magpy Documentation

Release 1.0

Oliver Laslett

Dec 28, 2017

Contents

CHAPTER 1

Installation

The easiest way to install magpy is using the conda repositories. The only requirement is that you have anaconda or miniconda installed on your system. Alternatively, you can build the C++ and python code from source. You might want to do this if you only want the C++ library or if you would like to access the Intel accelerated code.

1.1 Conda

Note packages only exist for Linux and Mac OSX.

1. Go to https://conda.io/miniconda.html and download the Miniconda package manager
2. Create a new conda environment and install magpy into it:

```
$ conda create -n <env_name> python=3
$ source activate <env_name>
$ conda install -c owlas magpy
```

3. Launch python and import magpy

```
$ python
$ >>> import magpy
```

1.2 From source

The instructions below will guide you through the process of building the code from source on **Linux**.

1. Clone the magpy code

```
$ git clone https://github.com/owlas/magpy
```

2. You'll need a C++11 compatible compiler (g++>=4.9 recommended)

3. You will also the LAPACK and BLAS libraries. On Debian systems these can be obtained through the apt repositories

```
$ apt install liblapacke-dev
```

4. Build the magpy C++ library with with your compiler as <CXX>

```
$ cd magpy
$ make CXX=<CXX> libmoma.so
```

5. You can build and run the tests from the same makefile

```
$ cd magpy
$ make CXX=<CXX> run-tests
```

6. To build the python interface you'll need to obtain all the python dependencies in the `requirements.txt` file.

7. Once you have all of the dependencies you can install magpy

```
$ cd magpy
$ CXX=<CXX> pip install .
$ python
$ >>> import magpy
```

1.3 From source (Intel compilers)

Magpy has been optimised for Intel architectures and you can take advantage of this by taking a few extra steps:

1. Clone the magpy code

```
$ git clone https://github.com/owlas/magpy
```

2. Ensure you have the Intel compilers in your path (`icc` and `icpc`)

3. Tell magpy where to find your MKL files

```
$ export MKLROOT=/path/to/mkl/install/directory
```

4. You will also the LAPACK and BLAS libraries. On Debian systems these can be obtained through the apt repositories

```
$ apt install liblapacke-dev
```

5. Build the magpy C++ library with the intel compilers. The correct build flags should be taken care of for you

```
$ cd magpy
$ make CXX=icpc libmoma.so
```

6. You can build and run the tests from the same makefile

```
$ cd magpy
$ make CXX=icpc run-tests
```

7. To build the python interface you'll need to obtain all the python dependencies in the `requirements.txt` file.

8. Once you have all of the dependencies you can install magpy

```
$ cd magpy
$ CC=icc CXX=icpc pip install .
$ python
$ >>> import magpy
```

CHAPTER 2

Getting started - a single particle

In this tutorial, we'll simulate the stochastic dynamics of a single nanoparticle. We model clusters of nanoparticles using the `magpy.Model` class. In this case we only have a single particle in our cluster. The first step is to import magpy.

```
In [1]: import magpy as mp
```

To create our model, we need to specify the geometry and material properties of the system. The units and purpose of each property is defined below.

Name	Description	Units
Radius	The radius of the spherical particle	m
Anisotropy	Magnitude of the anisotropy	J/m^3
Anisotropy axis	Unit vector indicating the direction of the anisotropy	•
Magnetisation	Saturation magnetisation of every particle in the cluster	A/m
Magnetisation direction	Unit vector indicating the initial direction of the magnetisation	•
Location	Location of the particle within the cluster	m
Damping	The damping constant of every particle in the cluster	•
Temperature	The ambient temperature of the cluster (fixed)	K

Note: radius, anisotropy, anisotropy_axis, magnetisation_direction, and location vary for each particle and must be specified as a list.

```
In [2]: single_particle = mp.Model(
            radius = [12e-9],
            anisotropy = [4e4],
            anisotropy_axis = [[0., 0., 1.]],
```

```
        magnetisation_direction = [[1., 0., 0.]],
        location = [[0., 0., 0.]],
        damping = 0.1,
        temperature = 300.,
        magnetisation = 400e3
    )
```

2.1 Simulate

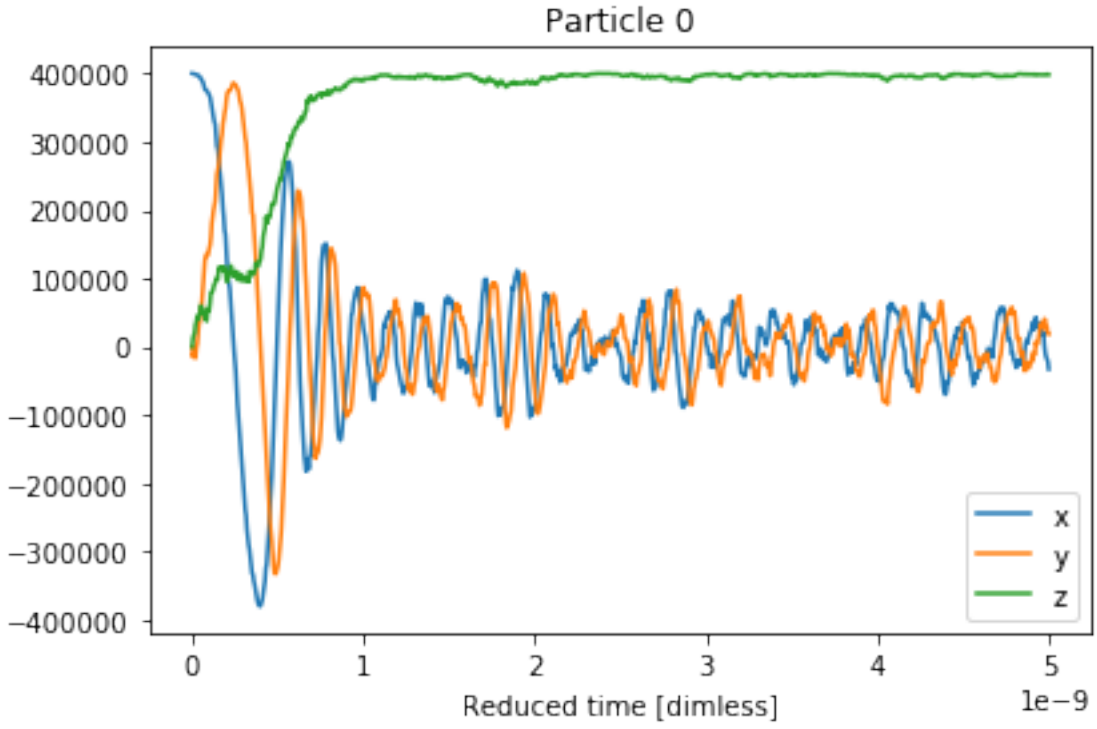
A simulation in magpy consists of simulating the magnetisation vector of the particle in time. In the model above we specified the initial magnetisation vector along the x-axis and the anisotropy along the z-axis. Since it is energetically favourable for the magnetisation to align with its anisotropy axis, we should expect the magnetisation to move toward the z-axis. With some random fluctuations.

The `simulate` function is called with the following parameters: - `end_time` the length of the simulation in seconds - `time_step` the time step of the integrator in seconds - `max_samples` in order to save memory, the output is down/upsampled as required. So if you simulate a billion steps, you can only save the state at 1000 regularly spaced intervals. - `seed` for reproducible simulations you should always choose the seed.

```
In [3]: results = single_particle.simulate(
            end_time = 5e-9,
            time_step = 1e-14,
            max_samples=1000,
            seed = 1001
        )
```

The `x,y,z` components of the magnetisation can be visualised with the `.plot()` function.

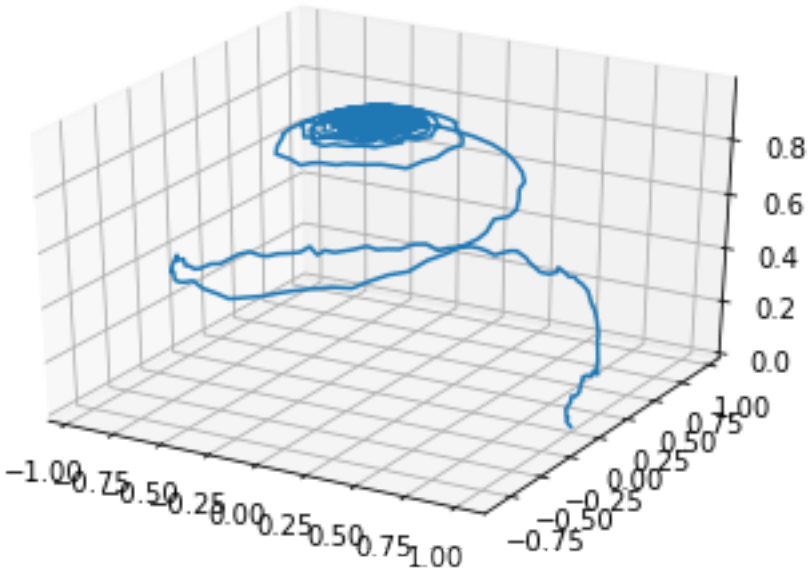
```
In [4]: results.plot()
```



We can also access this data directly and plot it however we like! In this example, we normalise the magnetisation and plot it in 3d space.

In [5]:
```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
%matplotlib inline

Ms = 400e3
time = results.time
mx = results.x[0] / Ms # particle 0
my = results.y[0] / Ms # particle 0
mz = results.z[0] / Ms # particle 0

fg = plt.figure()
ax = fg.add_subplot(111, projection='3d')
ax.plot3D(mx, my, mz)
```

Out[5]: [<mpl_toolkits.mplot3d.art3d.Line3D at 0x7f4f95319a58>]



CHAPTER 3

Thermal equilibrium of a single particle

In a large ensemble of identical systems each member will have a different state due to thermal fluctuations, even if all the systems were initialised in the same initial state.

As we integrate the dynamics of the ensemble we will have a distribution of states (i.e. the states of each member of the system). However, as the ensemble evolves, the distribution over the states eventually reaches a stationary distribution: the Boltzmann distribution. Even though the state of each member in the ensemble contitues to fluctuate, the ensemble as a whole is in a stastistical equilibrium (thermal equilibrium).

For an ensemble of single particles, we can compute the Boltzmann distribution by hand. In this example, we compare the analytical solution with the result of simulating an ensemble with Magpy.

3.1 Problem setup

A single particle has a uniaxial anisotropy axis K and a magnetic moment of three components (x,y,z components). The angle θ is the angle between the magnetic moment and the anisotropy axis.

3.2 Boltzmann distribution

The Boltzmann distribution represents of states over the ensemble; here the state is the **solid angle** $\phi = \sin(\theta)$ (i.e. the distribution over the surface of the sphere). The distribution is parameterised by the temperature of the system and the energy landscape of the problem.

$$p(\theta) = \frac{\sin(\theta) e^{-E(\theta)/(K_B T)}}{Z}$$

where Z is called the partition function:

$$Z = \int_\theta \sin(\theta) e^{-E(\theta)/(K_B T)} \mathrm{d}\theta$$

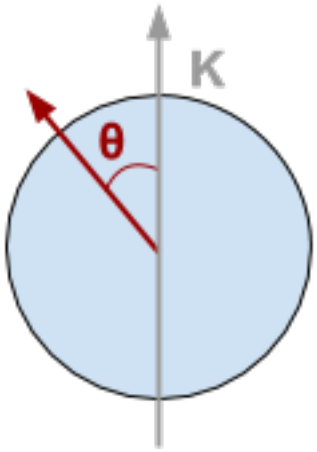


Fig. 3.1: Diagram of a single particle

3.3 Stoner-Wohlfarth model

The energy function for a single domain magnetic nanoparticle is given by the Stoner-Wohlfarth equation:

$$\frac{E(\theta)}{K_B T} = -\sigma \cos^2 \theta$$

where σ is called the normalised anisotropy strength:

$$\sigma = \frac{KV}{K_B T}$$

3.3.1 Functions for analytic solution

```
In [2]: import numpy as np

        # anisotropy energy of the system
        def anisotropy_e(theta, sigma):
            return -sigma*np.cos(theta)**2
```

```
In [17]: # numerator of the Boltzmann distribution
         # (i.e. without the partition function Z)
         def p_unorm(theta, sigma):
             return np.sin(theta)*np.exp(-anisotropy_e(theta, sigma))
```

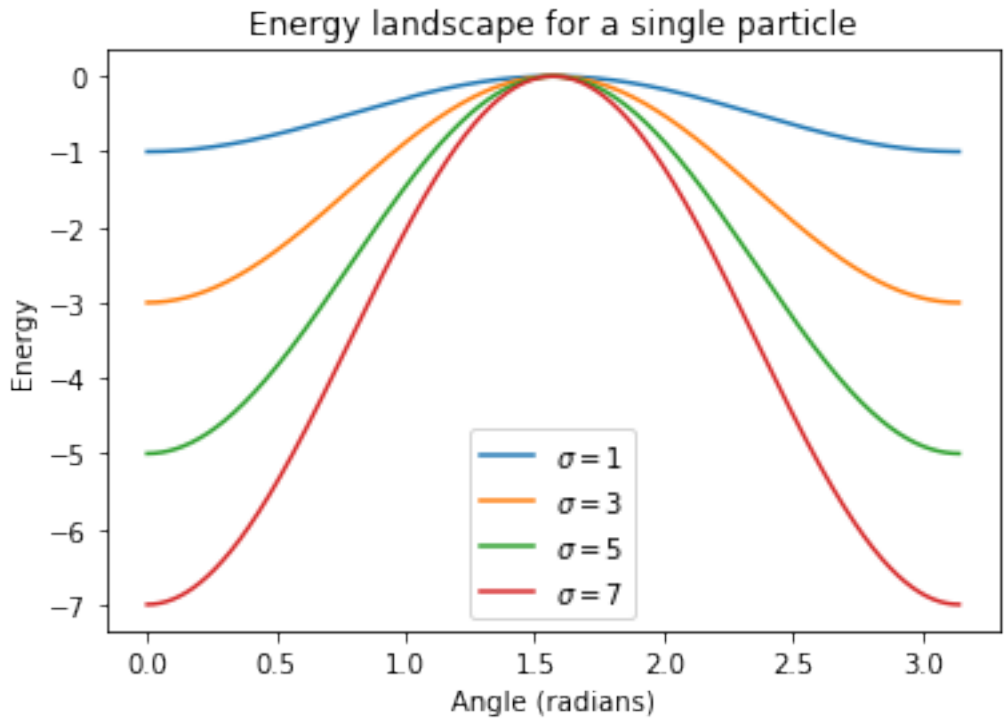
We use the quadrature rule to numerically evaluate the partition function Z.

```
In [30]: from scipy.integrate import quad

         # The analytic Boltzmann distribution
         def boltzmann(thetas, sigma):
             Z = quad(lambda t: p_unorm(t, sigma), 0, thetas[-1])[0]
             distribution = np.array([
                 p_unorm(t, sigma) / Z for t in thetas
             ])
             return distribution
```

Energy landscape

We can plot the energy landscape (energy as a function of the system variables)

```
In [31]: import matplotlib.pyplot as plt
         %matplotlib inline

In [74]: thetas = np.linspace(0, np.pi, 1000)
         sigmas = [1, 3, 5, 7]

         e_landscape = [anisotropy_e(thetas, s) for s in sigmas]
         for s, e in zip(sigmas, e_landscape):
             plt.plot(thetas, e, label='$\sigma={}$'.format(s))
         plt.legend(); plt.xlabel('Angle (radians)'); plt.ylabel('Energy')
         plt.title('Energy landscape for a single particle');
```

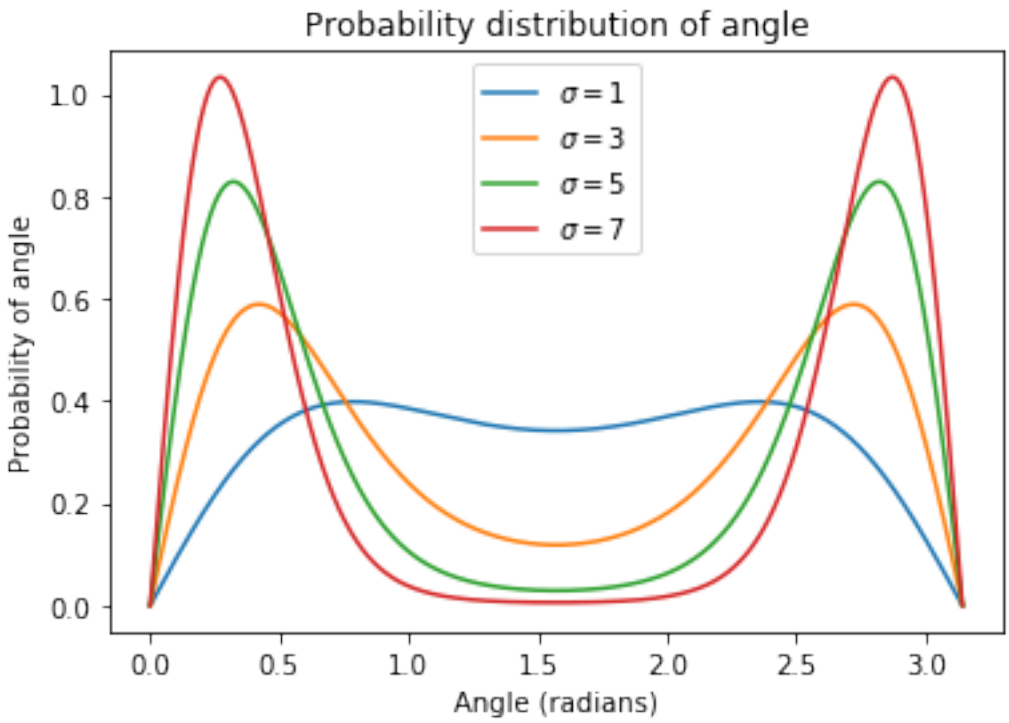


We observe that: - The energy of the system has two minima: one alongside each direction of the anisotropy axis. - The minima are separated by a maxima: perpendicular to the anisotropy axis. - Stronger anisotropy increases the size of the energy barrier between the two minima.

Equilibrium distribution (Boltzmann)

We can also plot the equilibrium distribution of the system, which is the probability distribution over the system states in a large ensemble of systems.

```
In [73]: p_dist = [boltzmann(thetas, s) for s in sigmas]

         for s, p in zip(sigmas, p_dist):
             plt.plot(thetas, p, label='$\sigma={}$'.format(s))
         plt.legend(); plt.xlabel('Angle (radians)')
         plt.ylabel('Probability of angle')
         plt.title('Probability distribution of angle');
```



What does this mean? If we had an ensemble of single particles, the distribution of the states of those particles varies greatly depending on σ. Remember we can decrease σ by reducing the anisotropy strength or particle size or by increasing the temperature. - When σ is high, most of the particles in the ensemble will be found closely aligned with the anisotropy axis. - When σ is low, the states of the particles are more evenly distributed.

3.3.2 Magpy equilibrium

Using Magpy, we can simulate the dynamics of the state of a single nanoparticle. If we simulate a large ensemble of these systems for 'long enough', the distribution of states will reach equilibrium. If Magpy is implemented correctly, we should recover the analytical distribution from above.

Set up the model

Select the parameters for the single particle

```
In [61]: # These parameters will determine the distribution
         K = 1e5
         r = 7e-9
         T = 300
         kdir = [0., 0., 1.]

         # These parameters affect the dynamics but
         # have no effect on the equilibrium
         Ms = 400e3
         location = [0., 0., 0.]
         alpha=1.0
         initial_direction = [0., 0., 1.]

         # Normalised anisotropy strength KV/KB/T
         V = 4./3 * np.pi * r**3
```

```
kb = mp.core.get_KB()
sigma = K * V / kb / T
print(sigma)
```

```
34.68792671050298
```

```
In [62]: import magpy as mp

         single_particle = mp.Model(
             anisotropy=[K],
             anisotropy_axis=[kdir],
             damping=alpha,
             location=[location],
             magnetisation=Ms,
             magnetisation_direction=[initial_direction],
             radius=[r],
             temperature=T
         )
```

Create an ensemble

From the single particle we create an ensemble of 10,000 identical particles.

```
In [63]: particle_ensemble = mp.EnsembleModel(
             base_model=single_particle, N=10000
         )
```

Simulate

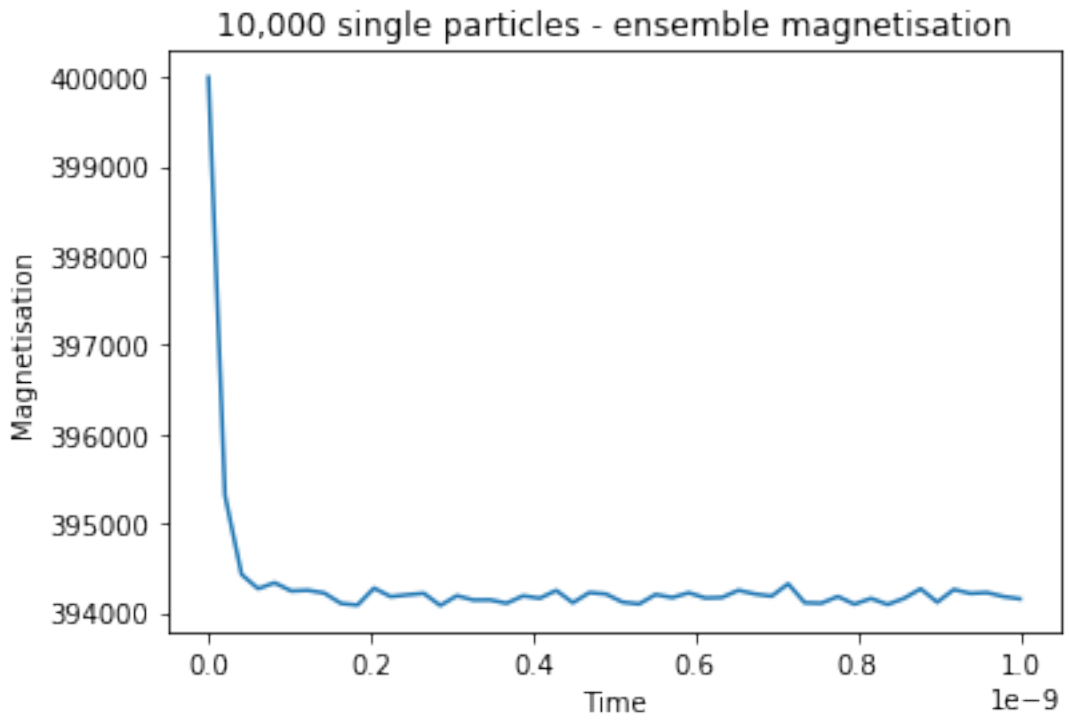
Now we simulate! We don't need to simulate for very long because σ is very high and the system will reach equilibrium quickly.

```
In [66]: res = particle_ensemble.simulate(
             end_time=1e-9, time_step=1e-12, max_samples=50,
             random_state=1001, implicit_solve=True
         )
```

```
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    0.0s remaining:    0.0s
[Parallel(n_jobs=1)]: Done   2 out of   2 | elapsed:    0.1s remaining:    0.0s
[Parallel(n_jobs=1)]: Done   3 out of   3 | elapsed:    0.1s remaining:    0.0s
[Parallel(n_jobs=1)]: Done   4 out of   4 | elapsed:    0.1s remaining:    0.0s
[Parallel(n_jobs=1)]: Done 10000 out of 10000 | elapsed:  3.8min finished
```

Check that we have equilibriated

```
In [71]: plt.plot(res.time, res.ensemble_magnetisation())
         plt.title('10,000 single particles - ensemble magnetisation')
         plt.xlabel('Time'); plt.ylabel('Magnetisation');
```



We can see that the system has reached a local minima. We could let the simulation run until the ensemble relaxes into both minima but it would take a very long time because the energy barrier is so high in this example.

Compute theta

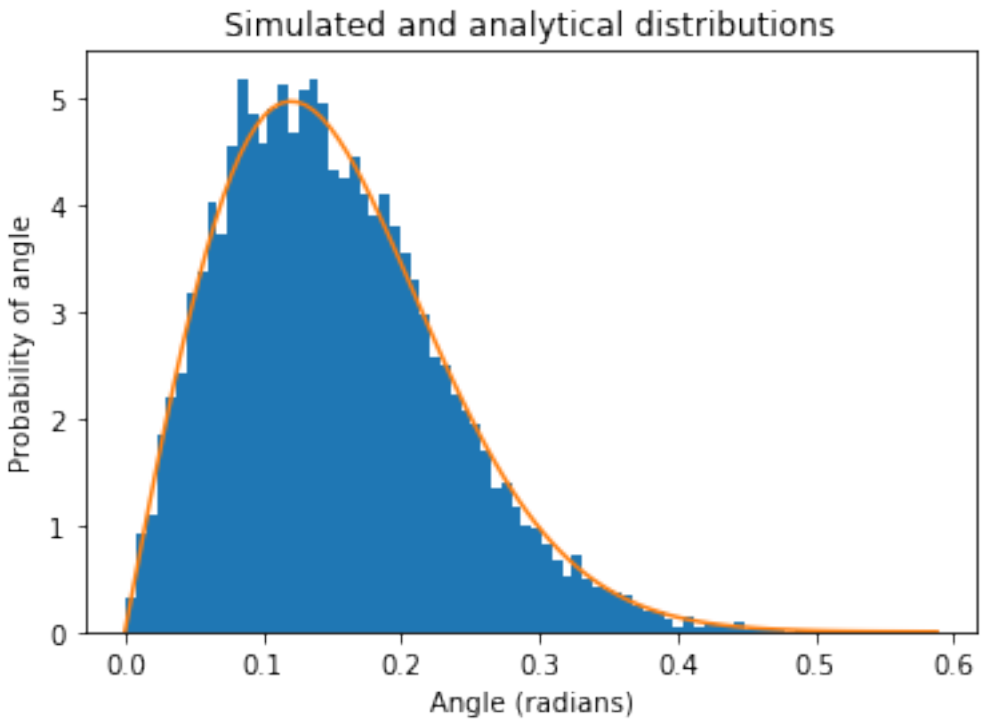
The results of the simulation are x,y,z coordinates of the magnetisation of each particle in the ensemble. We need to convert these into angles.

```
In [76]: M_z = np.array([state['z'][0] for state in res.final_state()])
         m_z = M_z / Ms
         simulated_thetas = np.arccos(m_z)
```

Compare to analytical solution

Now we compare our empirical distribution of states to the analytical distribution that we computed above.

```
In [102]: theta_grid = np.linspace(0.0, simulated_thetas.max(), 100)
          analytical_probability = boltzmann(theta_grid, sigma)

          plt.hist(simulated_thetas, normed=True, bins=80, label='Simulated');
          plt.plot(theta_grid, analytical_probability, label='Analytical')
          plt.title('Simulated and analytical distributions')
          plt.xlabel('Angle (radians)'); plt.ylabel('Probability of angle');
```



The results look good! We could simulate an even bigger ensemble to produce a smoother empirical distribution.

CHAPTER 4

Two particle equilibrium

If you haven't read the `One particle equilibrium` notebook yet, go and read it now.

In the previous notebook we showed that we can use Magpy to compute the correct thermal equilibrium for a single particle. However, we also need to check that the interactions are correctly implemented by simulating the thermal equilibrium of multiple interacting particles.

In this notebook we'll simulate an ensemble of two particle systems with Magpy. Instead of computing the distribution analytically, we will use the Metropolis Markov-Chain Monte-Carlo technique to generate the correct equilibrium.

Acknowledgements

Many thanks to Jonathon Waters for the terse python implementation of the Metropolis algorithm!

Problem setup

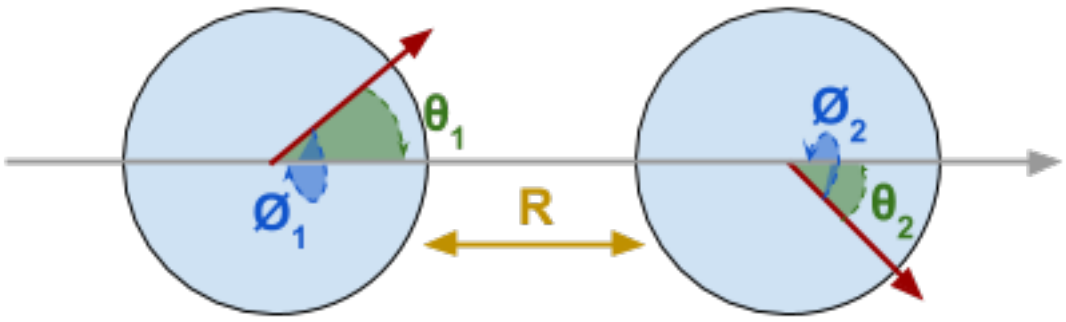


Fig. 4.1: Diagram of two anisotropy-aligned particles

In this example the system comprises two identical particles separated by a distance R. The particles have their anisotropy axes in the same direction. We are interested in the following four variables: the angle between the particle's moments and the anisotropy axis θ_1, θ_2 and the rotational (azimuth) angle of the particles around the anisotropy axis ϕ_1, ϕ_2

Modules

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
        from mpl_toolkits.mplot3d import axes3d
        from tqdm import tqdm_notebook
        #import tqdm
        import magpy as mp
        %matplotlib inline
```

4.1 Metropolis MCMC

4.1.1 Energy terms

Anisotropy

The energy contribution from the anisotropy of a single particle i is:

$$E_i^a = V_i \vec{K}_i \cdot \vec{m}_i$$

```
In [2]: def e_anisotropy(moments, anisotropy_axes, V, K, particle_id):
            cos_t = np.sum(moments[particle_id, :]*anisotropy_axes[particle_id, :])
            return -K*V*cos_t**2
```

Dipolar interaction energy

The energy contribution from N particles $j = 1, 2, \ldots, N$ interacting with a single particle i:

$$E_i^d = \sum_j \frac{V_i^2 M_s^2 \mu_0 \left(3(\vec{m}_i \cdot \vec{r}_{ij})(\vec{m}_j \cdot \vec{r}_{ij}) - \vec{m}_i \cdot \vec{m}_j\right)}{4\pi \left|r\right|^3}$$

```
In [3]: def e_dipole(moments, positions, Ms, V, particle_id):
            mu_0 = mp.core.get_mu0()
            mask = np.ones(moments.shape[0], dtype=bool)
            mask[particle_id] = False
            rs = positions[mask]-positions[particle_id, :]
            mod_rs = np.linalg.norm(rs, axis=1)
            rs[:, 0] = rs[:, 0] / mod_rs
            rs[:, 1] = rs[:, 1] / mod_rs
            rs[:, 2] = rs[:, 2] / mod_rs
            m1_m2 = np.sum(moments[particle_id, :]*moments[mask], axis=1)
            m1_r = np.sum(moments[particle_id, :]*rs, axis=1)
            m2_r = np.sum(moments[mask]*rs, axis=1)
            numer = (V**2)*(Ms**2)*mu_0*(3*m1_r*m2_r - m1_m2)
            denom = 4*np.pi*np.power(mod_rs, 3)
            return -np.sum(numer/denom)
```

Total energy

The total energy contribution from a single particle in the ensemble is:

$$E_i = E_i^a + E_i^d$$

```
In [4]: def e_total(moments, positions, anisotropy_axes, Ms, V, K, particle_id):
            return (
                e_dipole(moments, positions, Ms, V, particle_id)
                + e_anisotropy(moments, anisotropy_axes, V, K, particle_id)
            )
```

4.1.2 The Monte-Carlo algorithm

1. Initialise each spin in the system
2. Randomly choose a particle in the system and change it's orientation
3. Compute ΔE the change in total energy arising from changing the particle orienation
4. if

- $\Delta E < 0$ then we accept the new state and store it
- $\Delta E > 0$ we accept the new state and store it with probability $p = e^{\Delta E/(K_B T)}$
- otherwise we reject the new state

5. Return to 2 until desired number of samples

Once we run this loop many times, we'll have a list of accepted samples of the system state. The distribution of this ensemble of states is **guaranteed** to converge to the true distribution. Monte-Carlo is much faster than numerical integration methods when we have many particles.

```
In [5]: def sphere_point():
            theta = 2*np.pi*np.random.rand()
            phi = np.arccos(1-2*np.random.rand())
            return np.array([np.sin(phi)*np.cos(theta), np.sin(phi)*np.sin(theta), np.cos(phi)])

        def MH(positions, ani_axis, spins, Neq, Nsamps, SampRate, Ms, V, K, T, seed=42):
            np.random.seed(seed)
            k_b = mp.core.get_KB()
            test = np.copy(spins)
            Ntot = Neq+Nsamps*SampRate
            Out = np.zeros([spins.shape[0], spins.shape[1], Nsamps])
            ns = 0
            for n in tqdm_notebook(range(Ntot)):
                # pick a random spin
                i = int(np.random.rand(1)*positions.shape[0])
                # pick a random dir
                test[i, :] = sphere_point()
                dE = e_total(test, positions, ani_axis, Ms, V, K, i) - \
                     e_total(moments, positions, ani_axis, Ms, V, K, i)
                if(np.random.rand(1) < np.exp(-dE/(k_b*T))):
                    spins[i, :] = test[i, :]
                else:
                    test[i, :] = spins[i, :]
                if (n >= Neq and (n-Neq)%SampRate == 0):
                    Out[:, :, ns] = np.copy(spins)
                    ns += 1
            return Out
```

4.1.3 Parameter set up

Now we set the parameters for the two particle system. Both particles are identical and have their anisotropy axes aligned with the z direction.

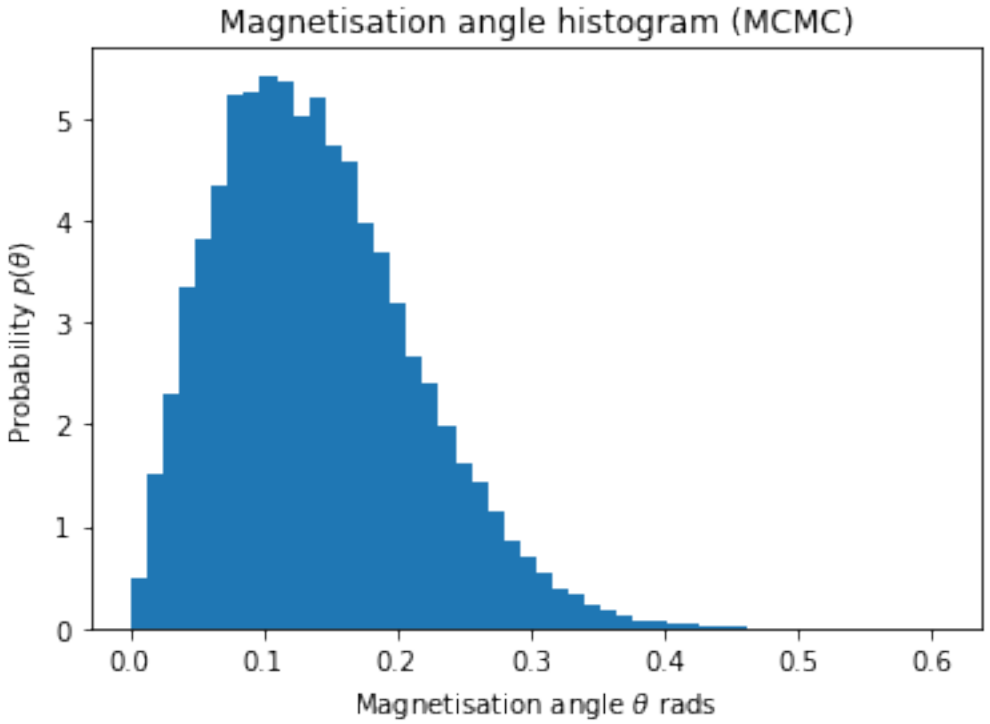
```
In [6]: N = 2 # Two particles
        T = 330 # temperature
        K = 1e5 # anisotropy strength
        R = 9e-9 # distance between two particles
        r = 7e-9 # radius of the particles
        V = 4./3 * np.pi * r**3 # volume of particle
        Ms = 4e5 # saturation magnetisation

        #                       particle 1    particle 2
        positions = np.array([[0., 0., 0.], [0., 0., R]])
        moments = np.array([sphere_point(), sphere_point()])
        anisotropy_axes = np.array([[0., 0., 1.], [0., 0., 1.]])
```

4.1.4 Run the MCMC sampler!

This will take some time

```
In [7]: output = MH(positions, anisotropy_axes, moments, 100000, 600000, 20, Ms, V, K, T, 0)
```

```
In [9]: thetas = np.arccos(output[:, 2, :])
        plt.hist(thetas[0], bins=50, normed=True)
        plt.title('Magnetisation angle histogram (MCMC)')
        plt.xlabel('Magnetisation angle $\\theta$ rads')
        plt.ylabel('Probability $p(\\theta)$');
```



4.2 Magpy - Dynamical Simulation

We now use Magpy to simulate a large ensemble of the identical two-particle system. Once the ensemble has reached a stationary distribution, we determine the distribution of magnetisation angles over the ensemble. We expect this distribution to match the equilibrium distribution determined by the MCMC sampler.

4.2.1 Define the Magpy model

```
In [10]: # additionally we must specify damping
         alpha = 0.1

         # We build a model of the two particles
         base_model = mp.Model(
             anisotropy=[K,K],
             anisotropy_axis=anisotropy_axes,
             damping=alpha,
             location=positions,
             magnetisation=Ms,
             magnetisation_direction=moments,
             radius=[r, r],
             temperature=T
         )

         # Create an ensemble of 50,000 identical models
         ensemble = mp.EnsembleModel(50000, base_model)
```

4.2.2 Simulate the ensemble!

Now we run the dynamical simulation using an implicit solver. Each model is simulated for 1ns.

```
In [11]: res = ensemble.simulate(end_time=1e-9, time_step=1e-12,
                                  max_samples=500, random_state=1002,
                                  n_jobs=-1, implicit_solve=True,
                                  interactions=True)
```

```
[Parallel(n_jobs=-1)]: Done   2 tasks      | elapsed:    0.1s
[Parallel(n_jobs=-1)]: Done 176 tasks      | elapsed:    1.9s
[Parallel(n_jobs=-1)]: Done 536 tasks      | elapsed:    5.8s
[Parallel(n_jobs=-1)]: Done 1040 tasks      | elapsed:   11.5s
[Parallel(n_jobs=-1)]: Done 1688 tasks      | elapsed:   18.9s
[Parallel(n_jobs=-1)]: Done 2480 tasks      | elapsed:   27.4s
[Parallel(n_jobs=-1)]: Done 3416 tasks      | elapsed:   37.3s
[Parallel(n_jobs=-1)]: Done 4496 tasks      | elapsed:   48.5s
[Parallel(n_jobs=-1)]: Done 5720 tasks      | elapsed:  1.0min
[Parallel(n_jobs=-1)]: Done 7088 tasks      | elapsed:  1.3min
[Parallel(n_jobs=-1)]: Done 8600 tasks      | elapsed:  1.5min
[Parallel(n_jobs=-1)]: Done 10256 tasks      | elapsed:  1.8min
[Parallel(n_jobs=-1)]: Done 12056 tasks      | elapsed:  2.2min
[Parallel(n_jobs=-1)]: Done 14000 tasks      | elapsed:  2.5min
[Parallel(n_jobs=-1)]: Done 16088 tasks      | elapsed:  2.9min
[Parallel(n_jobs=-1)]: Done 18320 tasks      | elapsed:  3.3min
[Parallel(n_jobs=-1)]: Done 20696 tasks      | elapsed:  3.7min
[Parallel(n_jobs=-1)]: Done 23216 tasks      | elapsed:  4.1min
[Parallel(n_jobs=-1)]: Done 25880 tasks      | elapsed:  4.6min
[Parallel(n_jobs=-1)]: Done 28688 tasks      | elapsed:  5.1min
[Parallel(n_jobs=-1)]: Done 31640 tasks      | elapsed:  5.6min
```

```
[Parallel(n_jobs=-1)]: Done 34736 tasks      | elapsed:  6.2min
[Parallel(n_jobs=-1)]: Done 37976 tasks      | elapsed:  6.8min
[Parallel(n_jobs=-1)]: Done 41360 tasks      | elapsed:  7.4min
[Parallel(n_jobs=-1)]: Done 44888 tasks      | elapsed:  8.0min
[Parallel(n_jobs=-1)]: Done 48560 tasks      | elapsed:  8.7min
[Parallel(n_jobs=-1)]: Done 50000 out of 50000 | elapsed:  8.9min finished
```

4.2.3 Compute the final state

We use the `Results.final_state()` function to determine the state of each member of the ensemble after 1ns of simulation. The magnetisation angle is computed as the cosine of the z-axis component of magnetisation.

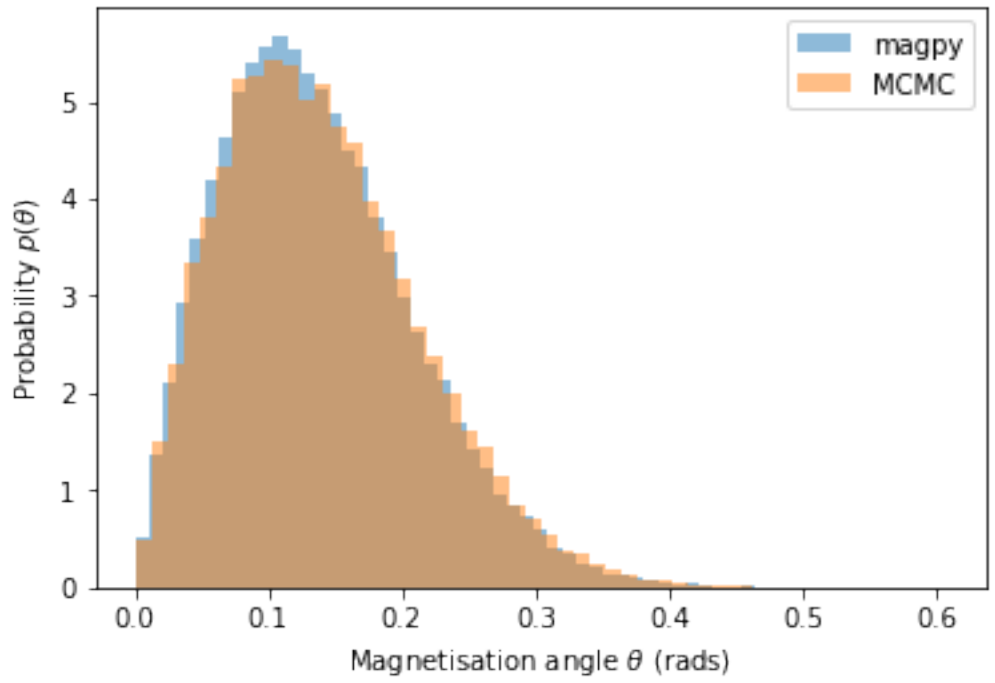
```
In [12]: m_z0 = np.array([state['z'][0] for state in res.final_state()])/Ms
         m_z1 = np.array([state['z'][1] for state in res.final_state()])/Ms
         theta0 = np.arccos(m_z0)
         theta1 = np.arccos(m_z1)
```

4.3 Compare results

4.3.1 Single variable comparison

Below we compare the magnetisation angle distribution for a single particle as simulated with Magpy and the MCMC algorithm.

```
In [13]: plt.hist(theta0, bins=50, alpha=0.5, normed=True, label='magpy')
         plt.hist(thetas[0], bins=50, alpha=0.5, normed=True, label='MCMC')
         plt.legend();
         plt.xlabel('Magnetisation angle $\\theta$ (rads)')
         plt.ylabel('Probability $p(\\theta)$');
```

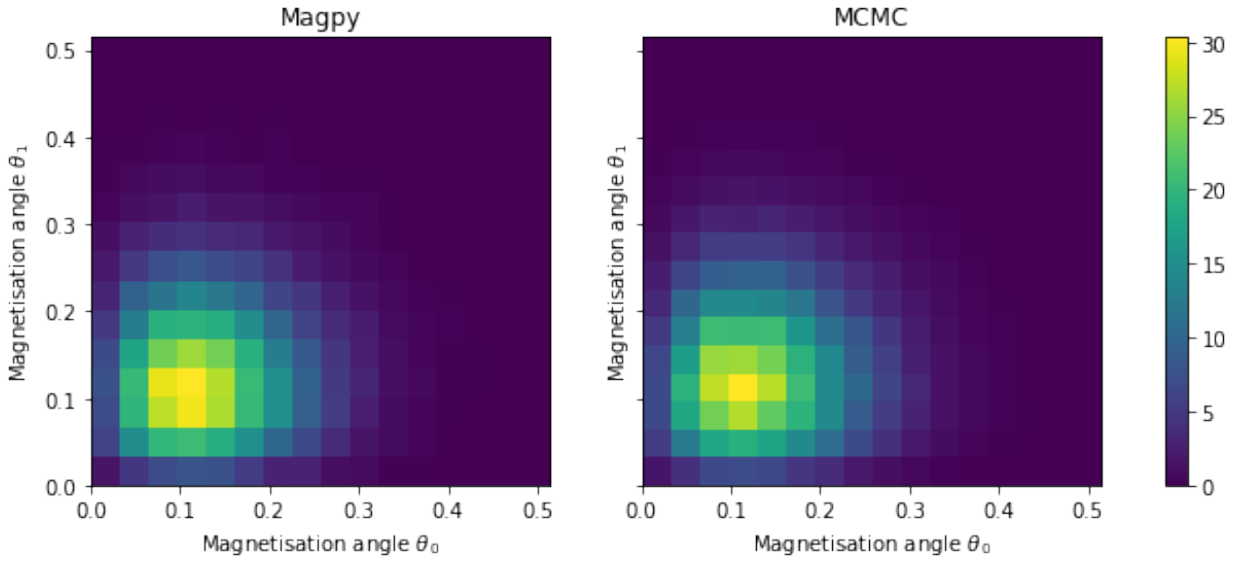


The results look to be a good match!

4.3.2 Join distribution comparison

Below we compare the joint distribution of θ_0 and θ_1 (the magnetisation angle of both particles). In other words, this is the probability distribution over the entire state space. It is important to compare the joint distributions because the two particles interact with one another, creating a dependence between the two magnetisation angles.

```
In [26]: fg, axs = plt.subplots(ncols=2, figsize=(11,4), sharey=True)
         histdat = axs[0].hist2d(theta0, theta1, bins=16, normed=True)
         axs[1].hist2d(thetas[0], thetas[1], bins=histdat[1], normed=True);
         for ax, title in zip(axs, ['Magpy', 'MCMC']):
             ax.set_xlabel('Magnetisation angle $\\theta_0$')
             ax.set_ylabel('Magnetisation angle $\\theta_1$')
             ax.set_title(title)
         fg.colorbar(histdat[3], ax=axs.tolist());
```



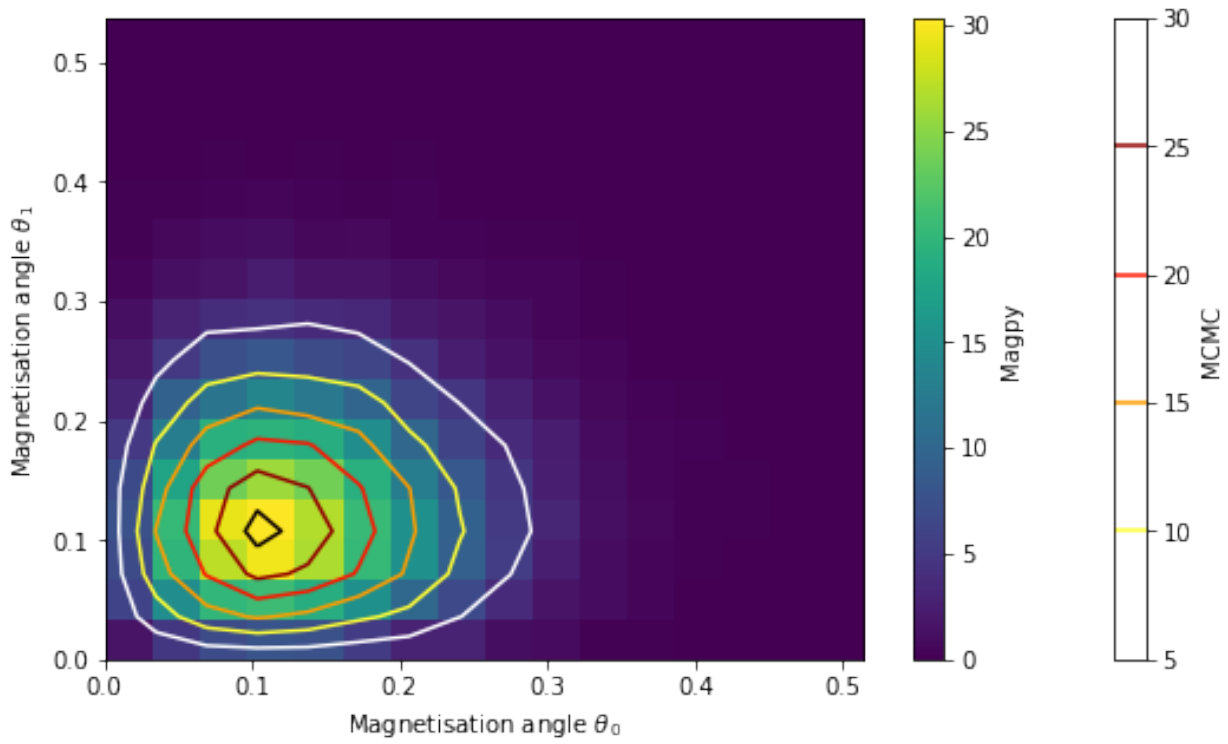
Alternatively compare using a kernel density function

An alternative method to visually compare the two distributions is to construct a kernel density estimation one set of results and overaly it on a histogram of the other.

```
In [27]: from scipy.stats import gaussian_kde
         kde = gaussian_kde(thetas)

         tgrid_x = np.linspace(theta0.min(), theta0.max(), 16)
         tgrid_y = np.linspace(theta1.min(), theta1.max(), 16)
         tgrid_x, tgrid_y = np.meshgrid(tgrid_x, tgrid_y)
         Z = np.reshape(kde(np.vstack([tgrid_x.ravel(), tgrid_y.ravel()])).T, tgrid_x.shape)
```

```
In [57]: fg, ax = plt.subplots(figsize=(9,5))

         hist = ax.hist2d(theta0, theta1, bins=16, normed=True)
         contour = ax.contour(tgrid_x, tgrid_y, Z, cmap='hot_r')

         fg.colorbar(contour, label='MCMC')
         fg.colorbar(hist[3], label='Magpy')
```

```
        ax.set_xlabel('Magnetisation angle $\\theta_0$')
        ax.set_ylabel('Magnetisation angle $\\theta_1$');
```



4.4 Sanity check: no interactions

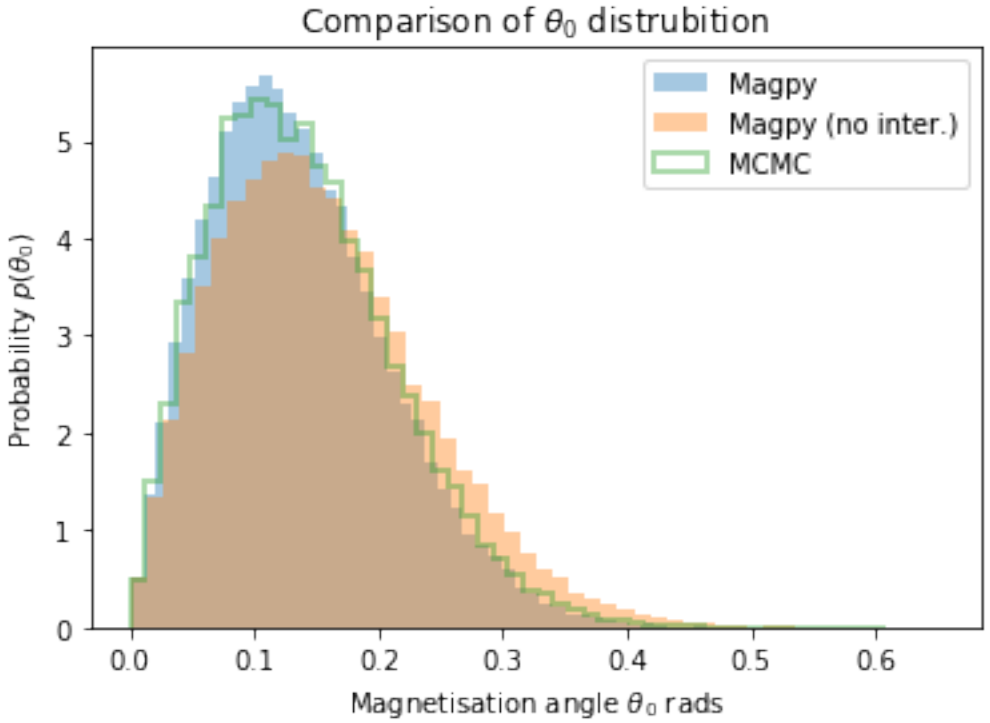
To ensure that the interactions are having a significant effect on the joint distribution, we simulate the same system but disable the interaction false (simply set `interactions=False`).

```
In [58]: res_noi = ensemble.simulate(end_time=1e-9, time_step=1e-12,
                                     max_samples=500, random_state=1002,
                                     n_jobs=-1, implicit_solve=True,
                                     interactions=False)
         m_z0 = np.array([state['z'][0] for state in res_noi.final_state()])/Ms
         m_z1 = np.array([state['z'][1] for state in res_noi.final_state()])/Ms
         theta0_noi = np.arccos(m_z0)
         theta1_noi = np.arccos(m_z1)
```

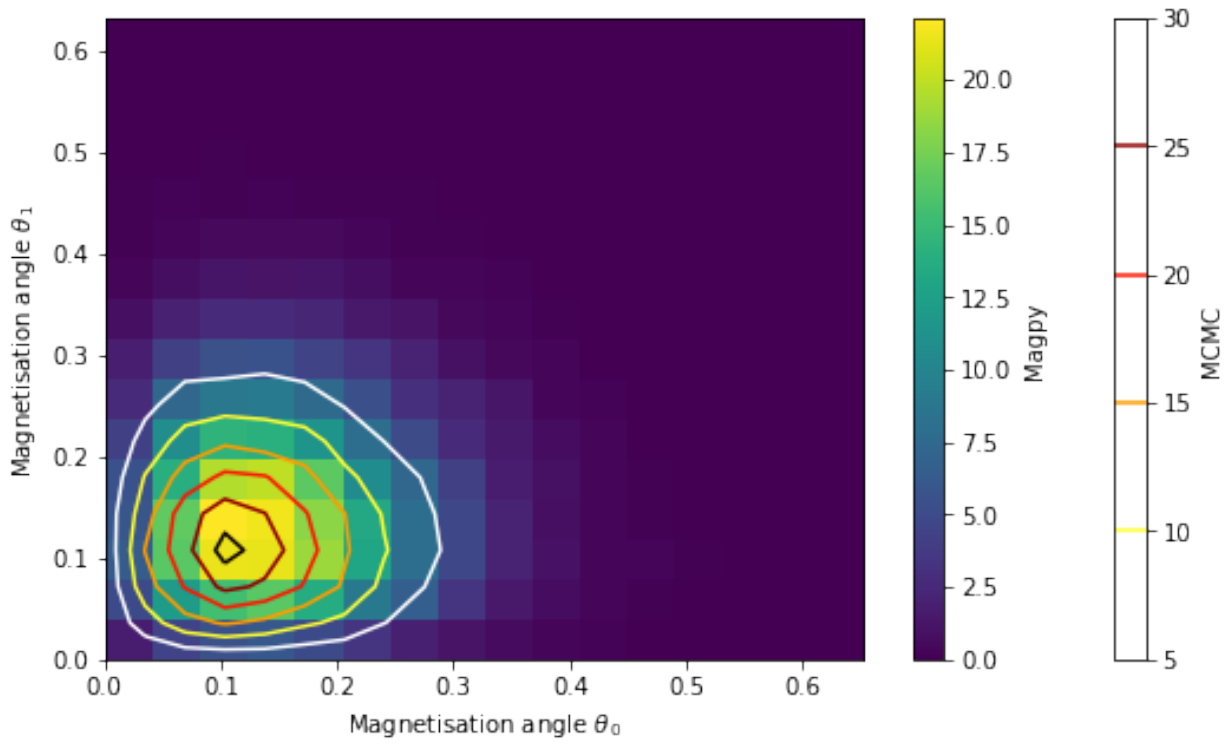
```
[Parallel(n_jobs=-1)]: Done   2 tasks      | elapsed:    0.1s
[Parallel(n_jobs=-1)]: Done 176 tasks      | elapsed:    1.6s
[Parallel(n_jobs=-1)]: Done 536 tasks      | elapsed:    4.7s
[Parallel(n_jobs=-1)]: Done 1040 tasks      | elapsed:    8.9s
[Parallel(n_jobs=-1)]: Done 1688 tasks      | elapsed:   14.2s
[Parallel(n_jobs=-1)]: Done 2480 tasks      | elapsed:   20.7s
[Parallel(n_jobs=-1)]: Done 3416 tasks      | elapsed:   28.4s
[Parallel(n_jobs=-1)]: Done 4496 tasks      | elapsed:   37.2s
[Parallel(n_jobs=-1)]: Done 5720 tasks      | elapsed:   47.4s
[Parallel(n_jobs=-1)]: Done 7088 tasks      | elapsed:   58.5s
[Parallel(n_jobs=-1)]: Done 8600 tasks      | elapsed:  1.2min
[Parallel(n_jobs=-1)]: Done 10256 tasks      | elapsed:  1.4min
[Parallel(n_jobs=-1)]: Done 12056 tasks      | elapsed:  1.7min
```

```
[Parallel(n_jobs=-1)]: Done 14000 tasks      | elapsed:  2.0min
[Parallel(n_jobs=-1)]: Done 16088 tasks      | elapsed:  2.3min
[Parallel(n_jobs=-1)]: Done 18320 tasks      | elapsed:  2.6min
[Parallel(n_jobs=-1)]: Done 20696 tasks      | elapsed:  2.9min
[Parallel(n_jobs=-1)]: Done 23216 tasks      | elapsed:  3.3min
[Parallel(n_jobs=-1)]: Done 25880 tasks      | elapsed:  3.6min
[Parallel(n_jobs=-1)]: Done 28688 tasks      | elapsed:  4.0min
[Parallel(n_jobs=-1)]: Done 31640 tasks      | elapsed:  4.5min
[Parallel(n_jobs=-1)]: Done 34736 tasks      | elapsed:  4.9min
[Parallel(n_jobs=-1)]: Done 37976 tasks      | elapsed:  5.4min
[Parallel(n_jobs=-1)]: Done 41360 tasks      | elapsed:  5.9min
[Parallel(n_jobs=-1)]: Done 44888 tasks      | elapsed:  6.4min
[Parallel(n_jobs=-1)]: Done 48560 tasks      | elapsed:  6.9min
[Parallel(n_jobs=-1)]: Done 50000 out of 50000 | elapsed:  7.1min finished
```

In [63]:
```
plt.hist(theta0, bins=50, normed=True, alpha=0.4, label='Magpy')
plt.hist(theta0_noi, bins=50, normed=True, alpha=0.4, label='Magpy (no inter.)');
plt.hist(thetas[0], bins=50, histtype='step', lw=2, normed=True, alpha=0.4, label='MCMC')
plt.legend();
plt.xlabel('Magnetisation angle $\\theta_0$ rads')
plt.ylabel('Probability $p(\\theta_0)$');
plt.title('Comparison of $\\theta_0$ distrubition');
```



In [66]:
```
fg, ax = plt.subplots(figsize=(9,5))

hist = ax.hist2d(theta0_noi, theta1_noi, bins=16, normed=True)
contour = ax.contour(tgrid_x, tgrid_y, Z, cmap='hot_r')

fg.colorbar(contour, label='MCMC')
fg.colorbar(hist[3], label='Magpy')
ax.set_xlabel('Magnetisation angle $\\theta_0$')
ax.set_ylabel('Magnetisation angle $\\theta_1$');
```



The results show that the distributions cleary deviate when we ignore interactions.

CHAPTER 5

Convergence Tests

In this notebook, we visualise the results from the `magpy` convergence tests. These tests ensure that the numerical methods are implemented correctly.

Running the tests Before executing this notebook, you'll need to run the convergence tests. In order to do this you must: 1. Clone the Magpy repository at https://github.com/owlas/magpy 2. Compile the Magpy library 3. Compile and run the convergence tests: - `cd /path/to/magpy/directory` - `make libmoma.so` - `make test/convergence/run`

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
        %matplotlib inline

        datapath = '../../../test/convergence/output/'
```

5.1 Task 1 - 1D non-stiff SDE

We simulate the following 1-dimensional SDE, which has an analytic solution

$$dX(t) = aX(t)dt + bX(t)dW(t)$$

- $a = -1.0$
- $b = 1.0$

```
In [2]: # load results
        path = datapath + 'task1/'
        files = !ls {path}
        results = {name: np.fromfile(path + name) for name in files if name!='dt'}
        dts = np.fromfile(path + 'dt')
```

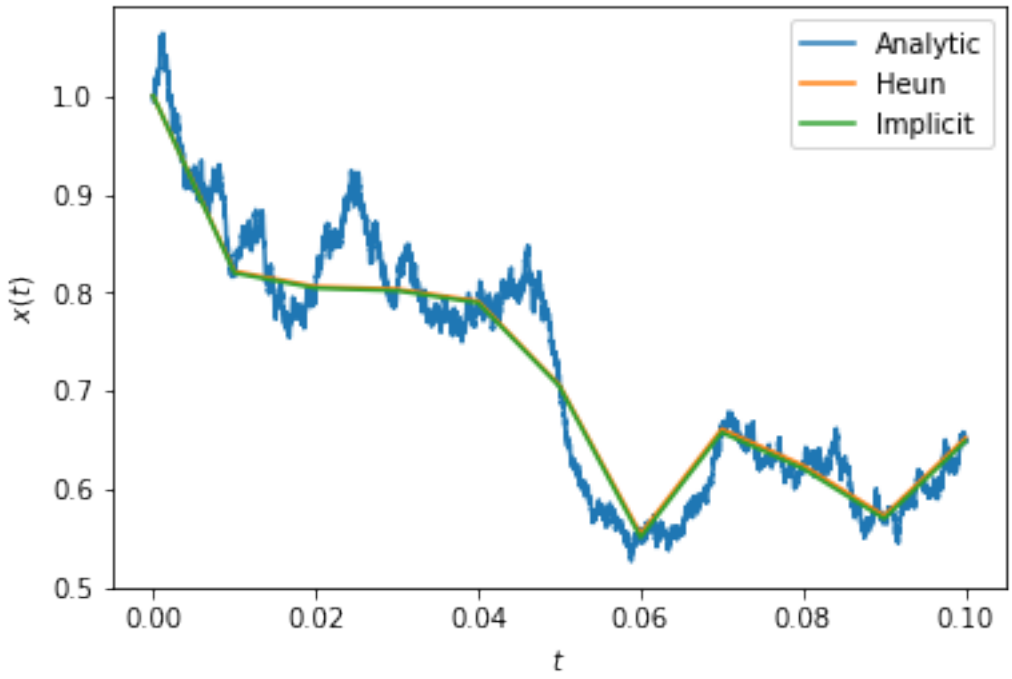
```
In [3]: tvecs = {
            i: dts[i] * np.arange(len(results['heun'+str(i)]))
            for i in range(len(dts))
        }
```

The following plot shows $x(t)$ for: - analytic - Heun with large time step - implicit with large time step

Both methods look accurate

```
In [4]: plt.plot(tvecs[0], results['true'], label='Analytic')
        plt.plot(tvecs[3], results['heun3'], label='Heun')
        plt.plot(tvecs[3], results['implicit3'], label='Implicit')
        plt.xlabel('$t$'); plt.ylabel('$x(t)$'); plt.legend()
```

```
Out[4]: <matplotlib.legend.Legend at 0x7f58a13b4dd8>
```



5.2 Task 2 - 1D stiff SDE

We introduce stiffness into the 1D problem in order to compare the explicit (Heun) and implicit methods. This is done here by creating a large separation in timescales (the deterministic dynamics are fast, the random dynamics are slow).

$$dX(t) = aX(t)dt + bX(t)dW(t)$$

- $a = -20.0$
- $b = 5.0$

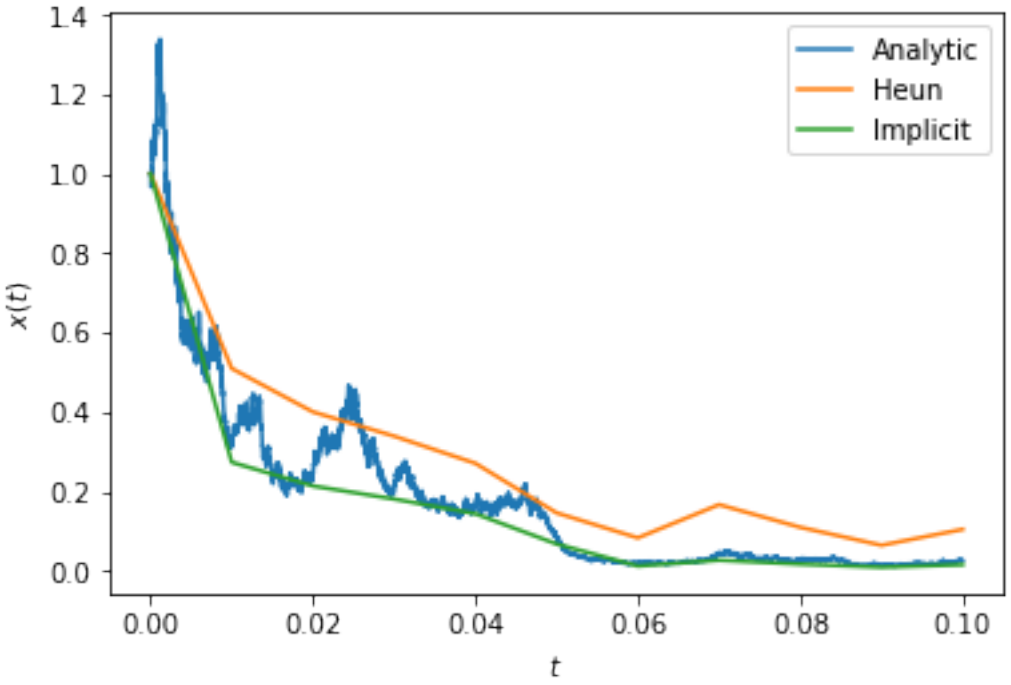
```
In [5]: # load results
        path = datapath + 'task2/'
        files = !ls {path}
        results = {name: np.fromfile(path + name) for name in files if name!='dt'}
        dts = np.fromfile(path + 'dt')
```

```
In [6]: tvecs = {
            i: dts[i] * np.arange(len(results['heun'+str(i)]))
            for i in range(len(dts))
        }
```

The plot of $x(t)$ shows that the explicit solver performs poorly on the stiff problem, as expected. The implicit solution looks accurate.

```
In [7]: plt.plot(tvecs[0], results['true'], label='Analytic')
        plt.plot(tvecs[3], results['heun3'], label='Heun')
        plt.plot(tvecs[3], results['implicit3'], label='Implicit')
        plt.xlabel('$t$'); plt.ylabel('$x(t)$'); plt.legend()
```

```
Out[7]: <matplotlib.legend.Legend at 0x7f58a107c940>
```



5.3 Task 3 - 1D unstable system

We introduce instability by simulating a system that drifts to infinity.

$$dX(t) = aX(t)dt + bX(t)dW(t)$$

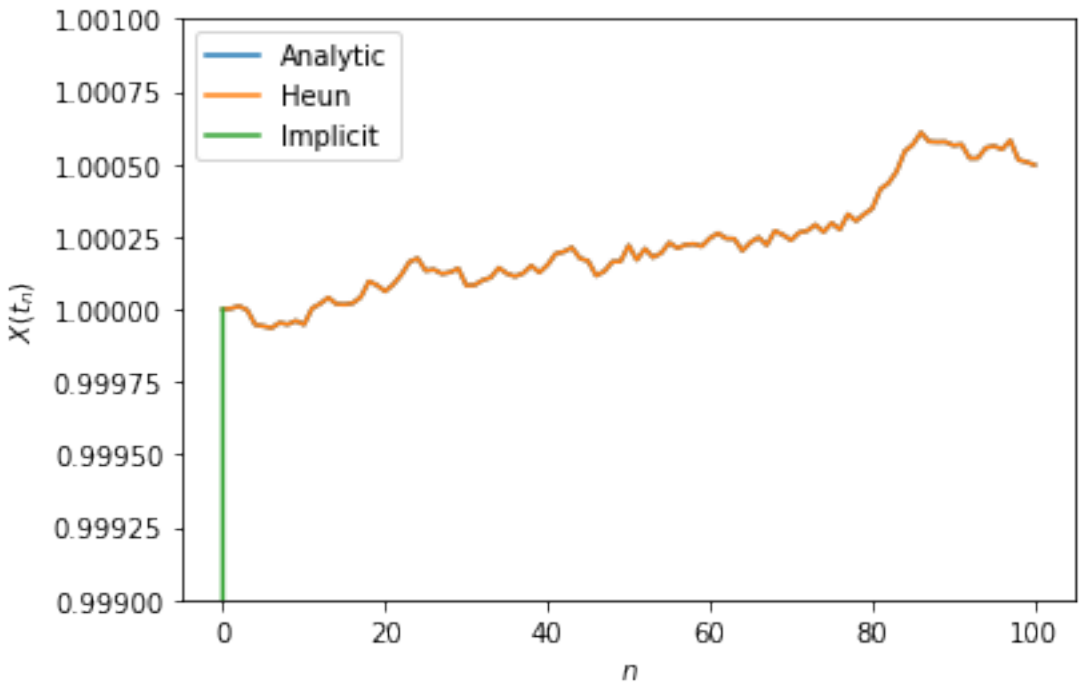
- $a = 1.0$
- $b = 0.1$

```
In [8]: # load results
        path = datapath + 'task3/'
        files = !ls {path}
        results = {name: np.fromfile(path + name) for name in files if name!='dt'}
```

The implicit solver blows up for these unstable problems. The explicit solver is able to track the trajectory closely.

```
In [9]: plt.plot(results['true'], label='Analytic')
        plt.plot(results['heun'], label='Heun')
        plt.plot(results['implicit'], label='Implicit')
        plt.legend(), plt.ylabel('$X(t_n)$'); plt.xlabel('$n$')
        plt.ylim(0.999, 1.001)
```

```
Out[9]: (0.999, 1.001)
```



5.4 Task 4 - Zero-temperature LLG convergence

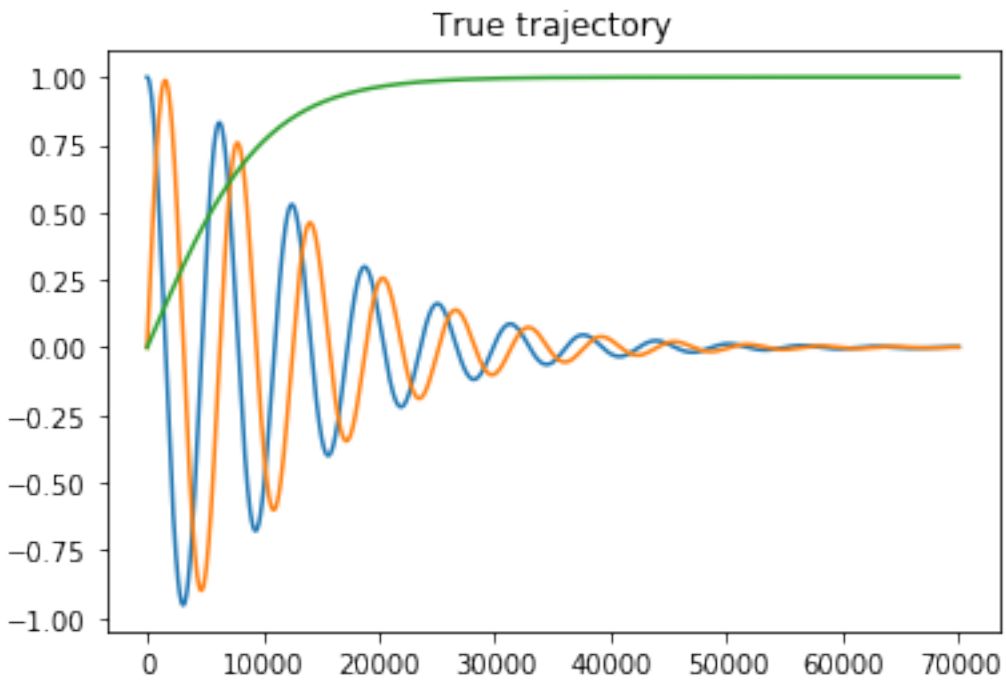
At $T = 0$ the Landau-Lifshitz-Gilbret equation for a single particle is deterministic and has a known solution. We compare the ability of the explicit and implicit methods to integrate the LLG.

```
In [10]: # Load results
         path = datapath + 'task4/'
         files = !ls {path}
         results = {name: np.fromfile(path + name).reshape((-1,3)) for name in files if name!='dt'}
         dts = np.fromfile(path + 'dt')
```

Below is an example of the true trajectory of the x,y,z coordinates of magnetisation.

```
In [11]: plt.plot(results['true'])
         plt.title('True trajectory')
```

```
Out[11]: <matplotlib.text.Text at 0x7f58a0f334e0>
```

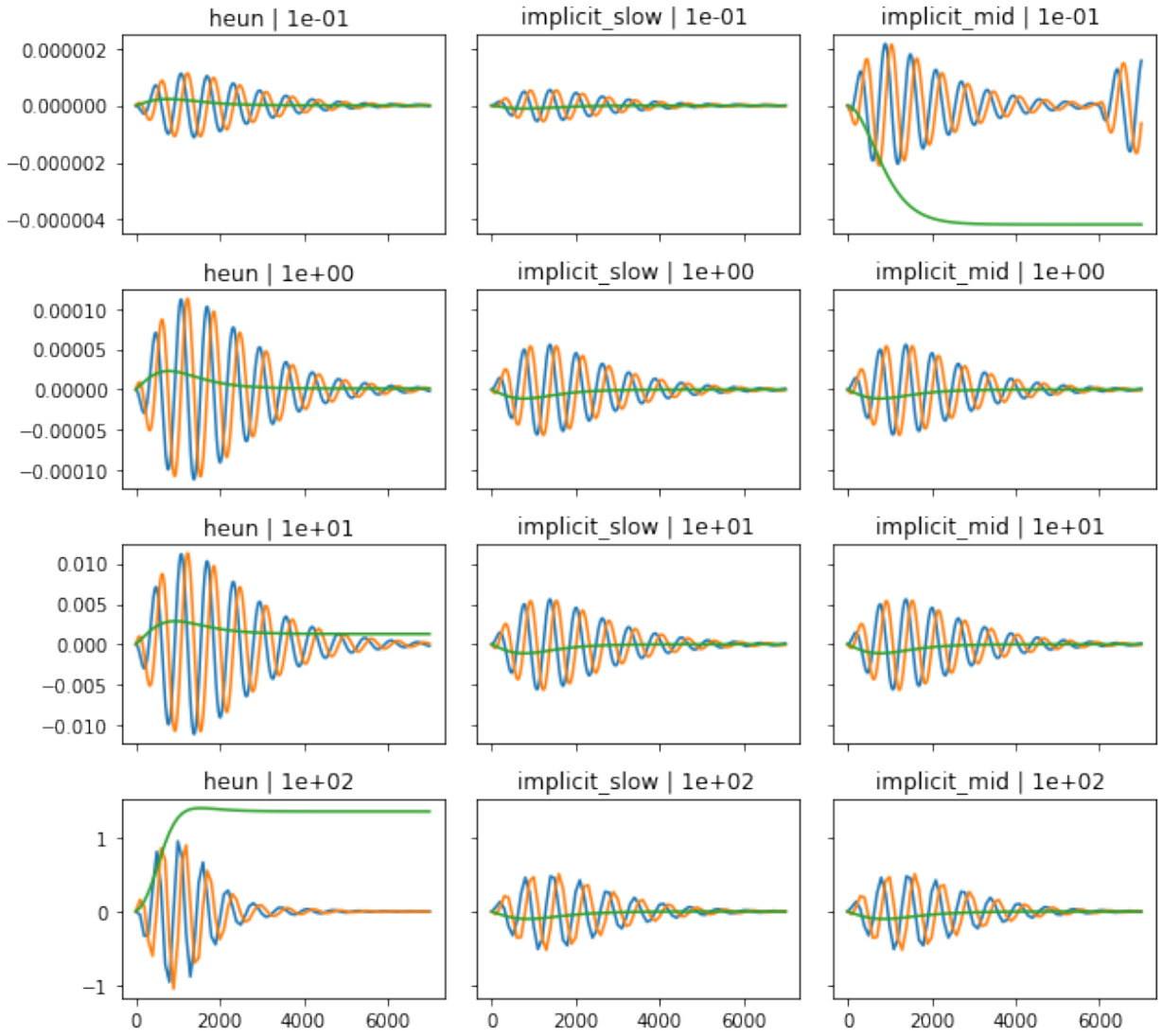


5.4.1 Residual plots

Below we compare three integrators: - Explicit (Heun) - Implicit slow (uses a high tolerance in the internal Quasi-newton solver) - Implicit mid (uses a medium tolerance in the internal Quasi-newton solver)

We compare the residuals (difference between the true and estimated trajectories) in x,y,z for a number of time step sizes. Note that we are using reduced time (see docs elsewhere).

```
In [12]: integ_names = ['heun', 'implicit_slow', 'implicit_mid']
         dt_indices = range(len(dts))

         fg, axs = plt.subplots(nrows=len(dts), ncols=len(integ_names),
                                sharey='row', sharex=True,
                                figsize=(3*len(integ_names),2*len(dts)))
         for ax_row, dt_idx in zip(axs, dt_indices):
             for ax, name in zip(ax_row, integ_names):
                 mag = results[name + str(dt_idx)]
                 true = results['true'][::10**dt_idx]
                 time = dts[dt_idx] * np.arange(mag.shape[0])
                 ax.plot(time, mag-true)
                 ax.set_title('{} | {:1.0e} '.format(name, dts[dt_idx]))
         plt.tight_layout()
```



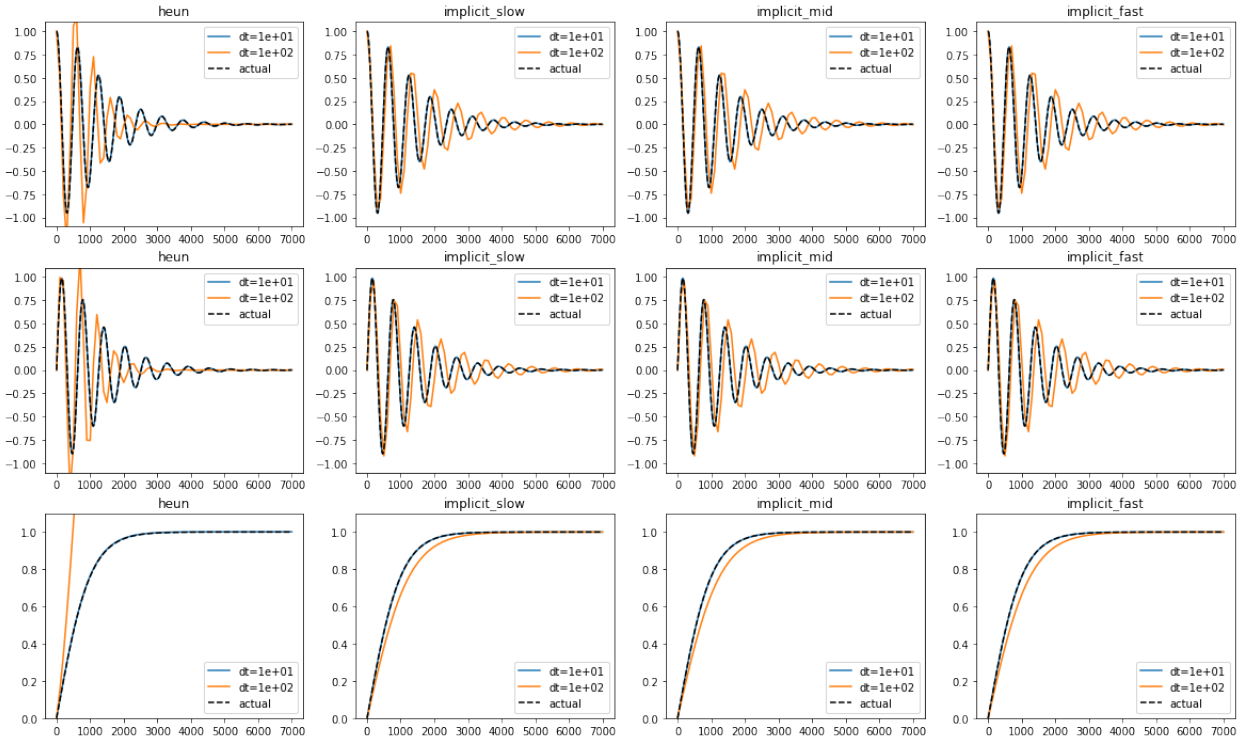
From the above results we make three observations: 1. For the same time step, the implicit scheme (slow) is more accurate than the explicit scheme 2. The implicit method is stable at larger time steps compared to the explicit scheme (see dt=1e-2) 3. Guidelines for setting the implicit quasi-Newton tolerance: - As we reduce the time step, the required tolerance on the quasi-Newton solver must be smaller. - If the tolerance is not small enough the solution blows up (see top right) - If the tolerance is small enough, decreasing the tolerance further has little effect (compare 2nd and 3rd column).

5.4.2 Compare trajectories

We can also investigate the estimated trajectories of the x,y,z components for each solver and tolerance level.

```
In [13]: intes = ['heun', 'implicit_slow', 'implicit_mid', 'implicit_fast']
         dt_indices = range(2,4)

         fg, axs = plt.subplots(nrows=3, ncols=len(intes), figsize=(5*len(intes), 12))
         for axrow, direction in zip(axs, range(3)):
             for ax, inte in zip(axrow, intes):
                 for idx in dt_indices:
                     numerical = results[inte+str(idx)][:,direction]
```

```
        time = dts[idx] * np.arange(numerical.size)
        ax.plot(time, numerical, label='dt={:1.0e}'.format(dts[idx]))
    actual = results['true'][:,direction]
    time = dts[0] * np.arange(actual.size)
    ax.plot(time, actual, 'k--', label='actual')
    ax.legend()
    ax.set_title(inte)
    ax.set_ylim(0 if direction==2 else -1.1 ,1.1)
```



5.5 Task 5 - stochastic LLG global convergence

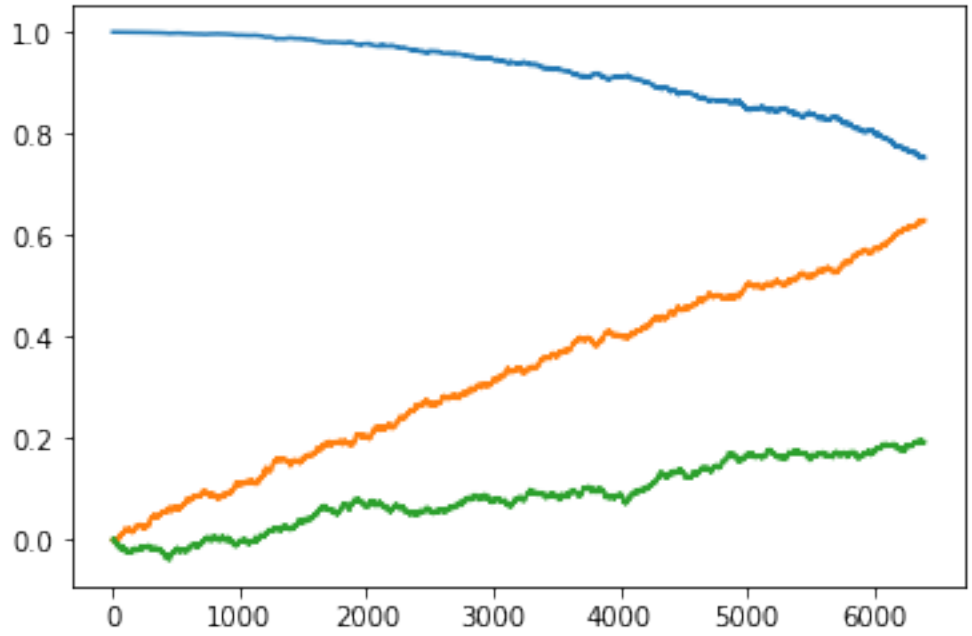
We now perform finite-temperature simulations of the LLG equation using the explicit and implicit scheme. We determine the global convergence (i.e. the increase in accuracy with decreasing time step).

We expect the convergence to be linear in `log-log` space and have a convergence rate (slope) of 0.5

Files are ordered from smallest timestep at index `0` to lowest at index `4`.

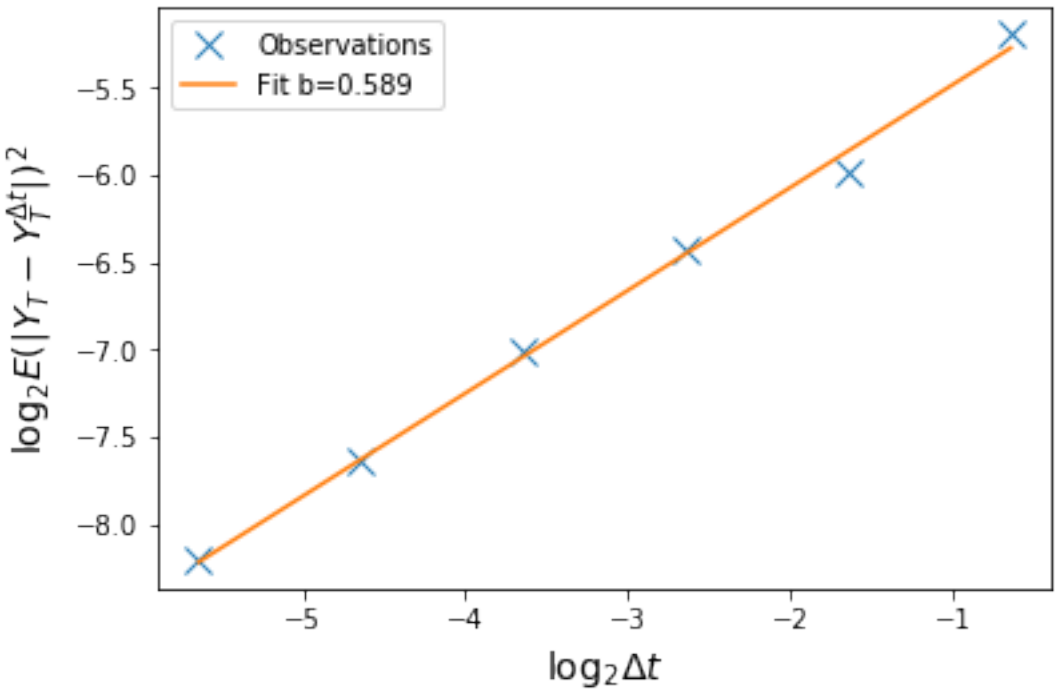
Below we show an example of the problem, which we are solving.

```
In [14]: x = np.fromfile(datapath+'task5/example_sol').reshape((-1,3))
         plt.plot(x);
```



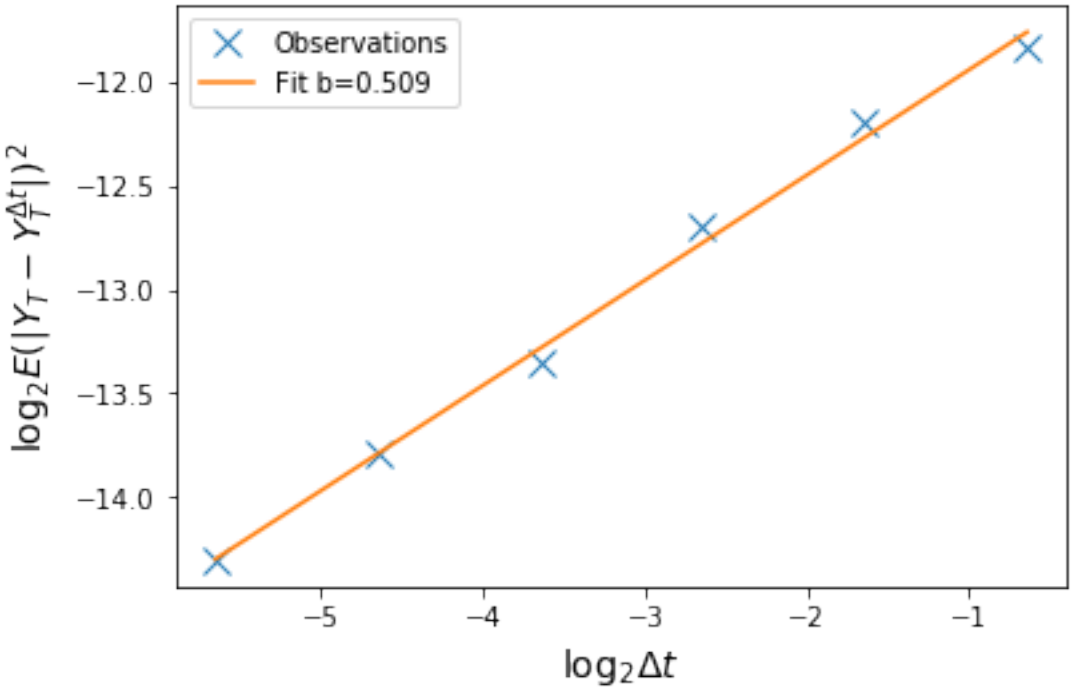
5.5.1 Implicit midpoint

```
In [17]: fnames = !ls {datapath}task5/implicit*
         global_sols = [np.fromfile(fname).reshape((-1,3)) for fname in fnames]

In [18]: # Compute difference between solutions at consecutive timesteps
         diffs = np.diff(global_sols, axis=0)

         # Take err as L2 norm
         err = np.linalg.norm(diffs, axis=2)

         # Compute expected error
         Eerr = np.mean(err, axis=1)

In [19]: # Load the dt values
         dts = np.fromfile(datapath+'task5/dt')[1:]

In [20]: # Fit a straight line
         a,b = np.linalg.lstsq(np.stack([np.ones_like(dts), np.log2(dts)]).T, np.log2(Eerr))[0]

In [21]: plt.plot(np.log2(dts), np.log2(Eerr), 'x', ms=10, label='Observations')
         plt.plot(np.log2(dts), a + np.log2(dts)*b, '-', label='Fit b={:.3f}'.format(b))
         plt.xlabel('$\\log_2 \\Delta t$', fontsize=14)
         plt.ylabel('$\\log_2 E (\\left| Y_T-Y_T^{\\Delta t}\\right|)^2$', fontsize=14)
         plt.legend()

Out[21]: <matplotlib.legend.Legend at 0x7f589d97e4a8>
```



5.5.2 Heun

```
In [22]: fnames = !ls {datapath}task5/heun*
         global_sols = [np.fromfile(fname).reshape((-1,3)) for fname in fnames]

In [23]: # Compute difference between solutions at consecutive timesteps
         diffs = np.diff(global_sols, axis=0)

         # Take err as L2 norm
         err = np.linalg.norm(diffs, axis=2)

         # Compute expected error
         Eerr = np.mean(err, axis=1)

In [24]: # Fit a straight line
         a,b = np.linalg.lstsq(np.stack([np.ones_like(dts), np.log2(dts)]).T, np.log2(Eerr))[0]

In [25]: plt.plot(np.log2(dts), np.log2(Eerr), 'x', ms=10, label='Observations')
         plt.plot(np.log2(dts), a + np.log2(dts)*b, '-', label='Fit b={:.3f}'.format(b))
         plt.xlabel('$\\log_2 \\Delta t$', fontsize=14)
         plt.ylabel('$\\log_2 E (\\left| Y_T-Y_T^{\\Delta t}\\right|)^2$', fontsize=14)
         plt.legend()

Out[25]: <matplotlib.legend.Legend at 0x7f589d9cfb70>
```



5.5.3 Results

Both methods converge correctly and have a rate of 0.5. This validates the implementation of the integrators.

CHAPTER 6

Python API Documentation

Detailed information on classes, methods, and functions in the magpy package.

6.1 magpy.core module

MagicMock is a subclass of Mock with default implementations of most of the magic methods. You can use MagicMock without having to configure the magic methods yourself.

If you use the *spec* or *spec_set* arguments then *only* magic methods that exist in the spec will be created.

Attributes and the return value of a *MagicMock* will also be *MagicMocks*.

6.2 magpy.data module

magpy.data.**grab_results**(*repo_path*, *name*)

magpy.data.**magpy_actor**()

magpy.data.**shelve_results**(*results*, *repo_path*, *name*)

6.3 magpy.geometry package

6.3.1 Submodules

magpy.geometry.arkus module

Coordinates for Arkus cluster geometries.

This module contains the coordinates of particles arranged into Arkus cluster geometries of varying size. Arkus clusters are *minimum energy* clusters that represent tightly packed clusters of spherical particles. Their definition (and name) are derived from:

N. Arkus, V. N. Manoharan, and M. P. Brenner, *Phys. Rev. Lett.* **103**, 118303 (2009). *http://dx.doi.org/10.1103/PhysRevLett.103.118303*

Clusters of 1-5 particles have only one possible arrangement. Clusters of 6 or more particles have a variety of different configurations, which are accessed using a configuration id. The available configuration ranges are:

- 1-5 particles: [0]
- 6 particles: [0-1]
- 7 particles: [0-5]
- 8 particles; [0-12]

The coordinates of the particles within the cluster are normalised to a unit distance between each pair (where possible). The distance between the particles can be controlled by multiplying by a scaling factor.

`magpy.geometry.arkus.`**`ARKUS`**
dict – a dictionary containing the geometries of Arkus clusters of 1-8 particles. The dictionary is accessed *ARKUS[n_particles][configuration_id]* and returns an *np.ndarray* of shape *(n_particles,3)* containing the coordinates of particles in the cluster.

Examples

```
>>> ARKUS[1][0]
array([[0,0,0]])
>>> ARKUS[2][0]
array([[0,0,0],
       [0,0,1]])
>>> ARKUS[2][0] * 1e-9 # apply a scaling factor
array([[0,0,0],
       [0,0,1e-9]])
```

magpy.geometry.coordinates module

`magpy.geometry.coordinates.`**`arkus_cluster_coordinates`**(*n_particles*, *configuration_id*, *R*)

Coordinates of particles in an Arkus cluster.

Returns an array of coordinates for each particle in an Arkus cluster of size *n_particles* with a specified *configuration_id*. Each configuration id represents a cluster of *n_particles* with a different arrangement.

See *magpy.geometry.arkus* for more information on Arkus geometries and available configurations.

Parameters

- **`n_particles`** (*int*) – cluster size
- **`configuration_id`** (*int*) – configuration id of the Arkus cluster see [link] for more information on available configurations for each cluster size
- **`R`** (*float*) – distance between particles

Returns shape *(n_particles, 3)* array of the coordinates

Return type np.ndarray

magpy.geometry.coordinates.**arkus_cluster_random_configuration_id**(*n_particles*)
A randomly drawn configuration id for an Arkus cluster.

Returns a random configuration id for an Arkus cluster of a specified size. Each configuration contains *n_particles* in a different arrangement.

See *magpy.geometry.arkus* for more information on Arkus geometries and available configurations.

Parameters **n_particles** (*int*) – Arkus cluster size

Returns a random configuration id

Return type int

magpy.geometry.coordinates.**arkus_random_cluster_coordinates**(*n_particles*, *R*)
Coordinates of particles in a random Arkus cluster configuration.

Returns an array of coordinates for particles in an Arkus cluster of size *n_particles* with a random configuration. Each configuration has a different arrangement of particles. See [link] for info.

See *magpy.geometry.arkus* for more information on Arkus geometries and possible configurations.

Parameters

- **n_particles** (*int*) – Arkus cluster size
- **R** (*float*) – point to point Euclidean distance between coordinates

Returns shape *(n_particles,3)* array of coordinates of particles

Return type np.ndarray

magpy.geometry.coordinates.**chain_coordinates**(*n_particles*, *R*, *direction=<sphinx.ext.autodoc._MockObject object>*)
Coordinates of particles along a straight chain.

Returns an array of coordinates for particles arranged in a perfectly straight chain and regularly spaced. The chain can have any direction and number of particles but will start at the origin.

Example

..code-block:: python

```
>>> chain_coordinates(3, 2.5, direction=[1,0,0])
np.array([0,0,0],
         [2.5,0,0],
         [5,0,0])
>>> # Only unit vector of direction is used:
>>> chain_coordinates(2, 3.0, direction=[3,4,0])
np.array([0,0,0],
         [1.8, 2.4, 0])
```

Parameters

- **n_particles** (*int*) – size of the particle chain
- **R** (*float*) – point to point Euclidean distance between points
- **direction** (*np.ndarray, optional*) – direction in 3d space to construct the chain. The magnitude of the direction is has no effect, only its unit direction. Default value is the *z*-axis *np.array([0,0,1])*

Returns shape *(n_particles,3)* array of coordinates of particles

Return type np.ndarray

6.4 magpy.initial_conditions module

magpy.initial_conditions.**random_point_on_unit_sphere**()
Point randomly drawn from a uniform distribution on the unit sphere.

Randomly picking a point on the unit sphere requires choosing the point so that each solid angle on the sphere is equally likely.

Set *np.random.seed(seed)* for reproducible results.

Returns a 3d array with the x,y,z coordinates of a random point

Return type np.ndarray

magpy.initial_conditions.**random_quaternion**()
Random uniformly distributed quaternion

A quaternion defines an 3d axis of rotation and a corresponding angle of rotation. A random quaternion is drawn by first drawing a random rotation axis from the uniform random sphere and a uniform angle. Random quaternions are used for random rotation operations on arbitrary geometries.

Returns a random quaternion from a uniform distribution

Return type transforms3d.Quaternion

magpy.initial_conditions.**uniform_random_axes**(*N*)
Random 3d vectors (axes) uniformly distributed on unit sphere.

Returns *N* 3d unit vectors drawn from a uniform distribution over the unit sphere.

Set *np.random.seed(seed)* for reproducible results.

Parameters **N** (*int*) – number of random axes to draw.

Returns a 2d array size (*Nx3*). Random 3d unit vectors on the uniform random sphere.

Return type np.ndarray

6.5 magpy.model module

class magpy.model.**DOModel**(*radius*, *anisotropy*, *initial_probabilities*, *magnetisation*, *damping*, *temperature*, *field_shape='constant'*, *field_frequency=0.0*, *field_amplitude=0.0*)
Bases: object

A probabilistic model of a single particle

A magpy DOModel is a probabilitic model of a single magnetic nanoparticle with a uniaxial anisotropy axis. The model has just two possible states: up and down. The model is defined by the material properties of the particle, an external field (applied along the anisotropy axis), and an initial probability vector (length 2).

The model is simulated to solve the probability of the system being up and down over time. The particle is optionally subjected to a time-varying field along the anisotropy axis. The field can be constant or sine/square varying with a desired frequency and amplitude.

Parameters

- **radius** (*double*) – radius of the spherical particle
- **anisotropy** (*double*) – anisotropy constant for the uniaxial anisotropy axis
- **initial_probabilities** (*ndarray[double,2]*) – initial probability of the particle down and up state respectively.
- **magnetisation** (*double*) – saturation magnetisation of all particles in the cluster (ampres / meter). Saturation magnetisation cannot vary between particles.
- **damping** (*double*) – the damping constant for the particle.
- **temperature** (*double*) – the ambient temperature in Kelvin for the particle.
- **field_shape** (*str, optional*) – can be either 'constant', 'square' or 'sine' describing the time-varying shape of the alternating field. The field is always applied along the anisotropy axis. Default is 'constant'
- **field_frequency** (*double, optional*) – the frequency of the applied field in Hz. Default is 0Hz.
- **field_amplitude** (*double, optional*) – the amplitude of the applied field in Ampres / meter. Default is 0A/m

simulate(*end_time*, *time_step*, *max_samples*)

Simulate the state probabilities for the particle

Simulate the time-varying probabilities of the up/down states of the particle. The system is described by a master equation, which is defined by the transition rates between the up and down state. The master equation is solved numerically using an explicit RK45 solver.

In order to save memory, the user is required to specify the *max_samples*. The output of the time-integrator is up/downsampled to *max_samples* regularly spaced intervals using a first-order-hold interpolation. This is useful for long simulations with very small time steps that, without downsampling, would produce GBs of data very quickly.

Parameters

- **end_time** (*float*) – time to end the simulation (in seconds)
- **time_step** (*float*) – time step for time-integration solver
- **max_samples** (*int*) – number of regularly spaced samples of the output

Returns

a ***magpy.results.Results*** **object containing** the time-dependent magnetisation of the particle.

Return type magpy.Results

class magpy.model.**EnsembleModel**(*N*, *base_model*, ***kwargs*)

Bases: object

Ensemble of particle clusters

The EnsembleModel class represents a non-interacting ensemble of particle clusters. It aims to provide a more user-friendly alternative than handling a large number of magpy.Model instances manually.

Every member of the ensemble is copied from a base *magpy.Model* and is updated from a list of varying parameters. Parameters that are not specified as keyword arguments will be identical for every member of the ensemble and equivalent to that parameter's value in the base model.

Parameters that should vary for each particle are specified as a keyword argument whose value is a list of parameters of length *N* (where the i'th value of the list correpsonds to the parameter's value for the i'th member of the cluster)

Parameters

- **N** (*int*) – number of clusters in the ensemble
- **base_model** (*magpy.Model*) – the base model provides the default parameters for every member of the ensemble.
- ****kwargs** – each argument may be a magpy.Model parameter and a corresponding list of *N* parameter values, which override the base model parameters for the i'th member of the ensemble.

simulate(*end_time*, *time_step*, *max_samples*, *random_state*, *renorm=False*, *interactions=True*, *n_jobs=1*, *implicit_solve=True*, *implicit_tol=1e-09*)

Simulate the dynamics of an ensemble of particle clusters

Simulate the time-varying dynamics of an ensemble of particle clusters of interacting macrospins. The time-varying dynamics are described by the Landau-Lifshitz-Gilbert stochastic differential equation, which is integrated using an explicit or implicit numerical scheme.

In order to save memory, the user is required to specify the *max_samples*. The output of the time-integrator is up/downsampled to *max_samples* regularly spaced intervals using a first-order-hold interpolation. This is useful for long simulations with very small time steps that, without downsampling, would produce GBs of data very quickly.

There are two time-integration schemes available:

- a fully implicit midpoint scheme **:cpp:function:'integrator::implicit_midpoint'**
- an explicit predictor-corrector method (Heun scheme) **:cpp:function:'integrator::heun'**

Parameters

- **end_time** (*float*) – time to end the simulation (in seconds)
- **time_step** (*float*) – time step for time-integration solver
- **max_samples** (*int*) – number of regularly spaced samples of the output
- **random_state** (*int, optional*) – the state is used to generate seeds for each of the individual simulations. Set for reproducible results.
- **renorm** (*bool, optional*) – default is False. If True the magnetisation of each particle is rescaled (using the 2-norm) to unity at every time step.
- **interactions** (*bool, optional*) – default is True. If False the interactions between particles are switched off.
- **implicit_solve** (*bool, optional*) – default is True. If True a fully-implicit stochastic solver is used. If False the explicit Heun scheme is used.
- **implicit_tol** (*float, optional*) – if using the implicit solver *implicit_tol* sets the tolerance of the internal Newton-Raphson method. Default is 1e-9

Returns

a *magpy.results.Results* object containing the time-dependent magnetisation of the particle system.

Return type magpy.Results

class magpy.model.**Model**(*radius*, *anisotropy*, *anisotropy_axis*, *magnetisation_direction*, *location*, *magnetisation*, *damping*, *temperature*, *field_shape='constant'*, *field_frequency=0.0*, *field_amplitude=0.0*)

Bases: object

A cluster of interacting magnetic nanoparticles

A magpy Model describes a cluster of magnetic nanoparticles. The cluster is defined by the material properties of the individual particles, their relative positions in 3D space, and an initial magnetisation vector. Each particle is modelled as by a single macrospin with a uniaxial anisotropy axis.

The model may be simulated to obtain the time-dependent dynamics of the magnetisation vector of the individual particles within the cluster.

The particle cluster is optionally subjected to a time-varying magnetic field, applied along the z-axis. The field can be constant or sine/square varying with a desired frequency and amplitude.

Parameters

- **radius** (*list of double*) – radius of each spherical particle in the ensemble in meters
- **anisotropy** (*list of double*) – anisotropy constant for the uniaxial anisotropy axis of each particle
- **anisotropy_axis** (*list of ndarray[double,3]*) – unit axis for the direction of the anisotropy for each particle
- **magnetisation_direction** (*list of ndarray[double,3]*) – initial direction of the magnetisation vector for each particle
- **location** (*list of ndarray[double,3]*) – location of each particle in the cluster described by x,y,z coordinates
- **magnetisation** (*double*) – saturation magnetisation of all particles in the cluster (ampres / meter). Saturation magnetisation cannot vary between particles.
- **damping** (*double*) – the damping parameter for all particles in the cluster. Damping cannot vary between particles.
- **temperature** (*double*) – the ambient temperature in Kelvin for the particle cluster.
- **field_shape** (*str, optional*) – can be either 'constant', 'square' or 'sine' describing the time-varying shape of the alternating field. The field is always applied along the z-axis. Default is 'constant'
- **field_frequency** (*double, optional*) – the frequency of the applied field in Hz. Default is 0Hz.
- **field_amplitude** (*double, optional*) – the amplitude of the applied field in Ampres / meter. Default is 0A/m

simulate(*end_time*, *time_step*, *max_samples*, *seed=1001*, *renorm=False*, *interactions=True*, *implicit_solve=True*, *implicit_tol=1e-09*)

Simulate the dynamics of the particle cluster

Simulate the time-varying dynamics of the cluster of interacting macrospins. The time-varying dynamics are described by the Landau-Lifshitz-Gilbert stochastic differential equation, which is integrated using an explicit or implicit numerical scheme.

In order to save memory, the user is required to specify the *max_samples*. The output of the time-integrator is up/downsampled to *max_samples* regularly spaced intervals using a first-order-hold interpolation. This is useful for long simulations with very small time steps that, without downsampling, would produce GBs of data very quickly.

There are two time-integration schemes available:

- a fully implicit midpoint scheme **:cpp:function:‘integrator::implicit_midpoint‘**
- an explicit predictor-corrector method (Heun scheme) **:cpp:function:‘integrator::heun‘**

Parameters

- **end_time** (*float*) – time to end the simulation (in seconds)
- **time_step** (*float*) – time step for time-integration solver
- **max_samples** (*int*) – number of regularly spaced samples of the output
- **seed** (*int, optional*) – default value is 1001. The random seed for randam number generation of the thermal noise. Set for reproducible results.
- **renorm** (*bool, optional*) – default is False. If True the magnetisation of each particle is rescaled (using the 2-norm) to unity at every time step.
- **interactions** (*bool, optional*) – default is True. If False the interactions between particles are switched off.
- **implicit_solve** (*bool, optional*) – default is True. If True a fully-implicit stochastic solver is used. If False the explicit Heun scheme is used.
- **implicit_tol** (*float, optional*) – if using the implicit solver *implicit_tol* sets the tolerance of the internal Newton-Raphson method. Default is 1e-9

Returns

a *magpy.results.Results* object containing the time-dependent magnetisation of the particle system.

Return type magpy.Results

6.6 magpy.results module

class magpy.results.**EnsembleResults**(*results*)

Bases: object

Results from a simulation of an ensemble of particle clusters

The EnsembleResults object holds the resulting *magpy.Results* objects for an ensemble of simulated particle clusters. It provides a user-friendly alternative to handling a large collection of *magpy.Results* instances and implemetns methods for computing ensemble-wide properties.

Parameters **results** (*list[magpy.Results]*) – results for each particle cluster in the ensemble

time

(np.ndarray): 1d array of length *M*. Time in seconds for each sample in the ensemble results.

field

(np.ndarray): 1d array of length *M*. Field amplitude at each point in time. Field is always applied along the z-axis.

energy_dissipated(*start_time=None*, *end_time=None*)

Total energy dissipated by the ensemble.

A simulation with a constant or zero applied field will dissipate no energy. The energy dissipated by an ensemble of magnetic particle clusters subjected to an alternating field is the area of the hysteresis loop (magnetisation-field plane).

The energy dissipated may be computed for the entire simulation or within a specific time window, defined by *start_time* and *end_time*

Parameters

- **start_time** (*double, optional*) – the start of the time window for computing energy dissipated. Default value *None* uses the start of the simulation.
- **end_time** (*double, optional*) – the end of the time window for computing energy dissipated. Default value *None* uses the end of the simulation.

Returns total energy dissipated by the ensemble during the time window

Return type double

ensemble_magnetisation(*direction='z'*)

Total magnetisation of entire ensemble

The total magnetisation of an ensemble of particle clusters. The ensemble magnetisation is the average value of the magnetisation of each particle particle cluster in the ensemble at each point in time. The component (*x*,'y','z') along which the magnetisation may be specified. The default value is *z*, which is the same direction as the applied magnetic field.

Parameters **direction** (*str, optional*) – direction of magnetisation *x*, *y* or *z*. Default value is *z*.

Returns

1d array of length *M* containing the ensemble magnetisation for each point in *self.time*

Return type np.ndarray

final_cycle_energy_dissipated(*field_frequency*)

Energy dissipated by the final cycle of the magnetic field.

A simulation with a constant or zero applied field will dissipate no energy. The energy dissipated by an ensemble of magnetic particle clusters subjected to an alternating field is the area of the hysteresis loop (magnetisation-field plane).

Use this function to compute the energy dissipated by the final cycle (i.e. period) of the applied alternating magnetic field if the total simulation time contains multiple cycles of the field (i.e. is longer than the period of the applied field). A common use case for this is to simulate a large number field cycles to reach equilibrium and then compute the energy dissipated during a single cycle of the field in equilibrium.

Parameters **field_frequency** (*double*) – the frequency of the applied magnetic field

Returns energy dissipated during the last cycle of the applied magnetic field.

Return type double

final_state()

State of each ensemble member at the end of the simulation.

The final state of each particle cluster in the ensemble at the end of the simulation time. The state of each particle cluster is the value of magnetisation vector of every particle in the cluster.

Returns a list of nested dictionaries like *{'x': {0: m_x, ..., N-1: m_x}, 'y': ...}*. The dictionaries contain the final value of the magnetisation vector for each of the *N* particles in the cluster.

Return type list[dict]

magnetisation(*direction='z'*)
Total magnetisation of each member of the ensemble

The total magnetisation of cluster is computed by summing the components of the magnetisation vector for each particle in the cluster. The component (*x*,'y','z') along which the magnetisation may be specified. The default value is *z*, which is the same direction as the applied magnetic field.

Parameters **direction** (*str, optional*) – direction of magnetisation *x*, *y* or *z*. Default value is *z*.

Returns list containing a length *M* 1d array containing the total magnetisation of each particle cluster in the ensemble.

Return type list[np.ndarray]

class magpy.results.**Results**(*time*, *field*, *x*, *y*, *z*, *N*)
Bases: object

Results of a simulation of a single particle cluster

The results contain the time-varying magnetisation and field resulting from stochastic simulation of a particle cluster consisting of *N* particles.

Parameters

- **time** (*np.ndarray*) – 1d array of length *M*. Time in seconds for each sample in the results
- **field** (*np.ndarray*) – 1d array of length *M*. Field amplitude at each point in time. Field is always applied along the z-axis.
- **x** (*dict*) – *{0: np.ndarray, ..., N-1: np.ndarray}* key, value pair is an interger particle id and a corresponding 1d array of length *M* for each of the *N* particles in the cluster. 1d array is the x-coordinate of the particle magnetisation vector at each point in time.
- **y** (*dict*) – *{0: np.ndarray, ..., N-1: np.ndarray}* key, value pair is an interger particle id and a corresponding 1d array of length *M* for each of the *N* particles in the cluster. 1d array is the y-coordinate of the particle magnetisation vector at each point in time.
- **z** (*dict*) – *{0: np.ndarray, ..., N-1: np.ndarray}* key, value pair is an interger particle id and a corresponding 1d array of length *M* for each of the *N* particles in the cluster. 1d array is the z-coordinate of the particle magnetisation vector at each point in time.
- **N** (*int*) – number of particles in the ensemble

final_state()
The state of the cluster at the end of the simulation.

Returns the state of the particle cluster at the end of the simulation time.

Returns

a nested dictionary ***{'x': {0: m_x, ..., N-1: m_x}, 'y': ...}*** containing the final value of the magnetisation vector for each of the *N* particles in the cluster.

Return type dict

magnetisation(*direction='z'*)
Computes the total magnetisation of the cluster

Computes the total time-varying magnetisation of the cluster in a desired direction. The total magnetisation is simply the sum of the individual magnetisation vector components in the specified direction (x,y, or z).

Parameters **direction** (*str, optional*) – the direction of magnetisation *x*, *y* or *z*. Default value is *z*.

Returns

1d array of length *M* the total magnetisation at each point in *self.time*.

Return type np.ndarray

plot()

Plots the magnetisation from the results

Plots the x,y,z coordinates of the magnetisation vector for every particle in the particle cluster.

Returns matplotlib figure handle containing the resulting plot axes.

CHAPTER 7

C++ API Documentation

The magpy C++ library powers the core functionality of the python interface. The library doesn't depend on the python features and is compiled independently of the python code. The C++ API is organised around functions in namespaces and uses classes sparingly. Use the namespace list for browsing the code base.

7.1 Namespace list

7.1.1 Namespace constants

namespace constants

Variables

const double **KB** =1.38064852e-23

const double **MU0** =1.25663706e-6

const double **GYROMAG** =1.76086e11

7.1.2 Namespace curry

Warning: doxygennamespace: Cannot find namespace "curry" in doxygen xml output for project "magpy-api" from directory: ../doxygen/doxyxml

7.1.3 Namespace distances

namespace distances

Compute distances between points.

Functions for computing distances between points in 3D space.

Author Oliver Laslett

Date 2017

Functions

std::vector<std::vector<std::array<double, 3>>> **pair_wise_distance_vectors**(std::vector<std::array<double, 3>> *points*)

Computes the distance vector between points in 3d space.

Given a list of x,y,z coordinates, compute the distance vector between every pair of x,y,z coordinates in the list.

Return NxN std::vector matrix of distances between each pair of points

Parameters

- `points`: length N vector of arrays of x,y,z coordinates

std::vector<std::vector<double>> **pair_wise_distance_magnitude**(std::vector<std::array<double, 3>> *points*)

Computes the Euclidean distance between pairs of points.

std::vector<std::vector<std::array<double, 3>>> **pair_wise_distance_unit_vectors**(std::vector<std::array<double, 3>> *points*)

Computes the unit distance vector between a list of points.

std::vector<std::vector<double>> **pair_wise_distance_magnitude**(std::vector<std::vector<std::array<double, 3>>> *distance_vectors*)

Computes the Euclidean distance from a matrix of distance vectors.

Given a matrix of distance vectors, compute the 2norm of the vector corresponding to the Euclidean distance.

Return NxN matrix of Euclidean distance values

Parameters

- `distance_vectors`: NxN symmetrical matrix of distance vectors (x,y,z coordinates)

std::vector<std::vector<std::array<double, 3>>> **pair_wise_distance_unit_vectors**(std::vector<std::vector<std::array<double, 3>>> *distance_vectors*, std::vector<std::vector<double>> *distance_magnitudes*)

Computes the unit distance vectors from distance vectors and their magnitudes.

7.1.4 Namespace dom

namespace dom

discrete orientation model for magnetic particles

Functions

void **transition_matrix**(double *W, **const** double *k*, **const** double *v*, **const** double *T*, **const** double *h*, **const** double *ms*, **const** double *alpha*)

Compute the 2x2 transition matrix for a single particle.

Assumes uniaxial anisotropy and an applied field h<1 Only valid for large energy barriers: $(1-h)^2>>1$

Parameters

- `W`: transition matrix [2x2]
- `k`: anisotropy strength constant for the uniaxial anisotropy
- `v`: volume of the particle in meter^3
- `T`: temperature of environment in Kelvin
- `h`: dimensionless applied field - normalised by $H_k = \frac{2K}{\mu_0 M_s}$
- `ms`: saturation magnetisation
- `alpha`: dimensionless damping constant

void **master_equation_with_update**(double **derivs*, double **work*, **const** double *k*, **const** double *v*, **const** double *T*, **const** double *ms*, **const** double *alpha*, **const** double *t*, **const** double **state_probabilities*, **const** std::function<double) double > *applied_field*Computes master equation for particle in time-dependent field.

Computes the applied field in the z-direction at time t. Uses the field value to compute the transition matrix and corresponding master equation derivatives for the single particle.

Parameters

- `derivs`: master equation derivatives [length 2]
- `work`: vector [length 4]
- `k`: anisotropy strength constant for the uniaxial anisotropy
- `v`: volume of the particle in meter^3
- `T`: temperature of environment in Kelvin
- `ms`: saturation magnetisation
- `alpha`: dimensionless damping constant
- `t`: time at which to evaluate the external field
- `state_probabilities`: the current state probabilities for each of the 2 states (up and down) [length 2]
- `applied_field`: a scalar function takes a double and returns a double. Given the current time should return normalised field $h = (t)$ where the field is normalised by H_k

double **single_transition_energy**(double *K*, double *V*, double *h*)

Compute the energy dissipated during one transition of the particle.

Return the energy dissipated in one transition

Parameters

- `K`: anisotropy constant of the particle

- V: volume of the particle
- h: applied field on the particle, reduced by the anisotropy field $H_k = \frac{2K}{\mu_0 M_s}$

7.1.5 Namespace driver

namespace driver

High level interface to differential equation integrators.

Drivers wrap the single step integrators and provide an interface for simulating differential equations for multiple steps from the initial condition. Drivers also handle memory management of the work arrays needed by the integrators.

Author Oliver Laslett

Date 2017

Functions

void **rk4**(double **states*, **const** double **initial_state*, **const** std::function<void(double *, **const** double *, **const** double)> *derivs*, **const** size_t *n_steps*, **const** size_t *n_dims*, **const** double *step_size*)

void **eulerm**(double **states*, **const** double **initial_state*, **const** double **wiener_process*, **const** std::function<void(double *, **const** double *, **const** double)> *drift*, **const** std::function<void(double *, **const** double *, **const** double)> *diffusion*, **const** size_t *n_steps*, **const** size_t *n_dims*, **const** size_t *n_wiener*, **const** double *step_size*)

void **heun**(double **states*, **const** double **initial_state*, **const** double **wiener_process*, **const** std::function<void(double *, double *, **const** double *, **const** double)> *sde*, **const** size_t *n_steps*, **const** size_t *n_dims*, **const** size_t *n_wiener*, **const** double *step_size*)

void **implicit_midpoint**(double **x*, **const** double **x0*, **const** double **dw*, **const** std::function<void(double *, double *, double *, double *, **const** double *, **const** double, **const** double)> *sde*, **const** size_t *n_dim*, **const** size_t *w_dim*, **const** size_t *n_steps*, **const** double *t0*, **const** double *dt*, **const** double *eps*, **const** size_t *max_iter*)

7.1.6 Namespace energy

namespace energy

Functions

double **anisotropy_r**()

double **zeeman_r**()

double **dipolar_r**()

7.1.7 Namespace field

namespace field

Contains functions for computing magnetic fields.

Functions for computing effective fields from anistoropy and a range of time-varying applied fields.

Author Oliver Laslett

Date 2017

Enums

enum options

Values:

SINE

SQUARE

CONSTANT

Functions

template <typename... *T*>
std::function<double (double) > **bind_field_function**

std::function<doubledouble, T...> *func*, T... *bind_args*Bind field parameters into a field function.

Field functions are of the form `std::function(double<double>)` This function allows additional arguments to be bound to functions and returns the function as a field function.

Return field function

Parameters

- `func`: function to bind
- `bind_args`: arguments to bind

double **constant** (**const** double *t*, **const** double *h*)

A constant applied field.

A simple placeholder function representing a constant field. Always returns the same value.

Return the constant field amplitude at all values of time

Parameters

- `t`: time (parameter has no effect)
- `h`: applied field amplitude

double **sinusoidal** (**const** double *t*, **const** double *h*, **const** double *f*)

A sinusoidal alternating applied field.

Returns the value of a sinusoidally varying field at any given time.

Return the value of the varying applied field at time `t`

Parameters

- `t`: time

- h: applied field amplitude
- f: applied field frequency

double **square**(**const** double *t*, **const** double *h*, **const** double *f*)

A square wave switching applied field.

An alternating applied field with a square shape centred around zero. i.e. it alternates between values −h and h

Return the value of the square wave applied field at time t

Parameters

- t: time
- h: applied field amplitude
- f: applied field frequency

double **square_fourier**(**const** double *t*, **const** double *h*, **const** double *f*, **const** size_t *n_compononents*)

A square wave applied field of finite Fourier components.

An approximate square wave is computed from a finite number of Fourier components. The square wave alternates between −h and h.

Return the value of the square wave applied field at time t

Parameters

- t: time
- h: applied field amplitude
- f: applied field frequency
- n_components: number of Fourier components to compute

void **multi_add_applied_Z_field_function**(double **heff*, **const** std::function<double)**const** double > *&hfunc*, **const** double *t*, **const** size_t *N*Add the applied field in the z direction.

void **uniaxial_anisotropy**(double **h_anis*, **const** double **magnetisation*, **const** double **anis_axis*)

Effective field contribution from uniaxial anisotropy.

The effective field experienced by a single particle with a uniaxial anisotropy.

Parameters

- h_anis: effective field [length 3]
- mag: the magnetisation of the particle of [length 3]
- axis: the anisotropy axis of the particle [length 3]

void **multi_add_uniaxial_anisotropy**(double **h*, **const** double **states*, **const** double **axes*, **const** double **k_reduced*, **const** size_t *N*)

Add the uniaxial anisotropy term term to the field of N particles.

Parameters

- h: effective field incremented with the anisotropy term
- states: magnetic state of the particles (length 3N)

- `axes`: anisotropy axis of each particle (length 3N)
- `k_reduced`: the reduced anisotropy constant for each particle (length N)
- `N`: number of particles

void **uniaxial_anisotropy_jacobian**(double **jac*, **const** double **axis*)

Jacobian of the uniaxial anisotropy effective field term.

The Jacobian of a particle's uniaxial anisotropy with respect to it's magnetisation value. $J_h(m) = \frac{\partial h(m)}{\partial m}$

Parameters

- `jac`: the jacobian of the effective field [length 3x3]
- `axis`: the anisotropy axis of the particle [length 3]

void **multi_add_uniaxial_anisotropy_jacobian**(double **jac*, **const** double **axes*, **const** double **anis*, **const** size_t *N*)

Jacobian of the uniaxial anisotropy effective field term for many particles.

The Jacobian of a particle's uniaxial anisotropy with respect to it's magnetisation value. $J_h(m) = \frac{\partial h(m)}{\partial m}$

Parameters

- `jac`: the jacobian of the effective field [length 3Nx3N]
- `axis`: the anisotropy axis of the particle [length 3N]
- `N`: number of particles

void **multi_add_dipolar**(double **field*, **const** double *ms*, **const** double *k_av*, **const** double **v_reduced*, **const** double **mag*, **const** double **dists*, **const** double **dist_cubes*, **const** size_t *N*)

field is Nx3 and is the effective field on each particle ms is the same for all particles k_av is the average anisotropy constant for all particles v_reduced is the reduced volume of each particle v_av is the average volume for all particles mag is Nx3 long and is the magnetic state of each of the N particles dists is NxNx3 long and is the distance between each pair of particles N is the number of particles dist_cubes is NxN and is the reduced cubed distance between each pair

double **dipolar_prefactor**(**const** double *ms*, **const** double *k_av*)

Prefactor term (helper for field::dipolar)

prefactor is $\frac{\mu 0 M_s^2}{8\pi K}$

Return prefactor term

Parameters

- `ms`: saturation magnetisation
- `k_av`: average anisotropy constant for system

void **dipolar_add_p2p_term**(double **out*, **const** double *vj*, **const** double *rij3*, **const** double **mj*, **const** double **dist*, **const** double *prefactor*)

void **zero_all_field_terms**(double **h*, **const** size_t *N*)

Set all N values of the field to zero.

Parameters

- `h`: pointer to array of N doubles will all be set to zero
- `N`: number of elements in array

7.1.8 Namespace integrator

namespace integrator

Numerical methods for differential equations.

Numerical methods for simulating the time evolution of deterministic and stochastic ordinary nonlinear differential equations. All integrators compute a single step of the solution.

Author Oliver Laslett

Date 2017

Functions

void **rk4**(double *next_state, double *k1, double *k2, double *k3, double *k4, **const** double *current_state, **const** std::function<void) double *, **const** double *, **const** double > *derivs*, **const** size_t *n_dims*, **const** double *t*, **const** double *h*

void **heun**(double *next_state, double *drift_arr, double *trial_drift_arr, double *diffusion_matrix, double *trial_diffusion_matrix, **const** double *current_state, **const** double *wiener_steps, **const** std::function<void) double *, double *, **const** double *, **const** double > *sde*, **const** size_t *n_dims*, **const** size_t *wiener_dims*, **const** double *t*, **const** double *step_size*

void **eulerm**(double *states, double *diffusion_matrix, **const** double *initial_state, **const** double *wiener_process, **const** std::function<void) double *, **const** double *, **const** double > *drift*, **const** std::function<voiddouble *, **const** double *, **const** double> *diffusion*, **const** size_t *n_dims*, **const** size_t *n_wiener*, **const** double *t*, **const** double *step_size*

template <**class** CSTATES, **class** CDIFF>
void **milstein**(CSTATES &*next_state*, **const** CSTATES &*current_state*, **const** CSTATES &*drift*, **const** CDIFF &*diffusion*, **const** CSTATES &*wiener_increments*, **const** double *step_size*)

int **implicit_midpoint**(double *x, double *dwm, double *a_work, double *b_work, double *adash_work, double *bdash_work, double *x_guess, double *x_opt_tmp, double *x_opt_jac, lapack_int *x_opt_ipiv, **const** double *x0, **const** double *dw, **const** std::function<void) double *, double *, double *, double *, **const** double *, **const** double, **const** double > *sde*, **const** size_t *n_dim*, **const** size_t *w_dim*, **const** double *t*, **const** double *dt*, **const** double *eps*, **const** size_t *max_iter*

void **rk45**(double *next_state, double *temp_state, double *k1, double *k2, double *k3, double *k4, double *k5, double *k6, double *h_ptr, double *t_ptr, **const** double *current_state, **const** std::function<void) double *, **const** double *, **const** double > *ode*, **const** size_t *n_dims*, **const** double *eps*RK45 Cash-Karp adaptive step deterministic ODE solver.

Solves ODEs :) returns the next state and the time step to be used in the next instance what about the actual time step that it used? That's also important!

namespace ck_butcher_table

Cash-Karp parameter table for RK45.

Variables

constexpr double **c11** = 0.2

constexpr double **c21** = 3.0/40.0

constexpr double **c22** = 9.0/40.0

constexpr double **c31** = 3.0/10.0

constexpr double **c32** = -9.0/10.0

constexpr double **c33** = 6.0/5.0

constexpr double **c41** = -11.0/54.0

constexpr double **c42** = 2.5

constexpr double **c43** = -70.0/27.0

constexpr double **c44** = 35.0/27.0

constexpr double **c51** = 1631.0/55296.0

constexpr double **c52** = 175.0/512.0

constexpr double **c53** = 575.0/13824.0

constexpr double **c54** = 44275.0/110592.0

constexpr double **c55** = 253.0/4096.0

constexpr double **hc1** = 0.2

constexpr double **hc2** = 0.3

constexpr double **hc3** = 0.6

constexpr double **hc4** = 1.0

constexpr double **hc5** = 7.0/8.0

constexpr double **x11** = 37.0/378.0

constexpr double **x13** = 250.0/621.0

constexpr double **x14** = 125.0/594.0

constexpr double **x16** = 512.0/1771.0

constexpr double **x21** = 2825.0/27648.0

constexpr double **x23** = 18575.0/48384.0

constexpr double **x24** = 13525.0/55296.0

constexpr double **x25** = 277.0/14336.0

constexpr double **x26** = 0.25

7.1.9 Namespace integrator::ck_butcher_table

namespace integrator::**ck_butcher_table**

Cash-Karp parameter table for RK45.

Variables

constexpr double **c11** = 0.2

constexpr double **c21** = 3.0/40.0

constexpr double **c22** = 9.0/40.0

constexpr double **c31** = 3.0/10.0

constexpr double **c32** = -9.0/10.0

constexpr double **c33** = 6.0/5.0

constexpr double **c41** = -11.0/54.0

constexpr double **c42** = 2.5

constexpr double **c43** = -70.0/27.0

constexpr double **c44** = 35.0/27.0

constexpr double **c51** = 1631.0/55296.0

constexpr double **c52** = 175.0/512.0

constexpr double **c53** = 575.0/13824.0

constexpr double **c54** = 44275.0/110592.0

constexpr double **c55** = 253.0/4096.0

constexpr double **hc1** = 0.2

constexpr double **hc2** = 0.3

constexpr double **hc3** = 0.6

constexpr double **hc4** = 1.0

constexpr double **hc5** = 7.0/8.0

constexpr double **x11** = 37.0/378.0

constexpr double **x13** = 250.0/621.0

constexpr double **x14** = 125.0/594.0

constexpr double **x16** = 512.0/1771.0

constexpr double **x21** = 2825.0/27648.0

constexpr double **x23** = 18575.0/48384.0

constexpr double **x24** = 13525.0/55296.0

constexpr double **x25** = 277.0/14336.0

constexpr double **x26** = 0.25

7.1.10 Namespace io

namespace io

Functions

template <**typename** T>
int **write_array**(**const** std::string *fname*, T **const** **arr*, **const** size_t *len*)

7.1.11 Namespace llg

namespace llg

Functions for evaluating the Landau-Lifshitz-Gilbert equation.

Includes the basic equation as well as Jacobians and combined functions to update fields during integration.

Author Oliver Laslett

Date 2017

Functions

void **drift**(double **deriv*, **const** double **state*, **const** double *time*, **const** double *alpha*, **const** double **heff*)

Deterministic drift component of the stochastic LLG.

Parameters

- `deriv`: drift derivative of the deterministic part of the stochastic llg [length 3]
- `state`: current state of the magnetisation vector [length 3]
- `t`: time (has no effect)
- `alpha`: damping ratio
- `the`: effective field on the magnetisation [length 3]

void **drift_jacobian**(double **deriv*, **const** double **state*, **const** double *time*, **const** double *alpha*, **const** double **heff*, **const** double **heff_jac*)

Jacobian of the deterministic drift component of the stochastic LLG Since, in the general case, the effective field is a function of the magnetisation, the Jacobian of the effective field must be known in order to compute the Jacobian of the drift component.

Parameters

- `jac`: Jacobian of the drift [length 3x3]
- `m`: state of the magnetisation vector [length 3]
- `t`: time (has no effect)
- `a`: damping ratio α
- `h`: effective field acting on the magnetisation [length 3]
- `hj`: Jacobian of the effective field evaluated at the current value of `m` [length 3x3]

void **diffusion**(double **deriv*, **const** double **state*, **const** double *time*, **const** double *sr*, **const** double *alpha*)

The stochastic diffusion component of the stochastic LLG.

Parameters

- `deriv`: diffusion derivatives [length 3x3]
- `state`: current state of the magnetisation [length 3]
- `t`: time (has no effect)
- `sr`: normalised noise power of the thermal field (see notes on LLG normalisation for details)

- `alpha`: damping ratio

void **diffusion_jacobian**(double **jacobian*, **const** double **state*, **const** double *time*, **const** double *sr*, **const** double *alpha*)

Jacobian of the stochastic diffusion component of the LLG.

Parameters

- `jacobian`: Jacobian of the LLG diffusion [length 3x3]
- `state`: current state of the magnetisation vector [length 3]
- `t`: time (has no effect)
- `sr`: normalised noise power of the thermal field (see notes on LLG normalisation for details)
- `alpha`: damping ratio

void **sde_with_update**(double **drift*, double **diffusion*, double **heff*, **const** double **current_state*, **const** double *drift_time*, **const** double *diffusion_time*, **const** double **happ*, **const** double **anisotropy_axis*, **const** double *alpha*, **const** double *noise_power*)

Computes drift and diffusion of LLG after updating the field.

The effective field is first computed based on the applied field and current state of the magnetisation. This is then used to compute the current drift and diffusion components of the LLG. Assumes uniaxial anisotropy.

Parameters

- `drift`: deterministic component of the LLG [length 3]
- `diffusion`: stochastic component of the LLG [length 3x3]
- `heff`: effective field including the applied field contribution [length 3]
- `state`: current state of the magnetisation [length 3]
- `a_t`: time at which to evaluate the drift
- `b_t`: time at which to evaluate the diffusion
- `happ`: the applied field at time `a_t` [length 3]
- `aaxis`: the anisotropy axis of the particle [length 3]
- `alpha`: damping ratio
- `sr`: normalised noise power of the thermal field (see notes on LLG normalisation for details)

void **jacobians_with_update**(double **drift*, double **diffusion*, double **drift_jac*, double **diffusion_jac*, double **heff*, double **heff_jac*, **const** double **current_state*, **const** double *drift_time*, **const** double *diffusion_time*, **const** double **happ*, **const** double **anisotropy_axis*, **const** double *alpha*, **const** double *noise_power*)

Computes effective field, drift, diffusion and Jacobians of LLG.

The effective field is first computed based on the applied field and current state of the magnetisation. This is then used to compute the drift, diffusion, and their respective Jacobians. Assumes uniaxial anisotropy.

Parameters

- `drift`: deterministic component of the LLG [length 3]
- `diffusion`: stochastic component of the LLG [length 3x3]

- drift_jac: Jacobian of the deterministic component [length 3x3]
- diffusion_jac: Jacobian of the diffusion component [length 3x3x3]
- heff: effective field including the applied field contribution [length 3]
- heff_jac: Jacobian of the effective field [length 3x3]
- state: current state of the magnetisation [length 3]
- a_t: time at which to evaluate the drift
- b_t: time at which to evaluate the diffusion
- happ: the applied field at time a_t [length 3]
- aaxis: the anisotropy axis of the particle [length 3]
- alpha: damping ratio
- s: normalised noise power of the thermal field (see notes on LLG normalisation for details)

void **multi_drift**(double **deriv*, **const** double **state*, **const** double **alphas*, **const** double **heff*, **const** size_t *N_particles*)

Deterministic drift component of the stochastic LLG for many particles.

Parameters

- deriv: drift derivative of the deterministic part of the stochastic llg for each particle [length 3xN]
- state: current state of the magnetisation vectors [length 3xN]
- t: time (has no effect)
- alpha: damping ratio
- heff: the effective field on each particle [length 3xN]
- N_particles: the number of particles

void **multi_diffusion**(double **deriv*, **const** double **state*, **const** double **field_strengths*, **const** double **alphas*, **const** size_t *N_particles*)

Compute 3x3 block diagonal multi diffusion.

Note zero terms are not written.

void **multi_stochastic_llg_field_update**(double **drift*, double **diffusion*, double **heff*, **const** std::function<void(double *, **const** double *, **const** double > *heff_func*, **const** double **state*, **const** double *t*, **const** double **alphas*, **const** double **field_strengths*, **const** size_t *N_particles*)

Updates field and computes LLG for N interacting particles.

heff_fuc is a function that returns the effective field given the current state and the current time. This can be whatever you want e.g. cubic anisotropy terms and interactions. EZEEE.

void **multi_drift_quasijacobian**(double **jac*, **const** double **m*, **const** double **alphas*, **const** double **h*, **const** double **hj*, size_t *N_particles*)

Computes the Jacobian of the drift for N interacting particles.

Assumes that jac is zero'd (i.e. function will not fill in 0 entries)

void **multi_diffusion_jacobian**(double **jacobian*, **const** double **state*, **const** double **therm_field_strengths*, **const** double **alphas*, **const** size_t *N_particles*)

Computes the Jacobian of the diffusion for N interacting particles.

Only computes non-zero values. jacobian must be 0 initialised before function call.

void **multi_stochastic_llg_jacobians_field_update**(double **drift*, double **diffusion*, double **drift_jac*, double **diffusion_jac*, double **heff*, double **heff_jac*, **const** double **state*, **const** double *t*, **const** double **alphas*, **const** double **field_strengths*, **const** size_t *N_particles*, **const** std::function<void(double *, **const** double *, **const** double *, **const** double *, **const** double *, **const** double *, **const** double *, **const** double *, **const** double *)> *heff_func*, **const** std::function<voiddouble *, **const** double *, **const** double> *heff_jac_func*)

Computes all fields, drift/diffusion, jacobians for N particles.

The effective field is first computed based on the applied field and current state of the magnetisation. This is then used to compute the drift, diffusion, and their respective Jacobians. Assumes uniaxial anisotropy.

Parameters

- `drift`: deterministic component of the LLG [length 3]
- `diffusion`: stochastic component of the LLG [length 3x3]
- `drift_jac`: Jacobian of the deterministic component [length 3x3]
- `diffusion_jac`: Jacobian of the diffusion component [length 3x3x3]
- `heff`: effective field including the applied field contribution [length 3]
- `heff_jac`: Jacobian of the effective field [length 3x3]
- `state`: current state of the magnetisation [length 3]
- `a_t`: time at which to evaluate the drift
- `b_t`: time at which to evaluate the diffusion
- `happ`: the applied field at time `a_t` [length 3]
- `aaxis`: the anisotropy axis of the particle [length 3]
- `alpha`: damping ratio
- `s`: normalised noise power of the thermal field (see notes on LLG normalisation for details)

7.1.12 Namespace mnp

namespace mnp

Typedefs

using mnp::axis = typedef std::array<double,3>

7.1.13 Namespace moma_config

Warning: doxygennamespace: Cannot find namespace "moma_config" in doxygen xml output for project "magpy-api" from directory: ../doxygen/doxyxml

7.1.14 Namespace optimisation

namespace optimisation

Functions

int **newton_raphson_1**(double *x_root, **const** std::function<double) **const** double > *f*, **const** std::function<double**const** double> *fdash*, **const** double *x0*, **const** double *eps* = 1e-7, **const** size_t *max_iter* = 1000

int **newton_raphson_noinv**(double *x_root, double *x_tmp, double *jac_out, lapack_int *ipiv, int *lapack_err_code, **const** std::function<void) double *, double *, **const** double * > *func_and_jacobian*, **const** double *x0, **const** lapack_int *dim*, **const** double *eps* = 1e-7, **const** size_t *max_iter* = 1000

Variables

const int **SUCCESS** = 0
Optimisation success return code.

const int **MAX_ITERATIONS_ERR** = 1
Optimisation maximum iterations reached error code.

const int **LAPACK_ERR** = 2
Otimisation internal LAPACK error code.

This error code indicates an error ocurred in an internal LAPACK call. Further investigation will be needed to determine the cause.

7.1.15 Namespace simulation

namespace simulation

Functions

std::vector<**struct** *results*> **full_dynamics**(**const** std::vector<double> *thermal_field_strengths*, **const** std::vector<double> *reduced_anisotropy_constants*, **const** std::vector<double> *reduced_particle_volumes*, **const** std::vector<*d3*> *anisotropy_unit_axes*, **const** std::vector<*d3*> *initial_magnetisations*, **const** std::vector<std::vector<*d3*>> *interparticle_unit_distances*, **const** std::vector<std::vector<double>> *interparticle_reduced_distance_magnitudes*, **const** std::function<double) **const** double > *applied_field*, **const** double *average_anisotropy*, **const** double *average_volume*, **const** double *damping_constant*, **const** double *saturation_magnetisation*, **const** double *time_step*, **const** double *end_time*, *Rng* &*rng*, **const** bool *renorm*, **const** bool *interactions*, **const** bool *use_implicit*, **const** double *eps*, **const** int *max_samples*

std::vector<*results*> **full_dynamics**(**const** std::vector<double> *radius*, **const** std::vector<double> *anisotropy*, **const** std::vector<*d3*> *anisotropy_axes*, **const** std::vector<*d3*> *magnetisation_direction*, **const** std::vector<*d3*> *location*, **const** double *magnetisation*, **const** double *damping*, **const** double *temperature*, **const** bool *renorm*, **const** bool *interactions*, **const** bool *use_implicit*, **const** double *eps*, **const** double *time_step*, **const** double *end_time*, **const** size_t *max_samples*, **const** long *seed*, **const** *field*::*options* *field_option* = field::CONSTANT, **const** double *field_amplitude* = 0.0, **const** double *field_frequency* = 0.0)

struct *results* **dom_ensemble_dynamics**(**const** double *volume*, **const** double *anisotropy*, **const** double *temperature*, **const** double *magnetisation*, **const** double *alpha*, **const** std::function<double) double > *applied_field*, **const** std::array<double, 2> *initial_mags*, **const** double *time_step*, **const** double *end_time*, **const** int *max_samples*

void **save_results**(**const** std::string *fname*, **const** **struct** *results* &*res*)
Save results to disk.

Saves the contents of a results struct to disk. Given a file name 'foo' the following files are written to disk 'foo.mx', 'foo.my', 'foo.mz', 'foo.field', 'foo.time', 'foo.energy'.

Parameters

- `fname`: /path/to/filename prefix for files
- `res`: results struct to save

void **zero_results**(**struct** *results* &*res*)
Initialise results memory to zero.

Parameters

- `res`: results struct to zero

void **reduce_to_system_magnetisation**(double **mag*, **const** double **particle_mags*, **const** size_t *N_particles*)

7.1.16 Namespace stochastic

namespace stochastic

Functions

void **master_equation**(double **derivs*, **const** double **transition_matrix*, **const** double **current_state*, **const** size_t *dim*)

Evaluates the derivatives of the master equation given a transition matrix.

The master equation is simply a linear system of ODEs, with coefficients described by the transition matrix. $\frac{\mathrm{d}x}{\mathrm{d}t} = Wx$

Parameters

- `derivs`: the master equation derivatives [length dim]
- `transition_matrix`: the [dim x dim] transition matrix (row-major) W
- `current_state`: values of the state vector x

7.1.17 Namespace trap

namespace trap

numerical schemes for computing the area under curves

Author Oliver Laslett

Functions

double **trapezoidal**(double **x*, double **y*, size_t *N*)

double **one_trapezoid**(double *x1*, double *x2*, double *fx1*, double *fx2*)

7.2 File list

7.2.1 File constants.hpp

namespace constants

Variables

const double **KB** =1.38064852e-23

const double **MU0** =1.25663706e-6

const double **GYROMAG** =1.76086e11

7.2.2 File curry.hpp

Warning: doxygenfile: Cannot find file "curry.hpp

7.2.3 File distances.cpp

Functions

std::vector<std::vector<double>> **pair_wise_distance_magnitude**(std::vector<std::array<double, 3>> *points*)

Computes the Euclidean distance between pairs of points.

namespace distances

Compute distances between points.

Functions for computing distances between points in 3D space.

Author Oliver Laslett

Date 2017

7.2.4 File distances.hpp

namespace distances

Compute distances between points.

Functions for computing distances between points in 3D space.

Author Oliver Laslett

Date 2017

Functions

std::vector<std::vector<std::array<double, 3>>> **pair_wise_distance_vectors**(std::vector<std::array<double, 3>> *points*)

Computes the distance vector between points in 3d space.

Given a list of x,y,z coordinates, compute the distance vector between every pair of x,y,z coordinates in the list.

Return NxN std::vector matrix of distances between each pair of points

Parameters

- `points`: length N vector of arrays of x,y,z coordinates

std::vector<std::vector<double>> **pair_wise_distance_magnitude**(std::vector<std::array<double, 3>> *points*)

Computes the Euclidean distance between pairs of points.

std::vector<std::vector<std::array<double, 3>>> **pair_wise_distance_unit_vectors**(std::vector<std::array<double, 3>> *points*)

Computes the unit distance vector between a list of points.

std::vector<std::vector<double>> **pair_wise_distance_magnitude**(std::vector<std::vector<std::array<double, 3>>> *distance_vectors*)

Computes the Euclidean distance from a matrix of distance vectors.

Given a matrix of distance vectors, compute the 2norm of the vector corresponding to the Euclidean distance.

Return NxN matrix of Euclidean distance values

Parameters

- `distance_vectors`: NxN symmetrical matrix of distance vectors (x,y,z coordinates)

std::vector<std::vector<std::array<double, 3>>> **pair_wise_distance_unit_vectors**(std::vector<std::vector<std::array<double, 3>>> *distance_vectors*, std::vector<std::vector<double>> *distance_magnitudes*)

Computes the unit distance vectors from distance vectors and their magnitudes.

7.2.5 File dom.cpp

7.2.6 File dom.hpp

namespace dom

discrete orientation model for magnetic particles

Functions

void **transition_matrix**(double *W, **const** double *k*, **const** double *v*, **const** double *T*, **const** double *h*, **const** double *ms*, **const** double *alpha*)

Compute the 2x2 transition matrix for a single particle.

Assumes uniaxial anisotropy and an applied field h<1 Only valid for large energy barriers: $(1-h)^2>>1$

Parameters

- `W`: transition matrix [2x2]
- `k`: anisotropy strength constant for the uniaxial anisotropy
- `v`: volume of the particle in meter^3
- `T`: temperature of environment in Kelvin
- `h`: dimensionless applied field - normalised by $H_k = \frac{2K}{\mu_0 M_s}$
- `ms`: saturation magnetisation
- `alpha`: dimensionless damping constant

void **master_equation_with_update**(double *derivs, double *work, **const** double *k*, **const** double *v*, **const** double *T*, **const** double *ms*, **const** double *alpha*, **const** double *t*, **const** double *state_probabilities, **const** std::function<double) double > *applied_field*Computes master equation for particle in time-dependent field.

Computes the applied field in the z-direction at time t. Uses the field value to compute the transition matrix and corresponding master equation derivatives for the single particle.

Parameters

- `derivs`: master equation derivatives [length 2]
- `work`: vector [length 4]
- `k`: anisotropy strength constant for the uniaxial anisotropy
- `v`: volume of the particle in meter^3
- `T`: temperature of environment in Kelvin
- `ms`: saturation magnetisation
- `alpha`: dimensionless damping constant
- `t`: time at which to evaluate the external field
- `state_probabilities`: the current state probabilities for each of the 2 states (up and down) [length 2]
- `applied_field`: a scalar function takes a double and returns a double. Given the current time should return normalised field $h = (t)$ where the field is normalised by H_k

double **single_transition_energy**(double *K*, double *V*, double *h*)

Compute the energy dissipated during one transition of the particle.

Return the energy dissipated in one transition

Parameters

- `K`: anisotropy constant of the particle
- `V`: volume of the particle
- `h`: applied field on the particle, reduced by the anisotropy field $H_k = \frac{2K}{\mu_0 M_s}$

7.2.7 File energy.hpp

namespace energy

Functions

double **anisotropy_r**()

double **zeeman_r**()

double **dipolar_r**()

7.2.8 File field.cpp

Defines

_USE_MATH_DEFINES

namespace field
Contains functions for computing magnetic fields.

Functions for computing effective fields from anistoropy and a range of time-varying applied fields.

Author Oliver Laslett

Date 2017

7.2.9 File field.hpp

namespace field
Contains functions for computing magnetic fields.

Functions for computing effective fields from anistoropy and a range of time-varying applied fields.

Author Oliver Laslett

Date 2017

Enums

enum options
Values:

SINE

SQUARE

CONSTANT

Functions

template <typename... *T*>
std::function<double (double) > **bind_field_function**
std::function<doubledouble, T...> *func*, T... *bind_args*Bind field parameters into a field function.

Field functions are of the form `std::function(double<double>)` This function allows additional arguments to be bound to functions and returns the function as a field function.

Return field function

Parameters

- `func`: function to bind
- `bind_args`: arguments to bind

double **constant** (**const** double *t*, **const** double *h*)
A constant applied field.

A simple placeholder function representing a constant field. Always returns the same value.

Return the constant field amplitude at all values of time

Parameters

- `t`: time (parameter has no effect)
- `h`: applied field amplitude

double **sinusoidal**(**const** double *t*, **const** double *h*, **const** double *f*)
A sinusoidal alternating applied field.

Returns the value of a sinusoidally varying field at any given time.

Return the value of the varying applied field at time `t`

Parameters

- `t`: time
- `h`: applied field amplitude
- `f`: applied field frequency

double **square**(**const** double *t*, **const** double *h*, **const** double *f*)
A square wave switching applied field.

An alternating applied field with a square shape centred around zero. i.e. it alternates between values `-h` and `h`

Return the value of the square wave applied field at time `t`

Parameters

- `t`: time
- `h`: applied field amplitude
- `f`: applied field frequency

double **square_fourier**(**const** double *t*, **const** double *h*, **const** double *f*, **const** size_t *n_compononents*)
A square wave applied field of finite Fourier components.

An approximate square wave is computed from a finite number of Fourier components. The square wave alternates between `-h` and `h`.

Return the value of the square wave applied field at time `t`

Parameters

- `t`: time
- `h`: applied field amplitude
- `f`: applied field frequency
- `n_components`: number of Fourier components to compute

void **multi_add_applied_Z_field_function**(double **heff*, **const** std::function<double)**const** double > *&hfunc*, **const** double *t*, **const** size_t *N*Add the applied field in the z direction.

void **uniaxial_anisotropy**(double **h_anis*, **const** double **magnetisation*, **const** double **anis_axis*)
Effective field contribution from uniaxial anisotropy.

The effective field experienced by a single particle with a uniaxial anisotropy.

Parameters

- `h_anis`: effective field [length 3]
- `mag`: the magnetisation of the particle of [length 3]
- `axis`: the anisotropy axis of the particle [length 3]

void **multi_add_uniaxial_anisotropy**(double **h*, **const** double **states*, **const** double **axes*, **const** double **k_reduced*, **const** size_t *N*)

Add the uniaxial anisotropy term term to the field of N particles.

Parameters

- `h`: effective field incremented with the anisotropy term
- `states`: magnetic state of the particles (length 3N)
- `axes`: anisotropy axis of each particle (length 3N)
- `k_reduced`: the reduced anisotropy constant for each particle (length N)
- `N`: number of particles

void **uniaxial_anisotropy_jacobian**(double **jac*, **const** double **axis*)

Jacobian of the uniaxial anisotropy effective field term.

The Jacobian of a particle's uniaxial anisotropy with respect to it's magnetisation value. $J_h(m) = \frac{\partial h(m)}{\partial m}$

Parameters

- `jac`: the jacobian of the effective field [length 3x3]
- `axis`: the anisotropy axis of the particle [length 3]

void **multi_add_uniaxial_anisotropy_jacobian**(double **jac*, **const** double **axes*, **const** double **anis*, **const** size_t *N*)

Jacobian of the uniaxial anisotropy effective field term for many particles.

The Jacobian of a particle's uniaxial anisotropy with respect to it's magnetisation value. $J_h(m) = \frac{\partial h(m)}{\partial m}$

Parameters

- `jac`: the jacobian of the effective field [length 3Nx3N]
- `axis`: the anisotropy axis of the particle [length 3N]
- `N`: number of particles

void **multi_add_dipolar**(double **field*, **const** double *ms*, **const** double *k_av*, **const** double **v_reduced*, **const** double **mag*, **const** double **dists*, **const** double **dist_cubes*, **const** size_t *N*)

field is Nx3 and is the effective field on each particle ms is the same for all particles k_av is the average anisotropy constant for all particles v_reduced is the reduced volume of each particle v_av is the average volume for all particles mag is Nx3 long and is the magnetic state of each of the N particles dists is NxNx3 long and is the distance between each pair of particles N is the number of particles dist_cubes is NxN and is the reduced cubed distance between each pair

double **dipolar_prefactor**(**const** double *ms*, **const** double *k_av*)

Prefactor term (helper for field::dipolar)

prefactor is $\frac{\mu 0 M_s^2}{8\pi K}$

Return prefactor term

Parameters

- `ms`: saturation magnetisation
- `k_av`: average anisotropy constant for system

void **dipolar_add_p2p_term**(double **out*, **const** double *vj*, **const** double *rij3*, **const** double **mj*, **const** double **dist*, **const** double *prefactor*)

void **zero_all_field_terms**(double *h, **const** size_t N)
Set all N values of the field to zero.

Parameters

- h: pointer to array of N doubles will all be set to zero
- N: number of elements in array

7.2.10 File integrators.cpp

Integrator implementation

Typedefs

using sde_function = std::function<void(double *, **const** double *, **const** double)>

using sde_func = std::function<void(double *, double *, **const** double *, **const** double)>

using sde_jac = std::function<void(double *, double *, double *, double *, **const** double *, **const** double, **const** double)>

namespace integrator
Numerical methods for differential equations.

Numerical methods for simulating the time evolution of deterministic and stochastic ordinary nonlinear differential equations. All integrators compute a single step of the solution.

Author Oliver Laslett

Date 2017

namespace driver
High level interface to differential equation integrators.

Drivers wrap the single step integrators and provide an interface for simulating differential equations for multiple steps from the initial condition. Drivers also handle memory management of the work arrays needed by the integrators.

Author Oliver Laslett

Date 2017

7.2.11 File integrators.hpp

Header includes various numerical methods for estimating solutions to stochastic and deterministic ODEs. Methods included in this are: -ODEs: RK4 -SDEs: Euler, Heun

namespace driver
High level interface to differential equation integrators.

Drivers wrap the single step integrators and provide an interface for simulating differential equations for multiple steps from the initial condition. Drivers also handle memory management of the work arrays needed by the integrators.

Author Oliver Laslett

Date 2017

Functions

void **rk4**(double **states*, **const** double **initial_state*, **const** std::function<void) double *, **const** double *, **const** double
> *derivs*, **const** size_t *n_steps*, **const** size_t *n_dims*, **const** double *step_size*

void **eulerm**(double **states*, **const** double **initial_state*, **const** double **wiener_process*, **const** std::function<void) double *, **const** double *, **const** double
> *drift*, **const** std::function<voiddouble *, **const** double *, **const** double> *diffusion*, **const** size_t *n_steps*, **const** size_t *n_dims*, **const** size_t *n_wiener*, **const** double *step_size*

void **heun**(double **states*, **const** double **initial_state*, **const** double **wiener_process*, **const** std::function<void) double *, double *, **const** double *, **const** double
> *sde*, **const** size_t *n_steps*, **const** size_t *n_dims*, **const** size_t *n_wiener*, **const** double *step_size*

void **implicit_midpoint**(double **x*, **const** double **x0*, **const** double **dw*, **const** std::function<void) double *, double *, double *, double *, **const** double *, **const** double, **const** double
> *sde*, **const** size_t *n_dim*, **const** size_t *w_dim*, **const** size_t *n_steps*, **const** double *t0*, **const** double *dt*, **const** double *eps*, **const** size_t *max_iter*

namespace integrator

Numerical methods for differential equations.

Numerical methods for simulating the time evolution of deterministic and stochastic ordinary nonlinear differential equations. All integrators compute a single step of the solution.

Author Oliver Laslett

Date 2017

Functions

void **rk4**(double **next_state*, double **k1*, double **k2*, double **k3*, double **k4*, **const** double **current_state*, **const** std::function<void) double *, **const** double *, **const** double
> *derivs*, **const** size_t *n_dims*, **const** double *t*, **const** double *h*

void **heun**(double **next_state*, double **drift_arr*, double **trial_drift_arr*, double **diffusion_matrix*, double **trial_diffusion_matrix*, **const** double **current_state*, **const** double **wiener_steps*, **const** std::function<void) double *, double *, **const** double *, **const** double
> *sde*, **const** size_t *n_dims*, **const** size_t *wiener_dims*, **const** double *t*, **const** double *step_size*

void **eulerm**(double **states*, double **diffusion_matrix*, **const** double **initial_state*, **const** double **wiener_process*, **const** std::function<void) double *, **const** double *, **const** double
> *drift*, **const** std::function<voiddouble *, **const** double *, **const** double> *diffusion*, **const** size_t *n_dims*, **const** size_t *n_wiener*, **const** double *t*, **const** double *step_size*

template <**class** CSTATES, **class** CDIFF>
void **milstein**(CSTATES &*next_state*, **const** CSTATES &*current_state*, **const** CSTATES &*drift*, **const** CDIFF &*diffusion*, **const** CSTATES &*wiener_increments*, **const** double *step_size*)

int **implicit_midpoint**(double **x*, double **dwm*, double **a_work*, double **b_work*, double **adash_work*, double **bdash_work*, double **x_guess*, double **x_opt_tmp*, double **x_opt_jac*, lapack_int **x_opt_ipiv*, **const** double **x0*, **const** double **dw*, **const** std::function<void) double *, double *, double *, double *, **const** double *, **const** double, **const** double
> *sde*, **const** size_t *n_dim*, **const** size_t *w_dim*, **const** double *t*, **const** double *dt*, **const** double *eps*, **const** size_t *max_iter*

void **rk45**(double **next_state*, double **temp_state*, double **k1*, double **k2*, double **k3*, double **k4*, double **k5*, double **k6*, double **h_ptr*, double **t_ptr*, **const** double **current_state*, **const** std::function<void) double *, **const** double *, **const** double > *ode*, **const** size_t *n_dims*, **const** double *eps*RK45 Cash-Karp adaptive step deterministic ODE solver.

Solves ODEs :) returns the next state and the time step to be used in the next instance what about the actual time step that it used? That's also important!

namespace ck_butcher_table

Cash-Karp parameter table for RK45.

Variables

constexpr double **c11** = 0.2

constexpr double **c21** = 3.0/40.0

constexpr double **c22** = 9.0/40.0

constexpr double **c31** = 3.0/10.0

constexpr double **c32** = -9.0/10.0

constexpr double **c33** = 6.0/5.0

constexpr double **c41** = -11.0/54.0

constexpr double **c42** = 2.5

constexpr double **c43** = -70.0/27.0

constexpr double **c44** = 35.0/27.0

constexpr double **c51** = 1631.0/55296.0

constexpr double **c52** = 175.0/512.0

constexpr double **c53** = 575.0/13824.0

constexpr double **c54** = 44275.0/110592.0

constexpr double **c55** = 253.0/4096.0

constexpr double **hc1** = 0.2

constexpr double **hc2** = 0.3

constexpr double **hc3** = 0.6

constexpr double **hc4** = 1.0

constexpr double **hc5** = 7.0/8.0

constexpr double **x11** = 37.0/378.0

constexpr double **x13** = 250.0/621.0

constexpr double **x14** = 125.0/594.0

constexpr double **x16** = 512.0/1771.0

constexpr double **x21** = 2825.0/27648.0

constexpr double **x23** = 18575.0/48384.0

constexpr double **x24** = 13525.0/55296.0

constexpr double **x25** = 277.0/14336.0

constexpr double **x26** = 0.25

7.2.12 File io.hpp

namespace io

Functions

template <typename T>
int **write_array**(**const** std::string *fname*, T **const** **arr*, **const** size_t *len*)

7.2.13 File llg.cpp

7.2.14 File llg.hpp

namespace llg

Functions for evaluating the Landau-Lifshitz-Gilbert equation.

Includes the basic equation as well as Jacobians and combined functions to update fields during integration.

Author Oliver Laslett

Date 2017

Functions

void **drift**(double **deriv*, **const** double **state*, **const** double *time*, **const** double *alpha*, **const** double **heff*)

Deterministic drift component of the stochastic LLG.

Parameters

- `deriv`: drift derivative of the deterministic part of the stochastic llg [length 3]
- `state`: current state of the magnetisation vector [length 3]
- `t`: time (has no effect)
- `alpha`: damping ratio
- `the`: effective field on the magnetisation [length 3]

void **drift_jacobian**(double **deriv*, **const** double **state*, **const** double *time*, **const** double *alpha*, **const** double **heff*, **const** double **heff_jac*)

Jacobian of the deterministic drift component of the stochastic LLG Since, in the general case, the effective field is a function of the magnetisation, the Jacobian of the effective field must be known in order to compute the Jacobian of the drift component.

Parameters

- `jac`: Jacobian of the drift [length 3x3]
- `m`: state of the magnetisation vector [length 3]
- `t`: time (has no effect)

- a: damping ratio α
- h: effective field acting on the magnetisation [length 3]
- hj: Jacobian of the effective field evaluated at the current value of m [length 3x3]

void **diffusion**(double **deriv*, **const** double **state*, **const** double *time*, **const** double *sr*, **const** double *alpha*)

The stochastic diffusion component of the stochastic LLG.

Parameters

- deriv: diffusion derivatives [length 3x3]
- state: current state of the magnetisation [length 3]
- t: time (has no effect)
- sr: normalised noise power of the thermal field (see notes on LLG normalisation for details)
- alpha: damping ratio

void **diffusion_jacobian**(double **jacobian*, **const** double **state*, **const** double *time*, **const** double *sr*, **const** double *alpha*)

Jacobian of the stochastic diffusion component of the LLG.

Parameters

- jacobian: Jacobian of the LLG diffusion [length 3x3]
- state: current state of the magnetisation vector [length 3]
- t: time (has no effect)
- sr: normalised noise power of the thermal field (see notes on LLG normalisation for details)
- alpha: damping ratio

void **sde_with_update**(double **drift*, double **diffusion*, double **heff*, **const** double **current_state*, **const** double *drift_time*, **const** double *diffusion_time*, **const** double **happ*, **const** double **anisotropy_axis*, **const** double *alpha*, **const** double *noise_power*)

Computes drift and diffusion of LLG after updating the field.

The effective field is first computed based on the applied field and current state of the magnetisation. This is then used to compute the current drift and diffusion components of the LLG. Assumes uniaxial anisotropy.

Parameters

- drift: deterministic component of the LLG [length 3]
- diffusion: stochastic component of the LLG [length 3x3]
- heff: effective field including the applied field contribution [length 3]
- state: current state of the magnetisation [length 3]
- a_t: time at which to evaluate the drift
- b_t: time at which to evaluate the diffusion
- happ: the applied field at time a_t [length 3]
- aaxis: the anisotropy axis of the particle [length 3]
- alpha: damping ratio

- `sr`: normalised noise power of the thermal field (see notes on LLG normalisation for details)

void **jacobians_with_update**(double **drift*, double **diffusion*, double **drift_jac*, double **diffusion_jac*, double **heff*, double **heff_jac*, **const** double **current_state*, **const** double *drift_time*, **const** double *diffusion_time*, **const** double **happ*, **const** double **anisotropy_axis*, **const** double *alpha*, **const** double *noise_power*)

Computes effective field, drift, diffusion and Jacobians of LLG.

The effective field is first computed based on the applied field and current state of the magnetisation. This is then used to compute the drift, diffusion, and their respective Jacobians. Assumes uniaxial anisotropy.

Parameters

- `drift`: deterministic component of the LLG [length 3]
- `diffusion`: stochastic component of the LLG [length 3x3]
- `drift_jac`: Jacobian of the deterministic component [length 3x3]
- `diffusion_jac`: Jacobian of the diffusion component [length 3x3x3]
- `heff`: effective field including the applied field contribution [length 3]
- `heff_jac`: Jacobian of the effective field [length 3x3]
- `state`: current state of the magnetisation [length 3]
- `a_t`: time at which to evaluate the drift
- `b_t`: time at which to evaluate the diffusion
- `happ`: the applied field at time `a_t` [length 3]
- `aaxis`: the anisotropy axis of the particle [length 3]
- `alpha`: damping ratio
- `s`: normalised noise power of the thermal field (see notes on LLG normalisation for details)

void **multi_drift**(double **deriv*, **const** double **state*, **const** double **alphas*, **const** double **heff*, **const** size_t *N_particles*)

Deterministic drift component of the stochastic LLG for many particles.

Parameters

- `deriv`: drift derivative of the deterministic part of the stochastic llg for each particle [length 3xN]
- `state`: current state of the magnetisation vectors [length 3xN]
- `t`: time (has no effect)
- `alpha`: damping ratio
- `heff`: the effective field on each particle [length 3xN]
- `N_particles`: the number of particles

void **multi_diffusion**(double **deriv*, **const** double **state*, **const** double **field_strengths*, **const** double **alphas*, **const** size_t *N_particles*)

Compute 3x3 block diagonal multi diffusion.

Note zero terms are not written.

void **multi_stochastic_llg_field_update**(double **drift*, double **diffusion*, double **heff*, **const** std::function<void) double *, **const** double *, **const** double > *heff_func*, **const** double **state*, **const** double *t*, **const** double **alphas*, **const** double **field_strengths*, **const** size_t *N_particles*Updates field and computes LLG for N interacting particles.

heff_fuc is a function that returns the effective field given the current state and the current time. This can be whatever you want e.g. cubic anisotropy terms and interactions. EZEEE.

void **multi_drift_quasijacobian**(double **jac*, **const** double **m*, **const** double **alphas*, **const** double **h*, **const** double **hj*, size_t *N_particles*)

Computes the Jacobian of the drift for N interacting particles.

Assumes that jac is zero'd (i.e. function will not fill in 0 entries)

void **multi_diffusion_jacobian**(double **jacobian*, **const** double **state*, **const** double **therm_field_strengths*, **const** double **alphas*, **const** size_t *N_particles*)

Computes the Jacobian of the diffusion for N interacting particles.

Only computes non-zero values. jacobian must be 0 initialised before function call.

void **multi_stochastic_llg_jacobians_field_update**(double **drift*, double **diffusion*, double **drift_jac*, double **diffusion_jac*, double **heff*, double **heff_jac*, **const** double **state*, **const** double *t*, **const** double **alphas*, **const** double **field_strengths*, **const** size_t *N_particles*, **const** std::function<void) double *, **const** double *, **const** double > *heff_func*, **const** std::function<voiddouble *, **const** double *, **const** double> *heff_jac_func*Computes all fields, drift/diffusion, jacobians for N particles.

The effective field is first computed based on the applied field and current state of the magnetisation. This is then used to compute the drift, diffusion, and their respective Jacobians. Assumes uniaxial anisotropy.

Parameters

- `drift`: deterministic component of the LLG [length 3]
- `diffusion`: stochastic component of the LLG [length 3x3]
- `drift_jac`: Jacobian of the deterministic component [length 3x3]
- `diffusion_jac`: Jacobian of the diffusion component [length 3x3x3]
- `heff`: effective field including the applied field contribution [length 3]
- `heff_jac`: Jacobian of the effective field [length 3x3]
- `state`: current state of the magnetisation [length 3]
- `a_t`: time at which to evaluate the drift
- `b_t`: time at which to evaluate the diffusion
- `happ`: the applied field at time `a_t` [length 3]
- `aaxis`: the anisotropy axis of the particle [length 3]
- `alpha`: damping ratio
- `s`: normalised noise power of the thermal field (see notes on LLG normalisation for details)

7.2.15 File mnps.hpp

namespace mnp

Typedefs

using mnp::axis = typedef std::array<double,3>

struct norm_params

Public Members

double **gamma**

double **alpha**

double **stability**

double **volume**

double **temperature**

axis **anisotropy_axis**

struct params

Public Members

double **gamma**

double **alpha**

double **saturation_mag**

double **diameter**

double **anisotropy**

axis **anisotropy_axis**

7.2.16 File moma_config.cpp

Warning: doxygenfile: Cannot find file "moma_config.cpp

7.2.17 File moma_config.hpp

Warning: doxygenfile: Cannot find file "moma_config.hpp

7.2.18 File optimisation.cpp

Numerical methods for optimisation and root finding.

Contains Newton Raphson methods for root finding.

7.2.19 File optimisation.hpp

Contains numerical methods for optimisation and root-finding.

Author Oliver W. Laslett

Date 2016

namespace optimisation

Functions

int **newton_raphson_1**(double *x_root, **const** std::function<double) **const** double
> *f*, **const** std::function<double**const** double> *fdash*, **const** double *x0*, **const** double *eps* = 1e-7, **const** size_t *max_iter* = 1000

int **newton_raphson_noinv**(double *x_root, double *x_tmp, double *jac_out, lapack_int *ipiv, int *lapack_err_code, **const** std::function<void) double *, double *, **const** double *
> *func_and_jacobian*, **const** double *x0*, **const** lapack_int *dim*, **const** double *eps* = 1e-7, **const** size_t *max_iter* = 1000

Variables

const int **SUCCESS** = 0
Optimisation success return code.

const int **MAX_ITERATIONS_ERR** = 1
Optimisation maximum iterations reached error code.

const int **LAPACK_ERR** = 2
Otimisation internal LAPACK error code.

This error code indicates an error ocurred in an internal LAPACK call. Further investigation will be needed to determine the cause.

7.2.20 File rng.cpp

Class fr holding different random number generators.

7.2.21 File rng.hpp

Class for holding different random number generators.

Author Oliver W. Laslett

class Rng
#include <*rng.hpp*> Abstract class for random number generators.

Subclassed by *RngArray*, *RngMtDownsample*, *RngMtNorm*

Public Functions

virtual double **get**() = 0
Get a single random number.

Return a single draw from the random number generator

class RngMtNorm
#include <*rng.hpp*> Uses Mersenne Twister to generate normally distributed values.

Random numbers have a mean of zero and a user specified standard deviation.

Inherits from *Rng*

Public Functions

RngMtNorm(**const** unsigned long int *seed*, **const** double *std*)
Default constructor for *RngMtNorm*.

Parameters

- `seed`: seed for random number generator
- `std`: the standard deviation of the normally distributed numbers

double **get**()
Draw a single normally distributed value from the RNG.

Return single normally distributed number

Private Members

std::mt19937_64 **generator**
A Mersenne twister generator instance.

std::normal_distribution<double> **dist**
A normal distribution instance.

class RngMtDownsample
#include <*rng.hpp*> Generate normally distributed values with downsampling.

Uses the Mersenne Twister to generate normally distributed random numbers. Down-samples the stream of random numbers by summing consecutive draws along each dimension. Function is usually used for generating coarse Wiener processes

Inherits from *Rng*

Public Functions

RngMtDownsample(**const** unsigned long int *seed*, **const** double *std*, **const** size_t *dim*, **const** size_t *down_factor*)
Default constructor for *RngMtDownsample*.

double **get**()
Get a single downsampled value from the random number generator.

Private Functions

void **downsample_draw**()

Private Members

std::mt19937_64 **generator**
A Mersenne Twister generator instance.

std::normal_distribution<double> **dist**
A normal distribution instance.

size_t **current_dim**
Stores the current state of the output dimenstion.

std::vector<double> **store**
Stores consecutive random numbers.

const size_t **D**
The number of dimensions required.

const size_t **F**
The number of consecutive random numbers to downsample.

class RngArray
#include <*rng.hpp*> Provides an *Rng* interface to a predefined array of numbers.

Inherits from *Rng*

Public Functions

RngArray(**const** double **arr*, size_t *arr_length*, size_t *stride* = 1)
Default constructor for *RngArray*.

Parameters

- `_arr`: a predefined array of random numbers
- `_arr_length`: length of the predefined array
- `_stride`: the number of consecutive elements to stride for each call to `.get()`

double **get**()
Get the next (possibly stridden) value from the array.

The first call will always return the value at the 0th index. Each subsequent call will stride the array (default value is 1) and return that value. A call that extends beyond the end of the array will result in an error.

Exceptions

- `std::out_of_range`: when a call attempts to get a value beyond the maximum length of the predefined array.

Private Members

unsigned int **i** =0
Internal state.

const size_t **max**
Maximum number of draws available.

const double ***arr**
Pointer to the predefined array of numbers.

const size_t **stride**
Number of values to stride for each draw.

7.2.22 File simulation.cpp

7.2.23 File simulation.hpp

Typedefs

using d3 = std::array<double, 3>

using rng_vec = std::vector<std::shared_ptr<*Rng*>, std::allocator<std::shared_ptr<*Rng*>>>

using sde_jac = std::function<void(double *, double *, double *, double *, **const** double *, **const** double, **const** double)>

namespace simulation

Functions

std::vector<**struct** *results*> **full_dynamics**(**const** std::vector<double> *thermal_field_strengths*, **const** std::vector<double> *reduced_anisotropy_constants*, **const** std::vector<double> *reduced_particle_volumes*, **const** std::vector<*d3*> *anisotropy_unit_axes*, **const** std::vector<*d3*> *initial_magnetisations*, **const** std::vector<std::vector<*d3*>> *interparticle_unit_distances*, **const** std::vector<std::vector<double>> *interparticle_reduced_distance_magnitudes*, **const** std::function<double)**const** double> *applied_field*, **const** double *average_anisotropy*, **const** double *average_volume*, **const** double *damping_constant*, **const** double *saturation_magnetisation*, **const** double *time_step*, **const** double *end_time*, *Rng* &*rng*, **const** bool *renorm*, **const** bool *interactions*, **const** bool *use_implicit*, **const** double *eps*, **const** int *max_samples*

std::vector<*results*> **full_dynamics**(**const** std::vector<double> *radius*, **const** std::vector<double> *anisotropy*, **const** std::vector<*d3*> *anisotropy_axes*, **const** std::vector<*d3*> *magnetisation_direction*, **const** std::vector<*d3*> *location*, **const** double *magnetisation*, **const** double *damping*, **const** double *temperature*, **const** bool *renorm*, **const** bool *interactions*, **const** bool *use_implicit*, **const** double *eps*, **const** double *time_step*, **const** double *end_time*, **const** size_t *max_samples*, **const** long *seed*, **const** *field*::*options* *field_option* = field::CONSTANT, **const** double *field_amplitude* = 0.0, **const** double *field_frequency* = 0.0)

struct *results* **dom_ensemble_dynamics**(**const** double *volume*, **const** double *anisotropy*, **const** double *temperature*, **const** double *magnetisation*, **const** double *alpha*, **const** std::function<double(double)> *applied_field*, **const** std::array<double, 2> *initial_mags*, **const** double *time_step*, **const** double *end_time*, **const** int *max_samples*

void **save_results**(**const** std::string *fname*, **const** **struct** *results* &*res*)

Save results to disk.

Saves the contents of a results struct to disk. Given a file name 'foo' the following files are written to disk 'foo.mx', 'foo.my', 'foo.mz', 'foo.field', 'foo.time', 'foo.energy'.

Parameters

- `fname`: /path/to/filename prefix for files
- `res`: results struct to save

void **zero_results**(**struct** *results* &*res*)

Initialise results memory to zero.

Parameters

- `res`: results struct to zero

void **reduce_to_system_magnetisation**(double **mag*, **const** double **particle_mags*, **const** size_t *N_particles*)

struct results

Public Functions

results(size_t *_N*)

Public Members

std::unique_ptr<double[]> **mx**

std::unique_ptr<double[]> **my**

std::unique_ptr<double[]> **mz**

std::unique_ptr<double[]> **field**

std::unique_ptr<double[]> **time**

size_t **N**

7.2.24 File stochastic_processes.cpp

7.2.25 File stochastic_processes.hpp

namespace stochastic

Functions

void **master_equation**(double **derivs*, **const** double **transition_matrix*, **const** double **current_state*, **const** size_t *dim*)

Evaluates the derivatives of the master equation given a transition matrix.

The master equation is simply a linear system of ODEs, with coefficients described by the transition matrix. $\frac{\mathrm{d}x}{\mathrm{d}t} = Wx$

Parameters

- `derivs`: the master equation derivatives [length dim]
- `transition_matrix`: the [dim x dim] transition matrix (row-major) W
- `current_state`: values of the state vector x

7.2.26 File trap.cpp

Implementation of trapezoidal integration scheme.

Author Oliver W. Laslett (2016)

7.2.27 File trap.hpp

namespace trap

numerical schemes for computing the area under curves

Author Oliver Laslett

Functions

double **trapezoidal**(double **x*, double **y*, size_t *N*)

double **one_trapezoid**(double *x1*, double *x2*, double *fx1*, double *fx2*)

7.3 Class list

7.3.1 Class Rng

class Rng

Abstract class for random number generators.

Subclassed by *RngArray*, *RngMtDownsample*, *RngMtNorm*

7.3.2 Class RngArray

class RngArray

Provides an *Rng* interface to a predefined array of numbers.

Inherits from *Rng*

7.3.3 Class RngMtDownsample

class RngMtDownsample

Generate normally distributed values with downsampling.

Uses the Mersenne Twister to generate normally distributed random numbers. Down-samples the stream of random numbers by summing consecutive draws along each dimension. Function is usually used for generating coarse Wiener processes

Inherits from *Rng*

7.3.4 Class RngMtNorm

class RngMtNorm

Uses Mersenne Twister to generate normally distributed values.

Random numbers have a mean of zero and a user specified standard deviation.

Inherits from *Rng*

7.4 Struct list

7.4.1 Struct mnp::norm_params

struct mnp::**norm_params**

7.4.2 Struct mnp::params

struct mnp::**params**

7.4.3 Struct simulation::results

struct simulation::**results**

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Python Module Index

m

Index

Symbols

A

C

D

E

F

G

I

L

M

O

P

R

S

T

U