# A NOTE ON UBERMAG TUTORIAL

CHANGJIAN XIE

Significance:

(1)

Questions:

(1) The most questions are seen at `https://github.com/ubermag/workshop/issues`.

## 1. INTRODUCTION

The series talk for micromagnetics can be found in `https://www.spintalks.org/tutorials`. The part of `Ubermag`: Instructor: Dr. Marijan Beg (University of Southampton). A repository for this workshop has been set up at `https://github.com/ubermag/workshop`. we can go to `https://hub.mybinder.turing.ac.uk/user/ubermag-workshop-wrjjytqo/notebooks/tutorials/index.ipynb` to find the tutorial. The installation of Ubermag can visit `https://www.youtube.com/channel/UC7MSqVQSMFV42R1jAYmKGLg`. Ubermag is python package and it can be installed in many different ways (pip, miniconda, anaconda-navigator, docker, ...), here we recommend installation using miniconda.

(1) Install Miniconda;
(2) `conda install --channel conda-forge ubermag`.

The detailed installation instructions is demonstrated here.

(1) Ubermag installation on Windows using miniconda: we should goto `https://docs.conda.io/en/latest/miniconda.html` to download the required miniconda in your PC. We can choose `python 3` and then install that version of miniconda. We open miniconda prompt. To create environment at the beginning, `create -n ubermag python=3.8`, when the environment created, `conda activate ubermag`, we can change the prompt of `base` into `ubermag`. We can type `conda install --channel conda-forge ubermag`, five minutes later, the ubermag would be installed. Now we wanna run some tests. We can do `mkdir ubermag-test`, `cd ubermag-test`. In this directory, we run the test since it will produce some `log` file in your system. To test ubermag, type `python -c "import ubermag; ubermag.test()"`. We can goto `jupyter notebook &`, that's it.
(2) Ubermag installation on Linux (or Unix) using miniconda: The first thing is still to download the miniconda and install it by `bash Miniconda3*.sh`, type `conda create -n ubermag python=3`, `conda activate ubermag`, Now do the installation `conda install --channel conda-forge ubermag`. we're gonna do the test, `mkdir ubermag-test`, `cd ubermag-test`, `python -c "import ubermag; ubermag.test()"`. start `jupyter notebook &`.

(3) Ubermag installation on MacOS using miniconda: The fist thing is still download and install the miniconda by `bash Miniconda*.sh` and follow the instructions. `conda create -n ubermag python=3`, `conda activate ubermag`, `conda install --channel conda-forge ubermag`, `python -c "import ubermag; ubermag.test()"`. start `jupyter notebook &`.

The series of talk contains three sessions

(1) Session 1 (video: https://app.vidgrid.com/view/lkmoiGhTlA8z/?sr=9LObtB): 12-1:30 pm (ET) Thursday, June 18. Introduction to micromagnetics and Ubermag;
(2) Session 2: 12-1:30 pm (ET) Thursday, June 25. Micromagnetic models and drivers in Ubermag;
(3) Session 3: 12-1:30 pm (ET) Thursday, July 2. Data analysis and visualisation.

Some goal would be made:

(1) Set up and run a micromagnetic simulation in ubermag;
(2) Analyze and visualize data;
(3) Understand the documentation and learn more advanced topics on your own.

1.1. **Seesion 1.** Due to the limited access to GPU machines, we are going to be focusing on OOMMF. By installing ubermag, OOMMF is going to be installed automatically. ubermag interface is same for both OOMMF and mumax3. `mumax3c` is currently available from the development branch, and we expect to demonstrate `mumax3c` in the last session. The current situation makes it a bit difficult to make a release.

The aim of the workshop is to gain some basic understanding of micromagnetics and to be able to run useful simulations in the end. One has known that micromagnetics is a field of physics dealing with the prediction of magnetic behaviours at sub-micrometer length scales from Wiki. Magnetization field in continuum approximation, magnetization is considered to be a continuous vector field. Magnetization $\boldsymbol{m}(\boldsymbol{r}, t)$ is a function of both space $\boldsymbol{r}$ and time $t$ which indicates that

$$\boldsymbol{M} = \boldsymbol{M}(\boldsymbol{r}, t).$$

In micromagnetics, there are some assumptions here.

(1) magnetization field $\boldsymbol{M}(\boldsymbol{r}, t)$ is differentiable (continuous and slowly changing) with respect to both space $\boldsymbol{r}$ and time $t$.
(2) The magnetization field norm is constant (time invariant). Constant norm $\boldsymbol{M}$ is represented by saturation magnetization $M_s = |\boldsymbol{M}| = \text{const}$. Very often, magnetization is represented by a normalized magnetization field $\boldsymbol{m}(\boldsymbol{r}, t) = \boldsymbol{M}(\boldsymbol{r}, t)/M_s$, and $|\boldsymbol{m}| = m_x^2 + m_y^2 + m_z^2 = 1$.

As for discretization, in order to solve the magnetization field numerically, we have to divide it into smaller "chunks". There are two main ways how we can discretize the field: finite-differences (such as oommf, $\mu$mag, fidimag, Nmag) and finite-elements (such as femag). The discretization must be large enough to ignore the crystal structure of the material (the continuum approximation), and small enough to spatially resolve different magnetization configurations (like domain walls, vortices, or skyrmions). We discretize the whole magnetic sample into cells, we have the magnetization $\boldsymbol{M} = \frac{\sum_{i \in V} \mu_i}{V}$ at each cell with $\mu$ is the magnetic moments. In

finite-differences, magnetization field $\boldsymbol{M}$ is discretized (at each time step) so that a single vector is assigned to each discretization cell. $\boldsymbol{M}(\boldsymbol{r}_i, t_i) = (M_x, M_y, M_z)$. Now Let's look at a little bit energy equation (Hamiltonian) which is a mapping of magnetization field $\boldsymbol{m} = \boldsymbol{m}(\boldsymbol{r}, t)$ to energy density (scalar) field:

$$\mathcal{W}(\boldsymbol{m}) = \mathcal{W}_1(\boldsymbol{m}) + \mathcal{W}_2(\boldsymbol{m}) + \cdots = \sum_i \mathcal{W}_i(\boldsymbol{m}). \tag{1}$$

By integrating $\mathcal{W}(\boldsymbol{m})$ over the entire sample volume $V$, the energy functional is

$$E[\boldsymbol{m}] = \int_V \mathcal{W}(\boldsymbol{m})\, dV. \tag{2}$$

The process is presented in Figure 1. Now let's look at all the energy term:

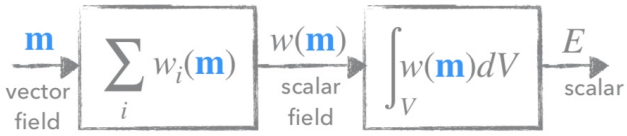



FIGURE 1. Illustration of energy computation

(I) The Zeeman energy (parameter: $\boldsymbol{H}$). It aligns $\boldsymbol{m}$ to $\boldsymbol{H}$ of scale $A/m$ which is presented in Figure 2. The energy would be $\mathcal{W}_z = -\boldsymbol{M} \cdot \boldsymbol{B} = -\mu_0 M_s \boldsymbol{m} \cdot \boldsymbol{H}$ with $\boldsymbol{B} = \mu_0 \boldsymbol{H}$, $\boldsymbol{M} = M_s \boldsymbol{m}$. The parameter in ubermag is actually $\boldsymbol{H}$ and not $\boldsymbol{B}$.

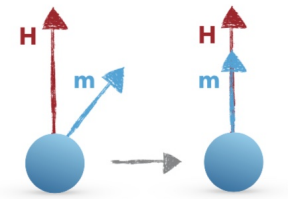



FIGURE 2. Illustration of Zeeman energy which aligns $\boldsymbol{m}$ to $\boldsymbol{H}$.

(II) The uniaxial anisotropy energy (parameter: $K$ and $\boldsymbol{u}$). This interaction aligns $\boldsymbol{m}$ along the vector $\boldsymbol{u}$ or perpendicular to $\boldsymbol{u}$. The energy shown in Figure 3 would be

$$\begin{aligned} \mathcal{W}_{\text{ua}} &= -K(\boldsymbol{m} \cdot \boldsymbol{u})^2 = -K\cos^2(\theta) \\ &= -K(1 - \sin^2(\theta)) = -K + K\sin^2(\theta), \end{aligned} \tag{3}$$

with $|\boldsymbol{m}| = 1$, $|\boldsymbol{u}| = 1$, $K$ of scale $J/m^3$. Usually, we ignore the constant $-K$, we can rewrite (3) into $\mathcal{W}_{\text{ua}} = K\sin^2(\theta)$. Note that $\boldsymbol{m}$ and $\boldsymbol{u}$ are colinear for $K > 0$ or perpendicular each other for $K < 0$, such that this energy to be minimized. There are several mathematically equivalent ways of writing uniaxial anisotropy. Changing on the sign of $K$, changes the "character" of uniaxial anisotropy. We can choose some $K$ such that it's easy-axis or easy-plane. Easy axis uniaxial anisotropy (parameter $K > 0$,

$\boldsymbol{u}$) aligns $\boldsymbol{m}$ parallel or antiparallel to $\boldsymbol{u}$. The magnetic moment would tend to be along the axis. Easy plane uniaxial anisotropy (parameter $K < 0$, $\boldsymbol{u}$) aligns $\boldsymbol{m}$ perpendicular to $\boldsymbol{u}$. The magnetic moment would tend to be in plane. The other one is cubic uniaxial anisotropy (parameters: $K$ $(J/m^3)$, $\boldsymbol{u}_1$, $\boldsymbol{u}_2$ and $\boldsymbol{u}_3$) shown in Figure 4 which is

$$\mathcal{W}_{\text{ca}} = -K(m_1^2 m_2^2 + m_2^2 m_3^2 + m_1^2 m_3^2),$$

with $|\boldsymbol{m}| = 1$, $|\boldsymbol{u}| = 1$,and $m_i = \boldsymbol{m} \cdot \boldsymbol{u}_i$ which is the projection onto $\boldsymbol{u}_i$ of $\boldsymbol{m}$. We should be careful about $|\boldsymbol{u}_i| = 1$ and $\boldsymbol{u}_3 = \boldsymbol{u}_1 \times \boldsymbol{u}_2$.

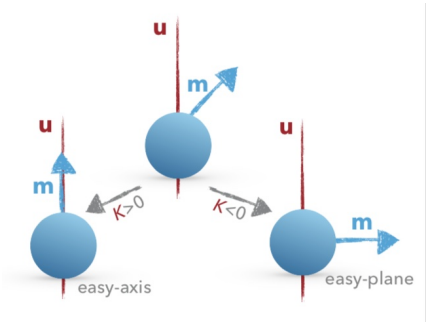



FIGURE 3. Illustration of uniaxial anisotropy energy which aligns $\boldsymbol{m}$ along the vector $\boldsymbol{u}$ or perpendicular to $\boldsymbol{u}$.

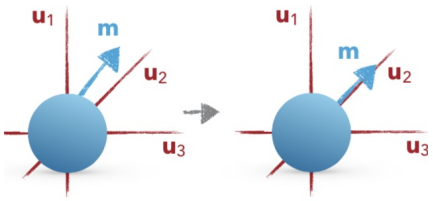



FIGURE 4. Illustration of cubic anisotropy energy.

(III) The (ferromagnetic) exchange energy (parameter: $A$ $(J/m)$). This interaction aligns all magnetic moments (in $\boldsymbol{m}$) parallel to each other (the direction we do not know). The energy shown in Figure 5 would be

(4) $$\mathcal{W}_{\text{ex}} = A\boldsymbol{m} \cdot \nabla^2 \boldsymbol{m} = A[(\nabla m_x)^2 + (\nabla m_y)^2 + (\nabla m_z)^2] = A(\nabla \boldsymbol{m})^2.$$

Note that using $A < 0$ (antiferromagnetic exchange) in micromagnetics is not justified. The Zeeman energy or uniaxial anisotropy energy are both local energy. The exchange energy is nonlocal and short-range.

(IV) (Atomistic level) T.O. formulation of Dzyaloshinskii-Moriya interaction (DMI) (parameter: $\boldsymbol{D}$ $(J/m^2)$). This interaction shown in Figure 6 aligns neighbouring magnetic moments (in $\boldsymbol{m}$) perpendicular to each other and perpendicular to vector $\boldsymbol{D}$ depends on material property. This energy would be

$$\mathcal{W}_{ij}^{\text{dmi}} = (\pm)\boldsymbol{D} \cdot (\boldsymbol{S}_i \times \boldsymbol{S}_j),$$

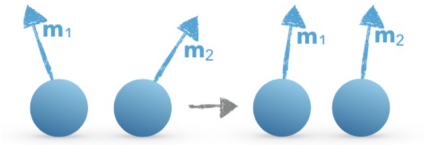



FIGURE 5. Illustration of exchange energy.

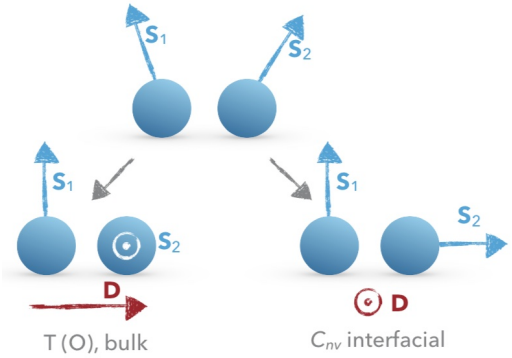



FIGURE 6. Illustration of DMI with spins $\boldsymbol{S}$.

here we should be careful about the sign. There is no clear convention.

From the macroscopic view (continuous limit), we have the DMI with magnetic moments shown in Figure 7 which is

$$\mathcal{W}_{\text{dmi}} = (\pm)\boldsymbol{D}\boldsymbol{m} \cdot (\nabla \times \boldsymbol{m}),$$

with $\boldsymbol{D} = D\boldsymbol{r}_{ij}$, $D$ is a scalar.

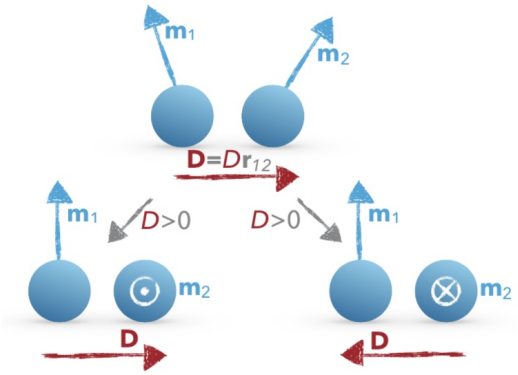



FIGURE 7. Illustration of DMI with magnetization $\boldsymbol{m}$.

The other is $C_{NV}$ formulation of Dzyaloshinskii-Moriya interaction (DMI) (same story, but different form), (parameter: $\boldsymbol{D}$ $(J/m^2)$), which also aligns

neighbouring magnetic moments (in $\boldsymbol{m}$) perpendicular to each other. The $C_{NV}$ formulation of DMI shown in Figure 8 is given by

$$\mathcal{W}_{\text{dmi}} = (\pm)\boldsymbol{D}(\boldsymbol{m} \cdot \nabla m_z - m_z \nabla \cdot \boldsymbol{m}),$$

where $\boldsymbol{D} = D(\boldsymbol{r}_{ij}) \times \hat{z}$.

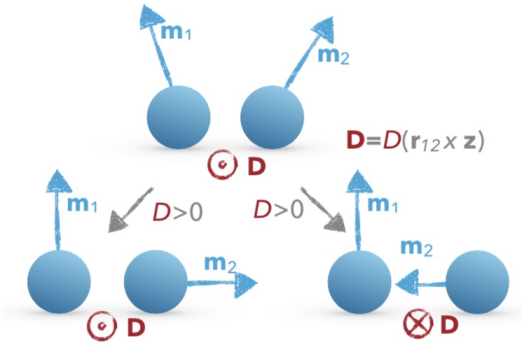



FIGURE 8. Illustration of DMI with magnetization $\boldsymbol{m}$.

(V) The demagnetization shown in Figure 9 which is a long range interaction. It is compute from $\boldsymbol{m}$ and no parameter is required. Each magnetic moment (in $\boldsymbol{m}$) "feels" the total magnetic field of all surrounding moments. However, It is computationally expensive. We will put more details in the next sessions.

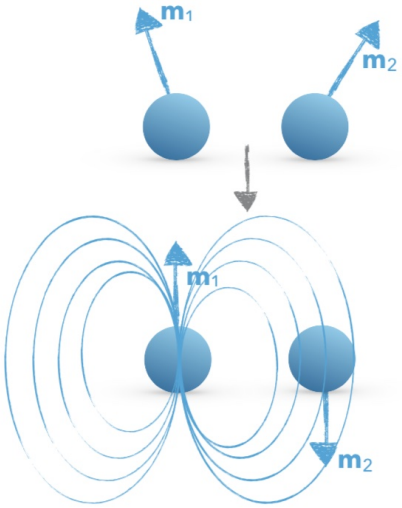



FIGURE 9. Illustration of demagnetization.

If we consider the coupling of exchange and Zeeman energy, All magnetic moments parallel to each other and parallel to H. The interesting

thing when we combine the exchange energy and DMI, all magnetic moments do not parallel or perpendicular to each other. The energies have to compete and reach a compromise. Now let's have a look at a complicated case in a 2D sample by considering the Zeeman, anisotropy, exchange, and DMI. Energies have to compete and reach a compromise. We initialize the configuration and relax that. This would happen the skyrmion with $T$ class DMI (bulk DMI).

The dynamics equation tells us how magnetization $\boldsymbol{m}$ wants to change in order to minimize its energy which is given by

$$\frac{d\boldsymbol{m}}{dt} = f_1(\boldsymbol{m}, \boldsymbol{H}_{\text{eff}}) + f_2(\boldsymbol{m}, \boldsymbol{H}_{\text{eff}}) + \cdots = \sum_i f_i(\boldsymbol{m}, \boldsymbol{H}_{\text{eff}}).$$

Effective field is computed as the first variational derivative of energy density:

$$\boldsymbol{H}_{\text{eff}} = -\frac{1}{\mu_0 M_s} \frac{\delta \mathcal{W}(\boldsymbol{m})}{\delta \boldsymbol{m}}.$$

The block process is shown in Figure 10.

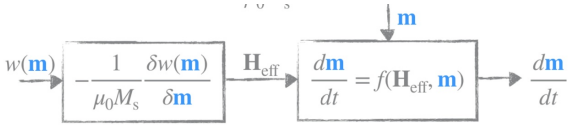



FIGURE 10. Illustration of calculating the dynamics.

Here, we have two terms in the dynamics equation. One is the precession term (parameter: $\gamma_0$) which makes $\boldsymbol{m}$ precess around $\boldsymbol{H}_{\text{eff}}$. The equation with gyrmmagnetic term would be

$$\frac{d\boldsymbol{m}}{dt} = -\gamma_0 \boldsymbol{m} \times \boldsymbol{H}_{\text{eff}},$$

where $\gamma_0 = \mu_0 \gamma$. The parameter in Ubermag is $\gamma_0$, not $\gamma$. The second term is damping term (parameter: $\alpha$) which makes $\boldsymbol{m}$ align with $\boldsymbol{H}_{\text{eff}}$. The equation would be

$$\frac{d\boldsymbol{m}}{dt} = \alpha \left( \boldsymbol{m} \times \frac{d\boldsymbol{m}}{dt} \right)$$

with damping constant $\alpha > 0$. The effect of two terms are shown in Figure 11.

If we combine two terms together, we get the Landau-Lifshitz-Gilbert equation would be

$$\frac{d\boldsymbol{m}}{dt} = -\gamma_0 \boldsymbol{m} \times \boldsymbol{H}_{\text{eff}} + \alpha \left( \boldsymbol{m} \times \frac{d\boldsymbol{m}}{dt} \right), \tag{5}$$

or

$$\frac{d\boldsymbol{m}}{dt} = -\frac{\gamma_0}{1+\alpha^2} \boldsymbol{m} \times \boldsymbol{H}_{\text{eff}} - \frac{\gamma_0 \alpha}{1+\alpha^2} \boldsymbol{m} \times (\boldsymbol{m} \times \boldsymbol{H}_{\text{eff}}). \tag{6}$$

In ubermag, we use the form of (6). Now we have some building blocks in ubermag for micromagnetic simulator in Figure 12.

In this session, we are going to familiarize ourselves with micromagnetics, Python, and Jupyter and we are going to spend most of the time looking at slides. Slides can be found in slides directory in the workshop repository. After slides (and a short

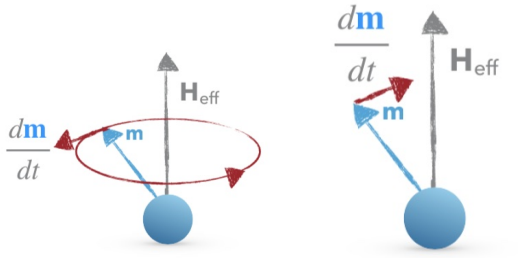



FIGURE 11. Illustration of precession term and damping term.

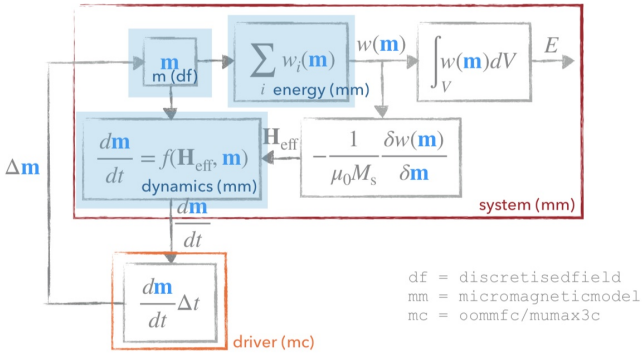



FIGURE 12. Illustration of micromagnetic simulator. This beginning from magnetization $\boldsymbol{m}$.

break), we are going to write our first ubermag simulation and have a look at some Python basics. We believe we do not need much Python knowledge in order to run ubermag. However, more you feel confident writing Python, more benefits you can get from ubermag. There are many resources online you can choose from. We can recommend the online one by Hans Fangohr https://fangohr.github.io/teaching/python/book.html. The binder of this talk is in https://hub-binder.mybinder.ovh/user/ubermag-workshop-nwrs1nwu/notebooks/tutorials/index.ipynb. Now we go to the first ubermag simulation https://hub-binder.mybinder.ovh/user/ubermag-workshop-nwrs1nwu/notebooks/tutorials/first-ubermag-simulation.ipynb. The main goal of this tutorial is to have a quick look at how a simple ubermag simulation inside Jupyter looks like and to make ourselves comfortable with Jupyter. We are going to try to guess what the meaning of Python commands in code cells is, and eventually try to identify the skeleton of ubermag simulation. Before we specify and run the simulation, we have to import Ubermag modules we intend to use. For defining micromagnetic models, we need to import `micromagneticmodel` and for defining finite-diference fields, we are going to import `discretisedfield`.

```
import micromagneticmodel as mm
import discretisedfield as df
```

The main object in ubermag is `mm.System`. In order to define the micromagnetic system we want to simulate, we have to specify:

(a) Energy equation;
(b) Dynamics equation (optional);
(c) Initial magentisation state configuration.

```
system = mm.System(name='first_ubermag_simulation')
```

We have to define the energy of the system. The energy equation for the first Ubermag simulaton is very simple and contains only the following energy terms:

(a) Exchange $A = 1\ pJ/m$;
(b) Zeeman $\boldsymbol{H} = (5 \times 10^6, 0, 0)\ A/m$;
(c) Demagnetization.

```
A = 1e-12  # exchange energy constant (J/m)
H = (5e6, 0, 0)  # external magnetic field in the x-direction (A/m)
system.energy = mm.Exchange(A=A) + mm.Demag() + mm.Zeeman(H=H)
```

The overview of the energy terms we introduced in Figure 13.

| name | parameters | class | comments |
|---|---|---|---|
| Zeeman | **H** (A/m) | `mm.Zeeman` | parameter is **H** not **B** |
| uniaxial anisotropy | $K$ (J/m³), **u** | `mm.UniaxialAnisotropy` | sign of $K$, $\|\mathbf{u}\|$=1 |
| exchange | $A$ (J/m) | `mm.Exchange` | $A$<0 not justified |
| DMI (T, O) | $D$ (J/m²), crystal class | `mm.DMI` | sign of $D$ |
| DMI ($C_{nv}$) | $D$ (J/m²), crystal class | `mm.DMI` | sign of $D$ |
| cubic anisotropy | $K$ (J/m³), $\mathbf{u}_1$, $\mathbf{u}_2$ | `mm.CubicAnisotropy` | sign of $K$, $\|\mathbf{u}\|$=1 |
| demagnetisation | None | `mm.Demag` | None |

FIGURE 13. Overview of energy terms.

We then define the dynamics equation which contains only precession and damping terms.

```
gamma0 = 2.211e5  # gyrotropic ratio parameter (m/As)
alpha = 0.2  # Gilbert damping
system.dynamics = mm.Precession(gamma0=gamma0) + mm.Damping(alpha=
                                alpha)
```

In terms of initial magnetization, we choose to simulate a cube with 50 nm edge length and discretise it into 10 cells in each direction (1000 in total). We initialise the system in positive $y$-direction, i.e. $\mathbf{m} = (0, 1, 0)$, which is different from the equlibrium state we expect for the external magnetic field applied in $x$-direction. For its norm (saturation magnetisation), we choose $M_\mathrm{s} = 8 \times 10^6$ A/m.

```
L = 50e-9  # cubic sample edge length (m)
region = df.Region(p1=(0, 0, 0), p2=(L, L, L))
region.k3d()
```

which gives the whole domain shown in Figure 14.

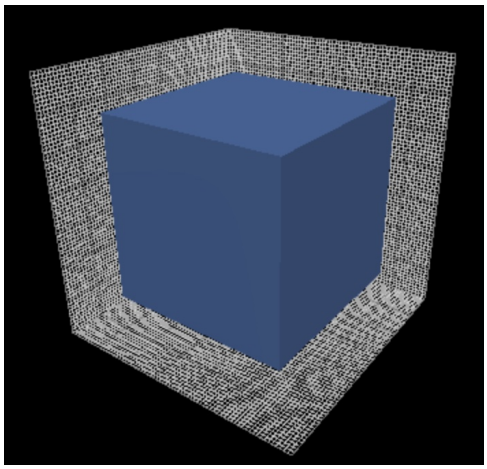


FIGURE 14. Region profile.

```
mesh = df.Mesh(region=region, n=(10, 10, 10)) # n parameter is the
                                    number of discretization cells
mesh.k3d()
# define the magnetization
Ms = 8e6  # saturation magnetization (A/m)
system.m = df.Field(mesh, dim=3, value=(0, 1, 0), norm=Ms)
```

We have defined the system object and now we can do some inspection to make sure we did not make a mistake. First, we are going to check the energy equation:

```
system.energy
```

This gives us a human-readable equation, which is actually the sum of terms we defined earlier. The same story for the dynamics. Similarly, we can check the system's dynamics equation (if we have defined it).

```
system.dynamics
```

Inspecting magnetization is sligtly more complicated because there are many things we can ask the magnetization for. Let us have a look at a few basic ones and the rest of them, we are going to explore in later sessions.

The region we defined is:

```
system.m.mesh.region.k3d()
```

Representation of the mesh:

```
system.m.mesh.k3d()
```

We give a quick 2D plot of the magnetization in the z-slice:

```
system.m.plane('z').mpl() # by default, slice cut from the center
system.m.plane(z=25e-9).mpl()
```

shown in Figure 15. Again, we can plot 3D magnetization

```
system.m.plane('z').k3d_vector(head_size=10)
```

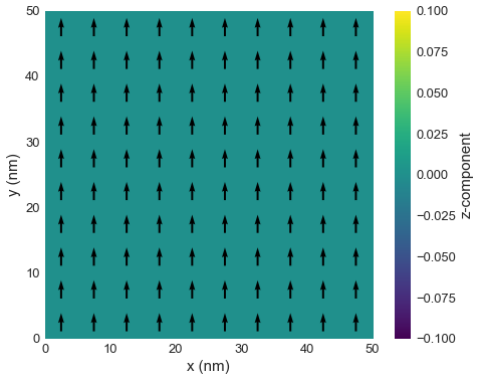



FIGURE 15. 2D plot of the magnetization in the z-slice.

shown in Figure 16.

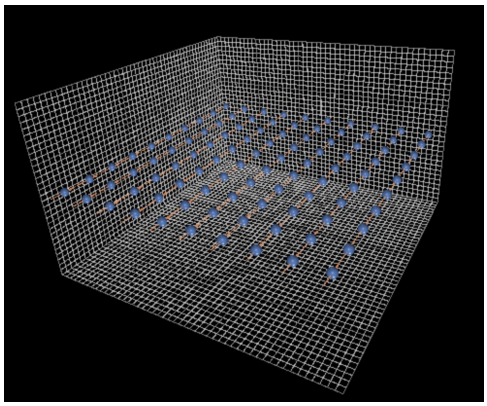


FIGURE 16. 3D plot of the magnetization.

From now on, we define the energy equation, the dynamics equation and initial magnetization. After the system object is created, we can minimize its energy (relax it) using the Minimization Driver (`MinDriver`). At this point, we choose the calculator we want to use. During this workshop, we are going to use OOMMF. Therefore, we import OOMMF calculator - `oommfc`.

```
import oommfc as mc # calculator is oommfc for oommf, mumax3c for
                                    mumax3

md = mc.MinDriver()
md.drive(system)
```

There are two main types of drivers and they are a part of specific micromagnetic calculator:

(a) Minimisation driver: iterates until $\boldsymbol{m} \times \boldsymbol{H}_{\text{eff}} = 0$ and using `md = oc.MinDriver()`
(b) Time driver: iterates until the maximum time is reached and using `td = oc.TimeDriver()`. After the driver is defined, it is applied to the system object using drive method `td.drive(system)`.

The system is now relaxed and its previous magnetization is now replaced with the new one, and we can plot its slice and compute its average magnetization

```
system.m.plane('z', n=(10, 10)).mpl() # now m is the relaxed
                                      magnetization
system.m.plane('z', n=(5, 5)).mpl()
```

shown in Figure 17. Note that the Zeeman field defined previously is along positive

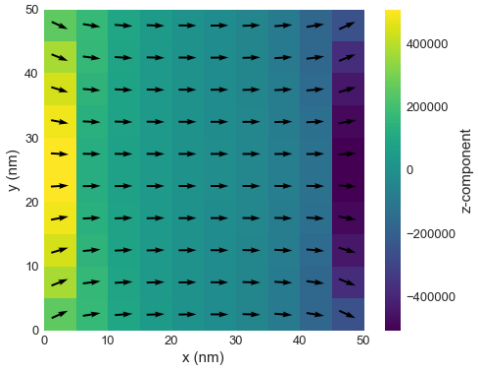



FIGURE 17. 2D plot of the relaxed magnetization.

$x$-axis.

Lots information we can see using

```
system.table.data
```

which gives more information about the relaxed data (`odt` file). Data analysis for this table will be given later.

We can also plot the 3D view shown in Figure 18. We get the averaged magne-

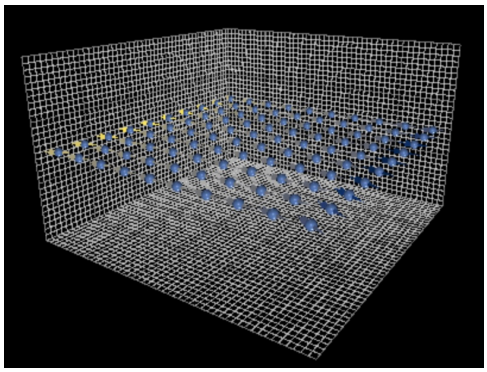


FIGURE 18. 3D plot of the relaxed magnetization.

tization by

```
system.m.average
```

is that $(7860158.235947054, -0.007357321005314588, 4.610046744346619e-11)$. If we wanna get the normalized magnetization, we can use

```
system.m.orientation.average
```

is that $(0.9825197794933807, -9.196651318821303e-10, 7.049916206369744e-18)$.

We can see that the magnetization is aligned along the $x$-direction, as expected having in mind we applied the external magnetic field in that direction. As for Python basics, we omit this part in this note. We can find it in https://notebooks.gesis.org/binder/jupyter/user/ubermag-workshop-awm1pkkb/notebooks/tutorials/python-basics.ipynb.

## 2. Extra useful questions

(1) Where does the damping term in LLG equation come from?
   (a) Landau, L. & Lifshits, E. On the theory of the dispersion of magnetic permeability in ferromagnetic bodies. Phys. Zeitsch. Der Sow., 8, 153–169 (1935). https://doi.org/10.1016/B978-0-08-036364-6.50008-9.
   (b) Gilbert, T. L. A phenomenological theory of damping in ferromagnetic materials. IEEE Transactions on Magnetics, 40(6), 3443–3449 (2014). https://doi.org/10.1109/TMAG.2004.836740.

(2) How to do calculations with complex numbers in Python? The Python library is `cmath`. Please refer to Python documentation https://docs.python.org/3.8/library/cmath.html.

### 2.1. session 2.

This session happens `12-1:30 pm (ET) Thursday, June 25.` which contains topics on micromagnetic models and drivers in Ubermag.

In the last session, we introduced some very basic concepts of Ubermag simulations, had a look at some basic Python syntax, and familiarized ourselves with Jupyter environment. In session 2, we are going to have a look into more details of Ubermag simulations, so we can start simulating some real-world problems. In the first half of the session, we are going to analyze the skeleton of Ubermag simulation and quickly go through the three main concepts (magnetization field, energy equation, and dynamics equation). In each tutorial, we introduce some data analysis and visualization concepts, which are then going to be the main focus of session 3. We here are going to simulate vortex dynamics, drive domain walls with a spin-polarized current, and have a look at the exercise.

Tutorial and exercises can be run in the cloud using Binder https://hub.mybinder.turing.ac.uk/user/ubermag-workshop-04iwnze4/notebooks/tutorials/index.ipynb. We do not install anything and no files will be created on our machine.

The plan for this session: we build on top of the basics we introduced in session 1. We are gonna focus on spatially varying fields, parameters, energy and dynamics equations, as well as the periodic boundary conditions. In addition, we are gonna start looking at some data analysis and visualization methods. The schematic of outline of session 2 is shown in Figure 19. We here have 9 tutorial:

(1) Magnetization field;
(2) Energy equation;
(3) Dynamics equation;
(4) Vortex dynamics;
(5) Spatially varying parameters 1;

(6) Spatially varying parameters 2;
(7) Periodic boundary conditions;
(8) Current induced domain wall motion;
(9) Exercise.

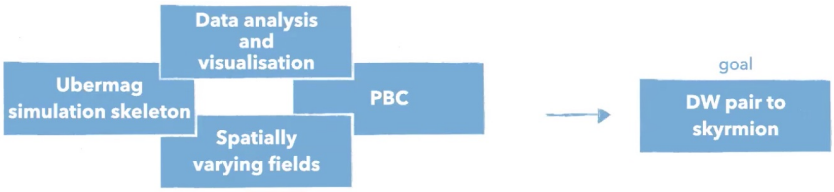



FIGURE 19. Outline of session 2

2.1.1. *Magnetization field.* One of the main objects, which we have to be specify as a part of the micromagnetic system is the magnetization field $\mathbf{M}$. In Ubermag, by defining the magnetization field, the mesh, the geometry of the sample, as well as the magnetization saturation $M_s$ are defined.

The Ubermag package we use to define finite difference regions, meshes, and fields is `discretisedfield` and we have to import it before we use it.

```
import discretisedfield as df  # df is just a shorter name
```

The first thing we want to define is the region. In finite differences, the region on which the mesh and the field are specified is always "cubic". In order to define a mesh, two points $p_1$ and $p_2$ must be passed to `discretisedfield.Region`. Those points correspond to two points between which the region spans. For instance, if we want to define the following region: $x \in [0, 100]$ $nm$, $y \in [-20, 20]$ $nm$, $z \in [0, 10]$ $nm$. Two points we need to pass are: $p_1 = (0\,\text{nm}, -20\,\text{nm}, 0\,\text{nm})$, $p_2 = (100\,\text{nm}, 20\,\text{nm}, 10\,\text{nm})$. In Ubermag, points are defined as length-3 iterables, for instance, tuples. Accordingly, the region we want is:

```
p1 = (0, -20e-9, 0)  # point 1
p2 = (100e-9, 20e-9, 10e-9)  # point 2

region = df.Region(p1=p1, p2=p2)
```

We can plot for help by

```
region.mpl?
```

There are several quantitites we can ask the region object for. Everything is in meters.

```
region.pmin  # minimum coordinate in the region p1
region.pmax  # maximum coordinate in the region p2
region.centre  # the region centre
region.edges  # edge lengths of the region
```

We can visualize the region using either matplotlib `mpl()` or `k3d()`.

```
region.mpl()
region.k3d()
```

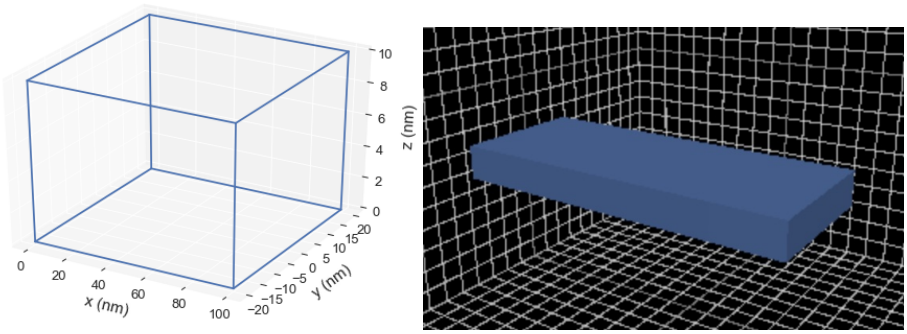



FIGURE 20. Region profile. Left for `mpl`; Right for `k3d`. This just gives us some feelings about the region geometry.

This gives the visualization in Figure 20. After the region is defined, we need to discretize it in order to define a mesh. There are two ways how a region can be discretized - by passing:

(a) Discretization cell size (`cell`), or
(b) Number of discretization cells in all three directions (`n`).

Let us say we want to discretize our region in cells of size $(10\,\text{nm}, 10\,\text{nm}, 10\,\text{nm})$. Then, we pass the region and the cell size to `Mesh` class.

```
cell = (10e-9, 10e-9, 10e-9)  # discretisation cell
mesh = df.Mesh(region=region, cell=cell)  # mesh definition
```

We can then inspect some basic parameters of the mesh.

```
mesh.n  # number of discretisation cells
len(mesh)  # total number of discretisation cells
```

Same story as before, we can visualize the region using either matplotlib `mpl()` or `k3d()`.

```
mesh.mpl()
mesh.k3d()
```

When we have the mesh object created, we can access the region using '.' operator. For instance:

```
mesh.region.centre
```

After we defined the mesh, we can define a finite difference field. For that, we use 'Field' class. We must provide the following 3 mandatory parameters:

(a) mesh (`mesh` as `discretisedfield.Mesh`);
(b) dimension of the value (e.g. `dim=1` for scalar, `dim=3` for vector.)
(c) value of the field (`value`).

For instance, let us define a 3D-vector field (`dim=3`) that is uniform in the $x$-direction `value=(1,0,0)` direction.

```
m = df.Field(mesh, dim=3, value=(1, 0, 0))
```

A simple visualisation of the field in $z$-slice through the middle is shown in Figure 21 by

```
m.plane('z').mpl()  # cut the middle in z-direction
m.plane(z=0).mpl()
```

Similarly, a three-dimensional interactive plot is:

```
m.k3d_vector(head_size=20)
```

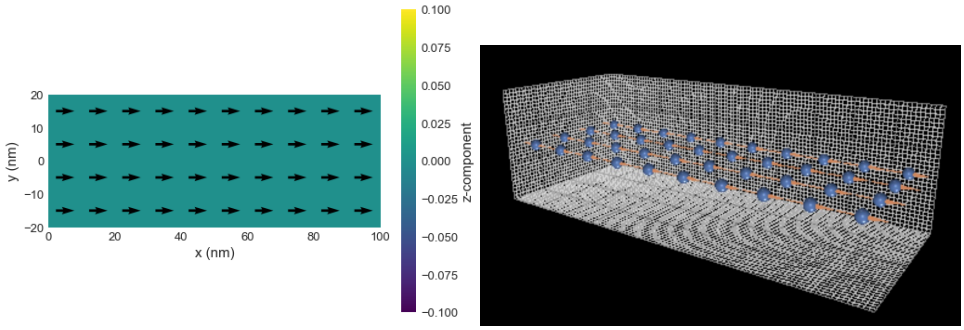



FIGURE 21. Magnetization profile of initialization. Lf for 2D, Rg for 3D.

The average value of the field is:

```
m   # the field objection in 3d: (m.x,m.y,m.z)
m.average # average magnetization
```

We can specify the norm of the field, by passing `norm` parameter. In the case of magnetization, norm corresponds to magnetization saturation $M_s$.

```
Ms = 8e6  # A/m
m = df.Field(mesh, dim=3, value=(0, 0, 1), norm=Ms)   # normalized
                                    magnetization, the value can be
                                    tuple, file, or function
```

Now, if we have a look at the average:

```
m.average # will return (0.0, 0.0, 8000000.0)
```

Now, What do we do for spatially varying field value, when we defined a uniform vector field, we used a tuple 'value=(1, 0, 0)' to define its value. However, we can also pass a callable (e.g. Python function) if we want to define a non-uniform field. This function takes the coordinate of the mesh cell as an input, and returns a value that the field should have at that point. The function must be defined at all points in the mesh.

```
def m_value(point):
  x, y, z = point  # unpack position into individual components
  if y > 0:
    return (1, 0, 0)  # return, not print
  else:
    return (-1, 1, 0)

m = df.Field(mesh, dim=3, value=m_value)
```

Now, we can plot it shown in Figure 22 using `mpl`.

```
m.plane('z').mpl()
```

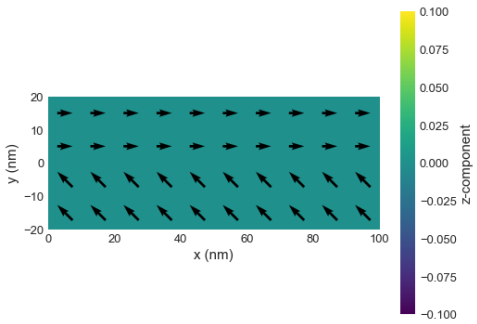



FIGURE 22. Initial magnetization

The field object can be treated as a callable - if we pass a point to the function, it will return the vector value of the field at that location:

```
point = (0, -10e-9, 0)
m(point)  # return  (-1.0, 1.0, 0.0)

point = (0, 10e-9, 0)
m(point)  # return (1.0, 0.0, 0.0)
m(m.mesh.region.centre)
```

If we consider spatially varying $M_s$. By defining a norm of a field, we can specify different geometries. More precisely, we can set $M_\mathrm{s} = 0$ where no magnetic material is present. For instance, let us assume we want to define a sphere of radius $r = 50\,\mathrm{nm}$, disrectized by 10 cells in each direction, and magnetize it in the negative $y$-direction. We plot this magnetization of setting in Figure 23.

```
r = 50e-9
n = (10, 10, 10)
region = df.Region(p1=(-r, -r, -r), p2=(r, r, r))
mesh = df.Mesh(region=region, n=n)

def Ms_value(point):
  x, y, z = point
  if x**2 + y**2 + z**2 < r**2:
    return Ms
  else:
    return 0

m = df.Field(mesh, dim=3, value=(0, -1, 0), norm=Ms_value)

m.plane('z').mpl()
```

We can inspect the defined domain using k3d and inspecting the field's norm.

```
m.norm.k3d_nonzero()
```

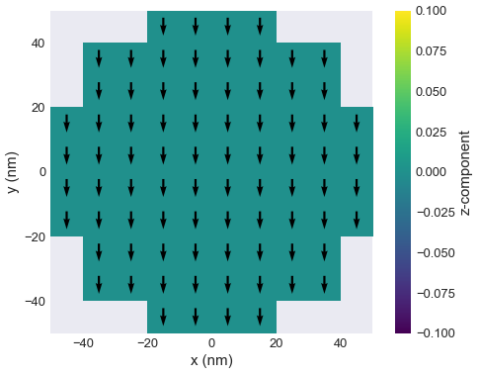



FIGURE 23. Initial magnetization

Let us define a thin-film disk sample of thickness $t = 10\,\text{nm}$ and diameter $d = 120\,\text{nm}$. The saturation magnetization $M_\text{s} = 10^7\text{A/m}$. The disk is centred at the origin (0, 0, 0) and the magnetization is $\mathbf{m} = (1, 0, 0)$ at all mesh points.

```
t = 10e-9  # thickness (m)
d = 120e-9  # diameter (m)
cell = (5e-9, 5e-9, 5e-9)  # discretisation cell size (m)
Ms = 1e7  # saturation magnetisation (A/m)

region = df.Region(p1=(-d/2, -d/2, -t/2), p2=(d/2, d/2, t/2))
mesh = df.Mesh(region=region, cell=cell)

def Ms_value(point):
  x, y, z = point
  if (x**2 + y**2)**0.5 < d/2:
    return Ms
  else:
    return 0

m = df.Field(mesh, dim=3, value=(1, 0, 0), norm=Ms_value)

m.norm.k3d_nonzero()
```

Now, let us say we want to extend the solution from the previous example so that the magnetization is:

$$\mathbf{m} = \begin{cases} (-1, 0, 0) & \text{for } y \leq 0 \\ (1, 1, 1) & \text{for } y > 0 \end{cases}$$

with saturation magnetization $M_\text{s} = 10^7\,\text{A}\,\text{m}^{-1}$. The plot is presented in Figure 24.

```
def m_value(pos):
  x, y, z = pos
  if y <= 0:
    return (-1, 0, 0)
  else:
    return (1, 1, 1)
```

```
m = df.Field(mesh, dim=3, value=m_value, norm=Ms_value)

m.plane('z', n=(15, 15)).mpl()
```

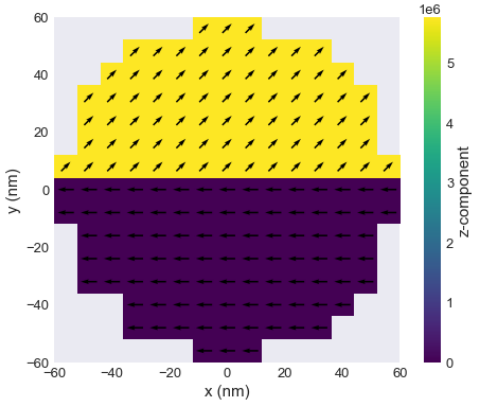



FIGURE 24. Initial magnetization

Here's exercise to go to how to define the random region both $M_s$.

**Exercise 1.** *Define the following sample in Figure 25 with* 10 *nm thickness: The*

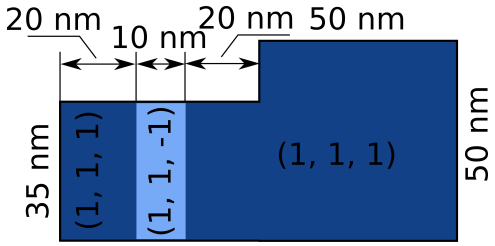



FIGURE 25. Geometry of the sample

*magnetization saturation is* $M_s = 8 \times 10^6\, A\, m^{-1}$ *and the magnetization direction is as shown in the figure.*

*The solution would be given by*

```
cell = (5e-9, 5e-9, 5e-9)  # discretisation cell size (m)
Ms = 8e6  # saturation magnetisation (A/m)

region = df.Region(p1=(0, 0, 0), p2=(100e-9, 50e-9, 10e-9))
mesh = df.Mesh(region=region, cell=cell)

def Ms_value(pos):
  x, y, z = pos
  if x < 50e-9 and y > 35e-9:
    return 0
  else:
    return Ms
```

```
def m_value(pos):
  x, y, z = pos
  if 20e-9 < x <= 30e-9:
    return (1, 1, -1)
  else:
    return (1, 1, 1)

m = df.Field(mesh, dim=3, value=m_value, norm=Ms_value)

m.plane('z').mpl()
```

*The result is shown in Figure 26.*

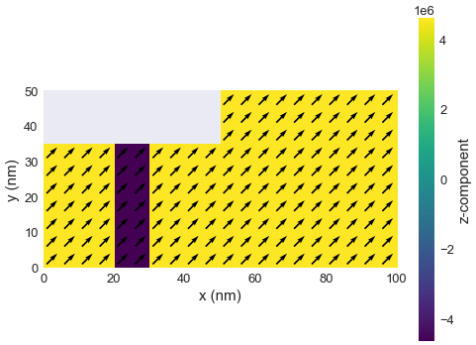



FIGURE 26. Magnetization profile

Now we want to go to tutorial 4.

2.1.2. *Vortex dynamics.* In this tutorial, we want to demonstrate how the system object can be driven using different drivers. In addition, we want to introduce some data analysis tools (e.g. using the `panda` module to get the data table), which we are going to focus on in more details later. As an example, we are going to simulate vortex dynamics. More precisely, we are going to move the vortex core by applying an in-plane external magnetic field and then turn off the field and look at the vortex dynamics.

The sample is a two-dimensional square sample with $d = 100\,\mathrm{nm}$ edge length and $5\,\mathrm{nm}$ thickness. The material is Permalloy.

(a) magnetization saturation: $M_\mathrm{s} = 8 \times 10^5\,\mathrm{A\,m^{-1}}$;
(b) exchange energy constant $A = 13\,\mathrm{pJ\,m^{-1}}$;
(c) gyrotropic ratio $\gamma = 2.211 \times 10^5\,\mathrm{m\,A^{-1}\,s^{-1}}$, and Gilbert damping $\alpha = 0.2$.

Considering the micromagnetics under zero-temperature, in order to obtain the vortex state after relaxation (to get the energy minimum), we have to initialize the magnetization the right way. A state which is similar to the vortex state is:

$$(m_x, m_y, m_z) = (-cy, cx, 10)$$

with $c = 10^9\,\mathrm{m^{-1}}$. The field we are going to apply during the relaxation is $H = 10^4\,\mathrm{A\,m^{-1}}$ in the positive $x$-direction.

We start by importing necessary modules and defining parameters:

```
import discretisedfield as df
import micromagneticmodel as mm
import oommfc as mc

# Geometry
L = 100e-9  # sample edge length (m)
thickness = 5e-9  # sample thickness (nm)

# Material (permalloy) parameters
Ms = 8e5  # saturation magnetisation (A/m)
A = 13e-12  # exchange energy constant (J/m)

# Dynamics (LLG equation) parameters
gamma0 = 2.211e5  # gyromagnetic ratio (m/As)
alpha = 0.2  # Gilbert damping
```

Now, we can start defining our system object:

```
system = mm.System(name='vortex_dynamics')

# Energy equation
system.energy = mm.Exchange(A=A) + mm.Demag() # for simplicity now

# Dynamics equation
system.dynamics = mm.Precession(gamma0=gamma0) + mm.Damping(alpha=
                                    alpha)  # gamma0 can be mm.consts.
                                    gamma0 in default
```

Defining the region and the mesh is relatively straightforward since the sample is cubic, but we have to initialize the magnetization as spatially varying and we need to define a Python function for that.

```
region = df.Region(p1=(-L/2, -L/2, 0), p2=(L/2, L/2, thickness))

mesh = df.Mesh(region=region, cell=(5e-9, 5e-9, 5e-9))

def m_init(point):
  x, y, z = point
  c = 1e9  # (1/m)
  return -c*y, c*x, 10  # should be tuple (-c*y, c*x, 10)

system.m = df.Field(mesh, dim=3, value=m_init, norm=Ms)
```

The system object is now defined and we can investigate some of its properties:

```
system.energy
system.dynamics
system.m.plane('z').mpl()
```

```
md = mc.MinDriver()
md.drive(system)

system.m.plane('z').mpl()
```

We plot the initialized magnetization, and relax the system presented in Figure 27. Now we have a relaxed vortex state, with its core at the centre of the sample. As the next step, we want to add an external magnetic field in the positive $x$-direction and hope to move the vortex core. We get the result shown in Figure 28.

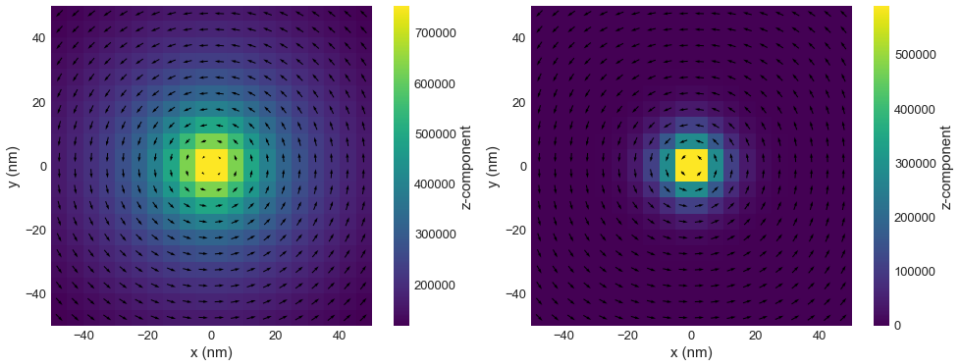



FIGURE 27. Magnetization profile: Lf for initialized magnetization; Rg for the relaxed magnetization.

```
H = (1e4, 0, 0)  # an external magnetic field (A/m)

system.energy += mm.Zeeman(H=H)

md.drive(system)
system.m.plane('z').mpl()
```

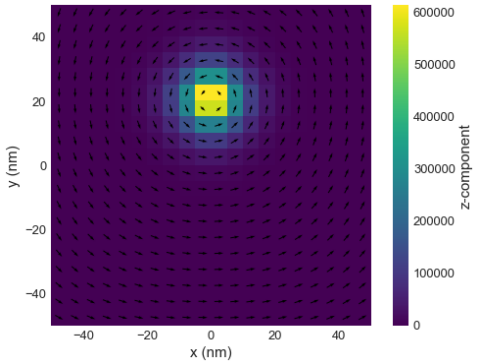



FIGURE 28. Magnetization profile applied the external field.

The vortex core is now moved in the positive $y$-direction. As the last step, we are going to turn off the field and simulate dynamics using `TimeDriver`. We are going to simulate for 5 ns and save the magnetization in 500 steps.

```
system.energy.zeeman.H  # return (10000.0,0,0)
system.energy -= mm.Zeeman(H=H)
```

```
system.energy.zeeman.H = (0, 0, 0)  # change the value of H

td = mc.TimeDriver()
td.drive(system, t=5e-9, n=500)
```

After the simulation finishes, we can plot the final magnetization state in Figure 29:

```
system.m.plane('z').mpl()
```

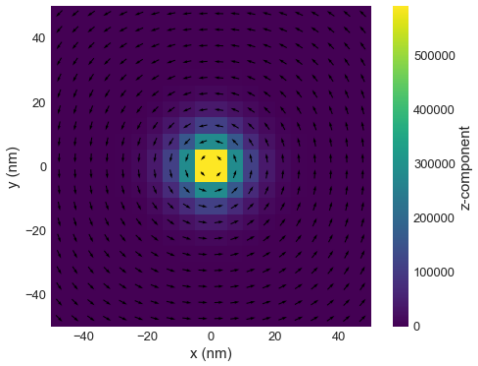



FIGURE 29. Magnetization profile using `TimeDriver` without the external field.

In the end of this tutorial, we are gonna shown some data analysis. The table in Figure 30 with scalar data collected during the simulation is:

```
system.m  # we can do it whatever we want for this field object
system.table.data # get the format of pandas
```

| | E | E_calc_count | max_dm/dt | dE/dt | delta_E | E_exchange | max_spin_ang | s |
|---|---|---|---|---|---|---|---|---|
| **0** | 1.853703e-18 | 43.0 | 233.280575 | -2.052828e-10 | -2.314942e-22 | 1.326230e-18 | 57.808185 | |
| **1** | 1.850967e-18 | 80.0 | 381.444041 | -3.279214e-10 | -5.165409e-22 | 1.333487e-18 | 58.286517 | |
| **2** | 1.847516e-18 | 117.0 | 510.874943 | -3.449882e-10 | -3.011803e-22 | 1.342260e-18 | 59.576209 | |
| **3** | 1.844316e-18 | 154.0 | 599.686479 | -2.882586e-10 | -2.459928e-22 | 1.349227e-18 | 60.742973 | |
| **4** | 1.841802e-18 | 191.0 | 676.446471 | -2.159121e-10 | -2.092790e-22 | 1.352728e-18 | 61.359905 | |
| **...** | ... | ... | ... | ... | ... | ... | ... | |
| **495** | 1.765512e-18 | 18364.0 | 1.354780 | -1.013063e-15 | -1.061037e-27 | 1.365677e-18 | 57.084570 | |
| **496** | 1.765512e-18 | 18401.0 | 1.331961 | -9.658800e-16 | -1.008047e-27 | 1.365677e-18 | 57.083946 | |
| **497** | 1.765512e-18 | 18438.0 | 1.311608 | -9.231580e-16 | -9.606871e-28 | 1.365677e-18 | 57.082998 | |
| **498** | 1.765512e-18 | 18475.0 | 1.287094 | -8.816371e-16 | -9.196201e-28 | 1.365677e-18 | 57.081626 | |
| **499** | 1.765512e-18 | 18512.0 | 1.266745 | -8.418461e-16 | -8.797218e-28 | 1.365677e-18 | 57.079980 | |

500 rows × 19 columns

FIGURE 30. Data table for data analysis.

What do we do: Now, let us plot the average $x$ and $y$ components of magnetization as a function of time in Figure 31, which gives us a rough idea of the vortex core position. To do that, we call the `mpl` method of the table object and pass `yaxis` list of columns we want to plot. The details of plotting scalar data will be covered later, when we focus on data analysis and visualization.

```
system.table.mpl(yaxis=['mx', 'my'])
```

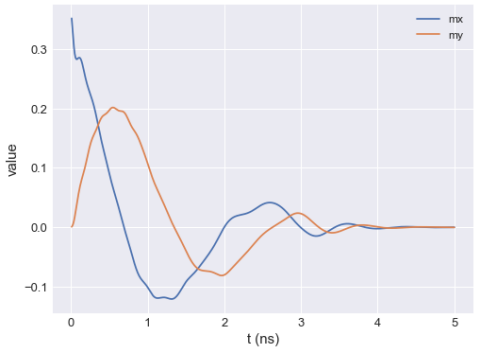



FIGURE 31. The average magnetization for components $m_x$ and $m_y$.

Finally, we are going to have a look at the magnetization field at different time steps. For that, we are going to use `micromagneticdata` module. We introduce some basics here, but the details will be covered later. The main object is `Data` which requires us to pass the name we used when we created the system object.

```
import micromagneticdata as md

data = md.Data(system.name)
```

After the data object is created, we can investigate in Figure 32 what drives we did so far:

```
data.info
```

| | drive_number | date | time | driver | t | n |
|---|---|---|---|---|---|---|
| **0** | 0 | 2020-06-25 | 14:44:46 | MinDriver | NaN | NaN |
| **1** | 1 | 2020-06-25 | 14:44:48 | MinDriver | NaN | NaN |
| **2** | 2 | 2020-06-25 | 14:44:51 | TimeDriver | 5.000000e-09 | 500.0 |

FIGURE 32. Data table for data analysis.

We are interested in the last drive with index 2. To get that drive from the data, we use drive method from the data object.

```
drive = data.drive(2)
data.drive(2).m0.average
print(data.drive(2).mif)  # we use it for oommf
```

Now, we can plot the magnetization at, for example, 200th step shown in Figure 33, as

```
drive.step(199).plane('z').mpl()
```

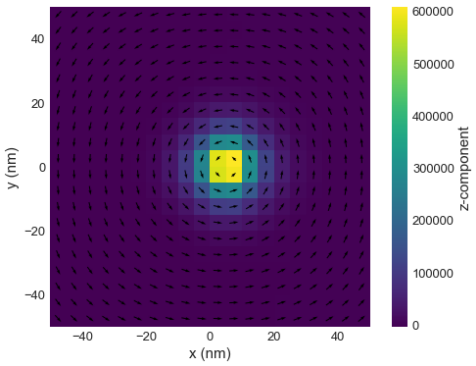



FIGURE 33. magnetization profile at 200th step.

```
def my_plot(n):
  data.drive(2).step(n).plane('z').mpl()

my_plot(2)
```

or interactively (we can build any way we want) as

```
@df.interact(n=data.drive(2).slider())
def my_plot(n):
  data.drive(2).step(n).plane('z').mpl()
```

or we want we change the color something as

```
@df.interact(n=data.drive(2).slider())
def my_plot(n, axis, color):
  data.drive(2).step(n).plane('z').mpl()
```

Let us now try to make our plot interactive, which is again going to be the topic of another session.

```
@df.interact(nstep=data.drive(2).slider(continuous_update=False))
def our_interactive_plot(nstep):
data.drive(2).step(nstep).plane('z').mpl()
```

We get a widget assigned to the plot, so we can interactively invetsigate the magnetization dynamics in Figure 34. As the last step, we can delete the directory created by ubermag:

```
# mc.delete(system)
```

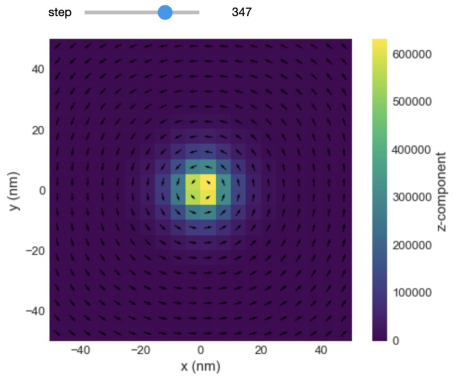



FIGURE 34. magnetization profile interactively.

2.1.3. *Spatially varying parameters 1.* In this tutorial, we explore different ways of defining a spatially varying parameter. As an example, we use the Zeeman energy term.

(I) Spatially constant **H**: Let us start by assembling a simple simulation where **H** does not vary in space. The sample is a "one-dimensional" chain of magnetic moments.

```
import oommfc as mc
import discretisedfield as df
import micromagneticmodel as mm

p1 = (-10e-9, 0, 0)
p2 = (10e-9, 1e-9, 1e-9)
cell = (1e-9, 1e-9, 1e-9)
region = df.Region(p1=p1, p2=p2)
mesh = df.Mesh(region=region, cell=cell)
```

The system has an energy equation, which consists of only Zeeman energy term.

```
H = (0, 0, 1e6)  # external magnetic field (A/m)
system = mm.System(name='zeeman_constant_H')
system.energy = mm.Zeeman(H=H)
```

We are going to minimize the system's energy using `oommfc.MinDriver` later. Therefore, we do not have to define the system's dynamics equation. Finally, we need to define the system's magnetization (`system.m`). We are going to make it random with $M_\mathrm{s} = 8 \times 10^5\,\mathrm{Am}^{-1}$.

```
import random
import discretisedfield as df

Ms = 8e5  # saturation magnetisation (A/m)

def m_fun(pos):
return [2*random.random()-1 for i in range(3)]  # give 3 of
                                    random numbers form $-1$ to
                                    $1$.
```

```
system.m = df.Field(mesh, dim=3, value=m_fun, norm=Ms)
```

The magnetization, we set is

```
system.m.k3d_vector(color_field=system.m.z)
```

Now, we can minimize the system's energy by using `oommfc.MinDriver`.

```
md = mc.MinDriver()
md.drive(system)
```

We expect that now all magnetic moments are aligned parallel to the external magnetic field (in the $z$-direction).

```
system.m.k3d_vector(color_field=system.m.z)
```

(II) Spatially varying $H$: There are two different ways how a parameter can be made spatially varying, by using: Dictionary or `discretisedfield.Field`.

In order to define a parameter using a dictionary, subregions must be defined in the mesh. Subregions are defined as a dictionary, whose keys are the strings and values are `discretisedfield.Region` objects, which take two corner points of the region as input parameters.

```
p1 = (-10e-9, 0, 0)
p2 = (10e-9, 1e-9, 1e-9)
cell = (1e-9, 1e-9, 1e-9)
subregions = {'subregion1': df.Region(p1=(-10e-9, 0, 0), p2=(0,
                                      1e-9, 1e-9)),
  'subregion2': df.Region(p1=(0, 0, 0), p2=(10e-9, 1e-9, 1e-9))
                                     }

region = df.Region(p1=p1, p2=p2)
mesh = df.Mesh(region=region, cell=cell, subregions=subregions)
```

Let us say we want to apply the external magnetic field $\mathbf{H}$ in region 1 in the $x$-direction and in region 2 in the negative $z$-direction. `H` is now defined as a dictionary:

```
H = {'subregion1': (1e6, 0, 0), 'subregion2': (0, 0, -1e6)}
```

The system object is

```
system = mm.System(name='zeeman_dict_H')
system.energy = mm.Zeeman(H=H)
system.m = df.Field(mesh, dim=3, value=m_fun, norm=Ms)
```

Its magnetization is

```
system.m.k3d_vector(color_field=system.m.z)
```

After we minimize the energy

```
md.drive(system)
```

The magnetization is as we expected.

```
system.m.k3d_vector(color_field=system.m.z)
```

```
mesh.mpl_subregions()
mesh.k3d_subregions()
```

The last thing is how we can use the `discretisedfield.Field` object. Let us say that the external magnetic field in Figure 35 varies in space as

$$\mathbf{H}(x, y, z) = (c^2 x, 0, c)$$

where $c = 10^9$ and the entire field is normalized with $H = 10^6\,\mathrm{Am}^{-1}$. The value of a spatially varying field is set using a Python function.

```
def H_fun(point):
  x, y, z = point
  c = 1e9
  return (c*c*x, 0, c)
```

The external magnetic field is

```
H = df.Field(mesh, dim=3, value=H_fun, norm=1e6)
H.plane('y').mpl(figsize=(15,3))
```

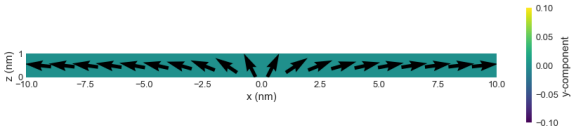



FIGURE 35. External magnetic field profile.

Same story as before. The system is

```
system = mm.System(name='zeeman_field_H')
system.energy = mm.Zeeman(H=H)
system.m = df.Field(mesh, dim=3, value=m_fun, norm=Ms)
```

and its magnetization is

```
system.m.k3d_vector(color_field=system.m.z)
```

After the energy minimization, the magnetization is:

```
md.drive(system)
system.m.k3d_vector(color_field=system.m.z)
```

Next thing is how to change the DMI. In this notebook, one data point from Figure 2 in [1] (here we look at the Bloch point from the boundary between different grains, absolute different DMI, one side of the grain $D > 0$, the other is $D < 0$) is simulated. We need to relax a 150 nm disk, which consists of two layers (top and bottom layers) with different sign of Dzyaloshinskii-Moriya constant $D$. The bottom layer with $D < 0$ has 20 nm thickness, whereas the top layer with $D > 0$ has 10 nm thickness. We start by importing the necessary modules and creating the mesh with two regions.

```
import oommfc as mc
import discretisedfield as df
import micromagneticmodel as mm

d = 150e-9
hb = 20e-9
ht = 10e-9
cell = (5e-9, 5e-9, 2.5e-9)
```

```
n = (31, 31, 5)
subregions = {'r1': df.Region(p1=(-d/2, -d/2, -hb), p2=(d/2, d/2, 0)),
  'r2': df.Region(p1=(-d/2, -d/2, 0), p2=(d/2, d/2, ht))}
p1 = (-d/2, -d/2, -hb)
p2 = (d/2, d/2, ht)
mesh = df.Mesh(p1=p1, p2=p2, n=n, subregions=subregions)
```

The mesh domain and the discretization cells are:

```
mesh.k3d()
```

and the two regions in Figure 36 we defined are: Now, we need to define the system

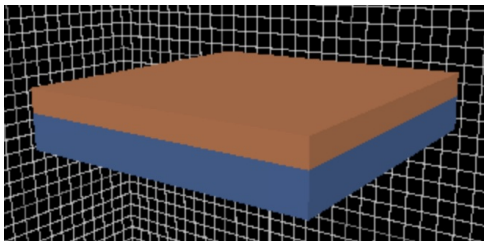


FIGURE 36. Geometry of the sample.

object, and by setting magnetization saturation, set the geometry to be a disk.

```
system = mm.System(name='bloch_point')
D = {'r1': 1.58e-3, 'r2': -1.58e-3, 'r1:r2': 1.58e-9} # r1 for top, r2
                                    for bottom, r1:r2 for boundary
Ms = 3.84e5
A = 8.78e-12

def Ms_fun(point):
  x, y, z = point
  if x**2 + y**2 <= (d/2)**2:
    return Ms
  else:
    return 0

system.energy = mm.Exchange(A=A) + mm.DMI(D=D, crystalclass='T') + mm.
                                   Demag()
system.m = df.Field(mesh, dim=3, value=(0, 0, 1), norm=Ms_fun)
```

Our sample is now:

```
system.m.norm.k3d_nonzero()
```

Now, we can minimize the system's energy by using `MinDriver`.

```
md = mc.MinDriver()
md.drive(system)
```

The out-of-plane magnetization component ($m_z$) in Figure 37 is now:

```
system.m.z.k3d_scalar(filter_field=system.m.norm)
```

We can see that two vortices with different orientation emerged. We can inspect this closer by plotting two layers of magnetization in two different layers in Figure 38:

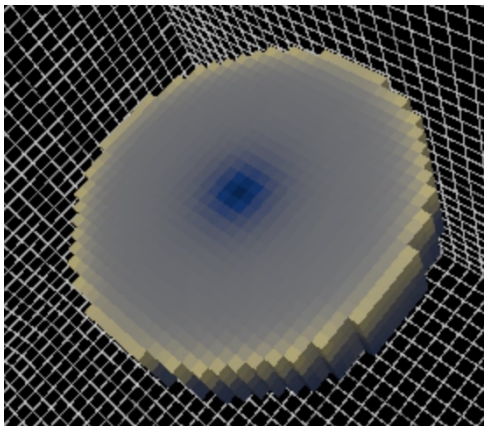


FIGURE 37. The out-of-plane magnetization component $m_z$.

```
import k3d
plot = k3d.plot()
system.m.plane(z=-10e-9, n=(20, 20)).k3d_vector(plot=plot, color_field
                                    =system.m.z, head_size=30)
system.m.plane(z=5e-9, n=(20, 20)).k3d_vector(plot=plot,color_field=
                                    system.m.z, head_size=30)
plot.display()
```

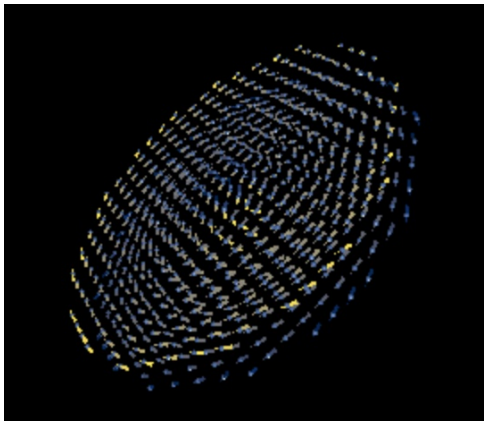


FIGURE 38. Two vortices with different orientation emerged.

We can now plot another cross section and see that the Bloch point emerged.

```
@df.interact(y=system.m.mesh.slider('y', continuous_update=False))
def my_plot(y):
  system.m.plane(y=y).mpl(figsize=(10, 5), vector_scale=1e7,
                                    scalar_field=getattr(system.m, 'x')
                                    )
```

2.1.4. *Periodic boundary conditions.* In this tutorial, we compute and relax a skyrmion in an interfacial-DMI (Cnv) material in a part of an infinitely large thin film.

```
import oommfc as mc
import discretisedfield as df
import micromagneticmodel as mm
```

We define mesh in cuboid through corner points `p1` and `p2`, and discretization cell size cell.

```
region = df.Region(p1=(-50e-9, -50e-9, 0), p2=(50e-9, 50e-9, 10e-9))
mesh = df.Mesh(region=region, cell=(5e-9, 5e-9, 5e-9), bc='xy') #
                                    boundary conditions could be x, y,
                                    z, xy, xyz, yzx periodic, specify
                                    neumann, direchlet.
```

The mesh we defined is:

```
mesh.k3d()
```

Now, we can define the system object by first setting up the Hamiltonian:

```
system = mm.System(name='skyrmion')

system.energy = (mm.Exchange(A=1.6e-11)
    + mm.DMI(D=4e-3, crystalclass='Cnv') # interfacial DMI
    + mm.UniaxialAnisotropy(K=0.2e6, u=(0, 0, 1))
    + mm.Zeeman(H=(0, 0, 1e5)))

system.energy
```

Now, we need to define a function to define the initial magnetization which is going to relax to skyrmion.

```
def m_init(point):
  """Function to set initial magnetisation direction:
  -z inside cylinder (r=10nm),
  +z outside cylinder.
  y-component to break symmetry.

  """
  x, y, z = point
  if (x**2 + y**2)**0.5 < 10e-9:
    return (0, 0, -1)
  else:
    return (0, 0, 1)

system.m = df.Field(mesh, dim=3, value=m_init, norm=1.1e6)
```

The initial magnetization is:

```
system.m.plane('z').mpl()
```

Finally we can minimize the energy and plot the magnetization in Figure 39.

```
# minimize the energy
md = mc.MinDriver()
md.drive(system)

# Plot relaxed configuration: vectors in z-plane
system.m.plane('z').mpl()
```

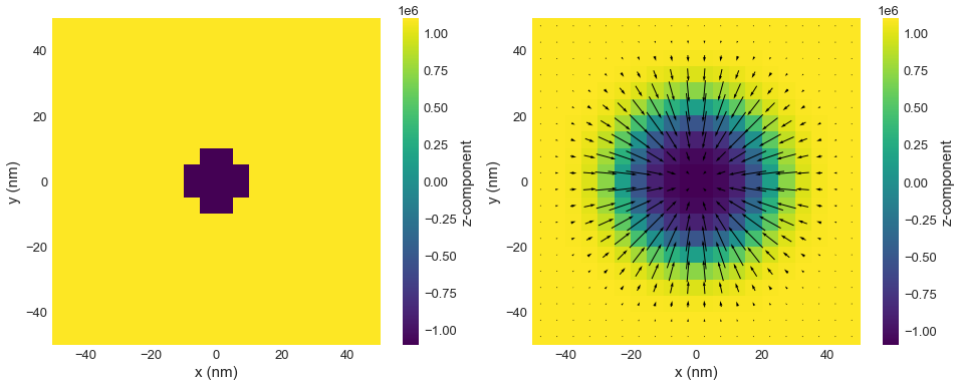



FIGURE 39. Magnetization profile: Lf for initialized magnetization; Rg for the relaxed magnetization.

we can plot in $z$-direction only in Figure 40 by

```
# Plot z-component only:
system.m.z.plane('z').mpl()

# 3d-plot of z-component
system.m.z.k3d_scalar(filter_field=system.m.norm)
```

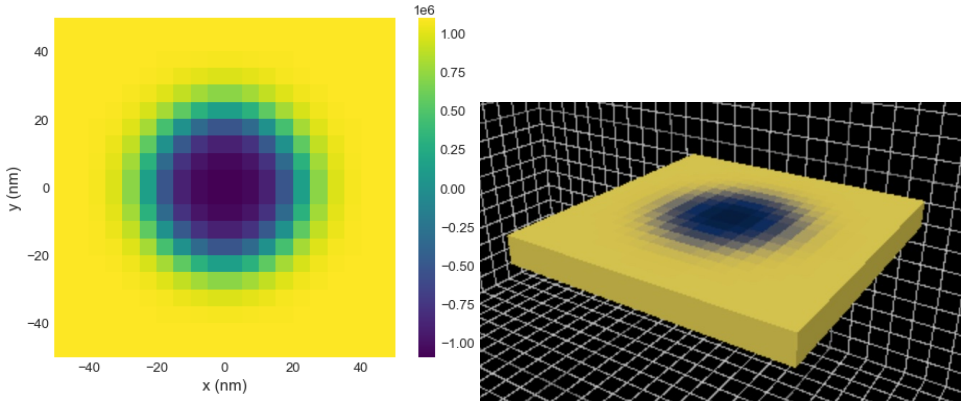



FIGURE 40. Magnetization in $z$-component.

Finally we can sample and plot the magnetization in Figure 41 along the line in the middle:

```
system.m.line(p1=(-49e-9, 0, 0), p2=(49e-9, 0, 0),n=20) # line object

system.m.line(p1=(-49e-9, 0, 0), p2=(49e-9, 0, 0),n=20).mpl()

system.m.orientation.z.line(p1=(-49e-9, 0, 0), p2=(49e-9, 0, 0), n=20)
                                    .mpl()
```

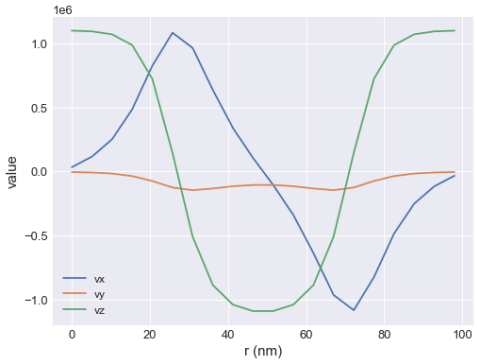



FIGURE 41. Sample and plot the magnetization in the middle.

Finally let us compute the skyrmion number

$$S = \frac{1}{4\pi} \int \mathbf{m} \cdot \left( \frac{\partial \mathbf{m}}{\partial x} \times \frac{\partial \mathbf{m}}{\partial y} \right) dxdy$$

```
import math
m = system.m.orientation.plane('z')    # plane means the surface
                                       integral

1/(4*math.pi) * (m @ (m.derivative('x') & m.derivative('y'))).
                                       surface_integral  # & means cross
                                       product, @ means dot product
# return -0.9244762913457669
```

```
system.m.orientation.plane('z').topological_charge()  # return -0.
                                       9244762913457671
system.m.orientation.plane('z').topological_charge(method='berg-
                                       luescher')  # return -0.
                                       999745566485005 # this method would
                                        be better
```

2.1.5. *Current induced domain wall motion.* What is the idea here: In this tutorial we show how Zhang-Li spin transfer torque (STT) can be included in micromagnetic simulations. To illustrate that, we will try to move a domain wall pair using spin-polarized current.

Let us simulate a two-dimensional sample (nano-strip) with length $L = 500\,\mathrm{nm}$, width $w = 20\,\mathrm{nm}$ and discretisation cell $(2.5\,\mathrm{nm}, 2.5\,\mathrm{nm}, 2.5\,\mathrm{nm})$. The material parameters are:

(a) exchange energy constant $A = 15\,\mathrm{pJ\,m^{-1}}$;
(b) Dzyaloshinskii-Moriya energy constant $D = 3\,\mathrm{mJ\,m^{-2}}$;
(c) uniaxial anisotropy constant $K = 0.5\,\mathrm{MJ\,m^{-3}}$ with easy axis $\mathbf{u}$ in the out of plane direction $(0, 0, 1)$;
(d) gyrotropic ratio $\gamma = 2.211 \times 10^5\,\mathrm{m\,A^{-1}\,s^{-1}}$, and Gilbert damping $\alpha = 0.3$.

What we want to do: we want to put the domain wall pair in this nanostrip applied spin polarized current.

```
import oommfc as mc
import discretisedfield as df
import micromagneticmodel as mm

# Definition of parameters
L = 500e-9  # sample length (m)
w = 20e-9  # sample width (m)
d = 2.5e-9  # discretisation cell size (m)
Ms = 5.8e5  # saturation magnetisation (A/m)
A = 15e-12  # exchange energy constant (J/)
D = 3e-3  # Dzyaloshinkii-Moriya energy constant (J/m**2)
K = 0.5e6  # uniaxial anisotropy constant (J/m**3)
u = (0, 0, 1)  # easy axis
gamma0 = 2.211e5  # gyromagnetic ratio (m/As)
alpha = 0.3  # Gilbert damping

# Mesh definition
p1 = (0, 0, 0)
p2 = (L, w, d)
cell = (d, d, d)
region = df.Region(p1=p1, p2=p2)
mesh = df.Mesh(region=region, cell=cell)

# Micromagnetic system definition
system = mm.System(name='domain_wall_pair')
system.energy = mm.Exchange(A=A) + \
    mm.DMI(D=D, crystalclass="Cnv") + \
    mm.UniaxialAnisotropy(K=K, u=u)
system.dynamics = mm.Precession(gamma0=gamma0) + mm.Damping(alpha=
                                    alpha)
```

Because we want to move a DW pair, we need to initialize the magnetization in an appropriate way before we relax the system.

```
def m_value(point):
  x, y, z = point
  if 20e-9 < x < 40e-9:
    return (0, 0, -1)
  else:
    return (0, 0, 1)

system.m = df.Field(mesh, dim=3, value=m_value, norm=Ms)

system.m.z.plane('z').k3d_scalar()
```

Now, we can relax the magnetization.

```
md = mc.MinDriver()
md.drive(system)

system.m.z.plane('z').k3d_scalar()
```

Now we can add the STT term to the dynamics equation. in Figure 42.

```
ux = 400  # velocity in x-direction (m/s)
beta = 0.5  # non-adiabatic STT parameter

system.dynamics += mm.ZhangLi(u=ux, beta=beta)  # please notice the
                                    use of '+=' operator
```

And drive the system for 0.5 ns:

```
td = mc.TimeDriver()
td.drive(system, t=0.5e-9, n=100)

system.m.z.plane('z').k3d_scalar()
```

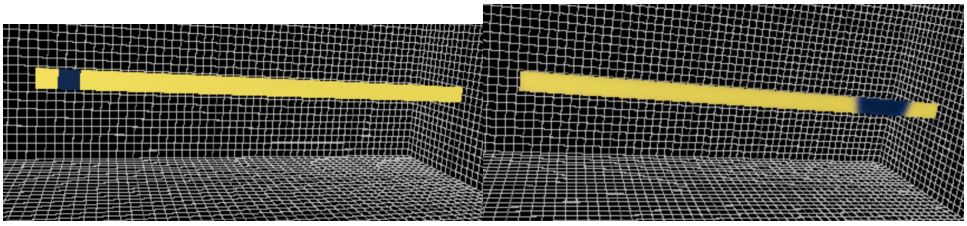


FIGURE 42. Domain wall driven by STT.

We see that the DW pair has moved to the positive $x$ direction. Now, let us visualize the motion using interactive plot.

```
import k3d
import micromagneticdata as md

data = md.Data(system.name)

plot = k3d.plot()

@df.interact(n=data.drive(1).slider())
def my_plot(n):
  data.drive(1).step(n).z.k3d_scalar(plot=plot, interactive_field=
                                     system.m)

plot.display()
```

**Exercise 2.** *Modify the previous code to obtain one domain wall instead of a domain wall pair and move it using the same current.*

```
# Definition of parameters
L = 500e-9  # sample length (m)
w = 20e-9  # sample width (m)
d = 2.5e-9  # discretisation cell size (m)
Ms = 5.8e5  # saturation magnetisation (A/m)
A = 15e-12  # exchange energy constant (J/)
D = 3e-3  # Dzyaloshinkii-Moriya energy constant (J/m**2)
K = 0.5e6  # uniaxial anisotropy constant (J/m**3)
u = (0, 0, 1)  # easy axis
gamma0 = 2.211e5  # gyromagnetic ratio (m/As)
alpha = 0.3  # Gilbert damping

# Mesh definition
p1 = (0, 0, 0)
p2 = (L, w, d)
cell = (d, d, d)
```

```
region = df.Region(p1=p1, p2=p2)
mesh = df.Mesh(region=region, cell=cell)

# Micromagnetic system definition
system = mm.System(name='domain_wall')
system.energy = mm.Exchange(A=A) + \
    mm.DMI(D=D, crystalclass='Cnv') + \
    mm.UniaxialAnisotropy(K=K, u=u)
system.dynamics = mm.Precession(gamma0=gamma0) + mm.Damping(alpha=alpha
                                    )

def m_value(point):
  x, y, z = point
  # Modify the following line
  if 20e-9 < x:
    return (0, 0, -1)
  else:
    return (0, 0, 1)
  # We have added the y-component of 1e-8 to the magnetization to be
                                    able to
  # plot the vector field. This will not be necessary in the long run.

system.m = df.Field(mesh, dim=3, value=m_value, norm=Ms)

system.m.z.plane('z').k3d_scalar()
```

```
md = mc.MinDriver()
md.drive(system)

system.m.z.plane('z').k3d_scalar()
```

```
ux = 400  # velocity in x direction (m/s)
beta = 0.5  # non-adiabatic STT parameter

system.dynamics += mm.ZhangLi(u=ux, beta=beta)

td = mc.TimeDriver()
td.drive(system, t=0.5e-9, n=100)

system.m.z.plane('z').k3d_scalar()
```

2.1.6. *Energy equation.* Apart from the magnetization field (and dynamics equation), energy equation must be defined for a micromagnetic system. In this tutorial, we explore how an energy equation can be specified in Ubermag.

The Ubermag subpackage we use for defining micromagnetic models is `micromagneticmodel` and `discretisedfield` we are going to use for defining magnetization field.

```
import discretisedfield as df
import micromagneticmodel as mm
```

As an example, let us define a one-dimensional chain of magnetic moments.

```
p1 = (0, 0, 0)
p2 = (10e-9, 1e-9, 1e-9)
n = (10, 1, 1)

region = df.Region(p1=p1, p2=p2)
```

```
mesh = df.Mesh(region=region, n=n)

mesh.k3d()
```

To demonstrate how an energy equation is specified, we are first going to have a look at Exchange and DMI individually and then use them both in our energy equation.

(I) Exchange energy term: We use $A = 8\,\mathrm{pJ\,m^{-1}}$ on a non-uniform magnetization configuration with $M_\mathrm{s} = 1 \times 10^6\,\mathrm{Am^{-1}}$.

```
system = mm.System(name='exchange')
system.energy = mm.Exchange(A=8e-12)

Ms = 1e6  # saturation magnetization (A/m)

def m_initial(point):
  x, y, z = point
  if x <= 5e-9:
    return (0, 1, 1)
  else:
    return (1, 0, 0)

system.m = df.Field(mesh, dim=3, value=m_initial, norm=Ms)
system.m.k3d_vector(head_size=3)
```

The energy equation is

```
system.energy
```

Now, let us relax the system:

```
import oommfc as mc

md = mc.MinDriver()
md.drive(system)

system.m.k3d_vector(head_size=3)
```

(II) Dzyaloshinskii-Moriya energy term: There are two parameters that must be defined for Dzyaloshinskii-Moriya energy term. The first one is the DMI energy constant $D$ and the other is the crystalographic class. Crystalographic class can be: `T`, `O`, `Cnv`, or `D2d`.

Again, we demonstrate its effect by starting from a uniform configuration, with $D = 3\,\mathrm{mJ\,m^{-2}}$ and crystallographic class T.

```
system = mm.System(name='dmi')
system.energy = mm.DMI(D=3e-3, crystalclass='T')
system.m = df.Field(mesh, dim=3, value=(0, 0, 1), norm=Ms)

md.drive(system)

system.m.k3d_vector(head_size=3)
```

The energy equation is

```
system.energy
```

Now, we can change the crystallographic class to `Cnv` and see what happens to the relaxed magnetization.

```
system.energy.dmi.crystalclass = 'Cnv'
system.energy
```

```
md.drive(system)
system.m.k3d_vector(head_size=3)
```

(III) Multiple energy terms in energy equation: Multiple energy terms are added to the energy equation by applying a binary '+' operator between individual energy term objects. As an example, we want to relax a one-dimensional chain of magnetic moments of length $L = 10\,\mathrm{nm}$ with 20 discretization cells. The energy equation consists of:
(a) exchange energy with $A = 1 \times 10^{-11}\,\mathrm{J\,m^{-1}}$, and
(b) Dzyaloshinskii-Moriya energy with $D = 4\pi A/L \approx 12.56 \times 10^{-3}\,\mathrm{J\,m^{-2}}$ and 'T' crystal class.
The magnetization saturation is $M_\mathrm{s} = 8 \times 10^{6}\,\mathrm{A\,m^{-1}}$ and we start with a uniform magnetization state.

```
A = 1e-11  # exchange energy constant (J/m)
D = 12.56e-3  # DMI energy constant (J/m**2)
Ms = 8e6 # Saturation magnetisation (A/m)

region = df.Region(p1=(0, 0, 0), p2=(10e-9, 1e-9, 1e-9))
mesh = df.Mesh(region=region, n=(20, 1, 1))

system = mm.System(name='exchange_and_DMI')
system.energy = mm.Exchange(A=A) + mm.DMI(D=D, crystalclass='T'
                                          )
system.m = df.Field(mesh, dim=3, value=(0, 1, 1), norm=Ms)

system.energy
```

```
md.drive(system)

system.m.plane('y').k3d_vector(color_field=system.m.z)
```

We can also sample the magnetization along the sample in Figure 43.

```
line = system.m.line(p1=system.m.mesh.region.pmin, p2=system.m.
                                        mesh.region.pmax, n=19)
line.mpl(marker='o')
```

**Exercise 3.** *Assemble the appropriate energy equation in order to obtain a skyrmion state in a disk with* 100 *nm diameter and* 5 *nm thickness. In addition, choose the appropriate initial magnetization state.*

*The solution is list here in Figure 44.*

```
d = 100e-9
thickness = 5e-9
cell = 3*(5e-9,)

p1 = (-d/2, -d/2, 0)
p2 = (d/2, d/2, thickness)
region = df.Region(p1=p1, p2=p2)
mesh = df.Mesh(region=region, cell=cell)
system = mm.System(name='skyrmion')
```

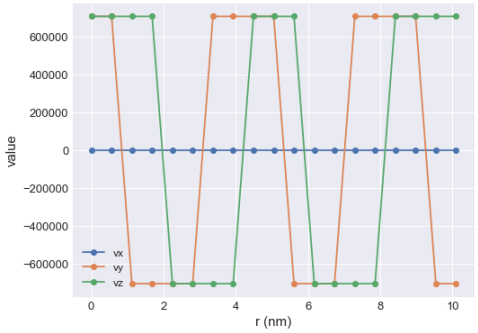



FIGURE 43. sample the magnetization along the sample.

```
system.energy = mm.Exchange(A=8.78e-12) + mm.DMI(D=1.58e-3,
                                    crystalclass='T') + mm.Demag()

def m_initial(point):
  x, y, z = point
  if x**2 + y**2 < (d/4)**2:
    return (0, 0, -1)
  else:
    return (0, 0, 1)

def Ms_fun(point):
  x, y, z = point
  if x**2 + y**2 < (d/2)**2:
    return 3.84e5
  else:
    return 0

system.m = df.Field(mesh, dim=3, value=m_initial, norm=Ms_fun)

md.drive(system)

system.m.plane('z').mpl(figsize=(10, 10))
```

```
system.m.z.k3d_scalar(filter_field=system.m.norm)
```

2.1.7. *Dynamics equation.* The dynamics of magnetization field $\mathbf{m}$, without external excitations (e.g. spin-polarized current) is governed by the Landau-Lifshitz-Gilbert (LLG) equation

$$\frac{d\mathbf{m}}{dt} = \underbrace{-\gamma_0(\mathbf{m}\times\mathbf{H}_{\text{eff}})}_{\text{precession}} + \underbrace{\alpha\left(\mathbf{m}\times\frac{d\mathbf{m}}{dt}\right)}_{\text{damping}},$$

where $\gamma_0 = \mu_0\gamma$ is the gyromagnetic ratio, $\alpha$ is the Gilbert damping, and $\mathbf{H}_{\text{eff}} = -\frac{1}{\mu_0 M_{\text{s}}}\frac{\delta w(\mathbf{m})}{\delta \mathbf{m}}$ is the effective field. It consists of two terms: precession and damping. In this tutorial, we will explore some basic properties of this equation to understand how to define it in simulations.

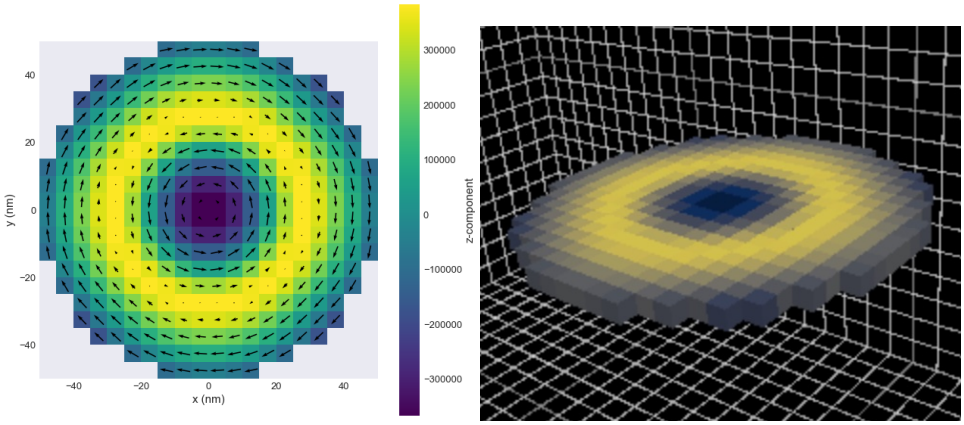



FIGURE 44. Magnetization profile.

We will study the simplest "zero-dimensional" case - macrospin. In the first step, after we import necessary modules and create the mesh which consists of a single discretization cell.

```
import oommfc as mc
import discretisedfield as df
import micromagneticmodel as mm

# Define a macrospin mesh (i.e. one discretization cell).
p1 = (0, 0, 0)  # first point of the mesh domain (m)
p2 = (1e-9, 1e-9, 1e-9)  # second point of the mesh domain (m)
n = (1, 1, 1)  # discretization cell size (m)

region = df.Region(p1=p1, p2=p2)
mesh = df.Mesh(region=region, n=n)
```

Now, we can create a micromagnetic system object.

```
system = mm.System(name='macrospin')
```

Let us assume we have a simple Hamiltonian which consists of only Zeeman energy term

$$w = -\mu_0 M_\mathrm{s} \mathbf{m} \cdot \mathbf{H},$$

where $M_\mathrm{s}$ is the saturation magnetization, $\mu_0$ is the magnetic constant, and $\mathbf{H}$ is the external magnetic field. We apply the external magnetic field with magnitude $H = 2 \times 10^6\,\mathrm{A\,m^{-1}}$ in the positive $z$ direction.

```
H = (0, 0, 2e6)  # external magnetic field (A/m)
system.energy = mm.Zeeman(H=H)
```

In the next step we can define the system's dynamics. Let us assume we have $\gamma_0 = 2.211 \times 10^5\,\mathrm{m\,A^{-1}\,s^{-1}}$ and $\alpha = 0.1$.

```
gamma0 = 2.211e5  # gyromagnetic ratio (m/As)
alpha = 0.1  # Gilbert damping
```

```
system.dynamics = mm.Precession(gamma0=gamma0) + mm.Damping(alpha=
                                                    alpha)
```

To check what is our dynamics equation:

```
system.dynamics
```

Before we start running time evolution simulations, we need to initialize the magnetization. In this case, our magnetization is pointing in the positive $x$ direction with $M_\mathrm{s} = 8 \times 10^6\,\mathrm{A\,m^{-1}}$. The magnetization is defined using `Field` class from the `discretisedfield` package we imported earlier.

```
initial_m = (1, 0, 0)  # vector in x direction
Ms = 8e6  # magnetisation saturation (A/m)

system.m = df.Field(mesh, dim=3, value=initial_m, norm=Ms)
```

Now, we can run the time evolution using `TimeDriver` for $t = 0.1\,\mathrm{ns}$ and save the magnetization configuration in $n = 200$ steps.

```
td = mc.TimeDriver()
td.drive(system, t=0.1e-9, n=200)
```

How different system parameters vary with time, we can inspect by showing the system's datatable.

```
system.table.data
```

However, in our case it is much more informative if we plot the time evolution of magnetization $z$ component $m_z(t)$.

```
system.table.mpl(yaxis=['mz'])
```

Similarly, we can plot all three magnetization components shown in Figure 45.

```
system.table.mpl(yaxis=['mx', 'my', 'mz'])
```

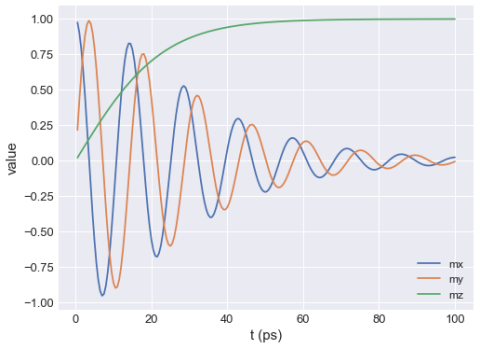



FIGURE 45. The magnetization along the sample.

We can see that after some time the macrospin aligns parallel to the external magnetic field in the $z$ direction. We can also have a look at the dynamics using an interactive plot.

**Exercise 4.** *Modify Gilbert damping and set it to* $\alpha = 0.005$ *and see what happens with the dynamics. The solution is implemented here in Figure 46.*

```
system.dynamics.damping.alpha = 0.005
system.m = df.Field(mesh, dim=3, value=initial_m, norm=Ms)

td.drive(system, t=0.1e-9, n=200)

system.table.mpl(yaxis=['mx', 'my', 'mz'])
```

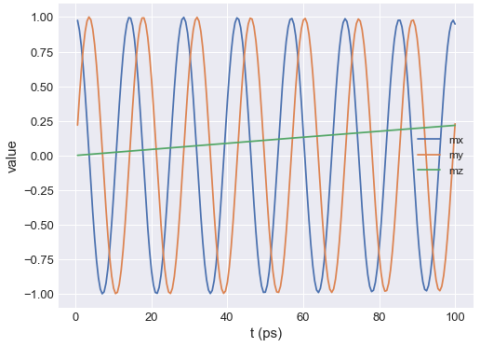



FIGURE 46. The magnetization along the sample.

**Exercise 5.** *Repeat the simulation with* $\alpha = 0.1$ *and* $\mathbf{H} = (0, 0, -2 \times 10^6)\, A\, m^{-1}$. *The result is shown in Figure 47.*

```
system.energy.zeeman.H = (0, 0, -2e6)
system.dynamics.damping.alpha = 0.1
system.m = df.Field(mesh, dim=3, value=initial_m, norm=Ms)

td.drive(system, t=0.1e-9, n=200)

system.table.mpl(yaxis=['mx', 'my', 'mz'])
```

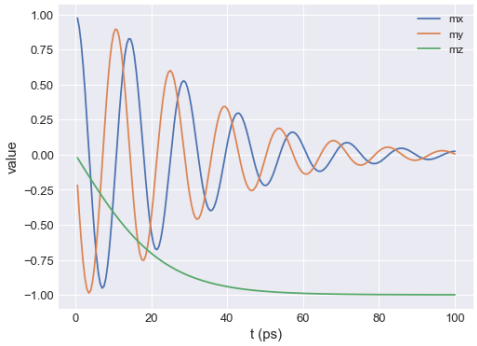



FIGURE 47. The magnetization along the sample.

**Exercise 6.** *Keep using $\alpha = 0.1$. Change the field from* `H = (0, 0, -2e6)` *to* `H = (0, -1.41e6, -1.41e6)`*, and plot $m_x(t)$, $m_y(t)$ and $m_z(t)$ as above. Can you explain the (initially non-intuitive) output? The solution is implemented here in Figure 48.*

```
system.energy.zeeman.H = (0, -1.41e6, -1.41e6)

td.drive(system, t=0.1e-9, n=200)

system.table.mpl(yaxis=['mx', 'my', 'mz'])
```

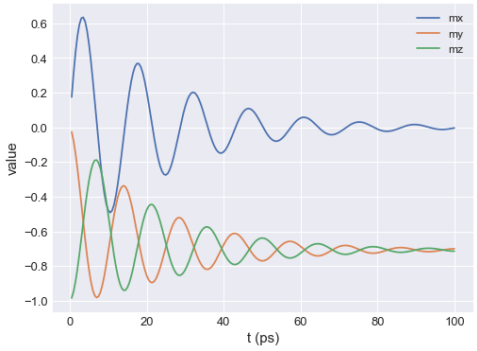



FIGURE 48. The magnetization along the sample.

2.1.8. *Exercise: Domain wall pair conversion.* We want to simulate a domain wall conversion in a two-dimensional thin film sample with:

(a) exchange energy constant $A = 15\,\mathrm{pJ\,m^{-1}}$;
(b) Dzyaloshinskii-Moriya energy constant $D = 3\,\mathrm{mJ\,m^{-2}}$;
(c) uniaxial anisotropy constant $K = 0.5\,\mathrm{MJ\,m^{-3}}$ with $\hat{\mathbf{u}} = (0, 0, 1)$ in the out of plane direction;
(d) gyrotropic ratio $\gamma = 2.211 \times 10^5\,\mathrm{m\,A^{-1}\,s^{-1}}$, and Gilbert damping $\alpha = 0.3$..

Please carry out the following steps:

(a) Create the following geometry in Figure 49 with discretization cell size $(2\,\mathrm{nm}, 2\,\mathrm{nm}, 2\,\mathrm{nm})$:

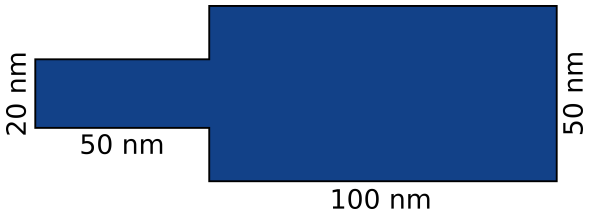



FIGURE 49. Geometry of thin film.

(b) Initialize the magnetization so that when relaxes, a domain pair is present in the narrower part of the geometry;
(c) Relax the system. Is a domain wall pair contained in the constrained part?

(d) Apply the spin polarized current in the positive $x$ direction with velocity $\mathbf{u} = (400, 0, 0)\,\mathrm{m\,s^{-1}}$, with $\beta = 0.5$;
(e) Evolve the system over 0.2 ns. What did you get? see reference [2] and therein.

Here we put the domain wall in the narrower part, and apply the spin polarized current in the $x$-direction, and it push the domain wall pair into the right region. After the previous tutorials, we have implemented here.

```
import oommfc as oc
import discretisedfield as df
import micromagneticmodel as mm
%matplotlib inline

Ms = 5.8e5  # saturation magnetisation (A/m)
A = 15e-12  # exchange energy constant (J/)
D = 3e-3  # Dzyaloshinkii-Moriya energy constant (J/m**2)
K = 0.5e6  # uniaxial anisotropy constant (J/m**3)
u = (0, 0, 1)  # easy axis
gamma0 = 2.211e5  # gyromagnetic ratio (m/As)
alpha = 0.3  # Gilbert damping

system = mm.System(name='dw_pair_conversion')
system.energy = mm.Exchange(A=A) + mm.DMI(D=D, crystalclass="Cnv") +
                                   mm.UniaxialAnisotropy(K=K, u=u)
system.dynamics = mm.Precession(gamma0=2.211e5) + mm.Damping(alpha=
                                   alpha)

p1 = (0, 0, 0)
p2 = (150e-9, 50e-9, 2e-9)
cell = (2e-9, 2e-9, 2e-9)

region = df.Region(p1=p1, p2=p2)
mesh = df.Mesh(region=region, cell=cell)

def Ms_fun(point):
  x, y, z = point
  if x < 50e-9 and (y < 15e-9 or y > 35e-9):
    return 0
  else:
    return Ms

def m_init(point):
  x, y, z = point
  if 30e-9 < x < 40e-9:
    return (0.1, 0.1, -1)
  else:
    return (0.1, 0.1, 1)

system.m = df.Field(mesh, dim=3, value=m_init, norm=Ms_fun)

system.m.z.plane('z').k3d_scalar(filter_field=system.m.norm)
```

```
md = oc.MinDriver()
md.drive(system)

system.m.z.plane('z').k3d_scalar(filter_field=system.m.norm)
```

```
ux = 400  # velocity in x direction (m/s)
beta = 0.5  # non-adiabatic STT parameter

system.dynamics += mm.ZhangLi(u=ux, beta=beta)
```

```
td = oc.TimeDriver()
td.drive(system, t=0.2e-9, n=200)

system.m.z.plane('z').k3d_scalar(filter_field=system.m.norm)
```

As a result, we got a skyrmion formed in the wider region in Figure 50.

```
import micromagneticdata as md

data = md.Data(system.name)

@df.interact(n=data.drive(1).slider(continuous_update=False))
def myplot(n):
  data.drive(1).step(n).z.plane('z').mpl_scalar(figsize=(15, 5),
                                    filter_field=system.m.norm)
```

2.2. **session 3.** This session will cover data analysis and visualization. The video can be accessed by https://app.vidgrid.com/view/kX41BkJONWe6/?sr=QfBefC. This is the last session of our online workshop. The main topic of this session is going to be data analysis and visualization. However, similar to the previous sessions, we are going to go through tutorials, which are going to introduce a mixture of simulation techniques and micromagnetic concepts as well. The outline of this tutorial is as follows,

(a) Choosing runner; Multiple energy terms of the same class; RKKY energy term; Time-dependent field; Negative exchange energy constant; Energy term computations; Field operations 1; Field operations 2; Field file formats;
(b) Line basics and visualization; Table basics; Table visualization; Table interactive plots; Region visualization; Mesh visualization; Field mpl visualization; Field k3d visualization; Interactive plotting; Various topics.

2.2.1. *Choosing runner.* In this tutorial, we show how to choose different "runners" to run your simulations. This is helpful if you want to change OOMMF installation you want to use. This is in particular helpful if you want to run OOMMF inside Docker, which allows us to run simulations on a "small linux machine", which is automatically pulled from the cloud, simulations are run inside, and in the end it is destroyed automatically. This all happens in the background and requires no special assistance from the user. In order to use Docker (which is a small linux machine), we need to have it installed on our machine - you can download it here: https://www.docker.com/products/docker-desktop.

For that example, we simulate a skyrmion in a sample with periodic boundary conditions.

```
import oommfc as mc
import discretisedfield as df
import micromagneticmodel as mm
```

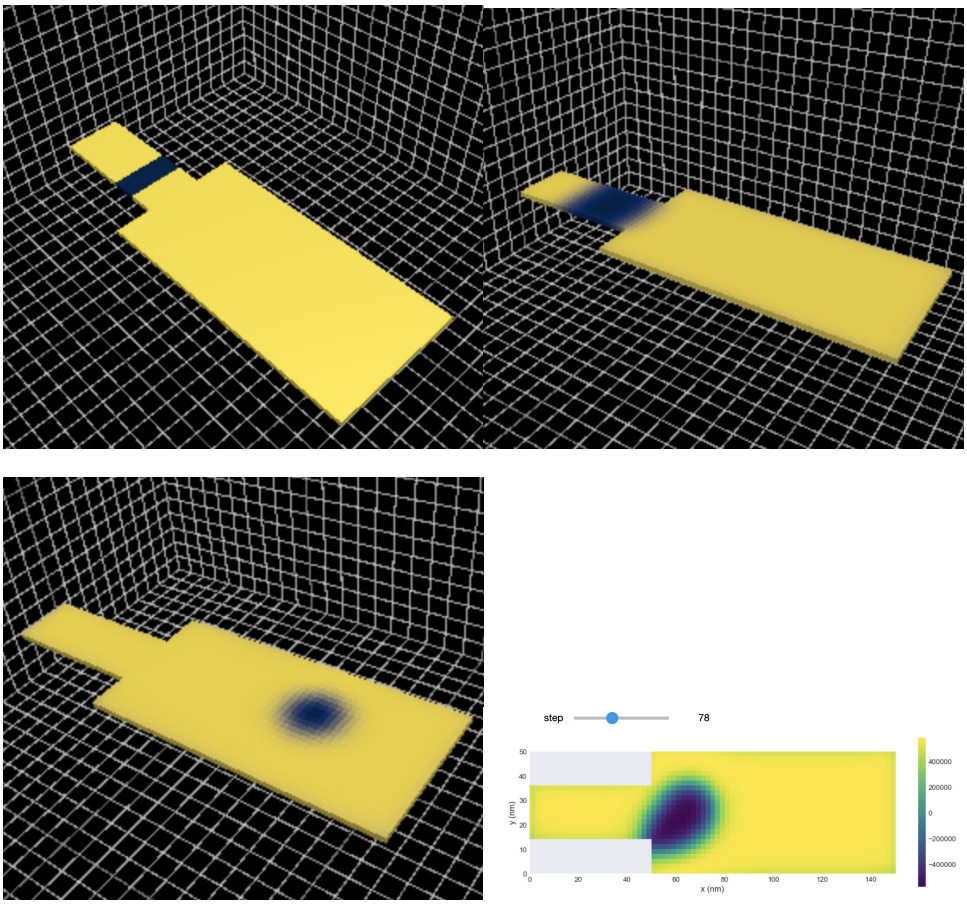



FIGURE 50. Shyrmion driven by STT.

We define mesh in cuboid through corner points `p1` and `p2`, and discretization cell size `cell`. To define periodic boundary conditions, we pass an additional argument `bc`. Let us assume we want the periodic boundary conditions in $x$ and $y$ directions.

```
region = df.Region(p1=(-50e-9, -50e-9, 0), p2=(50e-9, 50e-9, 10e-9))
mesh = df.Mesh(region=region, cell=(5e-9, 5e-9, 5e-9), bc='xy')
```

Now, we can define the system object:

```
system = mm.System(name='skyrmion')

system.energy = (mm.Exchange(A=1.6e-11)
    + mm.DMI(D=4e-3, crystalclass='Cnv')
    + mm.UniaxialAnisotropy(K=0.51e6, u=(0, 0, 1))
    + mm.Zeeman(H=(0, 0, 0.2e5)))

Ms = 1.1e6

def m_init(points):
  x, y, z = points
```

```
  if (x**2 + y**2)**0.5 < 10e-9:
    return (0, 0, -1)
  else:
    return (0, 0, 1)


# create system with above geometry and initial magnetisation
system.m = df.Field(mesh, dim=3, value=m_init, norm=Ms)
```

Now, we can define the runner object. There are three main runners you can use:

(a) Tcl runner: if we want to point ubermag to the particular `oommf.tcl` file;
(b) Exe runner: if we have OOMMF executable;
(c) Docker runner: if we want to run simulations inside Docker container.

```
tcl_runner = mc.oommf.TclOOMMFRunner(oommf_tcl='path/to/my/oommf.tcl')
```

```
exe_runner = mc.oommf.ExeOOMMFRunner(oommf_exe='oommf')
```

```
docker_runner = mc.oommf.DockerOOMMFRunner(image='ubermag/oommf')
```

**Remark 1.** *IMPORTANT: On Windows, if OOMMF does not support some energy terms, choosing runner happens automatically in the background and requires no assistance from the user. However, you can still be explicit and tell ubermag how you want to run the simulation.*

Now, when we drive the system, we can pass the runner to the `drive` method and get the result in Figure 51:

```
# NBVAL_SKIP
md = mc.MinDriver()
md.drive(system, runner=docker_runner)

# Plot relaxed configuration: vectors in z-plane
system.m.plane('z').z.mpl()
```

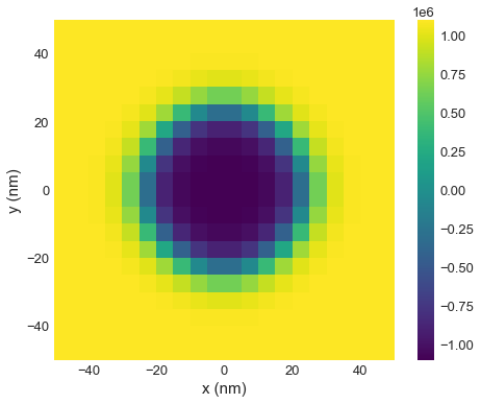



FIGURE 51. The relaxed magnetization.

The first time we run the simulation, it is going to take some time for docker to pull an image from the cloud, but after that, the image will be known by docker, so there will be no delays for any further runs.

2.2.2. *Multiple energy terms of the same class.* Here we demonstrate how to define multiple energy terms of the same class in the energy equation. For the sample, we choose a one-dimensional chain of magnetic moments.

```
import random
import oommfc as mc
import discretisedfield as df
import micromagneticmodel as mm

p1 = (-10e-9, 0, 0)
p2 = (10e-9, 1e-9, 1e-9)
cell = (1e-9, 1e-9, 1e-9)
region = df.Region(p1=p1, p2=p2)
mesh = df.Mesh(region=region, cell=cell)
```

The mesh shown in Figure 52 is

```
mesh.k3d()
```

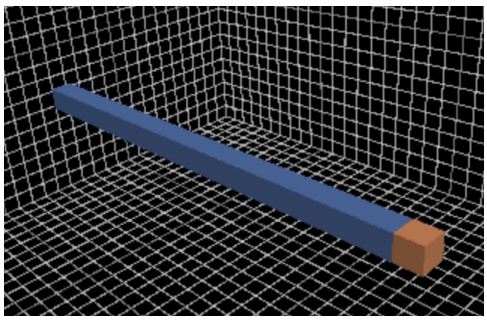


FIGURE 52. The size of sample.

Let us say that the system has an energy equation, which consists of two Zeeman energy terms.

```
H1 = (0, 0, 1e6)  # applied to z-direction
H2 = (1e6, 0, 0)  # applied to x-direction

system = mm.System(name='multiple_terms')
```

Now, if we try to add two Zeeman energy terms, we get an exception raised.

```
try:
  system.energy = mm.Zeeman(H=(0, 0, 1e5)) + mm.Zeeman(H=(0, 1e5, 0))
except ValueError:
  print('Exception raised.')
```

This is because different energy terms must have different names, so they can be uniquely identified. So, we have to give names to our energy terms:

```
system.energy = mm.Zeeman(H=H1, name='zeeman1') + mm.Zeeman(H=H2, name
                                    ='zeeman2')
```

We need to define the system's magnetization (`system.m`). We are going to make it random with $M_\mathrm{s} = 8 \times 10^5\,\mathrm{Am}^{-1}$.

```
random.seed(1)  # guarantee the result same at each time

Ms = 8e5  # saturation magnetisation (A/m)

def m_fun(point):
  return [2*random.random()-1 for i in range(3)]

system.m = df.Field(mesh, dim=3, value=m_fun, norm=Ms)
```

Now, we can minimize the system's energy:

```
md = mc.MinDriver()
md.drive(system)
```

We expect that now all magnetic moments are aligned parallel or antiparallel to the anisotropy axis (in the $z$-direction). in Figure 53

```
system.m.plane('y').mpl(figsize=(10, 3), scalar_clim=[-0.1, 0.1])
```

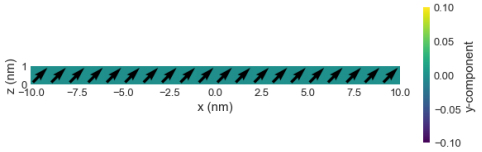



FIGURE 53. The relaxed magnetization.

We can see that magnetization is aligned with the sum of fields $H1+H2$. Finally, let us have a look at the table:

```
system.table.data
```

We can see that energy terms are marked with the names we gave to energy terms when we defined the energy equation.

2.2.3. *RKKY energy term.* In this tutorial we demonstrate how to use RKKY energy term in ubermag simulations. We start by importing the modules we are going to use. Please note that we are importing `random` as well, so we can initialize our magnetization as a random state.

```
import random
import discretisedfield as df
import micromagneticmodel as mm
import oommfc as mc

random.seed(2)  # this way we ensure reproducibility
```

Our sample consists of three subregions. More precisely, two magnetic, and a non-magnetic spacer in Figure 54.

```
p1 = (0, 0, 0)
p2 = (60e-9, 60e-9, 22e-9)
region = df.Region(p1=p1, p2=p2)
subregions={'bottom': df.Region(p1=(0, 0, 0), p2=(100e-9, 100e-9, 10e-
                                9)),
    'spacer': df.Region(p1=(0, 0, 10e-9), p2=(100e-9, 100e-9, 12e-9)),
```

```
    'top': df.Region(p1=(0, 0, 12e-9), p2=(100e-9, 100e-9, 22e-9))}
mesh = df.Mesh(region, n=(20, 20, 11), subregions=subregions)

mesh.k3d_subregions()
```

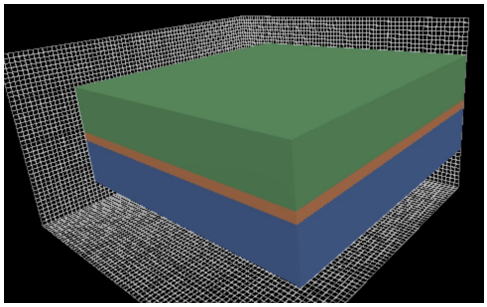


FIGURE 54. The shape of sample.

Our energy equation consists of exchange, uniaxial anisotropy and RKKY energy terms. In order to define RKKY interaction, we have to pass `sigma` and `sigma2` parameters, as well as subregions between which RKKY occurs. More precisely, by passing two subregions, two closest mutually facing surfaces are going to be identified automatically. In addition, please note that we set up norm of the field using a dictionary and the value using a `lambda` function. We could have done the same thing by writing full Python functions, but here we want to show that there are many different ways how a field can be defined.

```
system = mm.System(name='rkky')
system.energy = (mm.Exchange(A=1e-12) +
    mm.RKKY(sigma=-1e-4, sigma2=0, subregions=['bottom', 'top']) +
    mm.UniaxialAnisotropy(K=1e5, u=(1, 0, 0)))

norm = {'bottom': 8e6, 'top': 8e6, 'spacer': 0}
system.m = df.Field(mesh, dim=3, value=lambda point: [2*random.random
                                    ()-1 for i in range(3)], norm=norm)
```

The initial magnetization in Figure 55 is:

```
system.m.plane('y').mpl(figsize=(15, 4))
```

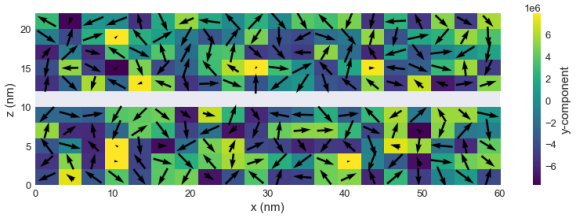



FIGURE 55. The initialized magnetization.

And the energy equation:

```
system.energy
```

Now we can relax the system, and plot its magnetization in Figure 56.

```
md = mc.MinDriver()
md.drive(system)
```

```
system.m.plane('y').mpl(figsize=(12, 3))
```

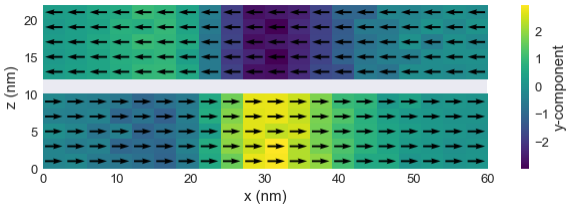



FIGURE 56. The relaxed magnetization.

We can see that two layers are "antiferromagnetically coupled" because we used negative $\sigma$.

2.2.4. *Time-varying field.* In this tutorial, we introduce how a time-dependent external magnetic field can be defined in energy equation. We start by importing the modules we are going to use:

```
import oommfc as mc
import discretisedfield as df
import micromagneticmodel as mm
```

For the sample, we choose a one-dimensional chain of magnetic moments.

```
p1 = (-10e-9, 0, 0)
p2 = (10e-9, 1e-9, 1e-9)
cell = (1e-9, 1e-9, 1e-9)
region = df.Region(p1=p1, p2=p2)
mesh = df.Mesh(region=region, cell=cell)
```

The mesh is:

```
mesh.k3d()
```

Now, we can define the system object and add a time-dependent sine-wave field to our energy equation. We need to pass:

(a) Field $H$ which is multiplied by time-dependent function at each time step; wave shape `wave='sin'`;
(b) frequency $f$, and time shift $t_0$.

Accordingly, the time-dependent Zeeman field is then:

$$\mathbf{H}(t) = \mathbf{H}_{\text{amp}} \sin(2\pi f(t - t_0))$$

.

```
system = mm.System(name='time_dependent_field')

system.energy = mm.Exchange(A=1.6e-11) + mm.Zeeman(H=(1e6, 2e6, 3e6),
                                        wave='sin', f=1e9, t0=1e-9)
system.dynamics = mm.Precession(gamma0=mm.consts.gamma0) + mm.Damping(
                                        alpha=1e-5)

system.m = df.Field(mesh, dim=3, value=(0, 0, 1), norm=1.1e6)
```

Now, we can run the simulation using `TimeDriver`:

```
td = mc.TimeDriver()
td.drive(system, t=5e-9, n=200)
```

In the system table Figure 57, there are columns with field values:

```
system.table.data
```

| | E | E_calc_count | max_dm/dt | dE/dt | delta_E | E_exchange | max_spin_ang_exchange | max_spin_ang_ |
|---|---|---|---|---|---|---|---|---|
| **0** | -1.297563e-20 | 25.0 | 4.434031e+03 | -5.147568e-10 | -1.812700e-21 | 0.0 | 0.0 | |
| **1** | -2.562001e-20 | 56.0 | 8.766104e+03 | -4.954602e-10 | -1.204656e-21 | 0.0 | 0.0 | |
| **2** | -3.759093e-20 | 105.0 | 1.290846e+04 | -4.636140e-10 | -1.221901e-21 | 0.0 | 0.0 | |
| **3** | -4.863542e-20 | 184.0 | 1.673341e+04 | -4.207088e-10 | -6.112468e-22 | 0.0 | 0.0 | |
| **4** | -5.848036e-20 | 299.0 | 2.014752e+04 | -3.675974e-10 | -2.915541e-22 | 0.0 | 0.0 | |
| **...** | ... | ... | ... | ... | ... | ... | ... | |
| **195** | 4.436379e-20 | 20614.0 | 1.906705e+04 | -3.837992e-10 | -2.268306e-22 | 0.0 | 0.0 | |
| **196** | 3.422674e-20 | 20687.0 | 1.474584e+04 | -4.221479e-10 | -5.046823e-22 | 0.0 | 0.0 | |
| **197** | 2.328011e-20 | 20742.0 | 1.004530e+04 | -4.502211e-10 | -1.083878e-21 | 0.0 | 0.0 | |
| **198** | 1.178128e-20 | 20779.0 | 5.087140e+03 | -4.673789e-10 | -2.358762e-21 | 0.0 | 0.0 | |
| **199** | 6.088592e-34 | 20804.0 | 2.629445e-10 | -4.731610e-10 | -2.217703e-21 | 0.0 | 0.0 | |

FIGURE 57. The data table.

We plot the time dependent field in Figure 58 by the code as

```
system.table.mpl(yaxis=['Bx_zeeman', 'By_zeeman', 'Bz_zeeman'])
```

Similarly, we can define a cardinal sine wave ("sinc pulse"). We need to pass:

(a) Field `H` which is multiplied by time-dependent function at each time step; wave shape `wave='sinc'`;
(b) cut-off frequency `f`, and time shift `t0`.

Accordingly, the time-dependent Zeeman field shown in Figure 59 is then:

$$\mathbf{H}(t) = \mathbf{H}_{\text{amp}}\text{sinc}(2\pi f_c(t-t_0)) = \mathbf{H}_{\text{amp}}\frac{\text{sinc}(2\pi f_c(t-t_0))}{2\pi f_c(t-t_0)}$$

.

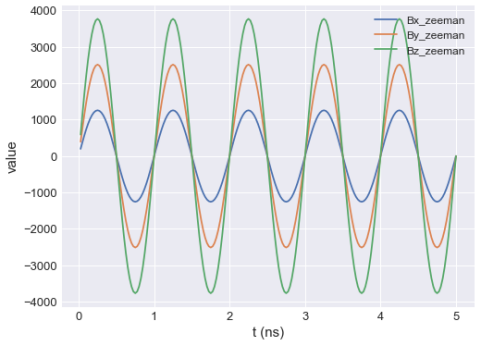



FIGURE 58. The time dependent field.

```
system = mm.System(name='time_dependent_field')

system.energy = (mm.Exchange(A=1.6e-11) +
    mm.Zeeman(H=(1e6, 2e6, 3e6), wave='sinc', f=1e9, t0=5e-9))
system.dynamics = mm.Precession(gamma0=mm.consts.gamma0) + mm.Damping(
                                    alpha=1e-5)

system.m = df.Field(mesh, dim=3, value=(0, 0, 1), norm=1.1e6)

td = mc.TimeDriver()
td.drive(system, t=10e-9, n=200)

system.table.mpl(yaxis=['Bx_zeeman', 'By_zeeman', 'Bz_zeeman'])
```

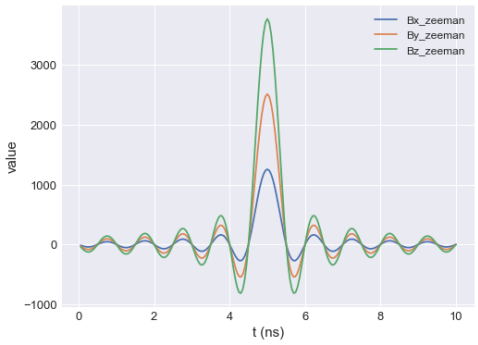



FIGURE 59. The time dependent field.

2.2.5. *Negative exchange energy constant.* In this tutorial, we show how to set up a negative value of exchange energy constant between subregions in the mesh. We start by importing the modules we are going to use. We import `random`, because we plan to initialize our magnetization as a random state.

```
import random
import discretisedfield as df
import micromagneticmodel as mm
```

```
import oommfc as mc
```

In order to make sure we always have the same random state generated, we set up the seed.

```
random.seed(2)
```

Our sample in Figure 60 consists of three identical magnetic layers, which are (for some reason) coupled antiferomagnetically.

```
p1 = (0, 0, 0)
p2 = (60e-9, 60e-9, 30e-9)
region = df.Region(p1=p1, p2=p2)
subregions={'r1': df.Region(p1=(0, 0, 0), p2=(100e-9, 100e-9, 10e-9)),
    'r2': df.Region(p1=(0, 0, 10e-9), p2=(100e-9, 100e-9, 20e-9)),
    'r3': df.Region(p1=(0, 0, 20e-9), p2=(100e-9, 100e-9, 30e-9))}
mesh = df.Mesh(region, n=(20, 20, 9), subregions=subregions)

mesh.k3d_subregions()
```

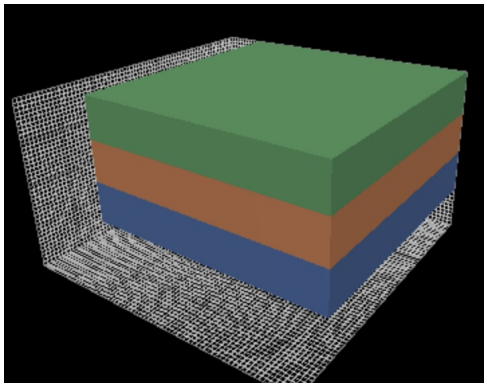


FIGURE 60. The shape of sample.

We use a dictionary to set up parameter $A$. For individual subregions, we use `r1`, `r2`, and `r3`, whereas to define $A$ between different subregions, we put a colon in dictionary key - for instance, `r1:r2`.

```
system = mm.System(name='negative_A')

A = 1e-12
Adict = {'r1': A, 'r2': A, 'r3': A, 'r1:r2': -A, 'r2:r3': -A}

system.energy = mm.Exchange(A=Adict) + mm.UniaxialAnisotropy(K=1e4, u=
                                    (1, 0, 0))
```

As we mentioned previously, we are going to initialize the system using a random state in Figure 61.

```
def m_random(point):
  return [2*random.random()-1 for i in range(3)]

system.m = df.Field(mesh, dim=3, value=m_random, norm=8e6)

system.m.plane('y').mpl(figsize=(12, 3))
```

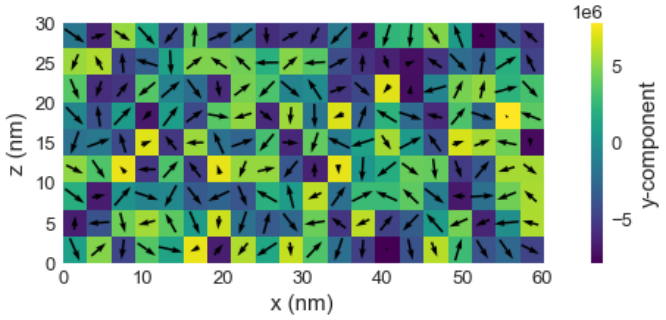



FIGURE 61. The initialized random magnetization.

Now, we can minimize the system's energy and plot its magnetization in Figure 62.

```
md = mc.MinDriver()
md.drive(system)

system.m.plane('y').mpl(figsize=(12, 3))
```

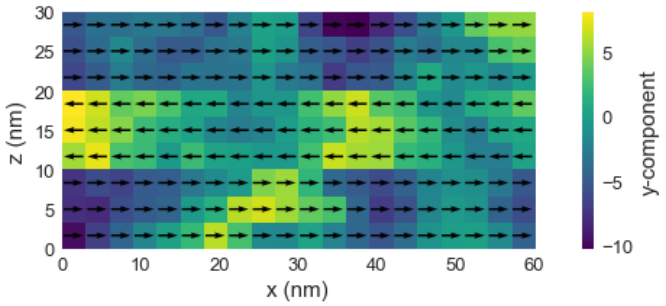



FIGURE 62. The relaxed magnetization.

We can see that all three regions are uniformly magnetized, but mutually coupled antiferromagnetically.

2.2.6. *Energy term computations.* Sometimes it is necessary to compute certain properties of energy terms, such as energy, effective field, or energy density, without driving the system and updating magnetization field.

As usual, we start by importing the modules we are going to use:

```
import oommfc as mc
import discretisedfield as df
import micromagneticmodel as mm
```

We define a cube mesh with edge length 10 nm and cell discretization edge 1 nm shown in Figure 63.

```
region = df.Region(p1=(0, 0, 0), p2=(10e-9, 10e-9, 10e-9))
mesh = df.Mesh(region=region, n=(10, 10, 10))
mesh.k3d()
```

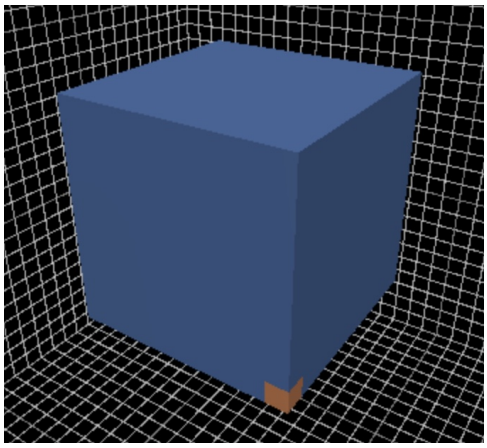


FIGURE 63. The shape of sample.

Now we define the system object and its energy equation.

```
system = mm.System(name='energy_computations')

A = 1e-11  # exchange energy constant (J/m)
H = (0.1/mm.consts.mu0, 0, 0)  # external magnetic field (A/m)
K = 1e3  # uniaxial anisotropy constant (J/m3)
u = (1, 1, 1)  # uniaxial anisotropy axis

system.energy = (mm.Exchange(A=A) +
    mm.Demag() +
    mm.Zeeman(H=H) +
    mm.UniaxialAnisotropy(K=K, u=u))

system.energy
```

We will now initialize the magnetization in the $(0, 0, 1)$ direction with $M_s = 8 \times 10^5\,\mathrm{A/m}$ and relax the magnetization.

```
Ms = 8e5  # saturation magnetization (A/m)

system.m = df.Field(mesh, dim=3, value=(0, 0, 1), norm=Ms)
```

All computations are performed using `mc.compute`, where `mc` is the micromagnetic calculator we choose at import. `compute` function takes two arguments:

(1) Property we want to compute;
(2) System object.

For instance, total effective field in Figure 64 is:

```
Heff = mc.compute(system.energy.effective_field, system)
Heff.plane('x').mpl()
```

Similarly, the effective field of an individual energy term is:

```
Hex_eff = mc.compute(system.energy.exchange.effective_field, system)
```

Because we initialized the system with the uniform state, we expect this effective field to be zero.

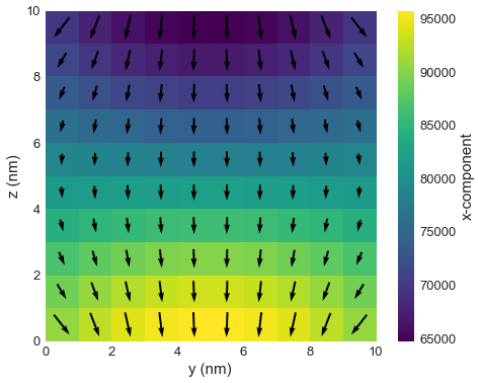



FIGURE 64. The total effective field.

```
Hex_eff.average
```

The energy density in Figure 65 is:

```
w = mc.compute(system.energy.density, system)
w.plane('x').mpl()
```

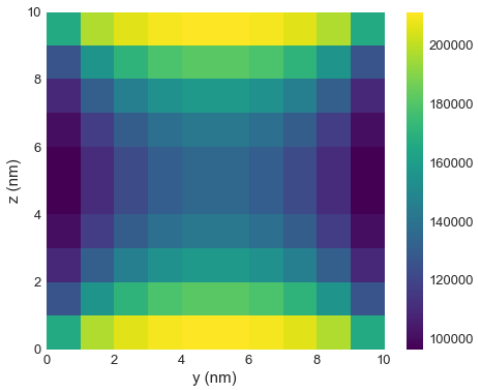



FIGURE 65. The energy density.

Similarly, the energy (volume integral of energy density) is:

```
E = mc.compute(system.energy.energy, system)
print(f'The energy of the system is {E} J.')
```

Now, we can relax the system.

```
md = mc.MinDriver()
md.drive(system)
```

If we now compute the energy, we can show that the energy decreased:

```
E = mc.compute(system.energy.energy, system)
print(f'The energy of the system is {E} J.')
```

The relaxed magnetization in Figure 66 is:

```
system.m.plane('x').mpl()
```

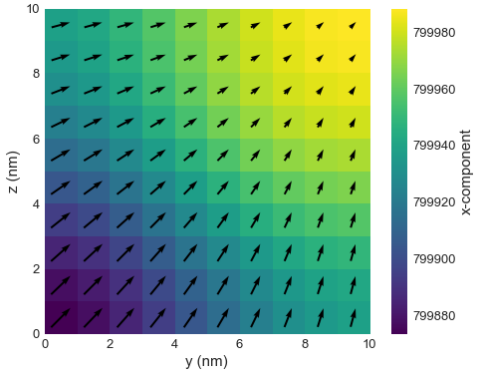



FIGURE 66. The relaxed magnetization.

The exchange energy is:

```
mc.compute(system.energy.exchange.energy, system)
```

We can also check the sum of all individual energy terms and check if it the same as the total energy.

```
total_energy = 0
for term in system.energy:
  total_energy += mc.compute(term.energy, system)

E = mc.compute(system.energy.energy, system)
# A simpler way:
# E = sum([mc.compute(getattr(term, 'energy'), system) for term in
                                        system.energy])

print(f'The sum of energy terms is {total_energy} J.')
print(f'The energy of the system is {E} J.')
```

2.2.7. *Field operations 1.* In this tutorial, we show some mathematical operations that can be performed on field objects in Ubermag. For that, we are going to use a skyrmion example in Figure 67.

```
import oommfc as mc
import discretisedfield as df
import micromagneticmodel as mm

region = df.Region(p1=(-50e-9, -50e-9, 0), p2=(50e-9, 50e-9, 10e-9))
mesh = df.Mesh(region=region, cell=(5e-9, 5e-9, 5e-9), bc='xy')

system = mm.System(name='skyrmion')

system.energy = (mm.Exchange(A=1.6e-11)
    + mm.DMI(D=4e-3, crystalclass='Cnv')
    + mm.UniaxialAnisotropy(K=0.2e6, u=(0, 0, 1))
    + mm.Zeeman(H=(0, 0, 1e5)))
```

```
def m_init(point):
  x, y, z = point
  if (x**2 + y**2)**0.5 < 10e-9:
    return (0, 0, -1)
  else:
    return (0, 0, 1)

system.m = df.Field(mesh, dim=3, value=m_init, norm=1.1e6)

# minimize the energy
md = mc.MinDriver()
md.drive(system)

# Plot relaxed configuration: vectors in z-plane
system.m.plane('z').mpl()
```

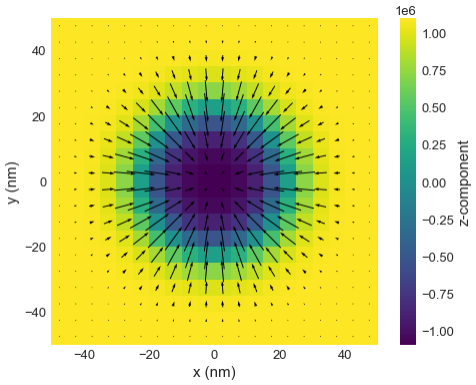



FIGURE 67. The relaxed magnetization (cnv skrymion).

The magnetization field is in `system.m`. To access a numpy array of a magnetization field:

```
system.m.array
```

This exposes the field to `numpy` and all operations present in numpy can be used. However, there are a lot of convenience functions, which are part of the `Field` object.

To get an individual component in Figure 68:

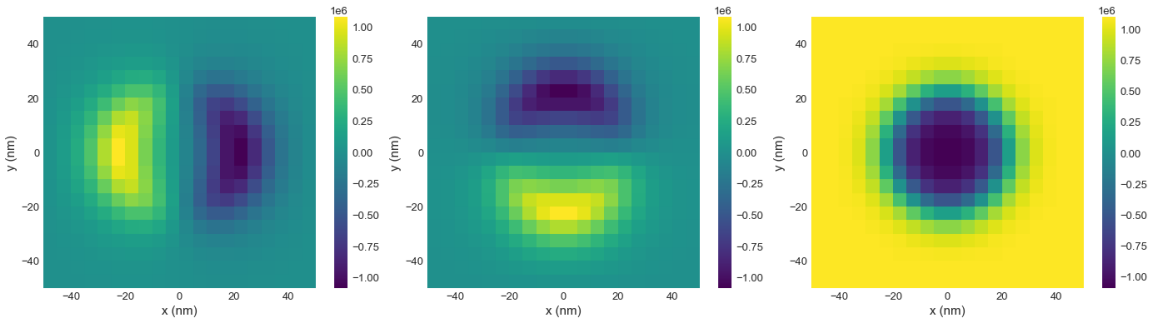
```
system.m.x
system.m.x.plane('z').mpl()

system.m.y
system.m.y.plane('z').mpl()

system.m.z
system.m.z.plane('z').mpl()
```

The magnetization is by default not normalized.

```
system.m.average
```




(A) $\boldsymbol{m}_x$ (B) $\boldsymbol{m}_y$ (C) $\boldsymbol{m}_z$

FIGURE 68. The individual magnetization $\boldsymbol{m}_x$, $\boldsymbol{m}_y$ and $\boldsymbol{m}_z$.

To get normalized magnetization:

$$\mathbf{m} = \frac{\mathbf{M}}{M_{\mathrm{s}}}$$

.

```
system.m.orientation
system.m.orientation.average
```

Just like we can get the normalized magnetization, we can get the norm itself.

```
abs(system.m)
```

**Remark 2** (Algebra operations)**.** *Let us define two fields:*

```
region = df.Region(p1=(0, 0, 0), p2=(10e-9, 10e-9, 10e-9))
mesh = df.Mesh(region=region, n=(10, 10, 10))

f1 = df.Field(mesh, dim=3, value=(1, 1, 0))
f2 = df.Field(mesh, dim=3, value=(2, 1, 3))
```

```
f1.average
f2.average
f1 + f2
(f1 + f2).average
f1 - f2
(f1 - f2).average
```

*Basic multiplication is not defined between vector fields. In that case, we perform either dot or cross product, which we are going to discuss later.*

```
# NBVAL_SKIP
f1 * f2  # both are vector fields, will not work

f1 / f2  # both are vector fields, will not work
```

*Scalar with vector field:*

```
f1.x * f2
f1.x * f2.y

f1 / f2.x
```

```
f1.x ** 2
f1 += f2
f1 -= f2
f1 *= f2.x
f2 /= f2.y
```

*Scalar field divided by vector field is not allowed:*

```
# NBVAL_SKIP
f2.x / f1 # will not work
f1 ** 2 # will not work
```

*Vector products: As the title says, these products are applied between vector fields only. Dot product is implemented through `@` operator:*

```
f1 @ f2
```

*Cross product between vector fields is performed using and operator:*

```
f1 & f2
```

*Vector calculus:*

```
f1.derivative('x')  # Directional derivative defined on both scalar and
                                    vector fields.
f1.x.grad # Gradient defined on scalar fields.
f1.div  # Divergence defined on vector fields;
f1.curl # Curl defined on vector fields;
f1.laplace  # Laplace operator defined on both vector and scalar fields
                                   ;
f1.x.laplace

f1.volume_integral  # Volume integral
f1.plane('z').surface_integral  # Surface integral
f1.x.grad.div.grad.curl.y.grad.volume_integral  # Operation pipelines
```

*Here we implement skyrmion number calculations using operations on fields:*

$$S = \frac{1}{4\pi}\int \mathbf{m}\cdot\left(\frac{\partial \mathbf{m}}{\partial x}\times\frac{\partial \mathbf{m}}{\partial y}\right)dxdy$$

```
import math

m = system.m.orientation.plane('z')
S = (m @ (m.derivative('x') & m.derivative('y'))).surface_integral / (4
                                   *math.pi)
S
```

*Or using Ubermag function:*

```
m.topological_charge()
```

*Using Berg-Luescher method*

```
m.topological_charge(method='berg-luescher')  # much more better
```

2.2.8. *Field operations 2.* In this notebook, we show (and compare) computing individual energy terms:

```
import discretisedfield as df
import micromagneticmodel as mm
import oommfc as mc
```

As an example, we use skyrmion magnetization field in Figure 69.

```
# Geometry
L = 100e-9
thickness = 5e-9
cell = (5e-9, 5e-9, 5e-9)
p1 = (-L/2, -L/2, 0)
p2 = (L/2, L/2, thickness)
region = df.Region(p1=p1, p2=p2)
mesh = df.Mesh(region=region, cell=cell, bc='xy')

# Parameters
Ms = 3.84e5
A = 8.78e-12
D = 1.58e-3
K = 1e4
u = (0, 0, 1)
H = (0, 0, 1e5)
system = mm.System(name='skyrmion')
system.energy = mm.Exchange(A=A) + mm.DMI(D=D, crystalclass='T') + mm.
                                   Zeeman(H=H) + mm.UniaxialAnisotropy
                                   (K=K, u=u)

def m_initial(point):
  x, y, z = point
  if x**2 + y**2 < (L/4)**2:
    return (0, 0, -1)
  else:
    return (0, 0, 1)

system.m = df.Field(mesh, dim=3, value=m_initial, norm=Ms)

md = mc.MinDriver()
md.drive(system)

system.m.plane('z').mpl(figsize=(9, 7))
```

The normalized magnetization for

$$\mathbf{m} = \frac{\mathbf{M}}{M_{\mathrm{s}}}$$

can be done by

```
m = system.m.orientation

m.plane('z').mpl(scalar_clim=(-1, 1))
```

As for Zeeman, we have the representation in Figure 70. The code for energy density would be

```
wdf = - mm.consts.mu0 * Ms * m @ H
wmc = mc.compute(system.energy.zeeman.density, system)
wdf.allclose(wmc)
```

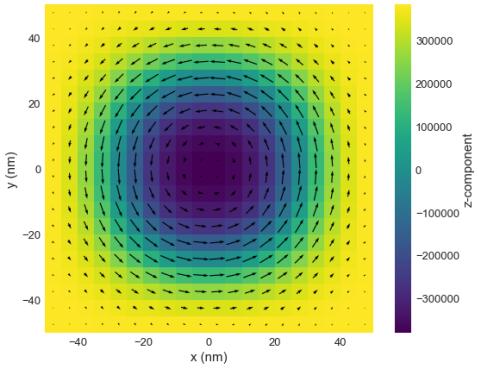



FIGURE 69. The relaxed magnetization.

| property | equation in continuous form | discretisedfield form |
|---|---|---|
| Energy density | $w = -\mu_0 M_\mathrm{s} \mathbf{m} \cdot \mathbf{H}$ | – mu0 * Ms * m @ H |
| Energy | $E = -\int_V \mu_0 M_\mathrm{s} \mathbf{m} \cdot \mathbf{H} dV$ | (– mu0 * Ms * m @ H).volume_integral |
| Effective field | $\mathbf{H}_\mathrm{eff} = \mathbf{H}$ | H |

FIGURE 70. Zeeman property.

The energy would be

```
Edf = (- mm.consts.mu0 * Ms * m @ H).volume_integral
Emc = mc.compute(system.energy.zeeman.energy, system)
print(f'df: {Edf}')
print(f'mc: {Emc}')
print(f'rerr: {abs(Edf-Emc)/Edf * 100} %')
```

The effective field would be

```
Hdf = df.Field(mesh, dim=3, value=H)
Hmc = mc.compute(system.energy.zeeman.effective_field, system)
Hdf.allclose(Hmc)
```

As for uniaxial anisotropy, we have the representation in Figure 71. The energy

| property | equation in continuous form | discretisedfield form |
|---|---|---|
| Energy density | $w = K\lvert\mathbf{m} \times \mathbf{u}\rvert^2$ | K * abs(m & u)**2 |
| Energy | $E = \int_V K\lvert\mathbf{m} \times \mathbf{u}\rvert^2 dV$ | (K * abs(m & u)**2).volume_integral |
| Effective field | $\mathbf{H}_\mathrm{eff} = \frac{2K}{\mu_0 M_\mathrm{s}}(\mathbf{m} \cdot \mathbf{u})\mathbf{u}$ | 2 * K / (mu0 * Ms) * (m @ u) * u |

FIGURE 71. Uniaxial anisotropy property.

density would be

```
wdf = K * abs(m & u)**2
wmc = mc.compute(system.energy.uniaxialanisotropy.density, system)
wdf.allclose(wmc)
```

The energy would be

```
Edf = (K * abs(m & u)**2).volume_integral
Emc = mc.compute(system.energy.uniaxialanisotropy.energy, system)
print(f'df: {Edf}')
print(f'mc: {Emc}')
print(f'rerr: {abs(Edf-Emc)/Edf * 100} %')
```

The effective field would be

```
Hdf = 2 * K / (mm.consts.mu0 * Ms) * (m @ u) * u
Hmc = mc.compute(system.energy.uniaxialanisotropy.effective_field,
                                        system)
Hdf.allclose(Hmc)
```

As for exchange energy, we have the representation in Figure 72. The exchange

| property | equation in continuous form | discretisedfield form |
|---|---|---|
| Energy density | $w = -A\mathbf{m} \cdot \nabla^2\mathbf{m}$ | – A * m @ m.laplace |
| Energy | $E = -\int_V A\mathbf{m} \cdot \nabla^2\mathbf{m} dV$ | (– A * m @ m.laplace).volume_integral |
| Effective field | $\mathbf{H}_{\text{eff}} = \frac{2A}{\mu_0 M_s}\nabla^2\mathbf{m}$ | 2 * A / (mu0 * Ms) * m.laplace |

FIGURE 72. Exchange property.

energy density would be

```
wdf = - A * m @ m.laplace
wmc = mc.compute(system.energy.exchange.density, system)
wdf.allclose(wmc)
```

The exchange energy would be

```
Edf = (- A * m @ m.laplace).volume_integral
Emc = mc.compute(system.energy.exchange.energy, system)
print(f'df: {Edf}')
print(f'mc: {Emc}')
print(f'rerr: {abs(Edf-Emc)/Edf * 100} %')
```

The exchange effective field would be

```
Hdf = 2 * A / (mm.consts.mu0 * Ms) * m.laplace
Hmc = mc.compute(system.energy.exchange.effective_field, system)
Hdf.allclose(Hmc)
```

As for DMI (T), we have the representation in Figure 73. The DMI energy density would be

```
wdf = D * m @ m.curl
wmc = mc.compute(system.energy.dmi.density, system)
wdf.allclose(wmc)
```

| property | equation in continuous form | discretisedfield form |
|---|---|---|
| Energy density | $w = \boldsymbol{D}\mathbf{m} \cdot (\nabla \times \mathbf{m})$ | D * m @ m.curl |
| Energy | $E = \int_V \boldsymbol{D}\mathbf{m} \cdot (\nabla \times \mathbf{m}) dV$ | (D * m @ m.curl).volume_integral |
| Effective field | $\mathbf{H}_{\text{eff}} = -\frac{2D}{\mu_0 M_s}(\nabla \times \mathbf{m})$ | – 2 * D / (mu0 * Ms) * m.curl |

FIGURE 73. DMI property.

The DMI energy would be

```
Edf = (D * m @ m.curl).volume_integral
Emc = mc.compute(system.energy.dmi.energy, system)
print(f'df: {Edf}')
print(f'mc: {Emc}')
print(f'rerr: {abs(Edf-Emc)/Edf * 100} %')
```

The DMI effective field would be

```
Hdf = - 2 * D / (mm.consts.mu0 * Ms) * m.curl
Hmc = mc.compute(system.energy.dmi.effective_field, system)
Hdf.allclose(Hmc)
```

Here we try to implement an (oversimplified) micromagnetic calculator presented in Figure 74 we had a look at in the first session: We start by defining some basic

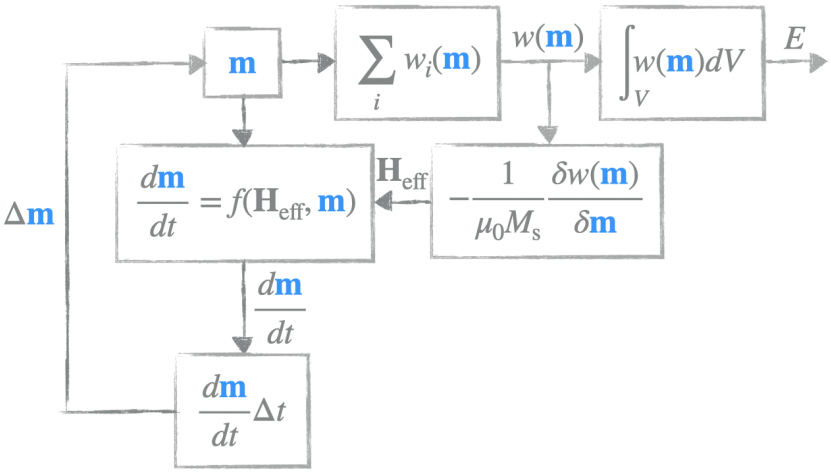



FIGURE 74. Micromagnetic calculator.

parameters:

```
# Geometry
L = 100e-9
thickness = 5e-9
cell = (5e-9, 5e-9, 5e-9)
p1 = (-L/2, -L/2, 0)
```

```
p2 = (L/2, L/2, thickness)
region = df.Region(p1=p1, p2=p2)
mesh = df.Mesh(region=region, cell=cell, bc='xy')

# Material parameters
Ms = 3.84e5
A = 8.78e-12
D = 1.58e-3
K = 1e4
u = (0, 0, 1)
H = (0, 0, 1e5)
alpha = 1

def m_initial(point):
  x, y, z = point
  if x**2 + y**2 < (L/4)**2:
    return (0, 0, -1)
  else:
    return (0, 0, 1)

# Magnetisation field (M=Ms*m)
M = df.Field(mesh, dim=3, value=m_initial, norm=Ms)

M.plane('z').z.mpl()
```

The result presented in Figure 75. Up to this point, everything should be exactly

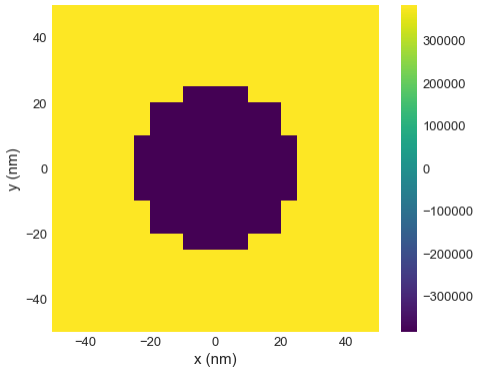



FIGURE 75. magnetization field.

the same as we saw previously.

In the next step, we are going to implement functions which we are going to use to compute effective field and magnetization time-derivative. The effective field we want is:

$$\mathbf{H}_{\text{eff}} = f(\mathbf{m}) = \frac{2A}{\mu_0 M_s}\nabla^2\mathbf{m} - \frac{2D}{\mu_0 M_s}(\nabla \times \mathbf{m}) + \frac{2K}{\mu_0 M_s}(\mathbf{m}\cdot\mathbf{u})\mathbf{u} + \mathbf{H}$$

.

Dynamics equation we are going to use consists only of Damping term (in order to make simulations faster):

$$\frac{d\mathbf{m}}{dt} = f(\mathbf{m}, \mathbf{H}_{\text{eff}}) = -\frac{\gamma_0 \alpha}{1+\alpha^2}\mathbf{m} \times (\mathbf{m} \times \mathbf{H}_{\text{eff}})$$

.

```
def Heff_function(m):
  return (2*A / (mm.consts.mu0*Ms) * m.laplace -
    2*D / (mm.consts.mu0 * Ms) * m.curl +
    2*K / (mm.consts.mu0*Ms) * (m @ u) * u +
    df.Field(mesh, dim=3, value=H))

def dmdt_function(m, Heff):
  return -(mm.consts.gamma0*alpha)/(1+alpha**2) * m & (m & Heff)
```

Now, we can try to perform the time integration, so that at each step we update our magnetization:

$$\mathbf{m}_{i+1} = \mathbf{m}_i + \frac{d\mathbf{m}_i}{dt}\Delta t$$

.

So, our ubermag time driver would be:

```
T = 0.1e-9  # simulation time (s)
n = 100  # number of steps
dt = T/n

for i in range(n):
  m = M.orientation
  m += dmdt_function(m, Heff_function(m)) * dt
  M = df.Field(mesh, dim=3, value=m, norm=Ms)
```

Finally, we can plot the magnetization in Figure 76:

```
M.plane('z').mpl()
```

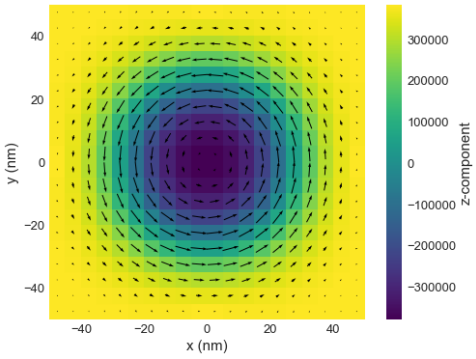



FIGURE 76. magnetization field.

2.2.9. *Saving and reading fields.* There are three main file formats to which a `discretisedfield.Field` object can be saved:

(a) [VTK](https://vtk.org/) for visualization using e.g., [ParaView](https://www.paraview.org/) or [Mayavi](https://docs.enthought.com/mayavi/mayavi/);

(b) OOMMF [Vector Field File Format (OVF)](https://math.nist.gov/oommf/doc/userguide12a5/userguide/Vector_Field_File_Format_OV.html) for exchanging fields with micromagnetic simulators;
(c) HDF5.

Let us say we have a nanosphere sample:

$$x^2 + y^2 + z^2 <= r^2$$

with $r = 5\,\text{nm}$. The space is discretized into cells with dimensions $(0.5\,\text{nm}, 0.5\,\text{nm}, 0.5\,\text{nm})$. The value of the field at $(x, y, z)$ point is $(-cy, cx, cz)$, with $c = 10^9$. The norm of the field inside the cylinder is $10^6$.

Let us first build that field in Figure 77.

```
import discretisedfield as df

r = 5e-9
cell = (0.5e-9, 0.5e-9, 0.5e-9)

region = df.Region(p1=(-r, -r, -r), p2=(r, r, r))
mesh =  df.Mesh(region=region, cell=cell)

def norm_fun(point):
  x, y, z = point
  if x**2 + y**2 + z**2 <= r**2:
    return 1e6
  else:
    return 0

def value_fun(point):
  x, y, z = point
  c = 1e9
  return (-c*y, c*x, c*z)

field = df.Field(mesh, dim=3, value=value_fun, norm=norm_fun)

field.plane('z').mpl()
```

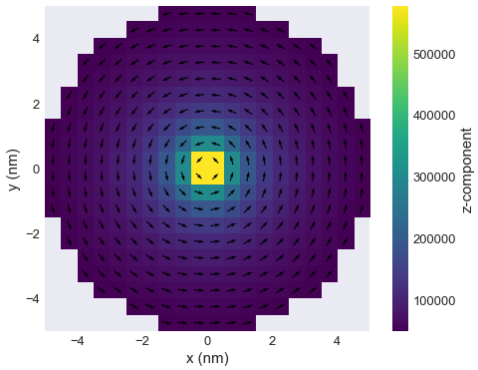



FIGURE 77. magnetization field.

As for the phase fo writing files, the main method used for saving field in different files is `discretisedfield.Field.write()`. It takes `filename` as an argument, which is a string with one of the following extensions:

(a) `.vtk` for saving in the VTK format;
(b) `.ovf`, `.omf`, `.ohf` for saving in the OVF format;
(c) `.h5` or `.hdf5` for saving an HDF5 file.

Let us firstly save the field in the VTK file.

```
vtkfilename = 'my_vtk_file.vtk'
field.write(vtkfilename)
```

Next, we can save the field in the OVF format and check whether it was created in the current directory.

```
ovffilename = 'my_omf_file.omf'
field.write(ovffilename)
```

There are three different possible representations of an OVF file: one ASCII (`txt`) and two binary (`bin4` or `bin8`). ASCII `txt` representation is a default representation when `discretisedfield.Field.write()` is called. If any different representation is required, it can be passed via `representation` argument. For example, if we want to save a binary-8 file:

```
field.write(ovffilename, representation='bin8')
```

In the same way, we can save the HDF5 file:

```
hdf5filename = 'my_omf_file.hdf5'
field.write(hdf5filename)
```

As for the phase of reading files, the method for reading field files is a class method `discretisedfield.Field.fromfile()`. By passing a `filename` argument, it reads the file and creates a `discretisedfield.Field` object. It is not required to pass the representation of the OVF file to the `discretisedfield.Field.fromfile()` method, because it can retrieve it from the content of the file.

VTK file:

```
field_read = df.Field.fromfile(vtkfilename)
field_read.plane('z').mpl()  # to compare with the field we wrote into
                                the file
```

OVF file:

```
field_read = df.Field.fromfile(ovffilename)
field_read.plane('z').mpl()
```

HDF5 file:

```
field_read = df.Field.fromfile(hdf5filename)
field_read.plane('z').mpl()
```

Convert one file format to another file format:

```
ovffile = 'somefile.omf'
df.Field.fromfile(ovffile).write('somefile.hdf5')
```

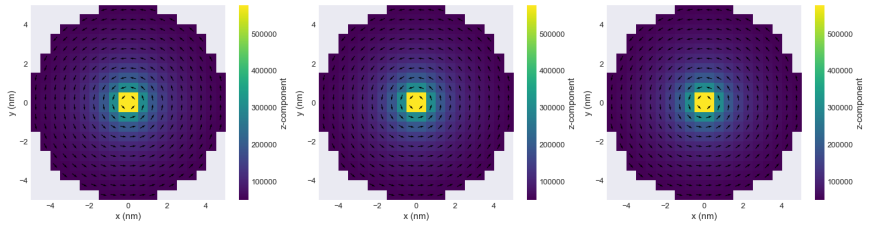



FIGURE 78. Reading files in vtk, ovf, hdf5 format from left to right.

2.2.10. *Line object.* In this tutorial, we show the basics of the line object. We start from the following field:

```
import discretisedfield as df

mesh = df.Mesh(p1=(-10e-9, -10e-9, -10e-9), p2=(15e-9, 10e-9, 5e-9),
                                    cell=(1e-9, 1e-9, 1e-9))

def value_fun(point):
  x, y, z = point
  c = 1e9
  return (x*c, 2*y*c, 3*z*c)

field = df.Field(mesh, dim=3, value=value_fun, norm=1e6)
```

We create a line object by passing two points between which the line spans as well as the number of points we want to sample on that line:

```
line = field.line(p1=(-10e-9, 0, 0), p2=(15e-9, 0, 0), n=25)
```

We can ask for the data on the line:

```
line.data
```

Dimension of the value:

```
line.dim
```

Length of the line:

```
line.length
```

The number of points:

```
line.n
```

The columns storing coordinates of points are:

```
line.point_columns
```

Similarly, the columns storing field values are:

```
line.value_columns
```

By default, value columns start with `v`, but we can rename them:

```
line.value_columns = ['mx', 'my', 'mz']
```

Line data is now:

```
line.data
```

As for the line visualization, default plot is:

```
line.mpl()
```

We can change the size of the plot:

```
line.mpl(figsize=(10, 5))
```

Also, we can limit the values on the x-axis:

```
line.mpl(xlim=[5e-9, 20e-9])
```

and we can choose what we want to plot:

```
line.mpl(yaxis=['mx', 'my'])
```

Similar to all other plots, we can build interactive plots:

```
@df.interact(xlim=line.slider(continuous_update=False),
  yaxis=line.selector())
def myplot(xlim, yaxis):
  return line.mpl(figsize=(10, 6), yaxis=yaxis, marker='o', xlim=xlim)
```

2.2.11. *Table basics.* OOMMF saves the scalar data of micromagnetic simulations in `.odt` files, whereas mumax$^3$ saves it in `.txt` files. `ubermagtable` is a convenience tool which provides functions for reading, manipulating, and visualizing data from those files.

As an example, we are going to use a simple macrospin example:

```
import oommfc as mc
import discretisedfield as df
import micromagneticmodel as mm

# Define a macrospin mesh (i.e. one discretisation cell).
p1 = (0, 0, 0)  # first point of the mesh domain (m)
p2 = (1e-9, 1e-9, 1e-9)  # second point of the mesh domain (m)
n = (1, 1, 1)  # discretisation cell size (m)

Ms = 8e6  # magnetisation saturation (A/m)
H = (0, 0, 2e6)  # external magnetic field (A/m)
gamma0 = 2.211e5  # gyromagnetic ratio (m/As)
alpha = 0.1  # Gilbert damping

region = df.Region(p1=p1, p2=p2)
mesh = df.Mesh(region=region, n=n)

system = mm.System(name='macrospin')
system.energy = mm.Zeeman(H=H)
system.dynamics = mm.Precession(gamma0=gamma0) + mm.Damping(alpha=
                                    alpha)
system.m = df.Field(mesh, dim=3, value=(1, 0, 0), norm=Ms)

td = mc.TimeDriver()
td.drive(system, t=0.1e-9, n=200)
```

Table data is now loaded into `pandas.DataFrame` and it can be accessed via `data` attribute.

```
system.table.data
```

Apart from the data, units for individual columns are stored in `units` attribute.

```
system.table.units
```

This returns a dictionary, whose keys are column names and its values are the units. If table data was obtained by using time driver, table data store different values which are all a function of time. Column names of all time-dependent data can be obtained using `data_columns`:

```
system.table.data_columns
```

Similarly, column name storing time data is:

```
system.table.time_column
```

The total length of time interval over which data was recorded is:

```
system.table.length
```

2.2.12. *Table interactive plot.* In previous tutorials we showed how to plot the data as well as some widgets which are a part of the table object. In this tutorial, we show how an interactive plot can be built.

Same as in previous tutorials, we show use a macrospin example:

```
import oommfc as mc
import discretisedfield as df
import micromagneticmodel as mm

# Define a macrospin mesh (i.e. one discretisation cell).
p1 = (0, 0, 0)  # first point of the mesh domain (m)
p2 = (1e-9, 1e-9, 1e-9)  # second point of the mesh domain (m)
n = (1, 1, 1)  # discretisation cell size (m)

Ms = 8e6  # magnetisation saturation (A/m)
H = (0, 0, 2e6)  # external magnetic field (A/m)
gamma0 = 2.211e5  # gyromagnetic ratio (m/As)
alpha = 0.1  # Gilbert damping

region = df.Region(p1=p1, p2=p2)
mesh = df.Mesh(region=region, n=n)

system = mm.System(name='macrospin')
system.energy = mm.Zeeman(H=H)
system.dynamics = mm.Precession(gamma0=gamma0) + mm.Damping(alpha=
                                    alpha)
system.m = df.Field(mesh, dim=3, value=(1, 0, 0), norm=Ms)

td = mc.TimeDriver()
td.drive(system, t=0.1e-9, n=200)
```

By calling `mpl` method, default plot is shown in Figure 79:

```
system.table.mpl()
```

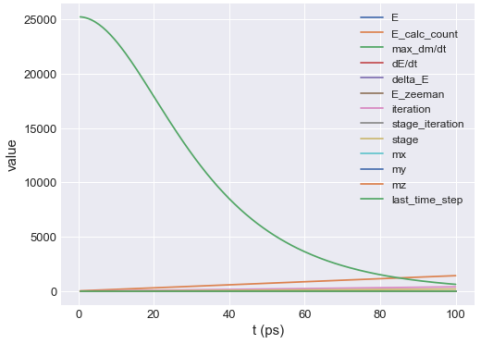



FIGURE 79. Table plot.

By default, all data columns are plotted. To select only certain data columns, `yaxis` can be passed. `yaxis` is a list of strings, where each string matches one of the columns. For instance, if we want to plot the average magnetization components, the plot in Figure 80 is:

```
system.table.mpl(yaxis=['mx', 'my', 'mz'])
```

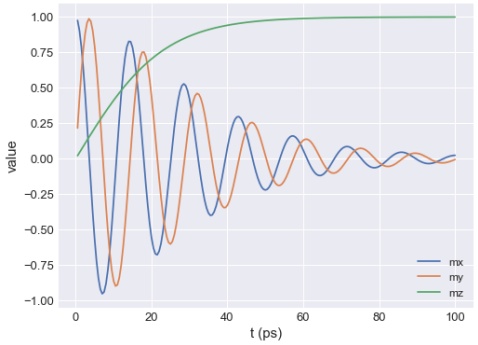



FIGURE 80. Table plot.

Now, let us say we want to interactively choose data columns we want to plot on y-axis of the plot. We start by putting our plotting inside a function and exposing the argument we want to choose:

```
def my_interactive_plot(yaxis):
  system.table.mpl(yaxis=yaxis)
```

Having the function defined, we have to call it with required `yaxis` argument in order to get the plot. For example,

```
my_interactive_plot(yaxis=['mx', 'my', 'mz'])
```

To interactively change the values of `yaxis`, we have to:

(a) decorate the function using `ubermagtable.interact` and
(b) assign a widget to the `yaixs` in the `ubermagtable.interact` argument list.

```
# NBVAL_IGNORE_OUTPUT
@df.interact(yaxis=system.table.selector())
def my_interactive_plot(yaxis):
  system.table.mpl(yaxis=yaxis)
```

The plot, together with a selector widget is shown. In order to select multiple data columns from the widget list, we hold `Ctrl` (Windows, Linux) or `Cmd` (MacOS) and select multiple data columns.

Another argument of the plot we can interact with is the range of time values on the horizontal axis. For that, we have to expose `xlim` and assign it a slider widget.

```
# NBVAL_IGNORE_OUTPUT
@df.interact(yaxis=system.table.selector(),
    xlim=system.table.slider())
def my_interactive_plot(yaxis, xlim):
  system.table.mpl(figsize=(10,5), yaxis=yaxis, xlim=xlim, marker='o')
```

This way, by exposing the axes and passing any allowed `matplotlib.pyplot.plot` argument, we can customize the plot any way we like (as long as it is allowed by `matplotlib`).

2.2.13. *matplotlib visualisation.* In this tutorial, we give an overview of how objects from `discretisedfield` can be visualized using matplotlib, as well as, how the plots can be configured.

As an example, we are going to use the following field:

```
import discretisedfield as df
import matplotlib.pyplot as plt

region = df.Region(p1=(-10e-9, -10e-9, -10e-9), p2=(15e-9, 10e-9, 5e-9
                                    ))
mesh = df.Mesh(region=region,
  cell=(1e-9, 1e-9, 1e-9),
  subregions={'r1': df.Region(p1=(-10e-9, -10e-9, -10e-9), p2=(0, 10e-
                                    9, 5e-9)),
    'r2': df.Region(p1=(0e-9, -10e-9, -10e-9), p2=(15e-9, 10e-9, 5e-9)
                                    )})

def value_fun(point):
  x, y, z = point
  c = 1e9
  if x < -5e-9:
    return (0, y*c, z*c)
  else:
    return (x*c, y*c, z*c)

field = df.Field(mesh, dim=3, value=value_fun, norm=1e6)
```

As for scalar field visualization, before we can plot the field using `matplotlib`, we have to define the plane we want to plot.

```
field.x.plane('z').mpl_scalar()
```

In order to remove the cells where field is zero from the plot, `filter_field` must be passed:

```
field.x.plane('z').mpl_scalar(filter_field=field.x)
```

Colorbar is added to the plot by default. However, we can remove it from the plot:

```
field.x.plane('z').mpl_scalar(filter_field=field.x, colorbar=False)
```

Similarly, we can change the colorbar label:

```
field.x.plane('z').mpl_scalar(filter_field=field.x, colorbar_label='
                                    something')
```

We can change the size of the plot:

```
field.x.plane('z').mpl_scalar(figsize=(12, 8), filter_field=field.x,
                                    colorbar_label='something')
```

Sometimes it is necessary to adjust the limits on the colorbar:

```
field.x.plane('z').mpl_scalar(figsize=(9, 5), filter_field=field.x,
                                    colorbar_label='something', clim=(0
                                    , 1e6))
```

Multiplier is computed internally, but it can be explicitly changed using `multiplier`:

```
field.x.plane('z').mpl_scalar(figsize=(9, 5), filter_field=field.x,
                                    colorbar_label='something',
                                    multiplier=1e-12)
```

In order to save the plot, we pass `filename` and the plot is saved as PDF.

```
field.x.plane('z').mpl_scalar(filename='scalar.pdf')
```

`mpl_scalar` is based on `matplotlib.imshow`, so any argument accepted by it can be passed:

```
field.x.plane('z').mpl_scalar(interpolation='bilinear')
```

The way we can change the axes, we address at the end of this tutorial. As for the vector field visualization. Default plot in Figure 81 is:

```
field.plane('z').mpl_vector()
```

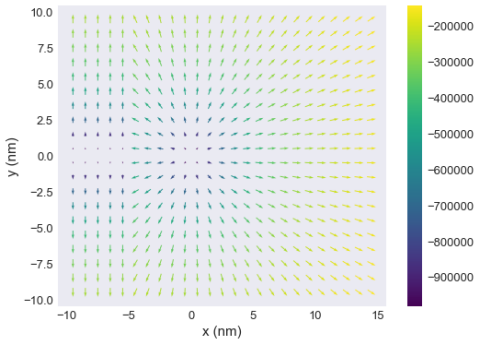



FIGURE 81. Vector field visualization.

Reduce the number of vectors.

```
field.plane('z', n=(20, 10)).mpl_vector()
```

Turn off color by

```
field.plane('z', n=(20, 10)).mpl_vector(color=False)
```

Use different color field by

```
field.plane('z', n=(20, 10)).mpl_vector(color_field=field.x, colorbar=
                                        True)
```

Turn off colorbar by

```
field.plane('z', n=(20, 10)).mpl_vector(color_field=field.x, colorbar=
                                        False)
```

Add colorbar label by

```
field.plane('z', n=(20, 10)).mpl_vector(color_field=field.x,
                                        colorbar_label='something')
```

Change colormap by

```
field.plane('z', n=(20, 10)).mpl_vector(color_field=field.x,
                                        colorbar_label='something', cmap='
                                        plasma')
```

Colormap limit

```
field.plane('z', n=(20, 10)).mpl_vector(color_field=field.x,
                                        colorbar_label='something', clim=(0
                                        , 1e6))
```

Multiplier

```
field.plane('z', n=(20, 10)).mpl_vector(color_field=field.x,
                                        colorbar_label='something',
                                        multiplier=1)
```

Figsize

```
field.plane('z', n=(20, 10)).mpl_vector(figsize=(5, 3))
```

Saving plot

```
field.plane('z', n=(20, 10)).mpl_vector(figsize=(5, 3), filename='
                                        vector.pdf')
```

Quiver argument

```
field.plane('z', n=(20, 10)).mpl_vector(headwidth=8)
```

Scale

```
field.curl.plane('z', n=(20, 10)).mpl_vector(scale=5e15)
```

Note that `mpl` is a convenience function which allows simple building of plots. Default behaviour is:

```
field.plane('x').mpl()
```

Changing figsize to fit colorbar

```
field.plane('x').mpl(figsize=(10, 6))
```

Change scalar field

```
field.plane('x').mpl(figsize=(10, 6), scalar_field=field.y)
```

Turn off colorbar

```
field.plane('x').mpl(figsize=(10, 6), scalar_field=field.y,
                                      scalar_colorbar=False)
```

Add scalar colorbar label

```
field.plane('x').mpl(figsize=(10, 6), scalar_field=field.y,
                                      scalar_colorbar_label='something')
```

Add filter field

```
field.plane('z').mpl(figsize=(10, 6), scalar_filter_field=field.x,
                                      scalar_field=field.y,
                                      scalar_colorbar_label='something')
```

Scalar colormap

```
field.plane('z').mpl(figsize=(10, 6), scalar_filter_field=field.x,
                                      scalar_field=field.y,
                                      scalar_colorbar_label='something',
                                      scalar_cmap='plasma')
```

Colorbar limit:

```
field.plane('z').mpl(figsize=(10, 6), scalar_filter_field=field.x,
                                      scalar_field=field.y,
                                      scalar_colorbar_label='something',
                                      scalar_cmap='plasma', scalar_clim=(
                                      0, 1e6))
```

Vector field

```
field.plane('z').mpl(figsize=(10, 6), scalar_filter_field=field.x,
                                      scalar_field=field.y,
                                      scalar_colorbar_label='something',
                                      scalar_cmap='plasma',scalar_clim=(0
                                      , 1e6),vector_field=field.div,
                                      vector_scale=3e16)
```

Color vector field:

```
field.plane('z').mpl(figsize=(10, 6), scalar_filter_field=field.x,
                                      scalar_field=field.y,
                                      scalar_colorbar_label='something',
                                      scalar_cmap='plasma', scalar_clim=(
                                      0, 1e6), vector_field=field.curl,
                                      vector_color_field=field.y,
                                      vector_color=True, vector_scale=
                                      7e15)
```

Add vector colorbar

```
field.plane('z').mpl(figsize=(12, 6), scalar_filter_field=field.x,
                                      scalar_field=field.y,
                                      scalar_colorbar_label='something',
                                      scalar_cmap='plasma', scalar_clim=(
                                      0, 1e6), vector_field=field.div,
                                      vector_color_field=field.y,
                                      vector_color=True,vector_colorbar=
                                      True, vector_scale=3e16)
```

Vector cmap

```
field.plane('z').mpl(figsize=(12, 6), scalar_filter_field=field.x,
                                      scalar_field=field.y,
                                      scalar_colorbar_label='something',
                                      scalar_cmap='plasma', scalar_clim=(
                                      0, 1e6), vector_field=field,
                                      vector_color_field=field.y,
                                      vector_color=True,vector_colorbar=
                                      True, vector_colorbar_label='vector
                                      ', vector_cmap='hsv')
```

Vector clim

```
field.plane('z').mpl(figsize=(12, 6), scalar_filter_field=field.x,
                                      scalar_field=field.y,
                                      scalar_colorbar_label='something',
                                      scalar_cmap='plasma', scalar_clim=(
                                      0, 1e6), vector_field=field,
                                      vector_color_field=field.y,
                                      vector_color=True,vector_colorbar=
                                      True, vector_colorbar_label='vector
                                      ',vector_cmap='hsv', vector_clim=(0
                                      , 1e6))
```

Save plot

```
field.plane('x').mpl(figsize=(12, 6), scalar_field=field.plane('x').
                                      angle,scalar_colorbar_label='
                                      something',scalar_cmap='twilight',
                                      vector_field=field,
                                      vector_color_field=field.y,
                                      vector_color=True,vector_colorbar=
                                      True, vector_colorbar_label='vector
                                      ', vector_cmap='hsv', vector_clim=(
                                      0, 1e6),multiplier=1e-12, filename=
                                      'mpl.pdf')
```

In order to change the axes on which the plot is added, we have to expose them:

```
import matplotlib.pyplot as plt

fig = plt.figure(figsize=(10, 5))
ax = fig.add_subplot(111)

field.plane('z').mpl(ax=ax)

ax.set_xlabel('my x coordinate in nm')
ax.set_title('My cool plot')
```

We can access `Stack Overflow` https://stackoverflow.com to ask some questions there. Scientific python ecosystem contains

(a) numpy: linear algebra;
(b) scipy: numerical analysis;
(c) matplotlib: 2d (some 3d) plotting;
(d) pandas: big data for python;
(e) scikit-learn: machine learning.

Why did we start with OOMMF:

(1) Probably the most widely used micromagnetic simulation tool;

(2) Developed at NIST, US, since 1998 by Michael Donahue and Don Porter;
(3) Cited over 2200 times in scientific publications;
(4) Written in C++ and Tcl;
(5) Finite difference code;
(6) Very often used for comparisons between codes (https://math.nist.gov/oommf/).

**Remark 3** (Ubermag Documentation)**.** *Github organization: https://github.com/ubermag. Each package has an online documentation: use* `[package_name].readthedocs.io`*. There is API reference which is always up-to-date and covers functionality. Available in the cloud (Binder).*

## References

[1] B. Marijan, R. A. Pepper, D. Cortés-Ortuño, A. Bilal, B. Marc-Antonio, G. Downing, T. Kluyver, H. Ondrej, and F. Hans, *Stable and manipulable Bloch point*, Scientific Reports (Nature Publisher Group) **9** (2019), no. 1.
[2] Y. Zhou and M. Ezawa, *A reversible conversion between a skyrmion and a domain-wall pair in a junction geometry*, Nature comm. **5** (2014), no. 1, 1–8.