch_Py =  [-3.4351784609779937e-09, 0.0, 0.0]
```

We can see that the exchange field is indeed very large around x=0.

Note that one of the fundamental problem of micromagnetic simulations is that the magnetisation must not vary significantly from one site to another. In this example, we have manually violated this requirement *only to demonstrate* how the magnetisation can be modified, and to see that this is reflected in the dependant fields (such as demag and exchange) immediately.

2.8 Example: IPython

The basics of this file are as in *Example: Demag field in uniformly magnetised sphere*: a ferromagnetic sphere is studied, and initially configured to have homogeneous magnetisation.

Here is the source code of `sphere_ipython.py`

```
import nmag
from nmag import SI

# Create simulation object
sim = nmag.Simulation()

# Define magnetic material
Py = nmag.MagMaterial(name="Py",
                      Ms=SI(1e6,"A/m"),
                      exchange_coupling=SI(13.0e-12, "J/m"))

# Load mesh
sim.load_mesh("sphere1.nmesh.h5", [("sphere", Py)], unit_length=SI(1e-9,"m"))

# Set initial magnetisation
sim.set_m([1, 0, 0])

# Activate interactive python session
nmag.ipython()

print "Back in main code"
```

To execute this script, we have to give its name to the *nsim* executable, for example (on linux):

```
$ nsim sphere_ipython.py
```

The new command appearing here is:

```
nmag.ipython()
```

This calls an interactive python interpreter (this is like the standard `ipython` interpreter called from the command prompt).

Once we are "inside" this ipython interpreter, we can interactively work with the simulation object. We demonstrate this with the transcript of such a session:

```
$ nsim sphere_ipython.py

<snip>

In [1]: sim.get_subfield("H_demag")
Out[1]:
array([[ -3.30028671e+05,  -2.57073101e+01,  -4.63461085e+01],
       [ -3.30518650e+05,  -2.65364907e+01,   2.12323921e+02],
       [ -3.30380750e+05,  -1.34382835e+02,   1.94635283e+01],
       ...,
       [ -3.30063839e+05,   4.56312711e+01,  -1.31204248e+02],
       [ -3.30056243e+05,  -3.23341645e+01,  -2.26732582e+02],
       [ -3.29950815e+05,   4.44150291e+01,  -5.41700794e+01]])

In [2]: sim.set_m([0,0,1])

In [3]: sim.get_subfield("H_demag")
Out[3]:
array([[ -6.86773473e+01,   4.44496808e+01,  -3.30084368e+05],
       [ -2.83792944e+02,   1.78935681e+02,  -3.30268314e+05],
       [ -2.04396266e+02,   2.48374212e+02,  -3.30180923e+05],
       ...,
       [ -1.02055030e+02,  -9.53215211e+01,  -3.30239401e+05],
       [  1.94875407e+02,   1.22757584e+02,  -3.29771010e+05],
       [  6.16259262e+01,   1.66071597e+02,  -3.29848851e+05]])
```

Note that within ipython, one can just press the TAB key to autocomplete object names, functions and commands.

You can leave the ipython environment by pressing CTRL+D. For the script shown here, this will print `Back in main code` before the end of the script is reached. The `ipython()` command is occasionally a handy debugging feature: in order to investigate the behaviour of the system "on the spot", one can insert an `ipython` call into the script which will open an interactive command line.

2.9 Example: Pinning Magnetisation

In this example we show how to pin (*i.e.* fix) magnetisation in certain parts of a material.

2.9.1 Pinning simulation script

```
import nmag
from nmag import SI, si

# Create simulation object
sim = nmag.Simulation()

# Define magnetic material: PermAlloy
Py = nmag.MagMaterial(name="Py",
```

```
                    Ms=SI(0.86e6, "A/m"),
                    exchange_coupling=SI(13.0e-12, "J/m"))

# Load mesh
sim.load_mesh("sphere1.nmesh.h5", [("sphere", Py)], unit_length=SI(1e-9, "m"))

# Set initial magnetisation to +x direction
sim.set_m([1, 0, 0])

# Pin magnetisation at center in radius of 4e-9m
def pin_at_center((x, y, z)):
  if (x*x + y*y + z*z) < (4e-9)*(4e-9):
    return 0.0 # Inside the 4nm sphere -> pin
  else:
    return 1.0 # Outside -> do not pin

sim.set_pinning(pin_at_center)

# Apply external field in +y direction
unit = 0.5*si.Tesla/si.mu0 # 500mT in A/m
sim.set_H_ext([0*unit, 1*unit, 0*unit])

# Relax the magnetisation
sim.relax()
```

2.9.2 Pinning magnetisation

In order to allow the user to fix the magnetisation, nmag provides a scalar field, the so-called *pinning field*: its value at each site is used as a scale factor for `dm/dt`, hence by setting it to 0 at certain locations of the mesh we can force magnetisation to remain constant at these locations for the entire simulation.

We set the pinning field using *set_pinning* (which is used like *set_m* and *set_H_ext*, except that it is a scalar field whereas the latter are vector fields) such that magnetisation is fixed at sites with distance less than 4 nm from the sphere's center. First we define a Python function which we decide to call `pin_at_center`:

```
def pin_at_center((x, y, z)):
  if (x*x + y*y + z*z) < (4e-9)*(4e-9):
    return 0.0
  else:
    return 1.0
```

The function is called for each site of the mesh and receives the site position as an argument, a 3-tuple `(x, y, z)` containing the three components `x`, `y` and `z` (three floating point numbers), given in metres. The function returns either 0.0 (which means the magnetisation at this position is pinned) or 1.0 (in which case there is no pinning), for the given position vector.

The formula in the `if` statement simply evaluates the magnitude of the vector `(x, y, z)` by squaring each component. This number is then compared against (4nm)^2. As a result, the magnetisation is pinned at all the mesh nodes that are located within a sphere with center `(0, 0, 0)` and radius 4 nm. All the nodes that are located outside this sphere can change their magnetisation as usual.

Second, we need to tell nmag that it should use this function to decide where the magnetisation should be pinned:

```
sim.set_pinning(pin_at_center)
```

Note the slightly counterintuitive fact that value 1 means "no pinning".

Finally we apply an external field of 0.5 T in +y direction, and use *relax* to compute the equilibrium configuration.

The *relax* command:

```
sim.relax()
```

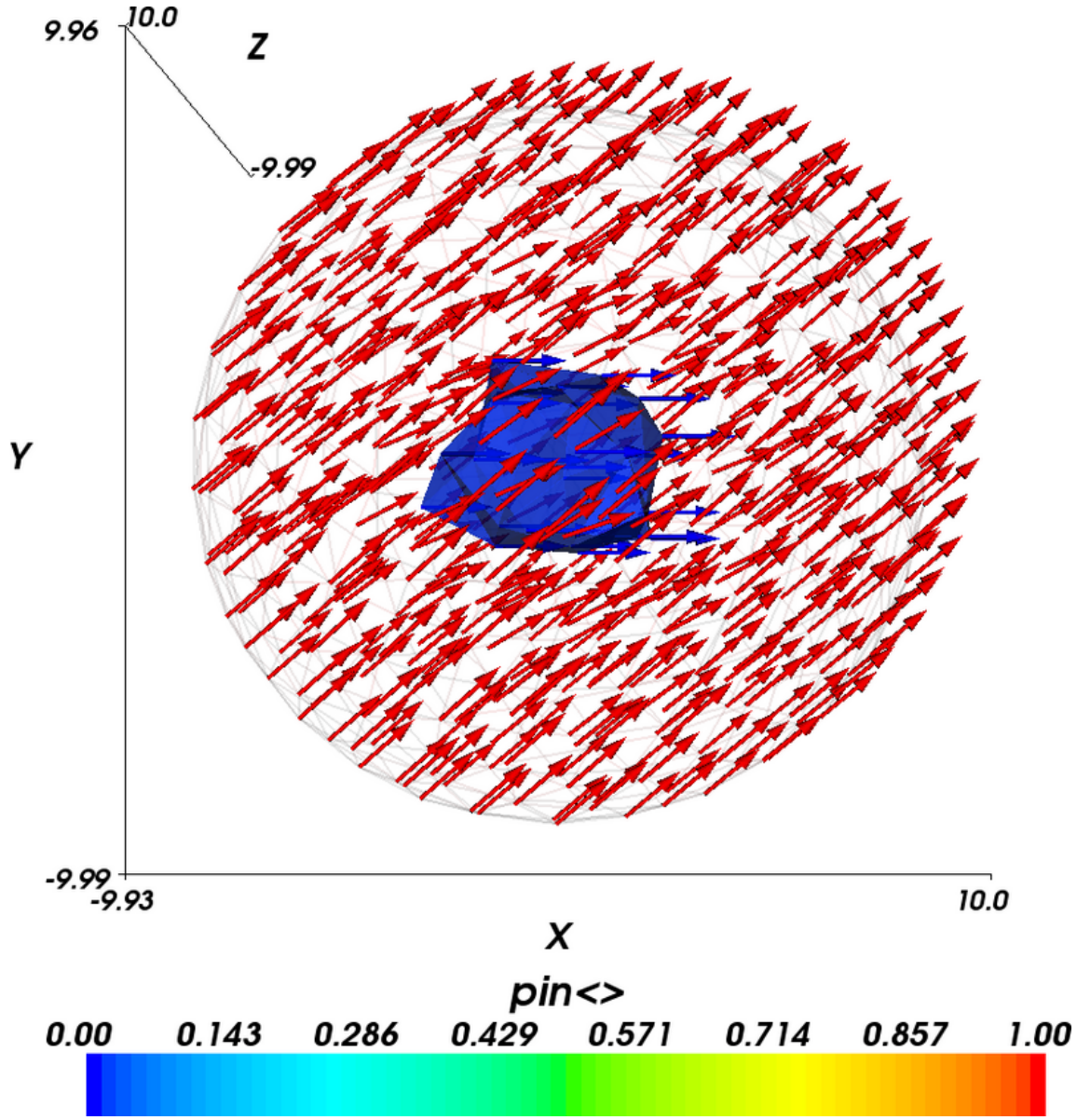
will save the fields and averages at convergence (this is the default of the *relax* command).

2.9.3 Visualisation

After running the example via `nsim sphere.py` we convert the equilibrium data to VTK format:

```
$ nmagpp --vtk=sphere.vtk sphere
```

We would first like to verify that the pinning field has been set up properly. Hence we use *MayaVi* to visualise it by showing an isosurface of the pinning field (shown in blue), together with the magnetisation vector field.

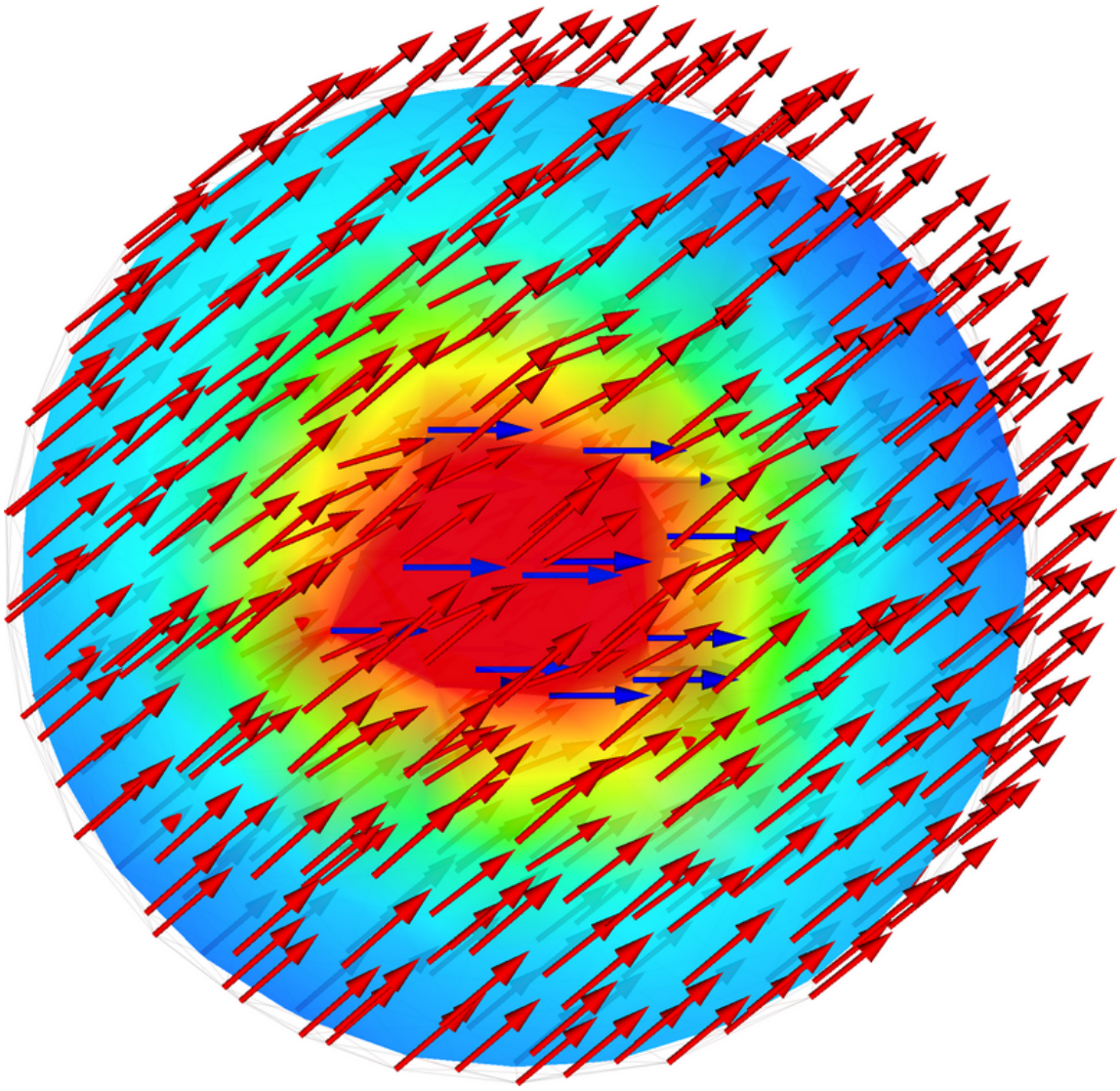


The blue blob in the center of the sphere is the collection of those tetrahedra that have corners just inside the 4nm sphere. Because we have not generated the mesh to have nodes coinciding with the 4nm sphere, the shape of the blue region is not particularly spherical.

In the above diagram, we also see the magnetisation vectors of the final configuration. Their colour corresponds to the pinning field at their location. It can be seen that the blue magnetisation vectors emerging from the central region of the sphere are all pointing (strictly) in the x-direction. The magnetisation vectors outside the blue sphere are coloured red. The applied field drives these vectors to point into the y-direction. However, the magnetisation in the centre is pinned and the exchange interaction requires a gradual spatial change of magnetisation. This explains the spatial variation of the magnetisation.

The next figure shows the same data as the last figure but in addition a `ScalarCutPlane` (in MayaVi terminology) has been introduced which is coloured according to the x-component of the magnetisation. Red corresponds

to 1.0 and blue corresponds to 0.73 (we have not shown the legend to provide a larger main plot). This demonstrates the gradual change from the pinned magnetisation in the centre to the outside.



2.10 Example: Uniaxial anisotropy

In this example we would like to simulate the development of a Bloch type domain wall on a thin cobalt bar of dimension 504 x 1 x 1 nm (`bar.nmesh.h5`) due to uniaxial anisotropy.

2.10.1 Uniaxial anisotropy simulation script

```
import nmag
from nmag import SI, every, at
from numpy import array
import math

# Create simulation object (no demag field!)
sim = nmag.Simulation(do_demag=False)

# Define magnetic material (data from OOMMF materials file)
Co = nmag.MagMaterial(name="Co",
                      Ms=SI(1400e3, "A/m"),
```

```
                        exchange_coupling=SI(30e-12, "J/m"),
                        anisotropy=nmag.uniaxial_anisotropy(axis=[0, 0, 1], K1=SI(520e3, "J/m^3")))

# Load the mesh
sim.load_mesh("bar.nmesh.h5", [("bar", Co)], unit_length=SI(1e-9,"m") )

# Our bar is subdivided into 3 regions:
# - region A: for x < offset;
# - region B: for x between offset and offset+length
# - region C: for x > offset+length;
# The magnetisation is defined over all the three regions,
# but is pinned in region A and C.
offset = 2e-9   # m (meters)
length = 500e-9 # m

# Set initial magnetisation
def sample_m0((x, y, z)):
  # relative_position goes linearly from -1 to +1 in region B
  relative_position = -2*(x - offset)/length + 1
  mz = min(1.0, max(-1.0, relative_position))
  return [0, math.sqrt(1 - mz*mz), mz]

sim.set_m(sample_m0)

# Pin magnetisation outside region B
def sample_pinning((x, y, z)):
  return x >= offset and x <= offset + length

sim.set_pinning(sample_pinning)

# Save the magnetisation along the x-axis
def save_magnetisation_along_x(sim):
  f = open('bar_mag_x.dat', 'w')
  for i in range(0, 504):
    x = array([i+0.5, 0.5, 0.5]) * 1e-9
    M = sim.probe_subfield_siv('M_Co', x)
    print >>f, x[0], M[0], M[1], M[2]

# Relax the system
sim.relax(save=[(save_magnetisation_along_x, at('convergence'))])
```

We shall now discuss the `bar.py` script step-by-step:

In this particular example we are solely interested in energy terms resulting from exchange interaction and anisotropy. Hence we disable the demagnetisation field as follows:

```
sim = nmag.Simulation(do_demag=False)
```

We then create the material `Co` used for the bar, cobalt in this case, which exhibits *uniaxial_anisotropy* in z direction with phenomenological anisotropy constant `K1 = SI(520e3, "J/m^3")`:

```
Co = nmag.MagMaterial(name="Co",
                        Ms=SI(1400e3, "A/m"),
                        exchange_coupling=SI(30e-12, "J/m"),
                        anisotropy=nmag.uniaxial_anisotropy(axis=[0, 0, 1], K1=SI(520e3, "J/m^3")))
```

After loading the mesh, we set the initial magnetisation direction such that it rotates from +z to -z while staying in the plane normal to x direction (hence suggesting the development of a Bloch type domain wall):

```
def sample_m0((x, y, z)):
  # relative_position goes linearly from -1 to +1 in region B
  relative_position = -2*(x - offset)/length + 1
  mz = min(1.0, max(-1.0, relative_position))
  return [0, math.sqrt(1 - mz*mz), mz]
```

We further pin the magnetisation at the very left (x < offset = 2 nm) and right (x > offset + length = 502 nm) of the bar (note that the pinning function may also just return a python truth value rather than the number 0.0 or 1.0):

```
def sample_pinning((x, y, z)):
  return x >= offset and x <= offset + length

sim.set_pinning(sample_pinning)
```

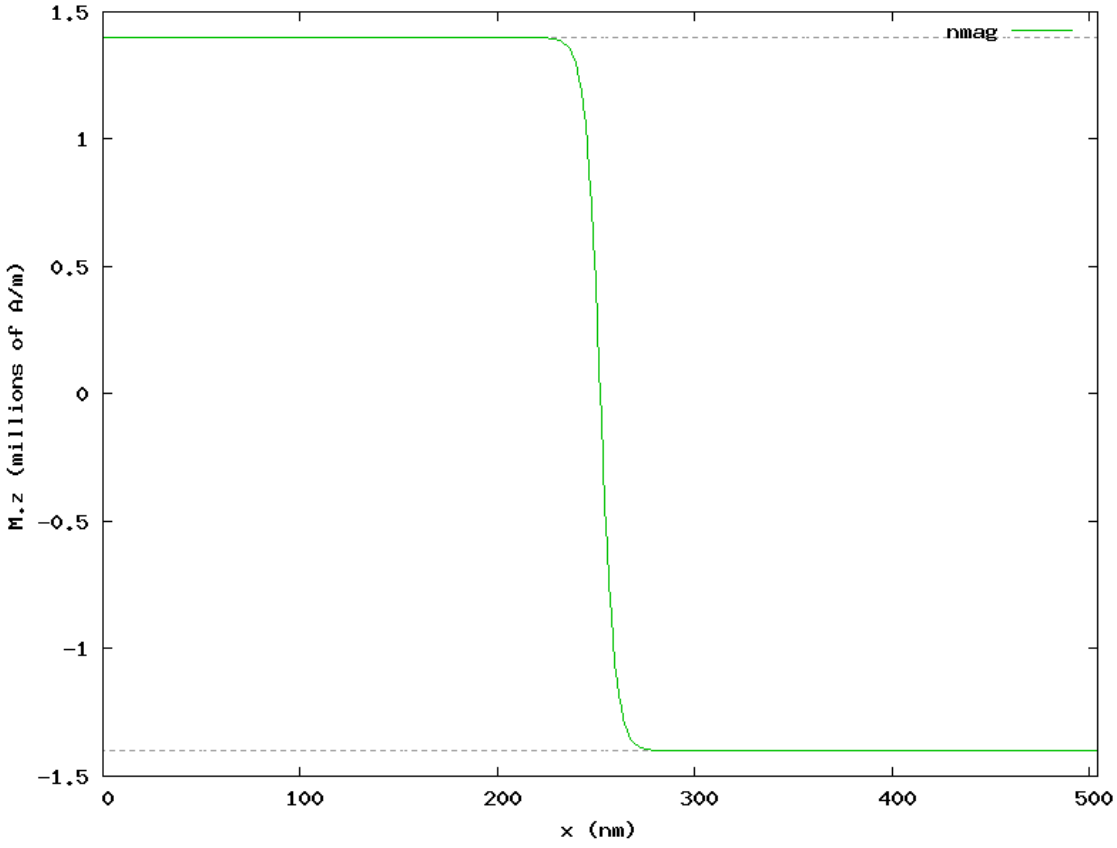
Finally, we relax the system to find the equilibrium magnetisation configuration, which is saved to the file `bar_mag_x.dat` in a format understandable by Gnuplot.

2.10.2 Visualization

We can then use the following Gnuplot script to visualize the equilibrium magnetisation:

```
set term png giant size 800, 600
set out 'bar_mag_x.png'
set xlabel 'x (nm)'
set ylabel 'M.z (millions of A/m)'

plot [0:504] [-1.5:1.5] \
  1.4 t "" w l 0, -1.4 t "" w l 0, \
  'bar_mag_x.dat' u ($1/1e-9):($4/1e6) t 'nmag' w l 2
```

The resulting plot clearly shows that a Bloch type domain wall has developed:



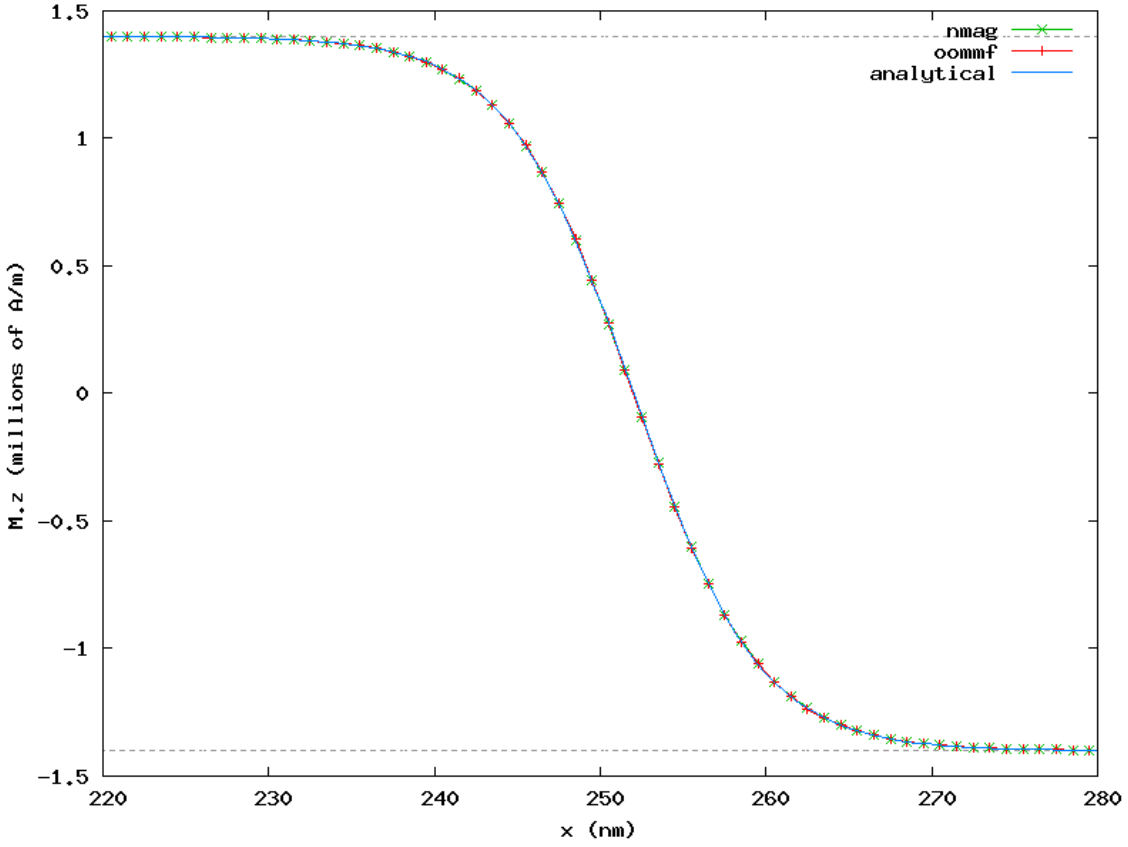
The figure shows also that the Bloch domain wall is well localized at the center of the bar, in the region where x goes from 200 to 300 nm.

2.10.3 Comparison

After simulating the same scenario with OOMMF (see `oommf/bar.mif`), we can compare results using another Gnuplot script:

```
#set term postscript enhanced eps color
set term png giant size 800, 600
set out 'bar_mag_x_compared.png'

set xlabel 'x (nm)'
set ylabel 'M.z (millions of A/m)'

Mz(x) = 1400e3 * cos(pi/2 + atan(sinh((x - 252e-9)/sqrt(30e-12/520e3))))

plot [220:280] [-1.5:1.5] \
  1.4 t "" w l 0, -1.4 t "" w l 0, \
  'bar_mag_x.dat' u ($1/1e-9):($4/1e6) t 'nmag' w lp 2, \
  'oommf/bar_mag_x.txt'u ($1/1e-9):($4/1e6) t 'oommf' w lp 1, \
  Mz(x*1e-9)/1e6 ti 'analytical' w l 3
```

which generates the following plot showing good agreement of both systems:



The plot shows also the known analytical solution:

```
Mz(x) = Ms * cos(pi/2 + atan(sinh((x - x_wall)/sqrt(A/K1))))
```

The plot shows only a restricted region located at the center of the bar, thus allowing an easier comparison between the three sets of data.

2.11 Example: Cubic Anisotropy

In this example we will study the behaviour of a 10 x 10 x 10 nm iron cube with *cubic_anisotropy* in an external field.

2.11.1 Cubic anisotropy simulation script

```
import nmag
from nmag import SI, si

# Create the simulation object
sim = nmag.Simulation()

# Define the magnetic material (data from OOMMF materials file)
Fe = nmag.MagMaterial(name="Fe",
                      Ms=SI(1700e3, "A/m"),
                      exchange_coupling=SI(21e-12, "J/m"),
                      anisotropy=nmag.cubic_anisotropy(axis1=[1, 0, 0],
                                                       axis2=[0, 1, 0],
                                                       K1=SI(48e3, "J/m^3")))

# Load the mesh
sim.load_mesh("cube.nmesh", [("cube", Fe)], unit_length=SI(1e-9, "m"))

# Set the initial magnetisation
sim.set_m([0, 0, 1])

# Launch the hysteresis loop
Hs = nmag.vector_set(direction=[1.0, 0, 0.0001],
                     norm_list=[0, 1, [], 19, 19.1, [], 21, 22, [], 50],
                     units=0.001*si.Tesla/si.mu0)
sim.hysteresis(Hs)
```

We will now discuss the `cube.py` script step-by-step:

After creating the simulation object we define a magnetic material `Fe` with cubic anisotropy representing iron:

```
Fe = nmag.MagMaterial(name="Fe",
                      Ms=SI(1700e3, "A/m"),
                      exchange_coupling=SI(21e-12, "J/m"),
                      anisotropy=nmag.cubic_anisotropy(axis1=[1,0,0],
                                                       axis2=[0,1,0],
                                                       K1=SI(48e3, "J/m^3")))
```

We load the mesh and set initial magnetisation pointing in +z direction (that is, in a local minimum of anisotropy energy density).

Finally, we use *hysteresis* to apply gradually stronger fields in +x direction (up to 50 mT):

```
Hs = nmag.vector_set(direction=[1.0, 0, 0.0001],
                     norm_list=[0, 1, [], 19, 19.1, [], 21, 22, [], 50],
                     units=0.001*si.Tesla/si.mu0)
```

Note that we sample more often the region between 19 and 21 mT where magnetisation direction changes rapidly due to having crossed the anisotropy energy "barrier" between +z and +x (as can be seen in the graph below).

2.11.2 Analyzing the result

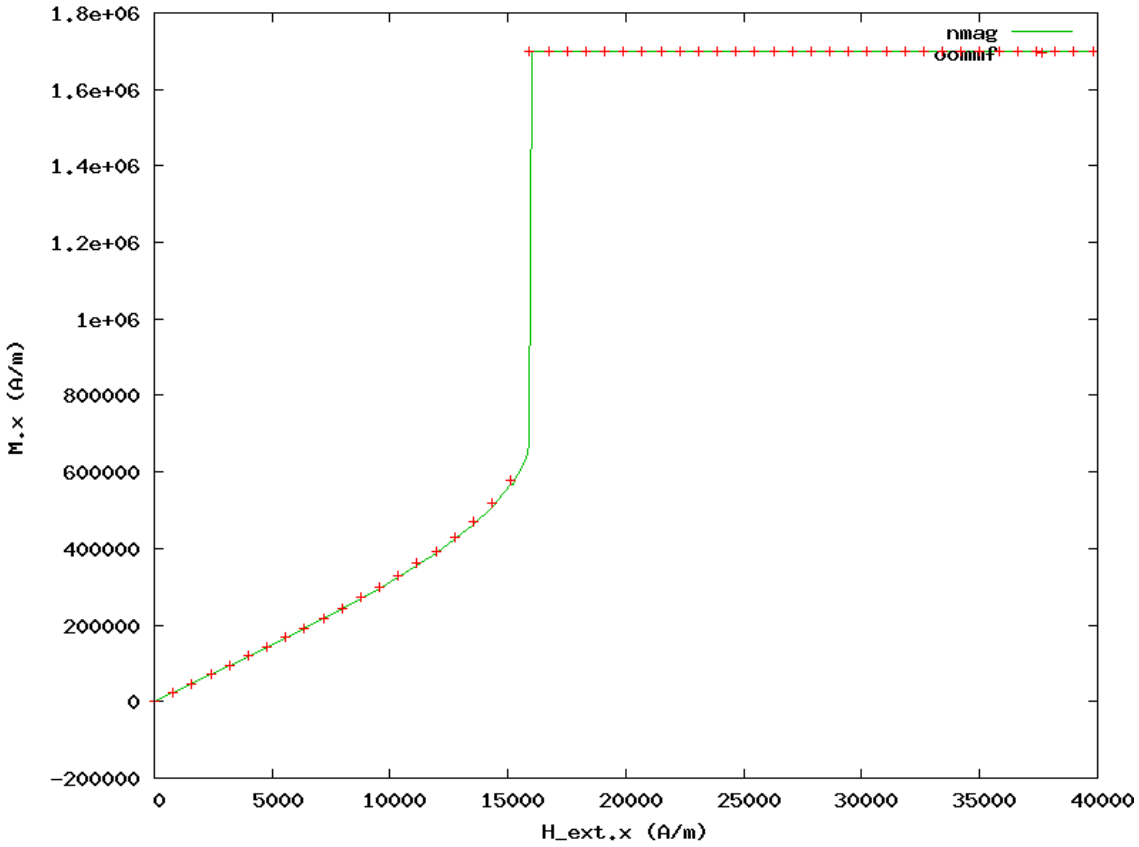
First, we extract the magnitude of the applied field and the x component of magnetisation:

```
ncol cube H_ext_0 M_Fe_0 > cube_hext_vs_m.txt
```

Then we compare the result with OOMMF's result (generated from the equivalent scene description `oommf/cube.mif`) using the following Gnuplot script:

```
set term png giant size 800,600
set out 'cube_hext_vs_m.png'
set xlabel 'H_ext.x (A/m)'
set ylabel 'M.x (A/m)'
plot 'cube_hext_vs_m.txt' t 'nmag' w l 2,\
 'oommf/cube_hext_vs_m.txt' u ($1*795.77471545947674):2 ti 'oommf' w p 1
```

which gives the following result:



Nmag provides advanced capabilities to conveniently handle arbitrary-order anisotropy energy functions. Details can be found in the documentation of the *MagMaterial* class.

2.12 Example: Arbitrary Anisotropy

In this example we discuss the script `coin.py` which shows how the user can include in his simulations a customised magnetic anisotropy.

2.12.1 Arbitrary anisotropy simulation script

```
import nmag
from nmag import SI, every, at
from nsim.si_units import si
import math

# Create simulation object (no demag field!)
```

```
sim = nmag.Simulation(do_demag=False)

# Function to compute the scalar product of the vectors a and b
def scalar_product(a, b): return a[0]*b[0] + a[1]*b[1] + a[2]*b[2]

# Here we define a function which returns the energy for a uniaxial
# anisotropy of order 4.
K1 = SI(43e3, "J/m^3")
K2 = SI(21e3, "J/m^3")
axis = [0, 0, 1]        # The (normalised) axis
def my_anisotropy(m):
    a = scalar_product(axis, m)
    return -K1*a**2 - K2*a**4

my_material = nmag.MagMaterial(name="MyMat",
                               Ms=SI(1e6, "A/m"),
                               exchange_coupling=SI(10e-12, "J/m"),
                               anisotropy=my_anisotropy,
                               anisotropy_order=4)

# Load the mesh
sim.load_mesh("coin.nmesh.h5", [("coin", my_material)], unit_length=SI(1e-9, "m"))

# Set the magnetization
sim.set_m([-1, 0, 0])

# Compute the hysteresis loop
Hs = nmag.vector_set(direction=[1.0, 0, 0.0001],
                     norm_list=[-0.4, -0.35, [], 0, 0.005, [], 0.15, 0.2, [], 0.4],
                     units=si.Tesla/si.mu0)

sim.hysteresis(Hs, save=[('fields', 'averages', at('convergence'))])
```

We simulate the hysteresis loop for a ferromagnetic thin disc, where the field is applied orthogonal to the axis of disc. This script includes one main element of novelty, which concerns the way the magnetic anisotropy is specified. In previous examples we found lines such as:

```
my_material = nmag.MagMaterial(name="MyMat",
                               Ms=SI(1e6, "A/m"),
                               exchange_coupling=SI(10e-12, "J/m"),
                               anisotropy=nmag.uniaxial_anisotropy(axis=[0, 0, 1],
                                                                   K1=SI(43e3, "J/m^3"),
                                                                   K2=SI(21e3, "J/m^3")))
```

where the material anisotropy was specified using the provided functions `nmag.uniaxial_anisotropy` (*uniaxial_anisotropy*) and `nmag.cubic_anisotropy` (*cubic_anisotropy*). In this example we are using a different approach to define the anisotropy. First we define the function `my_anisotropy`, which returns the energy density for the magnetic anisotropy:

```
# Here we define a function which returns the energy for a uniaxial
# anisotropy of order 4.
K1 = SI(43e3, "J/m^3")
K2 = SI(21e3, "J/m^3")
axis = [0, 0, 1]        # The (normalised) axis
def my_anisotropy(m):
    a = scalar_product(axis, m)
    return -K1*a**2 - K2*a**4
```

Note that the function returns a SI object with units "J/m^3" (energy density). The reader may have recognised the familiar expression for the uniaxial anisotropy: in fact the two code snippets we just presented are defining exactly the same anisotropy, they are just doing it in different ways. The function `scalar_product`, which we have used in the second code snippet just returns the scalar product of two three dimensional vectors `a` and `b` and is defined in the line above:

```
def scalar_product(a, b): return a[0]*b[0] + a[1]*b[1] + a[2]*b[2]
```

The function `my_anisotropy` has to be specified in the material definition: instead of passing `anisotropy=nmag.uniaxial_anisotropy(...)` we just pass `anisotropy=my_anisotropy` to the material constructor:

```
my_material = nmag.MagMaterial(name="MyMat",
                               Ms=SI(1e6, "A/m"),
                               exchange_coupling=SI(10e-12, "J/m"),
                               anisotropy=my_anisotropy,
                               anisotropy_order=4)
```

An important point to notice is that here we also provide an anisotropy order. To understand what this number is, we have to explain briefly what is going on behind the scenes. nsim calculates the values of the user provided function for an appropriately chosen set of normalised vectors, it then finds the polynomial in `mx`, `my` and `mz` (the components of the normalised magnetisation) of the specified order, which matches the sampled values.

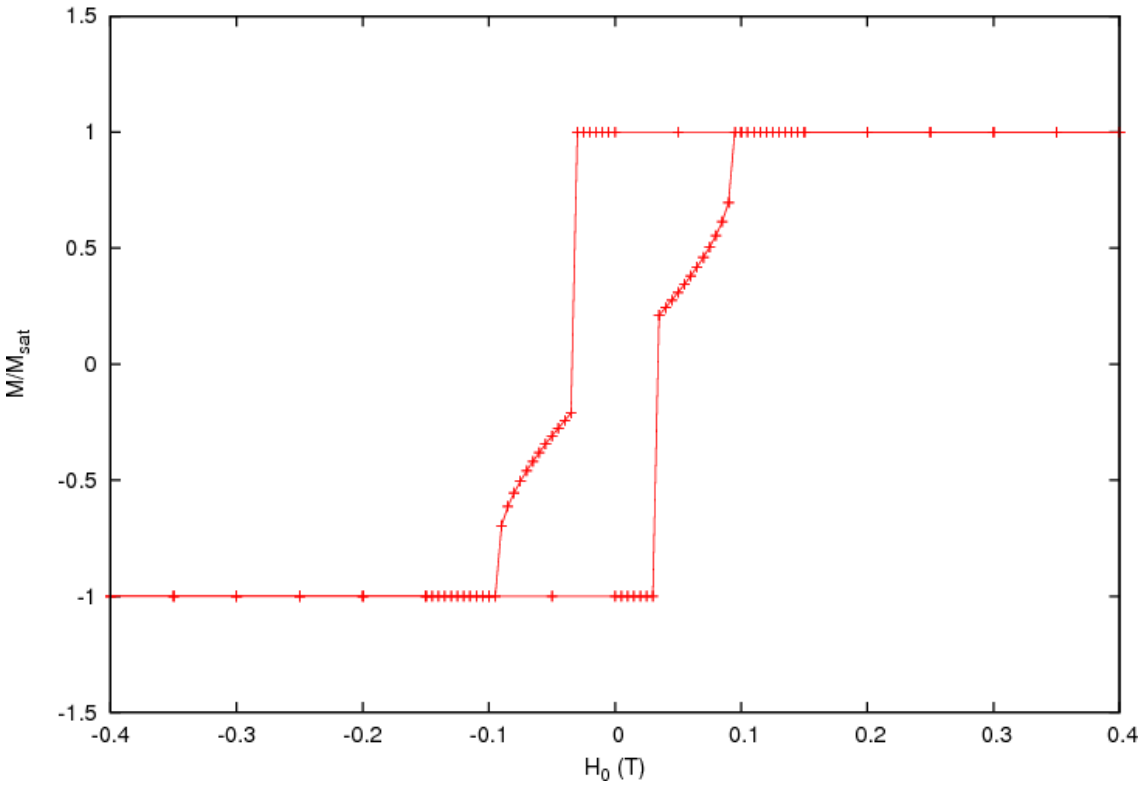
The strength of this approach stands in the fact that the user has to provide just the energy density for the custom anisotropy. nsim is taking care of working out the other quantities which are needed for the simulation, such as the magnetic field resulting from the provided anisotropy energy, which would require a differentiation of the energy with respect to the normalised magnetisation.

However the user must be sure that the provided function can be expressed by a polynomial of the specified order in `mx`, `my` and `mz`. In the present case we are specifying `anisotropy_order=4` because the energy for the uniaxial anisotropy can be expressed as a 4th-order polynomial in `mx`, `my` and `mz`.

In some cases the user may find useful to know that the functions `nmag.uniaxial_anisotropy` and `nmag.cubic_anisotropy` can be added: the resulting anisotropy will have as energy the sum of the energies of the original anisotropies.

2.12.2 The result

The steps involved to extract and plot the data for the simulation discussed in the previous section should be familiar to the user at this point of the manual. We then just show the graph obtained from the results of the script `coin.py`.



During the switching the system falls into an intermediate state, where the magnetisation is nearly aligned with the anisotropy easy axis.

2.13 Restart example

Micromagnetic simulations can last for many hours or even many days. It is then important to be able to save periodically the state of the simulation, in such a way that, if a hardware failure or a power cut occurs, the simulation can be restarted exactly at the point where its state was last saved. In this example we show how an nmag script can be modified to be "restartable". The only thing the user needs to do is to periodically save the state of the simulation in what we call a "restart file". The simulation can then be restarted using the appropriate command line option.

The restart feature applies only to the *hysteresis* method.

2.13.1 Saving the state of the simulation

We re-consider the cubic anisotropy example (*Cubic anisotropy simulation script*) and replace the last line:

```
sim.hysteresis(Hs)
```

with the following lines:

```
from nmag import every, at
sim.hysteresis(Hs, save=[('averages', 'fields', at('stage_end')),
                         ('restart', at('stage_end') | every('step', 1000))])
```

The first two lines reproduce the default behaviour: the fields and their averages are saved at the end of each stage. The third line specifies that the restart file should be saved at the end of each stage and also every 1000 steps.

For convenience the modified script `cube_restartable.py` is shown below:

```
import nmag
from nmag import SI, si

# Create the simulation object
sim = nmag.Simulation()

# Define the magnetic material (data from OOMMF materials file)
Fe = nmag.MagMaterial(name="Fe",
                      Ms=SI(1700e3, "A/m"),
                      exchange_coupling=SI(21e-12, "J/m"),
                      anisotropy=nmag.cubic_anisotropy(axis1=[1, 0, 0],
                                                       axis2=[0, 1, 0],
                                                       K1=SI(48e3, "J/m^3")))

# Load the mesh
sim.load_mesh("cube.nmesh", [("cube", Fe)], unit_length=SI(1e-9, "m"))

# Set the initial magnetisation
sim.set_m([0, 0, 1])

# Launch the hysteresis loop
Hs = nmag.vector_set(direction=[1.0, 0, 0.0001],
                     norm_list=[0, 1, [], 19, 19.1, [], 21, 22, [], 50],
                     units=0.001*si.Tesla/si.mu0)
from nmag import every, at
sim.hysteresis(Hs, save=[('averages', 'fields', at('stage_end')),
                         ('restart', at('stage_end') | every('step', 1000))])
```

2.13.2 Starting and restarting the simulation

We will now demonstrate how the discussed nmag script can be restarted. To do that, we will have to interrupt it artificially. We start the simulation in the usual way:

```
$ nsim cube_restartable.py
```

We interrupt the execution after the hysteresis loop has started and several stages have been computed. Do this by pressing simultaneously the keys CTRL and C (in the same terminal window where nsim was started), thus simulating what could have been the result of a power cut. We then use the command:

```
$ ncol cube_restartable stage step time
```

to see at what point of the hysteresis loop the simulation was interrupted. We obtain (for this particular interruption):

```
 1            330  3.320127110062e-11
 2            480  5.042492488627e-10
 3            640  9.926580643272e-10
 4            805  1.464971830453e-09
 5            980  1.927649646634e-09
 6           1150  2.406521613682e-09
 7           1340  2.882400372552e-09
 8           1515  3.371522550051e-09
 9           1705  3.863380029345e-09
10           1920  4.365560120394e-09
11           2095  4.893234441813e-09
12           2295  5.436617525896e-09
13           2480  5.997866344586e-09
14           2680  6.570733097131e-09
15           2890  7.172534305054e-09
16           3100  7.803577637245e-09
17           3315  8.462827284047e-09
```

The simulation was interrupted at the seventeenth stage. We now try to run the simulation again with the command:

```
$ nsim cube_restartable.py
```

obtaining the following output:

```
<snip>
NmagUserError: Error: Found old file ./cube_restartable_dat.ndt -- cannot proceed.
To start a simulation script with old data files present you either need
to use '--clean' (and then the old files will be deleted), or use '--restart'
in which case the run will be continued.
```

nsim suggests the possible alternatives. We can start the simulation from scratch with the command (but this will override any data from the previous run):

```
$ nsim cube_restartable.py --clean
```

or we can continue from the configuration which was last saved:

```
$ nsim cube_restartable.py --restart
```

Here we choose the second possibility. After the simulation has finished we issue again the command `ncol cube_restartable stage step time`, obtaining the following output:

```
    1            330  3.320127110062e-11
    2            480  5.042492488627e-10
    3            640  9.926580643272e-10
    4            805  1.464971830453e-09
    5            980  1.927649646634e-09
    6           1150  2.406521613682e-09
    7           1340  2.882400372552e-09
```

```
    8           1515  3.371522550051e-09
    9           1705  3.863380029345e-09
   10           1920  4.365560120394e-09
   11           2095  4.893234441813e-09
   12           2295  5.436617525896e-09
   13           2480  5.997866344586e-09
   14           2680  6.570733097131e-09
   15           2890  7.172534305054e-09
   16           3100  7.803577637245e-09
   17           3315  8.462827284047e-09
stage           step          #time
   <>             <>           #<s>
   18           3715  8.519843629989e-09
   19           3975  9.300878866142e-09
   ...
```

The two lines between stage 17 and 18 stand as a reminder that the simulation was restarted at that point. (They need to be removed manually from the `cube_restartable_dat.ndt` file, before *ncol* can work in the usual way on the ndt file.)

2.14 Applying a field that changes both in time and in space

2.14.1 Idea: pass simulation object to field-setting function

You can simulate an applied field which both changes in space and time: this may be useful to mimic the effect of a write head on the magnetic grains of an hard disk while the head is moving. The way we do this is by changing the applied field every `delta_t` picoseconds. This means that the applied field won't change continuously in time: it will be piecewise constant in time (but, in general, it can be non uniform in space). You can do something like:

```
import math

def set_H(sim):
  width = 10.0         # nm
  v = 100.0            # nm/ns == m/s
  H_amplitude = 0.5e6 # A/m

  t = float(sim.time/SI(1e-9, 's')) # get the time in ns
  center = (v*t, 0, 0) # center of the applied field region
  def H(r):
    x, y, z = [ri/1e-9 - ci for ri, ci in zip(r, center)]
    factor = H_amplitude*math.exp(-(x*x + y*y + z*z)/(width*width))
    return [factor, factor, factor]

  sim.set_H_ext(H, unit=SI('A/m'))

sim.relax(do=[(set_H, every('time', SI(50e-12, 's'))),
              ('exit', at('time', SI(1000e-12, 's')))])
```

The function `set_H` is called every 50 ps and does the following: it sets a new field from the function `H(r)`. This function sets a field which directed along the direction [1, 1, 1] and almost vanishes outside a sphere with radius ~ 30.0 nm. The center of this sphere moves along the direction [1, 0, 0] with velocity 100 nm/ns, thus simulating the motion of a write head in a hard disk. Obviously the piece of code is not complete, it shows only the technique in order to have a field changing in time and space. For a complete example see the next section.

2.14.2 Complete example: simple moving write-head example

Here is a simulation of five cubes made of cobalt and a write-head which moves on the top of the cubes and applies a time-varying field in order to change their magnetisation. At the beginning the magnetisation of all the cubes is

pointing in the [0, 0, 1] direction. After the write-head has passed over the cubes, the magnetisation of cube 1, 3 and 5 are switched in the opposite direction, while cube 2 and 4 have unchanged magnetisation. This is possible because the write-head field, which is space-dependent (being intense only inside a sphere of radius 15-20 nm), changes also in time. It indeed translates in space, but also change in intensity, being directed in the [0, 0, -1] direction when the sphere is at the center of cube 1, 3 and 5 and in the [0, 0, 1] direction when the center of the sphere is in cube 2 and 4.

Here is the geo file used to generate the mesh (Netgen):

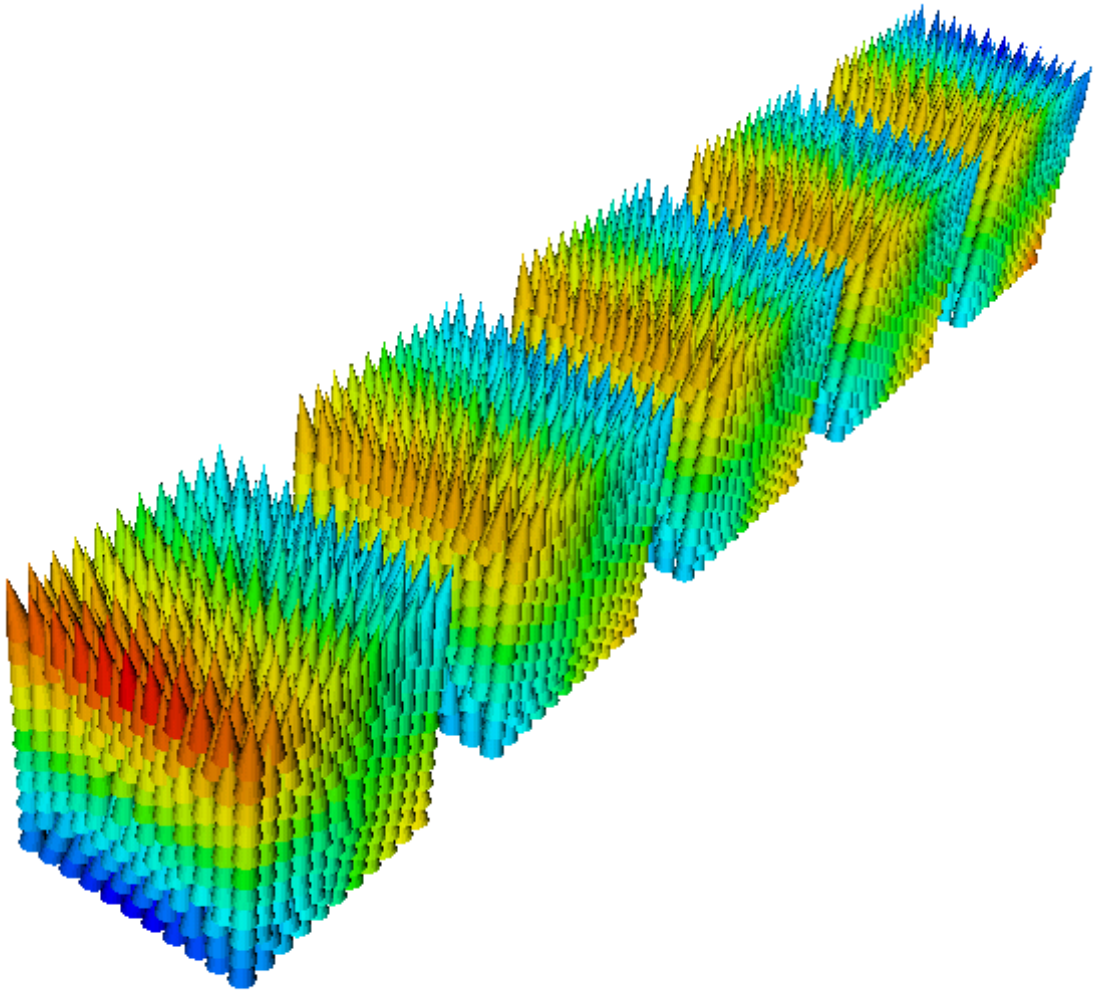
```
<pre>
algebraic3d

# cubes
solid cube1 = orthobrick (    0, 0, 0;  20.0, 20.0, 20.0) -maxh = 2;
solid cube2 = orthobrick ( 30.0, 0, 0;  50.0, 20.0, 20.0) -maxh = 2;
solid cube3 = orthobrick ( 60.0, 0, 0;  80.0, 20.0, 20.0) -maxh = 2;
solid cube4 = orthobrick ( 90.0, 0, 0; 110.0, 20.0, 20.0) -maxh = 2;
solid cube5 = orthobrick (120.0, 0, 0; 140.0, 20.0, 20.0) -maxh = 2;

tlo cube1;
tlo cube2;
tlo cube3;
tlo cube4;
tlo cube5;
```

And here is the full listing of the example:

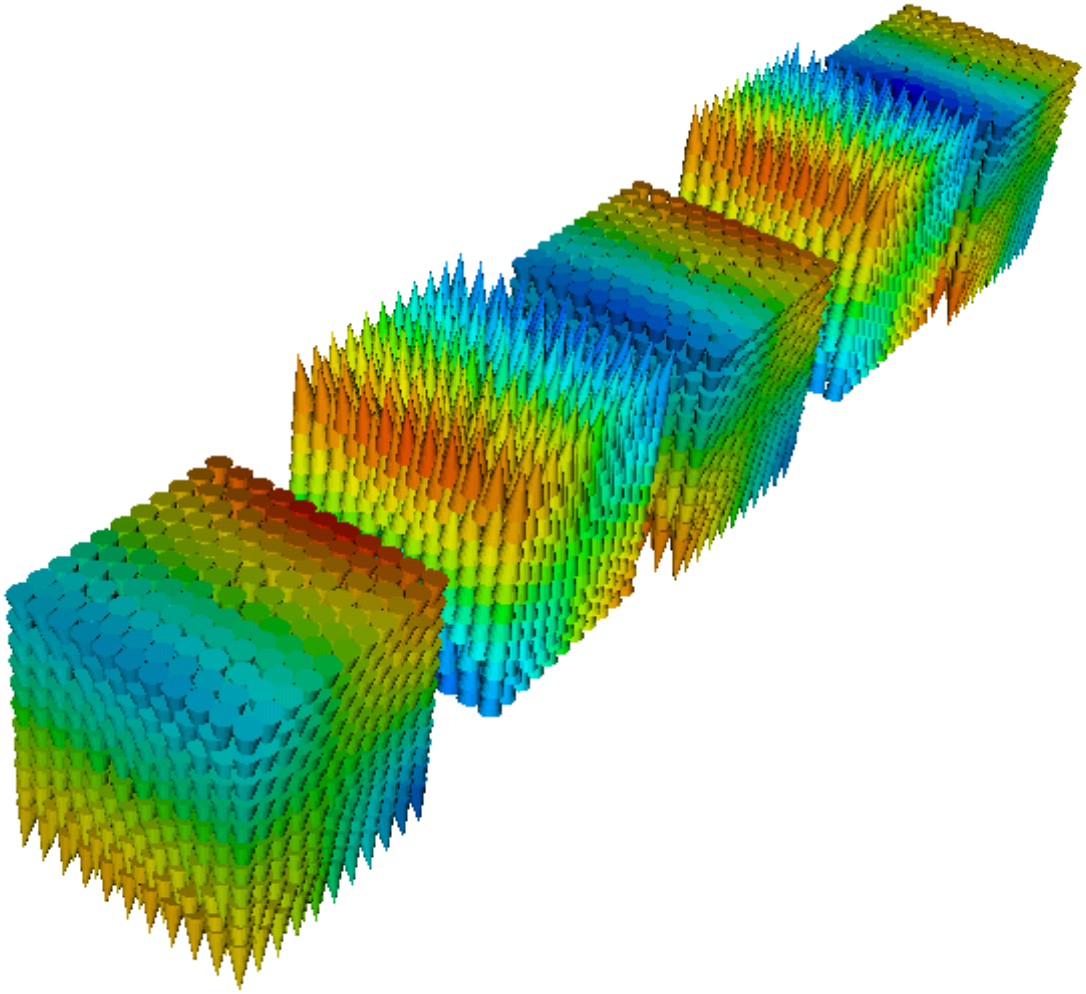
```
from nmag.common import *
import math

# Define magnetic material (data from OOMMF materials file)
mat_Co = MagMaterial(name="Co",
                     Ms=SI(1400e3, "A/m"),
                     exchange_coupling=SI(30e-12, "J/m"),
                     anisotropy=uniaxial_anisotropy(axis=[0, 0, 1],
                                                    K1=SI(520e3, "J/m^3")))
sim = Simulation()
sim.load_mesh("cubes.nmesh.h5",
              [('cube1', mat_Co), ('cube2', mat_Co), ('cube3', mat_Co),
               ('cube4', mat_Co), ('cube5', mat_Co)],
              unit_length=SI(1e-9, 'm'))

sim.set_m([0, 0, 1])

sim.relax(save=[('fields', at('convergence'))])

t0 = [sim.time]

def set_H(sim):
    t = float((sim.time - t0[0])/SI(1e-9, 's'))  # get time in ns
    width = 10.0                                 # nm
    v = 25.0                                     # nm/ns = m/s
    H_amplitude = 4.0e6*math.sin(math.pi*t)      # A/m
    center = (v*t, 20, 10)
    print "CENTER IN", center
    def H(r):
        x, y, z = [ri/1e-9 - ci for ri, ci in zip(r, center)]
        factor = H_amplitude*math.exp(-(x*x + y*y + z*z)/(width*width))
        return [0, 0, -factor]

    sim.set_H_ext(H, unit=SI('A/m'))

set_H(sim)
```

```
sim.set_params(stopping_dm_dt=0*degrees_per_ns)

sim.relax(save=[('fields', every('time', SI(200e-12, 's'), first=t0[0]))],
          do=[(set_H, every('time', SI(50e-12, 's'), first=t0[0])),
              ('exit', at('time', SI(6000e-12, 's')))])
```

Here is the magnetisation at the beginning of the simulation, after the first relax command (whose purpose is just to find the zero field magnetisation configuration):

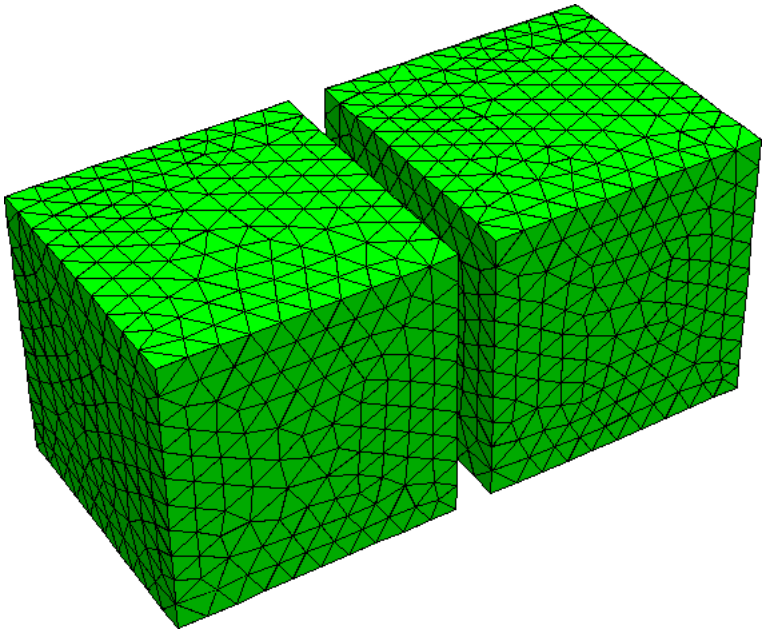


and here is the magnetisation after the write-head has passed over the cubes:



2.15 Example: two different magnetic materials

In this example, we study the dynamics of a simple system consisting of two 15 nm x 15 nm x 15 nm cubes close to one another (with 2 nm spacing along the x-axis). We take the right cube to be made of PermAlloy and the left cube to be made of Cobalt, with the magnetic anisotropy axis pointing in z-direction. The mesh has been generated with Netgen from the geometry file `two_cubes.geo`.



We use the `two_cubes.py` script to carry out the simulation:

```
import nmag
from nmag import SI, every, at

sim = nmag.Simulation()

# define magnetic material Cobalt (data from OOMMF materials file)
Co = nmag.MagMaterial(name="Co",
                      Ms=SI(1400e3, "A/m"),
                      exchange_coupling=SI(30e-12, "J/m"),
                      anisotropy=nmag.uniaxial_anisotropy(axis=[0,0,1], K1=SI(520e3, "J/m^3")))

# define magnetic material Permalley
Py = nmag.MagMaterial(name="Py",
                      Ms=SI(860e3,"A/m"),
                      exchange_coupling=SI(13.0e-12, "J/m"))

# load mesh
sim.load_mesh("two_cubes.nmesh.h5",
              [("cube1", Py),("cube2", Co)],
              unit_length=SI(1e-9,"m")
              )

# set initial magnetisation along the
# positive x axis for both cubes, slightly off in z-direction
sim.set_m([0.999847695156, 0, 0.01745240643731])

ns = SI(1e-9, "s") # corresponds to one nanosecond

sim.relax(save = [('averages', every('time', 0.01*ns)),
                  ('fields', every('time', 0.05*ns) | at('convergence'))])
```

The script is very similar to the one used in *Example 2: Computing the time development of a system*. However, here we have two materials. The related changes are that we define two magnetic materials, and assign them to objects `Co` and `Py`.

When loading the mesh:

```
sim.load_mesh("two_cubes.nmesh.h5",
              [("cube1", Py),("cube2", Co)],
              unit_length=SI(1e-9,"m")
```

```
    )
```

we need to assign regions 1 and 2 in the mesh file (which correspond to the two cubes) to the materials. This is done with this list of tuples:

```
[("cube1", Py),("cube2", Co)]
```

The first list entry is `("cube1", Py)` and tells nmag that we would like to refer to the region 1 as `cube1`, and that we would like to assign the material `Py` to this region. This entry refers to region 1 because it is the *first* entry in the list.

The second list entry is `("cube2", Co)` and tells nmag that we would like to refer to the region 2 as `cube2`, and that we would like to assign the material `Co` to this region.

If there was a region 3 in the mesh file, we would add a third list entry, for example ("cylinder",Co) for a Co cylinder.

Note that at this stage of nmag, the region name (such as `cube1`, `cube2`, etc) are not used in the simulation, apart from diagnostic purposes in progress messages.

Physically, what happens in this system is that the magnetisation of the Cobalt cube aligns rather fast with the anisotropy direction and then slowly forces the magnetisation of the PermAlloy cube into the opposite direction (through the action of the stray field) to minimise total energy of the configuration.

The Initial magnetisation is taken to point in x-direction. As this is an unstable equilibrium direction for the magnetisation anisotropy of the Cobalt cube, we slightly distort the initial magnetisation by adding a tiny component in +z-direction.

It is instructive to compare the *Field*s and *Subfield*s for this particular example with the list of fields and subfields for a single-material simulation. In effect, all the fields that are related to the properties of some particular magnetic component carry multiple subfields. In particular, there is only one `H_ext` field, as the externally applied field is experienced in the same way by all materials, but the `M*H` energy density associated with `H_ext` has a dependency on the magnetic component (through `M`), so we have two subfields `E_ext_Py` and `E_ext_Co` in the field `E_ext`.

The situation is virtually identical with `H_demag`/`E_demag` and the related charge density `rho` and magnetic scalar potential `phi`. All the other relevant fields in this example turn out to be related to a particular magnetic component.

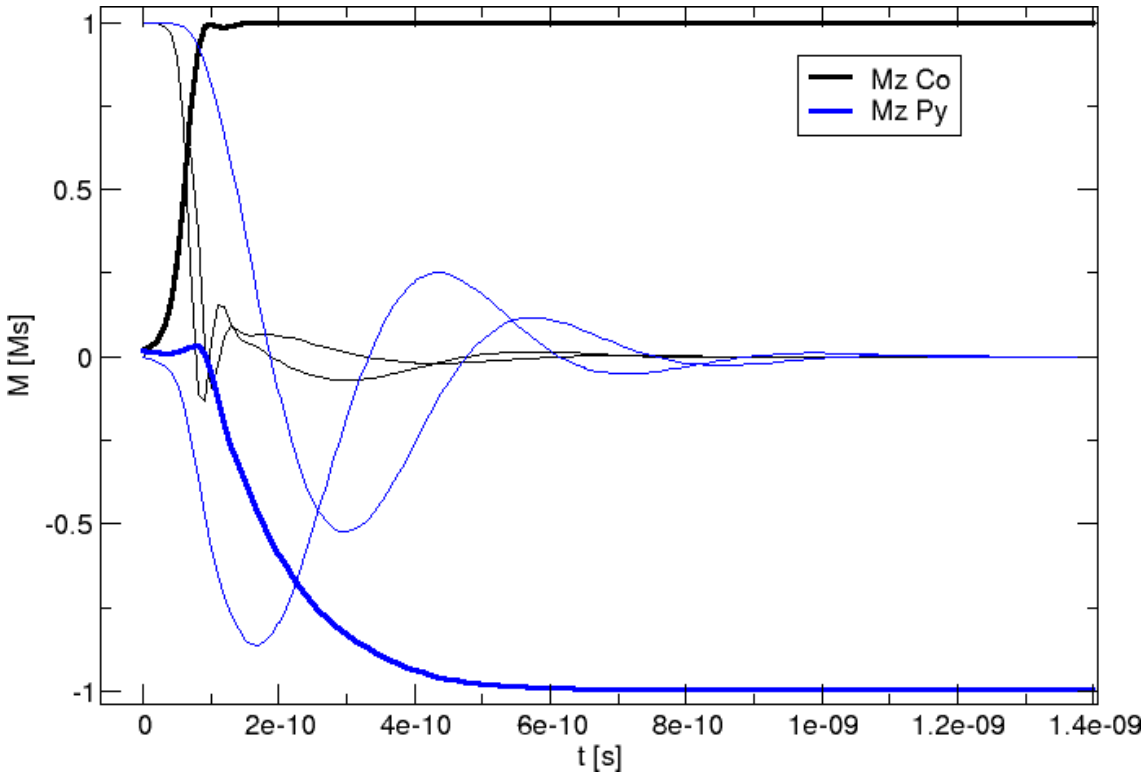
Field	Subfield(s)	Comment
m	m_Py, m_Co	normalised magnetisation
M	M_Py, M_Co	magnetisation
H_total	H_total_Py, H_total_Co	total effective field
H_ext	H_ext	external (applied) field (only one)
E_ext	E_ext_Py, E_ext_Co	energy density of Py due to external field
H_anis	H_anis_Py, H_anis_Co	crystal anisotropy field
E_anis	E_anis_Py, E_anis_Co	crystal anisotropy energy density
H_exch	H_exch_Py, H_exch_Co	exchange field
E_exch	E_exch_Py, E_exch_Co	exchange energy
H_demag	H_demag	demagnetisation field (only one)
E_demag	E_demag_Py, E_demag_Co	demagnetisation field energy density
phi	phi	scalar potential for H_demag
rho	rho	magnetic charge density (div M)
H_total	H_total_Py, H_total_Co	total effective field

The issue of multiple magnetic components becomes much more interesting when we study multi-component alloys, i.e. if we associate more than one type of magnetisation to a single region in the mesh. Usually, we will then also have to introduce some "generalized anisotropy energy" term of the form `E=c*M_a*M_b` that depends on more than a single magnetisation subfield (see *More than one magnetic material, exchange coupled*).

Once we have run the simulation using:

```
nsim two_cubes.py
```

we can analyse the results. For example, we can plot the magnetisation of the two materials against time:



The blue lines represent the (soft) permalloy and the black lines show the (hard) cobalt. Each thick line denotes the z-component of the corresponding material.

This plot has been created with the following command:

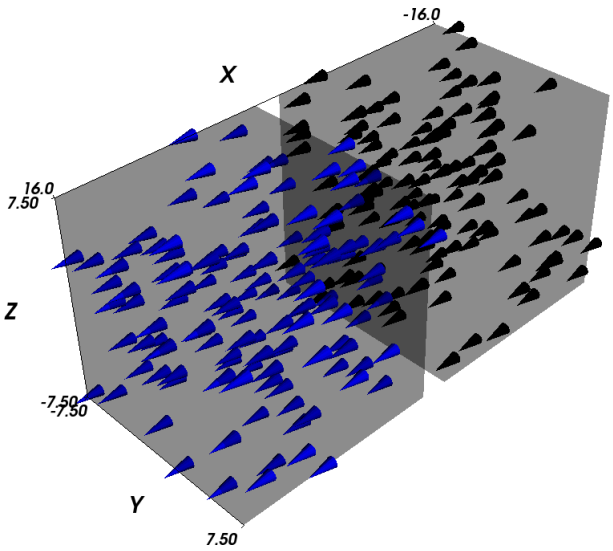
```
ncol two_cubes 0 m_Co_0 m_Co_1 m_Co_2 m_Py_0 m_Py_1 m_Py_2 | xmgrace -nxy -
```

We can further convert the field data into vtk files:

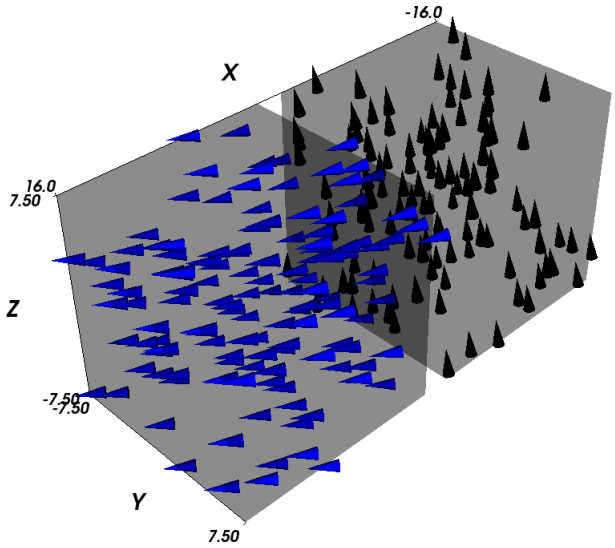
```
nmagpp --vtk=two_cubes.vtk two_cubes_dat.h5
```

and visualise their content. We start with the initial configuration (Permalloy in blue on the left, Cobalt in black on the right, only 10 percent of the actual magnetisation vectors on the mesh nodes are shown to improve the readability of the plots):

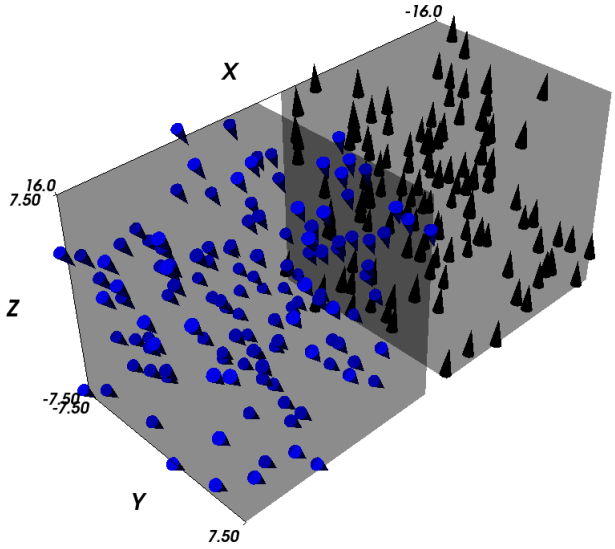
Time T=0 ps:



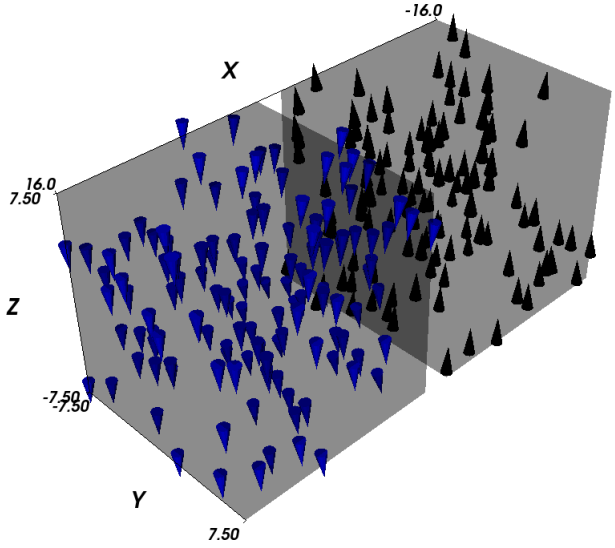
Time T=1e-10s=0.1ns: Cobalt is already pointing up, i.e. in the direction of the anisotropy axis, while Permalloy has just started to rotate.



Time T=0.3ns: Cobalt has reached its final configuration (pointing up) and Permalloy is still rotating, but already pointing down (to minimise the interaction energy between the cubes to the demagnetisation stray fields).



Time T=1 ns: The final configuration has been reached.



2.16 Example: Larmor precession

This example shows how to derive the period of the Larmor precession for the magnetisation and compare the result from simulation to the analytical solution. It is inspired by an example from the magpar documentation (http://magnet.atp.tuwien.ac.at/scholz/magpar/doc/html/examples.html#sphere_larmor).

We us the `larmor.py` script:

```
import nmag
from nmag import SI, every, at, si


#create simulation object and switch off
#the computation of the demagnetising field
sim = nmag.Simulation(do_demag = False)

# define magnetic material so that Js = mu0*Ms = 1 T
Py = nmag.MagMaterial(name="Py",
                      Ms=1.0*si.Tesla/si.mu0,
                      exchange_coupling=SI(13.0e-12, "J/m"),
                      llg_damping = SI(0.0)
                      )
# load mesh
sim.load_mesh("sphere1.nmesh.h5",
              [("sphere", Py)],
              unit_length=SI(1e-9,"m")
             )

# set initial magnetisation
sim.set_m([1,1,1])

# set external field
Hs = nmag.vector_set(direction=[0.,0.,1.],
                     norm_list=[1.0],
                     units=1e6*SI('A/m')
                    )

ps = SI(1e-12, "s")  # ps corresponds to one picosecond

# let the magnetisation precess around the direction of the applied field
sim.hysteresis(Hs,
               save=[('averages', every('time', 0.1*ps))],
               do=[('exit', at('time', 300*ps))])
```

We turn off computation of the demagnetising field:

```
sim = nmag.Simulation(do_demag = False)
```

and set the damping term in the LLG equation to zero:

```
llg_damping = SI(0.0)
```

We set saturation magnetisation to Js = 1 T (see *Library of useful si constants*):

```
Ms=1.0*si.Tesla/si.mu0
```

We use a sphere as the magnetic object and, starting from a uniform magnetic configuration along the [1,1,1] direction:

```
sim.set_m([1,1,1])
```

To compute the time development in the presence of a static field pointing in the z-direction, we ''abuse" the hysteresis command (because this way we can conveniently save the data at equidistant time intervals). To do this, we need to find the sequence of applied fields (here it is only one, of course):

```
Hs = nmag.vector_set( direction=[0.,0.,1.],
                      norm_list=[1.0],
                      units=1e6*SI('A/m')
                     )
```

and then use the hysteresis command:

```
sim.hysteresis(Hs,
               save=[('averages', every('time', 0.1*ps))],
               do=[('exit', at('time', 300*ps))])
```

The *hysteresis* command will save the averages (which is what we need to for the fit below) every 0.1 pico seconds. Once we reach the time of 300 pico seconds, the method will exit.

The dynamics of the magnetisation is driven only by the Zeeman effect, with a torque:

$$\mathbf{T} = \mu_0 \mathbf{m} \times \mathbf{H}_{\text{ext}}$$

acting on the magnetisation m which is orthogonal to both m and H; thus causing the magnetisation to precess around the applied field direction. The frequency of the precession, called f_Larmor, is given by:

$$f_{\text{Larmor}} = \frac{\gamma}{2\pi} \cdot \mu_0 \left|\mathbf{H}_{\text{ext}}\right|$$

where the parameter gamma, called gyromagnetic ratio, is taken to have the following value (see [2]):

$$\gamma = \frac{g \cdot e}{2m_e} \approx 1.7588 \times 10^{11}\,\mathrm{T}^{-1}\mathrm{s}^{-1}$$

so that f_Larmor = 35.176 GHz and the period T = 1/f_Larmor = 0.0284284 ns.

We save the average magnetisation every 0.1 ps in order to have a sufficient number of points to compute the period T.

We execute the script as usual:

```
$ nsim larmor.py
```

and extract the (spatially averaged) magnetisation data for all save time steps:

```
$ ncol larmor time m_Py_0 m_Py_1 m_Py_2 > data.txt
```

Using Gnuplot, we extract the value of the Larmor period T from the x-component of the magnetisation:

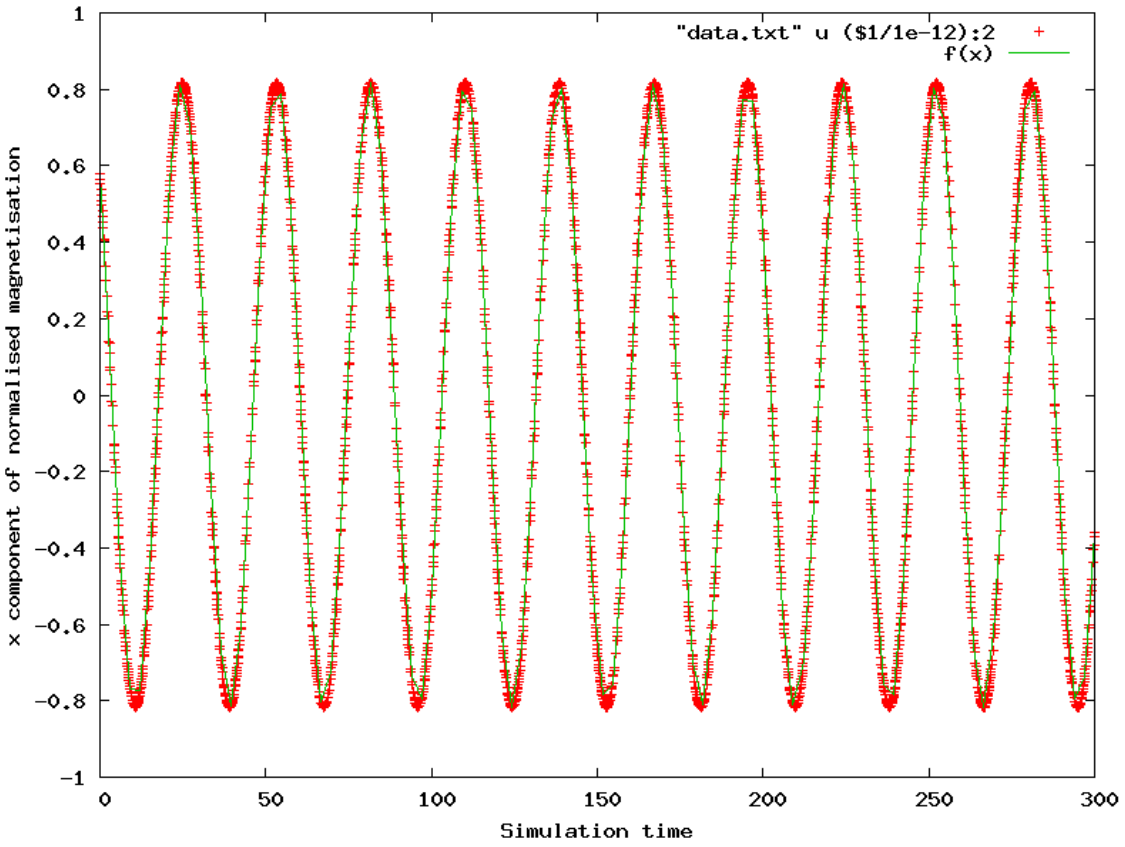
```
$ gnuplot
```

and the following command plots the x component of the magnetisation as a function of the simulation time, together with a fit for a function `f(x)` (where `x` represents time):

```
gnuplot> f(x) = A*sin(2*pi*x/B + C) + D
gnuplot> B = 30
gnuplot> fit f(x) "data.txt" u  ($1/1e-12):2 via A,B,C,D
gnuplot> plot "data.txt" u ($1/1e-12):2, f(x)
```

The result is the following image:

[2] See the OOMMF manual, and in Werner Scholz thesis, after (3.7), llg_gamma_G = 2.210173e5 m/(As).



The values for B in the fit, which corresponds to the unknown period T, is initially set to 30 in order to help Gnuplot fit the curve. Such fit on T gives the value 28.4293; this value corresponds to 0.0284293 ns when rescaled by the 10e12 factor used for the plotting, and shows a difference starting from the 5th digit when compared to the analytical solution of 0.0284284 ns.

2.17 Example: 1D periodicity

2.17.1 Introduction periodic boundary conditions ("macro geometry")

Concerning the simulation of periodic magnetic structures, there are a few somewhat subtle issues to be taken into account, both with respect to the demagnetising and the exchange field.

The issue with the exchange field is that we may encounter situations where the magnetic material crosses the boundary of an elementary cell: a periodic array of non-touching spheres in a cubic lattice is fundamentally different from its complement, a cubic lattice made of spherical holes, insofar as that in the latter case, it is impossible to do a simulation using periodic boundary conditions without identifying degrees of freedom that live on boundaries of the simulation cell. Nmag can deal with this automatically, provided the mesh file contains periodicity information, i.e. data on how to identify nodes on exterior faces.

As for the demagnetising field, the most important problem is that one cannot ignore the effect of the faraway boundaries of the system: a 100 nm x 100 nm x 100 nm cell made of magnetic material in the center of a large (three-dimensional) periodic array will experience very different demagnetising fields depending on the shape of the outer boundaries of this array. Assuming spatially constant magnetisation, if these cells form a "macroscopic" (tree-dimensional) sphere, H_demag will be -1/3 M, while for a flat box, H_demag may be very close to -M. Nmag takes these "macro-geometry" effects into account by allowing the user to provide a geometrical layout for a finite number (say, 100-1000) of cells that approximates the shape of the faraway outer boundary of the system.

The macro geometry approach is described in [3] which may serve as a more detailed instruction to the concept.

[3] Hans Fangohr, Giuliano Bordignon, Matteo Franchin, Andreas Knittel, Peter A. J. de Groot, Thomas Fischbacher. *A new approach to (quasi) periodic boundary conditions in micromagnetics: the macro geometry*, Journal of Applied Physics **105**, 07D529 (2009), Online at

2.17.2 1d example

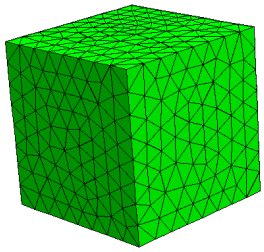
In this example, we simulate a single cell in the middle of a long one-dimensional periodic array where for the purpose of computing the demagnetising field, we take three extra copies of this cell to the left and three copies to the right along the x axis. (For real applications, one would use more copies. The only effect of additional copies are to increase the setup time needed to compute an internal boundary/boundary interaction matrix.)

The next *Example: 2D periodicity* demonstrates the macro geometry concept for a thin film. This is followed by the *Spin-waves example* which includes exchange coupling between periodic copies (and is of more practical value).

The mesh of the central simulation cell used is described in `cube.geo` which reads:

```
algebraic3d

# prism
solid prism = orthobrick (-7.50, -7.50, -7.50; 7.50, 7.50, 7.50) -maxh = 1.8000;
tlo prism;
```

Note that the mesh is centered around the origin. This is recommended for periodic simulations. (We need to document this better.) The resulting mesh is this (the periodic copies are not shown):



The script `periodic1.py` reads:

```
import nmag
from nmag import SI

# define magnetic material
Py = nmag.MagMaterial(name="Py",
                      Ms=SI(1e6,"A/m"),
                      exchange_coupling=SI(13.0e-12, "J/m")
                      )

# size of simulation cell, plus extra spacing
# to avoid exchange interaction across interfaces
# between repeated copies of the simulation cell.
x_lattice = 15.01  # the spacing is 0.01
y_lattice = 0.0
z_lattice = 0.0


# list to store the lattice points where the periodic
# copies will be placed
lattice_points = []

for xi in range(-3,4):
    lattice_points.append([xi*x_lattice,0.0*y_lattice,0.0*z_lattice])

#  create data structure pbc for this macro geometry
pbc = nmag.SetLatticePoints(vectorlist=lattice_points, scalefactor=SI(1e-9,'m'))

#create simulation object,  passing macro geometry data structure
```

http://link.aip.org/link/?JAP/105/07D529

```
sim = nmag.Simulation(periodic_bc=pbc.structure)

# load mesh
sim.load_mesh("cube1.nmesh.h5", [("repeated-cube-1D", Py)], unit_length=SI(1e-9,"m") )

# set initial magnetisation along the periodic axis
sim.set_m([1.0,0,0])

# compute the demagnetising field
sim.advance_time(SI(0,"s"))

# probe demag field at the centre of the cube, function
# returns an SI-Value ('siv')
H_demag = sim.probe_subfield_siv('H_demag', [0,0,0])

print "H_demag_x at centre of cube = ", H_demag[0]
print "H_demag_y at centre of cube = ", H_demag[1]
print "H_demag_z at centre of cube = ", H_demag[2]
```

Setup can be splitted into three steps. In the first step we set the x_lattice parameter to be slightly larger than the dimension of the unit cell (in order not to have any overlap between the cells) and set the y_lattice and z_lattice parameters to zero to indicate no periodidicity along these directions

```
x_lattice = 15.01 # the spacing is 0.01
y_lattice = 0.0
z_lattice = 0.0
```

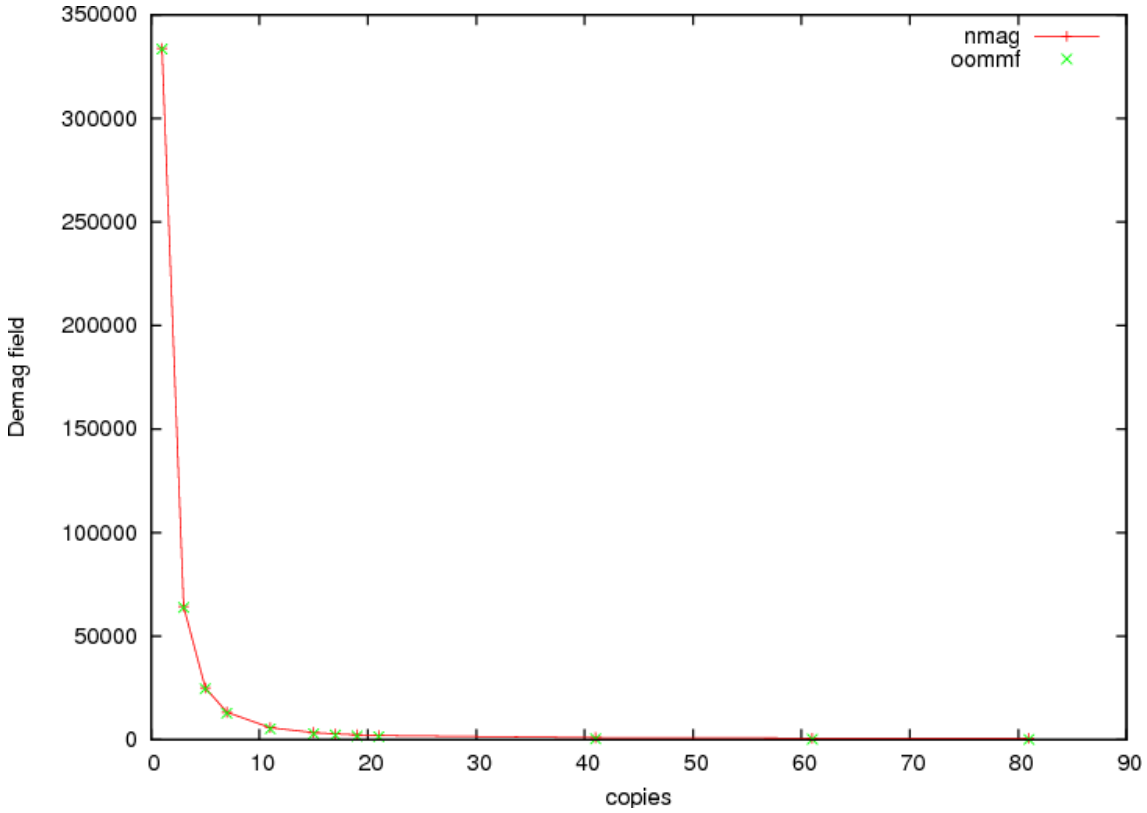
In the second step we define the lattice points where we want the periodic copies to be:

```
for xi in range(-3,4):
    lattice_points.append([xi*x_lattice,0.0*y_lattice,0.0*z_lattice])
```

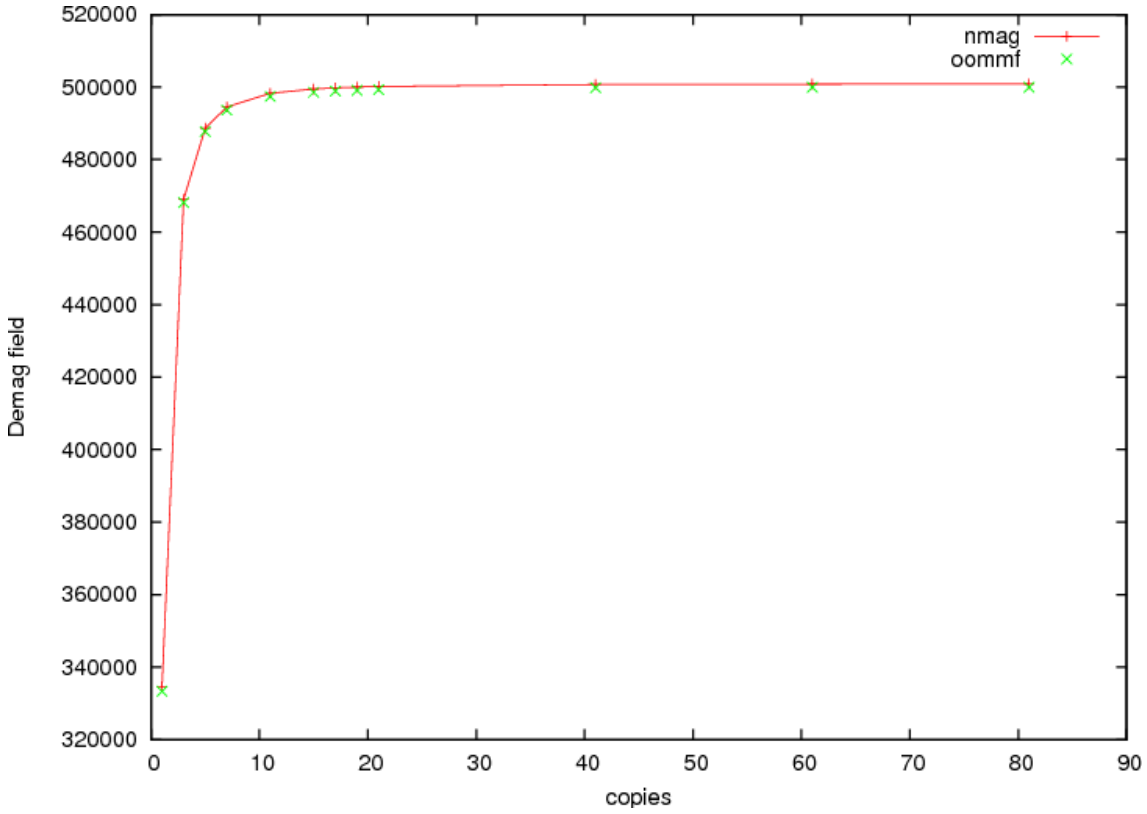
and in the third step we define the object whose structure attribute will be used as the parameter in the definition of the simulation object

```
pbc = nmag.SetLatticePoints(vectorlist=lattice_points, scalefactor=SI(1e-9,'m'))

#create simulation object
sim = nmag.Simulation(periodic_bc=pbc.structure)
```

The remaining part of the script computes the demagnetisation field at the center of the cube. This calculation can be carried out for a varying number of copies of the simulation cell. The next figures show components of demagnetising field in the center of the cube as a function of the number of periodic copies. As in the code above, we impose an uniform magnetisation along the periodic x-axis. The first figure shows the demagnetisation field along the x-axis, and the second figure along the y-axis. In both figures, we have added green crosses that have been obtained by computing the demagfield using OOMMF (where in OOMMF we have actually made the simulation cell larger and larger to represent the growing number of periodic copies).



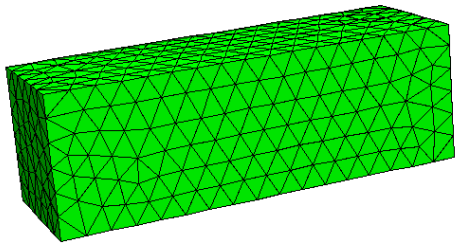
Demagnetising field as a function of the number of periodic copies with the magnetisation aligned along the periodic axis.



Demagnetising field as a function of the number of periodic copies with the magnetisation aligned along an axis orthogonal to the periodic one.

2.18 Example: 2D periodicity

This example is another application of the macro-geometry feature, where we now deal with a 2D "thin film" system. The unit cell is a `30x10x10 nm^3` prism



where we take 10 copies in x- and 40 copies in y-direction to create the macro geometry.

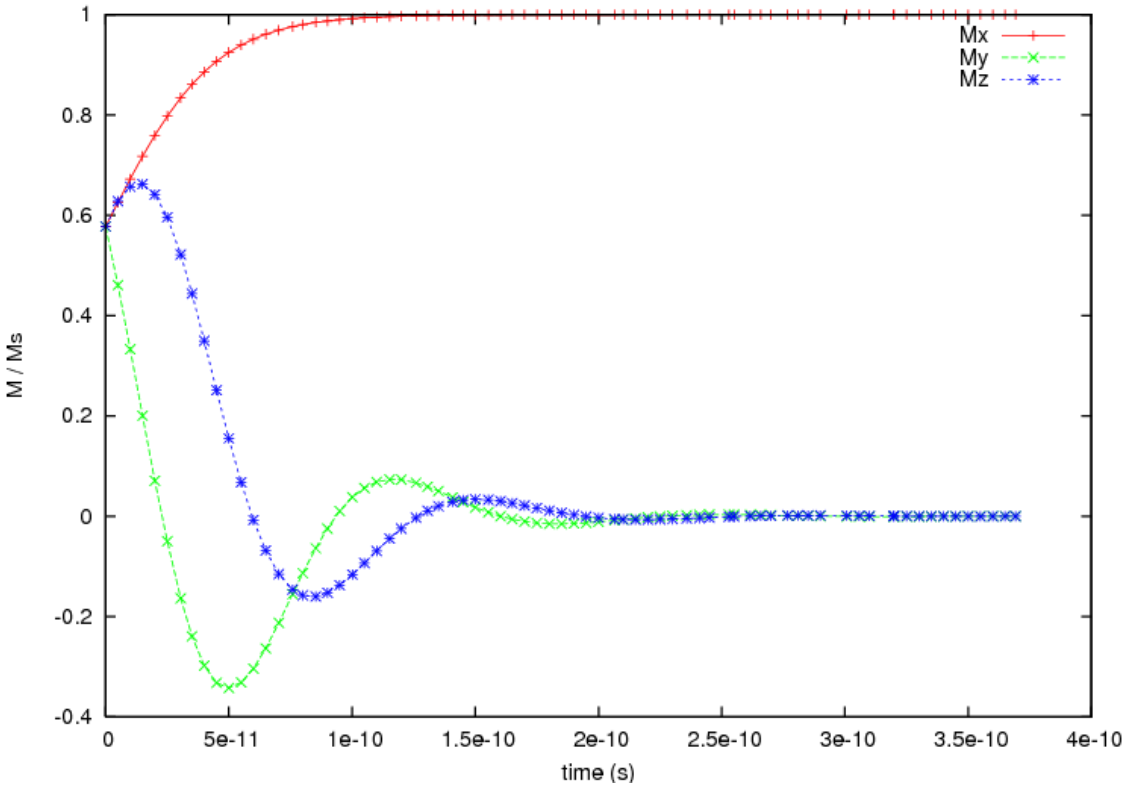
The script `no_periodic.py` simulates behaviour of just the unit cell of size 30x10x10 nm^3 (without any periodic copies):

```
import nmag
from nmag import SI, every, at

# define magnetic material
Py = nmag.MagMaterial(name="Py",
                      Ms=SI(1e6,"A/m"),
                      exchange_coupling=SI(13.0e-12, "J/m")
                      )

#create simulation object
sim = nmag.Simulation()

# load mesh
sim.load_mesh("prism.nmesh.h5", [("no-periodic", Py)], unit_length=SI(1e-9,"m") )

# set initial magnetisation
sim.set_m([1.,1.,1.])

# loop over the applied field
s = SI(1,"s")

sim.relax(save=[('averages','fields', every('time',5e-12*s) | at('convergence'))])
```

and the relaxation curves are obtained via:

```
set term postscript enhanced color
set out 'no_periodic.ps'
set xlabel 'time (s)'
set ylabel 'M / Ms'
plot 'plot_no_periodic.dat' u 1:2 ti 'Mx' w lp, \
     'plot_no_periodic.dat' u 1:3 ti 'My' w lp, \
     'plot_no_periodic.dat' u 1:4 ti $
```

which creates the following plot:



From this plot we can see that with using only the unit cell the magnetisation aligns along the x-axis at equilibrium.

We now move to the macro geometry of a thin film with dimensions 400x300x10nm^3 which is realised in `periodic2.py`.

```
import nmag
from nmag import SI, every, at

# define magnetic material
Py = nmag.MagMaterial(name="Py",
                      Ms=SI(1e6,"A/m"),
                      exchange_coupling=SI(13.0e-12, "J/m")
                      )

# size of the simulation cell, plus extra spacing
x_lattice = 30.01  # the spacing is 0.01 to avoid exchange coupling
y_lattice = 10.01  # between repeated copies of the simualtion cell
z_lattice = 0.0


# list to store the lattice points where the periodic
# copies of the simulation cell will be placed
lattice_points = []

for xi in range(-4,6):
    for yi in range(-19,21):
        lattice_points.append([xi*x_lattice,yi*y_lattice,0.0*z_lattice])

# create data structure pbc for this macro geometry
pbc = nmag.SetLatticePoints(vectorlist=lattice_points, scalefactor=SI(1e-9,'m'))

#create simulation object, passing macro geometry data structure
sim = nmag.Simulation(periodic_bc=pbc.structure)


# load mesh
```

```
sim.load_mesh("prism.nmesh.h5", [("repeated-prism-2D", Py)], unit_length=SI(1e-9,"m") )

# set initial magnetisation
sim.set_m([1.,1.,1.])

# loop over the applied field
s = SI(1,"s")
sim.relax(save=[('averages', 'fields', every('time',5e-12*s) | at('convergence'))])
```

As in the previous example, we first define the three unit vectors of the lattice, again slightly larger than the dimension of the unit cell to avoid overlapping (and thus to eleminate any exchange coupling across the interfaces for this demonstration of the demagnetisation effects):

```
x_lattice = 30.01  # the spacing is 0.01
y_lattice = 10.01  # the spacing is 0.01
z_lattice = 0.0
```

Then we define where the copies will be placed:

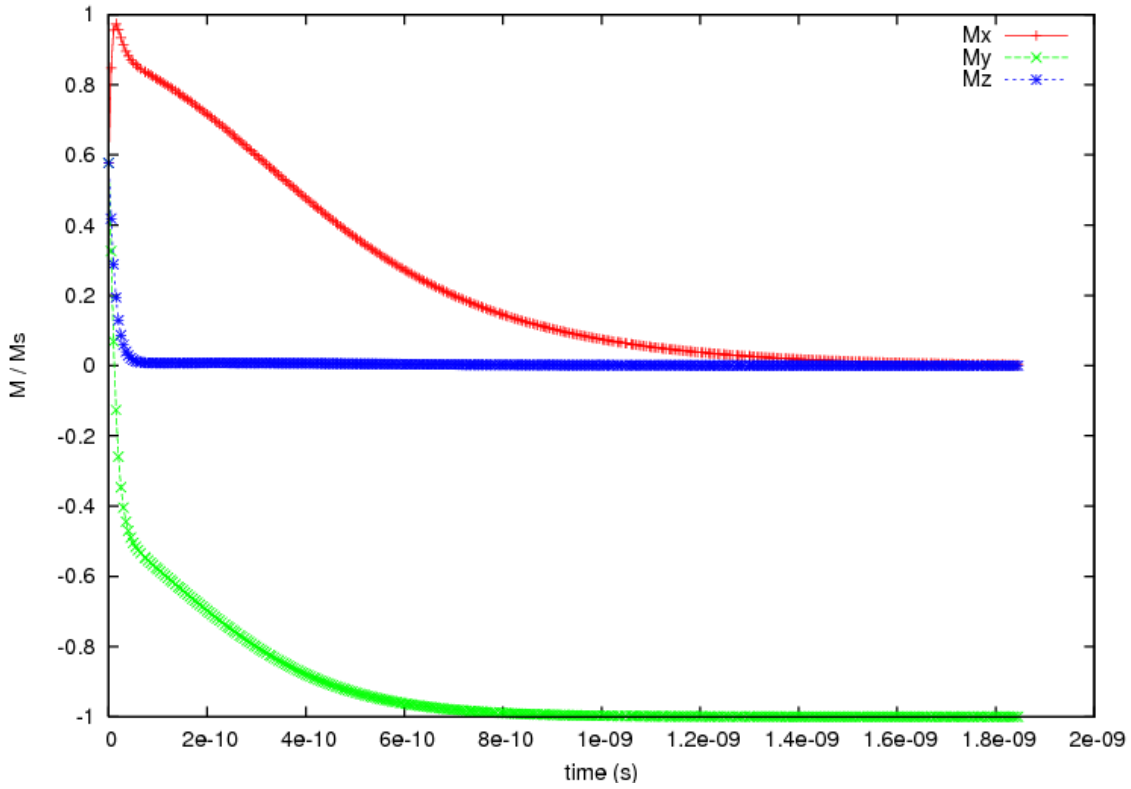
```
for xi in range(-4,6):
    for yi in range(-19,21):
        lattice_points.append([xi*x_lattice,yi*y_lattice,0.0*z_lattice])

# copies of the system along the x-axis
pbc = nmag.SetLatticePoints(vectorlist=lattice_points, scalefactor=SI(1e-9,'m'))
```

The simulation cell is (always) the one at the (0,0,0) lattice point. The for loops therefore place 4 copies of the simulation cell in the negative x direction [i.e. (-4,0,0), (-3,0,0), (-2,0,0), and (-1,0,0)] and 5 in the positive the x direction [i.e. (1,0,0), (2,0,0), (3,0,0), (4,0,0), (5,0,0)]. The translation vector (0,0,0) corresponds to the actual simulation cell.

Similarly, the inner for loop places 20 copies along the positive y-axis and 19 along the negative one.

We set the same initial configuration as before, with a uniform magnetisation along [1,1,1], and let the system evolve towards the equilibrium.

The outcome is shown in the following figure:



where we notice that the final configuration is now with the magnetisation aligned along the (negative) y axis, and not along the x axis as before. The alignment along the y-direction is expected, as now the macro geometry has a total size of 300.09 nm times 400.39 nm (30 nm x 10 copies plus spacings along the x direction times 10 nm x 40 copies plus spacings along the y direction) times 10nm (no periodic copies along the z direction), so the longest side now is along the y direction. The demagnetisation energy of the macro geometry drives the alignment of the magnetisation with the y-direction.

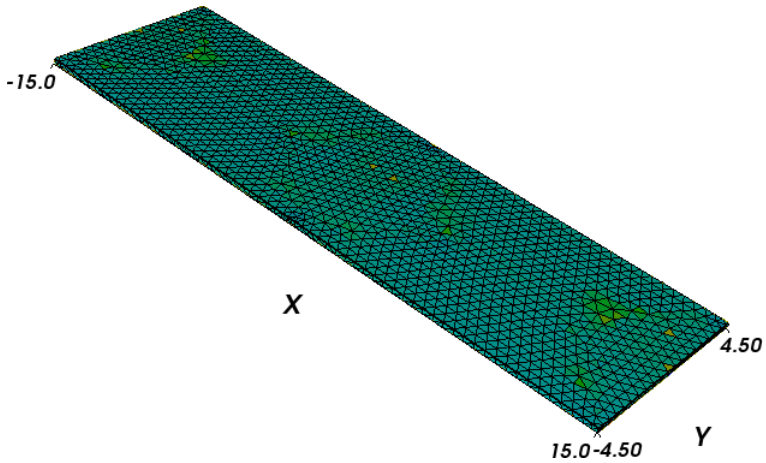
Other usage examples include this study [4] of an array of interacting triangular rings.

2.19 Example: Spin-waves in periodic system

Starting from a magnetisation out of equilibrium, we study the time development of the magnetisation, and track -visually- the spin waves.

The geometry is a thin film with dimensions 30 nm x 9 nm x 0.2 nm along the x,y and z axes, respectively. The mesh is centered at (0,0,0) and periodic along the x direction, so that the nodes with coordinates (15.0,y,z) will be considered as equivalent to the nodes with coordinates (-15.0,y,z).

[4] Giuliano Bordignon, Thomas Fischbacher, Matteo Franchin, Jurgen P. Zimmermann, Peter A. J. de Groot, Hans Fangohr, *Numerical studies of demagnetizing effects in triangular ring arrays*, Journal of Applied Physics **103** 07D932 (2008), online at http://eprints.soton.ac.uk/50995/



The mesh is contained in `periodic.nmesh` and has been produced using Netgen (from `periodic.geo`) and the *nmeshmirror* command to create required periodic structure

```
$ nmeshmirror netgen.nmesh 1e-6 1e-6 -1,0,0 periodic.nmesh
```

2.19.1 Relaxation script

To see how the system relaxes, we use the following script (`spinwaves.py`):

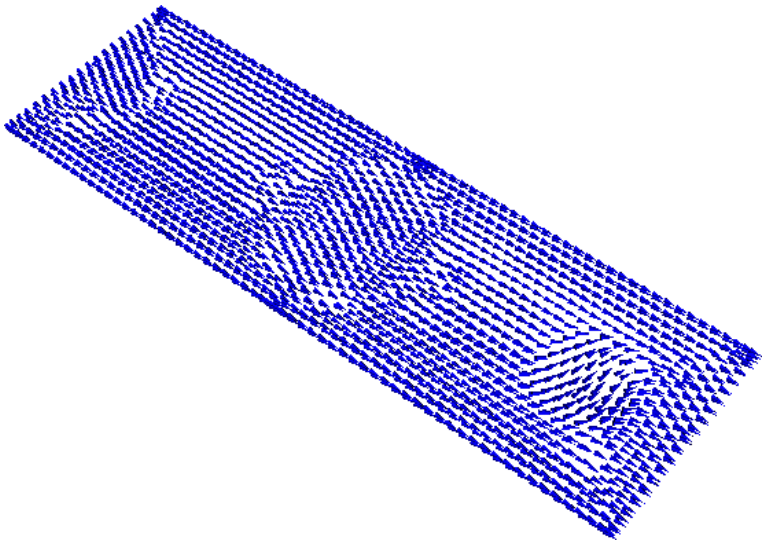
```
import nmag
from nmag import SI
import math

# define magnetic material
Py = nmag.MagMaterial(name="Py",
                      Ms=SI(1e6,"A/m"),
                      exchange_coupling=SI(13.0e-12, "J/m"),
                      llg_damping = SI(0.02,"")
                      )

# lattice spacings along the main axes;
# the value must be zero for no periodic copies,
# equal to the mesh dimension along the
# given axis otherwise
x_lattice = 30.0
y_lattice = 0.0
z_lattice = 0.0


# list to store the lattice points where the periodic
# copies will be placed
lattice_points = []

for xi in range(-1,2):
    lattice_points.append([xi*x_lattice,0.0*y_lattice,0.0*z_lattice])

# copies of the system along the x-axis
pbc = nmag.SetLatticePoints(vectorlist=lattice_points, scalefactor=SI(1e-9,'m'))

#create simulation object
sim = nmag.Simulation(periodic_bc=pbc.structure)

# load mesh
sim.load_mesh("periodic.nmesh", [("periodic-film", Py)], unit_length=SI(1e-9,"m") )

print ocaml.mesh_plotinfo_periodic_points_indices( sim.mesh.raw_mesh )
```

```
# function to set the magnetisation
def perturbed_magnetisation(pos):
    x,y,z = pos
    newx = x*1e9
    newy = y*1e9
    if 8<newx<14 and -3<newy<3:
        # the magnetisation is twisted a bit
        return [1.0, 5.*(math.cos(math.pi*((newx-11)/6.)))**3 *\
                        (math.cos(math.pi*(newy/6.)))**3, 0.0]
    else:
        return [1.0, 0.0, 0.0]


# set initial magnetisation
sim.set_m(perturbed_magnetisation)

# let the system relax generating spin waves
s = SI("s")
from nsim.when import every, at
sim.relax(save=[('averages','fields', every('time', 0.05e-12*s) | at('convergence'))],
          do=[('exit', at('time', 10e-12*s))])
```

To execute this script, we call the *nsim* executable, for example (on linux):

```
$ nsim spinwaves.py
```

As in the previous examples, we first need to import the modules necessary for the simulation, define the material of the magnetic object, load the mesh and set the initial configuration of the magnetisation. Here, we start from a spatially non-homogeneous configuration in order to excite spin waves. Nmag allows us to provide a function to be sampled on the mesh that defines a particular magnetisation configuration.



In our case, we use the function

```
def perturbed_magnetisation(pos):
    x,y,z = pos
    newx = x*1e9
    newy = y*1e9
    if 8<newx<14 and -3<newy<3:
        # the magnetisation is twisted a bit
        return [1.0, 5.*(math.cos(math.pi*((newx-11)/6.)))**3 *\
                        (math.cos(math.pi*(newy/6.)))**3, 0.0]
    else:
        return [1.0, 0.0, 0.0]
```

which is then passed on to *set_m*

```
# set initial magnetisation
sim.set_m(perturbed_magnetisation)
```

2.19.2 Visualising the magnetisation evolution

Once the calculation has finished, we can see how the system relaxed by means of snapshots of the magnetisation evolution.

The *nmagpp* command allows us to create vtk files from the data saved with the `save` option in the `relax` method:

```
nmagpp --vtk=fields spinwaves
```

The first few frames that show the evolution of the magnetic configuration are shown below.

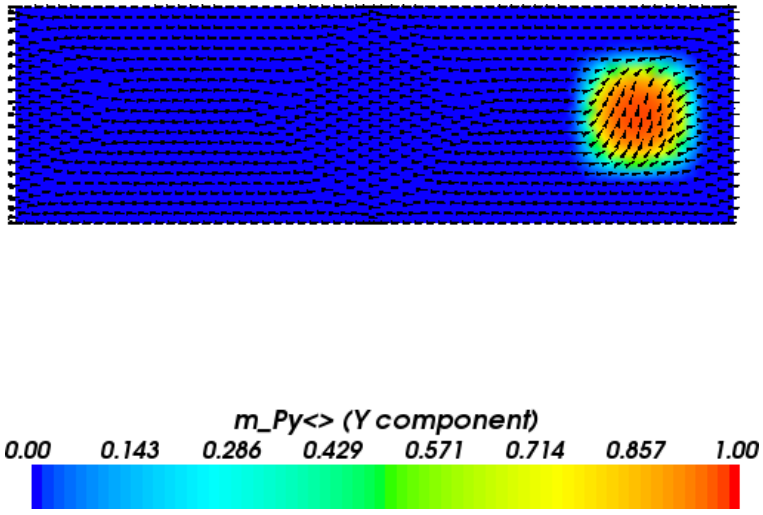


Figure 2.7: Initial magnetisation configuration.

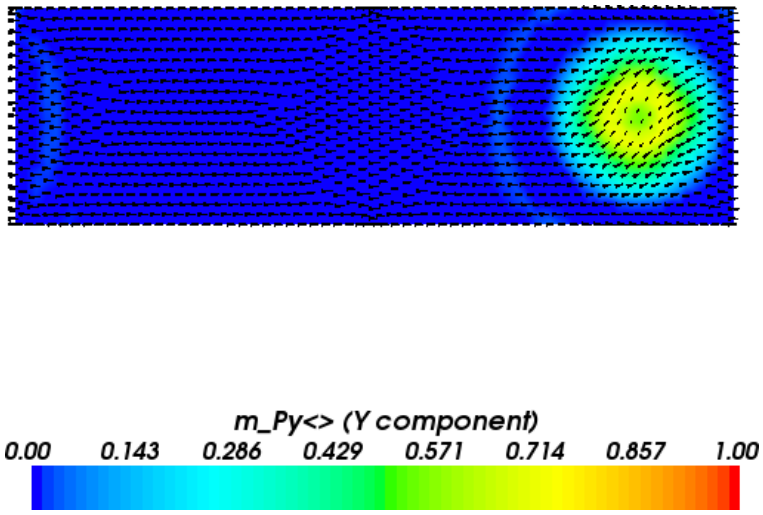


Figure 2.8: Magnetisation configuration after 0.15 ps. It is clearly visible that the spin waves travel from the center of the disturbance to the right and penetrate the system immediately from the left (due to the periodic boundary conditions in the x-direction).

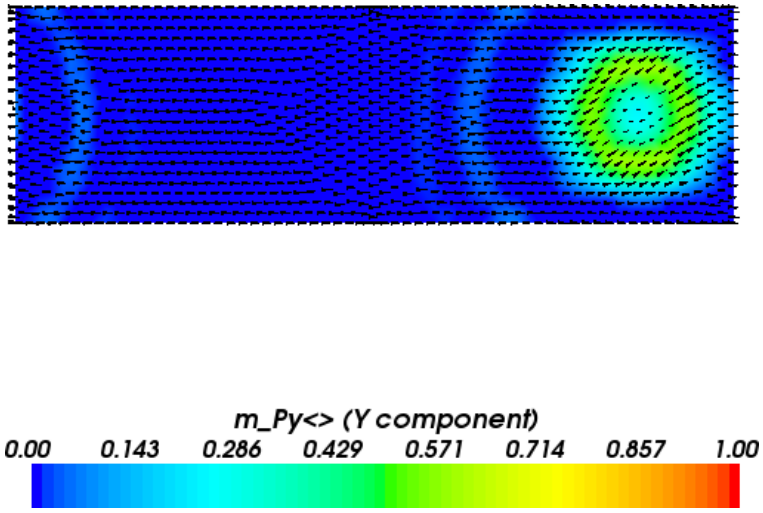


Figure 2.9: Magnetisation configuration after 0.25 ps.

2.20 Example: post processing of saved field data

Suppose we have saved spatially resolved fields (as, for example, in *Example 2: Computing the time development of a system*), and we would like to read those from the data file to process the data further.

We can use the *nmagpp* tool if it provides the required functionality.

Alternatively, we can write a Python script that:

1. reads the data from the `_dat.h5` file
2. carries out the required post processing and/or saves the data in (another) format.

The program `read_h5.py` demonstrates how to read the saved configuration with id=0 of the `m_Py` subfield, and to print this to the screen.

```
import nmag

#read data, positions, and sites from h5 file
m=nmag.get_subfield_from_h5file('bar_dat.h5','m_Py',id=0)
pos=nmag.get_subfield_positions_from_h5file('bar_dat.h5','m_Py')
site=nmag.get_subfield_sites_from_h5file('bar_dat.h5','m_Py')

#can carry out some sanity checks (but is not necessary)
assert m.shape == pos.shape
assert len(m) == len(site)

#print the data
for i in range(len(m)):
    print site[i], pos[i], m[i]
```

The functions *get_subfield_from_h5file*, *get_subfield_positions_from_h5file* and *get_subfield_sites_from_h5file* allow in principle to retrieve all the field data from the h5 files and stores this in the variables `m`, `pos`, and `site`, respectively.

The program, when run like this:

```
$ nsim read_h5.py
```

in the *Example 2: Computing the time development of a system* directory, produces output starting as follows (assuming the `bar_dat.h5` file exists):

```
[0] [ 0.  0.  0.] [ 0.70710677  0.          0.70710677]
[1] [  3.00000000e-09   0.00000000e+00   0.00000000e+00] [ 0.70710677  0.          0.70710677]
[2] [  6.00000000e-09   0.00000000e+00   0.00000000e+00] [ 0.70710677  0.          0.70710677]
[3] [  9.00000000e-09   0.00000000e+00   0.00000000e+00] [ 0.70710677  0.          0.70710677]
[4] [  1.20000000e-08   0.00000000e+00   0.00000000e+00] [ 0.70710677  0.          0.70710677]
[5] [  1.50000000e-08   0.00000000e+00   0.00000000e+00] [ 0.70710677  0.          0.70710677]
[6] [  1.80000000e-08   0.00000000e+00   0.00000000e+00] [ 0.70710677  0.          0.70710677]
[7] [  2.10000000e-08   0.00000000e+00   0.00000000e+00] [ 0.70710677  0.          0.70710677]
[8] [  2.40000000e-08   0.00000000e+00   0.00000000e+00] [ 0.70710677  0.          0.70710677]
[9] [  2.70000000e-08   0.00000000e+00   0.00000000e+00] [ 0.70710677  0.          0.70710677]
[10] [  3.00000000e-08   0.00000000e+00   0.00000000e+00] [ 0.70710677  0.          0.70710677]
[11] [  3.00000000e-08   3.00000000e-08   1.00000000e-07] [ 0.70710677  0.          0.70710677]
[12] [  3.00000000e-08   2.70000000e-08   1.00000000e-07] [ 0.70710677  0.          0.70710677]
```

We can see that the *Site* index is (here) just an integer, the position (in nanometre) is shown as a triplet of three scalars, and the normalised magnetisation is also a vector with three components.

The data (in the arrays `m`, `site` and `position` in this example) can be manipulated as explained in the *NumPy* documentation, because it is of type `numpy array`. Numpy provides a powerful matrix processing environment.

2.21 Example: Spin transfer torque (Zhang-Li model)

Nmag provides support for the Zhang-Li extension to the Landau-Lifshitz-Gilbert (LLG) equation [5], in order to model the interaction between a uniform electric current density and a spatially varying magnetisation. The extened LLG equation reads

$$\frac{\partial \vec{M}}{\partial t} = -\gamma\, \vec{M} \times \vec{H} + \frac{\alpha}{M_{\mathrm{sat}}}\, \vec{M} \times \frac{\partial \vec{M}}{\partial t} - \frac{v}{M_{\mathrm{sat}}^2}\, \vec{M} \times \left(\vec{M} \times \hat{j} \cdot \nabla \vec{M}\right) - \frac{\xi v}{M_{\mathrm{sat}}}\, \vec{M} \times \hat{j} \cdot \nabla \vec{M}$$

where the first two terms on the right-hand side are the normal LLG equation, and the extra terms come from the Zhang-Li model, and

$\vec{M}$	`Simulation.set_m`	is the magnetisation,
$\vec{H}$	`Simulation.set_H_ext`	is the effective magnetic field,
M_{sat}	`Ms`	is the saturation magnetisation,
γ	`llg_gamma_G`	is the gyromagnetic ratio,
α	`llg_damping`	is the damping parameter,
$\hat{j} \cdot \nabla$		derivative along $\hat{j}$, the direction of the current,

The central column shows the method which can be used to set the field (`Simulation.set_m` or `Simulation.set_H_ext`) or the name of the corresponding parameter in the material definition (for example, `mat = MagMaterial(Ms=SI(0.8e6, "A/m"), ...)`). The current density appears only throughout the quantity `v`, which we define as:

$$v = \frac{P\, j\, \mu_B}{e\, M_{\mathrm{sat}}\, (1 + \xi^2)}$$

with:

j	`Simulation.set_current_density`	is the norm of the current density,
P	`llg_polarisation`	is the degree of polarization of the spin current,
ξ	`llg_xi`	is the degree of non adiabadicity.
μ_B		is the Bohr magneton,
e		is the electron charge, e is positive : e = |e|

[5] S. Zhang and Z. Li, *Roles of Nonequilibrium Conduction Electrons on the Magnetization Dynamics of Ferromagnets*, Physical Review Letters **93**, 127204 (2004), online at http://link.aps.org/doi/10.1103/PhysRevLett.93.127204

In this and in the next examples we show how to set up a micromagnetic simulation including such spin transfer torque effects. We show how the current density can be specified and how the required parameters can be included in the material definitions.

As a first example, we consider a thin Permalloy film which develops a vortex in the center. We compute the dynamics of the vortex as a response to the application of a current.

2.21.1 Current-driven motion of a vortex in a thin film

The system under investigation is a 100 x 100 x 10 nm Permalloy film. The mesh is stored in the file `pyfilm.nmesh.h5`.

The simulation is subdivided in two parts:

- In part I, the system is relaxed to obtain the initial magnetisation configuration when the current **is not** applied, which is just a vortex in the center of the film.
- In part II, the vortex magnetisation obtained in part I is loaded and used as the initial magnetisation configuration. A current is applied and the magnetisation dynamics is analysed by saving periodically the data (the magnetisation, the other fields and their averages).

Here we use two separate simulation scripts to carry out part I and part II subsequentely. This is the approach that is easiest to understand. Once the basic ideas have become clear, it is often a good idea to write only one simulation script that carries out both part I and part II. (Indeed many of the parameters, such as the saturation magnetisation or the exchange coupling need to be specified in each of the two scripts leading to possible errors: for example if one decides to investigate a different material and changes the parameters just in one file and forgets the other, etc.). In the next section (*Example: Current-driven magnetisation precession in nanopillars*), we present a more robust approach, where both part I and part II are executed by just one script.

2.21.2 Part I: Relaxation

The first script carries out a normal micromagnetic simulation (i.e. no spin transfer torque), and determines the relaxed magnetisation configuration for a given geometry, material and initial configuration. It saves the final magnetisation to disk. Here is the full listing of `relaxation.py` :

```
# We model a bar 100 nm x 100 nm x 10 nm where a vortex sits in the center.
# This is part I: we just do a relaxation to obtain the shape of the vortex.
import math, nmag
from nmag import SI, at
from nsim.si_units.si import degrees_per_ns

# Define the material
mat_Py = nmag.MagMaterial(name="Py",
                          Ms=SI(0.8e6,"A/m"),
                          exchange_coupling=SI(13.0e-12, "J/m"),
                          llg_gamma_G=SI(0.2211e6, "m/A s"),
                          llg_damping=1.0)

# Define the simulation object and load the mesh
sim = nmag.Simulation()
sim.load_mesh("pyfilm.nmesh.h5", [("Py", mat_Py)], unit_length=SI(1e-9,"m"))

# Set a initial magnetisation which will relax into a vortex
def initial_m(p):
  x, y, z = p
  return [-(y-50.0e-9), (x-50.0e-9), 40.0e-9]

sim.set_m(initial_m)

# Set convergence parameters and run the simulation
sim.set_params(stopping_dm_dt=1.0*degrees_per_ns)
```

```
sim.relax(save=[('fields', at('step', 0) | at('stage_end'))])

# Write the final magnetisation to file "vortex_m.h5"
sim.save_restart_file("vortex_m.h5")
```

After importing the usual Nmag module and helper objects, we define the material, create the simulation object and load the mesh, (similar to what is shown in previous examples):

```
# Define the material
mat_Py = nmag.MagMaterial(name="Py",
                          Ms=SI(0.8e6,"A/m"),
                          exchange_coupling=SI(13.0e-12, "J/m"),
                          llg_gamma_G=SI(0.2211e6, "m/A s"),
                          llg_damping=1.0)

# Define the simulation object and load the mesh
sim = nmag.Simulation()
sim.load_mesh("pyfilm.nmesh.h5", [("Py", mat_Py)], unit_length=SI(1e-9,"m"))
```

Notice that the damping parameter `llg_damping` is set to a high value, to allow for quick relaxation of the magnetisation. We write a function `initial_m` that is being given the position of each site in the mesh as a vector `p` with three components, and which returns an initial magnetisation vector. This vector is chosen such that the initial magnetisation that is described by this function is likely to relax into a vortex configuration:

```
def initial_m(p):
  x, y, z = p
  return [-(y-50.0e-9), (x-50.0e-9), 40.0e-9]
```

The magnetisation at point `p` is obtained from a 90 degree rotation of the vector which connects `p` to the center of the film. This vector doesn't have to be normalised: Nmag will take care of normalising it for us.

We need to instruct the simulation object `sim` to use this function to set the magnetisation:

```
sim.set_m(initial_m)
```

We set the criterion to be used to decide when the magnetisation has relaxed. The value used here in *set_params* (i.e. one degree per nanosecond) is the default value (so we could omit this, but we change the value in the second part of this example):

```
sim.set_params(stopping_dm_dt=1.0*degrees_per_ns)
```

We finally run the simulation using the *relax* command until the convergence criterion dm/dt < 1 degree per nanosecond is fullfilled. In the process, we save spatially resolved data for all fields at step 0, and the same data at the end of the stage (i.e. when an equilibrium has been reached, just before the *relax* function returns):

```
sim.relax(save=[('fields', at('step', 0) | at('stage_end'))])
```

We finally save the relaxed magnetisation to a file using the function *save_restart_file*, so that we can use this in part 2 as the initial configuration:

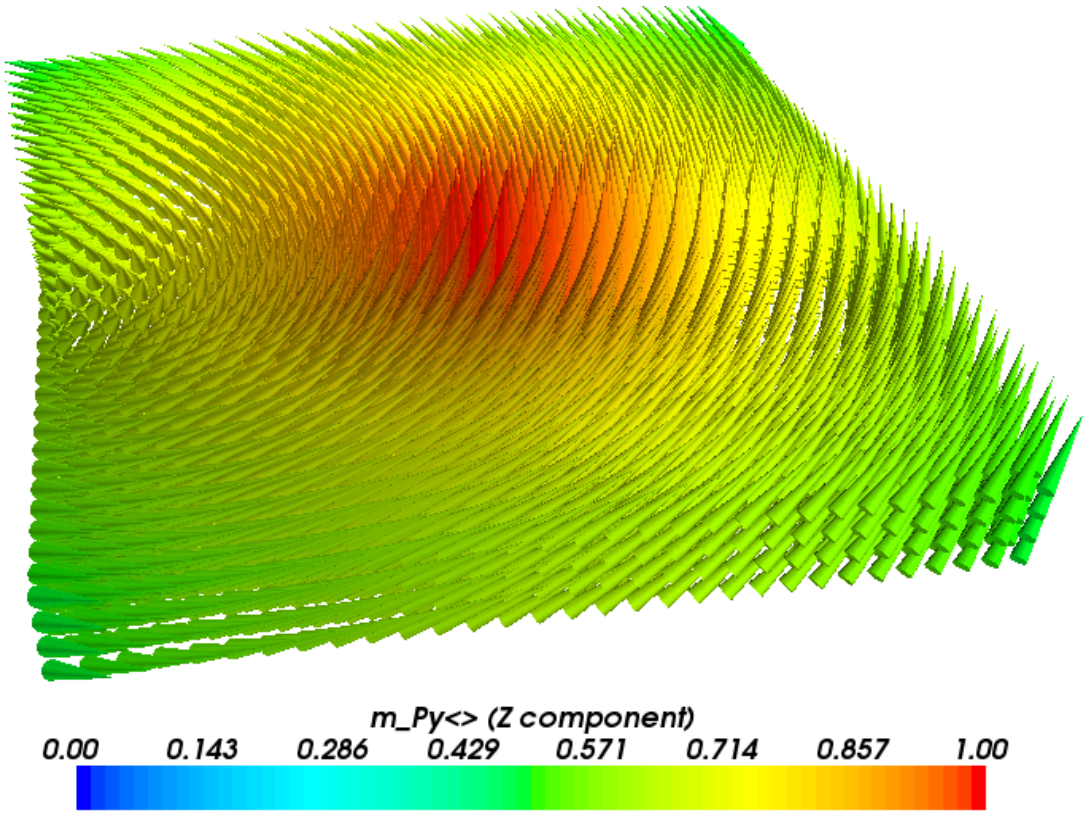
```
sim.save_restart_file("vortex_m.h5")
```

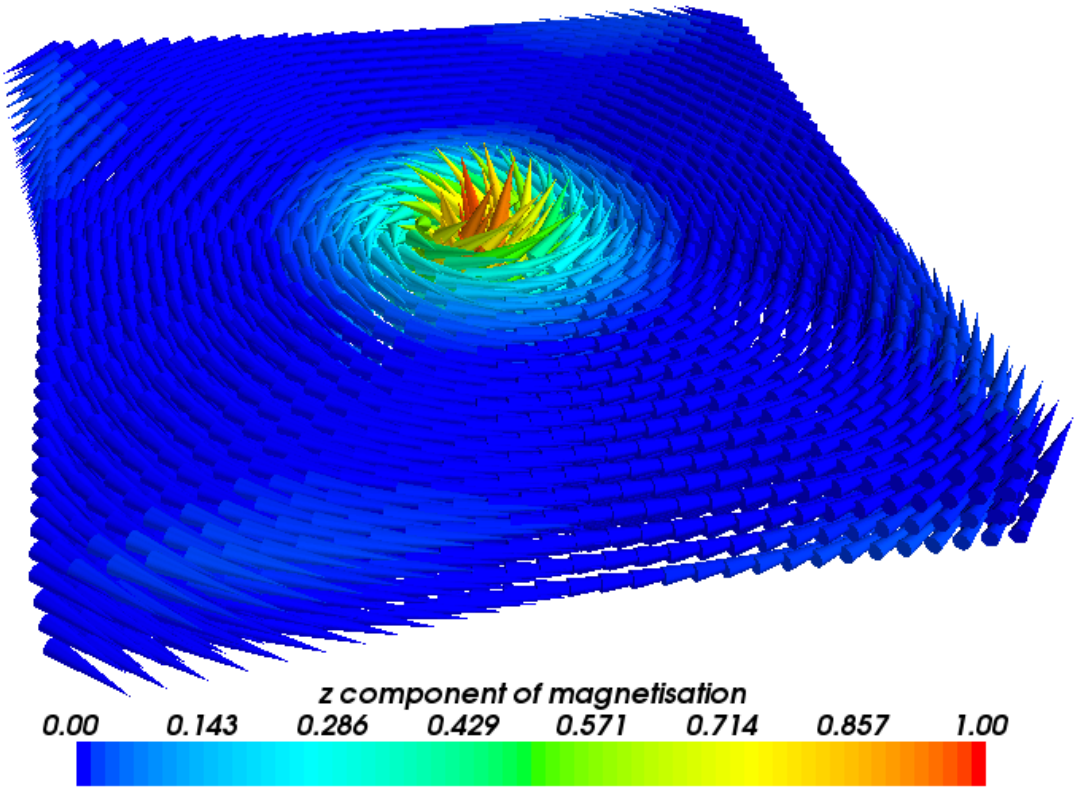
We can launch the script with the command:

```
$ nsim relaxation.py
```

The output files for this simulation will have the prefix `relaxation` in their names. The script saves the magnetisation at the beginning (before relaxation) and at the end (after relaxation). The magnetisations can be extracted and saved into vtk files using the command `nmagpp relaxation --vtk=m.vtk`, as usual. MayaVi can then be used to show the initial magnetisation (as described by the `initial_m` function):

The magnetisation at the end of the relaxation process:


m_Py<> (Z component)
0.00
0.143
0.286
0.429
0.571
0.714
0.857
1.00


z component of magnetisation
0.00
0.143
0.286
0.429
0.571
0.714
0.857
1.00

The relaxed vortex is much smaller than the initial one. The important thing to notice is that such a magnetisation configuration has now been saved into the file `vortex_m.h5` which will be used as the initial magnetisation for part II of this simulation, where we study the current driven dynamics of the vortex.

2.21.3 Part II: Current driven dynamics

For part II we need to use a slightly modified version of the script used for part I. Here is the full listing of `stt.py`:

```
# We model a bar 100 nm x 100 nm x 10 nm where a vortex sits in the center.
# This is part II: we load the vortex from file and apply a spin-polarised current

import nmag
from nmag import SI, every, at

# Define the material
mat_Py = nmag.MagMaterial(name="Py",
                          Ms=SI(0.8e6,"A/m"),
                          exchange_coupling=SI(13.0e-12, "J/m"),
                          llg_gamma_G=SI(0.2211e6, "m/A s"),
                          llg_polarisation=1.0,
                          llg_xi=0.05,
                          llg_damping=0.1)

# Define the simulation object and load the mesh
sim = nmag.Simulation()
sim.load_mesh("pyfilm.nmesh.h5", [("Py", mat_Py)], unit_length=SI(1e-9,"m"))

# Set the initial magnetisation: part II uses the one saved by part I
sim.load_m_from_h5file("vortex_m.h5")
sim.set_current_density([1e12, 0, 0], unit=SI("A/m^2"))

sim.set_params(stopping_dm_dt=0.0) # * WE * decide when the simulation should stop!

sim.relax(save=[('fields', at('convergence') | every('time', SI(1.0e-9, "s"))),
                ('averages', every('time', SI(0.05e-9, "s")) | at('stage_end'))],
          do = [('exit', at('time', SI(10e-9, "s")))])
```

We now discuss the script with particular emphasis on the differences with the first one. One difference lies in the material definition:

```
# Define the material
mat_Py = nmag.MagMaterial(name="Py",
                          Ms=SI(0.8e6,"A/m"),
                          exchange_coupling=SI(13.0e-12, "J/m"),
                          llg_gamma_G=SI(0.2211e6, "m/A s"),
                          llg_polarisation=1.0,
                          llg_xi=0.05,
                          llg_damping=0.1)
```

Here we use two new arguments for the *MagMaterial* class. The first is `llg_polarisation` which is used to specify the spin polarisation of the conduction electrons inside the given material. The second, `llg_xi`, is used to specify the degree of non-adiabaticity. Note that for the damping parameter, `llg_damping`, we are now using a smaller value, 0.1 (these values are not realistic for Permalloy).

The script then continues by creating the simulation object and loading the mesh (which is identical to the relaxation script shown in part I). The initial magnetisation is read from the `vortex_m.h5` file:

```
# Set the initial magnetisation: part II uses the one saved by part I
sim.load_m_from_h5file("vortex_m.h5")
```

Here we use the function *load_m_from_h5file* to load the magnetisation from the file `vortex_m.h5`, which was created in part I by using the function *save_restart_file*. We set the current density:

```
sim.set_current_density([1e12, 0, 0], unit=SI("A/m^2"))
```

The current density has norm `10^12 A/m^2` and is aligned in the `x` direction. We then disable the convergence check:

```
sim.set_params(stopping_dm_dt=0.0) # * WE * decide when the simulation should stop!
```

Here we decide that convergence should be reached when the magnetisation moves less than 0.0 degrees per nanosecond. This cannot happen and hence convergence is never reached: we'll tell the `relax` method to exit after a fixed amount of time has been simulated:

```
sim.relax(save=[('fields', at('convergence') | every('time', SI(1.0e-9, "s"))),
                ('averages', every('time', SI(0.05e-9, "s")) | at('stage_end'))],
          do = [('exit', at('time', SI(10e-9, "s")))])
```

We run the simulation for just 10 nanoseconds by forcing an exit with `('exit', at('time', SI(10e-9, "s")))`. We also save the fields every nanosecond and save the averages more often, every 50 picoseconds. The relax method will simulate a vortex "hit" by a spin polarised current and will save the averages so that we can see how the magnetisation changes in time.

To run the script (which takes of the order of half an hour) we use as usual:

```
$ nsim stt.py
```

The output files for this simulation will start with the prefix `stt`. The script saves the average magnetisation periodically in time. We can therefore plot it using the following gnuplot script:

```
set term postscript color eps enhanced
set out "m_of_t.eps"

set xlabel "time (ns)"
set ylabel "average magnetisation (10^6 A/m)"
plot [0:10] \
  "m_of_t.dat" u ($1*1e9):($2/1e6) t "<M_x>" w lp, \
  "" u ($1*1e9):($3/1e6) t "<M_y>" w lp, \
  "" u ($1*1e9):($4/1e6) t "<M_z>" w lp
```

to obtain the following graph:

2.21.4 Standard problem

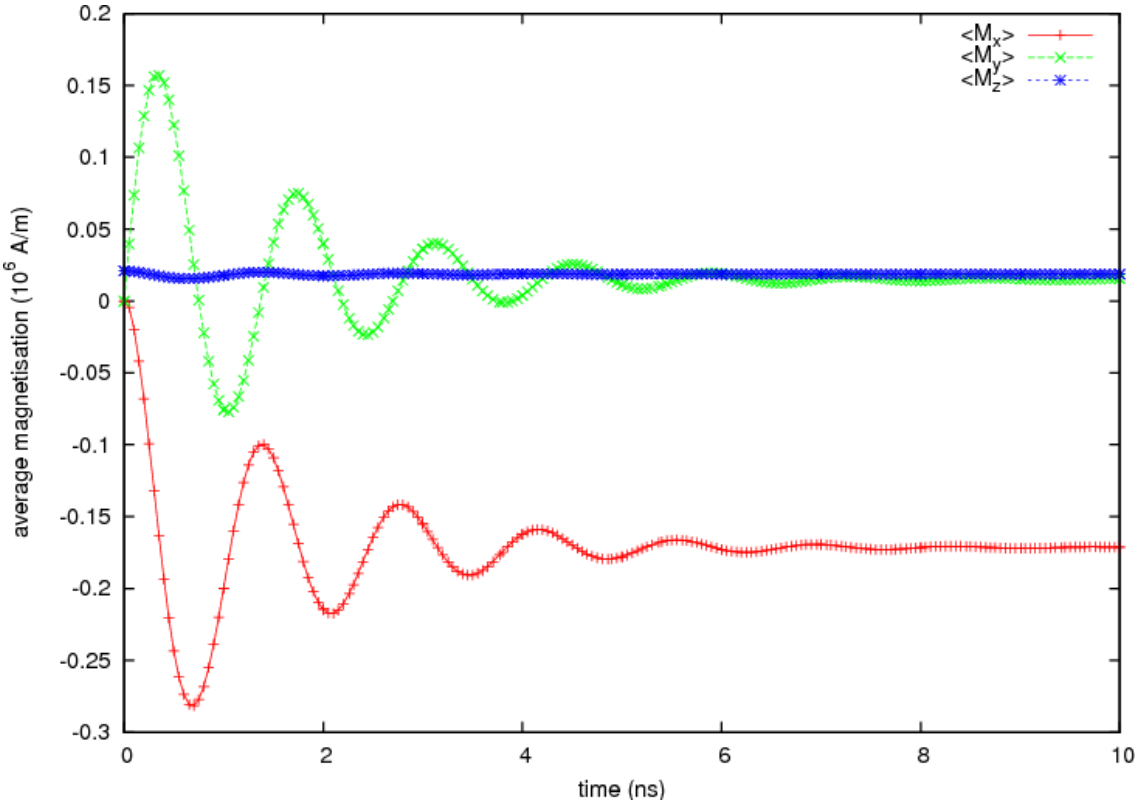
The simulation carried out here is a (coarse) version of the recently proposed standard problem for spin transfer torque micromagnetic studies [6].

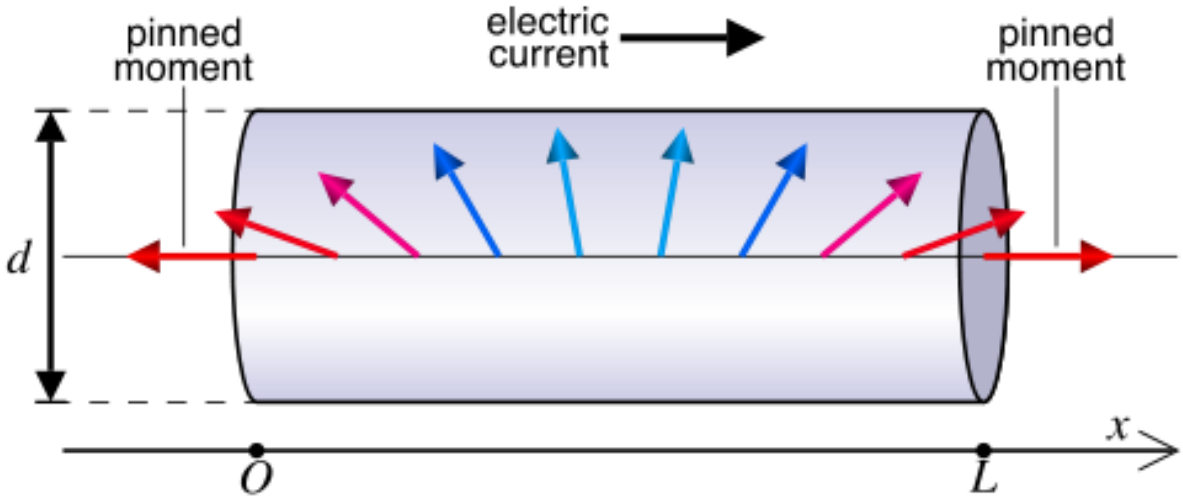
2.22 Example: Current-driven magnetisation precession in nanopillars

This is the second example we provide in order to illustrate the usage of the Zhang-Li extension to model spin-transfer-torque in Nmag. While in the *Current-driven motion of a vortex in a thin film* example we tried to present two scripts (one for initial relaxation, and one for the spin torque transfer simulation), sacrificing usability for the sake of clarity, here we'll try to present a real-life script, using the power of the Python programming language as much as it is needed to achieve our goal.

We consider a ferromagnetic nanopillar in the shape of a cylinder. We assume that the magnetisation in the nanopillar is pinned in the two faces of the cylinder along opposite directions: on the right face the magnetisation points to the right, while on the left face it points to the left. The magnetisation is then forced to develop a domain

[6] Massoud Najafi, Benjamin Kruger, Stellan Bohlens, Matteo Franchin, Hans Fangohr, Antoine Vanhaverbeke, Rolf Allenspach, Markus Bolte, Ulrich Merkt, Daniela Pfannkuche, Dietmar P. F. Moller, and Guido Meier, *Proposal for a Standard Problem for Micromagnetic Simulations Including Spin-Transfer Torque*, Journal of Applied Physics, in print (2009), preprint available at http://www.soton.ac.uk/~fangohr/publications/preprint/Najafi_etal_2009.pdf


0.2
0.15
0.1
0.05
0
-0.05
-0.1
-0.15
-0.2
-0.25
-0.3
0
2
4
6
8
10
time (ns)
average magnetisation (10^6 A/m)
<Mx>
<My>
<Mz>

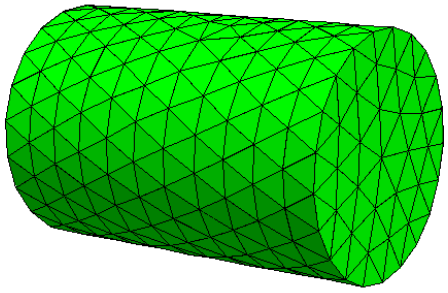

pinned
moment
electric
current
pinned
moment
d
x
O
L

wall. We then study how such an "artificial" domain wall interacts with a current flowing throughout the cylinder, along its axis.

By "artificial" we mean that the domain wall is developed as a consequence of the pinning, which we artificially impose. In real systems, the pinning can be provided through interface exchange coupling or may have a geometrical origin, in combination with suitable material parameters. The situation we consider here is described and studied in more detail in publications [7] and [8].

2.22.1 Two simulations in one single script

The nanopillar is made of Permalloy and has the shape of a cylinder with radius of 10 nm and length 30 nm. The mesh is loaded from the file `l030.nmesh.h5` which was created using Netgen from the file `l030.geo`.



The simulation is subdivided into two parts, similarly to the previous example:

- In part I, the system is relaxed to obtain the initial magnetisation configuration when the current is not applied.
- In part II the current is applied to the artificial domain wall whose shape was calculated in part I.

This time, however, we use just one single script to execute both parts of the simulation in one go. In particular, we define a function which takes some input parameters such as the current density, the damping, etc and uses them to carry out a simulation. We then call this function twice: once for part I and once for part II.

The full listing of the script `stt_nanopillar.py`:

```
import nmag, os, math
from nmag import SI, every, at
from nsim.si_units.si import degrees_per_ns

l = 30.0                        # The nanopillar thickness is 30 nm
hl = l/2                        # hl is half the nanopillar thickness
relaxed_m_file = "relaxed_m.h5" # File containing the relaxed magnetisation
mesh_name = "l030.nmesh.h5"     # Mesh name
mesh_unit = SI(1e-9, "m")       # Unit length for space used by the mesh

def run_simulation(sim_name, initial_m, damping, stopping_dm_dt,
                   j, P=0.0, save=[], do=[], do_demag=True):
  # Define the material
  mat = nmag.MagMaterial(
          name="mat",
          Ms=SI(0.8e6, "A/m"),
          exchange_coupling=SI(13.0e-12, "J/m"),
          llg_damping=damping,
          llg_xi=SI(0.01),
          llg_polarisation=P)
```

[7] Matteo Franchin, Thomas Fischbacher, Giuliano Bordignon, Peter de Groot, Hans Fangohr, *Current-driven dynamics of domain walls constrained in ferromagnetic nanopillars*, Physical Review B **78**, 054447 (2008), online at http://eprints.soton.ac.uk/59253,

[8] Matteo Franchin, Giuliano Bordignon, Peter A. J. de Groot, Thomas Fischbacher, Jurgen P. Zimmermann, Guido Meier, Hans Fangohr, *Spin-polarized currents in exchange spring systems*, Journal of Applied Physics **103**, 07A504 (2008), online at http://link.aip.org/link/?JAPIAU/103/07A504/1

```
  # Create the simulation object and load the mesh
  sim = nmag.Simulation(sim_name, do_demag=do_demag)
  sim.load_mesh(mesh_name, [("np", mat)], unit_length=mesh_unit)

  # Set the pinning at the top and at the bottom of the nanopillar
  def pinning(p):
    x, y, z = p
    tmp = float(SI(x, "m")/(mesh_unit*hl))
    if abs(tmp) >= 0.999:
      return 0.0
    else:
      return 1.0
  sim.set_pinning(pinning)

  if type(initial_m) == str:            # Set the initial magnetisation
    sim.load_m_from_h5file(initial_m)   # a) from file if a string is provided
  else:
    sim.set_m(initial_m)                # b) from function/vector, otherwise

  if j != 0.0:                          # Set the current, if needed
    sim.set_current_density([j, 0.0, 0.0], unit=SI("A/m^2"))

  # Set additional parameters for the time-integration and run the simulation
  sim.set_params(stopping_dm_dt=stopping_dm_dt,
                 ts_rel_tol=1e-7, ts_abs_tol=1e-7)
  sim.relax(save=save, do=do)
  return sim

# If the initial magnetisation has not been calculated and saved into
# the file relaxed_m_file, then do it now!
if not os.path.exists(relaxed_m_file):
  # Initial direction for the magnetisation
  def m0(p):
      x, y, z = p
      tmp = min(1.0, max(-1.0, float(SI(x, "m")/(mesh_unit*hl))))
      angle = 0.5*math.pi*tmp
      return [math.sin(angle), math.cos(angle), 0.0]

  save = [('fields', at('step', 0) | at('stage_end')),
          ('averages', every('time', SI(5e-12, 's')))]

  sim = run_simulation(sim_name="relaxation", initial_m=m0,
                       damping=0.5, j=0.0, save=save,
                       stopping_dm_dt=1.0*degrees_per_ns)
  sim.save_restart_file(relaxed_m_file)
  del sim

# Now we simulate the magnetisation dynamics
save = [('averages', every('time', SI(9e-12, 's')))]
do   = [('exit', at('time', SI(6e-9, 's')))]
run_simulation(sim_name="dynamics", initial_m=relaxed_m_file, damping=0.02,
               j=0.1e12, P=1.0, save=save, do=do, stopping_dm_dt=0.0)
```

After importing the required modules, we define some variables such as the length of the cylinder, `l`; the name of the file where to put the relaxed magnetisation, `relaxed_m_file`; the name of the mesh, `mesh_name`; its unit length, `mesh_unit`:

```
l = 30.0                        # The nanopillar thickness is 30 nm
hl = l/2                        # hl is half the nanopillar thickness
relaxed_m_file = "relaxed_m.h5" # File containing the relaxed magnetisation
mesh_name = "l030.nmesh.h5"     # Mesh name
mesh_unit = SI(1e-9, "m")       # Unit length for space used by the mesh
```

These quantities are used later in the script. For example, knowing the length of the nanopillar is necessary in order to set a proper initial magnetisation for the relaxation. By making this a parameter at the top of the program, we can change it there (if we wan to study the same system for a different l), and just run the script again.

We define the function `run_simulation`: we teach Python how to run a simulation given some parameters, such as the initial magnetisation, the damping, the current density, etc. The function is defined starting with the line:

```
def run_simulation(sim_name, initial_m, damping, stopping_dm_dt,
                   j, P=0.0, save=[], do=[], do_demag=True):
```

The arguments of the function (the names inside the parenthesis) are those parameters which must be choosen differently in part I and part II. For example, we decided to make the current density `j` an argument for the function `run_simulation`, because in part I `j` must be set to zero, while in part II it must be set to some value greater than zero. On the other hand, the saturation magnetisation does not appear in the argument list of the function, since it has the same value both in part I and part II.

A remark about the Python syntax: arguments such as `sim_name` must be specified explicitly when using the function `run_simulation`, while arguments such as `P=0` have a default value (0.0 in this case) and the user may omit them, meaning that Python will then use the default values.

We skip the explanation of the body of the function and focus on the code which follows it. We'll return later on the implementation of `run_simulation`. For now, the user should keep in mind that `run_simulation` just runs one distinct micromagnetic simulation every time it is called (and what simulation this is will depend on the parameters given to the function). The function returns the simulation object which it created.

We now comment the code which follows the function `run_simulation`:

```
# If the initial magnetisation has not been calculated and saved into
# the file relaxed_m_file, then do it now!
if not os.path.exists(relaxed_m_file):
  # Initial direction for the magnetisation
  def m0(pos):
      x, y, z = pos
      tmp = min(1.0, max(-1.0, float(SI(x, "m")/(mesh_unit*hl))))
      angle = 0.5*math.pi*tmp
      return [math.sin(angle), math.cos(angle), 0.0]

  save = [('fields', at('step', 0) | at('stage_end')),
          ('averages', every('time', SI(5e-12, 's')))]

  sim = run_simulation(sim_name="relaxation", initial_m=m0,
                       damping=0.5, j=0.0, save=save,
                       stopping_dm_dt=1.0*degrees_per_ns)
  sim.save_restart_file(relaxed_m_file)
  del sim
```

This piece of code carries out part I of the simulation: it relaxes the system starting from a sensible initial guess for the magnetisation and saves the relaxed magnetisation configuration so that it can be used in part II.

In more detail, it starts by checking (using the function `os.path.exists`) if a file containing the initial magnetisation exists. If this is not the case, then the following indented block will be executed, which computes and saves this initial magnetisation. If the file exists, the whole indented block is skipped, and we go straight to part II of the calculation.

In order to compute the relaxed configuration, an initial guess `m0` for the magnetisation is defined. Such magnetisation linearly rotates from left to right as the position changes from the left face to the right face of the cylinder. Here `x` is the x coordinate of the position vector `p` and `tmp = min(1.0, max(-1.0, float(SI(x, "m")/(mesh_unit*hl))))` is a continuous function which changes linearly from -1 to 1 when going from the left to the right face, keeping constant outside the cylinder.

In the code above we also define the variable `save` which is used to specify when and what should be saved to disk. Here we save the fields before and after the relaxation and save the averages every 5 picoseconds.

We then call the `run_simulation` function we defined above to relax the magnetisation. This function returns the simulation object `sim`, which we use to save the magnetisation using the *save_restart_file* function.

Once this is done, we delete the simulation object, releasing resources (memory) we have used for the simulation of part I. Note that for the relaxation, we use `j=0.0` (zero current density), `damping=0.5` (fast damping, to reach convergence quickly) and `stopping_dm_dt=1.0*degrees_per_ns` (this means that the simulation should end when the magnetisation moves slower than 1 degree per nanosecond).

The following part of the script deals with part II, the computation of the current-driven dynamics:

```
# Now we simulate the magnetisation dynamics
save = [('averages', every('time', SI(9e-12, 's')))]
do   = [('exit', at('time', SI(6e-9, 's')))]

run_simulation(sim_name="dynamics",
               initial_m=relaxed_m_file,
               damping=0.02,
               j=0.1e12,
               P=1.0,
               save=save,
               do=do,
               stopping_dm_dt=0.0)
```

Here we decide to save the averages every 9 picoseconds and exit the simulation after 6 nanoseconds. We use `stopping_dm_dt=0.0` to disable the convergence check (here we just want to simulate for a fixed amount of time). We also use full spin polarisation, `P=1.0`, we apply a current density of `j=0.1e12 A/m^2` and use a realistic damping parameter for Permalloy, `damping=0.02`. For the initial magnetisation we pass the name of the file where the relaxed magnetisation was saved in part I and we specify a simulation name `sim_name="dynamics"` which is different from the one used for the relaxation (which was `sim_name="relaxation"`). The simulation name will decide the prefix of any filenames that are being created when saving data. (If the simulation name is not specified, the name of the script file is used.)

We now return to discuss the function `run_simulation` and see how it carries out the actual simulations. First, the function defines the material:

```
# Define the material
mat = nmag.MagMaterial(
        name="mat",
        Ms=SI(0.8e6, "A/m"),
        exchange_coupling=SI(13.0e-12, "J/m"),
        llg_damping=damping,
        llg_xi=SI(0.01),
        llg_polarisation=P)
```

It uses the variable `P` which is passed as an argument to the function. Then the simulation object is created and the mesh is loaded:

```
# Create the simulation object and load the mesh
sim = nmag.Simulation(sim_name, do_demag=do_demag)
sim.load_mesh(mesh_name, [("np", mat)], unit_length=mesh_unit)
```

Note that `sim_name` is passed to the Simulation object, allowing the user to use different prefixes for the output files of the simulation. For example, if `sim_name = "relaxation"`, then the output files produced when saving the fields or their averages to disk will have names starting with the prefix `relaxation_`. On the other hand, if `sim_name = "dynamics"`, the names of these files will all start with the prefix `dynamics_`.

Using different simulation names allows us to save the data of part I and part II in different independent files. The function continues with the code above:

```
# Set the pinning at the left and right face of the nanopillar
def pinning(p):
  x, y, z = p
  tmp = float(SI(x, "m")/(mesh_unit*hl))
  if abs(tmp) >= 0.999:
```

```
    return 0.0
  else:
    return 1.0
sim.set_pinning(pinning)
```

which is used to pin the magnetisation at the left and right faces of the cylinder. Note here that `x` is the x component of the position of the mesh site and that:

```
tmp = float(SI(x, "m")/(mesh_unit*hl))
```

is equal to -1 at the right face, and to +1 at the left face. We then set the magnetisation. If `initial_m` is a string, then we assume it is the name of the file and load the magnetisation with the method `load_m_from_h5file`, otherwise we assume it is just a function and set the magnetisation in the usual way, using the method `set_m`:

```
if type(initial_m) == str:            # Set the initial magnetisation
  sim.load_m_from_h5file(initial_m)   # a) from file if a string is provided
else:
  sim.set_m(initial_m)                # b) from function/vector, otherwise
```

We then set the current density along the x direction (only if `j` is not zero):

```
if j != 0.0:                          # Set the current, if needed
  sim.set_current_density([j, 0.0, 0.0], unit=SI("A/m^2"))
```

Finally, we set tolerances, the stopping criterion and launch the simulation:

```
# Set additional parameters for the time-integration and run the simulation
sim.set_params(stopping_dm_dt=stopping_dm_dt,
               ts_rel_tol=1e-7, ts_abs_tol=1e-7)
sim.relax([None], save=save, do=do)
return sim
```

The *relax* function carries out the simulation, taking into account the stopping criterion and `save` and `do` actions. Finally, the function returns the simulation object which it created.

2.22.2 Results: precession of the magnetisation

We launch the script with:

```
$ nsim stt_nanopillar.py
```

The script runs both part I (output files starting with `relaxation_`) and part II (output files starting with `dynamics_`). The relaxed magnetisation can be extracted and saved into a vtk file using the command `nmagpp relaxation --vtk=m.vtk`. MayaVi can then be used to obtain the following picture:

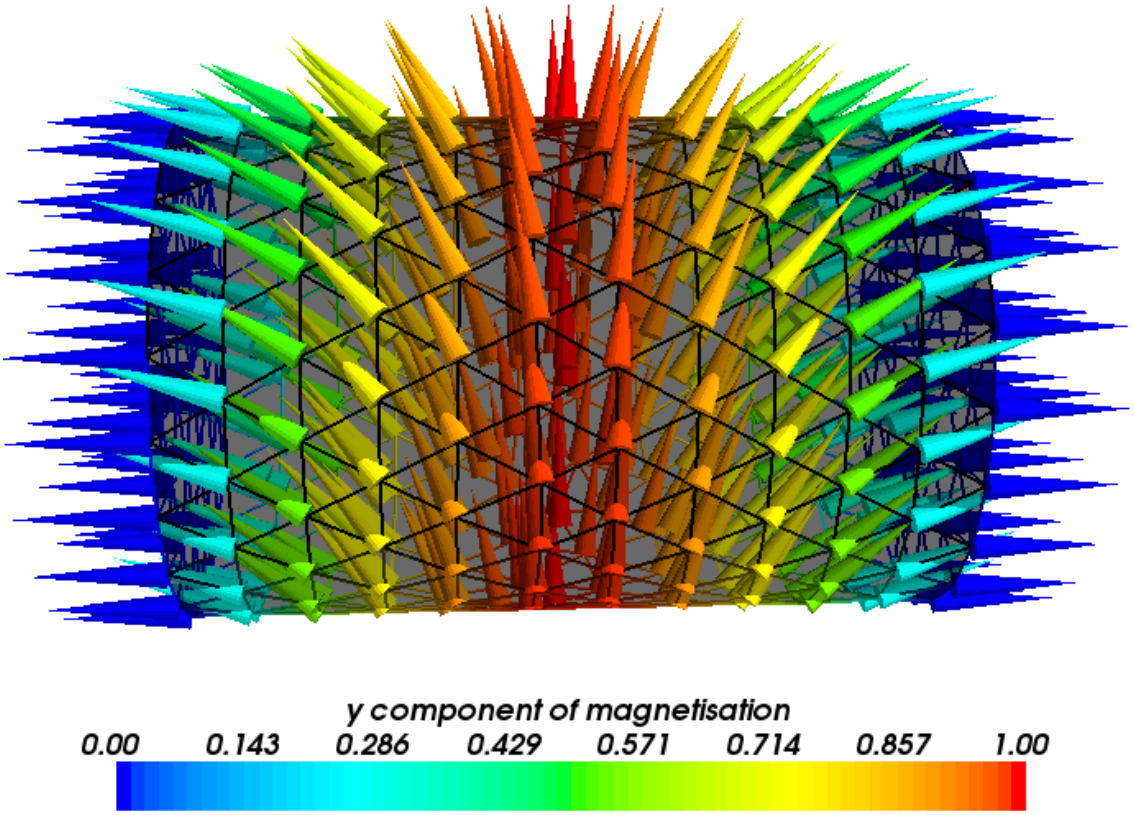
We can now take a look at the results obtained for the dynamics. The average magnetisation as a function of time can be extracted using:

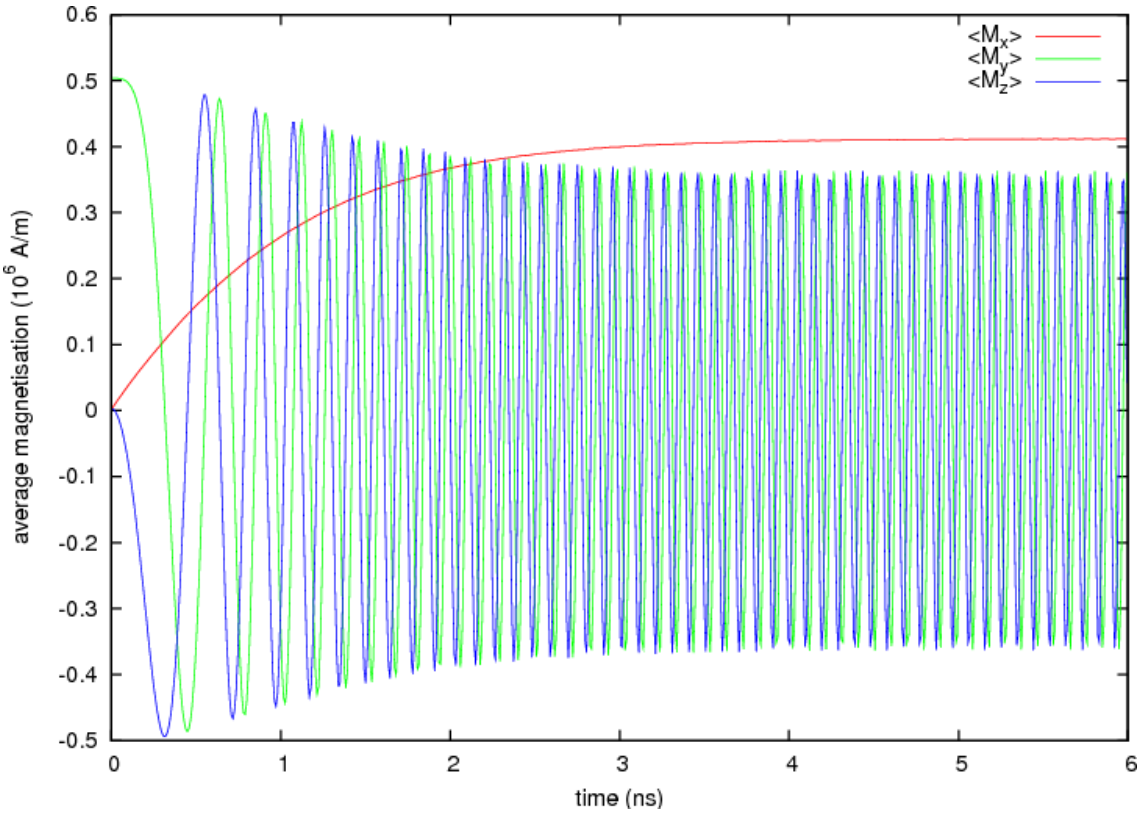
```
ncol dynamics time M_mat_0 M_mat_1 M_mat_2 > m_of_t.dat
```

We can use the following gnuplot script:

```
set term postscript color eps enhanced solid
set out "m_of_t.eps"

set xlabel "time (ns)"
set ylabel "average magnetisation (10^6 A/m)"
plot [0:6] \
  "m_of_t.dat" u ($1*1e9):($2/1e6) t "<M_x>" w l, \
  "" u ($1*1e9):($3/1e6) t "<M_y>" w l, \
  "" u ($1*1e9):($4/1e6) t "<M_z>" w l
```

and obtain the following graph:


y component of magnetisation
0.00
0.143
0.286
0.429
0.571
0.714
0.857
1.00


average magnetisation (10^6 A/m)
time (ns)
<Mx>
<My>
<Mz>
0.6
0.5
0.4
0.3
0.2
0.1
0
-0.1
-0.2
-0.3
-0.4
-0.5
0
1
2
3
4
5
6

The sinusoidal dependence of the y and z magnetisation components suggests that the magnetisation rotates around the nanopillar axis with a frequency which increases to approach its maximum value.

A more detailed discussion of results and interpretation is provided in the publications [1] and [2] mentioned in section *Example: Current-driven magnetisation precession in nanopillars*.

2.23 Mesh distortion for edge roughness simulation

The meshes used in micromagnetic simulations usually represent idealized geometries (for example, a nanowire might be modeled using a completely smooth cuboid mesh). Real-world materials, on the other hand, possess imperfections on various scales caused by fabrication processes (e.g., electron beam lithography or sputter deposition). This can potentially have a significant impact on the magnetization dynamics. The advantage of finite element-based simulations is that such effects can be simulated (at least qualitatively) by distorting the mesh in a suitable way. *nmeshpp* provides a means of distorting a given mesh in order to imitate roughness so that the resulting effects on simulations can be explored. Note that at the moment only edge roughness is supported. We first present an example in the following section and then go into the details of the command line interface and how the distortion process works.

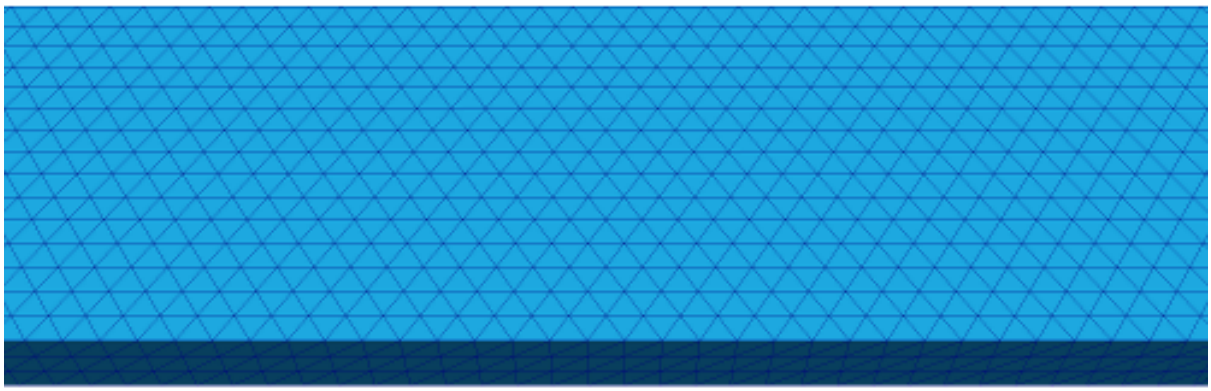
2.23.1 Example

Consider a nanowire with dimensions 800nm x 20nm x 5nm (for convenience we provide the corresponding mesh in the file `nanowire_800x20x5.nmesh`). [9]

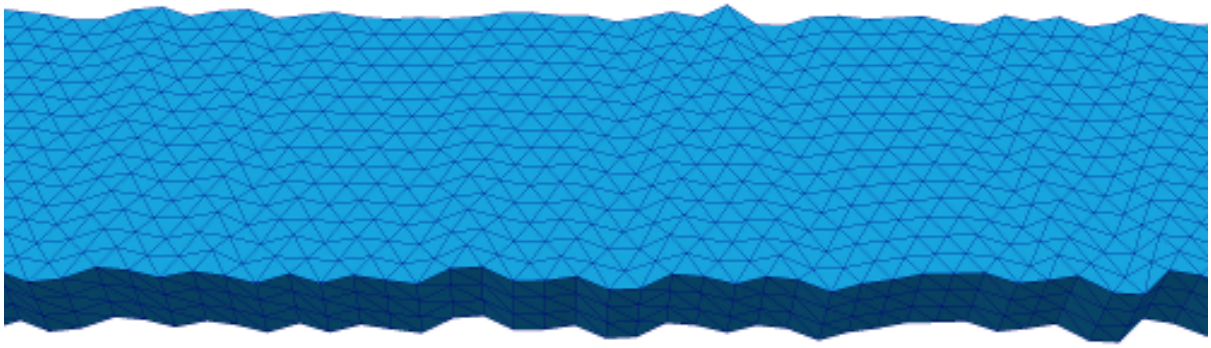
We distort this mesh using the following command:

```
nmeshpp --distort 0.4 --correlation-length 2.0 --seed 23 nanowire_800x20x5.nmesh \
          nanowire_800x20x5_distorted.nmesh
```

Intuitively, what this command does is to randomly displace the "front" and "rear" nodes of the mesh and to stretch/shrink the middle bits accordingly. The details of this process, as well as meaning of all the command line switches, are explained in the next section. The original and distorted mesh look like this (only part of each mesh is shown):



[9] The mesh file for the nanowire was produced using the `examesh` tool, which is included in the nmag distribution in the directory `utils/cubicmesh/` (note that it needs to be compiled before it can be used - just `cd` into this directory and type `make`). The exact command used to produce the mesh file was `examesh nanowire_800x20x5.nmesh,800:450,20:15,5:3`.



2.23.2 Details and command line options

Preliminary remark: As mentioned above, `nmeshpp` can only produce edge roughness at the moment. There is a slight chance that the user interface might change in the future when more functionality (such as surface roughness) is added.

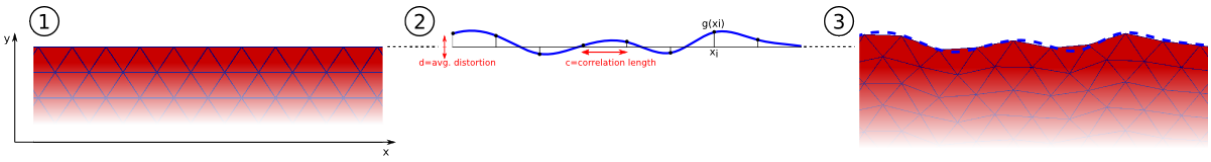
In this section we go into the details of the distortion process and explain the relevant command line options. The general usage is:

```
nmeshpp --front-rear-axis [X|Y|Z] --distort-along-axis [X|Y|Z] --distort D \
        --correlation-length C --seed S mesh_orig.nmesh mesh_distorted.nmesh
```

Only `--distort`, `--correlation-length` and the name of the input mesh are required arguments.

The overall distortion process works as follows. First, the surface nodes of the mesh are divided into "front" and "rear", depending on which side of the mesh they lie on. By default, this distinction is based on their y-coordinate (as in the example in the previous section), but this can be changed using the option `--front-rear-axis`. Next, a univariate "distortion function" `f(x)` is constructed based on the given command line parameters (the details of this process will be explained in a moment). This function specifies the amount by which each *front* node is displaced in y-direction (as a function of the x-coordinate of the node). Analogously, the *rear* nodes are displaced using a second, independently constructed distortion function `g(x)` (so that both sides of the mesh are distorted differently). The intermediate parts of the mesh are stretched to fit nicely between the new distorted sides.

The whole procedure is illustrated in the following picture. It shows a top view (i.e., along the z-axis) of the rear part of the nanowire from the previous section. The left hand side shows the original mesh, the right hand side shows the mesh after distortion with the function `g`, which is depicted in the middle. Note that the contour of the distorted mesh follows the outline of `g`.



The distortion functions `f` and `g` are constructed as follows. First we pick equidistant nodes `x_i` along the x-axis (note that these are just auxiliary entities and completely independent from the nodes of the mesh). Then random values `f(x_i)` and `g(x_i)` are assigned to each such node, chosen from a normal distribution with mean 0 and a certain standard deviation that determines the "amplitude" of the roughness. Finally, these random values are interpolated smoothly to obtain the continuous distortion functions `f(x)` and `g(x)`. In order to make the randomization reproducible, it is possible to specify a seed for the internal random number generator (by passing any integer value as an argument to `--seed`). Otherwise the output mesh is different each time because the random number generator is seeded using the system time or something similar.

The reason why we need these distortion functions at all and can't just randomly displace each mesh node individually is because then the result would strongly depend on the mesh spacing and the overall quality of the mesh. However, since we usually want roughness on a scale independent from the mesh spacing, we need some kind of correlation between the displacements of adjacent mesh nodes, hence the need for the distortion functions.

The parameters in the construction of `f` and `g` are:

- the distance between the nodes `x_i`, which can be controlled with the flag `--correlation-length`,
- the standard deviation of the underlying normal distribution, which must be specified using the command line switch `-d`, or `--distort`.

Note that depending on which edges of the mesh the roughness should be applied to (and on the way the mesh is oriented in the coordinate system), it may be necessary to apply the distortion in a direction different from the y-direction and also to consider `f` and `g` as functions of a different input axis (distinct from the x-axis). The first can again be controlled using the option `--front-rear-axis`, which was already mentioned above. The second can be adjusted using `--distort-along-axis`. For instance, if the roughness should displace the nodes in z-direction and if the amount of displacement should be a function of their y-coordinates, the command line arguments would be `--front-rear-axis Z --distort-along-axis Y`.

2.24 Compression of the Boundary Element Matrix using HLib

2.24.1 Hierarchical Matrices in Micromagnetism

Nmag uses the hybrid finite element method/boundary element method (hybrid FEM/BEM) to compute the demagnetisation field (as does Magpar). Not using this method, one would have to discretise a large part of space arround the magnetic structure (ideally all space). Using the hybrid FEM/BEM method, it is only necessary to discretise (and solve the equations for the demag field on that discretisation) those parts of space that are occupied by magnetic material.

A disadvantage of the hybrid FEM/BEM method is that it involves the assembly of a dense boundary element matrix **B**, whose number of elements scales quadratically with the number of surface nodes N of our finite element mesh, i.e. the matrix **B** has as many rows as there are surface nodes N in the mesh (and also as many columns).

This is in particular an issue when studying flat structures such as thin films. For example, imagine we model a thin film of side lengths 100 nm x 100 nm x 2nm. If we decide to double the side lengths to 200 nm x 200 nm x 2nm, then this roughly corresponds to an increase of surface node numbers N by a factor of 4. The matrix **B** will then grow in size by a factor 4^2=16 due to the doubling of the two side lengths by a factor of 2. In practice, the memory requirements of the matrix **B** often limit the size of a structure that can be modelled.

In order to improve the efficiency of the hybrid FEM/BEM, one can employ techniques which involve some kind of approximation of **B**, for example using hierarchical matrices.

The basic idea is to approximate submatrices of **B** by a data-sparse approximation where possible (within user-provided tolerance margins). In general the complexity of the storage requirements and execution time of simple operations like the matrix-vector product scale as O(N*log(N)), as compared to the quadratical costs N^2 using the standard matrix representation. For the use of HLib hierarchical matrices in micromagnetic simulations we are often mostly interested in the their reduced *memory* requirements.

The library HLib contains implementations of this hierarchical matrix methodology, and can be used with Nmag in order to run micromagnetic simulations in a more memory efficient way (see for example Knittel et al 105, 07D542 (2009), postprint pdf). .

2.24.2 Installation of HLib

In order to be able to use the HLib library and to obtain the HLib source code, you have to apply for an HLib licence as explained on http://hlib.org/license.html.

Once the HLib authors grant a licence, they will send their HLib tarball. Nmag will have to be compiled from source (see install from source) in the presence of this tarball to make use of the HLib matrix compression. (Nmag

will compile happily in the absence of this file, and in that case the boundary element matrix is stored 'in the normal way' as a full matrix.)

We describe the required steps for this in detail. We assume you downloaded the HLib tarball and the Nmag tarball in your home directory `~/` (but any other subdirectory will work fine). Then, if you issue a `ls` command, you get something like:

```
me@mymachine:~/$ ls
HLib-1.3p19.tar.gz  nmag-0.1-all.tar.gz
```

You can now untar the nmag tarball and enter the newly created directory:

```
me@mymachine:~/$ tar xzvf nmag-0.1-all.tar.gz
me@mymachine:~/$ cd nmag-0.1
```

Note that in this particular example we assume the Nmag version to be 0.1. For later versions, you'll have to change the tarball name and the paths accordingly (e.g. `nmag-X.Y.Z` for version `X.Y.Z`). Inside the directory `nmag-0.1` there is a directory called `hlib-pkg` and we need to copy (or move) the HLib tarball into this directory:

```
me@mymachine:~/nmag-0.1$ cp HLib-1.3p19.tar.gz hlib-pkg/
```

You can now compile Nmag with HLib support in the usual way:

```
me@mymachine:~/nmag-0.1$ make
```

The build system should recognise that the `hlib-pkg` directory contains a tarball and should prompt you asking what to do:

```
me@mymachine:~/nmag-0.1$ make
bash ./patches/hlib/hlib-untar.sh ./hlib-pkg HLib-1.3p19.tar.gz && \
          rm -f .deps_hlib_patch && make .deps_hlib_install; true
```

```
 It seems you want to compile Nmag with HLib support
 I'll need your confirmation in order to proceed...

I found ./hlib-pkg/HLib-1.3p19.tar.gz
Is this the HLib tarball you want to use? (yes/no) yes
```

Type `yes` and `ENTER`. The build system should untar the HLib tarball, it should patch it (HLib needs to be patched in order to be usable by Nmag) and it should install it in the right location with respect to the Nmag libraries. If all goes well, you should get an installation of Nmag which is capable of using HLib for the compression of the BEM matrix.

As you see, the only additional step which is required with respect to the normal procedure for compiling Nmag from source, is to put the HLib tarball inside the directory `nmag-0.1/hlib-pkg`.

The current nmag release requires Hlib version 1.3p19, to support HLib matrix compression.

2.24.3 Testing the HLib BEM Matrix compression

There is a test target `make checkhlib` which tests whether a demag field can be computed using the HLib and compares this with the result of the same calculation using a full BEM. If the deviations become large, the test will fail. To run the test, do

me@mymachine:~/nmag-0.1$ make checkhlib

The test should take less than 5 minutes. If it passes, then it appears that the hlib is used, and produces quantitatively appropriate approximations of the true solution.

2.24.4 Using HLib example 1: Demagnetisation Field of a Sphere

The properties of a hierarchical matrix depend much on the settings of different parameters and on the particular algorithm used to create the low-rank approximations. In Nmag, we only use the HCA II algorithm, which seems to be the most reliable amongst the commonly used algorithms, being still very efficient (see for example Knittel et al 105, 07D542 (2009), postprint pdf).

The performance and accuracy of the HCA II algorithm can be tuned by providing a number of parameters, which are collected inside a *HMatrixSetup* object. A default `HMatrixSetup` object is provided, where a reasonable choice of these parameters is made. The default parameters can be overriden by users.

We point the reader to the documentation of the *HMatrixSetup* class for a list and description of all avaliable parameters. The next example shows how to use HLib with the default values for the setup of the BEM matrix.

Using HLib with default parameters

The Nmag script `sphere_hlib.py` shows how Nmag can be used in order to compute the demagnetisation field within a sphere with a radius of 50 nm.

```
import nmag
import time
from nmag import SI

# When creating the simulation object, specify that the BEM hmatrix should be
# set up by using the default parameters.
sim = nmag.Simulation(phi_BEM=nmag.default_hmatrix_setup)

# Specify magnetic material, parameters chosen as in example 1
Py = nmag.MagMaterial(name="Py",
                      Ms=SI(1e6, "A/m"),
                      exchange_coupling=SI(13.0e-12, "J/m"))

# Load the mesh
sim.load_mesh('sphere.nmesh.h5',
              [('sphere', Py)],
              unit_length=SI(1e-9, 'm'))

# Set the initial magnetisation
sim.set_m([1,0,0])

# Save the demagnetisation field
sim.save_data(fields=['H_demag'])

# Probe the demagnetisation field at ten points within the sphere
for i in range(-5,6):
    x = i*1e-9
    Hdemag = sim.probe_subfield_siv('H_demag', [x,0,0])
    print "x=", x, ": H_demag = ", Hdemag
```

In this first example, we use default parameters for setting up the BEM matrix by passing the object `nmag.default_hmatrix_setup` to the `Simulation` object:

```
sim = nmag.Simulation(phi_BEM=nmag.default_hmatrix_setup)
```

This command specifies that the BEM matrix should be set up using the default parameters in `nmag.default_hmatrix_setup`. (The actual values of the parameters can be visualised on the screen by simply printing the object with `import nmag; print nmag.default_hmatrix_setup`.)

When running the simulation `sphere_hlib.py` using the usual command:

```
nsim sphere_hlib.py --clean,
```

it should print out the demagnetisation field at ten points along the line (x,0,0):

```
x= -5e-09 : H_demag =  [-333060.61988567741, -16.426569556599606, -63.649046900628299]
x= -4e-09 : H_demag =  [-333061.67213255615, -17.81158234138228, -65.112039406898973]
x= -3e-09 : H_demag =  [-333062.69422596297, -19.401486521725044, -66.015626464953897]
x= -2e-09 : H_demag =  [-333062.72991753434, -20.940683675745074, -66.988296036794026]
x= -1e-09 : H_demag =  [-333061.60282647074, -22.420106762492924, -68.042400926888646]
x= 0.0 : H_demag =  [-333060.29023012909, -23.736721821840622, -68.984395930340639]
x= 1e-09 : H_demag =  [-333058.66039082204, -24.758745874347209, -69.6797361890888]
x= 2e-09 : H_demag =  [-333055.87727687479, -24.635979967196079, -70.705429412122513]
x= 3e-09 : H_demag =  [-333054.17167091055, -24.9868363963913, -73.501799477569747]
x= 4e-09 : H_demag =  [-333052.78687652596, -25.388604442091431, -76.097088958697071]
x= 5e-09 : H_demag =  [-333051.43416558538, -25.507782471847442, -77.792885797356391]
```

As in *Example: Demag field in uniformly magnetised sphere* of our guided tour, we should obtain a constant magnetic induction of about [333333,0,0] [A/m]. Deviations from that value can be mainly ascribed to the discretisation errors of the finite element method (rather than the error due to the approximation with hierarchical matrices). To see this, we use `sphere_fullBEM.py` which carries out the same calculation but uses the normal full BEM. It reports:

```
x= -5e-09 : H_demag =  [-333065.71403658605, -5.2685406972238447, -55.70105442854085]
x= -4e-09 : H_demag =  [-333067.37484881631, -4.2116117445407726, -57.778611300679266]
x= -3e-09 : H_demag =  [-333068.83107133937, -3.7372238611028603, -59.825445387210245]
x= -2e-09 : H_demag =  [-333069.28217968839, -2.9635031726006642, -62.513814422201456]
x= -1e-09 : H_demag =  [-333067.6639511605, -1.5730916838594211, -66.546659227740889]
x= 0.0 : H_demag =  [-333066.04572263273, -0.18268019511817793, -70.579504033280344]
x= 1e-09 : H_demag =  [-333064.22835497675, 0.79797869001455679, -74.851480234723581]
x= 2e-09 : H_demag =  [-333060.20872696047, 2.9088218728650852, -77.0823444044496]
x= 3e-09 : H_demag =  [-333056.59267071093, 5.064110260421554, -80.187548021318634]
x= 4e-09 : H_demag =  [-333052.97641355224, 7.2199889195136837, -83.294534914159939]
x= 5e-09 : H_demag =  [-333051.27043353132, 9.4396856537516776, -85.662174893158024]
```

This shows that the error introduced by the HLib is of the order of 10 in 333333 (in this example). Note that the y and z component theoretically should be zero (for both calculations: with and without HLib), and that the error we see there (of the order of 60/333333 in the z-component) is coming from approximating the spherical shape with tetrahedra, and approximating the magnetisation with a piecewise linear function (not primarily from using the HLib approximation of the BEM).

HLib Memory usage

Nmag will also provide information on the memory requirements for the hierarchical matrix. First it will print to stdout (and here exceptionally not write to the log file) the following lines to the screen, which are each preceded by `HLib`:

```
HLib: Memory footprint of hierarchical matrix: 10.523720 MB.
HLib: Equivalent full matrix would require: 98.273628 MB.
HLib: The compression rate is 10.71%
```

The first line states the amount of memory required for the storage of the hierarchical matrix, the second one states the equivalent memory requirements when using the full boundary element matrix, and the last line gives the corresponding compression rate. Furthermore Nmag creates the file *memory_info.dat*, which in our example looks like:

```
Number of surface nodes:                3589
Size of hierarchical matrix:        10.52 MB
Total size of inadmissible leaves:   1.40 MB
Total size of admissible leaves:     8.96 MB
```

While the first two lines should be relatively self-explanatory, the third line states the total amount of memory needed to store the matrix blocks which cannot be approximated, while the fourth line gives the equivalent number for the approximated matrix blocks. Additionally, one can obtain the memory used for the hierarchical tree structure itself, by computing the difference between the size of the hierarchical matrix and the sum of the total sizes of the admissible and inadmissible leaves.

Changing the Parameters of HLib

Let us assume we want to run the simulation of the last section again, but this time we would like to reduce the time needed to assemble our hierarchical matrix. To achieve this, we coarsen the hierarchical tree by increasing the parameter `nmin` to 50, reassign the parameter `eps_aca` to 1e-5 in order to decrease the accuracy of the HCA II algorithm, and reduce the accuracy of the numerical integration by setting the parameter `quadorder` to 2.

To use non-default settings in a new script `sphere_hlib2.py` we add one line to create an `HMatrixSetup` object

```
#create an HLib object
hms = nmag.HMatrixSetup(nmin=50, eps_aca=1e-5, quadorder=2)
```

This object is then passed to the `Simulation` object:

```
sim = nmag.Simulation(phi_BEM=hms)
```

In order to make the time measurement you can just run the nsim command with a preceding 'time', i.e.

```
time nsim sphere_hlib2.py --clean
```

do the same with the sphere_hlib.py script, and compare the execution times. Alternatively, search for the string like "Populating BEM took 25.094362 seconds" in the log file/output. The execution time of the second script should be smaller (see also *Using HLib Example 2: Thin Films*).

For completeness: the Hdemag values computed with this script are:

```
x= -5e-09 : H_demag =  [-333060.73884748813, -5.7471691393211453, -56.164777361260889]
x= -4e-09 : H_demag =  [-333062.34355895646, -4.6973695734449556, -58.195233383426051]
x= -3e-09 : H_demag =  [-333063.7357911733, -4.2543955018989577, -60.199068292632305]
x= -2e-09 : H_demag =  [-333064.14913635491, -3.5107100192801424, -62.841949236542568]
x= -1e-09 : H_demag =  [-333062.54691465426, -2.1473409122582736, -66.824386136704007]
x= 0.0 : H_demag =  [-333060.94469295366, -0.78397180523640564, -70.806823036865438]
x= 1e-09 : H_demag =  [-333059.14023403701, 0.15188988623380831, -75.030255790251871]
x= 2e-09 : H_demag =  [-333055.17692864774, 2.2289146769013355, -77.213961296827563]
x= 3e-09 : H_demag =  [-333051.63216875959, 4.3434799953307275, -80.273150211395659]
x= 4e-09 : H_demag =  [-333048.08718075219, 6.4586908275326955, -83.334113807086638]
x= 5e-09 : H_demag =  [-333046.47566667694, 8.6375699926922742, -85.648195356633963]
```

2.24.5 Using HLib Example 2: Thin Films

In this example we consider square thin films with a thickness of 10 nm (in z-direction), and a varying edge length (in x and y directions) between 20 and 130 nm . The magnetisation within those films is initially homogeneously aligned and points out-of-plane. We then use Nmag's *relax* routine in order to evolve the magnetisation field to an energetically (meta-)stable state.

In order to estimate the efficiency benefits of hierarchical matrices, the simulations are executed twice: (i) with and (ii) without hierarchical matrices. Optimal damping is ensured by setting the damping constant of the LLG equation to 1. To increase the efficiency of the relaxation the tolerance of the time-stepper has been increased to 1e-5 (see *Example: Timestepper tolerances*).

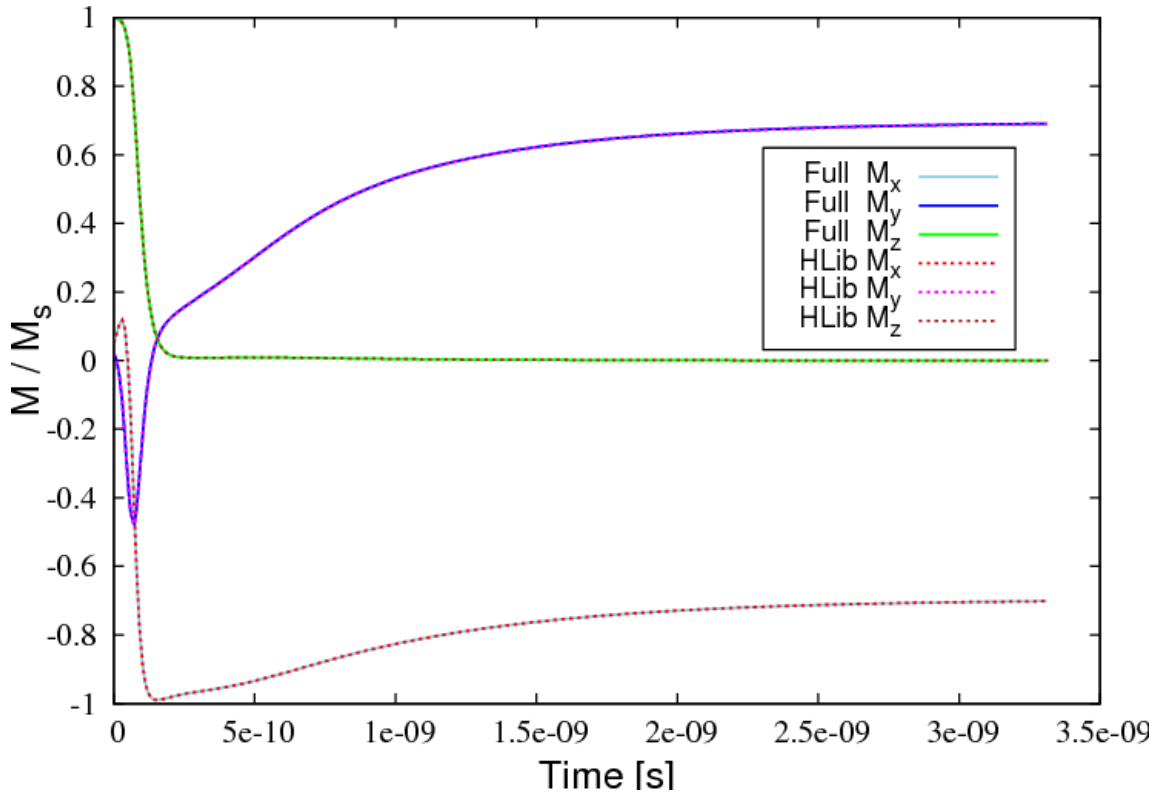
For our estimation of the efficiency we measure the time needed for the setup of our simulation (basically the time for populating the finite element and boundary element matrices), the time for relaxing the system, and the memory consumption at the end of the simulation, which should be roughly equal to the maximal value throughout the simulation.

For each film size and either use of the full BEM or the approximation through hierarchical matrices, a separate nsim script file (`thinfilm20_full.py`, `thinfilm40_full.py`, `thinfilm60_full.py`, ..., `thinfilm20_hlib.py`, etc.) has been written. It is important to start every simulation as a single process (by calling `nsim thinfilm20_full.py --clean` etc.), so that there are no overlaps in the memory access of different simulations. From every script a routine `run_simulation` which is imported from a local nsim module `simtools.py`, starts a simulation specified by its arguments (name of the simulation, name of the mesh file, name of hlib object in case hierarchical matrices are used, and the tolerance for the time integrator)

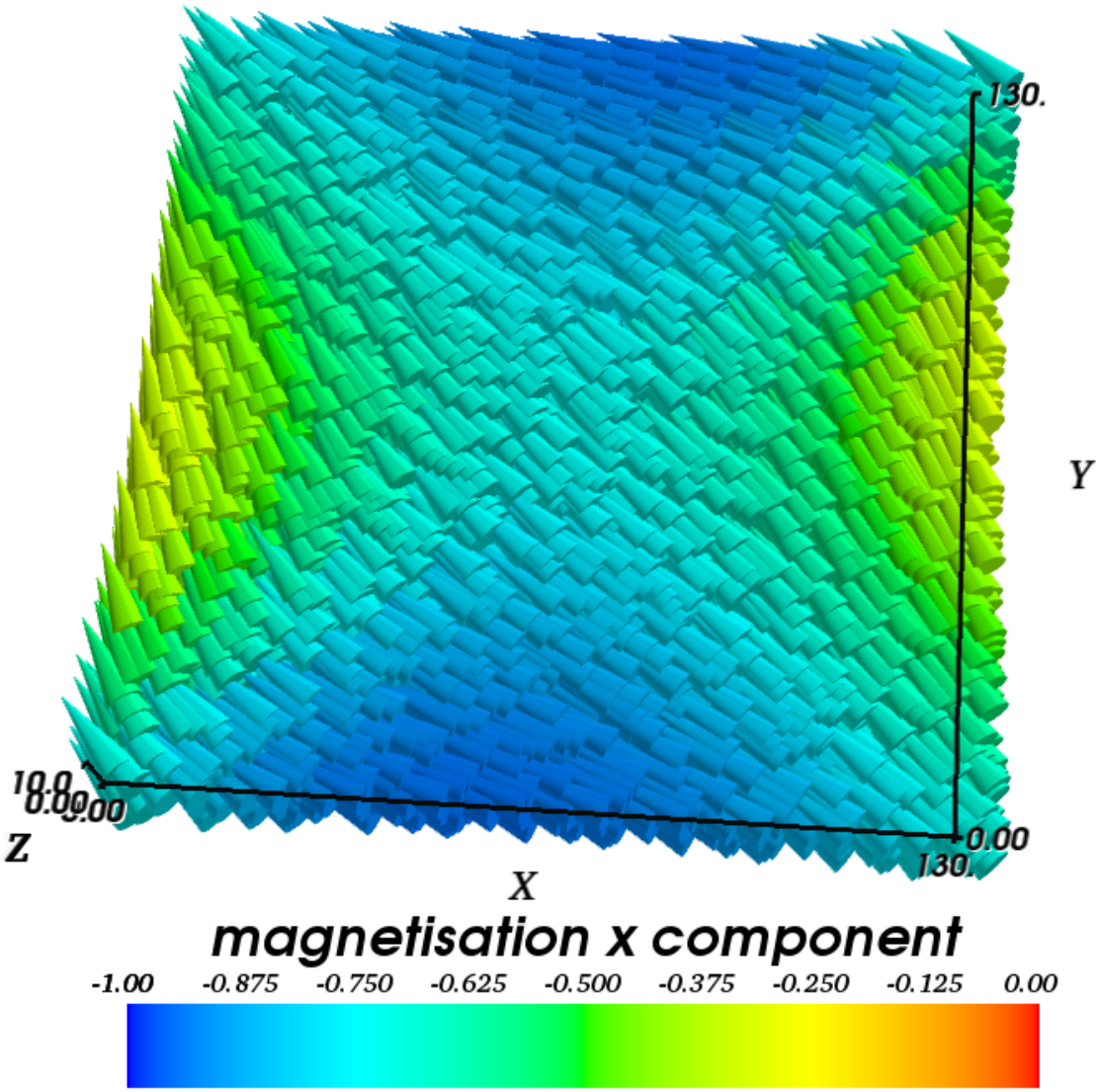
and returns the number of nodes of the mesh, the simulation's memory consumption and the setup- and relaxation times. These values are then written to a file `timings_hlib.dat` or `timings_full.dat`, respectively.

Beside extracting the information on the performance, it is also important to check, whether simulations using the full boundary element matrix and a hierarchical matrix approximation actually do the same, and that the simulated behaviour is physically correct.

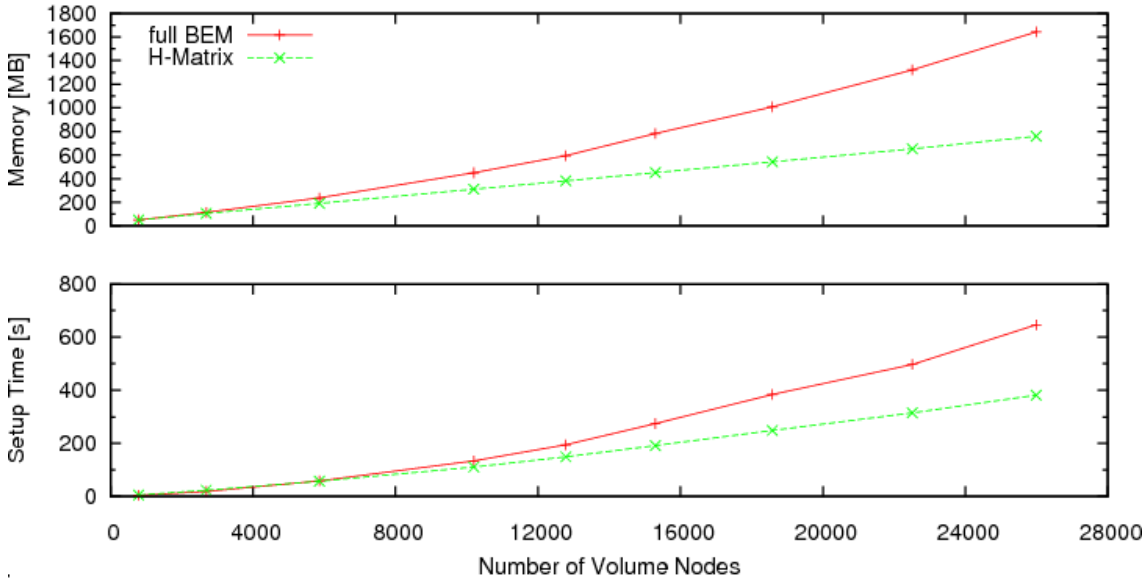
Looking at the spatially averaged magnetisation we find a very good agreement between both simulation types (example given for the film with an edge length of 100nm):



The magnetisation field moves from its out-of-plane configuration into the plane and relaxes into a high remanent state, which is aligned along the diagonal of the square base. The plot below shows a 3d visualisation of the relaxed magnetisation field (obtained with Mayavi2) for a thin film with an edge length of 130 nm.



We have run the simulations on a machine with an *AMD Athlon(tm) 64 X2 Dual Core Processor 3800+*, using only one core. The graphs below show the results of our efficiency test of hierarchical matrices. It can be seen that the memory requirements are reduced considerably. While the consumed memory increases (almost) linearly with the number of surface nodes N for the calculation with hierarchical matrices, the increase is of a higher order ($O(N^{4/3})$), when using the accurate boundary element matrix $\mathbf{B}$. The enhanced scaling behaviour allows for simulation of larger ferromagnetic structures. The graph on the memory consumption should enable users to estimate, whether they can simulate a certain structure with Nmag+HLib and the available hardware.



Besides the savings in memory, hierarchical matrices also reduce the time needed for the simulation setup considerably (see the bottom graph).

2.24.6 HLib and MPI

The HLib library that is available for academic use does not support parallel execution. It is thus stored on the master node, and cannot be distributed over several nodes. Simulations using the Hlib library can use MPI (for all other calculations).

2.25 Example: Calculation of dispersion curves

Nmag can be used to study the propagation of spin waves and to calculate dispersion curves. Here we consider a simulation script which shows how to do that. In particular, we study a long cylindrical wire made of Permalloy. We assume the magnetisation in the wire is relaxed along one axial direction (i.e. there are no domain walls inside the wire). One side of the wire is "perturbed" at time t=0 by a pulsed magnetic field. The spin waves generated on this side propagate towards the other. We want to study the propagation of spin waves and obtain the dispersion relation, i.e. a relation between the wave vector and the frequency of the spin-waves which propagate in the considered media. To calculate the dispersion relation we use the method developed by V. Kruglyak, M. Dvornik and O. Dmytriiev

In order to carry out such a numerical experiment, we first need to calculate the relaxed equilibrium magnetisation, i.e. the one which we perturb with the application of a pulsed field. Consequently, the simulation is split into two parts:

- In **part I**, the system is relaxed to obtain the initial magnetisation configuration for zero applied field. Such a state is then saved into a file to be used in part II;
- In **part II**, the magnetisation obtained in part I is loaded and used as the initial magnetisation configuration. A pulsed external magnetic field localised on one side of the wire is applied. The Landau-Lifshitz-Gilbert equation is integrated in time to compute the dynamical reaction to the applied stimulus. The configuration of the magnetisation is saved frequently to file, so that it can be studied and processed later.

The two parts are two simulations of the same system under different conditions. If we then decide to write two separate files for the two parts we end up duplicating the fragment of code which defines the materials and load the mesh. For this reason we split the simulation in three files:

- `"thesystem.py"`: defines the material which composes the nanowire and loads the mesh;
- `"relaxation.py"`: uses "thesystem.py" to setup the system and performs a relaxation with zero applied field. It saves the final magnetisation configuration to a file "m0.h5";

- "dynamics.py": uses "thesystem.py" to setup the system, loads the initial magnetisation configuration from the file "m0.h5" (produced by "relaxation.py"). It then applies a localised pulse of the applied magnetic field on one side of the wire and compute the dynamical response of the system saving the result to files.

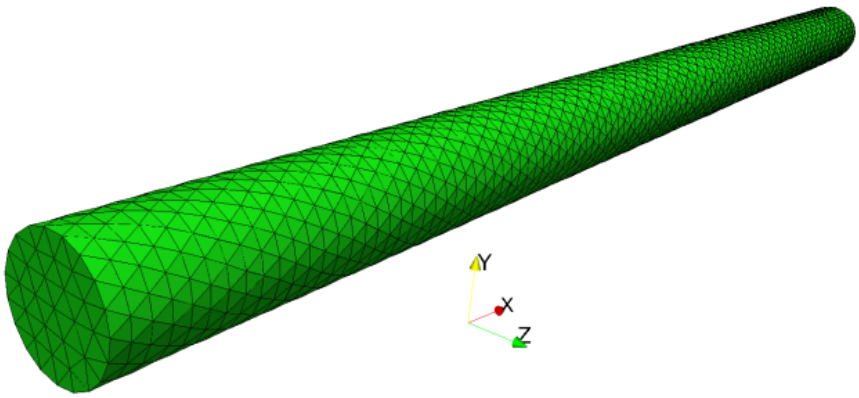
Consequently, in order to run the full simulation the user will have to type two commands on the shell, one for each part of the simulation:

```
$ nsim relaxation.py
$ nsim dynamics.py
```

(there is no need to type `nsim thesystem.py` as this file is implicitly "included" by the other two). In the next sections we will go through the three files which make up the numerical experiment.

2.25.1 The system: `thesystem.py`

The system under investigation is a cylindrical nanopillar of radius r=3 nm and length l=600 nm. The mesh file `cylinder.nmesh.h5` is obtained (using Netgen). (The figure shows a cylinder that is 100nm long.)



The geometry file given to Netgen `cylinder.geo` is shown below:

```
algebraic3d
solid nanopillar = cylinder (-300.0, 0, 0; 300.0, 0, 0; 3.0)
                   and plane (-300.0, 0, 0; -1, 0, 0)
                   and plane ( 300.0, 0, 0;  1, 0, 0)
                   -maxh=1.0;
tlo nanopillar;
```

The cylinder is made of Permalloy and as specified in the file `thesystem.py` shown below:

```
# In this file we define the material parameters and geometry of the system
# so that we can use it in two simulations: first during the relaxation,
# then during the dynamics

from nmag.common import *

nm = SI(1e-9, 'm')     # define nm as "nanometre"
ps = SI(1e-12, 's')    # define ps as "picosecond"

m0_filename = "m0.h5" # the file containing the equilibrium magnetisation

# A function which sets up the simulation with given name and damping
def simulate_nanowire(name=None, damping=0.5):
  permalloy = MagMaterial('Py',
                          Ms=SI(0.86e6, 'A/m'),
```

```
                        exchange_coupling=SI(13e-12, 'J/m'),
                        llg_damping=damping)

    s = Simulation(name)
    s.load_mesh("cylinder.nmesh.h5", [('nanopillar', permalloy)],
                unit_length=nm)
    return s
```

The file defines a few entities which are used in both the two parts of the simulations: `nm` is just an abbreviation for nanometer and similarly `ps` is an abbreviation for picosecond. `m0_filename` is the name of the file where the relaxed magnetisation will be saved (in part I) and loaded (in part II). Finally, the function `simulate_nanowire` deals with the portion of the setup which is common to both part I and part II. In particular, it defines a new material `permalloy`, creates a new simulation object `s`, load the mesh and associates to it the material `permalloy`. Such setup procedure is very similar to what has been encountered so far in the manual, the only element of novelty is that here we do it inside a function and return the created simulation object as a result of the function. The file defined here is not supposed to be run by itself, but rather to be used in part I and II.

2.25.2 Part I: `relaxation.py`

The source for the file `relaxation.py` is shown below:

```
# This script is used to compute the equilibrium configuration for the
# magnetisation, which is then used in the second part of the simulation,
# where the dynamics is actually studied.

from thesystem import simulate_nanowire, m0_filename, ps
from nmag.common import *

s = simulate_nanowire(name='relaxation', damping=0.5) # NOTE the high damping!
s.set_m([1, 0, 0])
s.relax(save=[('fields', at('time', 0*ps) | at('convergence'))])
s.save_restart_file(m0_filename)
```

The first two lines are used to import entities defined elsewhere. In particular, the first line in:

```
from thesystem import simulate_nanowire, m0_filename, ps
from nmag.common import *
```

tells to Python to load the file `thesystem.py` and "extract" from it the entities `simulate_nanowire`, `m0_filename`, `ps`. The second line does a similar thing and extracts all the quantities defined in the Nmag module `nmag.common`. This module defines some entities which are commonly used in simulations (such as `every`, `at`, `SI`, etc). The simulation is then set up using:

```
s = simulate_nanowire(name='relaxation', damping=0.5) # NOTE the high damping!
```

This line invokes the function `simulate_nanowire`, which does load the mesh and associate the material to it. The function returns the simulation object which is stored inside the variable `s` and is used in the following lines:

```
s.set_m([1, 0, 0])
s.relax(save=[('fields', at('time', 0*ps) | at('convergence'))])
s.save_restart_file(m0_filename)
```

Here we set the magnetisation along the axis of the nanopillar, we then relax the system to find the equilibrium magnetisation. We finally save such a configuration in the file `m0_filename`, i.e. with the name specified in `thesystem.py`.

2.25.3 Part II: `dynamics.py`

The source for the file `dynamics.py` is shown below:

```
from thesystem import simulate_nanowire, m0_filename, ps, nm
from nmag.common import *

# Details about the pulse
pulse_boundary = -300.0e-9 + 0.5e-9 # float in nm
pulse_direction = [0, 1, 0]
pulse_amplitude = SI(1e5, 'A/m')
pulse_duration = 1*ps

# Function which sets the magnetisation to zero
def switch_off_pulse(sim):
  sim.set_H_ext([0.0, 0.0, 0.0], unit=pulse_amplitude)

# Function which sets the pulse as a function of time/space
def switch_on_pulse(sim):
  def H_ext(r):
    if r[0] < pulse_boundary:
      return pulse_direction
    else:
      return [0.0, 0.0, 0.0]

  sim.set_H_ext(H_ext, unit=pulse_amplitude)

# Here we run the simulation: do=[....] is used to set the pulse
#   save=[...] is used to save the data.
s = simulate_nanowire('dynamics', 0.05)
s.load_m_from_h5file(m0_filename)
s.relax(save=[('fields', every('time', 0.5*ps))],
        do=[(switch_on_pulse, at('time', 0*ps)),
            (switch_off_pulse, at('time', pulse_duration)),
            ('exit', at('time', 200*ps))])
```

The simulation starts again importing a few entities from the file `thesystem.py`. In particular, the function `simulate_nanowire` is used to setup the system similarly to what was done for the relaxation. The next few lines define some variables which are used to define the geometry and duration of the magnetic field pulse:

```
# Details about the pulse
pulse_boundary = -300.0e-9 + 0.5e-9 # float in nm
pulse_direction = [0, 1, 0]
pulse_amplitude = SI(1e5, 'A/m')
pulse_duration = 1*ps
```

In this example the pulse is obtained by switching on the applied field (from (0, 0, 0) to (0, `pulse_amplitude`, 0)) in the region of the wire where x < `pulse_boundary`, which corresponds in this case to a layer of 0.5 nm thickness on one side of the cylinder. The pulse is switched on at t=0 and switched off at `pulse_duration`. We now examine the code and explain how all this is coded in the script. We start explaining the last few lines:

```
# Here we run the simulation: do=[....] is used to set the pulse
#   save=[...] is used to save the data.
s = simulate_nanowire('dynamics', 0.05)
s.load_m_from_h5file(m0_filename)
s.relax(save=[('fields', every('time', 0.5*ps))],
        do=[(set_pulse, at('time', 0*ps)),
            (set_to_zero, at('time', pulse_duration)),
            ('exit', at('time', 200*ps))])
```

Here we use the `simulate_nanowire` function which we defined in the file `thesystem.py` to setup the system and the materials. We then set the initial magnetisation configuration from the file saved in part I and carry out the time integration by calling the `relax` method of the simulation object `s`. The pulse is switched on and switched off by the instruction passed in the `do=[...]` argument. In particular, the argument `do` accepts a list of pairs (things to be done, at a given time). The code:

```
do=[(switch_on_pulse, at('time', 0*ps)),
    (switch_off_pulse, at('time', pulse_duration)),
    ('exit', at('time', 200*ps))]
```

specifies that:

- the function `switch_on_pulse` should be executed at time t=0 ps;
- the function `switch_off_pulse` should be executed at time t= `pulse_duration`;
- the simulation should terminate at time t=200 ps.

At the same time the argument `save=[('fields', every('time', 0.5*ps))]` of the relax method saves the field every 0.5 ps.

Let's now see how the pulse is actually switched on and off. To switch off the pulse we provide the function:

```
# Function which sets the magnetisation to zero
def switch_off_pulse(sim):
  sim.set_H_ext([0.0, 0.0, 0.0], unit=pulse_amplitude)
```

The function gets the simulation object, `sim`, as an argument and uses it together with the method `set_H_ext` to set the applied magnetic field to zero everywhere. The function to set up the simulation is a little bit more complicated:

```
# Function which sets the pulse as a function of time/space
def switch_on_pulse(sim):
  def H_ext(r):
    if r[0] < pulse_boundary:
      return pulse_direction
    else:
      return [0.0, 0.0, 0.0]

  sim.set_H_ext(H_ext, unit=pulse_amplitude)
```

Indeed, being the pulse localised (and hence non-uniform) in space, we need to define a function to be given to `set_H_ext`. The function checks whether the x component in the given point is lower than `pulse_amplitude` and sets the applied field to a value differt from zero only if that is really the case.

2.25.4 Postprocessing the data

Once the simulations are finished, the data (i.e. the values of the magnetisation saved every 0.5 ps) can be extracted from the file `dynamics_dat.h5` and postprocessed. We use the `nmagprobe` command for this. `nmagprobe` can perform several postprocessing tasks (detailed documentation can be obtained by typing `nmagprobe --help`). In this context it is used to probe the magnetisation along the axis of the cylinder at regular intervals of time. The values extracted are then Fourier transformed. The command we use is the following:

```
nmagprobe --verbose dynamics_dat.h5 --field=m_Py \
  --time=0,100e-12,101 --space=-300,300,201/0/0 --ref-time=0.0 \
  --scalar-mode=component,1 --out=real-space.dat \
  --ft-axes=0,1 --ft-out=norm --ft-out=rec-space.dat
```

Here we extract data for the magnetisation (option `--field=m_Py`) from the file `dynamics_dat.h5`.

- We probe the field over a cubic lattice in space and time. The lattice is four dimensional and consists of the points (t, x, y, z) with t=0, 1 ps, 2 ps, ..., 100 ps (101 values), x=-300 nm, -297 nm, -294 nm, ..., 300 nm (201 values) and y=z=0. The lattice is fully determined by the options `--time` and `--space`. In particular, the option `--time=0,100e-12,101` states that the lattice consists of 101 equispaced values going from 0 to 100e-12 s. The option `--space` accepts a similar expressions for each spatial coordinate separated by `/`;
- `--ref-time=0.0` tells to `nmagprobe` that after extracting the values for the field, m(t, x, y, z), it should compute the difference with respect to the given time, i.e. dm(t, x, y, z) = m(t, x, y, z) - m(0, x, y, z). We

add this option to `nmagprobe` because we are interested in the variation of the magnetisation with respect to the equilibrium configuration (t=0) rather than on its "absolute" value;

- `--scalar-mode=component,1` induces `nmagprobe` to extract the y component of dm and to use it as a scalar when writing the output and when doing the fourier transform (to extract the x component one should use `--scalar-mode=component,0`); We could also write `--scalar-mode=component,y` and `--scalar-mode=component,x`.
- `--out=real-space.dat` induces `nmagprobe` to save to file the data selected by the options discussed so far. In particular, the file `real-space.dat` will be filled with the values of the y-component of dm(t, x, y, z) along the selected lattice. That will be a text file which can be inspected with a text editor and used within plotting programs such as Gnuplot;
- `--ft-axes=0,1` specifies that the selected data should be Fourier transformed along the axis 0 (time) and 1 (x-space). This can also be written as `--ft-axes=t,x`.
- `--ft-out=norm` induces `nmagprobe` to compute the norm of the complex numbers coming from the Fourier-transform. These are the values which are finally saved to file;
- `--ft-out=rec-space.dat` specifies the output file for the Fourier-transformed data.

The command creates two files: `real-space.dat`, containing the y component of the magnetisation variation as a function of time and space, and `rec-space.dat`, containing the Fourier transform of such a quantity.

To plot the data in the two files we use the Gnuplot script `plot.gnp`:

```
set term png
set pm3d map

set out "real-space.png"
set title "y component of magnetisation variation"
set xlabel "position in axis (nm)"
set ylabel "time (ps)"
splot [] [0:] [-0.001:0.001] 'real-space.dat' u ($2):($1/1e-12):5 t ""

set out "rec-space.png"
set title "Fourier transform"
set xlabel "k (1/nm)"
set ylabel "omega (GHz)"
splot [] [0:] [0:1.7e-8] 'rec-space.dat' u (-$2):($1/(2*pi*1e9)):5 t ""
```

Here is the result after running the script with Gnuplot.

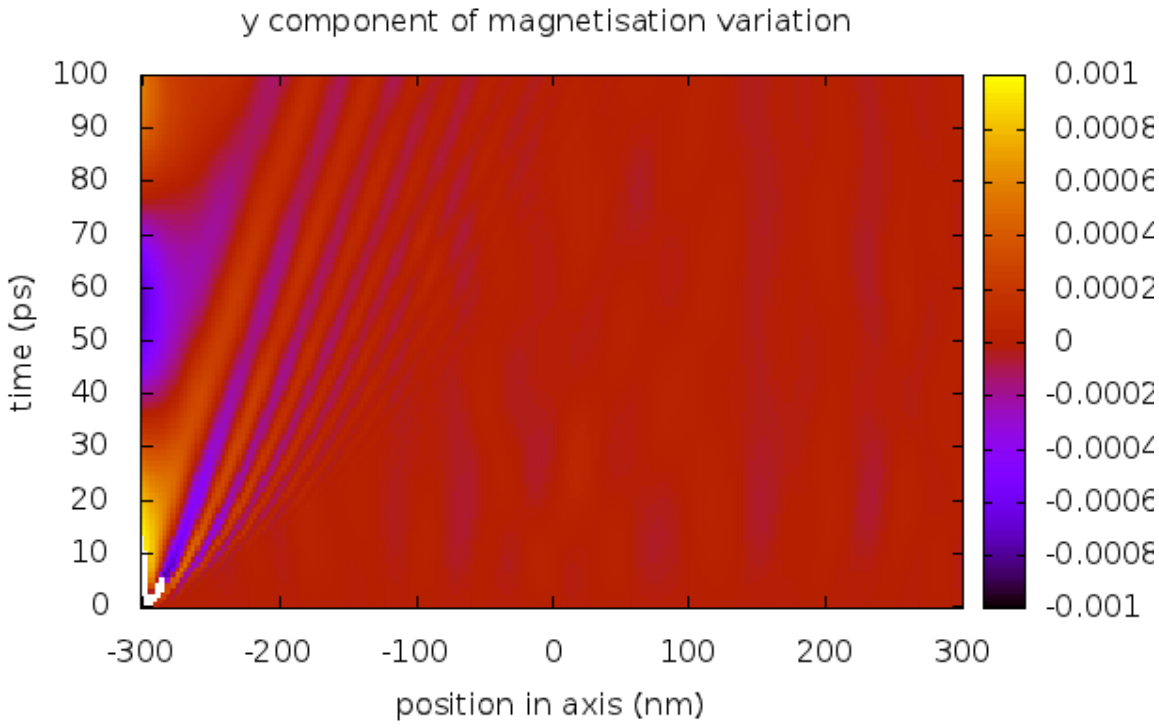
2.26 Example: Timestepper tolerances

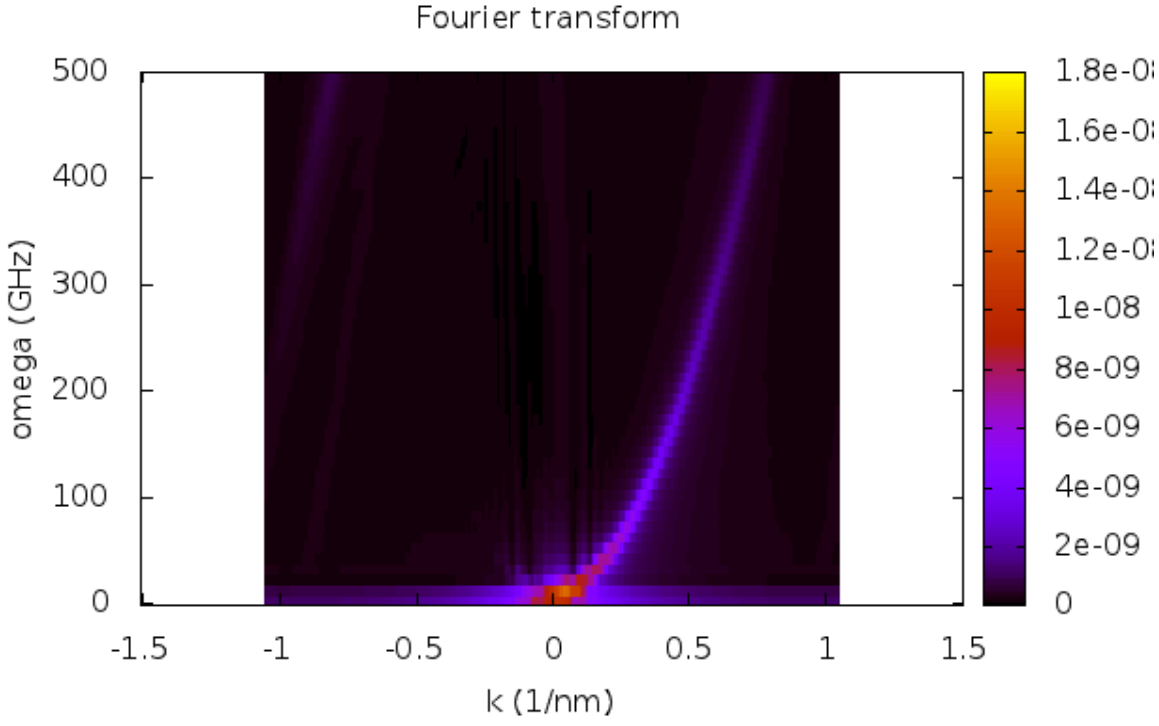
The tolerance settings of a simulation can greatly affect the performance, the accuracy and the usefulness of a simulation. Section *Solvers and tolerance settings* provides an overview. In this example, we demonstrate

- how the time integrator's tolerances can be set and
- how these tolerances affect the simulation results and performance.

The time integrator we use is the PVODE solver from the SUNDIALS package. It is optimised to deal with stiff systems of ordinary differential equations and is therefore very suited for micromagnetic simulations. It can also execute in parallel (i.e. across several CPUs at the same time using MPI). The computational challenge of the time integration lies in the different time scales associated with the (fast) exchange field and the (slower) demagnetisation field.

Sundials provides two parameters `rtol` and `atol` (see sundials documentation) to control the required accuracy of the calculations. Sundials uses these parameters to determine the number of iterations required to simulate a given amount of real time (for example one pico second). Equivalently, these parameters determine the amount of real time that can be simulated per iteration.


y component of magnetisation variation
time (ps)
position in axis (nm)
100
90
80
70
60
50
40
30
20
10
0
-300
-200
-100
0
100
200
300
0.001
0.0008
0.0006
0.0004
0.0002
0
-0.0002
-0.0004
-0.0006
-0.0008
-0.001


Fourier transform
omega (GHz)
k (1/nm)
500
400
300
200
100
0
-1.5
-1
-0.5
0
0.5
1
1.5
1.8e-08
1.6e-08
1.4e-08
1.2e-08
1e-08
8e-09
6e-09
4e-09
2e-09
0

It is common that the amount of time simulated per iteration varies throughout a simulation as different time step sizes are required to resolve the physics to the same accuracy level. (The *Data files (.ndt)* data file contains one column `last_step_dt` which provides the size of the time step. Use *ncol* to extract this data conveniently.)

The sundials tolerance parameters `rtol` and `atol` can be set in nmag using the `ts_rel_tol` and `ts_abs_tol` arguments in the *set_params* function. (The letters `ts` in `ts_rel_tol` and `ts_abs_tol` stand for Time Stepper).

The integration of the Landau Lifshitz and Gilbert equation is carried out on the *normalised* magnetisation, and the corresponding field (see *Fields and Subfields in Nmag*) is called `m` (the magnetisation with the saturation magnetisation magnitude is called capital `M` in nmag). Because this field is normalised, we set `rtol` and `atol` to the same value in this example, and refer to the value just as `tol`.

We use the program `bar_tol.py` that:

- re-uses the bar studied in *Example 2: Computing the time development of a system* but
- carries out the time integration for a number of different tolerance values.

```
import nmag
from nmag import SI

import time #python standard modules, used to measure run time

def run_sim(tol):
    """Function that is called repeatedly with different tolerance values.
    Each function call is carrying out one simulation.
    """
    mat_Py = nmag.MagMaterial(name="Py",
                              Ms=SI(0.86e6,"A/m"),
                              exchange_coupling=SI(13.0e-12, "J/m"),
                              llg_damping=0.5)

    #compose name of simulation to inlude value of tolerance
    sim = nmag.Simulation("bar_%.6f" % tol)

    sim.load_mesh("bar30_30_100.nmesh.h5",
                  [("Py", mat_Py)],
                  unit_length=SI(1e-9,"m"))

    sim.set_m([1,0,1])

    #set tolerance (has to be called after set_m())
    sim.set_params(ts_abs_tol=tol, ts_rel_tol=tol)

    dt = SI(2.5e-12, "s")

    timing = 0 #initialise variable to measure execution time

    for i in range(0, 121):
        timing -= time.time()     #start measuring time
        sim.advance_time(dt*i)    #compute time development for 300ps
        timing += time.time()     #stop measuring time
                                  #we exclude time required to save data

        sim.save_data()           #save averages every 2.5 ps

    #at end of simulation, write performance data into summary file
    f=open('resultsummary.txt','a') #open file to append
    f.write('%g %d %g\n' % (tol,sim.clock['step'],timing))
    f.close()


#main program
tols = [1e-1,1e-2,1e-3,1e-4,1e-5,1e-6]
```

```
for tol in tols:
    run_sim(tol)
```

From a conceptual point of view, we see something new here: the section of the code that starts with:

```
def run_sim(tol):
```

defines a function with name `run_sim` which will carry out a complete simulation every time it is called. It takes one argument: the parameter `tol`. The simulation name (which is re-used in the name of the *Data files (.ndt)* data file) contains the value of `tol`. For example, if the `tol=0.1`, then the name of the simulation is `bar_0.100000` and the name of the ndt data file is `bar_0.100000_dat.ndt`. We can thus call this function repeatedly for different values of `tol`, and each time a complete simulation will be run and new data files created. [10]

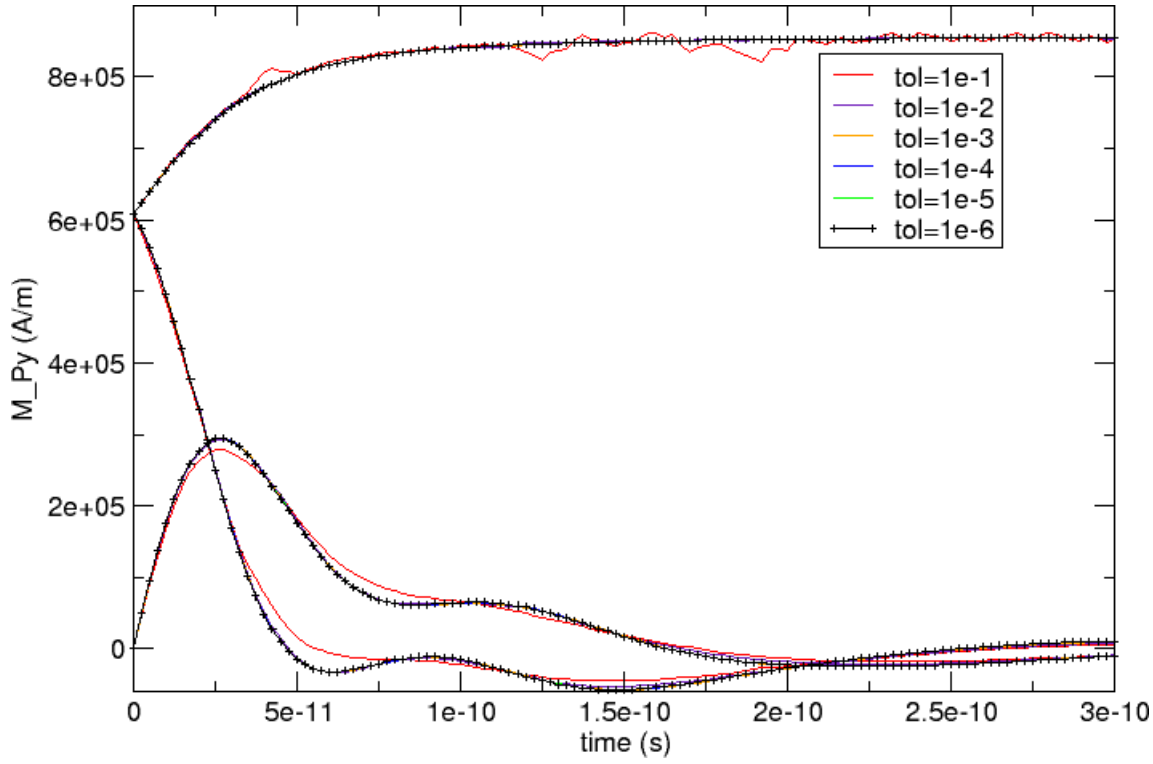
The main loop of the script:

```
#main program
tols = [1e-1,1e-2,1e-3,1e-4,1e-5,1e-6,1.0]

for tol in tols:
    run_sim(tol)
```

simply iterates over values `0.1`, `0.01`, `0.001`, `0.0001`, `0.00001` and `0.000001` and calls the function `run_sim` with a different tolerance value in every iteration of the for-loop.

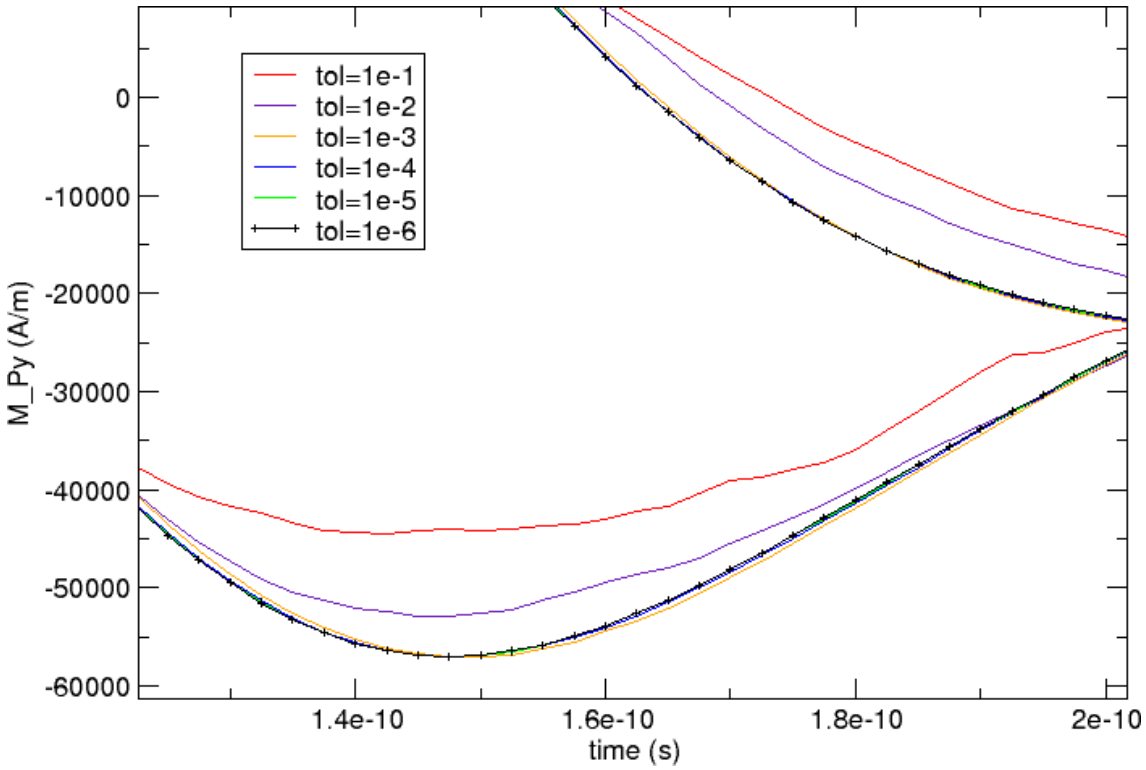
Once the program has finished, we have data files `bar_0.000001_dat.ndt`, `bar_0.000010_dat.ndt`, ... and `bar_0.100000_dat.ndt` that can be analysed and plotted in the usual way.

We show a plot of the x, y and z components of the magnetisation against time (as in *Example 2: Computing the time development of a system*) for each of the tolerance values. The run with `tol=1e-6` is the most accurate, and the corresponding black line has been tagged with little + characters.



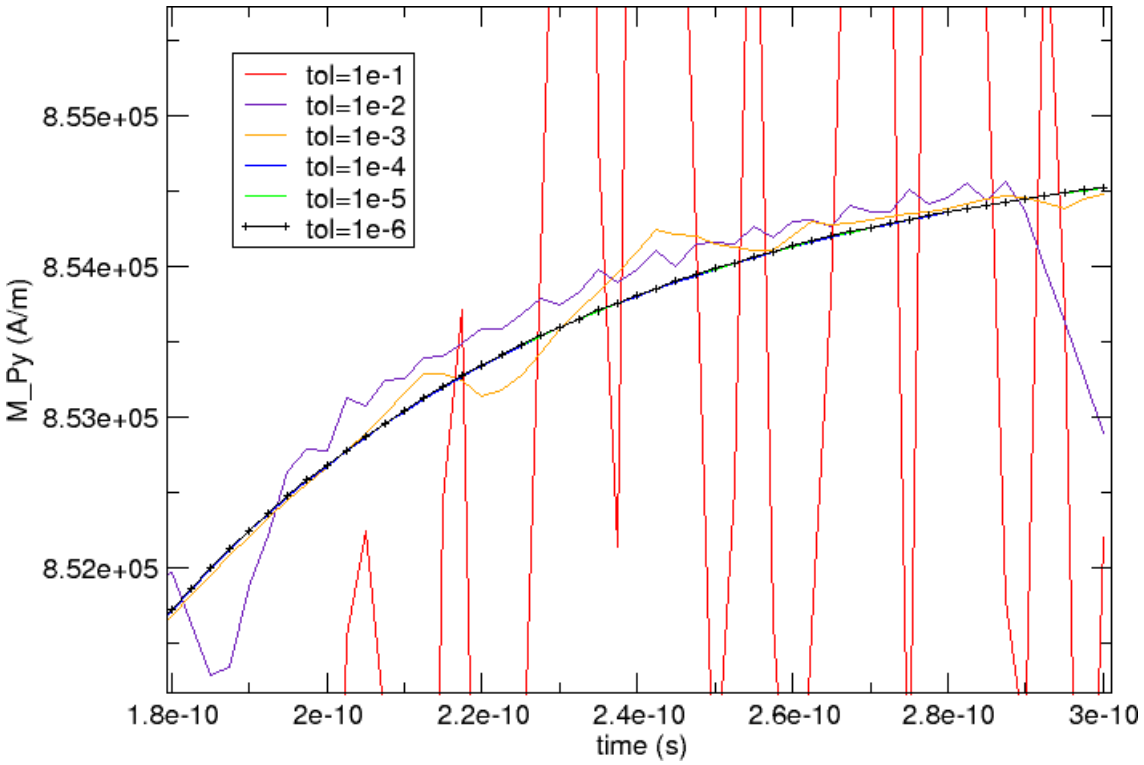
[10] We could, in fact, avoid re-creating all the operator matrices and the BEM, and just repeat the simulation with varying values of the `tol` parameter. However, this would mean that the data is written into the same file (so is slightly less convenient here). It would also be a less pedagogical example in this guided tour.

We can see that curves seem to coincide (at this scale) apart from the red `tol=1e-1` curve which deviates somewhat. We zoom in to region between 1.2e-10 seconds and 2e-10 seconds and focus on the lowers curves in the main plot:



The better resolution reveals that there is a clear deviation of the various curves: the red (0.1), indigo (0.01) and yellow (1e-3) curves approach the black (1e-6) curve in this order. The blue (1e-4) and green (1e-5) curves appear to coincide with the black reference curve.

Another zoom at the z-component of the magnetisation towards the end of the simulated time interval (time>1.8e-10 seconds) shows that the less accurate curves (red, and then indigo and yellow) show a large amount of jitter (although following the reference curve *on average*).



We conclude that we should use a tolerance of at most 1e-3 for this simulation; better 1e-4 or smaller.

In simulation work, we are of course interested to get the most accurate simulation results. However, in reality this is conflicting with the increased run time that is associated with more accurate simulations. In this example, we have written some performance data into `resultssummary.txt`. Reformatted, postprocessed and the rows re-ordered, this is the data complete with table headings:

tol	steps	CPU time (s)	CPU time per step (s)
0.000001	740	120.81	0.163
0.000010	356	62.37	0.175
0.000100	182	46.10	0.253
0.001000	119	66.36	0.558
0.010000	114	92.08	0.808
0.100000	88	94.69	1.076

The accuracy of the simulation results decreases from the top of the table downwards. We know from the graphs above that we should use a tolerance setting of 1e-4 or smaller to obtain fairly accurate results (assuming that the 1e-6 curve is used as a reference).

The number of iterations required increases from the tolerance 1e-4 to tolerance 1e-6 by a factor of 4 while the total CPU time increases by a factor of 2.6.

Looking at the greater tolerances 1e-3 and 0.01, we see that while the number of iterations required decreases, the CPU time is increasing. This is the first indication that at this tolerance level the system becomes difficult to treat efficiently by sundials (it basically appears to be noisy and stochastic equations are hard to integrate).

In summary,

- to minimise the simulation time, we need to choose a tolerance value as large as "possible".
- The definition of "possible" will depend on the context. A good way of obtaining a suitable tolerance value is to run the same simulation repeatedly with decreasing tolerance values. Once the resulting curves converge (as a function of decreasing tolerance settings), a good tolerance level has been found. (This would be 1e-4 for the example shown here.)

- Choosing the tolerance values to be too large, can be counter productive (and take much more CPU time than the lower accuracy level).
- The default value for the sundials tolerances is shown in the documentation of *set_params*. A simulation can often be accelerated significantly by increasing this value.
- A change of the tolerances has to be considered together with the convergence criterion for hysterises loop calculations (see next section: Hysteris loop calculation not converging? A word of warning ...)

2.26.1 Hysteris loop calculation not converging? A word of warning ...

The *hysteresis* and the *relax* command need to have a criterion how to decide when the simulation has reached a (meta)stable state and when the relaxation (at a given applied field) should be considered to have been reached. A common approach (which is used by OOMMF and nmag, for example) is to monitor the change of the (normalised) magnetisation with respect to time (i.e. dm/dt). If the absolute value of this drops below a given threshold, then one considers the system as converged (the *relax* command will return at this point, while the *hysteresis* command will move to the next field). This threshold can be changed from its default value with the *set_params* simulation method (the attribute is *stopping_dm_dt*).

The choice of the tolerances (*ts_rel_tol* and *ts_abs_tol*) *must* respect the chosen *stopping_dm_dt* value (or conversely the *stopping_dm_dt* needs to be adapted to work with the chosen tolerances): large values for the tolerances correspond to lower accuracy and to larger random fluctuations of dm/dt, which consequently may never become lower than *stopping_dm_dt*. In such a case the simulation never returns from the *relax* command, because the convergence criterion is never satisfied.

In all the examples we have studied, we have found that the default parameters work fine. However, if you find that a simulation never returns from the *hysteresis* or *relax* command, it is worth reducing the tolerances for the time stepper (on increasing *stopping_dm_dt*) to see whether this resolves the problem).

2.27 Example: Parallel execution (MPI)

Nmag‘s numerical core (which is part of the nsim multi-physics library) has been designed to carry out numerical computation on several CPUs simultaneously. The protocol that we are using for this is the wide spread Message Passing Interface (MPI). There are a number of MPI implementations; the best known ones are probably MPICH1, MPICH2 and LAM-MPI. Currently, we support MPICH1 and MPICH2.

Which mpi version to use? Whether you want to use mpich1 or mpich2 will depend on your installation: currently, the installation from source provides mpich2 (which is also used in the virtual machines) whereas the Debian package relies on mpich1 (no Debian package is provided after release 0.1-6163).

2.27.1 Using mpich2

Before the actual simulation is started, a *multi-purpose daemon* must be started when using MPICH2.

The ‘‘.mpd.conf‘‘ file

MPICH2 will look for a configuration file with name `.mpd.conf` in the user’s home directory. If this is missing, an attempt to start the multi-purpose daemon, will result in an error message like this:

```
$> mpd
configuration file /Users/fangohr/.mpd.conf not found
A file named .mpd.conf file must be present in the user's home
directory (/etc/mpd.conf if root) with read and write access
only for the user, and must contain at least a line with:
MPD_SECRETWORD=<secretword>
```

```
One way to safely create this file is to do the following:
  cd $HOME
  touch .mpd.conf
  chmod 600 .mpd.conf
and then use an editor to insert a line like
  MPD_SECRETWORD=mr45-j9z
into the file.  (Of course use some other secret word than mr45-j9z.)
```

If you don't have this file in your home directory, just follow the instructions above to create it with some secret word of your choice (Note that the above example is from a Mac OS X system: on Linux the home directory is usually under `/home/USERNAME` rather than `/Users/USERNAME` as shown here.)

Let's assume we have a multi-core machine with more than one CPU. This makes the mpi setup slightly easier, and is also likely to be more efficient than running a job across the network between difference machines.

First, we need to start the multi-purpose daemon:

```
$> mpd &
```

It will look for the file `~/.mpd.conf` as described above. If found, it will start silently. Otherwise it will complain.

Testing that nsim executes in parallel

First, let's make sure that `nsim` is in the search path. The command `which nsim` will return the location of the executable if it can be found in the search path. For example:

```
$> which nsim
/home/fangohr/new/nmag-0.1/bin/nsim
```

To execute nsim using two processes, we can use the command:

```
$> mpiexec -n 2 nsim
```

There are two useful commands to check whether nsim is aware of the intended MPI setup. The fist one is `ocaml.mpi_status()` which provides the total number of processes in the MPI set-up:

```
$> mpiexec -n 2 nsim
>>> ocaml.mpi_status()
MPI-status: There are 2 nodes (this is the master, rank=0)
>>>
```

The other command is `ocaml.mpi_hello()` and prints a short 'hello' from all processes:

```
>>> ocaml.mpi_hello()
>>> [Node   0/2] Hello from beta.kk.soton.ac.uk
[Node   1/2] Hello from beta.kk.soton.ac.uk
```

For comparison, let's look at the output of these commands if we start `nsim` *without* MPI, in which case only one MPI node is reported:

```
$> nsim
>>> ocaml.mpi_status()
MPI-status: There are 1 nodes (this is the master, rank=0)
>>> ocaml.mpi_hello()
[Node   0/1] Hello from beta.kk.soton.ac.uk
```

Assuming this all works, we can now start the actual simulation. To use two CPUs on the local machine to run the `bar30_30_100.py` program, we can use:

```
$> mpiexec -n 2 nsim bar30_30_100.py
```

To run the program again, using 4 CPUs on the local machine:

```
$> mpiexec -n 4 nsim bar30_30_100.py
```

Note that mpich2 (and mpich1) will spawn more processes than there are CPUs if necessary. I.e. if you are working on some Intel Dual Core processor (with 2 CPUs and one core each) but request to run your program with 4 (via the `-n 4` switch given to `mpiexec`), than you will have 4 processes running on the 2 CPUs.

If you want to stop the `mpd` daemon, you can use:

```
$> mpdallexit
```

For diagnostic purposes, the `mpdtrace` command can be use to track whether a multipurpose daemon is running (and which machines are part of the *mpi-ring*).

Advanced usage of mpich2

To run a job across different machines, one needs to start the multi-purpose daemons on the other machines with the `mpdboot` command. This will search for a file (in the current directory) with name `mpd.hosts` which should contain a list of hosts to participate (very similar to the `machinefile` in MPICH1).

To trace which process is sending what messages to the standard out, one can add the `-l` switch to the `mpiexec` command: then each line of standard output will be preceded by the rank of the process who has issued the message.

Please refer to the official MPICH2 documentation for further details.

2.27.2 Using mpich1

Note: Most users will use MPICH2 (if they have compiled Nmag from the tar-ball): see *Using mpich2*

Suppose we would like to run *Example 2: Computing the time development of a system* of the manual with 2 processors using MPICH1. We need to know the full path to the `nsim` executable. In a `bash` environment (this is pretty much the standard on Linux and Mac OS X nowadays), you can find the path using the `which` command. On a system where nsim was installed from the Debian package:

```
$> which nsim
/usr/bin/nsim
```

Let's assume we have a multi-core machine with more than one CPU. This makes the mpi setup slightly easier, and is also likely to be more efficient than running a job across the network between difference machines. In that case, we can run the example on 2 CPUs using:

```
$> mpirun -np 2 /usr/bin/nsim bar30_30_100.py
```

where `-np` is the command line argument for the Number of Processors.

To check that the code is running on more than one CPU, one of the first few log messages will display (in addition to the runid of the simulation) the number of CPUs used:

```
$> mpirun -np 2 `which nsim` bar30_30_100.py

    nmag:2008-05-20 12:50:01,177   setup.py  269    INFO Runid (=name simulation) is 'bar30_30_10
```

To use 4 processors (if we have a quad core machine available), we would use:

```
$> mpirun -np 4 /usr/bin/nsim bar30_30_100.py
```

Assuming that the `nsim` executable is in the path, and that we are using a bash-shell, we could shortcut the step of finding the `nsim` executable by writing:

```
$> mpirun -np 4 `which nsim` bar30_30_100.py
```

To run the job across the network on different machines simultaneously, we need to create a file with the names of the hosts that should be used for the parallel execution of the program. If you intend to use nmag on a cluster, your cluster administrator should explain where to find this machine file.

To distribute a job across `machine1.mydomain`, `machine2.mydomain`, and `machine3.mydomain` we need to create the file `machines.txt` with content:

```
machine1.mydomain
machine2.mydomain
machine3.mydomain
```

We then need to pass the name of this file to the `mpirun` command to run a (mpi-enabled) executable with mpich:

```
mpirun -machinefile machines.txt -np 3 /usr/bin/nsim bar30_30_100.py
```

For further details, please refer to the MPICH1 documentation.

2.27.3 Visualising the partition of the mesh

We use Metis to partition the mesh. Partitioning means to allocate certain mesh nodes to certain CPUs. Generally, it is good if nodes that are spatially close to each other are assigned to the same CPU.

Here we demonstrate how the chosen partition can be visualised. As an example, we use the *Example: demag field in uniformly magnetised sphere*. We are *Using mpich2*:

```
$> mpd &
$> mpiexec -l -n 3 nsim sphere1.py
```

The program starts, and prints the chose partition to stdout:

```
 nfem.ocaml:2008-05-28 15:11:07,757    INFO Calling ParMETIS to partition the me
sh among 3 processors
 nfem.ocaml:2008-05-28 15:11:07,765    INFO Processor 0: 177 nodes
 nfem.ocaml:2008-05-28 15:11:07,765    INFO Processor 1: 185 nodes
 nfem.ocaml:2008-05-28 15:11:07,766    INFO Processor 2: 178 nodes
```

If you can't find the information on the screen (=stdout), then have a look in `sphere1_log.log` which contains a copy of the log messages that have been printed to stdout.

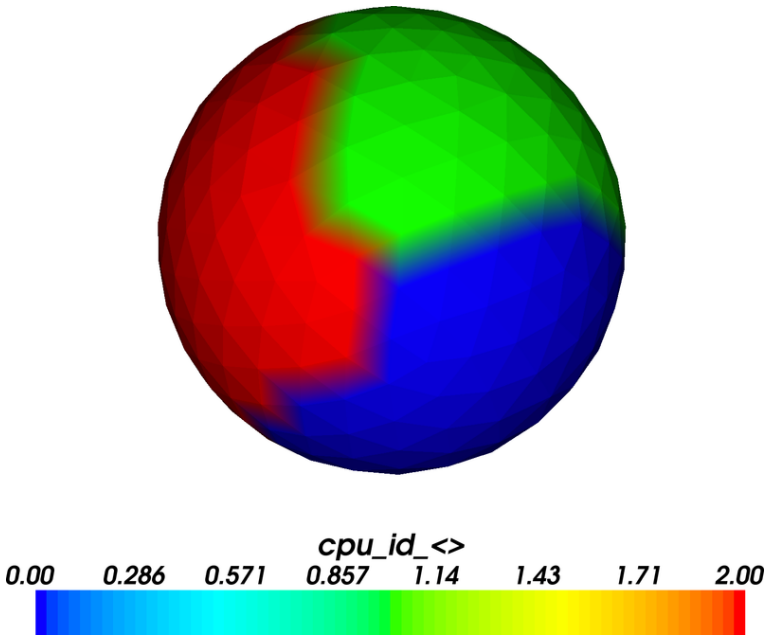
If we save any fields spatially resolved (as with the `sim.save_data(fields='all')` command), then nmag will create a file with name (in this case) `sphere1_dat.h5`. In addition to the field data that is saved, it also stores the finite element mesh *in the order that was used when the file was created*. In this example, this is the mesh ordered according to the output from the ParMETIS package. The first 177 nodes of the mesh in this order are assigned to CPU0, the next 185 are assigned to CPU1, and the next 178 are assigned to CPU2.

We can visualise this partition using the *nmeshpp* command (which we apply here to the mesh that is saved in the `sphere1_dat.h5` file):

```
$> nmeshpp --partitioning=[177,185,178] sphere1_dat.h5 partitioning.vtk
```

The new file `partitioning.vtk` contains only one field on the mesh, and this has assigned to each mesh node the id of the associated CPU. We can visualise this, for example, using:

```
$> mayavi -d partitioning.vtk -m SurfaceMap
```



The figure shows that the sphere has been divided into three areas which carry values 0, 1 and 2 (corresponding to the MPI CPU rank which goes from 0 to 2 for 3 CPUs). Actually, in this plot we can only see the surface nodes (but the volume nodes have been partitioned accordingly).

The process described here is a bit cumbersome to visualise the partition. This could in principle be streamlined (so that we save the partition data into the `_dat.h5` data file and can generate the visualisation without further manual intervention). However, we expect that this is not a show stopper and will dedicate our time to more pressing issues. (User feedback and suggestions for improvements are of course always welcome.)

2.27.4 Performance

Here is some data we have obtained on an IBM x440 system (with eight 1.9Ghz Intel Xeon processors). We use a test simulation (located in `tests/devtests/nmag/hyst/hyst.par`) which computes a hysteresis loop for a fairly small system (4114 mesh nodes, 1522 surface nodes, BEM size 18MB). We use overdamped time integration to determine the meta-stable states.

Both the setup and the time required to write data will not become significantly faster when run on more than one CPU. We provide:

total time: this includes setup time, time for the main simulation loop and time for writing data (measured in seconds)

total speedup: The speed up for the total execution time (i.e. ratio of execution time on one CPU to execution time on n CPUs).

sim time: this is the time spend in the main simulation loop (and this is where expect a speed up)

sim speedup: the speedup of the main simulation loop

CPUs	total time	total speedup	sim time	sim speedup
1	4165	1.00	3939	1.00
2	2249	1.85	2042	1.93
3	1867	2.23	1659	2.37
4	1605	2.60	1393	2.83

The numbers shown here have been obtained using mpich2 (and using the `ssm` device instead of the default `sock` device: this is available on Linux and resulted in a 5% reduction of execution time).

Generally, the (network) communication that is required between the nodes will slow down the communication. The smaller the system, the more communication has to happen between the nodes (relative to the amount of time spent on actual calculation). Thus, one expects a better speed up for larger systems. The performance of the

network is also crucial: generally, we expect the best speed up on very fast networks and shared memory systems (i.e. multi-CPU / multi-core architectures). We further expect the speed-up to become worse (in comparison to the ideal linear speed-up) with an increasing number of processes.

2.28 Restarting MPI runs

There is one situation that should be avoided when exploiting parallel computation. Usually, a simulation (involving for example a hysteresis loop), can be continued using the `--restart` switch. This is also true for MPI runs.

However, the number of CPUs used *must not change* between the initial and any subsequent runs. (The reason for this is that the `_dat.h5` file needs to store the mesh as it has been reordered for n CPUs. If we continue the run with another number of CPUs, the mesh data will not be correct anymore which will lead to errors when extracting the data from the `_dat.h5` file.)

Note also that there is currently no warning issued (in Nmag 0.1) if a user ventures into such a simulation.

2.29 More than one magnetic material, exchange coupled

To be written.

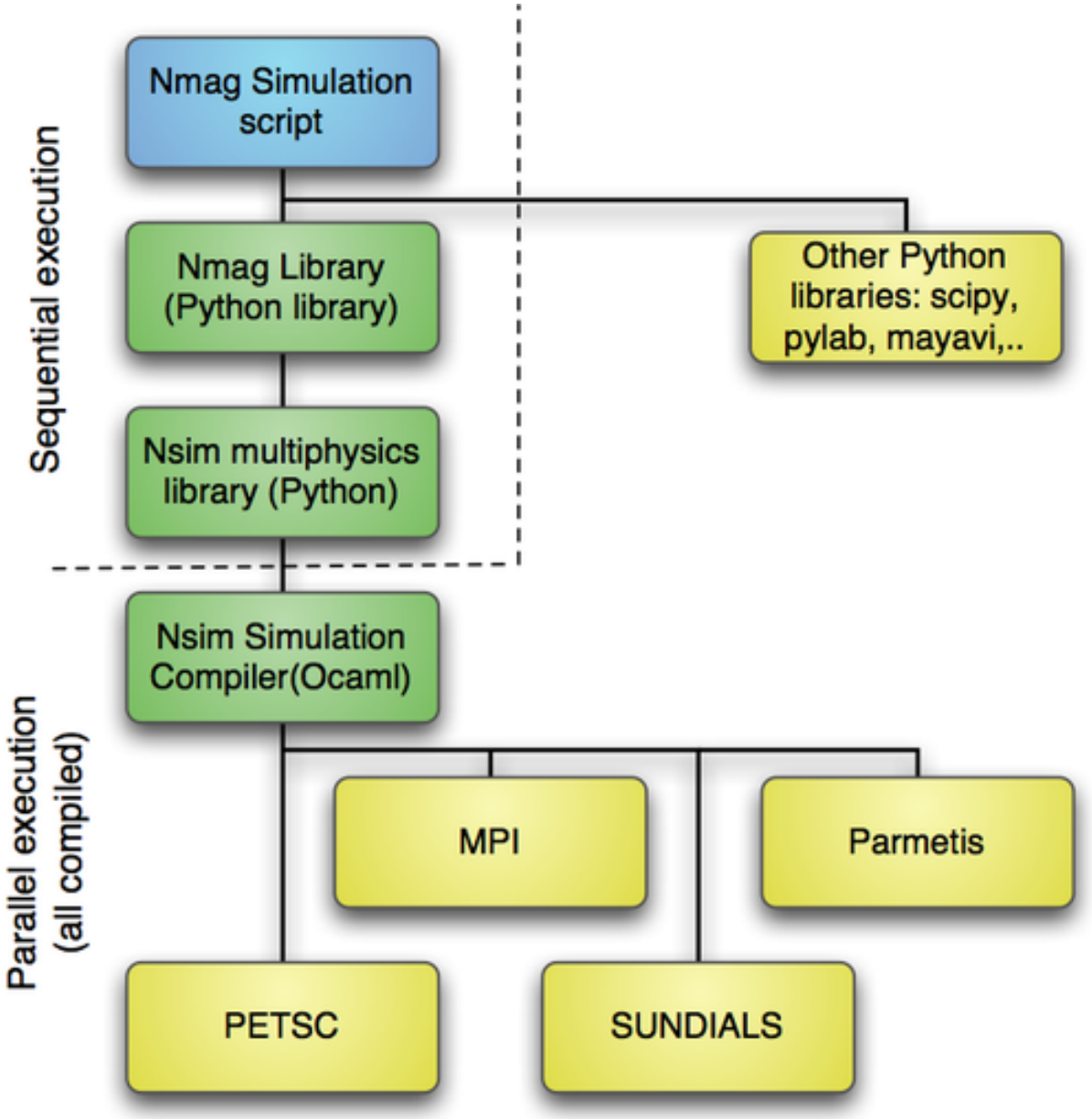
CHAPTER

THREE

BACKGROUND

In this section, we list some background information on the simulation package, some explanation of the philosophy behind it (which may explain some of the user interface choices that have been made) and explanation of some terms that are relevant.

3.1 Architecture overview



The Nmag environment that is described in this manual is shown as the blue box labelled `Nmag Simulation Script`. This is importing the `nmag` library – which is a Python library. This in turn, is built on the Nsim library Python module. The nsim Python module uses compiled code which is written in Objective Caml. At this level the execution can be parallel and this is also used to link together existing libraries (yellow boxes).

3.2 The nsim library

Nmag is the high-level user interface that provides micromagnetic capabilities to a general purpose finite element multi-physics field theory library called nsim. Therefore, many of the concepts used by Nmag are inherited from nsim.

This manual documents the high-level Nmag userinterface, it does not document nsim. Some of the internal details of nsim are explained in http://arxiv.org/abs/arXiv:0907.1587.

3.3 Fields and subfields

3.3.1 Field

The *Field* is the central entity within the *The nsim library*. It represents physical fields such as:

- magnetisation (usually a 3d vector field),
- the magnetic exchange field (usually a 3d vector field), or
- magnetic exchange energy (a scalar field).

A field may contain degrees of freedom of different type, which belong to different parts of a simulated object. For example, the magnetisation field may contain the effective magnetisation (density) for more than one type of magnetic atoms, which may make up different parts of the object studied. In order to deal with this, we introduce the concept of *Subfield*s: A Nmag/nsim field can be regarded as a collection of subfields. Most often, there only is one subfield in a field, but when it makes sense to group together multiple conceptually independent fields (such as the effective magnetisation of the iron atoms in a multilayer structure and that of some other magnetic metal also present in the structure), a field may contain more than one subfield: In particular, the magnetisation field `M` may contain subfields `M_Fe` and `M_Co`.

The question what subfields to group together is partly a question of design. For Nmag, the relevant choices have been made by the Nmag developers, so the user should not have to worry about this.

3.3.2 Subfield

Each field contains one or more *Subfield*s. For example, a simulation with two different types of magnetic material (for example Fe and Dy), has a field `m` for the normalised magnetisation and this would contain two subfields `m_Fe` and `m_Dy`.

(It is partly a question of philosophy whether different material magnetisations are treated as subfields in one field, or whether they are treated as two fields. For now, we have chosen to collect all the material magnetisations as different subfields in one field.)

Often, a field contains only one subfield and this may carry the same name as the field.

3.4 Fields and Subfields in Nmag

3.4.1 Example: one magnetic material

Assuming we have a simulation of one material with name PermAlloy (Py), we would have the following *Field*s and *Subfield*s:

Field	Subfield	Comment
m	m_Py	normalised magnetisation
M	M_Py	magnetisation
H_total	H_total_Py	total effective field
H_ext	H_ext	external (applied) field (only one)
E_ext	E_ext_Py	energy density of Py due to external field
H_anis	H_anis_Py	crystal anisotropy field
E_anis	E_anis_Py	crystal anisotropy energy density
H_exch	H_exch_Py	exchange field
E_exch	E_exch_Py	exchange energy
H_demag	H_demag	demagnetisation field (only one)
E_demag	E_demag_Py	demagnetisation field energy density for Py
phi	phi	scalar potential for H_demag
rho	rho	magnetic charge density (div M)
H_total	H_total_Py	total effective field

It is worth noting that the names of the fields are fixed whereas the subfield names are (often) material dependent and given by

- the name of the field and the material name (joined through '_') if there is one (material-specific) subfield for every magnetisation or
- the name of the field if there is only one subfield (such as the demagnetisation field or the applied external field)

This may seem a little bit confusing at first, but is easy to understand once one accepts the general rule that the material-dependent quantities - and only those - contain a material-related suffix. All atomic species experience the demagnetisation field in the same way, so this has to be `H_demag` (i.e. non-material-specific). On the other hand, anisotropy depends on the atomic species, so this is `H_anis_Py`, and therefore, the total effective field also has to be material-specific: `H_total_Py`. (All this becomes particularly relevant in systems where two types of magnetic atoms are embedded in the same crystal lattice.)

3.4.2 Example: two magnetic materials

This table from the *Example: two different magnetic materials* shows the fields and subfields when more than one material is involved:

Field	Subfield(s)	Comment
m	m_Py, m_Co	normalised magnetisation
M	M_Py, M_Co	magnetisation
H_total	H_total_Py, H_total_Co	total effective field
H_ext	H_ext	external (applied) field (only one)
E_ext	E_ext_Py, E_ext_Co	energy density of Py due to external field
H_anis	H_anis_Py, H_anis_Co	crystal anisotropy field
E_anis	E_anis_Py, E_anis_Co	crystal anisotropy energy density
H_exch	H_exch_Py, H_exch_Co	exchange field
E_exch	E_exch_Py, E_exch_Co	exchange energy
H_demag	H_demag	demagnetisation field (only one)
E_demag	E_demag_Py, E_demag_Co	demagnetisation field energy density
phi	phi	scalar potential for H_demag
rho	rho	magnetic charge density (div M)
H_total	H_total_Py, H_total_Co	total effective field

3.4.3 Obtaining and setting subfield data

Data contained in subfields can be written to files (using *save_data*), can be probed at particular points in space (*probe_subfield*, *probe_subfield_siv*), or can be obtained from all sites simultaneously (*get_subfield*). Some data can also be set (in particular the applied field `H_ext` using *set_H_ext* and all the subfields belonging to the field `m` using *set_m*).

3.4.4 Primary and secondary fields

There are two different types of fields in nmag: *primary* and *secondary* fields.

Primary fields are those that the user can set arbitrarily. Currently, these are the (normalised) magnetisation `m` and the external field `H_ext` (which can be modified with *set_m* and *set_H_ext*).

Secondary fields (which could also be called *dependent fields*) can not be set directly from the user but are computed from the primary fields.

3.5 Mesh

In finite element calculations, we need a mesh to define the geometry of the system. For development and debugging purposes, nsim includes some (at present undocumented) capabilities to generate toy meshes directly from geometry specifications, but for virtually all nsim applications, the user will have to use an external tool to generate a (tetrahedral) mesh file describing the geometry.

3.5.1 Node

Roughly speaking, a mesh is a tessellation of space where the support points are called *mesh nodes*. nmag uses an unstructured mesh (i.e. the cells filling up three-dimensional space are tetrahedra).

3.5.2 node id

Each node in the finite element mesh has an associated node id. This is an integer (starting from 0 for the first node).

This information is used when defining which node is connected to which (see *Finite element mesh generation* for more details), and when defining the *sites* at which the field degrees of freedom are calculated.

3.5.3 node position

The position (as a 3d vector) in space of a node.

3.6 Site

A *Mesh* has nodes, and each node is identified by its *node id*.

If we use *first order basis functions* in the finite element calculation, then a *site* is exactly the same as a *node*. In micromagnetism, we almost always use first order basis functions (because the requirement to resolve the exchange length forces us to have a very fine mesh, and usually the motivation of using higher order basis functions is to make the mesh coarser).

If we were to use *second* or *higher order base functions*, then we have more sites than nodes. In a second order basis function calculation, we identify sites by a tuple of *node id*.

3.7 SI object

We are using a special `SI` object to express physical entities (see also *SI*). Let us first clarify some terminology:

physical entity A pair (a,b) where a is a number (for example 10) and b is a product of powers of dimensions (for example m^1s^-1) which we need to express a physical quantity (in this example 10 m/s).

dimension SI dimensions: meters (m), seconds (s), Ampere (A), kilogram (kg), Kelvin (K), Mol (mol), candela (cd). These can be obtained using the *units* attribute of the *SI* object.

SI-value for a given physical entity (a,b) where a is the numerical value and b are the SI dimensions, this is just the numerical value a (and can be obtained with the *value* attribute of the *SI* object).

Simulation Units The dimensionless number that expressed an entity within the simulation core. This is irrelevant to the user, except in highly exotic situations.

There are several reasons for using SI objects:

- In the context of the micromagnetic simulations, the use of SI objects avoids ambiguity as the user has to specify the right dimensions and - where possible - the code will complain if these are unexpected units (such as in the definition of material parameters).
- The specification of units is more important when the micromagnetism is extended with other physical phenomena (moving towards multi-physics calculations) for which, in principle, the software cannot predict what units these will have.
- Some convenience in having a choice of how to specify, for example, magnetic fields (i.e. `A/m`, `T/mu0`, `Oe`). See also comments in *set_H_ext*.

3.7.1 Library of useful si constants

The `si` name space in nmag provides the following constants:

```
"""Some useful SI constants"""
import math
from lib import SI
kilogram = SI(1.0,["kg",1])  #: The kilogram
meter = SI(1.0,["m",1])      #: The meter
metre = meter                #  alternative spelling
Ampere = SI(1.0,["A",1])     #: The Ampere
Kelvin = SI(1.0,["K",1])     #: The Kelvin
second = SI(1.0,["s",1])     #: The second
candela = SI(1.0,["cd",1])   #: The candela
mol = SI(1.0,["mol",1])      #: The mol

#specific units for magnetism
Newton = kilogram*meter/second**2          #: Newton
mu0 = SI(4.0e-7*math.pi, "N/A^2")          #: vacuum permeability mu0
Tesla = kilogram/Ampere/second**2          #: Tesla
Gauss = 1e-4*kilogram/Ampere/second**2     #: Gauss
Oersted=Gauss/mu0                          #: Oersted
Oe=Oersted                                 #: Oersted
gamma0 = SI(-2.2137286285040001e5, "m/A s")#: gyromagnetic ratio gamma0

# units: degrees/nanosecond: Useful to specify the stopping_dm_dt
degrees_per_ns = SI(math.pi/180.0)/SI(1e-9, "s")

# other units and constants
Joule = SI("J")
bohr_magneton = 9.2740094980e-24*Joule/Tesla # Bohr magneton
positron_charge = SI(1.6021765314e-19, "C")
electron_charge =  -positron_charge
boltzmann_constant = SI(1.3806504e-23, "J/K")
plank_constant = 6.6260689633e-34*Joule*second
reduced_plank_constant = plank_constant/(2*math.pi)
```

To express the magnetisation in A/m equivalent to the polaration of 1 Tesla, we could thus use:

```
from nmag import si

myM = 1.5*si.Tesla/si.mu0
```

The command reference for *SI* provides some more details on the behaviour of SI objects.

3.8 Terms

3.8.1 Stage, Step, iteration, time, etc.

We use the same terminology for hysteresis loops as OOMMF (stage, step, iteration, time) and extend this slightly:

step A step is the smallest possible change of the fields. This corresponds (usually) to carrying out a time integration of the system over a small amount of time *dt*. Step is an integer starting from 0.

If we minimise energy (rather than computing the time development exactly), then a step may not necessarily refer to progressing the simulation through real time.

iteration Another term for Step (deprecated)

stage An integer to identify all the calculations carried out at one (constant) applied magnetic field (as in OOMMF).

time The time that has been simulated (typically of the order of pico- or nanoseconds).

id This is an integer (starting from 0) that uniquely identifies saved data. *I. e.* whenever data is saved, this number will increase by 1. It is available in the *h5 data file* and the *Data files (.ndt)* data files, and thus allows to match data in the ndt files with the corresponding (spatially resolved) field data in the h5 file.

stage_step The number of steps since we have started the current stage.

stage_time The amount of time that has been simulated since we started this stage.

real_time The amount of real time the simulation has been running (this is the [wall] execution time) and therefore typically of the order of minutes to days.

local_time A string (human readable) with the local time. Useful in data files to see when an entry was saved.

unix_time The number of (non-leap) seconds since 1.1.1970 - this is the same information as local_time but represented in a more computer friendly way for computing differences.

3.8.2 Some geek-talk deciphered

nmag uses some object orientation in the high-level user interface presented here. There are a few special terms used in object orientation that may not be familiar and of which we attempt to give a very brief description:

method A method is just a function that is associated to an object.

3.9 Solvers and tolerance settings

There are a number of linear algebra solvers and one solver for ordinary differential equations (ODEs) in nmag:

1. two solvers for the calculation of the demagnetisation field. Default values can be modified when creating the *Simulation* object (this user interface is not final – if you really feel you would like to change the defaults, please contact the *nmag team* so we can take your requirements into account in the next release).
2. one solver for the system of algebraic equations that results from the time integrator's implicit integration scheme.

   (We need to document the default settings and how to modify this.)
3. the ODE integrator.

   Setting of the tolerances for the ODE integrator can be done with *set_params*. An example of this is shown in section *Example: Timestepper tolerances*.

We expect that for most users, the tolerances of the ODE integrator are most important (see *Example: Timestepper tolerances*) as this greatly affects the performance of the simulation.

3.10 The equation of motion: the Landau-Lifshitz-Gilbert equation

The magnetisation evolution, as computed by the *advance_time* or the *hysteresis* methods of the `Simulation` class, is determined by the following equation of motion:

dM/dt = -llg_gamma_G * M x H + llg_damping * M x dM/dt,

which is the Landau-Lifshitz-Gilbert equation (we often use the abbreviation "LLG"), a vector equation, where `M`, `H` and `dM/dt` are three dimensional vectors and `x` represent the vector product. This equation is used to dermine the evolution of each component of the magnetisation. For example, if the system has two materials with name `m1` and `m2`, then the magnetisation has two components `M_m1` and `M_m2` and the equations:

dM_m1/dt = -llg_gamma_G_m1 * M_m1 x H_m1 + llg_damping_m1 * M_m1 x dM_m1/dt,

dM_m2/dt = -llg_gamma_G_m2 * M_m2 x H_m2 + llg_damping_m2 * M_m2 x dM_m2/dt,

determine the dynamics of `M_m1` and `M_m2`. Here `H_m1` and `H_m2` are the effective fields relative to the two components, while with `dM_m1/dt` and `dM_m2/dt` we denote the two time derivatives. The constant `llg_gamma_G_XX` in front of the precession term in the LLG equation is often called "gyromagnetic ratio", even if usually, in physics, the gyromagnetic ratio of a particle is the ratio between its magnetic dipole moment and its angular momentum (and has units `A s/kg`). It is then an improper nomenclature, but it occurs frequently in the literature. The `llg_damping_XX` constant is called damping constant. Notice that these two constants are specified on a per-material basis. This means that each material has its own pair of constants (`llg_gamma_G_m1`, `llg_damping_m1`) and (`llg_gamma_G_m2`, `llg_damping_m2`). The two constants are specified when the corresponding material is created using the *MagMaterial* class.

CHAPTER

FOUR

COMMAND REFERENCE

4.1 MagMaterial

Module: `nmag`

Object: `MagMaterial`

Class constructor information:

```
(self,
name,
Ms=SI(0.86e6, "A/m"),
llg_damping=SI(0.5),
llg_gamma_G=SI(2.210173e5, "m/A s"),
exchange_coupling=SI(1.3e-11, "J/m"),
anisotropy=None,
anisotropy_order=None,
do_precession=True)
```

Parameters

name [string] The name of the material. This will be used in the names of material dependent fields and subfields. Must be alphanumeric (i.e. contain only the characters 0-9_a-zA-Z) Examples: `'Py'`, `'Fe_1'`, `'Fe_2'`

Ms [SI Object] The saturation magnetisation of the material (in Ampere per meter).

Example (and default (PermAlloy) value): `SI(0.86e6,"A/m")`

llg_gamma_G [SI Object] The constant in front of the precession term in the LLG equation:

dM/dt = -llg_gamma_G * M x H + llg_damping * M x dM/dt

It is often called gyromagnetic ratio, even if usually, in physics, the gyromagnetic ratio of a particle is the ratio between its magnetic dipole moment and its angular momentum (and has units A*s/kg). It is then an improper nomenclature, but it occurs frequently in the literature.

Example (and default value): `SI(2.210173e5, "m/A s")`.

llg_damping [SI Object] The damping parameter (often called alpha). Optimum damping for 1.0, realistic values are of the order of 0.01. The default value (as in OOMMF) is 0.5.

Example (and default value): `SI(0.5,"")`

exchange_coupling [SI Object] The coupling strength for the exchange interaction in Joule per meter.

Example (and default value): `SI(1.3e-11, "J/m")`

anisotropy [PredefinedAnisotropy Object or function(vector) -> SI Object] Either a predefined anisotropy (such as returned by uniaxial_anisotropy or cubic_anisotropy), or a custom function (which must be polynomial in the components of `m`) `a(m)` that computes anisotropy energy density as a function of the (normalised) magnetisation direction `m`.

If you specify a custom anisotropy function, you also need to pass the order of the polynomial in the `anisotropy_order` parameter.

Default value is `None`, that is, no anisotropy term is used.

anisotropy_order [int] If a custom polynomial anisotropy function `a(m)` is specified, the order of the polynomial must be given in this parameter. This is not required for pre-defined uniaxial_anisotropy or cubic_anisotropy anisotropy functions.

Default value is `None`.

do_precession [True or False] Boolean that can switch off the precessional term in the LLG equation. This is useful to improve convergence speed when studying metastable configurations.

properties**: list of strings (default: ["magnetic","material"])** A list of additional properties this material will be associated with. Normally, users do not have to change this, but it is used internally when setting up discretized operators.

Example (and default value): `True`

4.1.1 uniaxial_anisotropy

Module: `nmag`

Object: `uniaxial_anisotropy`

Arguments: `(axis, K1, K2=0)`

Returns a predefined anisotropy modelling an uniaxial anisotropy energy density term:

```
E_anis = - K1 * <axis, m>^2 - K2 * <axis, m>^4
```

(where *m* is the (normalised) magnetization.)

Parameters

axis [vector (=list)] Easy axis (or hard axis, if K1 < 0; will be normalised).

K1 [SI Object] Second-order phenomenological anisotropy constant (as used in the equation above).

K2 [SI Object] Fourth-order phenomenological anisotropy constant (as used in the equation above).

Default value is `0`.

4.1.2 cubic_anisotropy

Module: `nmag`

Object: `cubic_anisotropy`

Arguments: `(axis1, axis2, K1, K2=0, K3=0)`

Returns a predefined anisotropy modelling a cubic anisotropy energy density term:

```
E_anis = K1 * (<axis1,m>^2 <axis2,m>^2 + <axis1,m>^2 <axis3,m>^2 + <axis2,m>^2 <axis3,m>^2)
       + K2 * (<axis1,m>^2 <axis2,m>^2 <axis3,m>^2)
       + K3 * (<axis1,m>^4 <axis2,m>^4 + <axis1,m>^4 <axis3,m>^4 + <axis2,m>^4 <axis3,m>^4)
```

(where *m* is the (normalised) magnetisation.)

Parameters

axis1 [vector (=list)] First cubic anisotropy axis (will be normalised).

axis2 [vector (=list)] Second cubic anisotropy axis (will be orthonormalised with regards to *axis1*).

K1 [SI Object] Fourth-order phenomenological anisotropy constant (as used in the equation above).

K2 [SI Object] Sixth-order phenomenological anisotropy constant (as used in the equation above).

Default value is `0`.

K3 [SI Object] Eigth-order phenomenological anisotropy constant (as used in the equation above).

Default value is `0`.

4.2 Simulation

Module: `nmag`

Object: `Simulation`

Class constructor information: `(self, name=None, phi_BEM=None, periodic_bc=None, do_demag=True, do_anisotropy_jacobian=False, temperature=None, thermal_delta_t=None, user_seed_T=0, timestepper_max_order=2, timestepper_krylov_max=300, ksp_tolerances={}, adjust_tolerances=False, use_pvode=True, lam_debugfile=None)`

Parameters

name [string] Name of the simulation object; this is used e.g. for prefixing filenames created by nmag.

Default value is the name of the current script (sans extension).

do_demag [bool] Pass `False` to disable the demagnetisation field.

do_anisotropy_jacobian [bool] Pass `True` to enable the inclusion of derivatives from the anisotropy into the Jacobian. (Complicated anisotropy terms may blow up memory requirements for the Jacobian.)

Default value is `True`.

temperature [SI Object] Simulated temperature (unless equal to None, stochastic thermal fluctuations will be enabled).

Currently not supported (since July 2008)

thermal_delta_t [SI Object] Time step to use when stochastic thermal fluctuations are enabled.

Currently not supported (since July 2008)

timestepper_max_order [int] Maximum order for the time integrator (we use the BDF method).

Default value is 2.

timestepper_krylov_max [int] Maximum dimension of the Krylov subspace to be used in the time integrator.

Default (recommended) value is 300.

ksp_tolerances**: dictionary** Keys to this dictionary are: DBC.rtol DBC.atol DBC.dtol DBC.maxits NBC.rtol NBC.atol NBC.dtol NBC.maxits

Values are the petsc KSP-solver tolerances for the Dirichlet and von Neumann Laplace solvers used internally to compute the magnetic scalar potential from magnetic charge density.

4.2.1 advance_time

Module: `nmag`

Object: `Simulation.advance_time`

Arguments: `(self, target_time, max_it=-1, exact_tstop=None)`

This method carries out the time integration of the Landau-Lifshitz and Gilbert equation.

Parameters

target_time [SI Object] The simulation will run until this time is reached. If the target_time is zero, this will simply update all fields.

max_it [integer] The maximum number of iterations (steps) to be carried out in this time integration call. If set to `-1`, then there is no limit.

exact_tstop [boolean] When exact_tstop is True, the time integration is advanced exactly up to the given target_time. When False, the time integration ends "close" to the target_time. The latter option can result in better performance, since the time integrator is free to choose time steps which are as wide as possible. When exact_tstop is not given, or is None, the default value for this option will be used. The default value can be set using the method set_params, which should hence be used to control the behaviour of the hysteresis and relax methods.

4.2.2 get_subfield

Module: `nmag`

Object: `Simulation.get_subfield`

Arguments: `(self, subfieldname, units=None)`

Given a subfieldname, this will return a numpy-array containing all the data (one element for each site).

Parameters

subfieldname [string] The name of the subfield, for example `m_Py` or `H_demag`.

units : SI object

Optional parameter. If it is provided, then the entity is expressed in these units. If it is not provided, then the correct SI dimensions for this subfield are looked up, and *SI-value*s are returned.

If you would like to see simulation units in the output, then use `SI(1)`.

In short: if you omit the second parameter, you will obtain SI values.

Returns *data* : numpy-array

4.2.3 get_subfield_positions

Module: `nmag`

Object: `Simulation.get_subfield_positions`

Arguments: `(self, subfieldname, pos_units=SI(1,['m',1.0]))`

This function provides the positions of the sites for data obtained with get_subfield.

Parameters

subfieldname [string] The name of the subfield, for example `m_Py` or `H_demag`.

pos_units [SI Object] Specifies the physical dimension in which positions are to be expressed. Default is `SI(1,'m')`, which means to return site positions in meters.

To obtain site positions in nanometers, use `SI(1e-9,'m')`.

Returns

pos [numpy-array] Array containing a position (i.e. 3 floating point numbers) for every site.

4.2.4 get_subfield_sites

Module: `nmag`

Object: `Simulation.get_subfield_sites`

Arguments: `(self, subfieldname)`

This function provides the node indices of the sites for data obtained with get_subfield.

Parameters

subfieldname [string] The name of the subfield, for example `m_Py` or `H_demag`.

Returns

data [numpy-array] Array containing a list of integers for every site. The integers within each list are node indices of the mesh. There will be only one integer per site in first order basis function calculations (which is the usual case in micromagnetics)

4.2.5 get_subfield_average

Module: `nmag`

Object: `Simulation.get_subfield_average`

Arguments: `(self, field_name, subfield_name=None)`

the average of the subfield `subfield_name` of the field `fieldname` as an SI object (or a list of [list of [list of [...]]] SI objects in the field is a vector, 2nd rank tensor etc.

Parameters

field_name [string] name of the field

subfield_name [string] name of the subfield

See also get_subfield_average_siv.

4.2.6 get_subfield_average_siv

Module: `nmag`

Object: `Simulation.get_subfield_average_siv`

Arguments: `(self, field_name, subfield_name=None)`

the average of the subfield `subfield_name` of the field `fieldname` as a single floating point number (or a list if it is a vector, or a list of list for matrices etc).

The number is expressed in SI units (hence the suffix _siv which stands for si value).

Parameters

field_name [string] name of the field

subfield_name [string] name of the subfield

Example:

```
ave_M = sim.get_subfield_average_siv("M","Py")
```

will obtain the average magnetisation of the subfield M_Py of field M, for example `ave_M = [100000.00,0.,0.]`

4.2.7 probe_subfield

Module: `nmag`

Object: `Simulation.probe_subfield`

Arguments: `(self, subfieldname, pos, unit=None)`

ven subfield name and position (SI object), return data (as SI object).

Note that `get_subfield_siv` has the same functionality but takes a list of floats for the position (instead of an SI object) and returns (a list of) float(s) which is just the *SI-value* of that physical entity.

If the subfield is not defined at that part of space, `None` is returned.

If the subfield does generally not exist, then a `KeyError` exception is thrown.

Parameters

subfieldname [string] The name of the subfield

pos [SI object] The position for which the data should be returned

unit [SI object] If you request the value for a subfield of a field that is part of nmag (i.e. fields M, m, H_demag, etc), then you do not need to provide this object.

If you request data of any other (multi-physics) fields, then this function needs to know the SI dimensions of that field (for the correct conversion from simulation units to SI units).

If incorrect dimensions are provided, the returned data is likely to be wrongly scaled.

Returns

data [[list [of list[of ...]]] SI objects] The returned object is an SI object for scalar subfields, a list of SI objects for vector fields, a list of list of SI objects for (rank 2) tensor fields, etc.

4.2.8 probe_subfield_siv

Module: `nmag`

Object: `Simulation.probe_subfield_siv`

Arguments: `(self, subfieldname, pos, unit=None)`

The same behaviour as `get_subfield` but the `pos` and return data are *SI-value*s (not SI objects).

If the subfield is not defined at that part of space, `None` is returned.

If the subfield does generally not exist, then a `KeyError` exception is thrown.

The input (position) and returned data is expressed in SI units but of type float.

Parameters

subfieldname [string] The name of the subfield

pos [list of floats] The position for which the data should be returned (in meters)

unit [SI object] If you request the value for a subfield of a field that is part of nmag (i.e. fields M, m, H_demag, etc), then you do not need to provide this object.

If you request data of any other (multi-physics) fields, then this function needs to know the SI dimensions of that field (for the correct conversion from simulation units to SI units).

If incorrect dimensions are provided, the returned data is likely to be wrongly scaled.

Returns

data [[list [of list[of ...]]] float] The returned object is a float for scalar subfields, a list of floats for vector fields, a list of list of floats for (rank 2) tensor fields, etc.

4.2.9 probe_H_demag_siv

Module: `nmag`

Object: `Simulation.probe_H_demag_siv`

Arguments: `(self, pos, pos_unit=SI(1,['m',1.0]), epsilon=1e-07)`

ME: this function returns a wrong value for points outside the mesh. For a sphere uniformly magnetised along +x, the x component of the demag field outside should be positive, while it is negative.

Compute the demag field at given position. Works inside and outside of magnetic materials. Note that most fields can only be probed where they are defined. This function computes the demag field at the given position on the fly, based on the boundary element method.

Note that for large distances away from the magnetic material, we expect this not to be very accurate. Furthermore, there is an awkward technical problem whenever the probe point lies in-plane with any of the surface triangles. These awkward limitations are strongly linked to the method used to compute the scalar potential internally and are intrinsically difficult to avoid. They will go away in the future when potential computations will be performed with Hlib.

Also, this function should (at present) not be used to probe the demag field for periodic structures.

Parameters

pos [list of floats] The SI numbers described the position in meters. A command like `probe_H_demag_siv([0,0,1e-9])` would thus probe the demag field one nanometer away (in z-direction) from the origin.

pos_unit [SI object] Optional argument that defaults to SI("m"). The full SI position is computed as pos*pos_unit. The above example could therefore be written as `probe_H_demag_siv([0,0,1],pos_unit=SI(1e-9,"m"))`.

epsilon [float] This parameter is used internally to compute the demag field via central differences from the magnetic potential if the observer point is in the exterior ("vacuum") region. It is the distance between the two points at which each of the field components is being computed (because the field is the negative gradient of the potential). The default value of 1e-7 should be sensible if normal simulation units are used (i.e. the mesh was provided with coordinates in the range 1-1000). Typically, this parameter should be ignored. Note that this number is measured in simulation units.

Returns A list of floats containing the demag field in SI units (i.e. A/m) at the specified position.

4.2.10 hysteresis

Module: `nmag`

Object: `Simulation.hysteresis`

Arguments:

```
(
self,
H_ext_list,
save=[('averages', 'fields', at('stage_end'))],
do=[],
convergence_check=every('step', 5)
)
```

This method executes a simulation where the applied field is set in sequence to the values specified in `H_ext_list`. The time integration proceeds with the same applied field until convergence is reached. At this point the field is changed to the next one in `H_ext_list` and the method `reinitialise()` is called to proceed with the simulation. The user can specify when to save data using the optional argument `save`.

This allows to carry out hysteresis loop computations and write the results to disk.

Technically we say that this function performs a multi-stage simulation. In our terminology, a stage is a part of the simulation where the field does not change. Therefore, every value for the applied field specified in `H_ext_list` corresponds to a different stage. Stages are numbered starting from 1, which corresponds to `H_ext_list[0]`. In general during stage number `i` the applied field is `H_ext_list[i-1]`.

Parameters

H_ext_list [list of values for the applied field] It is something like `[H1, H2, H3, ...]`, where `Hi` is the triple of components of the applied field, i.e. SI objects having units of "A/m";

save [list of pairs `(thing_to_save, when)`] `thing_to_save` is either a string or a function provided by the user and `when` is an instance of the class `When`, i.e. an object which contains the specification of when "the thing" has to be saved.

Possible string values for `thing_to_save` are:

- `"averages"`: to save the averages of all the fields together with other information (such as the stage number, the time reached, etc.). This is done calling the method `save_data()`. Refer to its documentation for further details;
- `"fields"`: to save all the fields. The method `save_data(fields='all')` is called for this purpose;
- `"restart"`: to save the current magnetisation configuration and all the information needed to restart the simulation.

do [list of pairs `(thing_to_do, when)`] is very similar to the `save` argument, but is usually used for other purposes. `thing_to_do` is either a string or a function provided by the user and `when` is an instance of the class `When`.

Possible string values for `thing_to_do` are:

- `"next_stage"`: induces the hysteresis method to advance to the next stage;
- `"exit"`: induces the hysteresis method to exit, even if the hysteresis computation has not still reached its end.

The user can provide his own function to save data. For example, the following three lines:

```
def my_fun(sim):
  sim.save_data()
sim.hysteresis(..., save=[(my_fun, every('step', 10))])
```

are equivalent to:

```
sim.hysteresis(..., save=[('averages', every('step', 10))])
```

To specify when something has to be saved the module `when` is used. The functions `at` and `every`, provided by this module, can refer to the following time variables:

- `step`: the step number from the beginning of the simulation;
- `stage_step`: the step number from the beginning of the current stage;
- `time`: the simulation time passed from the beginning of the simulation (measured in SI objects);
- `stage_time`: the simulation time passed from the beginning of the current stage;
- `stage`: the number of the current stage;
- `convergence`: a boolean value which is `True` if the convergence criterion is satisfied. Use in this way `at('convergence')`

Remember that you can combine time specifications using the operator | (or) and & (and):

```
every('stage', 2) & at('convergence') --> only at convergence
                                          of odd stages
every('step', 10) | at('convergence') --> at convergence
                                          and every 10 steps.
```

Some usage examples:

```
# Save fields (which implicitly will save the averages as well)
# when the magnetisation stops changing for each applied field
# (i.e. save at convergence):
sim.hysteresis(..., save=[('fields', at('convergence'))])

# Averages will be saved every 10 steps, fields (and
# implicitely averages) will be saved at convergence.
sim.hysteresis(..., save=[('averages', every('step', 10)),
                          ('fields', at('convergence'))])

# Each stage will not last more than 10 ps, even
# if the magnetisation is not relaxed yet.
sim.hysteresis(..., do=[('next_stage', at('stage_time', SI(1e-11, "s")))])

# Exit hysteresis loop simulation if the total number of
# steps exceeds 1e6, save fields every 100 steps and at
# convergence before that:
sim.hysteresis(..., save=[('fields', every('step', 100) |
                           at('convergence'))],
                    do =[('exit', at('step', 1e6))])

# Save averages every 0.1 ns (useful for fourier transform)
# leave after 20 ns (using the related relax_ command)
sim.relax(save=[('averages', every('time', SI(1e-10, 's')))],
          do  =[('exit', at('time', SI(20e-9, 's')))])

# Save averages every nanosecond, and fields every 100 ns.
sim.relax(save=[('averages',every('time', SI(1e-9, 's'))),
                ('fields',  every('time', SI(100e-9,'s')))])

# Save averages every nanosecond, and fields every 100 ns,
# save restart file every 1000 steps
sim.relax(save=[('averages',every('time', SI(1e-9, 's'))),
                ('fields',  every('time', SI(100e-9, 's'))),
                ('restart', every('step', 1000))])
```

If `save` is not given, averages and fields will be saved whenever the stage ends (this is the default behaviour).

4.2.11 load_mesh

Module: `nmag`

Object: `Simulation.load_mesh`

Arguments: `(self, filename, region_names_and_mag_mats, unit_length, do_reorder=False, manual_distribution=None)`

Parameters

filename [string] The file that contains the mesh in nmesh format (ascii or hdf5)

region_names_and_mag_mats [list of 2-tuples] A list of 2-tuples containing the region names and the magnetic materials associated to each region. For example, having two spheres (called `region_A` and `region B`) with materials A and B in the mesh, the argument would be [("region_A", A),("region_B",B)] where A and B must have been defined previously as `nmag.MagMaterial`.

Having two Materials X and Y both defined in region A (as in a magnetic two-component alloy), we would use [("region_A",[X,Y])].

unit_length [SI object] The SI object defines what a length of 1.0 in the mesh file corresponds to in reality. If the length 1.0 in the mesh corresponds to a nanometer, then this SI object would be given as SI(1e-9,"m")

do_reorder [bool] If set to True, metis will be called to reorder the mesh (aiming to bring together node ids that correspond to node locations that are spatially close to each other). If this doesn't make sense to you, you should probably leave the default (which is `False`).

Generally, we recommend to order a mesh using `nmeshpp --reordernodes mesh.nmesh orderedmesh.nmesh`, and *not to use* this reodering option here, if you think you need to order it.

If you know nmag really well (you are probably a member of the core team) then read on.

The use of `do_reorder` *can* make sense if either your mesh is not ordered already, or you provide a `manual_distribution` of nodes.

The use of `do_reorder` makes no sense, if you run on more than one CPU and leave the distribution of the nodes to nmag (i.e. you use the default `manual_distribution==None`).

manual_distribution [list of integers] This list (if provided) describes how many nodes are to be put onto which CPU under MPI-parallelized execution. If this is `None` (i.e. the default), then the distribution is done automatically (through metis). This parameter should generally not be used (unless you really know what you are doing).

Returns *mesh* : mesh object

4.2.12 load_m_from_h5file

Module: `nmag`

Object: `Simulation.load_m_from_h5file`

Arguments: `(self, filename, **kwargs)`

magnetisation stored in `filename` to set the magnetisation of the simulation. (If more than one magnetisation configurations have been saved in the file, it will load the first one.)

This can be used to retrieve the magnetisation saved in a restart file, and to set the current magnetisation of the simulation object to this magnetisation.

This restart file could have been written explicitely (using the save_restart_file method), or implicitly by providing a 'restart' action to the hysteresis/relax commands.

To simply continue a hysteresis/relax simulation using the `--restart` option, there is no need to use this function. It should only be used if lower-level manipulation is required (see for example *Current-driven motion of a vortex in a thin film*).

4.2.13 save_restart_file

Module: `nmag`

Object: `Simulation.save_restart_file`

Arguments: `(self, filename=None, fieldnames=['m'], all=False)`

rent magnetic configuration into file that can be used for restarting.

This function saves the current magnetisation, the time and all what is needed to restart the simulation exactly from the point it was invoked.

Parameters *filename* : string

The file into which the restart file is written. Defaults RUNID_restart.h5.

***fieldnames*: list** The fieldnames to be saved. Defaults to ['m']

***all*:bool** If true, then all fields will be saved.

This function is used by the hysteresis and relax commands to save a magnetic configuration from which a run can be continued (using –restart).

Example:

A common usecase for this function maybe to write the magnetic configuration that comes from a relaxation process to a file. And to load that configuration as the initial configuration for a subsequent (series of) simulation(s).

In this case, one may want to provide the filename explicitely. For example:

```
sim.save_restart_file(filename="relaxed_configuration.h5")
```

One can then use the load_m_from_h5file, to read this file `relaxed_configuration.h5` and to use it to set the magnetisation up in the subsequent simulation.

4.2.14 relax

Module: `nmag`

Object: `Simulation.relax`

Arguments: `(self, H_applied=None, save=[('averages', 'fields', at(stage_end, True))], do=[], convergence_check=every(5, 'step'))`

This method carries out the time integration of the LLG until the system reaches a (metastable) equilibrium. Internally, this uses the hysteresis loop command.

Parameters

H_applied [list of SI objects] For a 3-d simulation, the SI-objects Hx, Hy and Hz would be specified as `[Hx,Hy,Hz]`.

Default value is `None`, resulting in the currently applied external field `H_ext` being used.

save [Schedule object] Allows to define what data to save at what events. See documentation on the hysteresis method and on the `Schedule` object.

convergence_check : every object The default value (`every('step', 5)` specifies that we ask the time integrator to carry out 5 steps before we check for convergence. If in doubt, ignore this feature.

4.2.15 save_data

Module: `nmag`

Object: `Simulation.save_data`

Arguments: `(self, fields=None, avoid_same_step=False)`

Save the *averages* of all defined (subfields) into a ascii data file. The filename is composed of the simulation name and the extension `_dat.ndt`. The extension `ndt` stands for Nmag Data Table (analog to OOMMFs `.odt` extension for this kind of data file.

If `fields` is provided, then it will also save the spatially resolved fields to a file with extensions `_dat.h5`.

Parameters *fields* : None, 'all' or list of fieldnames

If None, then only spatially averaged data is saved into `*ndt` and `*h5` files.

If `all` (i.e. the string containing 'all'), then all fields are saved.

If a list of fieldnames is given, then only the selected fieldnames will be saved (i.e. ['m','H_demag']).

avoid_same_step : bool

If `True`, then the data will only be saved if the current `clock['step']` counter is different from the step counter of the last saved configuration. If `False`, then the data will be saved in any case. Default is `'False'`. This is internally used by the hysteresis command (which uses `avoid_same_step == True`) to avoid saving the same data twice.

The only situation where the step counter may not have changed from the last saved configuration is if the user is modifying the magnetisation or external field manually (otherwise the call of the time integrator to advance or relax the system will automatically increase the step counter).

4.2.16 set_m

Module: `nmag`

Object: `Simulation.set_m`

Arguments: `(self, values, subfieldname=None)`

Parameters

values [vector (=list), function or numpy array.] The values to be set. See more detailed explanation below.

This method sets the (normalised) magnetisation (i.e. the `m` field) to a particular value (or pattern).

It can be used in three different ways:

1. Providing a constant vector

   If given a vector, this function sets the `m` field to uniformly point in the given direction, everywhere.

   For example, to have the magnetisation point in +x-direction, we could call the function like this:

   ```
   sim.set_m([1,0,0])
   ```

   To point in a 45 degree direction between the x- and y-axis, we could use:

   ```
   sim.set_m([1,1,0])
   ```

   (The magnetisation will automatically be normalised.)

2. Providing a function

   If the magnetisation is meant to vary spatially, then a function can be given to the `set_m` method as in this example:

```
def my_magnetisation((x,y,z)):
    # get access to pi, cos and sin
    import math

    # change angle of Mx and My by 10 degree when x varies by 1nm
    angle = (x*1e9)*10./360*2*math.pi
    Mx = math.cos(angle)
    My = math.sin(angle)
    Mz = 0

    #return magnetisation vector for position (x,y,z)
    return (Mx,My,Mz)

sim.set_m(my_magnetisation)
```

   The function `my_magnetisation` returns the magnetisation vector corresponding to the given 3d position in space.

   This position `(x,y,z)` as given to the function is expressed in meters.

3. Providing a numpy array.

   If a numpy array is provided to set the values of the subfield, then the shape of this array has to match the shape of the subfield data. For example, if the subfield is the magnetisation of material X, and this material is defined on n mesh sites, then the array needs to have n entries. Each of those has to be a 3-component array, as the magnetisation vector has three components.

   Note: the Simulation.get_subfield() function can be used to obtain exactly such a numpy array for the relevant subfield.

   To read such a numpy array from a file, you can use the get_subfield_from_h5file function. However, you have to be sure that the node order in the mesh (that is stored in the _dat.h5 file) is the same as the mesh you are currently using in your simulation. This should certainly be the case if (i) both runs [i.e. the saved and the current] are based on the same mesh, and (ii) you only us one CPU [as using more than one results in repartitioning and reordering of the mesh]. We aim to not allow setting 'wrong' data here in the future, but currently such checking is not implemented. (fangohr 31/05/2008)

4.2.17 set_H_ext

Module: `nmag`

Object: `Simulation.set_H_ext`

Arguments: `(self, values, unit=None)`

Parameters

values [vector (=list), function or numpy array.] See set_m for an explanation of possible `values`.

unit : SI Object

An SI Object that is used as a multiplier for the `values`. This unit has to be physically compatible with Ampere per meter.

To set an applied field that is homogenous and points in +x-direction, one can use:

```
sim.set_H_ext([1e6, 0, 0], SI("A/m"))
```

which is equivalent to::

```
sim.set_H_ext([1, 0, 0], SI(1e6, "A/m"))
```

However, we could also define the field in Oersted:

```
from nmag.si import Oe
sim.set_H_ext([100, 0, 0],Oe)
```

or in Tesla/mu0:

```
from nmag.si import Tesla, mu0
sim.set_H_ext([1, 0, 0], Tesla/mu0)
```

4.2.18 set_pinning

Module: `nmag`

Object: `Simulation.set_pinning`

Arguments: `(self, values)`

Parameters *values* : vector (=list), function or numpy array.

This method sets the scalar pinning field which defines a local scale factor for `dm/dt`.

Default value is `1.0`, use `0.0` to force `dm/dt` to zero, that is, to "pin" (fix) magnetisation at a certain position.

Semantics of the *values* parameter match set_m.

4.2.19 set_params

Module: `nmag`

Object: `Simulation.set_params`

Arguments: `(self, stopping_dm_dt=None, ts_rel_tol=None, ts_abs_tol=None, exact_tstop=None)`

Set the parameters which control the accuracy and performance of the simulation.

Parameters

ts_rel_tol [float] the relative error tolerance (default is 1e-6) for the timestepper

ts_abs_tol [float] the absolute error tolerance (default is 1e-6) for the timestepper

stopping_dm_dt [SI object] the value used in the hysteresis and relax functions to decide whether convergence has been reached. If the largest value for dm/dt drops below `stopping_dm_dt`, then convergence has been reached.

The default value for `stopping_dm_dt` this is that the magnetisation changes less than one degree per nanosecond, i.e. `stopping_dm_dt = SI(17453292.519943293,['s',-1])`.

exact_tstop [bool] the value of exact_tstop which is used by the advance_time method when the optional argument is not given. This is also the value used by the relax and hysteresis methods. See the documentation of advance_time for further details.

Note that this command has to be issued *after* having created an m-field with the set_m command.

4.3 get_subfield_from_h5file

Module: `nmag`

Object: `get_subfield_from_h5file`

Arguments: `(*args, **nargs)`

Retrieve data from h5 file. Data are returned as an array of floating point number (in SI units).

This function should be used with care, as the order of the entries in the returned array depends on the partitioning of the mesh used when saving the data.

Analog to get_subfield (which returns subfield data for a subfield of a simulation object), but will retrieve data from saved `_dat.h5` file.

Note that the entries of the returned array are ordered accordingly to the mesh used in this simulation object.

Parameters

filename [string] The full name of the `_dat.h5` data file.

subfieldname [string] The name of the subfield to be retrieved.

id [integer] The `id` of the configuration to return (defaults to 0)

row : integer

If the `id` is not specified, the `row` can be used to address the data row with index `row`.

For example, the magnetisation may have been saved at some point during the simulation into a file (for example using the *Restart example* functionality, or using the save_data method for the first time to save the m-field (i.e. `sim.save_data(fields=['m'])` into a new file).

We can use `row=0` to read the first magnetisation configuration that has been written into this file (and `row=1` to access the second etc).

Returns numpy array

4.4 get_subfield_positions_from_h5file

Module: `nmag`

Object: `get_subfield_positions_from_h5file`

Arguments: `(filename, subfieldname)`

Analogous to get_subfield_positions (which returns the positions of nodes for a subfield of a simulation object), but will retrieve data from saved `_dat.h5` file.

Parameters

filename [string] The full name of the `_dat.h5` data file.

subfieldname [string] The name of the subfield to be retrieved.

Returns

numpy array The positions are returned as *si-value*s.

4.5 get_subfield_sites_from_h5file

Module: `nmag`

Object: `get_subfield_sites_from_h5file`

Arguments: `(filename, subfieldname)`

Analogous to get_subfield_sites (which returns the site ids of nodes for a subfield of a simulation object), but will retrieve data from saved `_dat.h5` file.

Parameters

filename [string] The full name of the `_dat.h5` data file.

subfieldname [string] The name of the subfield to be retrieved.

Returns

numpy array The ids are returned as *si-value*s.

4.6 HMatrixSetup

Module: `nmag`

Object: `HMatrixSetup`

Class constructor information: `(self, algorithm='HCA2', **kwargs)`

ass collecting the parameters needed in order to set up an HMatrix with HLib within Nmag.

The optional argument `algorithm` is by default set to "HCA2". At present no other values are supported. The user can then specify a number of parameters in order to fine-tune the setup of the HMatrix. `**kwargs` stands for one or more of the following parameters (see Hlib documentation for detailed descriptions of the parameers):

Parameters of HCA II

eps_aca [float] A heuristic parameter which influences the accuracy of HCA II. By default this parameter is set to 1e-7

poly_order [int] A second parameter which influences the accuracy of the HCA II. Its default setting is 4.

Parameter for recompression algorithm

eps [float] This parameter determines the accuracy of the recompression algorithm, which optimises a given hierarchical matrix. The default value is 0.001.

Parameters influencing the tree structure

cluster_strategy [string] algorithm to be used for creating the cluster tree. Available choices are 'regular' (cluster constructed splitting the bounding box of the surface in two smaller bounding boxes with half the size along x, then y, z, x, and so on), 'geometric' (similar to 'regular' but the splitting is done along longer dimension of the bounding box) 'regular_box' (behaves similarly to 'regular'), 'cardinality' (the bounding box is split into two bounding boxes containing the same number of points, cyclically along each dimension), 'pca' (clustering based on principal directions), 'default' (uses the default). The default clustering strategy is 'regular'.

eta [float] eta is a parameter which influences the so called admissibility criterion. As explained above, a subblock of the boundary element matrix basically describes the dipole potential at a cluster of surface nodes A generated by a different cluster B. The subblock can only be approximated when both cluster are spatially well separated. To have an objective measure of what 'well separated' means, an admissibility criterion has been introduced. The smaller the parameter eta is chosen, the more restrictive is the admissibility criterion. The default value is 2.0.

nmin [int] In order to be able to adjust the coarseness of the tree structure (a too fine tree structure would result in a higher amount of memory required for the storage of the tree itself), a parameter nmin has been introduced. It is the minimal number of lines or rows a submatrix within a leave can have, and is by default set to 30.

Parameter for the numerical quadrature

quadorder [int] The order of the Gaussian quadrature used to compute matrix entries for the low-rank matrix blocks. For the matrix blocks, which are not approximated, an analytical expression instead of numerical integration is used. By default, quadorder is set to 3.

4.7 SI

Module: `nmag`

Object: `SI`

Class constructor information: `(self, value, dimensions=[])`

Physical quantity in the SI units system.

This class allows to associate SI-dimensions (such as meter, kilogram, Ampere, seconds, candela and mol) with a floating point number.

The resulting object supports addition, subtraction, (which fails if the dimensions of the objects in a sum or difference disagree), multiplication and division.

There are different ways to create objects:

1. The most fundamental approach is to provide a value and a list of pairs where each pair is a character identifying the SI base unit and an integer that provides its power.

   Examples:

   (a) `v = SI(10,['m',1,'s',-1])` is the code to create an SI object v that represents 10 m/s.

   (b) `B = SI(0.6,['kg',1,'s',-2,'A',-1])` is the code to create an SI object T that represents 0.6 kg/(s^2 A) (i.e. 0.6 Tesla)

2. A more convenient way is to first define all the base units like this (these are already defined in the `si` submodule, so instead of the following lines below, we could also just write: `from si import meter,second,Ampere`):

   ```
   meter = SI(1,'m') # alternative spelling: metre
   second = SI(1,'s')
   Ampere = SI(1,'A')
   ```

   and then to use these SI objects to create more complex expressions:

   ```
   v = 10*meter/second
   B = 0.6*kilogram/second**2/Ampere
   ```

   Of course, short hand notations can be defined such as:

   ```
   T = kilogram/second**2/Ampere
   B = 0.6*Tesla
   ```

3. Finally, there is another convenient way:

   Instead of a SI dimension vector as in (1), it is possible to pass a string specifying dimensions. Examples are:

   "A/m", "V/m", "J/m^3", "m^2 s^(-2)", "m^-3 s^-1" etc.

   The dimensions parser will understand (in addition to m, kg, s, A, K, mol, cd): J, N, W, T, V, C, Ohm, H

A very basic demonstration of the SI object in use:

```
>>> a = SI(1,'m')
>>> b = SI(1e-3,'m')
>>> print a+b
<SI: 1.001  m >
```

```
>>> print a*b
<SI: 0.001  m^2 >
>>> print a/b
>>> <SI: 1000  >            #Note that this is dimensionless
                            #because we divided meters by meters
```

4.7.1 value

Module: `nmag`

Object: `SI.value`

Property information `None`

Read-only attribute to obtain (dimensionless) value of Physical Object

Returns

value [float] The numerical value.

Example:

```
>>> from nmag import SI
>>> H = SI(10, 'A/m')
>>> print H.value
10.0
>>> print H
<SI: 10  A / m >
```

4.7.2 units

Module: `nmag`

Object: `SI.units`

Property information `None`

Read-only attribute to obtain units of Physical Object (returned as list of pairs of dimension name and power)

4.7.3 in_units_of

Module: `nmag`

Object: `SI.in_units_of`

Arguments: `(self, unit_quantity)`

The object will be expressed in multiplies of 'unit_quantity'. This is useful to convert from one measurement convention (such as m/s) to another one (such as km/h). The return value is just a float.

The units of 'unit_quantity' have to be compatible with the units of the object itself (otherwise an exception is raised).

A simple example:

```
>>> d = SI(10,'m')
>>> inch = SI(2.54e-2,'m')
>>> d.in_units_of(inch)
393.70078740157478
```

Another example:

```
>>> m = SI(1,'m')
>>> s = SI(1,'s')
>>> velocity=2*m/s
>>> print velocity
<SI: 2  m / s >
>>> km = 1000*m
>>> h = 3600*s
>>> print velocity.in_units_of(km/h)
8.2
```

Parameters

unit_quantity [SI Object] The SI object itself (i.e. `self`) will be expressed in multiplies of this `unit_quantity`. ‘

Returns

float This is the number that, multiplied by the `unit_quantitity` will provide the SI quantity of the object itself.

4.7.4 is_compatible_with

Module: `nmag`

Object: `SI.is_compatible_with`

Arguments: `(self, physical_quantity)`

Returns True when the given physical quantity is compatible with the object itself.

Example:

```
>>> from nsim.si_units import SI
>>> m_per_sec = SI(1,'m')/SI(1,'s')
>>> km_per_hour = SI(1000,'m')/SI(3600,'s')
>>> Newton = SI(1,'kg')*SI(1,'m')/SI(1,'s')**2
>>> m_per_sec.is_compatible_with(Newton)
False
>>> m_per_sec.is_compatible_with(km_per_hour)
True
```

4.8 ipython

Module: `nmag`

Object: `ipython`

Arguments: `(globals=None, locals=None)`

Interactive python prompt (see *Example: IPython*).

4.9 Command line options

Nmag supports a number of command line options to configure its behaviour.

Suppose the simulation script is called `X.py`, then these OPTIONS can be specified like this:

```
nsim X.py OPTIONS
```

`X.py` needs to contain at least the line `import nmag` as this will process the command line options.

The available options are:

–clean to override any existing `_dat.h5` and `_dat.ndt` files. If this option is not provided and the data files exist already, then nmag will interrupt the execution without having modified the data files on the disk.

Example:

```
nsim X.py --clean
```

–loglevel this switch determines the amount of information that is being send to stdout (usually the screen) and also to the file `X_log.log`.

The available levels are in increasing order of detail:

error print no messages apart from errors

warning print warnings

info print a moderate amount of information (default)

info2 print slightly more information

debug print a lot of information (typically for developer and debugging use)

Example:

```
nsim X.py --loglevel info2
```

or:

```
nsim X.py --loglevel debug
```

–slavelog Log message from slave nodes (when running under MPI) are usually supressed. This switch activates them. Printing these messages will reduce the MPI performance somewhat as the messages are printed to stdout on each slave, and then have to be transferred through the network to the master process.

Note that any log-messages from the nodes will only go to stdout (whereas log messages from the master will also go into the log file, see *File names for log files*.)

Messages from slave nodes are preceeded by `S0X` where X is the rank of the node. I.e. log messages from slave node with rank 2, would start with `S02`.

Example:

```
nsim X.py --slavelog
```

–restart If a calculation of a hysteresis loop is interrupted (power cut, computer crash, exceeding allocated run time on cluster, etc), then the calculation can be carried out starting from the moment when the last restart file was saved (see *Restart example*).

This continuation is activated with the `--restart` switch.

Example:

```
nsim X.py --restart
```

Note that this functionality is only available for the hysteresis loop.

The command line options can be combined, for example:

```
nsim X.py --clean --loglevel debug
```

There are a few other switches (mostly for debugging) which can be seen using:

```
nsim X.py --help
```

CHAPTER

FIVE

FINITE ELEMENT MESH GENERATION

Finite element mesh generation is a difficult business, and one needs to get used to using at least one mesh generating software package to be able to create meshes for the geometries one wants to simulate.

A list of available free and commercial mesh generators is available at: http://www.andrew.cmu.edu/user/sowen/softsurv.html

For nmag one needs to create 'unstructured' meshes which means for three dimensional simulations that the mesh simplices are tetrahedra, and the surface elements are triangles.

We are not recommending any mesh generating software. We have used Netgen to generate most of the meshes for this manual. The Vienna/Sheffield group (Fidler and Schrefl) use the commercial mesh generator *GID* (http://gid.cimne.upc.es/).

The mesh format used by nmag is called nmesh and described in *Nmesh file format*.

The *nmeshimport* tool provides conversion from the following mesh formats into nmesh files:

- Netgen (neutral). Create mesh in Netgen, then go to `File->Export Filetype` and ensure that `Neutral Format` is selected. Then export the mesh with `File->Export Mesh`. (See also *Mesh generation* which is part of the *Guided Tour*.)
- There is a contributed import module for Gambit. Use at your own risk.
- Gmsh meshes written file format version 1.0 can be imported.

  If you already have the Gmsh mesh file in format 2.0, then you can use

  ```
  $> gmsh -3 -format msh1 -o outfile.msh infile.msh
  ```

  to create 'outfile.msh' which contains the mesh in the gmesh file format 1.0 that can be imported.

  If you create the mesh interactively, then

  - choose `FILE -> SAVE AS`,
  - select `Gmsh mesh (*.msh)` from the drop down list,
  - choose filename and click `OK`
  - When the `MSH Options` box appears, choose `Version 1.0` from the drop down list in the Format field.
  - click `OK`

  If you create your meshes automatically from the command line, then add `--format msh1` to the command line to instruct Gmsh to write in the 1.0 format.

5.1 Nmesh file format

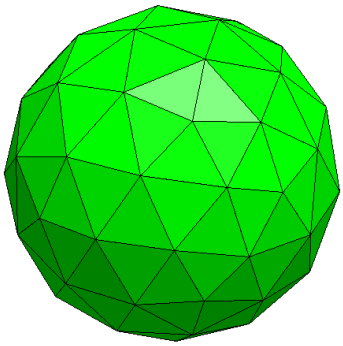
There are two nmesh file formats: *Ascii nmesh* and *Hdf5 nmesh*.

5.1.1 Ascii nmesh

This section describes the syntax of the nmesh ascii files. You only need to read this if you would like to know what the nmesh ascii format looks like. This is only necessary if you (i) need to convert nmesh ascii files into other mesh formats, or (ii) if you have generated a mesh in a format that is not supported by *nmeshimport*.

(You could in principle convert any data into the *nmesh hdf5* format but it is expected that it is easier to convert the mesh into a *nmesh ascii* file, and then use *nmeshpp* with option `--convert` to convert the mesh frnm nmesh ascii to nmesh hdf5.)

We describe the structure of the ascii nmesh file format using the following example: A `mesh file` for a sphere with radius 1 (with Netgen and this geometry file `sphere.geo`):



The mesh file looks as follows:

```
# PYFEM mesh file version 1.0
# dim = 3      nodes = 79      simplices = 174         surfaces = 148          periodic = 0
79
         0.000000000000            0.000000000000            0.000000000000
         1.000000000000            0.000000000000            0.000000000000
         0.911922000000            0.410365000000            0.000000000000
         0.941662000000            0.159249000000            0.296499000000
<snip>
         0.038305000000           -0.552912000000           -0.107777000000
        -0.533150000000            0.052091000000           -0.084880000000
174
     1          58           0          62          78
     1          33          78          36          50
     1          19          77           0          17
     1          67          75          78          72
<snip>
     1          58           0          53          62
     1          77          51          32          39
     1          78          67          63          72
     1          78          49          63          50
148
     -1 1      45 51 59
     1 -1      5 12 13
     1 -1      25 37 39
<snip>
     1 -1      19 30 34
     1 -1      42 44 54
     1 -1      32 45 51
```

We have removed a significant number of lines for the purpose of abbreviation in the places marked with `<snip>`. We discuss the different parts of this file in more detail:

- The file starts with two lines starting with `#`.
  - The first line contains a file format version string which needs to have exactly this form.
  - The second line contains a summary of the data, i.e.:

dim the dimensionality of the space in which the mesh is defined (usually 3, as we work with meshes in 3d space).

nodes the number of nodes (also called vertices), here `79`

simplices the number of simplices (also called volume elements), here `174`. In 3d, a simplex is a tetrahedron.

surfaces the number of surface elements, here `148`. In 3d, the surface elements are triangles.

periodic the number of periodic identifications of points.

- The next section contains the data for all the nodes. The first line of this section contains (again) the total number of nodes (`79`). All subsequent 79 lines in this section contain each the position of one node. Because the dimensionality of space in this example is 3, there are 3 floating point numbers per node (i.e. the x, y and z-component).
- The next section contains the data for the simplices. The first line of this section contains (again) the total number of simplices (here `174`). The next 174 lines contain the following information each:

  The first integer is a *region identifier*. In this example, we have only one region (the sphere). This is useful, if different magnetic materials are simulated at the same time. When the mesh is loaded into nmag, one assigns material properties to these regions.

  The next 4 integers (in 3 dimensions) are node identifiers. The 4 nodes listed here, form a simplex. Note that the very first node has index 0.
- The next section contains the data for the surface elements. The first line contains the number of surface elements (`148`). The next 148 lines contain each the following information:
  - The first two integers are the region identifiers between which the surface is sandwiched. If there is no simplex on one side of the surface, then the "outside" region identifier of `-1` will be used. (It is possible to use other negative numbers to discern between different parts of the outer boundary. This is occasionally important in nsim to specify Dirichlet and von Neumann boundary conditions along different parts of a boundary.)
  - The following integers are the node ids of the nodes that define the surface. (In this example, we have three nodes as the surface elements are triangles.)

  Note that this last section is only contained in the file to make the users' life easier (for, say, plotting of the mesh). This section on surface elements can be omitted and nmesh will read and process the mesh even if the surface elements are not specified (they can be computed from the node and simplex data provided in the other sections).
- The next section contains data about periodic points. The first line again is the number of data lines to follow. Each data line gives the node indices belonging to one set of points that are treated as copies of the same point. (I.e. Nmag will know that field degrees of freedom associated to points from such a set will have "mirage" copies and use this information internally e.g. in the time integrator and when building finite element matrix operators.)

5.1.2 Hdf5 nmesh

In addition to the *Ascii nmesh* format, there is another (binary and compressed) way of storing nmesh meshes. This is based on the hdf5 library which often is abbreviated as `h5`.

We recommend that this file-format to store meshes as it is a compressed binary file format, and thus much more space efficient. The *nmeshpp* tool can convert (using `--convert`) ascii nmesh files into hdf5 files. Likewise, using the *nmeshimport* tool with a target file name that ends in `.h5` will also choose this file format. We strongly recommend to use the extension `.nmesh.h5` (but `.h5` is sufficient to trigger saving meshes in hdf5 format).

For conversion of other mesh formats to a format readable by nmesh, we suggest to bring data into *Ascii nmesh* format, and then convert this ascii nmesh file to a .h5 file.

For completeness, we outline the `nmesh.h5` file format here. Knowledge of hdf5 or pytables may be useful to understand the underlying concepts.

The nmesh.h5 file contains the following nodes (this is output from pytables's `ptdump` program):

```
/ (RootGroup) ''
/etc (Group) 'Configuration and version data'
/etc/filetype (Array(1L,)) 'data file type'
/etc/fileversion (Array(1L,)) 'data file type version'
/mesh (Group) 'Mesh data'
/mesh/points (CArray(1154L, 3L), shuffle,
              zlib(5)) 'Positions of mesh nodes (=points)'
/mesh/simplices (CArray(4953L, 4L), shuffle, zlib(5))
             'Indices of nodes (starting from zero).
              Each row is one simplex.'
/mesh/simplicesregions (CArray(4953L,), shuffle, zlib(5))
             'Region ids (one for each simplex).'
```

In short, the position of the mesh nodes are stored in `/mesh/points` as 8byte-floats. The simplices (i.e. tetrahedra in 3d) are stored in `/mesh/simplices` as a set of integers (4 in 3d) per simplex which are the indices of the mesh nodes that form the simplex. We also need to store to what physical region each simplex belongs. Regions are coded by integer values (with 0 being vacuum, and -1 the area outside the mesh) and stored in `/mesh/simplicesregions`.

5.2 mesh file size

The following table shows the size of the mesh file used in *Example 2: Computing the time development of a system* stored in various formats.

Filename	size (kB)	type
bar30_30_100.neutral	1036	ascii
bar30_30_100.neutral.gz	246	gzipped ascii
bar30_30_100.nmesh	794	ascii
bar30_30_100.nmesh.h5	203	hdf5

The `.neutral` file is the mesh as written by Netgen in this particular format. The second line shows the file size of the same file after compressing with `gzip`. The third line shows the size of the mesh stored as an *Ascii nmesh* file while the last line gives the size of the corresponding *Hdf5 nmesh* file.

CHAPTER

SIX

EXECUTABLES

6.1 ncol

`ncol` is a utility to conveniently analyse *Data files (.ndt)* files.

Suppose we have data file with name X_dat.ndt. We can then use:

```
ncol X_dat.ndt
```

or simply:

```
ncol X
```

to display the content of the file. This is useful to quickly gain an overview of the data in the file. For the *Example 2: Computing the time development of a system*, the command is:

```
ncol bar
```

which produces this output:

```
0 :          #time            #<s>              0
1 :            id              <>              0
2 :          step              <>              0
3 :  last_step_dt             <s>              0
4 :    stage_time             <s>              0
5 :    stage_step              <>              0
6 :         stage              <>              0
7 :    E_total_Py      <kg/ms^2>  -260346.5776034
8 :           phi             <A>  2.50626665111e-07
9 :      E_ext_Py      <kg/ms^2>              0
10:     H_demag_0          <A/m>  -263661.6673782
11:     H_demag_1          <A/m>  -8.212128727093
12:     H_demag_2          <A/m>  -77027.64089399
13:     dmdt_Py_0         <A/ms>  -8.250903922407e+15
14:     dmdt_Py_1         <A/ms>  2.333345040949e+16
15:     dmdt_Py_2         <A/ms>  8.250903922407e+15
16:           pin              <>              1
17:   H_anis_Py_0          <A/m>              0
18:   H_anis_Py_1          <A/m>              0
19:   H_anis_Py_2          <A/m>              0
20:        m_Py_0             <>  0.7071067811865
21:        m_Py_1             <>              0
22:        m_Py_2             <>  0.7071067811865
23:        M_Py_0          <A/m>  608111.8318204
24:        M_Py_1          <A/m>              0
25:        M_Py_2          <A/m>  608111.8318204
26:     E_anis_Py      <kg/ms^2>              0
27:     E_exch_Py      <kg/ms^2>  3.114630036477e-11
28:           rho        <A/m^2>  3.469702141876e+13
```

```
29:       H_ext_0          <A/m>              0
30:       H_ext_1          <A/m>              0
31:       H_ext_2          <A/m>              0
32:  H_total_Py_0          <A/m>  -263661.6673782
33:  H_total_Py_1          <A/m>  -8.212128727085
34:  H_total_Py_2          <A/m>  -77027.64089399
35:    E_demag_Py      <kg/ms^2>  -260346.5776034
36:   H_exch_Py_0          <A/m>  2.037901097873e-11
37:   H_exch_Py_1          <A/m>              0
38:   H_exch_Py_2          <A/m>  2.037901097873e-11
39: maxangle_m_Py          <deg>              0
40:     localtime             <>  2007/10/04-20:46:28
41:      unixtime            <s>  1191527188.269
```

The four columns above show the following data: the first is just a line number count. The second is the *name* of the data. The third provides the *units* of this data type. The fourth displays the first data value in the file (typically corresponding to the configuration of the simulation when *save_data* was called the first time).

The meaning of the keywords `time`, `id`, `step`, `stage_time`, `stage_step`, `stage`, `localtime` and `unixtime` is explained in section *Stage, Step, iteration, time, etc.*. The role of the **id** counter is to provide a reference to the configuration that was saved, and it is a **unique identifier** of a physical configuration. It is used to identify configurations in the `_dat.h5` file (which stores spatially resolved fields) and to identify the corresponding (spatially averaged) data in the `_dat.ndt` file. This `id` is used to uniquely identify physical configurations in nmag. (See also: *Why can you not use the step as a unique identifier?*)

`last_step_dt` is the length of the last time step carried out by the timestepper. This is a useful indicator to learn about the stiffness of the system: the time step is adjusted automatically to achieve a certain accuracy, and thus the size of the time step reflects how hard it is to integrate the equations of motion.

The fields starting with `E_total_Py` down to `H_exch_Py_2` are all nsim subfields (see *fields*), and the data stored for these are spatially averaged numbers. For example, the subfield `M_Py_0` is the x-component of the Magnetisation of the material `Py` averaged over all the space where this material is defined.

The `maxangle_m_Py` is the maximum angle (in degree) of the change of the magnetisation from one node in the mesh to the next. It is important that this number is small: the equations on which the micromagnetic theory is based assume that the magnetisation changes slowly as a function of space. In the discretised solvers (this applies to nmag as it applies to OOMMF, Magpar and other codes), this means that the maximum angle between neighbouring sites should be kept small. How small is good enough? This is hard to say in general. We provide some (subjective) guidance: Values of 180 degrees (or -180 degrees) quite clearly indicate that the results of the calculations must not be trusted (i.e. they are wrong). Values around 90 degrees make the results highly questionable. Values of below 30 degrees indicate that the results are probably reliable. The smaller the value, the more accurate the results will be. If this is new to you, you may want to read the *Mini tutorial micromagnetic modelling* and in particular the section *What size of the cells (FD) and tetrahedra (FE) should I choose?*.

The general syntax for calling `ncol` is:

```
ncol [OPTIONS] datafile [COLS]
```

A list of options can be obtained with:

```
ncol --help
```

Available options include:

```
-h, --help            show this help message and exit
--scale="{col1:factor1,col2:factor2,col3:factor3}"
                      scale col1 by factor1, col2 by factor 2 etc
--last-of="column"    Select only the rows where 'column' changes.
-l                    Select only the last row for each stage (i.e.
                      typically the relaxed state)
--mod="field"         Compute the magnitude of given field, i.e.
                      '--mod H_demag' computes
                      sqrt(H_demag_0^2+H_demag_1^2+H_demag_2^2). More than
                      one field can be provided (comma separated) but there
```

```
                      must be no spaces between the fields. (I.e. '--mod
                      m_Py,H_ext'). These modulus entries will be printed
                      last (after any other COLS that have been provided),
                      and in the order given in the '--mod' switch.
--odt                 Expect to process odt file (as produced by OOMMF, see
                      http://math.nist.gov/oommf/ and http://math.nist.gov/o
                      ommf/doc/userguide12a3/userguide/Data_Table_File_Forma
                      t_ODT.html).
```

6.2 nmagpp

The stage `nmagpp` program is the NMAG data PostProcessor. It can be used to

- convert data stored in `RUNID_dat.h5` files into vtk files
- dump the data to the screen.

The documentation is available with the `--help` switch:

```
nmagpp --help
```

6.2.1 Inspecting the content

We describe some typical scenarios, using the data file `bar_dat.h5` that is generated in *Example 2: Computing the time development of a system*.

The `bar_dat.h5` file contains spatially resolved data for all fields in the simulation (because we have used the `save_data(fields='all')` command). Some of the functions of `nmagpp` apply to one or more fields (such as `--dump` and `--vtk`) and these can be specified through a `--fields` command line parameter. Similarly, the `--range` command will limit the number of saved configurations which will be processed.

Try `nmagpp --help` for further documentation. Some examples:

- Checking what at what configurations have been saved:

  ```
  nmagpp --idlist bar
  ```

  produces:

  ```
  id   stage   step     time fields
   0->     1      0        0 E_anis E_demag E_exch E_ext E_total H_anis H_demag  ... phi pin rh
  10->     1    312     5e-11 E_anis E_demag E_exch E_ext E_total H_anis H_demag  ... phi pin rh
  20->     1    495     1e-10 E_anis E_demag E_exch E_ext E_total H_anis H_demag  ... phi pin rh
  30->     1    603   1.5e-10 E_anis E_demag E_exch E_ext E_total H_anis H_demag  ... phi pin rh
  40->     1    678     2e-10 E_anis E_demag E_exch E_ext E_total H_anis H_demag  ... phi pin rh
  50->     1    726   2.5e-10 E_anis E_demag E_exch E_ext E_total H_anis H_demag  ... phi pin rh
  60->     1    762     3e-10 E_anis E_demag E_exch E_ext E_total H_anis H_demag  ... phi pin rh
  ```

The `id` is the same *unique identifier id* used in the *Data files (.ndt)* files that can be read with the *ncol* command. In particular, its purpose is to identify time steps saved in the ndt file with the corresponding data saved in the h5 data file.

Columns `time` (measured in seconds), `step` and `stage` are just providing some further information (see *Stage, Step, iteration, time, etc.*) Finally, the available (i.e. saved) fields for every configuration are listed. The list of fields is not displayed completely if it is long (unless the `--printall` switch is used).

6.2.2 Dumping data

Suppose we are interested in the magnetisation data stored at id 0. We restrict the data to the `m` field using the `--fields m` switch, and restrict the number `ids` to dump using `--range 0`:

```
nmagpp --fields m --range 0 --dump bar
```

produces output that starts like this:

```
field    : m
subfield : m_Py
time     : 0 * <SI: 1  s >
id       : 0
step     : 0
stage    : 0
field unit: <SI: 1  >
position unit: <SI: 1e-09  m >
row: 0
#Start (index in h5 table, dofsite, pos, data)
   0:   0 : (  0,  0,  0) : (    0.707107,            0,     0.707107)
   1:   1 : (  3,  0,  0) : (    0.707107,            0,     0.707107)
   2:   2 : (  6,  0,  0) : (    0.707107,            0,     0.707107)
   3:   3 : (  9,  0,  0) : (    0.707107,            0,     0.707107)
```

The first few rows provide some metadata such as which field and subfield the data is about, at what simulation time it was saved (here 0 seconds), what the id, step and stage is. It further shows the `field unit` and the `position unit`. These give the physical dimensions with which the numerical quantities from the table have to be multiplied to get dimensionful physical quantities. For example, the positions in the table are provided as `(0,0,0)`, `(3,0,0)`, `(6,0,0)` etc. These numbers have to be multiplied by `<SI: 1e-09 m >` = 1e-9 meters to obtain the actual positions in SI units. In other words, the position coordinate data is expressed in nanometers. In this particular example, the field data – the normalised magnetisation – is dimensionless.

Followed by the keyword `#Start` the actual data starts (in the next line). The format of the subsequent data lines is as follows:

- Column 1: index of the site in the h5 file. This mostly relevant for developers.
- Column 2: the index of the site. As long as we are dealing with first order basis functions (as is nearly always the case in micromagnetics), this is equivalent to the node id in the mesh.
- Columns 3, 4, 5: enclosed in parentheses, the position of the site is expressed in units of the `position unit`.
- Columns 6, 7, 8: enclosed in parentheses, the actual field data expressed in units of the `field unit`.

In short, the first line of the actual data:

```
0:   0 : (  0,  0,  0) : (    0.707107,            0,     0.707107)
```

tells us that the normalised magnetisation at node id `0`, and position `(0,0,0) nm` is pointing in the direction `(0.707107,0,0.707107)`.

Another example: Suppose we are interested in the magnetisation field `M` (this is the non-normalised magnetisation measured in Ampere per meter) at time 1e-10 seconds (i.e. id=20). We use this command:

```
nmagpp --fields M --dump --range 20 bar
```

to obtain output beginning like this:

```
field    : m
subfield : m_Py
time     : 1e-10 * <SI: 1  s >
id       : 20
step     : 495
stage    : 1
field unit: <SI: 1  >
position unit: <SI: 1e-09  m >
row: 2
#Start (index in h5 table, dofsite, pos, data)
   0          0 (           0            0            0 ) (    0.182556     0.525948     0.830694
   1          1 (           3            0            0 ) (    0.165008     0.534525     0.828888
```

```
2          2 (          6          0          0 ) (    0.104837     0.544846     0.831957
3          3 (          9          0          0 ) (    0.029925     0.552054     0.833272
4
```

In principle, this output data can be parsed by other tools to extract the positions and the data. However, it is hoped that other options of the nmagpp tool (such as the `--vtk` switch) already cover most of the situations where the need to convert data may arise. (If you would like to export the raw data into another file format or application, please contact the *nmag team* to request this feature, as it may be of interest to other people as well.)

It is further possible to access the data in the `_dat.h5` files directly from tailor written post-processing scripts. See *example: post processing of saved field data*.

6.2.3 Range of data to be processed

The `--range` switch allows a variety of ways to express which of the `ids` in the data file should be selected (for dumping to the screen, or conversion to a vtk file). Here are some examples:

```
--range 17                        #will select 17
--range "range(5,10)"             #will select [5,6,7,8,9]
--range "[2,5,10,42]"             #will select [2,5,10,42]
--range "range(10)+[20,25,31,42]" #will select [0,1,2,3,...,9,10,20,25,31,42]
--range "max(ids)"                #will select the last saved id
```

6.2.4 Conversion to vtk file

The command

nmagpp –range 0 –vtk test.vtk bar

will take the dataset with `id`=0 in the `bar_dat.h5` file and convert it to a (binary) vtk file with name `test.vtk`. For vtk files, the default is to convert all fields. However, if a field (or a list of fields) is specified using the `--field` option, then only this field is converted. This may be useful if disk space or conversion time is an issue.

We can convert multiple time steps into a set of vtk files with one command. For example, to convert for all saved configurations all fields into vtk files, use:

```
nmagpp --vtk alltest.vtk bar
```

This will create files `alltest-000000.vtk`, `alltest-000010.vtk`, `alltest-000020.vtk`, `alltest-000030.vtk`, `alltest-000040.vtk`, `alltest-000050.vtk`, and `alltest-000060.vtk`.

The conversion to vtk can be combined with the `--range` command. (See *Range of data to be processed*). For example, to convert every second saved configuration (i.e. ids 0, 20, 40) into vtk files, we could use:

```
nmagpp --range "range(0,60,20)" --vtk x.vtk bar
```

The string “range(0,60,20)” is a Python expression and will evaluate to [0,20,40] (because it is the list of integers starting from 0, going up to [but not including] 60, in steps of 20). This will create files `x-000000.vtk`, `x-000020.vtk` and `x-000040.vtk`.

6.2.5 Other features

Use:

nmagpp –help

to get an overview of other features of nmag, and further details.

6.3 nmeshpp

The `nmeshpp` program is the NMESHPreProcesser and NMESHPostProcessor. It provides quick access to some statistical information about nmesh meshes. The basic usage is

```
nmeshpp [OPTIONS] INPUTFILE [OUTPUTFILE]
```

where `INPUT` is the name of a nmesh file (either in ascii or hdf5 format), `OUTPUTFILE` is the name of the file to be written to (if required; this depends on the `OPTIONS`) and OPTIONS can be one or several of the options listed in the following subsections. We use the mesh file `bar30_30_100.nmesh.h5` from *Example 2* to illustrate the usage of *nmeshpp*.

6.3.1 General information (`--info`)

The command:

```
nmeshpp --info bar30_30_100.nmesh.h5
```

produces the following output:

```
====== Info output: ============================================================
3-dimensional mesh
18671 volume elements (3d)
 3438 surface elements (2d)
 4086 points
    1 simplex regions ([1])
    2 point regions ([-1, 1])
    2 region volumes ([0.0, 89999.999999999782])
 1721 boundary points (-> BEM size<=  22MB)
    0 periodic points (mirage=0, total=0)
a0: average=3.543451, std=0.581220, min=1.953689, max=5.708395
```

Starting from the top of the output, we are given the information that this is a three-dimensional mesh, with its number of *volume elements* (i.e. tetrahedra in 3d), *surface elements* (i.e. surface triangles) and *points*.

We are also given a list of *simplex regions* (which is just [1] here). If we had more than one region defined (say two disconnected spheres that are to be associated with different material), then we would have two entries here. The numbers given in this list are the identifiers of the regions: in this example there is only one region and it has the identifier 1.

The *point regions* is a list of all regions in which points are located. This includes of course region 1. Region -1 represents the vacuum around the meshed region. The points that are located on the surface of the bar are located both in the bar (region 1) and in the vacuum (region -1). Other negative region numbers (-2, -3) can be used to discern different pieces of a boundary. (While this feature is at present not used by Nmag, the underlying nsim framework provides capabilities to e.g. associate Dirichlet boundary conditions to a 1/-1 boundary and von Neumann boundary conditions to a 1/-2 boundary.)

The *region volumes* provide the geometrical volume of the regions. By convention, the vacuum has volume 0. In this example, the bar volume is meant to be 30x30x100=90000. The deviation from this due to limited numerical precision (and of the order of 1e-10).

The *boundary points* are the number of nodes located at the surface of the bar. This number is important if using the hybrid finite element/boundary element method to compute the demagnetisation field, as the boundary element matrix size will be proportional to the square of the number of boundary points. The size of the boundary element matrix is given as well (see *Memory requirements of boundary element matrix*).

The *periodic points* are the number of points that have *mirage images* in the mesh. There will always be zero periodic points (and thus zero mirage images) unless we are dealing with a periodic mesh (see *nmeshmirror* and *Example: Spin-waves in periodic system*).

Finally, we are given some information about the statistics of the edge lengths *a0* in the mesh: the average value, the standard deviation, the maximum and minimum value. This is important as in micromagnetics the angle of the magnetisation must not vary strongly from one node to the next. In practice, the edge length a0 should therefore

be (significantly) smaller than the exchange length (see *What size of the cells (FD) and tetrahedra (FE) should I choose?*)

6.3.2 Memory requirements of boundary element matrix

The boundary element matrix is densely populated matrix with s rows and s columns, where s is the number of surface nodes in the mesh. (Strictly, it is only the number of surface nodes that enclose a ferromagnetic material.) Assuming we use 8 bytes to store one floating point number, we can thus estimate the memory required to store this matrix. In the example above, we have 1721 boundary points, and thus 1721*1721=2961841 matrix entries. Each entry requires 8 byte, so the total memory requirement is 23694728 bytes, or approximately 23139 kilobytes or 23 megabytes.

The `nmeshpp -i` command can be used to quickly check how big the BEM matrix is. A computation is only feasible if the RAM of the computer can hold the boundary element matrix. (When carrying out a distributed calculation, it is sufficient if the total RAM of all machines can hold the matrix.)

6.3.3 Inspecting the quality of a mesh

The quality of a mesh can be defined in various ways. In micromagnetics, we usually want tetrahedra that have edges of nearly identical length (i.e. we do not want the tetrahedra to be flat).

`nmeshpp` uses the ratio of the radius of the in-sphere (the sphere that can just fit into a tetrahedron so that it touches the sides) to radius of the circumsphere (the sphere passing through the four corners), multiplied by the number of dimensions. This number is 1.0 for a perfect tetrahedron with identical edge lengths, and 0 for a completely flat (effectively 2-dimensional) tetrahedron.

The command:

```
nmeshpp -q bar30_30_100.nmesh.h5
```

computes a histogram of the distribution of this quality parameter for the bar mesh, and produces this output:

```
====== Quality output: ============================================================
[qual interval] counts = probability
[ 0.000- 0.100]      0 = 0.00%
[ 0.100- 0.200]      0 = 0.00%
[ 0.200- 0.300]      0 = 0.00%
[ 0.300- 0.400]      0 = 0.00%
[ 0.400- 0.500]      1 = 0.01% *
[ 0.500- 0.600]     42 = 0.22% *
[ 0.600- 0.700]    364 = 1.95% **
[ 0.700- 0.800]   2420 =12.96% ************
[ 0.800- 0.900]   8252 =44.20% ****************************************
[ 0.900- 1.000]   7592 =40.66% ************************************
```

6.3.4 Histogram of edge lengths

The command:

```
nmeshpp -a bar30_30_100.nmesh.h5
```

computes a histogram of the edge length distribution of the mesh:

```
====== a0 output: ================================================================
[a0   interval] counts = probability
[ 1.954- 2.329]    234 = 0.63% **
[ 2.329- 2.705]   1424 = 3.81% *******
[ 2.705- 3.080]   7921 =21.17% ************************************
[ 3.080- 3.456]   8790 =23.50% ****************************************
[ 3.456- 3.831]   7573 =20.24% ***********************************
```

```
[ 3.831- 4.207]   5884 =15.73% ************************
[ 4.207- 4.582]   3769 =10.08% ****************
[ 4.582- 4.957]   1385 = 3.70% *******
[ 4.957- 5.333]    365 = 0.98% **
[ 5.333- 5.708]     63 = 0.17% *

average   a0: <a0>   = 3.543451
stand dev a0: <a0^2> = 0.581220^2
min and max :        =(1.953689,5.708395)
```

6.4 Convert nmesh.h5 to nmesh file (and back)

The command:

```
nmeshpp -c mesh.nmesh.h5 mesh.nmesh
```

converts a the mesh `mesh.nmesh.h5` (in h5 format) to the `mesh.nmesh` (in ascii format). Works also in the reverse way. `nmeshpp` will save as `h5` file if the last extension of the file to write is `.h5`.

6.4.1 nmeshmirror

The `nmeshmirror` tool can create periodic meshes out of a non-periodic mesh. The geometry described by the non-periodic mesh has to be a cuboid. This can be mirrored along one (or more) of the planes defined by the sides of the cuboid.

The general usage is

```
nmeshmirror meshfile error1 error2 directions newfile remove
```

where:

- `meshfile` is the original (non-periodic) ASCII nmesh file
- `error1` is the maximum distance between two points in order to consider them coincident (case of points on mirroring planes)
- `error2` is the maximum distance between a point and the surface opposite to the one used as mirroring plane in order to consider the point periodic
- `directions` is a list of values 0,1 or -1, corresponding to the direction(s) over which the mesh is mirrored: 1 corresponds to mirroring along the positive given axis, -1 along the negative given axis and 0 corresponds to no mirroring along the given axis.

  For a three dimensional mesh, there are three options to mirror the mesh (along the x, y and z direction). In that case, the `directions` would be a list of three integers, for example `0,1,0` to mirror the input mesh on the xz plane that limits the mesh in the y direction.
- `newfile` is the name of the ASCII file with the new periodic mesh
- `remove` is an optional argument which takes the values 0 and 1 and removes the periodic points from the final mesh when is set to 1. The default value is 0.

Calling `orig.nmesh` the ASCII file of a 3D non-periodic mesh, an example of the use of nmeshmirror is the following, where the mesh is mirrored along the positive x-axis and the negative z-axis:

```
nmeshmirror orig.nmesh 1e-6 1e-6 1,0,-1 periodic.nmesh
```

resulting in a periodic mesh along the same axes.

6.4.2 nmeshsort

The `nmeshsort` script sorts the nodes of a mesh along a given axis (not recommended when using parmetis with multiple-object meshes). We expect this to be most relevant to developers.

The general usage is

```
nmeshsort meshfile axis newfile
```

where:

- `meshfile` is the original ASCII nmesh file
- `axis` is the axis over which the sorting takes place
- `newfile` is the name of the ASCII file with the new periodic mesh

Calling `orig.nmesh` the ASCII file of a 3D mesh, an example of the use of nmeshsort is the following, where the mesh is sorted along the z-axis:

```
nmeshsort orig.nmesh 2 sorted.nmesh
```

6.5 nmeshimport

The *nmeshimport* command can be used to read other mesh formats and write them into the nmesh format that can be read by nmag.

The `nmeshimport` tool can convert Netgen, Gambit and Gmsh files into nmesh files.

The general usage is:

```
nmeshimport OPTIONS INPUTFILE NMESHFILE
```

The OPTION to import from Netgen is `--netgen`. The (contributed) code for importing from a Gambit mesh file is `--gambit`. The OPTION to import from Gmsh is `--gmsh`.

Usage example: assuming we have a file `mymesh.neutral` created with Netgen and would like to convert it to `mymesh.nmesh.h5`, we could use this command:

```
nmeshimport --netgen mymesh.neutral mymesh.nmesh.h5
```

Use:

```
nmeshimport --help
```

to see all available features.

6.6 nsim

This is the main executable. It superficially appears to be a Python interpreter, but has extended functionality. In particular, it has support for parallel execution (using MPI), and contains extensions accessible in the additional built-in `ocaml` module which provides the additional functionality of the nsim multiphysics system. (Nmag is a Python library on top of nsim, which itself is implemented in Objective Caml.)

6.7 nsimversion

A script that provides some information about the version of the software.

If you need to report a bug/problem, please include the output of this program.

From release 0.2 onwards, please use:

```
nsim --version
```

instead.

CHAPTER

SEVEN

FILES AND FILE NAMES

7.1 mesh files (`.nmesh, .nmesh.h5`)

Files that contain a finite element mesh. See *Nmesh file format*.

7.2 Simulation scripts (`.py`)

Files that contain simulation program code. The ending is (by convention) `.py` which reflects that the programming language used is Python.

All the example codes provided in the *Guided Tour* are such simulation scripts.

7.3 Data files (`.ndt`)

`ndt` stands for Nmag Data Table, analog to `odt` files (OOMMF Data Table) for the OOMMF project. In fact, `ndt` and `odt` files are very similar.

`ndt` files are ascii files where each row corresponds to one time step (or, more generally, configuration of the system). The columns contain:

- metadata such as
  - a unique identifier for every row
  - the time at which the row was written
- (spatially) averaged *field* data

The first two lines contain information about what data is stored in the various columns:

1. The first line provides a header
2. The second line provides the SI units

All other lines contain the actual data.

The file can be loaded into any data processing software (such as MS Excel, Origin, Matlab, Gnuplot, ...). However, often it is more convenient to use the *ncol* tool to select the relevant columns, and only to pass the filtered data to a post-processing (e.g. plotting) program.

Data is written into the `ndt` file whenever the *save_data* method of the simulation object is called.

7.4 Data files (.h5)

The h5 data files store spatially resolved *fields*. The format is a binary and compressed hdf5 format to which we have convenient access via the pytables package for Python. The user should not have to worry about reading this file directly, but use the *nmagpp* tool to access the data.

7.5 File names for data files

The filenames for the *Data files (.ndt)* and *Data files (.h5)* are given by concatenation of the *simulation name*, the extension _dat. and the extension (.h5 or .ndt).

When a simulation object is created, for example in a file called mybar.py starting like this:

```
import nmag
sim = nmag.Simulation(name="bar")
```

then the simulation name is bar.

If no name is provided, i.e. the file mybar.py starts like this:

```
import nmag
sim = nmag.Simulation()
```

then the simulation name will be the *run id*. The *run id* is the filename of the simulation script (without the .py extension), i.e. the simulation name then will be mybar.

Let us assume for the rest of this section that the simulation name is bar. Once we use the *save_data* command, for example like this:

```
sim.save_data()
```

an ndt file will be created, with name bar_dat.ndt (= bar + _dat. + ndt).

Similarly, if we write the fields spatially resolved:

```
sim.save_data(fields='all')
```

a *h5 data file* with name bar_dat.h5 (= bar + _dat. + h5) will be created.

7.6 File names for log files

A log file is created that stores (most of) the messages displayed to stdout (i.e. the screen). The name of the log file starts with the name of the simulation script (without the .py extension), and ends with _log.log.

For example, a simulation script with name mybar.py will have an associated log file with name mybar_log.log.

Another three files will be created if the (undocumented) --dumpconf switch is provided. This are primarily of use to the developers and can usually be ignored:

- mybar_log.conf: This can be used to configure what data is logged.
- mybar_ocaml.conf: Configuration of some variables used in the ocaml code
- mybar_nmag.conf: Some variables used in the nmag code

CHAPTER

EIGHT

FREQUENTLY ASKED QUESTIONS

8.1 What is the difference between the OOMMF and nmag approach?

There are several aspects. One important point is the calculation of the demagnetisation field as this is a computationally very expensive step.

OOMMF is based on discretising space into small cuboids (often called 'finite differences'). One advantage of this method is that the demag field can be computed very efficiently (via fast Fourier transformation techniques). One disadvantage is that this methods works less well (i.e. less accurately) if the geometry shape does not align with a cartesian grid as the boundary then is represented as a staircase pattern.

nmag's finite elements discretise space into many small tetrahedra. The corresponding approach towards the computation of the demagnetisation field (which is the same as Magpar's method) is based on the Fredkin and Koehler Hybrid Finite Element/Boundary Element method. The advantage of this method (over OOMMF's approach) is that curved and spherical geometries can be spatially resolved much more accurately. However, this method of calculating the demagnetisation field is less efficient than OOMMF's approach for thin films. (In particular: memory requirements for the boundary element method grow as the square of the number of surface points.) Note that for simulation of thin films, the hybrid Finite Element/Boundary Element (as used by nmag and Magpar) is likely to require a lot of memory (see *Memory requirements of boundary element matrix*).

There are other points that are related to the fundamentally different discretisation approach used to turn a field theory problem (with a conceptually infinite number of degrees of freedom) into a finite problem: OOMMF assumes the magnetisation in every cell to be constant (with jumps at boundaries), while Nmag assumes magnetisation to be continuous and vary linearly within cells (thus slightly violating the constraint of constant magnitude within a cell of non-constant magnetisation).

8.2 ... So, this means the major difference is "cubes" vs. "tetrahedra"?

No. Simplicial mesh discretisation is fundamentally different from finite-difference discretisation. With OOMMF, say, magnetisation degrees of freedom are associated with the centers(!) of the cells, while with nmag, they are associated with corners. This conceptual difference has many implications, e.g. for the question how to conceptually deal with the exchange interaction between different materials.

8.3 Why do you have your own Python interpreter (`=nsim`)?

In order to provide the ability to run code in a distributed environment (using MPI), we cannot use the standard Python executable. (Technically speaking, a program started under MPI control will receive extra MPI-related command line arguments which upset the standard Python interpreter.) It so happens that – by providing our own Python executable which is called `nsim` – we have easier access to the low-level library of nsim which is written in Objective Caml.

8.4 What is nsim - I thought the package is called nmag?

The *The nsim library* is our general purpose multi-physics simulation environment. The corresponding executable is started through the *nsim* command. Nmag is a collection of scripts that provide micromagnetic functionality on top of nsim. For this reason, nsim is being mentioned a lot in the manual.

8.5 How fast is nmag in comparison to magpar?

Internally, some of the magpar and nmag core components are structurally very similar. In particular, the time integration routine is almost identical up to some philosophical issues such as how to keep the length of the magnetisation vector constant, and whether or not to use a symmetrical exchange matrix and a post-processing step rather than combining these into an asymmetrical matrix, etc. The actual wall clock time used will depend to a large degree on the requested accuracy of the calculations (see *example timestepper tolerances*).

Given equivalent tolerance parameters, we have found (the single-process version of) nmag to be about as fast as magpar. The computation of an individual velocity dM/dt is very similar in nmag and magpar, and about equally efficient. However, we observe that, depending on the particular problem, subtle differences in the philosophies underlying time integration can lead to noticeable differences in the number of individual steps required to do some particular simulation, which can be up to about 25% of simulation time in either direction.

Setup time is a different issue: nmag derives its flexibility from abstract approaches where magpar uses hard-coded compiled functions. Where magpar uses a hand-coded Jacobian, nmag employs the nsim core to symbolically compute the derivative of the equations of motion. There is a trade-off: the flexibility of being able to introduce another term into the equations of motion without having to manually adjust the code for the Jacobian comes at a price in execution time. Therefore, nmag's setup time at present is far larger than magpar's. This can be alleviated to a considerable degree by providing hard-coded "bypass routines" which can be used as alternatives to the symbolically founded methods for special situations that are frequently encountered (such as setting up a Laplace operator matrix). Conceptually, it is easy to add support for this but due to limited manpower, it has not happened yet.

In short: once the setup stage is over, nmag is about as fast as magpar. Magpar's setup time, however, is much smaller. Magpar is also more efficient in memory usage.

8.6 How do I start a time-consuming nmag run in the background?

While this is a Unix rather than a nmag issue, it comes up sufficiently often to address it here.

Well-known techniques to run programs in the background are:

- Using the "nohup" (no-hangup) command, as in:

  ```
  nohup nsim sphere1.py &
  ```

- Using the at-daemon for scheduling of command execution at given times:

  ```
  at now
  warning: commands will be executed using /bin/sh
  at> nsim example1.py
  at> <EOT>
  job 2 at Fri Dec 14 12:08:00 2007
  ```

- Manual daemonization by using a parent process which forks & exits, as in:

  ```
  perl -e 'exit(0) if fork(); exec "nsim sphere1.py"'
  ```

  (But if you know Unix to that degree, you presumably would not have asked in the first place.)

- One of the most elegant ways to start a process in the background is by using the "screen" utility, which is installed on a number of Unix systems. With "screen", it becomes possible to start a text terminal session in such a way that one can "detach" from it while keeping the session alive, and even log out and log in again much later and from a different machine, re-attaching the terminal session and continuing work from the point where it was left.

  While it is a good idea to read the documentation, most basic usage of "screen" requires the knowledge of three commands only:

  - With "screen -R", one can re-attach to a running session, automatically creating a new one if none was created before.
  - Within a "screen" session, Control+a is a prefix keyboard command for controlling "screen": Pressing Control-a and then Control-d will detach the session.
  - Control-a ? will being up a help screen showing all "screen" keyboard commands.

8.7 nmag claims to support MPI. So, can I run simulation jobs on multiple processors?

Yes. See *Example: Parallel execution (MPI)*.

8.8 How should I cite nmag?

Please cite:

- Thomas Fischbacher, Matteo Franchin, Giuliano Bordignon, and Hans Fangohr. *A Systematic Approach to Multiphysics Extensions of Finite-Element-Based Micromagnetic Simulations: Nmag*, in IEEE Transactions on Magnetics, **43**, 6, 2896-2898 (2007). (Available online)

8.9 Why can you not use the step as a unique identifier?

There are two reasons. Firstly, nmag may be extended in future to support effective energy minimisation in which case the `step` becomes somewhat meaningless (although it could probably still be used as an identifier if we identify minimisation iterations with steps). Secondly (and more importantly), in nmag, the user can modify the magnetisation directly using *set_m* (either scripted or interactively). This will change the configuration of the system without increasing the step counter of the time integrator. For this reason, we have the *unique identifier id*.

8.10 How to generate a mesh with more than one region using GMSH?

To assign different material properties to different objects, the mesher needs to assign different region number to different simplices of the mesh. The manual shows how to do this for netgen (see `two_cubes.geo`) file in example *Example: two different magnetic materials*).

How does one define different regions using GMSH? User Xu Shu (Wuhan, China) kindly provides this solution:

> Within GMSH, one has to firstly "add physical groups" and choose the two detached volumes separately to add them into different groups, then choose "edit" to redefine the number of the two groups, thus you can get two physical objects as you want.

8.11 Can I run more than one simulation in one directory?

If you want to run two (or more) simulations in the same directory, then this is fine as well as long as they have different *simulation names*.

The simulation name is either the string given to the constructor of the simulation object, or – if no name is defined explicitly – the name of the python file that contains the simulation script (without the `.py` extension). See *File names for data files* for a detailed example for this.

Data and log files will all start with the simulation name, followed by some specific appended string and specific file extensions. It is thus safe to run simulations with different names in the same directory.

8.12 Can I save data to an arbitrary directory?

8.12.1 Do you really need to do so?

First, consider whether you really need to save data in a different directory. Remember that you can run many simulation with one single script just using a different simulation name, like:

```
...
s1 = Simulation('one')
...
s2 = Simulation('two')
...
```

When you save the data for simulation `one` you get files like `one_dat.h5` and `one_dat.ndt`, while when dealing with simulation `two` you get `two_dat.h5` and `two_dat.nd5`. There is no interference between the two simulations (and in particular it is necessary to save thedata in different directories.)

8.12.2 How to save data to a different directory

When you run a simulation script which saves data from a simulation, the files are saved by default in the current working directory. In order to change this and save data into a directory called `./mydir/` you should start your

script in the following way:

```
import nmag
import nsim.features
fts = nsim.features.Features()
fts.set('etc', 'savedir', './mydir/')
...
...
```

Alternatively, you can change the current working directory at the beginning of the file with ordinary Python code:

```
import os
initial_dir = os.path.abspath(os.path.curdir)
os.chdir('./mydir')
...
...
os.chdir(initial_dir)
```

If the directory you want to write to does not exist then (in both the two example) you may have to create it first, with something like:

the_dir = './mydir' import os if not os.path.exists(the_dir):

os.mkdir(the_dir)

8.13 How to check the convergence of a simulation

How long it takes to run a simulation? This depends very much on what you are simulating and under what conditions (applied field, current, etc). Sometimes, however, your simulation may not be ending as quickly as you expected and you may want to check what is happening. It may be, indeed, that the simulation is not converging, which means that it may actually never end. One thing you can do in such a case is to take a look at the file `*_progress.txt`, where `*` stands for the simulation name (given to the `Simulation` class when creating the simulation oject). For example, if you created your simulation object with a line such as:

```
s = nmag.Simulation('one')
```

Then you may be looking for a file with name `one_progress.txt`. If you used simply `s = nmag.Simulation()` and your file is named `two.py` then you should look for a file with name `two_progress.txt`. This file contains statistics about the time integrator. You'll first get the current time, step number, etc. Then you'll get a list of rows each containing four columns, such as:

```
123 0.456 0.123 None
```

Column 1 is the step reached, an integer number which always increases. The file shows convergence statistics for the last few steps (it doesn't contain statistics for all the steps, since this would make it quickly very big). Column 2 contains the current value of max || dM/dt ||. Column 3 contains the stopping value of dM/dt. Convergence is reached when column 2 < column 3 for at least two times. If the simulations is going well, then you should see that column 2 contains numbers which are not oscillating rapidly and are rather decreasing or increasing "smoothly". This is what typically should happen, even if it can be that your simulation has really a bizarre dynamics which really oscillates in a frenetic way, so one should be careful when analysing the data. The fourth column contains an evaluation of the quality of the convergence according to what we just said. This number should be close to one when the convergence is smooth and close to zero when it is oscillating dramatically.

8.14 What to do in case of convergence problems

If your simulation has really a convergence problem, you can do two things:

- improve the tolerances `ts_abs_err` and `ts_rel_err` (decrease these numbers) by using the method `set_params` of the `Simulation` object;

- use a `do=[('next_stage', at('stage_time',SI(x, 's')))]` as an argument to the `hysteresis` method. This way you impose a maximum time `x` to spend in the computation of a stage (you should make sure this makes sense in your case).

8.15 How to visualise the difference between two fields defined over the same mesh

First save the data into two **ASCII** VTK files. For example:

```
nmagpp --vtk=m.vtk --vtkascii --fields=m simulation_name
```

Note the option `"--vtkascii"` to force the creation of a ASCII file. Let's say this command created the two files `m-000000.vtk` and `m-000001.vtk`. You can now use the library `pyvtk` to load the two files, compute the difference and save it back to a third file:

```
import numpy, pyvtk
a = pyvtk.VtkData("m-000000.vtk")
b = pyvtk.VtkData("m-000001.vtk")
va = a.point_data.data[0].vectors
vb = b.point_data.data[0].vectors
for i in range(len(va)):
    va[i] = list(numpy.array(va[i]) - numpy.array(vb[i]))
a.tofile("difference.vtk")
```

Save this text to a file named `diff.py` and run it as:

```
python diff.py
```

You'll get a third file with name `difference.vtk` containing the difference of the two fields.

If you are repeating this operation many times, it may become annoying to open again and again the `diff.py` file to change the names of the input files. You can then modify the script as follows:

```
import sys, numpy, pyvtk
a = pyvtk.VtkData(sys.argv[1])
b = pyvtk.VtkData(sys.argv[2])
va = a.point_data.data[0].vectors
vb = b.point_data.data[0].vectors
for i in range(len(va)):
    assert a.structure.points[i] == b.structure.points[i]
    va[i] = list(numpy.array(va[i]) - numpy.array(vb[i]))
a.tofile(sys.argv[3])
```

The name of the files are taken from the command line. You can then compute the difference using:

```
python diff.py a.vtk b.vtk a_minus_b.vtk
```

Notice that in the last version of the script we also added the line:

```
assert a.structure.points[i] == b.structure.points[i]
```

which does just check that the two files are using the same set of points (i.e. the same mesh).

8.16 How to re-sample data from a saved h5 file

(Available in Nmag-0.2.0)

You can load an h5 file like this

```
import ocaml
from nmag.h5probe import Fields
handler = Fields("infile.h5")
field = handler.set_field_data("m", "Py", 0)
```

And probe one of its fields:

```
position = [0, 1, 2] # In mesh units (typically is nanometres)
value = ocaml.probe_field(field, "m_Py", position)[0][1]
```

This way you can create two arrays: `rs` containing an array of points and `vs` containing the corresponding values. You can then use pyvtk to generate a VTK file from these:

```
import pyvtk

grid = pyvtk.UnstructuredGrid(rs)
data = pyvtk.PointData(pyvtk.Vectors(vs))
v = pyvtk.VtkData(grid, data)
v.tofile("outfile.vtk")
```

Here is a full example, which probes the magnetisation in the outer skin of a cylinder, in sections which are not equally spaced. Notice the usage of the function `float_set` to specify where the sampling should be denser (originally, here is where a domain wall was). The script should be used as `nsim probe.py infile.h5 outfile.vtk`:

```
import math
import sys

import pyvtk

import ocaml
from nmag.h5probe import Fields
from nmag import float_set

# First we probe the field in the required points
handler = Fields(sys.argv[1])
field = handler.set_field_data("m", "Py", 0)

xs = float_set([-150.0, -145.0, [], -15.0, -12.5, [], 15.0, 20.0, [], 50.0])
angles = float_set([0, [20], 2*math.pi])
R, R2 = (4.9, 5.1)

rs = []
vs = []
for x in xs:
  for angle in angles:
    r = [x, R*math.cos(angle), R*math.sin(angle)]
    rs.append([x, R2*math.cos(angle), R2*math.sin(angle)])
    vs.append(ocaml.probe_field(field, "m_Py", r)[0][1])

# Now we output the values to a VTK file
grid = pyvtk.UnstructuredGrid(rs)
data = pyvtk.PointData(pyvtk.Vectors(vs))
v = pyvtk.VtkData(grid, data)
v.tofile(sys.argv[2])
```

8.17 Notes on using GMSH to create a family of related meshes

If you want to create many meshes using Gmsh, you may first generate a mesh manually. Then you can create a Python script which uses this mesh as a template to quickly create a mesh for a different set of parameters. Below is such a script which shows how to do so. The mesh file (geo) has been enclosed between quotes `"""` and some

of the values for the points coordinates have been substituted with strings that the Python script substitutes with real values.

Note that we use `Mesh.CharacteristicLengthFactor = 5.0;` to control the discretisation of the mesh. We also use `Physical Volume(1) = {1};` to make sure that the mesh region is labeled starting from region number 1:

```
mesh = """
cl1 = 1;
Point(1) = {$x2$, 0, 0, cl1};
Point(2) = {$x2$, $x2$, 0, cl1};
Point(3) = {0, $x2$, 0, cl1};
Point(4) = {0, $x1$, 0, cl1};
Point(5) = {$x1$, 0, 0, cl1};
Point(6) = {$x0$, $x0$, 0, cl1};
Point(7) = {$x0$, $x1$, 0, cl1};
Point(8) = {$x1$, $x0$, 0, cl1};
Point(9) = {$x2$, 0, $y1$, cl1};
Point(10) = {$x1$, 0, $y1$, cl1};
Point(14) = {$x1$, $x0$, $y1$, cl1};
Point(18) = {$x0$, $x0$, $y1$, cl1};
Point(19) = {$x0$, $x1$, $y1$, cl1};
Point(23) = {0, $x1$, $y1$, cl1};
Point(27) = {0, $x2$, $y1$, cl1};
Point(31) = {$x2$, $x2$, $y1$, cl1};
Line(1) = {1, 5};
Line(2) = {5, 8};
Circle(3) = {8, 6, 7};
Line(4) = {7, 4};
Line(5) = {4, 3};
Line(6) = {3, 2};
Line(7) = {2, 1};
Line(11) = {9, 10};
Line(12) = {10, 14};
Circle(13) = {14, 18, 19};
Line(14) = {19, 23};
Line(15) = {23, 27};
Line(16) = {27, 31};
Line(17) = {31, 9};
Line(19) = {1, 9};
Line(20) = {5, 10};
Line(24) = {8, 14};
Line(28) = {7, 19};
Line(32) = {4, 23};
Line(36) = {3, 27};
Line(40) = {2, 31};
Line Loop(9) = {1, 2, 3, 4, 5, 6, 7};
Plane Surface(9) = {9};
Line Loop(21) = {1, 20, -11, -19};
Ruled Surface(21) = {21};
Line Loop(25) = {2, 24, -12, -20};
Ruled Surface(25) = {25};
Line Loop(29) = {3, 28, -13, -24};
Ruled Surface(29) = {29};
Line Loop(33) = {4, 32, -14, -28};
Ruled Surface(33) = {33};
Line Loop(37) = {5, 36, -15, -32};
Ruled Surface(37) = {37};
Line Loop(41) = {6, 40, -16, -36};
Ruled Surface(41) = {41};
Line Loop(45) = {7, 19, -17, -40};
Ruled Surface(45) = {45};
Line Loop(46) = {11, 12, 13, 14, 15, 16, 17};
```

```
Plane Surface(46) = {46};
Surface Loop(1) = {9, 46, 21, 25, 29, 33, 37, 41, 45};

Volume(1) = {1};

Physical Volume(1) = {1};

Mesh.CharacteristicLengthFactor = $discret$;
"""

def create_mesh(filename,
                inner_size=100.0,
                curvature=5.0,
                width=10.0,
                thickness=20.0,
                discretisation=2.5):
  global mesh
  s = str(mesh)
  x = 0.5*inner_size
  variables = [("x0", x - curvature),
               ("x1", x),
               ("x2", x + 0.5*width),
               ("y1", thickness),
               ("discret", discretisation)]
  for variable_name, variable_value in variables:
    s = s.replace("$%s$" % variable_name, str(variable_value))

  f = open(filename, "w")
  f.write(s)
  f.close()

create_mesh("dots.geo")
```

CHAPTER

NINE

USEFUL TOOLS

9.1 vtk

The homepage of the Visualisation ToolKit (vtk) is http://www.vtk.org. VTK provides is is an open source, freely available software system for 3D computer graphics, image processing, and visualization. It also provides a file-format which is called 'vtk'. A number of high-level user interfaces exist to visualise data provided in such vtk files. These include:

- *MayaVi* (http://mayavi.sourceforge.net/)
- VisIt (http://www.llnl.gov/visit/)
- ParaView (http://www.paraview.org/)

9.2 MayaVi

"MayaVi is a free, easy to use scientific data visualizer. It is written in Python and uses the Visualization Toolkit (VTK) for graphical rendering. MayaVi is free and distributed under the conditions of the BSD license. It is also cross platform and should run on any platform where both Python and VTK are available (which is almost any Unix, Mac OSX or Windows)." The MayaVi web page is http://mayavi.sourceforge.net/.

MayaVi has been used to generate many of the plots in this manual. Other tools are available for visualisation of vtk files (see *vtk*).

9.3 NumPy

Numerical Python (short *numpy*) is an extension library to Python that provides fast array operations and is designed for numerical work. This Python extension and documentation can be found at http://numpy.scipy.org/

CHAPTER

TEN

CONTACT

The nmag developer team can be contacted at `nmag@soton.ac.uk`.

Questions about the usage of nmag can also be send to the *nmag-users* mailing list.

CHAPTER

ELEVEN

MINI TUTORIAL MICROMAGNETIC MODELLING

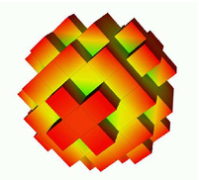
This section is intended for researchers who are just beginning to explore micromagnetic modelling. It is assumed that you have some knowledge on micromagnetics. We advise to read this whole section, and then to look the *Guided Tour* examples (or to explore other *Micromagnetic packages* at that point).

11.1 Introduction micromagnetic modelling

To carry out micromagnetic simulations, a set of partial differential equations have to be solved repeatedly. In order to be able to do this, the simulated geometry has to be spatially discretised. The two methods that are most widely spread in micromagnetic modelling are the so-called finite difference (FD) method and the finite element (FE) method. With either the FD or the FE method, we need to integrate the Landau-Lifshitz and Gilbert equation numerically over time (this is a coupled set of ordinary differential equations). All these calculations are carried out by the *Micromagnetic packages* and the user does not have to worry about these.

The finite difference method subdivides space into many small cuboids. Sometimes the name *cell* is used to describe one of these cuboids. (Warning: in finite difference simulations, the simulated geometry is typically enclosed by a (big) cuboid which is also referred to as *simulation cell*. Usually (!) it is clear from the context which is meant.) Typically, all simulation cells in one finite difference simulation have the same geometry. A typical size for such a cell could be a cube of dimensions 3nm by 3nm by 3nm.

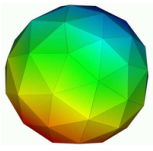
Let's assume we would like to simulate a sphere. The following picture



shows an approximation of the shape of the sphere by cubes. This is the finite difference approach. For clarity, we have chosen rather large cubes to resolve the sphere – in an actual simulation one would typically use a much smaller cell size in order to resolve geometry better.

On the other hand, the finite element method (typically) subdivides space into many small tetrahedra. The tetrahedra are sometimes referred to as the (finite element) mesh elements. Typically, the geometry of these tetrahedra does vary throughout the simulated region. This allows to combine the tetrahedra to approximate complicated geometries.

Using tetrahedra, the a discretised sphere looks like this:



The spherical shape is approximated better than with the finite differences.

The first step in setting up a micromagnetic simulation is to describe the geometry. In the case of finite difference calculations, it will depend on the package you use (currently there is only OOMMF freely available) how to tell the package what geometry you would like to use, and how small your simulation cells should be.

In the case of finite element calculations, you need to create a finite element mesh (see *Finite element mesh generation*).

11.2 What is better: finite differences or finite elements?

This depends on what you want to simulate. Here are some points to consider.

- Finite difference simulations are best when the geometry you simulate is of rectangular shape (i.e. a cube, a beam, a geometry composed of such objects, a T profile, etc). In these situations, the finite element discretisation of the geometry will not yield any advantage. (Assuming that the finite difference grid is aligned with the edges in the geometry.)
- Finite difference simulations need generally less computer memory (RAM). This is in particular the case if you simulate geometries with a big surface (such as thin films). See *Memory requirements of boundary element matrix* for a description of the memory requirements of the hybrid finite element/boundary element simulations (both Nmag and Magpar are in this category).

  If this turns out to be a problem for you, we suggest to read the section *Compression of the Boundary Element Matrix using HLib*.
- Finite element simulations are best suited to describe geometries with some amount of curvature, or angles other than 90 degrees. For such simulations, there is an error associated with the staircase discretisation that finite difference approaches have to use. This error is very much reduced when using finite elements.

  (We state for completeness that there are techniques to reduce the staircase effect in finite difference simulations but these are currently not available in open source micromagnetic simulation code.)
- For finite element simulations, the user has to create a finite element mesh. This requires some practice (mostly to get used to a meshing package), and in practice will take a significant amount of the time required to set up a finite element simulation.

11.3 What size of the cells (FD) and tetrahedra (FE) should I choose?

There are several things to consider:

- the smaller the cells or tetrahedra, the more accurate the simulation results.
- the smaller the cells or tetrahedra, the more cells and tetrahedra are required to describe a geometry. Memory requirements and execution time increase with the number of cells and tetrahedra. In practice this will limit the size of the system that can be simulated.
- the discretisation length (this is the edge length of the cells or the tetrahedra) should be *much smaller than the exchange length*. The reason for this is that in the derivation of the micromagnetic (Brown's) equations, one assumes that the magnetisation changes little in space (there is a Taylor expansion for the exchange interaction). Therefore, we need to choose a discretisation length so that the direction of the magnetisation vectors varies little from one site (cell center in FD, node of tetrahedron in FE) to the next. The difference of the magnetisation vector is sometimes referred to as the 'spin angle': a spin angle of 0 degrees, means that the magnetisation at neighbouring sites points in the same direction, whereas a spin angle of 180 degrees would mean that they point in exactly opposite directions.

  How much variation is acceptable, i.e. how big is the spin angle allowed to be? It depends on the accuracy required. Some general guidelines from M. Donahue [in email to H. Fangohr on 26 March 2002 referring to OOMMF] which we fully endorse :

`[Begin quote M. Donahue]`

- if the spin angle is approaching 180 degrees, then the results are completely bogus.
- over 90 degrees the results are highly questionable.
- Under 30 degrees the results are probably reliable.

`[end quote]`

It is *absolutely vital* that the spin angle does not become excessive if the simulation results are to be trusted. (It is probably the most common error in micromagnetics: one would like to simulate a large geometry, thus one has to choose the discretisation length large to get any results within reasonable time. However, the results are often completely useless if the spin angle becomes too large).

Because this is such an important issue, OOMMF – for example – provides `Max Spin Ang` data in its `odt` data table file (for the current configuration, the last stage, and the overall run). Nmag has a columns `maxangle_m_X` in the *Data files (.ndt)* file that provide this information (where `X` is the name of the magnetic material).

You will probably find that often a discretisation length of half the *Exchange length* or even about the *Exchange length* is chosen. If the spin angle stays sufficiently low during the whole simulation (including intermediate non-equilibrium configurations), then this may be acceptable.

The ultimate test (recommended by – among others – M. Donahue and the nmag team) is the following:

- cell size dependence test

  The best way to check whether the cell size has been chosen small enough, is to perform a series of simulations with increasing cell size. Suppose we are simulating Permalloy (Ni80Fe20 with Ms=8e5 A/m, A=1.3e-11) and the *Exchange length* l_1 is about 5nm. Suppose further we would like to use a cell size of 5nm for our simulations.

  We should then carry out the same simulation with smaller cell sizes, for example, 4nm, 3nm, 2nm, 1nm. Now we need to study (typically plot) some (scalar) entity of the simulation (such as the coercive field, or the remanence magnetisation) as a function of the cell size.

  Ideally, this entity should converge towards a constant value when we reduce the simulation cell size below a critical cell size. This critical cell size is the maximum cell size that should be used to carry out the simulations.

  Be aware that (i) it is often nearly impossible to carry out these simulations at smaller cell sizes [because of a lack of computational power] and (ii) this method is not 100% fool proof [the observed entity may appear to converge towards a constant but actually start changing again if the cell size is reduced even further].

  One should therefore treat the suggestions made above as advise on good practice, but never solely rely on this. Critical examination of the validity of simulation results is a fundamental part of any simulation work.

In summary, it is vital to keep the maximum spin angle small to obtain accurate results. One should always (!) check the value of the spin angle in the data files. One should also carry out a series of simulations where the spin angle is reduced from one simulation to the next while keeping all other parameters and the geometry the same. This should reveal any changes in the results that depend on the discretisation length.

11.3.1 Exchange length

There is sometimes confusion about what equation should be used to compute the exchange length. In this document, we refer to this equation for soft materials (where the demagnetisation energy is driving domain wall formation)

$$l_1 = \sqrt{\frac{2A}{\mu_0 M_s^2}}$$

and this equation for hard materials (with uniaxial pinning) where the crystal anisotropy governs domain wall lengths

$$l_2 = \sqrt{\frac{A}{K_1}}$$

If in doubt which of the two equations is the right one, compute both l_1 and l_2 and choose the minimum length as the relevant exchange length for this system.

11.3.2 Further reading

Micheal Donahue and co-workers have published a couple of papers on the effect of cell size on vortex mobility:

- M. J. Donahue and R. D. McMichael, Physica B, 233, 272-278 (1997)
- M. J. Donahue and D. G. Porter, Physica B, 343, 177-183 (2004)

and one which included a section on discretisation-induced Neel wall collapse

- M. J. Donahue, Journal of Applied Physics, 83, 6491-6493 (1998)

11.4 Micromagnetic packages

The following micromagnetic simulation packages are freely available on the internet:

- OOMMF (finite differences)
- Magpar (finite elements)
- nmag (finite elements)

These are general purpose packages. Some other (and partly closed source/commercial packages) are listed at http://math.nist.gov/oommf/otherlinks.html.

11.5 Summary

The most important points in short:

- choose a small discretisation length so that the spin angle stays well below 30 degrees.
- if you want to simulate thin films (or other geometries with a lot of surface [nodes]), with finite elements, consider how much memory you would need for the boundary element matrix (best to do this before you start creating the mesh).

CHAPTER

TWELVE

ACKNOWLEDGEMENTS

This work has been financially supported by the Engineering and Physical Science Research Council (EPSRC) (GR/T09156/01,EP/E0400631/1) in the United Kingdom, through funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under Grant Agreement no 233552, and by the University of Southampton.

We thank Thomas Schrefl, Wyn Williams, Michael Donahue, Richard Boardman and Jürgen Zimmermann for useful discussion that have supported the development of this tool.

Further acknowledgements go to the Magpar software and its main author Werner Scholz. Magpar has provided a finite element implementation of micromagnetics that has proved very useful in the development of nmag. Special thanks to Werner Scholz who has discussed various numerical problems with the nmag team in great depth.

We further thank the beta users for their helpful feedback, in particular Michael Martens, David Vokoun, Niels Wiese, Gabriel David Chaves O'Flynn and David Gonzales at the very early stages of the project.