JOOMMF
Interactive Micromagnetic Simulations in Jupyter

Marijan Beg, Ryan A. Pepper, Leoni Breth*, Ondrej Hovorka and Hans Fangohr

Faculty of Engineering and the Environment, University of Southampton, Southampton, SO17 1BJ, United Kingdom

*email: l.breth@soton.ac.uk

Introducing JOOMMF

- provides a user-friendly interface to the finite difference code **OOMMF** (http://math.nist.gov/oommf/)
- enables the use of **Jupyter notebooks** (jupyter.org), which can be run in any webbrowser (illustrated in Fig. 1)
- commands allow to run a full simulation workflow within a single notebook instead of using different tools in each step of the workflow (see Fig. 2)
- interaction of the frontend layer written in **Python 3** with OOMMF through .mif-configuration files[1]
- extendable to other code packages in the future
- freely available on github (**https://github.com/joommf**)
- website with documentation: **joommf.github.io**

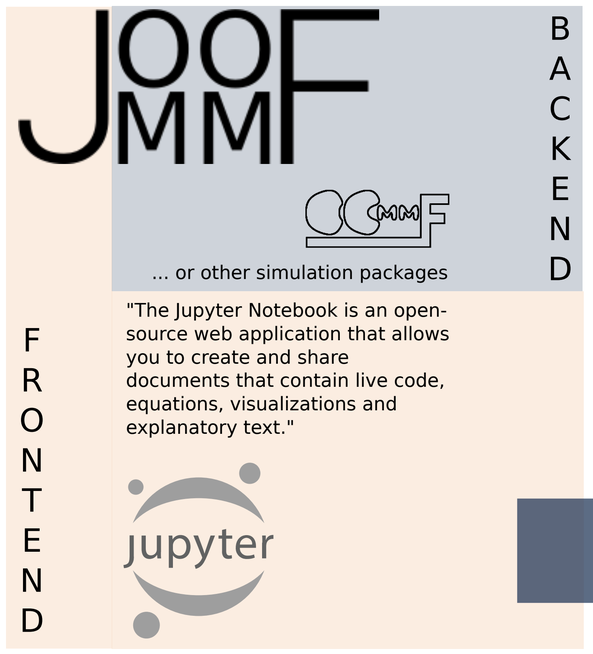


Fig. 1: **JOOMMF** = **J**upyter + **OOMMF**

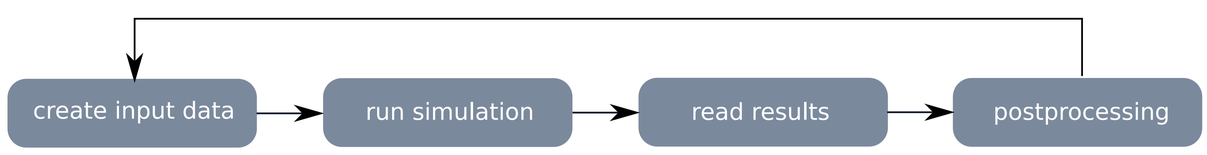
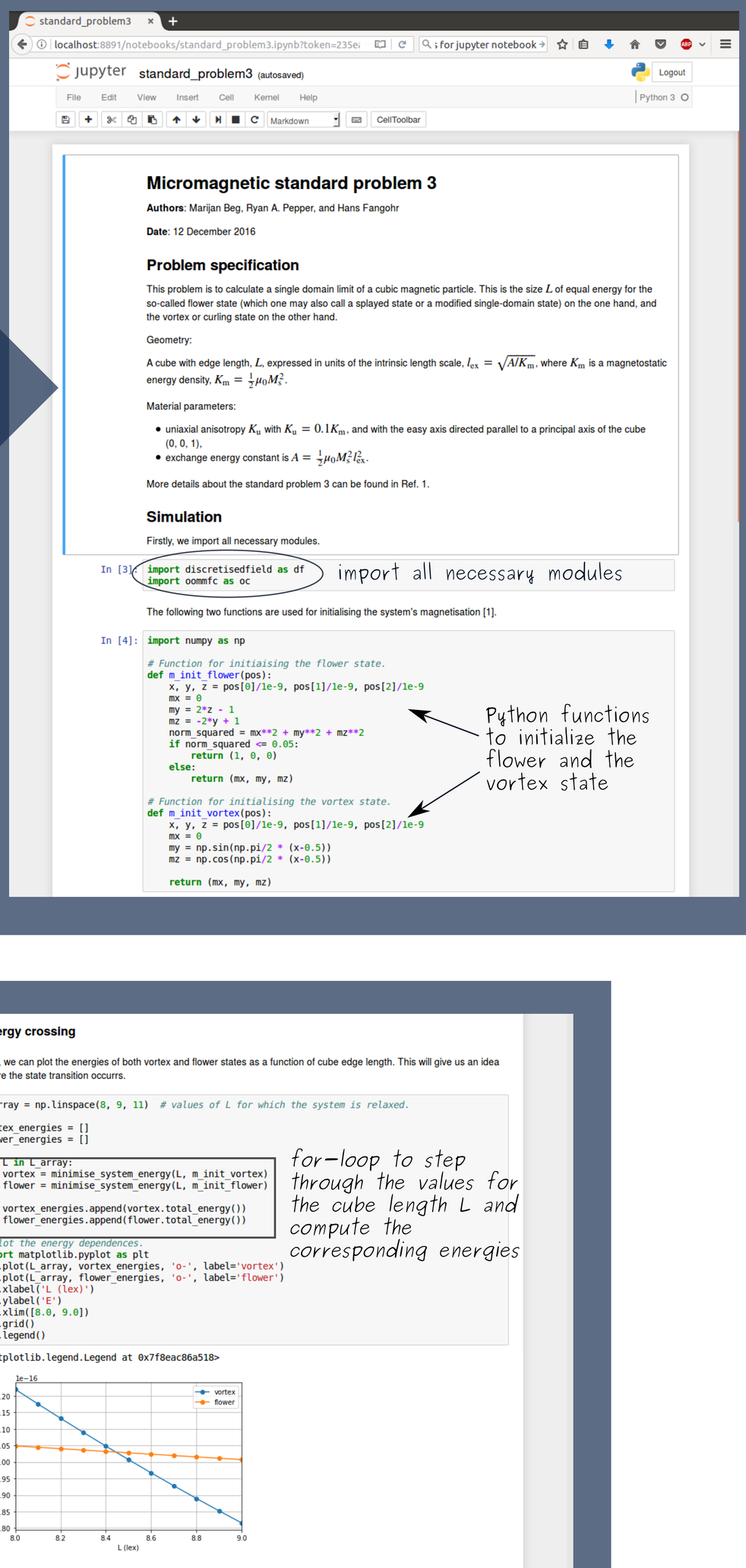


Fig. 2: Simulation workflow

Single domain limit of a cubic magnetic particle[2]



standard_problem3

localhost:8891/notebooks/standard_problem3.ipynb?token=235e

jupyter standard_problem3 (autosaved)

Logout

File Edit View Insert Cell Kernel Help Python 3

Markdown CellToolbar

Micromagnetic standard problem 3

Authors: Marijan Beg, Ryan A. Pepper, and Hans Fangohr

Date: 12 December 2016

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 magnetostatic 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.

Simulation

Firstly, we import all necessary modules.

In [3]:
```
import discretisedfield as df
import oommfc as oc
```
import all necessary modules

The following two functions are used for initialising the system's magnetisation [1].

In [4]:
```
import numpy as np

# Function for initiaising 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)
```
Python functions to initialize the flower and the vortex state

In [3]:
```
def minimise_system_energy(L, m_init):
    N = 16  # discretisation in one dimension
    cubesize = 100e-9  # cube edge length (m)
    cellsize = cubesize/N  # discretisation in all three dimensions.
    lex = cubesize/L  # exchange length.

    Km = 1e6  # magnetostatic energy density (J/m**3)
    Ms = np.sqrt(2*Km/oc.mu0)  # magnetisation saturation (A/m)
    A = 0.5 * oc.mu0 * Ms**2 * lex**2  # exchange energy constant
    K = 0.1*Km  # Uniaxial anisotropy constant
    u = (0, 0, 1)  # Uniaxial anisotropy easy-axis

    p1 = (0, 0, 0)  # Minimum sample coordinate.
    p2 = (cubesize, cubesize, cubesize)  # Maximum sample coordinate.
    cell = (cellsize, cellsize, cellsize)  # Discretisation.
    mesh = oc.Mesh(p1=(0, 0, 0), p2=(cubesize, cubesize, cubesize),
                   cell=(cellsize, cellsize, cellsize))  # Create a mesh object.

    system = oc.System(name="stdprob3")
    system.hamiltonian = oc.Exchange(A) + oc.UniaxialAnisotropy(K, u) + oc.Demag()
    system.m = df.Field(mesh, value=m_init, norm=Ms)

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

    return system
```
function, that returns the relaxed system object, argument L is the cube length in units of lex

the "system" object contains a set of properties, such as "hamiltonian" and "m"

here, the OOMMF configuration file (*.mif) is created for the energy minimization

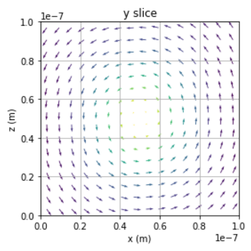
Relaxed magnetisation states

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

Vortex state:

In [4]:
```
%matplotlib inline
system = minimise_system_energy(8, m_init_vortex)
fig = system.m.plot_slice('y', 50e-9, xsize=4)
```
```
2017/3/16 14:54: Calling OOMMF (stdprob3/stdprob3.mif) ... [2.0s]
```

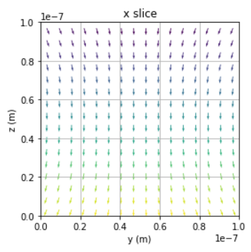
energy minimization is executed starting from the vortex state



Flower state:

In [5]:
```
system = minimise_system_energy(8, m_init_flower)
fig = system.m.plot_slice('x', 50e-9, xsize=4)
```
```
2017/3/16 14:54: Calling OOMMF (stdprob3/stdprob3.mif) ... [0.8s]
```

energy minimization is executed starting from the flower state



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 occurrs.

In [6]:
```
L_array = np.linspace(8, 9, 11)  # values of L for which the system is relaxed.

vortex_energies = []
flower_energies = []

for L in L_array:
    vortex = minimise_system_energy(L, m_init_vortex)
    flower = minimise_system_energy(L, m_init_flower)

    vortex_energies.append(vortex.total_energy())
    flower_energies.append(flower.total_energy())

# Plot the energy dependences.
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 (lex)')
plt.ylabel('E')
plt.xlim([8.0, 9.0])
plt.grid()
plt.legend()
```
for-loop to step through the values for the cube length L and compute the corresponding energies

Out[6]: <matplotlib.legend.Legend at 0x7f8eac86a518>

We now know that the energy crossing occurrs between $8l_{ex}$ and $9l_{ex}$, so a bisection algorithm can be used to find the exact crossing.

In [7]:
```
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.total_energy() - flower.total_energy()

cross_section = bisect(energy_difference, 8, 9, xtol=0.1)

print("The transition between vortex and flower states occurs at {}*lex".format(cross_section))
```
```
The transition between vortex and flower states occurs at 8.4375*lex
```
Python's "scipy"-module provides a wide range of useful algorithms for further data evaluation

References

[1] µMAG Site Directory http://www.ctcms.nist.gov/~rdm/mumag.org.html

[1] Beg. M., Pepper R., Fangohr H., AIP Advances 7, 056025 (2017)

[2] mumag Standard Problem #3: http://www.ctcms.nist.gov/~rdm/mumag.org.html

UNIVERSITY OF Southampton ICSS NGCM EPSRC Engineering and Physical Sciences Research Council OPEN DREAMKIT