## Contents:

1 Getting started 1
   1.1 Getting started 00: First Ubermag notebook ............................... 1
   1.2 Getting started 01: Geometry and magnetisation ........................ 4
   1.3 Getting started 02: Hamiltonian ........................................... 11
   1.4 Getting started 03: Dynamics ............................................... 18
   1.5 Getting started 04: Current induced domain wall motion ............ 23
   1.6 Exercise 01: Domain wall pair conversion ............................... 26
   1.7 Exercise 02: Vortex dynamics .............................................. 28

2 Tutorials 31
   2.1 Tutorial 00: Installation, testing, and upgrade .......................... 31
   2.2 Tutorial 01: Zeeman energy term ........................................... 32
   2.3 Tutorial 02: Exchange energy term ....................................... 34
   2.4 Tutorial 03: Uniaxial anisotropy energy term ........................... 37
   2.5 Tutorial 04: Dzyaloshinskii-Moriya energy term ......................... 41
   2.6 Tutorial 05: Cubic anisotropy energy term ............................... 44
   2.7 RKKY ................................................................. 48
   2.8 Adding multiple terms of the same class .................................. 49
   2.9 Negative exchange energy constant A ..................................... 51
   2.10 Tutorial 07: Standard problem 3 ......................................... 53
   2.11 Tutorial 08: Standard problem 4 ......................................... 58
   2.12 Tutorial 09: Standard problem 5 ......................................... 64
   2.13 Tutorial 10: FMR standard problem ...................................... 68
   2.14 Tutorial 11: Deriving energy values ...................................... 71
   2.15 Tutorial 12: Calculating a stray field using an airbox method .... 74
   2.16 Tutorial 13: Skyrmion in a disk ......................................... 77
   2.17 Running simulations inside Docker ....................................... 81
   2.18 Tutorial 15: Support, License, How to cite, and Acknowledgements 83

3 API Reference 85
   3.1 oommfc.CGEvolver .......................................................... 85
   3.2 oommfc.EulerEvolver ....................................................... 88
   3.3 oommfc.RungeKuttaEvolver ................................................. 91
   3.4 oommfc.SpinTEvolver ....................................................... 94
   3.5 oommfc.SpinXferEvolver .................................................... 97
   3.6 oommfc.MinDriver ........................................................... 100
   3.7 oommfc.TimeDriver .......................................................... 104
Index

3.8  ommfc.compute ......................................................... 108
3.9  ommfc.delete ............................................................. 108
3.10 ommfc.oommf ............................................................. 109
1.1 Getting started 00: First Ubermag notebook

Interactive online tutorial:

The goal of this tutorial is to familiarise users with basics of running Ubermag simulations in Jupyter notebook. The only thing you need to know for this tutorial is how to execute individual cells: this is done by pressing Shift + Return (Return = Enter).

1.1.1 Simple Ubermag simulation

Before we specify and run the simulation, we have to import Ubermag modules we intend to use. Because we are going to run OOMMF simulations, we will import oommfc and for creating finite difference fields, we are going to import discretisedfield.

```python
import oommfc as oc
import discretisedfield as df
import micromagneticmodel as mm

# The following line enables plotting inside the notebook.
%matplotlib inline
```

System

The object on which different drivers act on in Ubermag is `oommf System`. In order to define the micromagnetic system we intend to simulate, we have to specify:

1. Hamiltonian,
2. Dynamics equation,
3. Magnetisation configuration.
Hamiltonian

The Hamiltonian for the first Ubermag simulation is very simple and contains only:

1. exchange,
2. demagnetisation,
3. Zeeman energy terms.

We will apply an external magnetic field in the $x$-direction for the purpose of this demonstration:

\[
\text{system.energy} = \text{mm.Exchange}(A=A) + \text{mm.Demag()} + \text{mm.Zeeman}(H=H)
\]

Dynamics equation

The dynamics equation contains only precession and damping terms:

\[
\text{system.dynamics} = \text{mm.Precession}(\text{gamma0}=\text{gamma0}) + \text{mm.Damping}(\alpha=\alpha)
\]

Initial magnetisation

We initialise the system in positive $y$-direction, i.e. $(0, 1, 0)$, which is different from the equilibrium state we expect for the external Zeeman field applied in $x$ direction:

\[
\text{system.m} = \text{df.Field}(\text{mesh}, \text{dim}=3, \text{value}=(0, 1, 0), \text{norm}=\text{Ms})
\]

Inspect the properties of the system

We can check the characteristics of the system we defined by asking objects to represent themselves:

\[
A \left[ (\nabla m_x)^2 + (\nabla m_y)^2 + (\nabla m_z)^2 \right] - \frac{1}{2} \mu_0 M_s m \cdot H_0 - \mu_0 M_s m \cdot H
\]
We can also visualise the current magnetisation field (using MatPlotLib=mpl):

```
[9]: system.m.plane('z').mpl()
```

And we can visualise the same data using k3d_vectors - this is more suitable for interactive exploration:

```
[10]: system.m.plane('z').k3d_vector(head_size=20)
```

### Driving the system

After the system object is created, we can minimise its energy (relax it) using the Minimisation Driver (MinDriver).

```
[11]: md = oc.MinDriver()
    md.drive(system)
```

The system is now relaxed, and we can plot its slice and compute its average magnetisation.

```
[12]: system.m.plane('z').mpl()
```
We can see that the magnetisation is aligned along the $x$-direction, as expected having in mind we applied the external magnetic field in that direction.

### 1.2 Getting started 01: Geometry and magnetisation

Interactive online tutorial:

In this tutorial we explore how geometries and magnetisation states can be specified. The package we use to define finite difference meshes and fields is called discretisedfield and we have to import it before we start.

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

# The following line enables plotting inside the notebook.
%matplotlib inline
```

#### 1.2.1 Mesh

Let us say that we need to define a nanocube mesh with edge length $L = 100$ nm and discretisation cell $(d, d, d)$, with $d = 10$ nm. For that we need to define two points $p_1$ and $p_2$ between which the mesh spans and pass them (together with the discretisation cell) to the Mesh class:

```python
L = 100e-9  # edge length (m)
d = 10e-9  # cell size (m)

p1 = (0, 0, 0)  # first point of the cuboid containing simulation geometry
```
We can then inspect some basic parameters of the mesh:

**Edge length:**

```python
[3]: mesh.region.edges  # edge length
[3]: (1e-07, 1e-07, 1e-07)
```

**Number of discretisation cells in all three directions:**

```python
[4]: mesh.n  # number of cells
[4]: (10, 10, 10)
```

**Minimum mesh domain coordinate:**

```python
[5]: mesh.region.pmin  # minimum mesh domain coordinate
[5]: (0.0, 0.0, 0.0)
```

**Maximum mesh domain coordinate:**

```python
[6]: mesh.region.pmax  # maximum mesh domain coordinate
[6]: (1e-07, 1e-07, 1e-07)
```

**Visualise the mesh domain and a discretisation cell:**

Using k3d interactive plots:

```python
[7]: mesh.k3d()  
Output() 
```

Using matplotlib

```python
[8]: mesh.mpl() 
```
1.2.2 Field

After we defined a mesh, we can define different finite difference fields. For that, we use Field class. We need to provide:

1. Mesh,
2. Dimension of the data values,
3. Value of the field.

Let us define a 3D-vector field (dim=3) that is uniform in the (1, 0, 0) direction.

\[ m = \text{df.Field}(\text{mesh}, \text{dim}=3, \text{value}=(1, 0, 0)) \]

A simple slice visualisation of the mesh in the \( z \) direction at \( L/2 \) is:

\[ m\text{.plane('z').mpl()} \]

Similarly, a three-dimensional interactive visualisation is:
Spatially varying field

When we defined a uniform vector field, we used a tuple \((1, 0, 0)\) to define its value. However, we can also provide a Python function if we want to define a non-uniform field. This function takes the position in the mesh as input, and returns a value that the field should have at that point:

```python
[12]: def m_value(pos):
    x, y, z = pos  # unpack position into individual components
    if x > L/5:
        return (1, 0, 0)
    else:
        return (-1, 1, 0)

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

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

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

```python
[13]: point = (0, 0, 0)
    m(point)

[13]: (-1.0, 1.0, 0.0)
```

In micromagnetics, the saturation magnetisation \(M_s\) is typically constant (at least for each position). The Field constructor accepts an additional parameter `norm` which we can use for that:

```python
[14]: Ms = 8e6  # saturation magnetisation (A/m)
    m = df.Field(mesh, dim=3, value=m_value, norm=Ms)

m((50e-9, 0, 0))
```
**Spatially varying \( M_s \)**

By defining different norms, we can specify different geometries, so that \( M_s = 0 \) outside the mesh. For instance, let us assume we want to define a sphere of radius \( L/2 \) and magnetise it in the negative \( y \) direction.

```
[15]: mesh = df.Mesh(p1=(-L/2, -L/2, -L/2), p2=(L/2, L/2, L/2), cell=(d, d, d))

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

m = df.Field(mesh, dim=3, value=(0, -1, 0), norm=Ms_value)
m.plane('z').mpl()
```

Similarly, we can inspect the defined domain using k3d and inspecting the field’s norm.

```
[16]: m.norm.k3d_nonzero()
```

Output()

### 1.2.3 Exercise 1

**a)** The code in the next cell defines a thin film of thickness \( t \) in the \( xy \) plane. Extend it to define the disk geometry of thickness \( t = 10 \) nm and diameter \( d = 120 \) nm. The saturation magnetisation \( M_s = 10^7 \) A/m. The disk is centred around the origin \((0, 0, 0)\). The magnetisation is \( \mathbf{m} = (1, 0, 0) \) at all points.

```
[17]: 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(pos):
    x, y, z = pos
    # insert missing code here
    if (x**2 + y**2)**0.5 < d/2:
        return Ms
    else:
        return 0
    # return Ms

m = df.Field(mesh, dim=3, value=(1, 0, 0), norm=Ms_value)
m.plane('z', n=(20, 20)).k3d_vector(head_size=20)

Output()

b) Extend the previous example in the next cell so that the magnetisation is:

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

with saturation magnetisation $10^7$ A m$^{-1}$.

[18]: 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(pos):
    x, y, z = pos
    # Copy code from exercise 1a.
    if (x**2 + y**2)**0.5 < d/2:
        return Ms
    else:
        return 0
    # return Ms

def m_value(pos):
    x, y, z = pos
    # Insert missing code here to get the right magnetisation.
    if y <= 0:
        return (-1, 0, 0)
    else:
        return (1, 0, 0)
    # return (1, 0, 0)

m = df.Field(mesh, dim=3, value=m_value, norm=Ms_value)
m.plane('z', n=(20, 20)).k3d_vector(head_size=20)
1.2.4 Exercise 2

Extend the code in the following cell to define the following geometry with 10 nm thickness:

The magnetisation saturation is $8 \times 10^6$ A m$^{-1}$ and the magnetisation direction is as shown in the figure.

```
[19]:
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
    # Insert missing code here to get the right shape of geometry.
    if x < 50e-9 and y > 35e-9:
        return 0
    else:
        return Ms
    # 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()
```
1.3 Getting started 02: Hamiltonian

Interactive online tutorial:

The energy of a magnetic system can be computed using Hamiltonian, which consists of different energy terms. In addition, the effective field which affects the magnetisation dynamics in the LLG equation, is also computed from the system’s Hamiltonian.

In this tutorial, we explore different energy terms that can occur in the Hamiltonian.

1.3.1 Zeeman energy

The Zeeman energy tends to align all magnetic moments parallel to the external magnetic field \( H \), so that the energy density
\[
\omega_z = -\mu_0 M_s \mathbf{m} \cdot \mathbf{H}
\]
is minimum, where \( \mu_0 \) is the magnetic constant and \( M_s \) is the magnetisation saturation.

To demonstrate the effect of Zeeman energy we will apply an external magnetic field \( H = 10^6 \, \text{A} \, \text{m}^{-1} \) in the positive \( z \)-direction to the one-dimensional array of magnetic moments. For the initial magnetisation configuration, we will take the \((1, 0, 1)\) direction with \( M_s = 8 \times 10^6 \, \text{A} \, \text{m}^{-1} \).

```
[1]: import oommfc as oc
   import discretisedfield as df
   import micromagneticmodel as mm

   %matplotlib inline

   Our mesh is a one-dimensional array of discretisation cells.

[2]: p1 = (0, 0, 0)
    p2 = (1e-9, 1e-9, 1e-9)
    cell = (1e-9, 1e-9, 1e-9)

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

    mesh.k3d()
```
After we defined our system, we can minimize its energy using MinDriver.

We can see that after applying the magnetic field, all magnetic moments align parallel to $\mathbf{H}$. 
Exercise 1

Change the direction of applied field to be in the negative $x$ direction and inspect the minimum magnetisation state.

Uniaxial anisotropy

Uniaxial anisotropy energy tends to align magnetic moments to be on the anisotropy axis either parallel or antiparallel without a preferred direction. Its energy density is $w_a = -K (\mathbf{m} \cdot \mathbf{u})^2$, where $K$ is the anisotropy constant and $\mathbf{u}$ is the anisotropy axis.

Let's assume our anisotropy axis is in the $(1, 0, 1)$ direction with $K = 6 \times 10^6 \text{ J m}^{-3}$ and the initial magnetisation is defined as

$$
\mathbf{m} = \begin{cases} 
(-1, 0, -0.1) & \text{for } x \leq 5 \text{ nm} \\
(1, 0, 0.1) & \text{for } x > 5 \text{ nm}
\end{cases}
$$

with $M_s = 8 \times 10^6 \text{ A m}^{-1}$.
We can now relax the magnetisation.

```
[7]: md.drive(system)
system.m.plane('y').mpl()
```

Running OOMMF (ExeOOMMFRunner) [2020/06/12 00:36]... (1.6 s)

We see that in the relaxed state, all magnetic moments are aligned with the anisotropy axis. The initial configuration affects which orientation along the anisotropy direction is chosen.

### 1.3.3 Exchange energy

The Zeeman and uniaxial anisotropy are *local* energy terms; they only depend on the magnetisation at a given location. In contrast, the exchange energy represents a short range interaction and tends to align all spins parallel to each other without a preferential direction.

The exchange energy density is $w_{ex} = A[\nabla m_x)^2 + (\nabla m_y)^2 + (\nabla m_z)^2]$, where $A$ is the exchange energy constant. To demonstrate how it affects the magnetisation configuration, we will use $A = 8 \text{ pJ m}^{-1}$ on a non-uniform magnetisation configuration.

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

(continues on next page)
def m_initial(pos):
    x, y, z = pos
    if x <= 5e-9:
        return (0, 0, 1)
    else:
        return (1, 0, 0)

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

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

[9]: md.drive(system)
    system.m.plane('y').mpl()

Running OOMMF (ExeOOMMFRunner) [2020/06/12 00:36]... (1.5 s)

1.3.4 Dzyaloshinkii-Moriya energy

While the exchange interaction aims to align (neighbouring) magnetic moments parallel to each other, the Dzyaloshinskii-Moriya (DM) energy wants to align them perpendicular to each other.
The (interfacial) DMI energy density is \( w_{\text{dmi}} = D [m_z \nabla \cdot m - (m \cdot \nabla) m_z] \) (crystallographic class \( C_{nv} \)), with \( D \) being the DM constant. Again, we demonstrate its effect by starting from a uniform configuration, with \( D = 3 \times 10^3 \text{ J m}^{-2} \).

```python
[10]:
# Add OOMMF code here
```

1.3.5 Exchange and Zeeman energies

So far, we investigated the effect of individual energy terms in the Hamiltonian. Now, we look at how multiple energies in the Hamiltonian contribute to the final magnetisation configuration. We start with the simplest example where only exchange and Zeeman energies are present. As we showed before, exchange energy wants to align all magnetic moments parallel to each other without a preferential direction, while Zeeman energy wants them to be parallel to the external magnetic field. Therefore, we can expect that the minimum energy configuration is going to be the uniform state in the direction of an applied field \( \mathbf{H} \).

```python
[11]:
# Add OOMMF code here
```
Both energies are minimal in the above relaxed state: the magnetisation is uniform (minimising the exchange energy) and is aligned with the external field (minimising the Zeeman energy).

1.3.6 Competition of DMI and exchange

Here, we study the competition of the exchange and (interfacial) DMI energy.

Exercise 2

Relax a one-dimensional chain of magnetic moments of length $L = 20$ nm with discretisation cell size of $(d, d, d)$, with $d = 1$ nm. The Hamiltonian should consist of:

- exchange energy with $A = 1 \times 10^{-11}$ J m$^{-1}$, and
- Dzyaloshinskii-Moriya energy with $D = 4\pi A/L \approx 6.28 \times 10^{-3}$ J m$^{-2}$.
The magnetisation saturation is $M_s = 8 \times 10^6 \text{ A m}^{-1}$. Initialise the system with any uniform state.

Questions: - What is the relaxed state? - Double the value of DMI constant to be $D = 12.56 \times 10^3 \text{ J m}^{-2}$. What has changed?

```
[13]: system = mm.System(name='exchange_and_DMI')
A = 1e-11 # exchange energy constant (J/m)
D = 6.28e-3 # DMI energy constant (J/m**2)
Ms = 8e6 # Saturation magnetisation (A/m)

region = df.Region(p1=(0, 0, 0), p2=(20e-9, 1e-9, 1e-9))
mesh = df.Mesh(region=region, cell=(1e-9, 1e-9, 1e-9))
m_initial = (0, 1, 1) # uniform initial magnetisation
system.m = df.Field(mesh, dim=3, value=m_initial, norm=Ms)

system.m.plane('y').k3d_vector()
```

```
[14]: system.energy = mm.Exchange(A=A) + mm.DMI(D=D, crystalclass='Cnv')

md = oc.MinDriver() # create energy minimisation driver
md.drive(system)

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

```
[15]: # Uncomment and modify the value of D in the following line to double it
    # system.hamiltonian.dmi.D = D
    md.drive(system)
    system.m.plane('y').k3d_vector(color_field=system.m.z)
```

1.4 Getting started 03: Dynamics

Interactive online tutorial:

The dynamics of magnetisation field $\mathbf{m}$ is governed by the Landau-Lifshitz-Gilbert (LLG) equation

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

where $\gamma_0$ is the gyromagnetic ratio, $\alpha$ is the Gilbert damping, and $\mathbf{H}_{\text{eff}}$ 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.

We will study the simplest “zero-dimensional” case - macrospin. In the first step, after we import necessary modules (oommfc and discretisedfield), we create the mesh which consists of a single finite cell.
import oommfc as oc
import discretisedfield as df
import micromagneticmodel as mm
%matplotlib inline

# 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)
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

\[ w = -\mu_0 M_s \mathbf{m} \cdot \mathbf{H}, \]

where \( M_s \) is the saturation magnetisation, \( \mu_0 \) is the magnetic constant, and \( \mathbf{H} \) is the external magnetic field. We apply the external magnetic field with magnitude \( H = 2 \times 10^6 \) A m\(^{-1}\) in the positive \( z \) direction.

\[ \mathbf{H} = (0, 0, 2e6) \quad \# \text{external magnetic field (A/m)} \]

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

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

\[ \gamma_0 = 2.211e5 \quad \# \text{gyromagnetic ratio (m/As)} \]
alpha = 0.1  # Gilbert damping

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

To check what is our dynamics equation:

\[ -\frac{\gamma_0}{1+\alpha^2} \mathbf{m} \times \mathbf{H}_{\text{eff}} - \frac{\gamma_0 \alpha}{1+\alpha^2} \mathbf{m} \times (\mathbf{m} \times \mathbf{H}_{\text{eff}}) \]

Before we start running time evolution simulations, we need to initialise the magnetisation. In this case, our magnetisation is pointing in the positive \( x \) direction with \( M_s = 8 \times 10^6 \) A m\(^{-1}\). The magnetisation is defined using Field class from the discretisedfield package we imported earlier.

\[ \text{initial}_m = (1, 0, 0) \quad \# \text{vector in x direction} \]
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 \( t = 0.1 \) ns and save the magnetisation configuration in \( n = 200 \) steps.

\[ \text{td} = \text{oc.TimeDriver()} \]
\text{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.

```plaintext
<table>
<thead>
<tr>
<th>E</th>
<th>E_calc_count</th>
<th>max_dm/dt</th>
<th>dE/dt</th>
<th>delta_E</th>
</tr>
</thead>
<tbody>
<tr>
<td>-4.400762e-22</td>
<td>37.0</td>
<td>25204.415522</td>
<td>-8.798712e-10</td>
<td>-3.269612e-22</td>
</tr>
<tr>
<td>-8.797309e-22</td>
<td>44.0</td>
<td>25186.311578</td>
<td>-8.786077e-10</td>
<td>-4.396547e-22</td>
</tr>
<tr>
<td>-1.318544e-21</td>
<td>51.0</td>
<td>25156.186455</td>
<td>-8.765071e-10</td>
<td>-4.38134e-22</td>
</tr>
<tr>
<td>-1.756100e-21</td>
<td>58.0</td>
<td>25114.112032</td>
<td>-8.735776e-10</td>
<td>-4.375555e-22</td>
</tr>
<tr>
<td>-2.191985e-21</td>
<td>65.0</td>
<td>25060.188355</td>
<td>-8.698302e-10</td>
<td>-4.358857e-22</td>
</tr>
<tr>
<td>1</td>
<td>-2.009865e-20</td>
<td>1402.0</td>
<td>690.438568</td>
<td>-6.602614e-13</td>
</tr>
<tr>
<td>2</td>
<td>-2.009897e-20</td>
<td>1409.0</td>
<td>675.493807</td>
<td>-6.319876e-13</td>
</tr>
<tr>
<td>3</td>
<td>-2.009928e-20</td>
<td>1416.0</td>
<td>660.872303</td>
<td>-6.049242e-13</td>
</tr>
<tr>
<td>4</td>
<td>-2.009958e-20</td>
<td>1423.0</td>
<td>646.567078</td>
<td>-5.790193e-13</td>
</tr>
<tr>
<td>195</td>
<td>-2.009865e-20</td>
<td>396.0</td>
<td>1.0</td>
<td>0.017545</td>
</tr>
<tr>
<td>196</td>
<td>-2.009897e-20</td>
<td>398.0</td>
<td>1.0</td>
<td>0.021059</td>
</tr>
<tr>
<td>197</td>
<td>-2.009928e-20</td>
<td>400.0</td>
<td>1.0</td>
<td>0.023428</td>
</tr>
<tr>
<td>198</td>
<td>-2.009958e-20</td>
<td>402.0</td>
<td>1.0</td>
<td>0.024591</td>
</tr>
<tr>
<td>199</td>
<td>-2.009986e-20</td>
<td>404.0</td>
<td>1.0</td>
<td>0.024591</td>
</tr>
</tbody>
</table>

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

```plaintext
```

```
Similarly, we can plot all three magnetisation components

[10]: system.table.data.plot('t', ['mx', 'my', 'mz']);

We can see that after some time the macrospin aligns parallel to the external magnetic field in the $z$ direction.

1.4.1 Exercise 1

By looking at the previous example, explore the magnetisation dynamics for $\alpha = 0.005$ in the following code cell.

[11]: # Uncomment and modify the value of Gilbert damping in the following line.
# system.dynamics.damping.alpha = 1
system.m = df.Field(mesh, dim=3, value=initial_m, norm=Ms)
td.drive(system, t=0.1e-9, n=200)

system.table.data.plot('t', ['mx', 'my', 'mz']);
1.4.2 Exercise 2

Repeat the simulation with $\alpha = 0.1$ and $\mathbf{H} = (0, 0, -2 \times 10^6)$ A m$^{-1}$.

```python
[12]: # Uncomment and modify the value in the following line
    # system.hamiltonian.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.data.plot('t', ['mx', 'my', 'mz']);
```

Running OOMMF (ExeOOMMFRunner) [2020/06/14 11:17]... (3.3 s)
1.4.3 Exercise 3

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?

Interactive online tutorial:

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-polarised current.

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

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

1.5 Getting started 04: Current induced domain wall motion

Interactive online tutorial:

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-polarised current.

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

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

```python
import oommfc as oc
import discretisedfield as df
import micromagneticmodel as mm
%matplotlib inline
```
# 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 initialise the magnetisation in an appropriate way before we relax the system.

```python
[2]: def m_value(pos):
    x, y, z = pos
    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)
```

Now, we can relax the magnetisation.

```python
[3]: md = oc.MinDriver()
md.drive(system)
```

Now we can add the STT term to the dynamics equation.

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

(continues on next page)
And drive the system for 0.5 ns:

```
[5]: td = oc.TimeDriver()
    td.drive(system, t=0.5e-9, n=100)
    system.m.z.plane('z').k3d_scalar()
```

Running OOMMF (ExeOOMMFRunner) [2020/06/12 00:40]... (6.0 s)

Output()

We see that the DW pair has moved to the positive $x$ direction.

### 1.5.1 Exercise 1

Modify the code below (which is a copy of the example from above) to obtain one domain wall instead of a domain wall pair and move it using the same current.

```
[6]: # 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(pos):
        x, y, z = pos
        # Modify the following line
        if 20e-9 < x < 40e-9:
            return (0, 0, -1)
        else:
            return (0, 0, 1)
        # We have added the y-component of 1e-8 to the magnetisation to be able to
```
# plot the vector field. This will not be necessary in the long run.

```python
system.m = df.Field(mesh, dim=3, value=m_value, norm=Ms)
system.m.z.plane('z').k3d_scalar()
```

```
[7]: md = oc.MinDriver()
    md.drive(system)
system.m.z.plane('z').k3d_scalar()
```

Running OOMMF (ExeOOMMFRunner) [2020/06/12 00:40]... (8.2 s)

```
Output()
```

```
[8]: 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=0.5e-9, n=100)
```

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

Running OOMMF (ExeOOMMFRunner) [2020/06/12 00:40]... (5.6 s)

```
Output()
```

1.6 Exercise 01: Domain wall pair conversion

Interactive online tutorial:

We want to simulate a domain wall conversion in a two-dimensional thin film sample with:

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

Please carry out the following steps:

1. Create the following geometry with discretisation cell size (2 nm, 2 nm, 2 nm).
2. Initialise the magnetisation so that when relaxes, a domain pair is present in the narrower part of the geometry.
3. Relax the system. Is a domain wall pair contained in the constrained part?
4. Apply the spin polarised current in the positive $x$ direction with velocity $\mathbf{u} = (400, 0, 0)$ m s$^{-1}$, with $\beta = 0.5$.
5. Evolve the system over 0.2 ns. What did you get? [1]

1.6.1 References


**Solution**

[1]:

```python
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=gamma0) + 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(pos):
    x, y, z = pos
    if x < 50e-9 and (y < 15e-9 or y > 35e-9):
        return 0
    else:
        return Ms
```

(continues on next page)
```python
def m_init(pos):
    x, y, z = pos
    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)
```

Output()

```
[2]: md = oc.MinDriver()
    md.drive(system)

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

    Running OOMMF (ExeOOMMFRunner) [2020/06/12 00:42]... (1.9 s)
```

Output()

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

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

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

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

    Running OOMMF (ExeOOMMFRunner) [2020/06/12 00:42]... (6.9 s)
```

Output()

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

### 1.7 Exercise 02: Vortex dynamics

Interactive online tutorial:

We want to simulate vortex dynamics in a two-dimensional disk sample with \(d = 100\) nm diameter and \(5\) nm thickness with:

- magnetisation saturation \(M_s = 8 \times 10^5\) A m\(^{-1}\),
- exchange energy constant \(A = 13\) pJ m\(^{-1}\),
- gyrotropic ratio \(\gamma = 2.211 \times 10^5\) m A\(^{-1}\) s\(^{-1}\), and
- Gilbert damping \(\alpha = 0.2\).

Please carry out the following steps:

1. Initialise the system so that \((m_x, m_y, m_z) = (-Ay, Ax, 10)\), where \(A = 10^9\) m\(^{-1}\).
2. Minimise the system’s energy. What state did you obtain?
3. Apply an external magnetic field of \( H = 10^4 \text{ A m}^{-1} \) in the positive \( x \) direction and relax the system. Did the vortex core move in the positive \( y \) direction?

4. Turn off an external magnetic field and simulate the vortex dynamics for \( t = 5 \text{ ns} \) and save magnetisation in \( n = 500 \) steps. Plot all three components of magnetisation as a function of time.

### 1.7.1 Solution

```python
[1]:
import discretisedfield as df
import micromagneticmodel as mm
import oommfc as oc
%matplotlib inline

# Geometry
d = 100e-9  # disk diameter (m)
thickness = dx = dy = dz = 5e-9  # discretisation cell (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

region = df.Region(p1=(-d/2, -d/2, 0), p2=(d/2, d/2, thickness))
mesh = df.Mesh(region=region, cell=(dx, dy, dz))

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

def m_init(pos):
    x, y, z = pos
    A = 1e9  # (1/m)
    return -A*y, A*x, 10

system = mm.System(name='vortex_dynamics')
system.energy = mm.Exchange(A=A) + mm.Demag()
system.dynamics = mm.Precession(gamma0=gamma0) + mm.Damping(alpha=alpha)
system.m = df.Field(mesh, dim=3, value=m_init, norm=Ms_fun)

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

system.m.k3d_vector(color_field=system.m.z, head_size=10)

Running OOMMF (ExeOOMMFRunner) [2020/06/30 13:05]... (1.9 s)

Output()

[2]:
H = (1e4, 0, 0)
system.energy += mm.Zeeman(H=H)

(continues on next page)
oommf
c
md.drive(system)

```python
system.m.k3d_vector(color_field=system.m.z, head_size=10)
```

Running OOMMF (ExeOOMMFRunner) [2020/06/30 13:05]... (2.4 s)

Output()

```python
[3]: system.energy.zeeman.H = (0, 0, 0)

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

system.table.data.plot('t', ['mx', 'my', 'mz'])
```

Running OOMMF (ExeOOMMFRunner) [2020/06/30 13:05]... (9.6 s)

```python
[3]: <matplotlib.axes._subplots.AxesSubplot at 0x7fa0805efac0>
```

![Graph showing the plots of mx, my, and mz over time.](image-url)
2.1 Tutorial 00: Installation, testing, and upgrade

Interactive online tutorial:

2.1.1 Installation

We recommend installing oommfc using conda package manager.

If you do not already have it, Download Anaconda for your operating system and follow instructions to install it. For further information on the conda package, dependency, and environment management, please have a look at its documentation.

oommf is installed using conda by running

$ conda install --channel conda-forge oommfc

2.1.2 Testing

After the installation, you can test the oommfc package with

$ python3 -c "import oommfc; oommfc.test()"

2.1.3 Upgrade

If you used conda for the installation, upgrade oommfc by running:

$ conda upgrade oommfc
2.2 Tutorial 01: Zeeman energy term

Interactive online tutorial:

Zeeman energy density is computed as

\[ w_z = -\mu_0 M_s \mathbf{m} \cdot \mathbf{H} \]

where \( \mu_0 \) is the magnetic constant, \( M_s \) is the magnetisation saturation, \( \mathbf{m} \) is the normalised (\( \| \mathbf{m} \| = 1 \)) magnetisation, and \( \mathbf{H} \) is the external magnetic field. Zeeman energy term tends to align all magnetic moments parallel to the external magnetic field.

In oommfc, \( M_s \) and \( \mathbf{m} \) are part of the magnetisation field \( \text{system.m} \). Therefore, only external magnetic field \( \mathbf{H} \) should be provided as an input parameter to uniquely define the Zeeman energy term. \( \mathbf{H} \) can be constant in space or spatially varying.

2.2.1 Spatially constant \( \mathbf{H} \)

Let us start by assembling a simple simulation where \( \mathbf{H} \) does not vary in space. The sample is a “one-dimensional” chain of magnetic moments.

```python
[1]:
import oommfc as oc
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 a Hamiltonian, which consists of only Zeeman energy term.

```python
[2]:
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 minimise 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 magnetisation (\( \text{system.m} \)). We are going to make it random with \( M_s = 8 \times 10^5 \text{ Am}^{-1} \)

```python
[3]:
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)]

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

The magnetisation, we set is

```python
[4]:
system.m.k3d_vector(color_field=system.m.z)
Now, we can minimise the system’s energy by using oommfc.MinDriver.

```python
[5]:
md = oc.MinDriver()
md.drive(system)
```

Running OOMMF (ExeOOMMFRunner) [2020/06/12 00:36]... (1.7 s)

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

```python
[6]:
system.m.k3d_vector(color_field=system.m.z)
```

2.2.2 Spatially varying $\mathbf{H}$

There are two different ways how a parameter can be made spatially varying, by using: 1. Dictionary 2. discretisedfield.Field

**Dictionary**

In order to define a parameter using a dictionary, regions must be defined in the mesh. Regions 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.

```python
[7]:
p1 = (-10e-9, 0, 0)
p2 = (10e-9, 1e-9, 1e-9)
cell = (1e-9, 1e-9, 1e-9)
subregions = {'region1': df.Region(p1=(-10e-9, 0, 0), p2=(0, 1e-9, 1e-9)),
              'region2': 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. $\mathbf{H}$ is now defined as a dictionary:

```python
[8]:
H = {'region1': (1e6, 0, 0), 'region2': (0, 0, -1e6)}
```

The system object is

```python
[9]:
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 magnetisation is

```python
[10]:
system.m.k3d_vector(color_field=system.m.z)
```

Output()

After we minimise the energy

```python
[11]:
md.drive(system)
```

Running OOMMF (ExeOOMMFRunner) [2020/06/12 00:36]... (1.8 s)

The magnetisation is as we expected.
Let us say that the magnetisation field varies in space as

\[ H(x, y, z) = (c^2 x, 0, c) \]

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

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

The external magnetic field is

```python
H = df.Field(mesh, dim=3, value=H_fun, norm=1e6)
```

The system is

```python
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 magnetisation is

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

After the energy minimisation, the magnetisation is:

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

### 2.3 Tutorial 02: Exchange energy term

Interactive online tutorial:

Exchange energy density is computed as

\[ w_e = A(\nabla m)^2 \]

where \( m \) is the normalised (\(|m| = 1\)) magnetisation, and \( A \) is the exchange energy constant. Exchange energy term tends to align all magnetic moments parallel to each other. Direction in which they are going to point is not defined via exchange energy.

In oommfc, \( m \) is a part of the magnetisation field \( \text{system.m} \). Therefore, only exchange energy constant \( A \) should be provided as an input parameter to uniquely define the Exchange energy term. \( A \) can be constant in space or spatially varying.
2.3.1 Spatially constant $A$

Let us start by assembling a simple simulation where $A$ does not vary in space. The sample is a “one-dimensional” chain of magnetic moments.

```
[1]: import oommfc as oc
    import discretisedfield as df
    import micromagneticmodel as mm

    pl = (-10e-9, 0, 0)
    p2 = (10e-9, 1e-9, 1e-9)
    cell = (1e-9, 1e-9, 1e-9)

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

    The system has a Hamiltonian, which consists of only exchange energy term.

[2]: A = 1e-12 # exchange energy constant (J/m)
    system = mm.System(name='exchange_constant_A')
    system.energy = mm.Exchange(A=A)

    We are going to minimise 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 magnetisation (system.m). We are going to make it random with $M_s = 8 \times 10^5$ Am$^{-1}$

[3]: 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)]

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

    The magnetisation, we set is

[4]: system.m.k3d_vector(color_field=system.m.z)
    Output()

    Now, we can minimise the system’s energy by using oommfc.MinDriver.

[5]: md = oc.MinDriver()
    md.drive(system)

    Running OOMMF (ExeOOMFRRunner) [2020/06/12 00:37]... (1.7 s)

    We expect that now all magnetic moments are aligned parallel to each other.

[6]: system.m.k3d_vector(color_field=system.m.z)
    Output()

    Finally, we can delete the files created by oommfc.
```
**2.3.2 Spatially varying \( A \)**

There are two different ways how a parameter can be made spatially varying, by using: 1. Dictionary 2. discretisedfield.Field

**Dictionary**

In order to define a parameter using a dictionary, regions must be defined in the mesh. Regions 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.

```python
subregions = {'region1': df.Region(p1=(-10e-9, 0, 0), p2=(0, 1e-9, 1e-9)),
              'region2': 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)
```

The regions we defined are:

```python
mesh.k3d_subregions()
```

```
Output()
```

Let us say there is no exchange energy (\( A = 0 \)) in region 1, whereas in region 2 \( A = 10^{-12} \) Jm\(^{-1}\). Unlike Zeeman and anisotropy energy terms, exchange energy constant is defined between cells. Therefore, it is necessary also to define the value of \( A \) between the two regions. This is achieved by adding another item to the dictionary with key 'region1:region2'. \( A \) is now defined as a dictionary:

```python
A = {'region1': 0, 'region2': 1e-12, 'region1:region2': 0.5e-12}
```

The system object is

```python
system = mm.System(name='exchange_dict_A')
system.energy = mm.Exchange(A=A)
system.m = df.Field(mesh, dim=3, value=m_fun, norm=Ms)
```

Its magnetisation is

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

```
Output()
```

After we minimise the energy

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

```
Running OOMMF (ExeOOMMFRunner) [2020/06/12 00:37]... (2.0 s)
```

The magnetisation is as we expected. The magnetisation remains random in region 1, and it is aligned in region 2.
discretisedfield.Field

Let us now define the exchange energy in the similar way as in the previous example, but this time using discretisedfield.Field:

\[
\mathbf{A}'(x, y, z) = \begin{cases} 
10^{-12} & x \leq 0 \\
10^{-20} & x > 0 
\end{cases}
\]

This time, it is not possible to define the exchange energy constant between cells, but only in cells. Therefore, the exchange energy constant is then computed as the average between two discretisation cells.

```python
[14]:
def A_fun(pos):
    x, y, z = pos
    if x <= 0:
        return 1e-12
    else:
        return 1e-20

The exchange energy constant is

[15]:
A = df.Field(mesh, dim=1, value=A_fun)

The system is

[16]:
system = mm.System(name='exchange_field_A')
system.energy = mm.Exchange(A=A)
system.m = df.Field(mesh, dim=3, value=m_fun, norm=Ms)

and its magnetisation is

[17]:
system.m.k3d_vector(color_field=system.m.z)
Output()

After the energy minimisation, the magnetisation is:

[18]:
md.drive(system)
system.m.k3d_vector(color_field=system.m.z)

Running OOMMF (ExeOOMMFRunner) [2020/06/12 00:37]... (1.8 s)
Output()

2.4 Tutorial 03: Uniaxial anisotropy energy term

Uniaxial anisotropy energy density is computed as

\[ w_{ua} = -K(m \cdot u)^2 \]

where \(m\) is the normalised (\(|m| = 1\)) magnetisation, \(K\) is the uniaxial anisotropy constant, and \(u\) is the anisotropy axis. Uniaxial anisotropy energy term tends to align all magnetic moments parallel or antiparallel to the anisotropy axis.

In oommfc, \(m\) is a part of the magnetisation field \(system.m\). Therefore, only uniaxial anisotropy constant \(K\) and axis \(u\) should be provided as input parameters to uniquely define the energy term. Both \(K\) and \(u\) can be constant in space or spatially varying.
2.4.1 Spatially constant $K$ and $u$

Let us start by assembling a simple simulation where neither $K$ nor $u$ vary in space. The sample is a “one-dimensional” chain of magnetic moments.

```python
import oommfc as oc
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 is

```python
mesh.k3d()
```

```
Output()
```

The system has a Hamiltonian, which consists of only uniaxial anisotropy energy term.

```python
K = 1e5  # uniaxial anisotropy constant (J/m**3)
u = (0, 0, 1)  # uniaxial anisotropy axis
system = mm.System(name='uniaxialanisotropy_constant_K_u')
system.energy = mm.UniaxialAnisotropy(K=K, u=u)

We are going to minimise 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 magnetisation ($\text{system.m}$). We are going to make it random with $M_s = 8 \times 10^5$ Am$^{-1}$

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

The magnetisation, we set is

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

```
Output()
```

Now, we can minimise the system’s energy by using oommfc.MinDriver.

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

Running OOMMF (ExeOOMMFRunner) [2020/07/10 10:17]... (2.4 s)

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

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

```
Output()
```
2.4.2 Spatially varying $K$

There are two different ways how a parameter can be made spatially varying, by using: 1. Dictionary 2. discretisedfield.Field

**Dictionary**

In order to define a parameter using a dictionary, regions must be defined in the mesh. Regions 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.

```python
[8]: p1 = (-10e-9, 0, 0)
p2 = (10e-9, 1e-9, 1e-9)
cell = (1e-9, 1e-9, 1e-9)
subregions = {'region1': df.Region(p1=(-10e-9, 0, 0), p2=(0, 1e-9, 1e-9)),
              'region2': 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)
```

```python
[9]: mesh.k3d_subregions()
```

Let us say that there is no uniaxial anisotropy ($K = 0$) in region 1, whereas in region 2 it is $K = 10^5$ Jm$^{-3}$. $u$ is in the $z$-direction. $K$ is now defined as a dictionary:

```python
[10]: K = {'region1': 0, 'region2': 1e5}
```

The system object is

```python
[11]: system = mm.System(name='uniaxialanisotropy_dict_K')
system.energy = mm.UniaxialAnisotropy(K=K, u=u)
system.m = df.Field(mesh, dim=3, value=m_fun, norm=Ms)
```

Its magnetisation is

```python
[12]: system.m.k3d_vector(color_field=system.m.z)
```

After we minimise the energy

```python
[13]: md.drive(system)
```

Running OOMMF (ExeOOMMFRunner) [2020/07/10 10:17]... (2.8 s)

```python
[14]: system.m.k3d_vector(color_field=system.m.z)
```

The magnetisation is as we expected.
discretisedfield.Field

Let us define the spatially varying uniaxial anisotropy, so that

\[
\begin{cases}
  (0, 0, 1) & x \leq 0 \\
  (1, 0, 0) & x > 0
\end{cases}
\]

The value of \( u \) for the spatially varying anisotropy is set using a Python function.

```python
[15]:
def u_fun(pos):
    x, y, z = pos
    if x <= 0:
        return (0, 0, 1)
    else:
        return (1, 0, 0)
```

The uniaxial anisotropy parameters are

```python
[16]:
K = 1e5
u = df.Field(mesh, dim=3, value=u_fun)
```

The system is

```python
[17]:
system = mm.System(name='uniaxialanisotropy_field_u')
system.energy = mm.UniaxialAnisotropy(K=K, u=u)
system.m = df.Field(mesh, dim=3, value=m_fun, norm=Ms)
```

and its magnetisation is

```python
[18]:
system.m.k3d_vector(color_field=system.m.z)
Output()
```

After the energy minimisation, the magnetisation is:

```python
[19]:
md.drive(system)
system.m.k3d_vector(color_field=system.m.z)
```

Running OOMMF (ExeOOMMFRunner) [2020/07/10 10:17]... (2.7 s)

Output()

### 2.4.3 Higher-order uniaxial anisotropy

In order to define higher-order anisotropy term:

\[
\omega_{ua} = -K_1(m \cdot u)^2 - K_2(m \cdot u)^4
\]

anisotropy constants \( K_1 \) and \( K_2 \) must be passed. Similar to previous example, all parameters (\( K_1, K_2, \) and \( u \)) can be spatially varying (dictionary or field).

```python
[20]:
K1 = 1e5
K2 = 1e3
u = df.Field(mesh, dim=3, value=u_fun)
```

```python
[20]:
system = mm.System(name='uniaxialanisotropy_higher_order')
system.energy = mm.UniaxialAnisotropy(K1=K1, K2=K2, u=u)
system.m = df.Field(mesh, dim=3, value=m_fun, norm=Ms)
```
2.5 Tutorial 04: Dzyaloshinskii-Moriya energy term

Interactive online tutorial:

Dzyaloshinskii-Moriya energy density, depending on the crystallographic class, is computed as

\[ \mathcal{W}_{\text{DMI}} = \begin{cases} 
D \mathbf{m} \cdot (\nabla \times \mathbf{m}), & \text{for } T(O) \\
D (\mathbf{m} \cdot \nabla m_z - m_z \nabla \cdot \mathbf{m}), & \text{for } C_{nv} \\
D \mathbf{m} \cdot \left( \frac{\partial \mathbf{m}}{\partial x} \times \hat{x} - \frac{\partial \mathbf{m}}{\partial y} \times \hat{y} \right), & \text{for } D_{2d}
\end{cases} \]

where \( \mathbf{m} \) is the normalised (\( |\mathbf{m}| = 1 \)) magnetisation, and \( D \) is the DM energy constant. DMI energy term tends to align neighbouring magnetic moments perpendicular to each other.

In oommfc, \( \mathbf{m} \) is a part of the magnetisation field \( \text{system.m} \). Therefore, only DMI energy constant \( D \) should be provided as an input parameter to uniquely define the Exchange energy term. \( D \) can be constant in space or spatially varying.

2.5.1 Spatially constant \( D \)

Let us start by assembling a simple simple simulation where \( D \) does not vary in space. The sample is a “one-dimensional” chain of magnetic moments. We are going to choose \( C_{nv} \) as the crystallographic class.

```python
[1]: import oommfc as oc
    import discretisedfield as df
    import micromagneticmodel as mm
    %matplotlib inline

    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

[2]: mesh.k3d()

    Output()

    The system has a Hamiltonian, which consists of only DMI energy term.

[3]: D = 1e-3  # Dzyaloshinksii-Moriya energy constant (J/m**2)
    system = mm.System(name='dmi_constant_D')
    system.energy = mm.DMI(D=D, crystalclass='Cnv')
```
We are going to minimise 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 magnetisation (system.m). We are going to make it random with $M_s = 8 \times 10^5 \text{Am}^{-1}$

```python
import random
import discretisedfield as df

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

def m_fun(pos):
    """Return random 3d vectors for initial random magnetisation""
    return [2*random.random()-1, 2*random.random()-1, 2*random.random()-1]

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

The magnetisation, we have set as initial values looks like:

```
import random
import discretisedfield as df

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

def m_fun(pos):
    """Return random 3d vectors for initial random magnetisation""
    return [2*random.random()-1, 2*random.random()-1, 2*random.random()-1]

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

Now, we can minimise the system’s energy by using oommfc.MinDriver.

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

We expect that now all magnetic moments are aligned orthogonally to each other.

```
import random
import discretisedfield as df

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

def m_fun(pos):
    """Return random 3d vectors for initial random magnetisation""
    return [2*random.random()-1, 2*random.random()-1, 2*random.random()-1]

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

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

```
import random
import discretisedfield as df

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

def m_fun(pos):
    """Return random 3d vectors for initial random magnetisation""
    return [2*random.random()-1, 2*random.random()-1, 2*random.random()-1]

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

```
import random
import discretisedfield as df

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

def m_fun(pos):
    """Return random 3d vectors for initial random magnetisation""
    return [2*random.random()-1, 2*random.random()-1, 2*random.random()-1]

system.m = df.Field(mesh, dim=3, value=m_fun, norm=Ms)
```
2.5.2 Spatially varying $D$

In the case of DMI, there is only one way how a parameter can be made spatially varying - using a dictionary.

In order to define a parameter using a dictionary, regions must be defined in the mesh. Regions 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.

```python
[p1 = (-10e-9, 0, 0)
p2 = (10e-9, 1e-9, 1e-9)
cell = (1e-9, 1e-9, 1e-9)
subregions = {'region1': df.Region(p1=(-10e-9, 0, 0), p2=(0, 1e-9, 1e-9)),
              'region2': 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)
```

The regions we have defined are:

```python
mesh.k3d_subregions()
```

Output()

Let us say there is no DMI energy ($D = 0$) in region 1, whereas in region 2 $D = 10^{-3}$ Jm$^{-2}$. Unlike Zeeman and anisotropy energy terms, the DMI energy constant is defined between cells. Therefore, it is necessary to also define the value of $D$ between the two regions. This is achieved by adding another item to the dictionary with key 'region1:region2'. The object $D$ is now defined as a dictionary:

```python
D = {'region1': 0, 'region2': 1e-3, 'region1:region2': 0.5e-3}
```

The system object is

```python
system = mm.System(name='dmi_dict_D')
system.energy = mm.DMI(D=D, crystalclass='Cnv')
system.m = df.Field(mesh, dim=3, value=m_fun, norm=Ms)
```

Its initial (and random) magnetisation is

```python
system.m.k3d_vector(color_field=system.m.z)
system.m.plane('y').mpl()
```
After we minimise the energy

\[ \text{Running OOMMF (ExeOOMMFRunner) [2020/06/12 00:41]... (1.8 s)} \]

The magnetisation is as we expected. The magnetisation remains random in region 1, and it is orthogonally aligned in region 2.

\[ \text{system.m.k3d_vector(color_field=system.m.z)} \]
\[ \text{system.m.plane('y').mpl()} \]

2.6 Tutorial 05: Cubic anisotropy energy term

Interactive online tutorial:

Cubic anisotropy energy density is computed as

\[ w_{ca} = -K_1[(m \cdot u_1)^2(m \cdot u_2)^2 + (m \cdot u_2)^2(m \cdot u_3)^2 + (m \cdot u_1)^2(m \cdot u_3)^2] \]
where \( \mathbf{m} \) is the normalised \(|\mathbf{m}| = 1\) magnetisation, \( K_1 \) is the cubic anisotropy constant, and \( u_1 \) and \( u_2 \) are the anisotropy axes. Cubic anisotropy energy term tends to align all magnetic moments parallel or antiparallel to one of the three anisotropy axes.

In oommfc, \( \mathbf{m} \) is a part of the magnetisation field \( \text{system.m} \). Therefore, only cubic anisotropy constant \( K_1 \) and axes \( u_1 \) and \( u_2 \) should be provided as input parameters to uniquely define the energy term. All parameters can be constant in space or spatially varying.

### 2.6.1 Spatially constant \( K_1 \) and \( u \)

Let us start by assembling a simple simple simulation where neither \( K \) nor \( u \) vary in space. The sample is a “one-dimensional” chain of magnetic moments.

```python
import oommfc as oc
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 is

```python
# NBVAL_IGNORE_OUTPUT
mesh.k3d()
```

The system has a Hamiltonian, which consists of only cubic anisotropy energy term.

```python
K = 1e5  # cubic anisotropy constant (J/m**3)
u1 = (0, 0, 1)  # cubic anisotropy axis
u2 = (0, 1, 0)  # cubic anisotropy axis
system = mm.System(name='cubicanisotropy_constant_K_u')
system.energy = mm.CubicAnisotropy(K=K, u1=u1, u2=u2)
```

We are going to minimise 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 magnetisation \( \text{system.m} \). We are going to make it random with \( M_s = 8 \times 10^5 \text{Am}^{-1} \)

```python
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)]

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

The magnetisation, we set is

```python
# NBVAL_IGNORE_OUTPUT
system.m.k3d_vector(color_field=system.m.z)
```

2.6. Tutorial 05: Cubic anisotropy energy term
Now, we can minimise the system’s energy by using `oommf.MinDriver`.

```python
[6]: md = oc.MinDriver()
drive(system)
```

Running OOMMF (ExeOOMMFRunner) [2020/06/30 13:04]... (2.1 s)

We expect that now all magnetic moments are aligned parallel or antiparallel to the anisotropy axes.

```python
[7]: # NBVAL_IGNORE_OUTPUT
system.m.k3d_vector(color_field=system.m.z)
```

2.6.2 Spatially varying $K_1$

There are two different ways how a parameter can be made spatially varying, by using: 1. Dictionary 2. `discretisedfield.Field`

**Dictionary**

In order to define a parameter using a dictionary, regions must be defined in the mesh. Regions 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.

```python
[8]: p1 = (-10e-9, 0, 0)
p2 = (10e-9, 1e-9, 1e-9)
cell = (1e-9, 1e-9, 1e-9)
subregions = {'region1': df.Region(p1=(-10e-9, 0, 0), p2=(0, 1e-9, 1e-9)),
              'region2': 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)
```

```python
[9]: # NBVAL_IGNORE_OUTPUT
mesh.k3d_subregions()
```

Let us say that there is no cubic anisotropy ($K = 0$) in region 1, whereas in region 2 it is $K = 10^5 \text{ Jm}^{-3}$. $K$ is now defined as a dictionary:

```python
[10]: K = {'region1': 0, 'region2': 1e5}
```

The system object is

```python
[11]: system = mm.System(name='cubicanisotropy_dict_K')
system.energy = mm.CubicAnisotropy(K=K, u1=u1, u2=u2)
system.m = df.Field(mesh, dim=3, value=m_fun, norm=Ms)
```

Its magnetisation is

```python
[12]: # NBVAL_IGNORE_OUTPUT
system.m.k3d_vector(color_field=system.m.z)
```
After we minimise the energy

```python
[13]: md.drive(system)
```

Running OOMMF (ExeOOMMFRunner) [2020/06/30 13:04]... (2.5 s)

The magnetisation is as we expected.

```python
[14]: # NBVAL_IGNORE_OUTPUT
system.m.k3d_vector(color_field=system.m.z)
```

```plaintext
Output()
```

Let us define the spatailly varying uniaxial anisotropy, so that

$$K(x, y, z) = \begin{cases} 
0 & x \leq 0 \\
1e5 & x > 0 
\end{cases}$$

The value of $K$ for the spatially varying anisotropy is set using a Python function.

```python
[15]: def K_fun(pos):
    x, y, z = pos
    if x <= 0:
        return 0
    else:
        return 1e5
```

The uniaxial anisotropy parameters are

```plaintext
[16]: K = df.Field(mesh, dim=1, value=K_fun)
```

The system is

```plaintext
[17]: system = mm.System(name='cubicanisotropy_field_u')
system.energy = mm.CubicAnisotropy(K=K, u1=u1, u2=u2)
```

and its magnetisation is

```plaintext
[18]: # NBVAL_IGNORE_OUTPUT
system.m.k3d_vector(color_field=system.m.z)
```

```plaintext
Output()
```

After the energy minimisation, the magnetisation is:

```plaintext
[19]: # NBVAL_IGNORE_OUTPUT
md.drive(system)
```

Running OOMMF (ExeOOMMFRunner) [2020/06/30 13:04]... (2.2 s)

```plaintext
Output()
```
2.7 RKKY

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 initialise our magnetisation as a random state.

```python
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.

```python
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))}
```

```python
mesh = df.Mesh(region, n=(20, 20, 11), subregions=subregions)
```

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.

```python
system = mm.System(name='rkky')
```

```python
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))
```

```python
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 magnetisation is:

```python
system.m.plane('y').mpl(figsize=(15, 4))
```
And the energy equation:

\[
-\mathbf{A} \cdot \nabla^2 \mathbf{m} + \text{RKKY(bottom, top)} - K(\mathbf{m} \cdot \mathbf{u})^2
\]

Now we can relax the system, and plot its magnetisation.

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

### 2.8 Adding multiple terms of the same class

Here we demonstrate how to have multiple energy terms of the same class in the energy equation. For the sample, we choose a one-dimensional chain of magnetic moments.
```python
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 is

```}

```python
mesh.k3d()

Output()

The system has an energy equation, which consists of two Zeeman energy terms.

```}

```python
H1 = (0, 0, 1e6)
H2 = (1e6, 0, 0)
system = mm.System(name='multiple_terms')

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

```}

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

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:

```}

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

We are going to minimise 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 magnetisation (system.m). We are going to make it random with $M_s = 8 \times 10^5$ Am$^{-1}$

```}

```python
import random

random.seed(1)
Ms = 8e5  # saturation magnetisation (A/m)

def m_fun(pos):
    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 minimise the system’s energy by using oommfc.MinDriver.

```}

```python
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).

We can see that magnetisation is aligned with the sum of fields $H_1+H_2$. Finally, let us have a look at the table:

```
max_mxHxm E delta_E bracket_count line_min_count
0 0.003334 -2.843445e-20 -7.222237e-35 27.0 5.0
conjugate_cycle_count cycle_count cycle_sub_count energy_calc_count
0 21.0 23.0 2.0 33.0
E_zeeman1 E_zeeman2 iteration stage_iteration stage mx
0 -1.421723e-20 -1.421723e-20 30.0 30.0 0.0 0.707107
my mz
0 9.591067e-11 0.707107
```

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

### 2.9 Negative exchange energy constant $A$

In this tutorial, we show how to set up 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 initialise our magnetisation as a random state.

```
import discretisedfield as df
import micromagneticmodel as mm
import random
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 consists of three identical magnetic layers, which are (for some reason) coupled antiferomagnetically.
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, $r1:r2$.

As we mentioned previously, we are going to initialise the system using a random state.
We can see that all three regions are uniformly magnetised, but mutually coupled antiferromagnetically.

2.10 Tutorial 07: Standard problem 3

Interactive online tutorial:

2.10.1 Problem specification

This problem is to calculate a single domain limit of a cubic magnetic particle. This is the size $L$ of equal energy for the so-called flower state (which one may also call a splayed state or a modified single-domain state) on the one hand, and the vortex or curling state on the other hand.

Geometry:

A cube with edge length, $L$, expressed in units of the intrinsic length scale, $l_{ex} = \sqrt{A/K_m}$, where $K_m$ is a magneto-static energy density, $K_m = \frac{1}{2}\mu_0 M_s^2$.

Material parameters:

- uniaxial anisotropy $K_u$ with $K_u = 0.1K_m$, and with the easy axis directed parallel to a principal axis of the cube (0, 0, 1),
- exchange energy constant is $A = \frac{1}{2}\mu_0 M_s^2 l_{ex}^2$.

More details about the standard problem 3 can be found in Ref. 1.

2.10.2 Simulation

Firstly, we import all necessary modules.

```python
[1]:
import discretisedfield as df
import micromagneticmodel as mm
import oommfc as oc
```

The following two functions are used for initialising the system’s magnetisation [1].
import numpy as np

# Function for initialising the flower state.
def m_init_flower(pos):
    x, y, z = pos[0]/1e-9, pos[1]/1e-9, pos[2]/1e-9
    mx = 0
    my = 2*z - 1
    mz = -2*y + 1
    norm_squared = mx**2 + my**2 + mz**2
    if norm_squared <= 0.05:
        return (1, 0, 0)
    else:
        return (mx, my, mz)

# Function for initialising the vortex state.
def m_init_vortex(pos):
    x, y, z = pos[0]/1e-9, pos[1]/1e-9, pos[2]/1e-9
    mx = 0
    my = np.sin(np.pi/2 * (x-0.5))
    mz = np.cos(np.pi/2 * (x-0.5))
    return (mx, my, mz)

The following function is used for convenience. It takes two arguments:

- \( L \) - the cube edge length in units of \( l_{\text{ex}} \), and
- the function for initialising the system’s magnetisation.

It returns the relaxed system object.

Please refer to other tutorials for more details on how to create system objects and drive them using specific drivers.
Relaxed magnetisation states

Now, we show the magnetisation configurations of two relaxed states.

Vortex state:

```python
%matplotlib inline
system = minimise_system_energy(8, m_init_vortex)
system.m.plane('y').mpl()
```

Flower state:

```python
system = minimise_system_energy(8, m_init_flower)
system.m.plane('y').mpl()
```
Energy crossing

Now, we can plot the energies of both vortex and flower states as a function of cube edge length. This will give us an idea where the state transition occurs.

```python
import matplotlib.pyplot as plt
plt.plot(L_array, vortex_energies, 'o-', label='vortex')
plt.plot(L_array, flower_energies, 'o-', label='flower')
plt.xlabel('L (nm)')
plt.ylabel('E')
plt.xlim([8.0, 9.0])
plt.grid()
plt.legend()
```

(continues on next page)
We now know that the energy crossing occurs between $8L_{\text{ex}}$ and $9L_{\text{ex}}$, so a bisection algorithm can be used to find the exact crossing.

```python
from scipy.optimize import bisect
def energy_difference(L):
    vortex = minimise_system_energy(L, m_init_vortex)
    flower = minimise_system_energy(L, m_init_flower)
    return vortex.table.data.tail(1)['E'][0] - flower.table.data.tail(1)['E'][0]
cross_section = bisect(energy_difference, 8, 9, xtol=0.1)
print(f'The transition between vortex and flower states occurs at (cross_section)\times L_{\text{ex}} \rightarrow')
```

We now know that the energy crossing occurs between $8L_{\text{ex}}$ and $9L_{\text{ex}}$, so a bisection algorithm can be used to find the exact crossing.
2.10.3 References


2.11 Tutorial 08: Standard problem 4

Interactive online tutorial:

2.11.1 Problem specification

The sample is a thin film cuboid with dimensions:

- length $l_x = 500$ nm,
- width $l_y = 125$ nm, and
- thickness $l_z = 3$ nm.

The material parameters (similar to permalloy) are:

- exchange energy constant $A = 1.3 \times 10^{-11} \text{ J/m}$,
- magnetisation saturation $M_s = 8 \times 10^5 \text{ A/m}$.

Magnetisation dynamics are governed by the Landau-Lifshitz-Gilbert equation

$$\frac{dm}{dt} = -\gamma_0 (m \times H_{\text{eff}}) + \alpha \left( m \times \frac{dm}{dt} \right)$$

where $\gamma_0 = 2.211 \times 10^5 \text{ m A}^{-1} \text{ s}^{-1}$ and Gilbert damping $\alpha = 0.02$.

In the standard problem 4, the system is first relaxed at zero external magnetic field and then, starting from the obtained equilibrium configuration, the magnetisation dynamics are simulated for two external magnetic fields $B_1 = (-24.6, 4.3, 0.0) \text{ mT}$ and $B_2 = (-35.5, -6.3, 0.0) \text{ mT}$.

More detailed specification of Standard problem 4 can be found in Ref. 1.
2.11.2 Simulation

In the first step, we import the required discretisedfield and oommfc modules.

```python
import discretisedfield as df
import micromagneticmodel as mm
import oommfc as oc
```

Now, we can set all required geometry and material parameters.

```python
# Geometry
lx = 500e-9  # x dimension of the sample (m)
ly = 125e-9  # y dimension of the sample (m)
lz = 3e-9    # sample thickness (m)

# Material (permalloy) parameters
Ms = 8e5     # saturation magnetisation (A/m)
A = 1.3e-11  # exchange energy constant (J/m)

# Dynamics (LLG equation) parameters
gamma0 = 2.211e5  # gyromagnetic ratio (m/As)
alpha = 0.02      # Gilbert damping
```

2.11.3 First stage

In the first stage, we need to relax the system at zero external magnetic field.

We choose stdprob4 to be the name of the system. This name will be used to name all output files created by OOMMF.

```python
system = mm.System(name='stdprob4')
```

In order to completely define the micromagnetic system, we need to provide:

1. energy $E$
2. dynamics $\frac{dm}{dt}$
3. magnetisation $m$

The mesh is created by providing two points $p_1$ and $p_2$ between which the mesh domain spans and the size of a discretisation cell. We choose the discretisation to be $(5,5,3)$ nm.

```python
cell = (5e-9, 5e-9, 3e-9)  # mesh discretisation (m)
mesh = df.Mesh(p1=(0, 0, 0), p2=(lx, ly, lz), cell=cell)  # Create a mesh object.
```

We can visualise the mesh domain and a discretisation cell:

```python
%matplotlib inline
mesh.k3d()
```

Hamiltonian: In the second step, we define the system’s Hamiltonian. In this standard problem, the Hamiltonian contains only exchange and demagnetisation energy terms. Please note that in the first simulation stage, there is no applied external magnetic field. Therefore, we do not add Zeeman energy term to the Hamiltonian.
We can check what is the continuous model of system’s Hamiltonian.

\[
A \left[ (\nabla m_x)^2 + (\nabla m_y)^2 + (\nabla m_z)^2 \right] - \frac{1}{2} \mu_0 M_s m \cdot H
\]

**Dynamics:** Similarly, the system’s dynamics is defined by providing precession and damping terms (LLG equation).

\[
-\gamma_0 \frac{\alpha}{1+\alpha^2} m \times H_{\text{eff}} - \gamma_0 \alpha \frac{\alpha}{1+\alpha^2} m \times (m \times H_{\text{eff}})
\]

**Magnetisation:** Finally, we have to provide the magnetisation configuration that is going to be relaxed subsequently. We choose the uniform configuration in (1, 0.25, 0.1) direction, and as norm (magnitude) we set the magnetisation saturation \( M_s \). In order to create the magnetisation configuration, we create a Field object from the discretisedfield module.

Now, the system is fully defined.

**Energy minimisation:** The system (its magnetisation) is evolved using a particular driver. In the first stage, we need to relax the system - minimise its energy. Therefore, we create MinDriver object and drive the system using its drive method.

The system is now relaxed. We can plot it in various ways. We start with simple plot using default settings:
With a little bit of additional code, we can fine tune the plot as appropriate for the relevant results:

```python
import matplotlib.pyplot as plt

# make figure larger
fig, ax = plt.subplots(figsize=(20, 5))

# plot vectors on grid of 20 x 5 over the numerical resolution
system.m.plane('z', n=(20, 5)).mpl_vector(ax=ax)

# add colouring for mx-component to this plot
system.m.x.plane('z').mpl_scalar(ax=ax)
```
We can now obtain some numerical data characteristic to the magnetisation field as is useful for standard problem 4:

```python
print('The average magnetisation is {:.16f}'.format(system.m.average))
print('The magnetisation at the mesh centre {:.16f}'.format(
    system.m.mesh.region.centre, system.m(system.m.mesh.region.centre)))
```

The average magnetisation is (773766.2112987704, 99856.85901465818, -0.003159960962047351).
The magnetisation at the mesh centre (2.5e-07, 6.25e-08, 1.5e-09) is (799979.276019471, -5758.29309460144, -0.00502709656498145).

### 2.11.4 Second stage: field $\mathbf{B}_1$

In the second stage, we need to apply an external magnetic field $\mathbf{B}_1 = (-24.6, 4.3, 0.0)$ mT to the system. In other words, we have to add Zeeman energy term to the Hamiltonian.

```python
# Add Zeeman energy term to the Hamiltonian
H1 = (-24.6e-3/mm.consts.mu0, 4.3e-3/mm.consts.mu0, 0.0)
system.energy += mm.Zeeman(H=H1)
```

If we now inspect the Hamiltonian, we see that an additional Zeeman term is added.

```python
system.energy
```

Finally, we can run the simulation using `TimeDriver` this time. We run the magnetisation evolution for $t = 1$ ns, during which we save the system’s state $n = 200$ times.

```python
t = 1e-9  # simulation time (s)
n = 200   # number of data saving steps

td = oc.TimeDriver()  # create time driver
td.drive(system, t=t, n=n)  # drive the system
```

Running OOMMF (ExeOOMMFRunner) [2020/06/14 11:27]... (7.1 s)

### Postprocessing

When we drove the system using the `TimeDriver`, we specified that we want to save the magnetisation configuration $n = 200$ times. A detailed table of all computed parameters from the last simulation can be shown from the datatable (`system.dt`), which is a pandas dataframe [2].
For instance, if we want to show the last 10 rows in the table, we run:

```python
system.table.data.tail()
```

<table>
<thead>
<tr>
<th></th>
<th>E</th>
<th>E_calc_count</th>
<th>max_dm/dt</th>
<th>dE/dt</th>
<th>delta_E</th>
</tr>
</thead>
<tbody>
<tr>
<td>195</td>
<td>-2.676342e-18</td>
<td>3736.0</td>
<td>1162.533933</td>
<td>-1.697359e-09</td>
<td>-2.278389e-21</td>
</tr>
<tr>
<td>196</td>
<td>-2.685463e-18</td>
<td>3755.0</td>
<td>1260.317262</td>
<td>-1.935225e-09</td>
<td>-2.621610e-21</td>
</tr>
<tr>
<td>197</td>
<td>-2.695498e-18</td>
<td>3774.0</td>
<td>1384.782389</td>
<td>-2.057789e-09</td>
<td>-2.813912e-21</td>
</tr>
<tr>
<td>198</td>
<td>-2.705827e-18</td>
<td>3793.0</td>
<td>1351.611377</td>
<td>-2.053171e-09</td>
<td>-2.832075e-21</td>
</tr>
<tr>
<td>199</td>
<td>-2.715833e-18</td>
<td>3812.0</td>
<td>1191.086615</td>
<td>-1.931645e-09</td>
<td>-2.687310e-21</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>E_exchange</th>
<th>max_spin_ang</th>
<th>stage_max_spin_ang</th>
<th>run_max_spin_ang</th>
</tr>
</thead>
<tbody>
<tr>
<td>195</td>
<td>9.548545e-20</td>
<td>3.913558</td>
<td>4.281302</td>
<td>29.612838</td>
</tr>
<tr>
<td>196</td>
<td>8.637446e-20</td>
<td>4.157989</td>
<td>4.157989</td>
<td>29.612838</td>
</tr>
<tr>
<td>197</td>
<td>8.039753e-20</td>
<td>4.727728</td>
<td>4.727728</td>
<td>29.612838</td>
</tr>
<tr>
<td>198</td>
<td>7.586011e-20</td>
<td>4.669401</td>
<td>4.772024</td>
<td>29.612838</td>
</tr>
<tr>
<td>199</td>
<td>7.141300e-20</td>
<td>4.354488</td>
<td>4.669401</td>
<td>29.612838</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>E_demag</th>
<th>E_zeeman</th>
<th>iteration</th>
<th>stage_iteration</th>
<th>stage</th>
<th>mx</th>
</tr>
</thead>
<tbody>
<tr>
<td>195</td>
<td>-8.530166e-19</td>
<td>-3.624844e-18</td>
<td>784.0</td>
<td>3.0</td>
<td>195.0</td>
<td>-0.984260</td>
</tr>
<tr>
<td>196</td>
<td>-8.844369e-19</td>
<td>-3.656275e-18</td>
<td>788.0</td>
<td>3.0</td>
<td>196.0</td>
<td>-0.987086</td>
</tr>
<tr>
<td>197</td>
<td>-9.074607e-19</td>
<td>-3.683357e-18</td>
<td>792.0</td>
<td>3.0</td>
<td>197.0</td>
<td>-0.988092</td>
</tr>
<tr>
<td>198</td>
<td>-9.220444e-19</td>
<td>-3.703732e-18</td>
<td>796.0</td>
<td>3.0</td>
<td>198.0</td>
<td>-0.986964</td>
</tr>
<tr>
<td>199</td>
<td>-9.291427e-19</td>
<td>-3.716389e-18</td>
<td>800.0</td>
<td>3.0</td>
<td>199.0</td>
<td>-0.983765</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>my</th>
<th>mz</th>
<th>last_time_step</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>195</td>
<td>-0.010971</td>
<td>0.033987</td>
<td>1.374339e-12</td>
<td>9.80000e-10</td>
</tr>
<tr>
<td>196</td>
<td>0.021592</td>
<td>0.039286</td>
<td>1.37845e-12</td>
<td>9.85000e-10</td>
</tr>
<tr>
<td>197</td>
<td>0.057824</td>
<td>0.042604</td>
<td>1.37937e-12</td>
<td>9.90000e-10</td>
</tr>
<tr>
<td>198</td>
<td>0.095870</td>
<td>0.043794</td>
<td>1.374355e-12</td>
<td>9.95000e-10</td>
</tr>
<tr>
<td>199</td>
<td>0.133793</td>
<td>0.042832</td>
<td>1.375196e-12</td>
<td>1.00000e-09</td>
</tr>
</tbody>
</table>

Finally, we want to plot the average magnetisation configuration $my$ as a function of time $t$:

```python
myplot = system.table.data.plot('t', 'my')
```

We can plot another snap shot of the magnetisation at the end of the run, which shows the fluctuations in the simulation at this time:

```
2.11. Tutorial 08: Standard problem 4 63
```
2.11.5 References


2.12 Tutorial 09: Standard problem 5

Interactive online tutorial:

2.12.1 Problem specification

The sample is a thin film cuboid with dimensions:

- length \( l_x = 100 \) nm,
- width \( l_y = 100 \) nm, and
- thickness \( l_z = 10 \) nm.

The material parameters (similar to permalloy) are:

- exchange energy constant \( A = 1.3 \times 10^{-11} \) J/m,
- magnetisation saturation \( M_s = 8 \times 10^5 \) A/m.

Dynamics parameters are: \( \gamma_0 = 2.211 \times 10^5 \) m A\(^{-1}\) s\(^{-1}\) and Gilbert damping \( \alpha = 0.02 \).

In the standard problem 5, the system is firstly relaxed at zero external magnetic field, starting from the vortex state. Secondly spin-polarised current is applied in the \( x \) direction with \( u_x = -72.35 \) and \( \beta = 0.05 \).

More detailed specification of Standard problem 5 can be found in Ref. 1.
2.12.2 Simulation

In the first step, we import the required discretisedfield and oommfc modules.

```python
import oommfc as oc
import discretisedfield as df
import micromagneticmodel as mm
```

Now, we can set all required geometry and material parameters.

```python
# Geometry
lx = 100e-9  # x dimension of the sample (m)
ly = 100e-9  # y dimension of the sample (m)
lz = 10e-9   # sample thickness (m)
dx = dy = dz = 5e-9  # discretisation cell (nm)

# Material (permalloy) parameters
Ms = 8e5       # saturation magnetisation (A/m)
A = 1.3e-11   # exchange energy constant (J/m)

# Dynamics (LLG equation) parameters
gamma0 = 2.211e5 # gyromagnetic ratio (m/As)
alpha = 0.1     # Gilbert damping
ux = -72.35    # velocity in x direction
beta = 0.05    # non-adiabatic STT parameter
```

As usual, we create the system object with stdprob5 name.

```python
system = mm.System(name='stdprob5')
```

The mesh is created by providing two points p1 and p2 between which the mesh domain spans and the size of a discretisation cell. We choose the discretisation to be (5, 5, 5) nm.

```python
%matplotlib inline
region = df.Region(p1=(0, 0, 0), p2=(lx, ly, lz))
mesh = df.Mesh(region=region, cell=(dx, dy, dz))
mesh.k3d()
```

**Hamiltonian:** In the second step, we define the system’s Hamiltonian. In this standard problem, the Hamiltonian contains only exchange and demagnetisation energy terms. Please note that in the first simulation stage, there is no applied external magnetic field. Therefore, we do not add Zeeman energy term to the Hamiltonian.

```python
system.energy = mm.Exchange(A=A) + mm.Demag()
system.energy
```

```
A \left[ (\nabla m_x)^2 + (\nabla m_y)^2 + (\nabla m_z)^2 \right] - \frac{1}{2} \mu_0 M_s m \cdot H_d
```

**Magnetisation:** We initialise the system using the initial magnetisation function.

```python
def m_vortex(pos):
    x, y, z = pos[0]/1e-9-50, pos[1]/1e-9-50, pos[2]/1e-9
    return (-y, x, 10)

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

```python
system.m.plane(z=0).mpl()
```
Dynamics: In the first (relaxation) stage, we minimise the system’s energy and therefore we do not need to specify the dynamics equation.

Minimisation: Now, we minimise the system’s energy using MinDriver.

[7]: md = oc.MinDriver()
    md.drive(system)

Running OOMMF (ExeOOMMFRunner) [2020/06/14 11:28]... (2.1 s)

[8]: system.m.plane(z=0).mpl()

2.12.3 Spin-polarised current

In the second part of simulation, we need to specify the dynamics equation for the system.
Now, we can drive the system for 8 ns and save the magnetisation in $n = 100$ steps.

```python
[10]: td = oc.TimeDriver()
   td.drive(system, t=8e-9, n=100)
```

The vortex after 8 ns is now displaced from the centre.
2.12.4 References


2.13 Tutorial 10: FMR standard problem

Interactive online tutorial:

2.13.1 Problem specification

We choose a cuboidal thin film permalloy sample measuring $120 \times 120 \times 10 \text{nm}^3$. The choice of a cuboid is important as it ensures that the finite difference method employed by OOMMF does not introduce errors due to irregular boundaries that cannot be discretized well. We choose the thin film geometry to be thin enough so that the variation of magnetization dynamics along the out-of-film direction can be neglected. Material parameters based on permalloy are:

- exchange energy constant $A = 1.3 \times 10^{-11} \text{J/m}$,
- magnetisation saturation $M_s = 8 \times 10^5 \text{A/m}$,
- Gilbert damping $\alpha = 0.008$.

An external magnetic bias field with magnitude $80 \text{kA/m}$ is applied along the direction $e = (1, 0.715, 0)$. We choose the external magnetic field direction slightly off the sample diagonal in order to break the system’s symmetry and thus avoid degenerate eigenmodes. First, we initialize the system with a uniform out-of-plane magnetization $m_0 = (0, 0, 1)$. The system is allowed to relax for $5 \text{ns}$, which was found to be sufficient time to obtain a well-converged equilibrium magnetization configuration. We refer to this stage of simulation as the relaxation stage, and its final relaxed magnetization configuration is saved to serve as the initial configuration for the next dynamic stage. Because we want to use a well defined method that is supported by all simulation tools, we minimize the system’s energy by integrating the LLG equation with a large, quasistatic Gilbert damping $\alpha = 1$ for $5 \text{ns}$. In the next step (dynamic stage), a simulation is started using the equilibrium magnetization configuration from the relaxation stage as the initial configuration. Now, the direction of an external magnetic field is altered to $e = (1, 0.7, 0)$. This simulation stage runs for $T = 20 \text{ns}$ while the (average and spatially resolved) magnetization $M(t)$ is recorded every $\Delta t = 5 \text{ps}$. The Gilbert damping in this dynamic simulation stage is $\alpha = 0.008$.

Details of this standard problem specification can be found in Ref. 1.

2.13.2 Relaxation stage

Firstly, all required modules are imported.

```
[1]:
import oommfc as oc
import discretisedfield as df
import micromagneticmodel as mm
```

Now, we specify all simulation parameters.

```
[2]:
import numpy as np

lx = ly = 120e-9  # x and y dimensions of the sample(m)
lz = 10e-9       # sample thickness (m)
dx = dy = dz = 5e-9  # discretisation in x, y, and z directions (m)
```

(continues on next page)
Now, the system object can be created and mesh, magnetisation, hamiltonian, and dynamics are specified.

```python
mesh = df.Mesh(p1=(0, 0, 0), p2=(lx, ly, lz), cell=(dx, dy, dz))

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

system.energy = mm.Exchange(A=A) + mm.Demag() + mm.Zeeman(H=H)

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

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

Finally, the system is relaxed.

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

Running OOMMF (ExeOOMMFRunner) [2020/06/14 11:29]... (2.2 s)
```

We can now load the relaxed state to the Field object and plot the z slice of magnetisation.

```python
%matplotlib inline
system.m.plane('z', n=(10, 10)).mpl()
```

### 2.13.3 Dynamic stage

In the dynamic stage, we use the relaxed state from the relaxation stage.

```python
# Change external magnetic field.
H = 8e4 * np.array([0.81923192051904048, 0.57346234436332832, 0.0])

system.energy.zeeman.H = H
```

Finally, we run the multiple stage simulation using TimeDriver.
2.13.4 Postprocessing

From the obtained vector field samples, we can compute the average of magnetisation $y$ component and plot its time evolution.

```python
[8]: import matplotlib.pyplot as plt

t = system.table.data['t'].values
my = system.table.data['mx'].values

# Plot $<my>$ time evolution.
plt.figure(figsize=(8, 6))
plt.plot(t, my)
plt.xlabel('t (ns)')
plt.ylabel('my average')
plt.grid()
```

![Graph](image)

From the $<m_y>$ time evolution, we can compute and plot its Fourier transform.

```python
[9]: import scipy.fftpack

psd = np.log10(np.abs(scipy.fftpack.fft(my))**2)
```
f_axis = scipy.fftpack.fftfreq(4000, d=20e-9/4000)
plt.plot(f_axis/1e9, psd)
plt.xlim([6, 12])
plt.xlabel('f (GHz)')
plt.ylabel('Psa (a.u.)')
plt.grid()

2.13.5 References


2.14 Tutorial 11: Deriving energy values

Interactive online tutorial:

In this tutorial, we show how derived fields and values can be computed after the micromagnetic system is defined.

2.14.1 Simulation

First of all, as usual, we import oommfc and discretisedfield.

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

We define the cube mesh with edge length 10 nm and cell discretisation edge 1 nm.

   mesh = df.Mesh(p1=(0, 0, 0), p2=(10e-9, 10e-9, 10e-9), cell=(1e-9, 1e-9, 1e-9))
mesh.k3d()
Now we define the system object and its Hamiltonian.

```python
[3]:
    system = mm.System(name='system')
    A = 1e-11
    H = (0.1/mm.consts.mu0, 0, 0)
    K = 1e3
    u = (1, 1, 1)
    system.energy = mm.Exchange(A=A) +
        mm.Demag() +
        mm.Zeeman(H=H) +
        mm.UniaxialAnisotropy(K=K, u=u)
```

We will now initialise the system in (0, 0, 1) direction and relax the magnetisation.

```python
[4]:
    Ms = 8e5
    system.m = df.Field(mesh, dim=3, value=(0, 0, 1), norm=Ms)
```

### 2.14.2 Effective field

Total effective field is:

```python
[5]:
    oc.compute(system.energy.effective_field, system).plane('x').mpl()
```

Running OOMMF (ExeOOMMFRunner) [2020/06/12 00:54]... (2.2 s)

![Effective field diagram](image)

Whereas, the individual exchange effective field is:

```python
[6]:
    Hex_eff = oc.compute(system.energy.exchange.effective_field, system)
```

Running OOMMF (ExeOOMMFRunner) [2020/06/12 00:54]... (1.8 s)

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

```python
[7]:
    Hex_eff.average
```
2.14.3 Relax the system

```python
md = oc.MinDriver()
md.drive(system)
Running OOMMF (ExeOOMMFRunner) [2020/06/12 00:54]... (1.9 s)
```
Compute the energy (and demonstrate that the energy decreased) and plot its magnetisation:

```python
E = oc.compute(system.energy.energy, system)
print("The system's energy is \{\} J.".format(E))
```
```
Running OOMMF (ExeOOMMFRunner) [2020/06/12 00:54]... (1.8 s)
The system's energy is 5.35285533145e-20 J.
```

2.14.4 Computing energies of individual term

For instance, the exchange energy is:

```python
oc.compute(system.energy.exchange.energy, system)
Running OOMMF (ExeOOMMFRunner) [2020/06/12 00:54]... (1.9 s)
```
```
1.12170206436e-21
```
We can also check the sum of all individual energy terms and check if it is the same as the total energy.

```python
total_energy = 0
for term in system.energy:
    total_energy += oc.compute(term.energy, system)
print("The sum of energy terms is \{\} J.".format(total_energy))
print("The system's energy is \{\} J.".format(oc.compute(system.energy.energy, system)))
```
```
2.14. Tutorial 11: Deriving energy values 73
```
2.15 Tutorial 12: Calculating a stray field using an airbox method

Interactive online tutorial:

In order to calculate the stray field outside the sample, we have to define an “airbox” which is going to contain our sample. In this example we define a box with 100 nm edge length as a mesh which then contains a magnetic sample which is a cube with 50 nm dimensions. We achieve this by implementing a Python function for defining the Ms (norm_fun). Outside our sample the value of saturation magnetisation is zero.

```python
[1]: import discretisedfield as df
    import micromagneticmodel as mm
    import oommfc as oc
    %matplotlib inline

    region = df.Region(p1=(-100e-9, -100e-9, -100e-9), p2=(100e-9, 100e-9, 100e-9))
    mesh = df.Mesh(region=region, cell=(5e-9, 5e-9, 5e-9))

    def norm_fun(pos):
        x, y, z = pos
        if -50e-9 <= x <= 50e-9 and -50e-9 <= y <= 50e-9 and -50e-9 <= z <= 50e-9:
            return 8e5
        else:
            return 0

    system = mm.System(name='airbox_method')
    system.energy = mm.Exchange(A=1e-12) + mm.Demag()
    system.dynamics = mm.Precession(gamma0=mm.consts.gamma0) + mm.Damping(alpha=1)
    system.m = df.Field(mesh, dim=3, value=(0, 0, 1), norm=norm_fun)

    We can now plot the norm to confirm our definition.

[2]: system.m.norm.plane('z').mpl()
```
In the next step, we can relax the system and show its magnetisation.

```python
[3]: md = oc.MinDriver()
    md.drive(system)
    system.m.plane('z').mpl(figsize=(10, 10))
```

Running OOMMF (ExeOOMMFRunner) [2020/06/12 00:55]... (7.2 s)
Stray field can now be calculated as an effective field for the demagnetisation energy.

[4]: \texttt{stray\_field = oc.compute(system.energy.demag.effective\_field, system)}

Running OOMMF (ExeOOMMFRunner) [2020/06/12 00:55]... (2.2 s)

\texttt{stray\_field} is a \texttt{df.Field} and all operations characteristic to vector fields can be performed.

[9]: \texttt{stray\_field.plane('z').mpl(figsize=(8, 8), vector_scale=\_1e6)}
Interactive online tutorial:

In this tutorial, we compute and relax a skyrmion in an interfacial-DMI material in a confined disk like geometry.

We define mesh in cuboid through corner points \( p1 \) and \( p2 \), and discretisation cell size \( \text{cell} \).

```python
[2]:
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))
```

The mesh we defined is:

```python
[3]:
%matplotlib inline
mesh.k3d()
```

Now, we can define the system object by first setting up the Hamiltonian:
Disk geometry is set up by defining the saturation magnetisation (norm of the magnetisation field). For that, we define a function:

```python
Ms = 1.1e6

def Ms_fun(pos):
    """Function to set magnitude of magnetisation: zero outside cylindric shape, Ms inside cylinder.
    Cylinder radius is 50nm.
    ""
    x, y, z = pos
    if (x**2 + y**2)**0.5 < 50e-9:
        return Ms
    else:
        return 0
```

And the second function we need is the function to define the initial magnetisation which is going to relax to skyrmion.

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

The geometry is now:

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

and the initial magnetisation is:

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

(continues on next page)
Finally we can minimise the energy and plot the magnetisation.

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

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

Running OOMMF (ExeOOMMFRunner) [2020/06/12 00:57]... (1.9 s)
Finally we can sample and plot the magnetisation along the line:

```
[12]: system.m.z.line(p1=(-49e-9, 0, 0), p2=(49e-9, 0, 0), n=20).mpl()
```
2.16.1 Other

More details on various functionality can be found in the API Reference.

2.17 Running simulations inside Docker

In this tutorial, we show how to run simulations inside Docker. More precisely, we show how to choose a runner. Docker helps us 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, 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.

[1]:
```python
import oommfc as mc
import discretisedfield as df
import micromagneticmodel as mm
```

We define mesh in cuboid through corner points p1 and p2, and discretisation 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.

[2]:
```python
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:

[3]:
```python
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)))
```

(continues on next page)
Ms = 1.1e6

```python
def m_init(pos):
x, y, z = pos
    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, so our simulations are run inside Docker.

```python
[4]: runner = mc.oommf.DockerOOMMFRunner()
```

Finally we can minimise the energy and plot the magnetisation. When we create a `MinDriver` object, we pass `runner`.

**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.

```python
[5]: # NBVAL_SKIP
md = mc.MinDriver()
md.drive(system, runner=runner)

# md.drive(system)  # on Windows

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

Running OOMMF (DockerOOMMFRunner) [2020/07/01 21:28]... (3.5 s)
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.

2.18 Tutorial 15: Support, License, How to cite, and Acknowledgements

Interactive online tutorial:

2.18.1 Support

If you require any support or have any questions, you are welcome to raise an issue in our ubermag/help repository.

2.18.2 License

Licensed under the BSD 3-Clause “New” or “Revised” License. For details, please refer to the LICENSE file.

2.18.3 How to cite

If you use this package in your research, please cite it as:


2.18.4 Acknowledgements

Developed as a part of OpenDreamKit – Horizon 2020 European Research Infrastructure project (676541).
3.1 oommfc.CGEvolver

class oommfc.CGEvolver(**kwargs)
Conjugate-Gradient evolver.

Only attributes in _allowed_attributes can be defined. For details on possible values for individual attributes and their default values, please refer to Oxs_CGEvolver documentation (https://math.nist.gov/oommf/).

Examples

1. Defining evolver with a keyword argument.

```python
>>> import oommfc as oc
... >>> evolver = oc.CGEvolver(method='Polak-Ribiere')
```

2. Passing an argument which is not allowed.

```python
>>> import oommfc as oc
... >>> evolver = oc.CGEvolver(myarg=3)
Traceback (most recent call last):
  ... AttributeError: ...
```

3. Getting the list of allowed attributes.
>>> import oommfc as oc
...
>>> evolver = oc.CGEvolver()
>>> evolver._allowed_attributes
[...]

__class__
    alias of abc.ABCMeta

__delattr__
    Implement delattr(self, name).

__dir__()
    Default dir() implementation.

__eq__
    Return self==value.

__format__()
    Default object formatter.

__ge__
    Return self>=value.

__getattribute__
    Return getattr(self, name).

__gt__
    Return self>value.

__hash__
    Return hash(self).

__init__(**kwargs)
    It can be initialised with keyword arguments defined in _allowed_attributes, which is a list of strings.

    Raises AttributeError – If a keyword argument not in _allowed_attributes is passed.

__init_subclass__()
    This method is called when a class is subclassed.

    The default implementation does nothing. It may be overridden to extend subclasses.

__iter__()
    Iterator.

    It yields all defined attributes and their values.

Examples

1. Iterating through all defined attributes and their values.

>>> import micromagneticmodel as mm
...
>>> uniaxialanisotropy = mm.UniaxialAnisotropy(K=1e5, u=(0, 0, 1))
>>> for attr, value in uniaxialanisotropy:
...    print(f'attr = {value}({attr})')

(continues on next page)
K = 100000.0
u = (0, 0, 1)

__le__
  Return self<=value.
__lt__
  Return self<value.
__ne__
  Return self!=value.
__new__()
  Create and return a new object. See help(type) for accurate signature.
__reduce__()
  Helper for pickle.
__reduce_ex__()
  Helper for pickle.
__repr__()
  Representation string.
    Returns  Representation string.
    Return type  str

Examples

1. Getting representation string.

```python
>>> import micromagneticmodel as mm
...  
>>> exchange = mm.Exchange(A=1e-12)
>>> repr(exchange)
'Exchange(A=1e-12)'
>>> damping = mm.Damping(alpha=0.01)
>>> repr(damping)
'Damping(alpha=0.01)'
```

__setattr__
  Implement setattr(self, name, value).

__sizeof__()
  Size of object in memory, in bytes.
__str__
  Return str(self).

__subclasshook__()
  Abstract classes can override this to customize issubclass().
  This is invoked early on by abc.ABCMeta.__subclasscheck__(). It should return True, False or NotImple-
  mented. If it returns NotImplemented, the normal algorithm is used. Otherwise, it overrides the normal
  algorithm (and the outcome is cached).

__weakref__
  list of weak references to the object (if defined)
name
    Name.

    If the name was not provided during initialisation, the name of the object is the same as the name of the
    class in lowercase.

    Returns Name.
    Return type str

Examples

1. Getting names.

```python
>>> import micromagneticmodel as mm
... >>> ua = mm.UniaxialAnisotropy(K=5e6, u=(0, 0, 1))
>>> ua.name
'uniaxialanisotropy'
>>> damping = mm.Damping(alpha=0.01, name='my_damping')
>>> damping.name
'my_damping'
```

3.2 oommfc.EulerEvolver

class oommfc.EulerEvolver(**kwargs)
    Euler evolver.

    Only attributes in _allowed_attributes can be defined. For details on possible values for individual
    attributes and their default values, please refer to Oxs_EulerEvolver documentation (https://math.nist.gov/
    oommf/).

Examples

1. Defining evolver with a keyword argument.

```python
>>> import oommfc as oc
... >>> evolver = oc.EulerEvolver(min_timestep=0)
```

2. Passing an argument which is not allowed.

```python
>>> import oommfc as oc
... >>> evolver = oc.EulerEvolver(myarg=3)
Traceback (most recent call last):
  ... AttributeError: ...
```

3. Getting the list of allowed attributes.
>>> import oommfc as oc
...
>>> evolver = oc.EulerEvolver()
>>> evolver._allowed_attributes
[...]

__class__
    alias of abc.ABCMeta

__delattr__
    Implement delattr(self, name).

__dir__()
    Default dir() implementation.

__eq__
    Return self==value.

__format__()
    Default object formatter.

__ge__
    Return self>=value.

__getattribute__
    Return getattr(self, name).

__gt__
    Return self>value.

__hash__
    Return hash(self).

__init__(**kwargs)
    It can be initialised with keyword arguments defined in _allowed_attributes, which is a list of strings.

    Raises AttributeError – If a keyword argument not in _allowed_attributes is passed.

__init_subclass__()
    This method is called when a class is subclassed.

    The default implementation does nothing. It may be overridden to extend subclasses.

__iter__()
    Iterator.

    It yields all defined attributes and their values.

Examples

1. Iterating through all defined attributes and their values.

>>> import micromagneticmodel as mm
...
>>> uniaxialanisotropy = mm.UniaxialAnisotropy(K=1e5, u=(0, 0, 1))
>>> for attr, value in uniaxialanisotropy:
...    print(f'{attr} = {value}')
\[ K = 100000.0 \\
 u = (0, 0, 1) \]

___le__
Return self<=value.

___lt__
Return self<value.

___ne__
Return self!=value.

___new__()
Create and return a new object. See help(type) for accurate signature.

___reduce__()
Helper for pickle.

___reduce_ex__()
Helper for pickle.

___repr__()
Representation string.

Returns Representation string.

Return type str

Examples

1. Getting representation string.

```python
>>> import micromagneticmodel as mm
... >>> exchange = mm.Exchange(A=1e-12)
>>> repr(exchange)
'Exchange(A=1e-12)'
>>> damping = mm.Damping(alpha=0.01)
>>> repr(damping)
'Damping(alpha=0.01)'
```

___setattr__
Implement setattr(self, name, value).

___sizeof__()
Size of object in memory, in bytes.

___str__
Return str(self).

___subclasshook__()
Abstract classes can override this to customize issubclass().

This is invoked early on by abc.ABCMeta.__subclasscheck__(). It should return True, False or NotImple-
mented. If it returns NotImplemented, the normal algorithm is used. Otherwise, it overrides the normal
algorithm (and the outcome is cached).

___weakref__
list of weak references to the object (if defined)
name

Name.

If the name was not provided during initialisation, the name of the object is the same as the name of the
class in lowercase.

Returns Name.

Return type str

Examples

1. Getting names.

```python
>>> import micromagneticmodel as mm
... >>> ua = mm.UniaxialAnisotropy(K=5e6, u=(0, 0, 1))
>>> ua.name
'uniaxialanisotropy'
>>> damping = mm.Damping(alpha=0.01, name='my_damping')
>>> damping.name
'my_damping'
```

3.3 oommfc.RungeKuttaEvolver

class oommfc.RungeKuttaEvolver(**kwargs)

Runge-Kutta evolver.

Only attributes in _allowed_attributes can be defined. For details on possible values for individual
attributes and their default values, please refer to Oxs_RungeKuttaEvolver documentation (https://math.
nist.gov/oommf/).

Examples

1. Defining evolver with a keyword argument.

```python
>>> import oommfc as oc
... >>> evolver = oc.RungeKuttaEvolver(method='rk4')
```

2. Passing an argument which is not allowed.

```python
>>> import oommfc as oc
... >>> evolver = oc.RungeKuttaEvolver(myarg=3)
Traceback (most recent call last):
 ...
  raise AttributeError(f'missing allowed attribute: {k}')
AttributeError: ...
```

3. Getting the list of allowed attributes.
```python
>>> import oommfc as oc
...
>>> evolver = oc.RungeKuttaEvolver()
>>> evolver._allowed_attributes
[...]

__class__
   alias of abc.ABCMeta
__delattr__
   Implement delattr(self, name).
__dir__()
   Default dir() implementation.
__eq__
   Return self==value.
__format__()
   Default object formatter.
__ge__
   Return self>=value.
__getattribute__
   Return getattr(self, name).
__gt__
   Return self>value.
__hash__
   Return hash(self).
__init__(**kwargs)
   It can be initialised with keyword arguments defined in _allowed_attributes, which is a list of strings.

   Raises AttributeError – If a keyword argument not in _allowed_attributes is passed.
__init_subclass__()
   This method is called when a class is subclassed.

   The default implementation does nothing. It may be overridden to extend subclasses.
__iter__()
   Iterator.

   It yields all defined attributes and their values.

Examples

1. Iterating through all defined attributes and their values.
```
```
K = 100000.0
u = (0, 0, 1)

___le__
Return self<=value.

___lt__
Return self<value.

___ne__
Return self!=value.

___new__()  
Create and return a new object. See help(type) for accurate signature.

___reduce__()  
Helper for pickle.

___reduce_ex__()  
Helper for pickle.

___repr__()  
Representation string.

   Returns  Representation string.

   Return type  str

Examples

1. Getting representation string.

>>> import micromagneticmodel as mm
...
>>> exchange = mm.Exchange(A=1e-12)
>>> repr(exchange)
'Exchange(A=1e-12)'
>>> damping = mm.Damping(alpha=0.01)
>>> repr(damping)
'Damping(alpha=0.01)'

___setattr__
Implement setattr(self, name, value).

___sizeof__()
Size of object in memory, in bytes.

___str__
Return str(self).

___subclasshook__()
Abstract classes can override this to customize subclass().

   This is invoked early on by abc.ABCMeta.__subclasscheck__(). It should return True, False or NotImple-
   mented. If it returns NotImplemented, the normal algorithm is used. Otherwise, it overrides the normal
   algorithm (and the outcome is cached).

___weakref__
list of weak references to the object (if defined)

3.3. oommfc.RungeKuttaEvolver
oommfc

name
Name.

If the name was not provided during initialisation, the name of the object is the same as the name of the
class in lowercase.

Returns Name.

Return type str

Examples
1. Getting names.

```python
>>> import micromagneticmodel as mm
...
>>> ua = mm.UniaxialAnisotropy(K=5e6, u=(0, 0, 1))
>>> ua.name
'uniaxialanisotropy'
>>> damping = mm.Damping(alpha=0.01, name='my_damping')
>>> damping.name
'my_damping'
```

3.4 oommfc.SpinTEvolver

class oommfc.SpinTEvolver(**kwargs)
Zhang-Li evolver.

Only attributes in _allowedAttributes can be defined. For details on possible values for individual
attributes and their default values, please refer to Anv_SpinTEvolve documentation (https://www.zurich.ibm.com/st/nanomagnetism/spintevolve.html).

Examples
1. Defining evolver with a keyword argument.

```python
>>> import oommfc as oc
...
>>> evolver = oc.SpinTEvolver(method='rk4')
```

2. Passing an argument which is not allowed.

```python
>>> import oommfc as oc
...
>>> evolver = oc.SpinTEvolver(myarg=3)
Traceback (most recent call last):
  ...
AttributeError: ...
```

3. Getting the list of allowed attributes.
>>> import oommfc as oc
... 
>>> evolver = oc.SpinTEvolver()
>>> evolver._allowed_attributes
[...

__class__
    alias of abc.ABCMeta

__delattr__
    Implement delattr(self, name).

__dir__()
    Default dir() implementation.

__eq__
    Return self==value.

__format__()
    Default object formatter.

__ge__
    Return self>=value.

__getattr__
    Return getattr(self, name).

__gt__
    Return self>value.

__hash__
    Return hash(self).

__init__(*kwargs)
    It can be initialised with keyword arguments defined in _allowed_attributes, which is a list of strings.

    Raises AttributeError – If a keyword argument not in _allowed_attributes is passed.

__init_subclass__()
    This method is called when a class is subclassed.

    The default implementation does nothing. It may be overridden to extend subclasses.

__iter__()
    Iterator.

    It yields all defined attributes and their values.

Examples

1. Iterating through all defined attributes and their values.

    >>> import micromagneticmodel as mm
    ... 
    >>> uniaxialanisotropy = mm.UniaxialAnisotropy(\(K=1e5, u=(0, 0, 1)\))
    >>> for attr, value in uniaxialanisotropy:
    ...     print(f'\{attr\} = \{value\}')
K = 100000.0
u = (0, 0, 1)

__le__
    Return self<=value.

__lt__
    Return self<value.

__ne__
    Return self!=value.

__new__()
    Create and return a new object. See help(type) for accurate signature.

__reduce__()
    Helper for pickle.

__reduce_ex__()
    Helper for pickle.

__repr__()
    Representation string.

    Returns Representation string.

    Return type str

Examples

1. Getting representation string.

    >>> import micromagneticmodel as mm
    ...  
    >>> exchange = mm.Exchange(A=1e-12)
    >>> repr(exchange)
    'Exchange(A=1e-12)'
    >>> damping = mm.Damping(alpha=0.01)
    >>> repr(damping)
    'Damping(alpha=0.01)'

__setattr__
    Implement setattr(self, name, value).

__sizeof__()
    Size of object in memory, in bytes.

__str__
    Return str(self).

__subclasshook__()
    Abstract classes can override this to customize issubclass().

    This is invoked early on by abc.ABCMeta.__subclasscheck__(). It should return True, False or NotImple-
    mented. If it returns NotImplemented, the normal algorithm is used. Otherwise, it overrides the normal
    algorithm (and the outcome is cached).

__weakref__
    list of weak references to the object (if defined)
name

Name.

If the name was not provided during initialisation, the name of the object is the same as the name of the
class in lowercase.

**Returns** Name.

**Return type** str

**Examples**

1. Getting names.

```python
>>> import micromagneticmodel as mm
... >>> ua = mm.UniaxialAnisotropy(K=5e6, u=(0, 0, 1))
>>> ua.name
'uniaxialanisotropy'
>>> damping = mm.Damping(alpha=0.01, name='my_damping')
>>> damping.name
'my_damping'
```

### 3.5 oommfc.SpinXferEvolver

**class** oommfc.SpinXferEvolver(**kwargs)**

Slonczewski evolver.

Only attributes in _allowed_attributes can be defined. For details on possible values for individual
attributes and their default values, please refer to Oxs_SpinXferEvolve documentation (https://math.nist.gov/oommf/).

**Examples**

1. Defining evolver with a keyword argument.

```python
>>> import oommfc as oc
... >>> evolver = oc.SpinXferEvolver(method='rk4')
```

2. Passing an argument which is not allowed.

```python
>>> import oommfc as oc
... >>> evolver = oc.SpinXferEvolver(myarg=3)
Traceback (most recent call last):
  ... AttributeError: ...
```

3. Getting the list of allowed attributes.
>>> import oommfc as oc
...
>>> evolver = oc.SpinXferEvolver()
>>> evolver._allowed_attributes
[...]

__class__
alias of abc.ABCMeta

__delattr__
Implement delattr(self, name).

__dir__()
Default dir() implementation.

__eq__
Return self==value.

__format__()
Default object formatter.

__ge__
Return self>=value.

__getattribute__
Return getattr(self, name).

__gt__
Return self>value.

__hash__
Return hash(self).

__init__(**kwargs)
It can be initialised with keyword arguments defined in _allowed_attributes, which is a list of strings.

Raises AttributeError – If a keyword argument not in _allowed_attributes is passed.

__init_subclass__()
This method is called when a class is subclassed.

The default implementation does nothing. It may be overridden to extend subclasses.

__iter__()
Iterator.

It yields all defined attributes and their values.

Examples

1. Iterating through all defined attributes and their values.

```python
>>> import micromagneticmodel as mm
...
>>> uniaxialanisotropy = mm.UniaxialAnisotropy(K=1e5, u=(0, 0, 1))
>>> for attr, value in uniaxialanisotropy:
...     print(f'{attr} = {value}')
```
K = 100000.0
u = (0, 0, 1)

__le__
    Return self<=value.

__lt__
    Return self<value.

__ne__
    Return self!=value.

__new__(
    Create and return a new object. See help(type) for accurate signature.

__reduce__(
    Helper for pickle.

__reduce_ex__(
    Helper for pickle.

__repr__(
    Representation string.

    Returns Representation string.

    Return type str

Examples

1. Getting representation string.

>>> import micromagneticmodel as mm
...
>>> exchange = mm.Exchange(A=1e-12)
>>> repr(exchange)
'Exchange(A=1e-12)'
>>> damping = mm.Damping(alpha=0.01)
>>> repr(damping)
'Damping(alpha=0.01)'

__setattr__
    Implement setattr(self, name, value).

__sizeof__()
    Size of object in memory, in bytes.

__str__
    Return str(self).

__subclasshook__()
    Abstract classes can override this to customize issubclass().

    This is invoked early on by abc.ABCMeta.__subclasscheck__(). It should return True, False or NotImple-
    imented. If it returns NotImplemented, the normal algorithm is used. Otherwise, it overrides the normal
    algorithm (and the outcome is cached).

__weakref__
    list of weak references to the object (if defined)
name

Name.

If the name was not provided during initialisation, the name of the object is the same as the name of the
class in lowercase.

Returns Name.

Return type str

Examples

1. Getting names.

```python
>>> import micromagneticmodel as mm
...
>>> ua = mm.UniaxialAnisotropy(K=5e6, u=(0, 0, 1))
>>> ua.name
'uniaxialanisotropy'
>>> damping = mm.Damping(alpha=0.01, name='my_damping')
>>> damping.name
'my_damping'
```

### 3.6 oommfc.MinDriver

class oommfc.MinDriver(**kwargs)

Energy minimisation driver.

Only attributes in _allowed_attributes can be defined. For details on possible values for individual
attributes and their default values, please refer to Oxs_MinDriver documentation (https://math.nist.gov/oommf/).

Examples

1. Defining driver with a keyword argument.

```python
>>> import oommfc as oc
...
>>> md = oc.MinDriver(stopping_mxHxm=0.01)
```

2. Passing an argument which is not allowed.

```python
>>> import oommfc as oc
...
>>> md = oc.MinDriver(myarg=1)
Traceback (most recent call last):
...
AttributeError: ...
```

3. Getting the list of allowed attributes.
>>> import oommfc as oc
...
>>> md = oc.MinDriver()
>>> md._allowed_attributes
...

__class__
    alias of abc.ABCMeta

_delattr_
    Implement delattr(self, name).

dir()
    Default dir() implementation.

eq
    Return self==value.

format()
    Default object formatter.

ge
    Return self>=value.

getattribute
    Return getattr(self, name).

gt
    Return self>value.

_hash_
    Return hash(self).

init(**kwargs)
    It can be initialised with keyword arguments defined in _allowed_attributes, which is a list of strings.

    Raises AttributeError – If a keyword argument not in _allowed_attributes is passed.

_initializer_()
    This method is called when a class is subclassed.

    The default implementation does nothing. It may be overridden to extend subclasses.

_iter_
    Iterator.

It yields all defined attributes and their values.

Examples

1. Iterating through all defined attributes and their values.

>>> import micromagneticmodel as mm
...
>>> uniaxialanisotropy = mm.UniaxialAnisotropy(K=1e5, u=(0, 0, 1))

for attr, value in uniaxialanisotropy:
    ...
    print(f'{attr} = {value}')

(continues on next page)
\[ K = 100000.0 \\
\text{u} = (0, 0, 1) \]

\[ \text{\_le\_} \]
\text{return self<=value.}

\[ \text{\_lt\_} \]
\text{return self<value.}

\[ \text{\_ne\_} \]
\text{return self!=value.}

\[ \text{\_new\_}() \]
\text{create and return a new object. see help(type) for accurate signature.}

\[ \text{\_reduce\_}() \]
\text{helper for pickle.}

\[ \text{\_reduce\_ex\_}() \]
\text{helper for pickle.}

\[ \text{\_repr\_}() \]
\text{representation string.}

\text{\textbf{Returns} representation string.}

\text{\textbf{Return type} str}

\section*{Examples}

1. Getting representation string.

\begin{verbatim}
>>> import micromagneticmodel as mm
...
>>> exchange = mm.Exchange(A=1e-12)
>>> repr(exchange)
'Exchange(A=1e-12)'
>>> damping = mm.Damping(alpha=0.01)
>>> repr(damping)
'Damping(alpha=0.01)'
\end{verbatim}

\[ \text{\_setattr\_}() \]
\text{implement setattr(self, name, value).}

\[ \text{\_sizeof\_}() \]
\text{size of object in memory, in bytes.}

\[ \text{\_str\_}() \]
\text{return str(self).}

\[ \text{\_subclasshook\_}() \]
\text{abstract classes can override this to customize issubclass().}

\text{this is invoked early on by abc.ABCMeta.__subclasscheck__(). it should return true, false or notimplemented. if it returns notimplemented, the normal algorithm is used. otherwise, it overrides the normal algorithm (and the outcome is cached).}

\[ \text{\_weakref\_}() \]
\text{list of weak references to the object (if defined).}
drive (system, append=True, compute=None, runner=None, **kwargs)

Drives the system in phase space.

Takes micromagneticmodel.System and drives it in the phase space. If append=True and the system director already exists, drive will be appended to that directory. Otherwise, an exception will be raised. To save a specific value during an OOMMF run Schedule...

line can be passed using compute. To specify the way OOMMF is run, an oommfc.oommf.OOMMFRunner can be passed using runner.

This method accepts any other arguments that could be required by the specific driver.

**Parameters**

- **system** *(micromagneticmodel.System)* – System object to be driven.
- **append** *(bool)* – If True and the system directory already exists, drive or compute directories will be appended.
- **compute** *(str)* – Schedule...
- **runner** *(oommfc.oommf.OOMMFRunner)* – OOMMF Runner which is going to be used for running OOMMF. If None, OOMMF runner will be found automatically. Defaults to None.

**Raises** FileExistsError – If system directory already exists and append=False.

**Examples**

1. Drive system using minimisation driver (MinDriver).

```python
>>> import micromagneticmodel as mm
>>> import discretisedfield as df
>>> import oommfc as oc
...
>>> system = mm.System(name='my_cool_system')
>>> system.energy = mm.Exchange(A=1e-12) + mm.Zeeman(H=(0, 0, 1e6))
>>> mesh = df.Mesh(p1=(0, 0, 0), p2=(1e-9, 1e-9, 10e-9), n=(1, 1, 10))
>>> system.m = df.Field(mesh, dim=3, value=(1, 1, 1), norm=1e6)
...
>>> md = oc.MinDriver()
>>> md.drive(system)
Running OOMMF...
```

2. Drive system using time driver (TimeDriver).

```python
>>> system.energy.zeeman.H = (0, 1e6, 0)
...
>>> td = oc.TimeDriver()
>>> td.drive(system, t=0.1e-9, n=10)
Running OOMMF...
```

**name**

Name.

If the name was not provided during initialisation, the name of the object is the same as the name of the class in lowercase.

**Returns** Name.
**Examples**

1. Getting names.

```python
>>> import micromagneticmodel as mm
...  
>>> ua = mm.UniaxialAnisotropy(K=5e6, u=(0, 0, 1))
>>> ua.name
'uniaxialanisotropy'
>>> damping = mm.Damping(alpha=0.01, name='my_damping')
>>> damping.name
'my_damping'
```

### 3.7 oommfc.TimeDriver

**class** oommfc.TimeDriver(**kwargs)

Time driver.

Only attributes in `_allowed_attributes` can be defined. For details on possible values for individual attributes and their default values, please refer to Oxs_TimeDriver documentation (https://math.nist.gov/oommf/).

**Examples**

1. Defining driver with a keyword argument.

```python
>>> import oommfc as oc
...  
>>> td = oc.TimeDriver(total_iteration_limit=5)
```

2. Passing an argument which is not allowed.

```python
>>> import oommfc as oc
...  
>>> td = oc.TimeDriver(myarg=1)
Traceback (most recent call last):
  ...  
AttributeError: ...
```

3. Getting the list of allowed attributes.

```python
>>> import oommfc as oc
...  
>>> td = oc.TimeDriver()
>>> td._allowed_attributes
[...]

__class__
    alias of abc.ABCMeta
__delattr__  
Implement delattr(self, name).

__dir__()  
Default dir() implementation.

__eq__  
Return self==value.

__format__()  
Default object formatter.

__ge__  
Return self>=value.

__getattribute__  
Return getattr(self, name).

__gt__  
Return self>value.

__hash__  
Return hash(self).

__init__(**kwargs)  
It can be initialised with keyword arguments defined in _allowed_attributes, which is a list of strings.

Raises AttributeError – If a keyword argument not in _allowed_attributes is passed.

__init_subclass__()  
This method is called when a class is subclassed.

The default implementation does nothing. It may be overridden to extend subclasses.

__iter__()  
Iterator.

It yields all defined attributes and their values.

Examples

1. Iterating through all defined attributes and their values.

```python
>>> import micromagneticmodel as mm
... >>> uniaxialanisotropy = mm.UniaxialAnisotropy(K=1e5, u=(0, 0, 1))
>>> for attr, value in uniaxialanisotropy:
...     print(f'{attr} = {value}')
K = 100000.0
u = (0, 0, 1)
```

__le__  
Return self<=value.

__lt__  
Return self<value.

__ne__  
Return self!=value.
__new__()
Create and return a new object. See help(type) for accurate signature.

__reduce__()
Helper for pickle.

__reduce_ex__()
Helper for pickle.

__repr__()
Representation string.

Returns Representation string.
Return type str

Examples

1. Getting representation string.

```python
>>> import micromagneticmodel as mm
...
>>> exchange = mm.Exchange(A=1e-12)
>>> repr(exchange)
'Exchange(A=1e-12)'
>>> damping = mm.Damping(alpha=0.01)
>>> repr(damping)
'Damping(alpha=0.01)'
```

__setattr__
Implement setattr(self, name, value).

__sizeof__()
Size of object in memory, in bytes.

__str__
Return str(self).

__subclasshook__
Abstract classes can override this to customize issubclass().

This is invoked early on by abc.ABCMeta.__subclasscheck__(). It should return True, False or Not Implemented. If it returns Not Implemented, the normal algorithm is used. Otherwise, it overrides the normal algorithm (and the outcome is cached).

__weakref__
list of weak references to the object (if defined)

drive (system, append=True, compute=None, runner=None, **kwargs)
Drives the system in phase space.

Takes micromagneticmodel.System and drives it in the phase space. If append=True and the system director already exists, drive will be appended to that directory. Otherwise, an exception will be raised. To save a specific value during an OOMMF run Schedule... line can be passed using compute. To specify the way OOMMF is run, an oommfc.oommf.OOMMFRunner can be passed using runner.

This method accepts any other arguments that could be required by the specific driver.

Parameters
• **system** (*micromagneticmodel.System*) – System object to be driven.

• **append** (*bool*) – If True and the system directory already exists, drive or compute directories will be appended.

• **compute** (*str*) – Schedule... MIF line which can be added to the OOMMF file to save additional data. Defaults to None.

• **runner** (*oommfc.oommf.OOMMFRunner*) – OOMMF Runner which is going to be used for running OOMMF. If None, OOMMF runner will be found automatically. Defaults to None.

    **Raises** *FileExistsError* – If system directory already exists and append=False.

**Examples**

1. Drive system using minimisation driver (*MinDriver*).

```python
>>> import micromagneticmodel as mm
>>> import discretisedfield as df
>>> import oommfc as oc
...
>>> system = mm.System(name='my_cool_system')
>>> system.energy = mm.Exchange(A=1e-12) + mm.Zeeman(H=(0, 0, 1e6))
>>> mesh = df.Mesh(p1=(0, 0, 0), p2=(1e-9, 1e-9, 10e-9), n=(1, 1, 10))
>>> system.m = df.Field(mesh, dim=3, value=(1, 1, 1), norm=1e6)
...
>>> md = oc.MinDriver()
>>> md.drive(system)
Running OOMMF...
```

2. Drive system using time driver (*TimeDriver*).

```python
>>> system.energy.zeeman.H = (0, 1e6, 0)
...
>>> td = oc.TimeDriver()
>>> td.drive(system, t=0.1e-9, n=10)
Running OOMMF...
```

**name**

Name.

If the name was not provided during initialisation, the name of the object is the same as the name of the class in lowercase.

**Returns** *Name*.

**Return type** *str*

**Examples**

1. Getting names.
3.8 oommfc.compute

oommfcompute(func, system)

Computes a particular value of an energy term or energy container (energy, density, or effective_field).

Parameters

- func (callable) – A property of an energy term or an energy container.
- system (micromagneticmodel.System) – Micromagnetic system for which the property is calculated.

Returns Resulting value.

Return type numbers.Real, discretisedfield.Field

Examples

1. Computing values of energy terms.

>>> import micromagneticmodel as mm
>>> import oommfc as oc
... >>> system = mm.examples.macrospin()
>>> oc.compute(system.energy.zeeman.energy, system)
Running OOMMF...
-8.8...e-22
>>> oc.compute(system.energy.effective_field, system)
Running OOMMF...
Field(...)  
>>> oc.compute(system.energy.density, system)
Running OOMMF...
Field(...)

3.9 oommfc.delete

oommfdelete(system, silent=False)

Deletes micromagnetic system files.

This is a convenience function for deleting all of the data associated with a system object. More precisely, the directory with name the same as system.name is deleted. If silent=True is passed, no error is raised if the directory does not exist.
Parameters

- **system** (*micromagneticmodel.System*) – System whose files are deleted.
- **silent** (*bool, optional*) – If True, no error is raised if the directory does not exist.

Raises **FileNotFoundError** – If the directory with system.name does not exist and silent=False.

Examples

1. Delete system files.

```python
>>> import os
>>> import oommfc as mc
>>> import micromagneticmodel as mm
...
>>> system = mm.examples.macrospin()
>>> mc.delete(system)
>>> td = mc.TimeDriver()
>>> td.drive(system, t=1e-12, n=5, save=True)
Running OOMMF...
>>> os.path.exists(system.name)
True
>>> mc.delete(system)  # deletes directory
>>> os.path.exists(system.name)
False
```

3.10 oommfc.oommf

class oommfc.oommf.OOMMFRunner

Abstract class for running OOMMF.

call (*argstr, need_stderr=False*)

Calls OOMMF by passing *argstr* to OOMMF.

Parameters

- **argstr** (*str*) – Argument string passed to OOMMF.
- **need_stderr** (*bool*) – If need_stderr=True, standard error is captured. Defaults to False.

Raises **RuntimeError** – If an error occurred.

Returns When the OOMMF run was successful, 0 is returned.

Return type int

Examples

1. Getting OOMMF runner automatically and calling it.
```python
>>> import oommfc as oc
... >>> runner = oc.oommf.get_oommf_runner()
>>> runner.call(argstr='+version')
Running OOMMF...
CompletedProcess(...)
```

**errors()**

Returns the content of boxsii.errors OOMMF file.

- **Returns**: boxsii.errors OOMMF file.
- **Return type**: str

**platform()**

Returns platform seen by OOMMF.

- **Returns**: Platform.
- **Return type**: str

**Examples**

1. Getting platform.

```python
>>> import oommfc as oc
... >>> runner = oc.oommf.get_oommf_runner()
>>> runner.platform()
Running OOMMF...
'...'
```

**version()**

Returns the OOMMF version.

- **Returns**: OOMMF version.
- **Return type**: str

**Examples**


```python
>>> import oommfc as oc
... >>> runner = oc.oommf.get_oommf_runner()
>>> runner.version()
Running OOMMF...
'...'
```

class oommfc.oommf.TclOOMMFRunner(oommf_tcl)

OOMMF runner using path to oommf.tcl.

- **Parameters**: oommf_tcl (str) – Path to “oommf.tcl” file.
- **call (argstr, need_stderr=False)**
  
  Calls OOMMF by passing argstr to OOMMF.
Parameters

- **argstr**(str) – Argument string passed to OOMMF.
- **need_stderr**(bool) – If need_stderr=True, standard error is captured. Defaults to False.

Raises: RuntimeError – If an error occurred.

Returns: When the OOMMF run was successful, 0 is returned.

Return type: int

Examples

1. Getting OOMMF runner automatically and calling it.

```python
>>> import oommfc as oc
... runnner = oc.oommf.get_oommf_runner()
>>> runner.call(argstr='+version')
Running OOMMF...
[...]
```

Returns the content of boxsii.errors OOMMF file.

- **errors**()
  Returns boxsii.errors OOMMF file.

- Return type: str

Returns platform seen by OOMMF.

- **platform**()
  Returns Platform.

- Return type: str

Examples

1. Getting platform.

```python
>>> import oommfc as oc
... runner = oc.oommf.get_oommf_runner()
>>> runner.platform()
Running OOMMF...
'...'
```

Returns the OOMMF version.

- **version**()
  Returns OOMMF version.

- Return type: str
Examples


```python
>>> import oommfc as oc
... >>> runner = oc.oommf.get_oommf_runner()
>>> runner.version()
Running OOMMF...
'...',
```

class oommfc.oommf.ExeOOMMFRunner (oommf_exe='oommf')

OOMMF runner using OOMMF executable, which can be found on $PATH.

Parameters

oommf_exe (str) – Name of the OOMMF executable. Defaults to oommf.

call (argstr, need_stderr=False)

Calls OOMMF by passing argstr to OOMMF.

Parameters

• **argstr** (str) – Argument string passed to OOMMF.
  • **need_stderr** (bool) – If need_stderr=True, standard error is captured. Defaults to False.

Raises RuntimeError – If an error occured.

Returns When the OOMMF run was successful, 0 is returned.

Return type int

Examples

1. Getting OOMMF runner automatically and calling it.

```python
>>> import oommfc as oc
... >>> runner = oc.oommf.get_oommf_runner()
>>> runner.call(argstr='+version')
Running OOMMF...
CompletedProcess(...)
```

errors ()

Returns the content of boxsii.errors OOMMF file.

Returns boxsii.errors OOMMF file.

Return type str

platform ()

Returns platform seen by OOMMF.

Returns Platform.

Return type str
Examples

1. Getting platform.

```python
>>> import oommfc as oc
... >>> runner = oc.oommf.get_oommf_runner()
>>> runner.platform()
Running OOMMF...
'...
```

`version()`
Returns the OOMMF version.

- **Returns**: OOMMF version.
- **Return type**: str

Examples


```python
>>> import oommfc as oc
... >>> runner = oc.oommf.get_oommf_runner()
>>> runner.version()
Running OOMMF...
'...
```

class `oommf.oommf.DockerOOMMFRunner` *(docker_exe='docker', image='ubermag/oommf')*
OOMMF runner using Docker.

**Parameters**

- **docker_exe**(str) – Docker executable. Defaults to docker.
- **image**(str) – Docker image on DockerHub. Defaults to ubermag/oommf.

**call**(argstr, need_stderr=False)
Calls OOMMF by passing argstr to OOMMF.

- **Parameters**
  - **argstr**(str) – Argument string passed to OOMMF.
  - **need_stderr**(bool) – If need_stderr=True, standard error is captured. Defaults to False.

- **Raises**: RuntimeError – If an error occurred.
- **Returns**: When the OOMMF run was successful, 0 is returned.
- **Return type**: int

Examples

1. Getting OOMMF runner automatically and calling it.
```python
>>> import oommfc as oc
...  
>>> runner = oc.oommf.get_oommf_runner()
>>> runner.call(argstr='+version')
Running OOMMF...
CompletedProcess(...)
```

**errors()**

Returns the content of boxsii.errors OOMMF file.

Returns `boxsii.errors` OOMMF file.

Return type `str`

**platform()**

Returns platform seen by OOMMF.

Returns `Platform`.

Return type `str`

**Examples**

1. Getting platform.

```python
>>> import oommfc as oc
...  
>>> runner = oc.oommf.get_oommf_runner()
>>> runner.platform()
Running OOMMF...
'...'  
```

**version()**

Returns the OOMMF version.

Returns OOMMF version.

Return type `str`

**Examples**


```python
>>> import oommfc as oc
...  
>>> runner = oc.oommf.get_oommf_runner()
>>> runner.version()
Running OOMMF...
'...'  
```
• **use_cache** *(bool)* – The first call to this function will determine the best way to run OOMMF and cache it. Normally, subsequent calls will return the `OOMMFRunner` object from the cache. Setting this parameter to `False` will cause it to check for available methods again. Defaults to `True`.

• **envvar** *(str)* – Name of the environment variable containing the path to `oommf.tcl`. Defaults to `'OOMMFTCL'`.

• **oommf_exe** *(str)* – The name or path of the executable `oommf` command. Defaults to `'oommf'`.

• **docker_exe** *(str)* – The name or path of the docker command. Defaults to `'docker'`.

Returns An OOMMF runner.

Return type `oommfc.oommf.OOMMFRunner`

Raises `EnvironmentError` – If no OOMMF can be found on host.

**Examples**

1. Getting OOMMF Runner.

```python
>>> import oommfc as oc
...
>>> runner = oc.oommf.get_oommf_runner()
>>> isinstance(runner, oc.oommf.OOMMFRunner)
True
```

**oommfc.oommf.status()**

Run a macrospin example for 1 ps through `oommfc` and print the OOMMF status.

Returns If 0, the OOMMF is found and running. Otherwise, 1 is returned.

Return type int

**Examples**

1. Checking the OOMMF status.

```python
>>> import oommfc as oc
...
>>> oc.oommf.status()
Running OOMMF...
OOMMF found and running.
0
```

**oommfc.oommf.overhead()**

Run a macrospin example for 1 ps through `oommfc` and directly and return the difference in run times.

Returns The time difference (overhead) between running OOMMF though `oommfc` and directly.

Return type float
Examples

1. Getting the overhead time.

```python
>>> import oommfc as oc
...
>>> isinstance(oc.oommf.overhead(), float)
Running OOMMF...
True
```
Symbols

__class__ (oommf.CGEvolver attribute), 86
__class__ (oommf.EulerEvolver attribute), 89
__class__ (oommf.MinDriver attribute), 101
__class__ (oommf.RungeKuttaEvolver attribute), 92
__class__ (oommf.SpinTEvolver attribute), 95
__class__ (oommf.SpinXferEvolver attribute), 98
__class__ (oommf.TimeDriver attribute), 105
__delattr__ (oommf.CGEvolver attribute), 86
__delattr__ (oommf.EulerEvolver attribute), 89
__delattr__ (oommf.MinDriver attribute), 101
__delattr__ (oommf.RungeKuttaEvolver attribute), 92
__delattr__ (oommf.SpinTEvolver attribute), 95
__delattr__ (oommf.SpinXferEvolver attribute), 98
__delattr__ (oommf.TimeDriver attribute), 105
__dir__ (oommf.CGEvolver method), 86
__dir__ (oommf.EulerEvolver method), 89
__dir__ (oommf.MinDriver method), 101
__dir__ (oommf.RungeKuttaEvolver method), 92
__dir__ (oommf.SpinTEvolver method), 95
__dir__ (oommf.SpinXferEvolver method), 98
__dir__ (oommf.TimeDriver method), 105
__eq__ (oommf.CGEvolver attribute), 86
__eq__ (oommf.EulerEvolver attribute), 89
__eq__ (oommf.MinDriver attribute), 101
__eq__ (oommf.RungeKuttaEvolver attribute), 92
__eq__ (oommf.SpinTEvolver attribute), 95
__eq__ (oommf.SpinXferEvolver attribute), 98
__eq__ (oommf.TimeDriver attribute), 105
__format__ (oommf.CGEvolver method), 86
__format__ (oommf.EulerEvolver method), 89
__format__ (oommf.MinDriver method), 101
__format__ (oommf.RungeKuttaEvolver method), 92
__format__ (oommf.SpinTEvolver method), 95
__format__ (oommf.SpinXferEvolver method), 98
__format__ (oommf.TimeDriver method), 105
__getattribute__ (oommf.CGEvolver attribute), 86
__getattribute__ (oommf.EulerEvolver attribute), 89
__getattribute__ (oommf.MinDriver attribute), 101
__getattribute__ (oommf.RungeKuttaEvolver attribute), 92
__getattribute__ (oommf.SpinTEvolver attribute), 95
__getattribute__ (oommf.SpinXferEvolver attribute), 98
__getattribute__ (oommf.TimeDriver attribute), 105
__hash__ (oommf.CGEvolver attribute), 86
__hash__ (oommf.EulerEvolver attribute), 89
__hash__ (oommf.MinDriver attribute), 101
__hash__ (oommf.RungeKuttaEvolver attribute), 92
__hash__ (oommf.SpinTEvolver attribute), 95
__hash__ (oommf.SpinXferEvolver attribute), 98
__hash__ (oommf.TimeDriver attribute), 105
__gt__ (oommf.CGEvolver attribute), 86
__gt__ (oommf.EulerEvolver attribute), 89
__gt__ (oommf.MinDriver attribute), 101
__gt__ (oommf.RungeKuttaEvolver attribute), 92
__gt__ (oommf.SpinTEvolver attribute), 95
__gt__ (oommf.SpinXferEvolver attribute), 98
__gt__ (oommf.TimeDriver attribute), 105
__init__ (oommf.CGEvolver attribute), 86
__init__ (oommf.EulerEvolver attribute), 89
__init__ (oommf.MinDriver attribute), 101
__init__ (oommf.RungeKuttaEvolver attribute), 92
__init__ (oommf.SpinTEvolver attribute), 95
__init__ (oommf.SpinXferEvolver attribute), 98
__init__ (oommf.TimeDriver attribute), 105

Index
| __subclasshook__() | (oommfc.RungeKuttaEvolver method), 93 |
| __subclasshook__() | (oommfc.SpinTEvolver method), 96 |
| __subclasshook__() | (oommfc.SpinXferEvolver method), 99 |
| __subclasshook__() | (oommfc.TimeDriver method), 106 |
| __weakref__() | (oommfc.CGEvolver attribute), 87 |
| __weakref__() | (oommfc.EulerEvolver attribute), 90 |
| __weakref__() | (oommfc.MinDriver attribute), 103 |
| __weakref__() | (oommfc.RungeKuttaEvolver attribute), 93 |
| __weakref__() | (oommfc.SpinTEvolver attribute), 96 |
| __weakref__() | (oommfc.SpinXferEvolver attribute), 99 |
| __weakref__() | (oommfc.TimeDriver attribute), 107 |
| call() | (oommfc.oommf.DockerOOMMFRunner method), 113 |
| call() | (oommfc.oommf.ExeOOMMFRunner method), 112 |
| call() | (oommfc.oommf.OOMMFRunner method), 109 |
| call() | (oommfc.oommf.TclOOMMFRunner method), 110 |
| CGEvolver | (class in oommfc), 85 |
| compute() | (in module oommfc), 108 |
| delete() | (in module oommfc), 108 |
| DockerOOMMFRunner | (class in oommfc.oommf), 113 |
| drive() | (oommfc.MinDriver method), 102 |
| drive() | (oommfc.TimeDriver method), 106 |
| errors() | (oommfc.oommf.DockerOOMMFRunner method), 114 |
| errors() | (oommfc.oommf.ExeOOMMFRunner method), 112 |
| errors() | (oommfc.oommf.OOMMFRunner method), 110 |
| errors() | (oommfc.oommf.TclOOMMFRunner method), 111 |
| EulerEvolver | (class in oommfc), 88 |
| ExeOOMMFRunner | (class in oommfc.oommf), 112 |
| get_oommf_runner() | (in module oommfc.oommf), 114 |
| MinDriver | (class in oommfc), 100 |