Skyrmion Impossible: A Monte Carlo quest for a magnetic quasi-particle

***Mandatory Academic Integrity Declaration form**: https://forms.office.com/e/wSs46LtiwP. You must submit it with your coursework before the deadline. We strongly advise you to familiarise yourself with the form before starting the assessment to ensure you understand the guidelines and policies regarding academic integrity in this assessment.*

Computational science has emerged as the third pillar of research across many science and engineering disciplines [1]. Computational (*in silico*) studies complement experimental and theoretical approaches and are sometimes the only feasible way to address different research and commercial challenges. They become more widespread and reliable as the models, simulation techniques, and computational power advance. We use computational approaches to verify theory, explain and guide experiments, and propose product designs. In nanomagnetism, simulations have become well-established to explore different physics phenomena at nanometre scales and their applications, such as in data storage [2], information processing [3], neuromorphic computing [4], or tumor treatment [5].

Different models with different levels of abstraction and complexity are used in computational nanomagnetism – each with its advantages and disadvantages. In this coursework, we will implement a *Metropolis-Hastings Monte Carlo simulator on a two-dimensional Heisenberg spin-lattice and employ it to find a magnetic skyrmion* [6, 7].

Sounds scary, but it's not! You do not need to have any prior physics knowledge. At the end of the day, this is a coding and not a physics assessment. However, instead of working on an imaginary problem, we chose the topic of this assessment to be a real-world research problem. For example, the code you will develop was used to explain and complement the experimental results in one of the most important skyrmion papers published in *Nature* [8]. Everything physics-related you may need is in this document. However, if you finish earlier or would like to come back to this assessment after the deadline, you can read more about magnetic skyrmions, which have become a topic of intensive research due to their promising properties to revolutionise how we store and process data, in some of the review papers. [9–11]

It sounds like a lot of work, so let's slowly introduce different concepts and give guidance on how to get to the end.

Atomic spin and spin lattice

As children, we played with magnets and experienced attractive or repelling forces between them. Later, in school, we learned that magnet is a *dipole* – it has its north (N) and south (S) pole as we show in Fig. 1 (a). If we attempt to cut a magnet into smaller pieces, as we show in Fig. 1 (b), each piece will again be a dipole with both N and S poles. More precisely, we cannot (or still don't know how to) isolate a true magnetic monopole.

For simplicity, instead of drawing a magnet as a rectangle with marked north (N) and south (S) poles, as a convention, we draw a single vector as we show in Fig. 1 (c). We call that vector the *magnetisation vector* $\mathbf{M}$. Please note that, in this work, we will use boldface, e.g. $\mathbf{A}$, to indicate that a symbol is a vector instead of putting an arrow above the symbol $\vec{A}$. Vector's head (tip of the arrow) corresponds to the north pole (N), whereas its tail corresponds to the south pole (S).

We already said that cutting a magnet into smaller pieces results in magnetic dipoles. Now, think about the smallest magnetic dipole we can cut out. As we show in Fig. 1 (d), if we zoom in enough, we can see that magnetic materials (or at least the ones we are exploring in this coursework) consist of atoms in symmetrical structural arrangement. We call such a symmetrical structural arrangement a *crystal lattice*. Therefore, the smallest magnetic dipole is associated with an individual atom in a crystal lattice, as shown in Fig. 1 (e). We can think about the behaviour of a magnet as a collective behaviour of individual atomic dipoles. Accordingly, the basic building block in our simulations will be an atomic dipole. In literature, people use different names for atomic dipoles, for instance, magnetic moments, atomic moments, or spins. This work will refer to them as *spins*, with symbol $\mathbf{s}$.

Many concepts in magnetism have their roots in quantum physics. We do not have time or ambition to dive into the quantum world as we have to begin coding asap. Therefore, we will use the (simplified and modified for this work) Heisenberg's spin model and summarise it with the following approximations:

- Atoms are arranged in a two-dimensional (2D) lattice as shown in Fig. 2.
- Each atom is fixed in space, i.e. their distance from the nearest neighbours (lattice constant a) is constant in space and time.

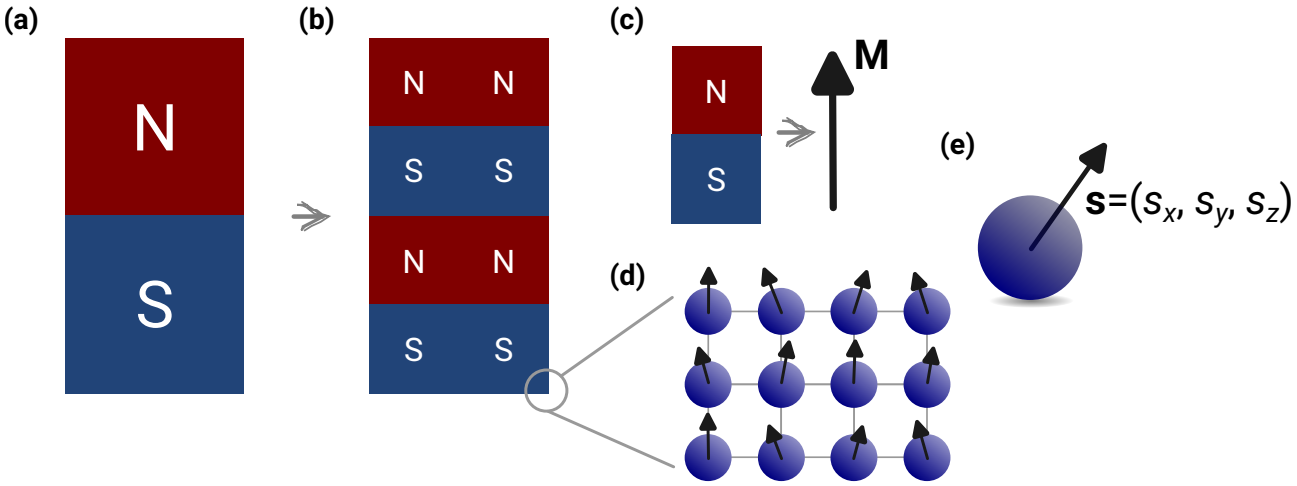

Figure 1: **The concepts of magnetic poles, magnetisation vector, lattice, and spin**. **(a)** Each magnet has a north (N) and a south (S) pole. By cutting the magnet into smaller pieces as shown in **(b)**, we cannot isolate a true magnetic monopole – each piece is again a magnetic dipole with both poles. For simplicity, instead of drawing a picture of a magnet with N and S poles, we draw a magnetisation vector $\mathbf{M}$ with its head and tail corresponding to the north and south poles, respectively. **(d)** If we keep zooming in, we will see that the smallest magnets are atoms arranged in a lattice. **(e)** The basic building block of the Heisenberg model is a spin vector $\mathbf{s}$.

- All atoms are magnetic: each atom has a *spin* $\mathbf{s}$, which is a vector $\mathbf{s} = (s_x, s_y, s_z)$ that can point in any direction, as shown in Fig. 1 (e).
- The magnitude of all spins is constant: $|\mathbf{s}| = 1$.

One of the main tasks of computational magnetism is to explore the collective behaviour of spins, i.e. to find out in what direction each spin in the lattice is pointing to. In Fig. 2, we show an example lattice with $n_x = 5$ and $n_y = 6$ atoms in the x and y directions, respectively. Each atom has its spin $\mathbf{s}_{i,j}$. The upper left spin, we mark with $\mathbf{s}_{0,0}$, whereas the lower right spin is $\mathbf{s}_{(n_x-1,n_y-1)} = \mathbf{s}_{45}$. Hmmm, this can be a bit confusing... Why is the x-axis vertical and y-axis horizontal? As we will see in a bit, we will use a NumPy array to store spins, and NumPy arrays are usually understood as `array[i, j]` – the first index i corresponds to a row and the second j to a column. Similarly, we think about points in space having coordinates (x, y) and not (y, x). Therefore, as a convention, we will think about the rows as the x-axis and columns as the y-dimension.

We will model and implement the spin-lattice in our simulation code using `Spins` class. In `mcsim/spins.py`, you can find the skeleton of the `Spins` class with `__init__` and some other methods already implemented. We can see that the data structure holding values of all individual spins `array` is a NumPy array. Its shape is $(n_x, n_y, 3)$, where n_x (rows) and n_y (columns) are the number of spins in the x and y directions, respectively. In addition, 3 is there because each spin is a vector with three components (s_x, s_y, s_z). For instance, the y-component (s_y) of the second spin from the top and the third from the left, would be `array[1, 2, 1]`.[1] Enough theory (for now)... let's begin coding.

Task 1: `mean` **property**

If we worked in a coffee shop and our boss asked us how well we did today, it's unlikely they would like to hear "Susan had a flat white and a croissant, Yuchen had an espresso, Marijan was grumpy and had nothing, etc.". Instead, they would like to hear a single number (an aggregate) – how much money we earned today. Similary, instead of looking at the individual components of all spins in `array` to inspect what magnetisation state has emerged in our lattice at the end of the simulation, we will look at the *mean*.

In this task, implement the `mean` property in `Spins` class. It should return a sequence (list, tuple, or NumPy array) $[\bar{s}_x, \bar{s}_y, \bar{s}_z]$, where $\bar{s}_c$ for any component $c = x, y, z$ is computed as

$$\bar{s}_c = \frac{1}{n_x n_y} \sum_{i=0}^{n_x-1} \sum_{j=0}^{n_y-1} s_{i,j,c} \tag{1}$$

Tests that could guide you to the solution are in `tests/test_spins.py::TestMean`.

[1] Please note that indexing in Python starts from 0 – instead of `array[2, 3, 2]` we have `array[1, 2, 1]`.

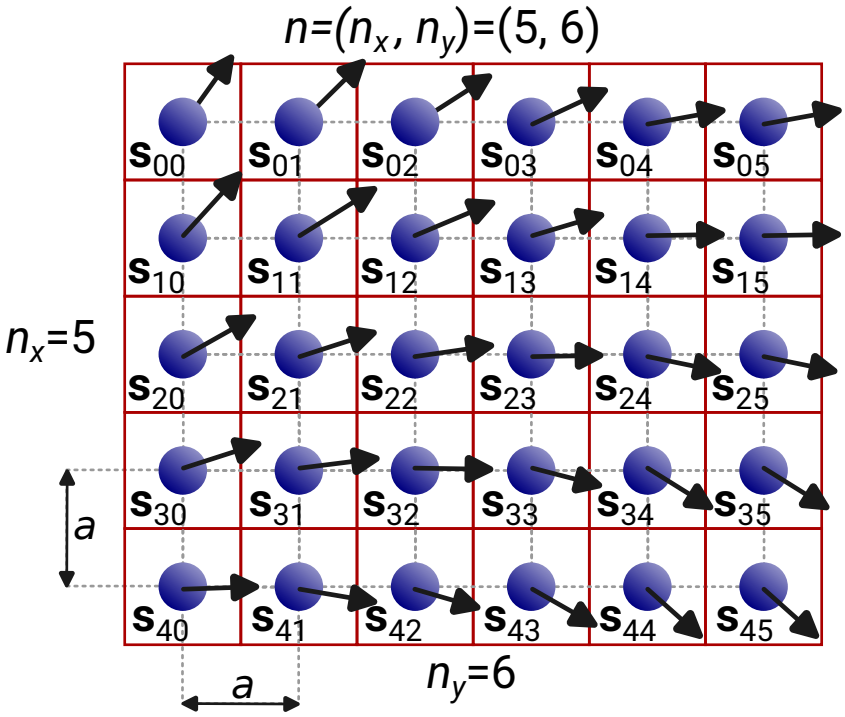

Figure 2: **An example of a two-dimensional lattice of classical Heisenberg spins**. Atoms are arranged in a two-dimensional crystal lattice with lattice constant a. Each atom in the lattice is magnetic and has the spin $\mathbf{s}$, which is a vector that can point in any direction: $\mathbf{s} = (s_x, s_y, s_z)$. The magnitude of all spins is constant in both space and time and equals to 1.

How do I run tests? In this work, we are using `pytest` package to test our code. All tests can be found in `mcsim/tests/test_*.py` files, and if you do not have Pytest installed, you can install it in your conda environment using `pip install pytest`.

We run pytest by typing `pytest` in our repository, or if we want a more detailed (verbose) output, we run `pytest -v`. Pytest will then explore all files in our repository and its subdirectories to find the ones whose filename begins with `test_` and execute all functions/classes that begin with `test` or `Test`.

In each test, we perform a particular set of commands and check their output using an `assert` statement. If the expressions in all `assert` statements result in `True`, our test passes. Otherwise, it fails, and from the output Pytest gives us, we can see why, i.e. we can compare the output we got with the expected output. Please note that even if your tests fail, it does not mean you will get zero points for your implementation – you can get partial points. On the other hand, if the tests are passing, it does not mean you will get the maximum number of points for that task. **You do not have to run tests – they are not mandatory**. Tests are simply there to help you with the code development, indicate if you are getting closer to the solution, and give you some piece of mind. However, you do not have to run them – if you think they are a distraction, just ignore them.

All tests will fail the first time you clone your repository and run `pytest`. This is because some methods rely on others. So, for tests to pass for one method, other methods need to be implemented. As you progress implementing your code, more tests will be passing. Running Pytest requires your code to be packaged, or at least `__init__.py` files to be present in your repository. Because of that, we have already given you those files so that you can run tests from Task 1.

Task 2: `__abs__` special (dunder) method

When we introduced the concept of the simplified and modified Heisenberg spin-lattice model, one of the assumptions was that the magnitude of spins, i.e. the norm, is constant. More precisely, it does not vary in space and time, and it always equals 1. Therefore, one of the operations our `Spins` class should be able to do is to compute the norm of all spins in the lattice. Instead of naming the method, for instance, `norm` or `magnitude`, we will overload `__abs__` operator so that when we call the built-in `abs` function on an object, which is an instance of `Spins` class, that method will be called to calculate the norm (magnitude) of all spins in the lattice.

In this task, implement `__abs__` special (dunder) method, which returns a NumPy array with shape $(n_x, n_y, 1)$. The norm of a spin at location i, j is computed as:

$$|\mathbf{s}_{i,j}| = \sqrt{\mathbf{s}_{i,j,x}^2 + \mathbf{s}_{i,j,y}^2 + \mathbf{s}_{i,j,z}^2} \tag{2}$$

To find out whether your function is returning the expected solutions for some of the test cases, please refer to the tests in `tests/test_spins.py::TestNormalise`.

A good start! We have introduced the concepts of spin and spin-lattice, gave an overview of approximations we will use, and even started coding by attempting to implement two methods to extend the capabilities of the `Spins` class. Let us explore how spins interact with each other and the environment by introducing four energy terms we will use to compute the system's energy (Hamiltonian).

Zeeman energy

The first energy term we will introduce is the Zeeman energy term. We know that if a magnetic dipole (e.g. a magnetic needle in compass) is placed into an external magnetic field (e.g. Earth's magnetic field), it will align itself parallel to the external magnetic field. The energy responsible for this is the *Zeeman energy*.

If we have an spin s, placed into an external magnetic field $\mathbf{B}$, as we show in Fig. 3 (left) we can compute the Zeeman energy as the following dot product:

$$e_\mathrm{z} = -\mathbf{s} \cdot \mathbf{B} \tag{3}$$

Let us now consider how this expression means that the spin s wants to align itself with the external magnetic field $\mathbf{B}$. All physical systems tend to reduce their energy and be in the state of (local) minimum energy. Think of a ball rolling down a hill – it naturally settles at the bottom because that's where it has the least energy and won't roll any further.

If we say that the angle between s and $\mathbf{B}$ is θ, then the Zeeman energy would be $e_\mathrm{z} = -|S||B|\cos\theta$. For what angle θ does the Zeeman energy have the minimal value? Due to the minus sign in front, it will be for θ for which $\cos\theta$ has the maximum value of 1. Accordingly, for $\theta = 0$, Zeeman energy is at its minimum. Aha, that's why Zeeman energy wants to align the spin parallel (in the same direction) as the external magnetic field $\mathbf{B}$ as we show in Fig. 3 (right).

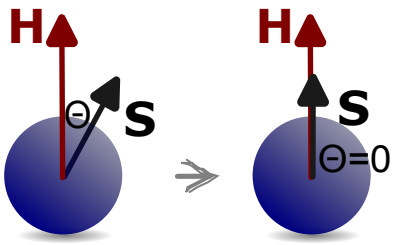

Figure 3: **Zeeman energy.** The Zeeman energy of an individual spin s in an external magnetic field $\mathbf{B}$ is $e_\mathrm{z} = -\mathbf{s} \cdot \mathbf{B}$. Therefore, Zeeman energy tends to align the spin parallel to the external magnetic field.

We compute the Zeeman energy for the entire two-dimensional lattice as the sum of energies of all individual spins.

$$E_\mathrm{z} = -\sum_{i=0}^{n_x-1}\sum_{j=0}^{n_y-1} \mathbf{s}_{i,j} \cdot \mathbf{B} \tag{4}$$

Task 3: `zeeman` **method**

The arrangement of spins, together with all the parameters we need to compute the total energy of the system will be provided by the user when they define an object of `System` class. You can find the `System` class in `mcsim/system.py`. In this task, implement `zeeman` method of the `System` class. This method should return the total Zeeman energy of the system as a `float` computed using Eq. 4. If we have a look at the `__init__` method of the `System` class, we can see that the attributes we need inside the `zeeman` method are `self.B` and `self.s.array`.

The relevant tests for this implementation are in `tests/test_system.py::TestZeeman`.

Uniaxial anisotropy energy

When atoms are arranged in a lattice, most often, unless the material is isotropic, certain directions are preferential for the spins to be aligned to. The energy "responsible" for that alignment is called the anisotropy energy. Different types of anisotropy exist, each with a different expression for computing the total energy. However, in this work, we will implement the *uniaxial anisotropy energy*. It tends to align spins $\mathbf{s}$ parallel or anti-parallel to the anisotropy vector $\mathbf{u}$, as we show in Fig. 4, and for a single spin $\mathbf{s}$, the energy is

$$e_{\mathrm{a}} = -K(\mathbf{s} \cdot \mathbf{u})^2 \tag{5}$$

where K is the anisotropy constant, which depends on the material we are simulating, and $\mathbf{u}$ is the anisotropy axis. Often, we assume that the anisotropy axis is a unit vector (its magnitude is 1): $|\mathbf{u}| = 1$. We can compare Eq. 3 and Eq. 5 and notice that the main difference is in the square of the dot product. Accordingly, unlike Zeeman energy, which tends to align all spins parallel to the external magnetic field, uniaxial anisotropy energy wants spins aligned parallel or anti-parallel to $\mathbf{u}$.

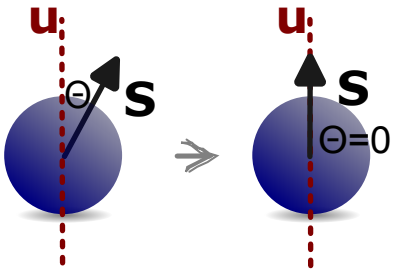

Figure 4: **Uniaxial anisotropy energy**. The uniaxial anisotropy energy of an individual spin $\mathbf{s}$ with anisotropy axis $\mathbf{u}$ and the anisotropy constant K is $e_{\mathrm{a}} = -K(\mathbf{s} \cdot \mathbf{u})^2$. Accordingly, uniaxial anisotropy energy tends to align the spin parallel or anti-parallel to the anisotropy axis $\mathbf{u}$.

The total uniaxial anisotropy energy of the entire system is

$$E_{\mathrm{a}} = -K \sum_{i=0}^{n_x-1} \sum_{j=0}^{n_y-1} (\mathbf{s}_{i,j} \cdot \mathbf{u})^2 \tag{6}$$

Task 4: `anisotropy` **method**

Implement `anisotropy` method of the `System` class. This method should return the total anisotropy energy of the system as a `float` computed using Eq. 6. Uniaxial anisotropy parameters K and $\mathbf{u}$ will be provided by the user when they define the `System` object. However, as we mentioned previously, we assume that the norm (magnitude) of $\mathbf{u}$ is normalised to 1 before computing the energy ($|\mathbf{u}| = 1$). Therefore, inside the `anisotropy` method, ensure that $\mathbf{u}$ is normalised before computing the energy. If we have a look at the `__init__` method of the `System` class, the attributes we need inside the `anisotropy` method are `self.K`, `self.u`, and `self.s.array`.

The relevant tests for this implementation are in `tests/test_system.py::TestAnisotropy`.

Exchange energy

So far, we explored the Zeeman and uniaxial anisotropy energy terms. Those energies are "local" – spins tend to align to an external magnetic field or the anisotropy axis, independent of the neighbouring spins in the lattice. In this work, we will also implement two energy terms that decide on the spin's preferential direction based on the spins of the nearest neighbours.

The first one is the *exchange* energy. If we have two neighbouring spins $\mathbf{s}_1$ and $\mathbf{s}_2$ (isolated or in a lattice), the exchange energy between them is

$$e_{\mathrm{ex}} = -J\mathbf{s}_1 \cdot \mathbf{s}_2 \tag{7}$$

where $J > 0$ is the exchange energy constant, which depends on the material we are simulating. From this expression, we can see that, since $J > 0$, exchange energy between spins $\mathbf{s}_1$ and $\mathbf{s}_2$ will be at its minimum when they are parallel to each other – they are pointing in the same direction, as we show in Fig. 5. Exchange energy does not introduce any preferential direction like Zeeman and anisotropy energies. It only wants to align all spins parallel to each other.

We already mentioned that the exchange energy is a short-range energy - each spin tends to be parallel to its *nearest neighbours*. To clarify, if we look at the lattice of spins we showed in Fig. 2, as an example, the nearest

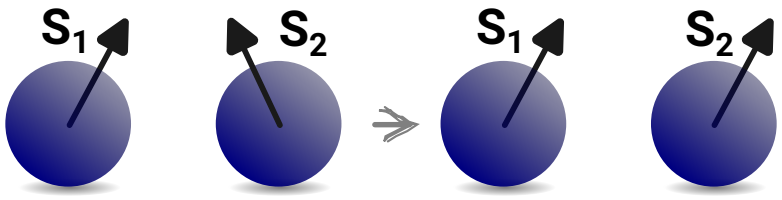

Figure 5: **Exchange energy**. The exchange energy between two spins $\mathbf{s}_1$ and $\mathbf{s}_1$ is $e_{\text{ex}} = -J\mathbf{s}_1 \cdot \mathbf{s}_2$, where $J > 0$ is the exchange energy constant. Therefore, exchange energy tends to align neighbouring spins parallel to each other and does not have a preferential direction.

neighbours of the spin $\mathbf{s}_{1,1}$ are $\mathbf{s}_{0,1}$, $\mathbf{s}_{1,0}$, $\mathbf{s}_{1,2}$, and $\mathbf{s}_{2,1}$. Spins on the diagonal of the unit cell, e.g. $\mathbf{s}_{0,2}$, are not the nearest neighbours. Therefore, we compute the total exchange energy as the sum of exchange energies between all nearest neighbours:

$$E_{\text{ex}} = -J\left[\sum_{i=0}^{n_x-1}\left(\sum_{j=0}^{n_y-2}\mathbf{s}_{i,j}\cdot\mathbf{s}_{i,j+1}\right) + \sum_{j=0}^{n_y-1}\left(\sum_{i=0}^{n_x-2}\mathbf{s}_{i,j}\cdot\mathbf{s}_{i+1,j}\right)\right] \tag{8}$$

Task 5: `exchange` **method**

Implement `exchange` method of the `System` class. This method should return the total exchange energy of the system as a `float` computed using Eq. 8. A user provides an exchange energy parameter $J > 0$ when they define the `System` object. The attributes of the `System` class we need to implement `exchange` method are `self.J` and `self.s.array`.

The relevant tests for this implementation are in `tests/test_system.py::TestExchange`.

Dzyaloshinskii-Moriya (DMI) energy

Although this energy term has been known for over 50 years, it became a celebrity only after it was predicted that it could give rise to non-trivial magnetisation states in nanomagnetic systems, such as magnetic skyrmions [6]. Like the exchange energy, this term is short-range, and the direction a spin wants to align depends on its nearest neighbours.

If we have two neighbouring spins $\mathbf{s}_1$ and $\mathbf{s}_2$, the Dzyaloshinskii-Moriya (DMI) energy between them will be

$$e_{\text{dmi}} = \mathbf{D} \cdot (\mathbf{s}_1 \times \mathbf{s}_2) \tag{9}$$

where $\mathbf{D}$ is the Dzyaloshinskii-Moriya vector, and it depends on the material we are simulating. Unlike the exchange energy we introduced previously, the material parameter $\mathbf{D}$ is a vector. From the energy expression for the two spins, we can see that this energy is minimal when spins $\mathbf{s}_1$ and $\mathbf{s}_2$ are perpendicular to each other (due to the cross product) and in the plane that is perpendicular to $\mathbf{D}$, as we show in Fig. 6. Therefore, the DMI energy tends to align spins perpendicular to its neighbours.

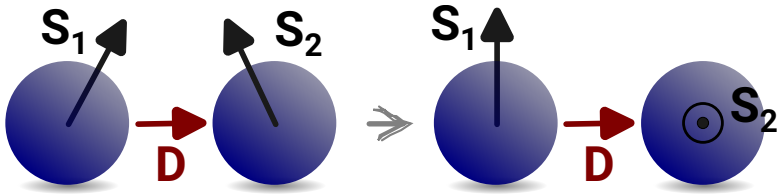

Figure 6: **Dzyaloshinskii-Moriya (DMI) energy**. The DMI energy between two spins $\mathbf{s}_1$ and $\mathbf{s}_1$ is $e_{\text{dmi}} = \mathbf{D}\cdot(\mathbf{s}_1\times\mathbf{s}_2)$, where $\mathbf{D} = D\mathbf{r}_{ij}$ is the DMI vector. DMI vector in this work is a vector with magnitude D (DMI parameter) and in the direction $\mathbf{r}_{12}$, where $\mathbf{r}_{12}$ is a unit vector from $\mathbf{s}_1$ to $\mathbf{s}_2$. DMI energy tends to align neighbouring spins perpendicular to each other and in the plane that is perpendicular to vector $\mathbf{D}$.

The Dzyaloshinskii-Moriya energy term occurs due to the magnetic systems' lack of inversion symmetry. This is either due to the non-centrosymmetric crystal lattice or because we have introduced asymmetry by stacking layers of different materials. In this work, we will consider the DM vector to be the consequence of the non-centrosymmetric lattice, and in that case, the DMI vector between two nearest neighbouring spins $\mathbf{s}_1$ and $\mathbf{s}_2$ is

$$\mathbf{D}_{12} = D\mathbf{r}_{12} \tag{10}$$

where scalar $D > 0$ is the DMI constant and $\mathbf{r}_{12}$ is a unit vector ($|\mathbf{r}_{12}| = 1$) pointing from $\mathbf{s}_1$ to $\mathbf{s}_2$. The unit vector $\mathbf{r}_{12}$ is independent of the values of spins $\mathbf{s}_1$ and $\mathbf{s}_2$. It only depends on the positions of spins $\mathbf{s}_1$ and $\mathbf{s}_2$ in the spin-lattice.

Accordingly, the total DMI energy for the entire two-dimensional lattice is

$$E_{\text{dmi}} = \sum_{i=0}^{n_x-1} \left(\sum_{j=0}^{n_y-2} \mathbf{D}_{i,j}^{i,j+1} \cdot (\mathbf{s}_{i,j} \times \mathbf{s}_{i,j+1}) \right) + \sum_{j=0}^{n_y-1} \left(\sum_{i=0}^{n_x-2} \mathbf{D}_{i,j}^{i+1,j} \cdot (\mathbf{s}_{i,j} \times \mathbf{s}_{i+1,j}) \right) \tag{11}$$

where, for example, $\mathbf{D}_{i,j}^{i,j+1}$ is a DMI vector between spins $\mathbf{s}_{i,j}$ and $\mathbf{s}_{i,j+1}$ computed using Eq. 10.

Task 6: `dmi` **method**

Implement the `dmi` method of the `System` class. This method should return the total Dzyaloshinskii-Moriya energy of the system as a `float` computed using Eq. 11. DMI energy parameter $D > 0$ is provided by a user when they create the `System` object. The attributes we need to implement `dmi` method are `self.D` and `self.s.array`. Please note that `self.D` is a scalar and not a vector. Therefore, the vector $\mathbf{D}_{ij}$ should be implemented implicitly in the calculation of the DMI energy.

The relevant tests for this implementation are in `tests/test_system.py::TestDMI`.

Metropolis-Hastings Monte Carlo algorithm

We have now introduced and implemented the energy terms we need. So, for any spin configuration, we can compute the system's total energy as the sum of all individual energy terms by calling the `energy` method of the `System` class. Please note that the system's total energy can be positive or negative. We can think about it as the potential energy in mechanics $\Pi = mgh$, where m is the mass, g the gravitational constant, and h is the height. But height with respect to what? The floor, the ground floor, the basement, the ceiling, or something else? We can choose the reference level arbitrarily. For instance, if we are calculating the potential energy with respect to the desk we are working on, we will get a positive value if the object is above the desk. On the other hand, if the object is sitting on the floor, we will get a negative potential energy because the height is negative – the object is under the desk.

We will introduce the Metropolis-Hastings Monte Carlo algorithm to minimise the system's energy and find the configuration of spins for which the energy is at a (local) minimum. We call such configurations (or states) equilibrium states. For simplicity, we will assume we are minimising the energy of our system at zero temperature ($T = 0\,\text{K}$).

The *Metropolis-Hastings Monte Carlo algorithm* includes the following steps:

1. Compute the total energy of the system E_0. In this step as well as in step 4, we compute the total energy by calling the `energy` method in the `System` class. The total energy we are computing is the sum of all four energy terms we implemented previously.

2. Randomly select one spin in the lattice. Randomly selected spin $\mathbf{s}_{i,j}$ must be drawn from a uniform distribution, i.e. each spin must have the same chance to be selected.

3. Randomly change the spin's direction to $\mathbf{s}'_{i,j}$. To perform this step, use function `random_spin(s0, alpha=0.1)` in `mcsim/driver.py`, where `s0` is the original spin you are changing, and `alpha` is a measure by how much you are changing the spin. Although chaging a spin sounds like a simple task, in reality, it is much more complicated. For this work, we are happy with this change. However, please feel free to explore the topic further if you are interested and modify `random_spin` function accordingly.

4. Compute the total energy E_1 of the system with modified spin $\mathbf{s}'_{i,j}$.

5. If the energy of the system after a random modification of a spin has dropped ($\Delta E = E_1 - E_0 < 0$), accept the changed spin and go back to step 1. On the other hand, if the total energy of the system has increased ($\Delta E = E_1 - E_0 \geq 0$), reject the change. In other words, return the original value of the spin and go back to step 1.

We repeat those steps n times. Usually, depending on the size of the system we are simulating, n varies. In this work, we will use the values of n on the order of $10^5 - 10^6$.

Task 7: `drive` **method**

Implement the `drive` method of the `Driver` class. This method accepts an object which is an instance of `System` as an input parameter and the number of iterations `n`. It does not return anything, but it modifies the `system` object passed to it. To access spins of the system, you need to access `system.s.array`. Similarly, to compute the system's energy, you will be calling `system.energy()` inside the `drive` method.

The relevant tests for this implementation are in `tests_test_driver.py`. Metropolis-Hastings Monte Carlo is a random algorithm. The final state of our energy minimisation (also called relaxation) will strongly depend on the random initial state, the number of steps, etc. Therefore, it might happen that some of the tests in `test_driver.py` occasionally fail since we are working on a real-world problem and might end up in different local energy minimum. Therefore, if you would like to mitigate this, you are allowed to increase the number of steps (`n`) when calling the `drive` method. You cannot modify any of the tolerances in tests.

In addition, it may take some time for the driver's tests to run. This is because we run the Metropolis-Hastings Monte Carlo algorithm for a large number of iterations. Only later, when you tackle the optimisation task, you will be able to speed up the algorithm and your tests will run faster. Therefore, either be patient with driver's tests or wait to optimise the code and then run the tests.

Visualisation

We have now implemented all the necessary functionality to minimise the energy of a two-dimensional spin-lattice. We will now implement the `plot` method in the `Spins` class, which we will use to visualise the spin lattice.

Task 8: `plot` **method**

You may have noticed that we missed implementing the `plot` method in `Spins` class. We will do it now. You have the freedom to design and implement the `plot` method to visualise the lattice of spins so that

- It is called by running `system.s.plot()` inside Jupyter notebook.
- You can add as many *keyword* arguments to the `plot` method. However, we should be able to call the `plot` method without passing any parameters. Therefore, please ensure the default values of keyword arguments are the ones you would like to see your plots with.
- You can only use `matplotlib`, including `mpl_toolkit`, as a plotting package when implementing your function.

There are no tests associated with this method – comparing plots to a reference solution is not really that simple. However, please ensure your implementation of the `plot` method works by running and saving the notebook in `my-research/magnetic-skyrmion.ipynb`. The result of your visualisation should be saved as the output of the last cell in that notebook.

Optimisation

While you were working on individual tasks, you might have already employed different optimisation techniques. In the Metropolis-Hastings Monte Carlo algorithm we implemented, each iteration should be performed as fast as possible because, at the end of the day, we need to run it in many iterations.

Task 9: Optimisation

In this task, optimise the code you have written and make the necessary changes to reduce the wall time. You are not required to modify any of the code we gave you. You do not need to submit any of the before or after profiling results.

Documentation

Task 10: Documentation

We have already discussed the importance of documentation, so let us introduce a task in which you can document your code. In this task, document the code you have written. Please ensure that the documentation is clear, concise, and informative.

Packaging

Let us ensure our simulation code is packaged for use in our research and for others who want to perform Metropolis-Hastings Monte Carlo simulations.

Task 11: Packaging

In this task, package your simulation code. Please pay attention to the following points

- The package name is `mcsim`.
- Modify any `__init__.py` files if necessary.
- Write `setup.py` or `pyproject.toml` file so that `mcsim` package can be installed with

  ```
  pip install .
  ```

- All dependencies – libraries required to run `mcsim` package – must be specified in `setup.py` or `pyproject.toml` file.
- Although the name of the package must not be modified, please choose freely the values of other package's metadata.

We will install and run your code in a well-defined way, so please ensure your code is well packaged so that the following steps can be executed:

1. We create a clean conda environment with Python 3.12 (`conda create -n mcsim python=3.12`) and activate it (`conda activate mcsim`).
2. We clone your repository by running `git clone url_to_your_github_repository`
3. We install `mcsim` package – first we navigate into your repository (`cd your-repository-name`) and then "pip-install" it (`pip install .`). Please note that we will run the `pip install .` command in the directory where `mcsim` directory and `setup.py` or `pyproject.toml` are.
4. We run tests with `pytest` in the same directory. The tests will include the ones we gave you and the additional ones we may include.

If you would like to see if your package can be installed and run correctly, you can check the GitHub Actions workflow in your repository. We have set up a workflow that will install and run your package in the same way we will do it. If the workflow is passing, it is very likely that your code is correctly packaged and the tests will pass.

Magnetic skyrmion

We have now implemented everything we need to run a simulation and attempt to find a magnetic skyrmion.

Task 12: Magnetic skyrmion

In `my-research/magentic-skyrmion.ipynb` Jupyter notebook, you will find the simulation workflow. In this task, run all cells in the simulation workflow notebook, save it, and submit it to your repository. Please ensure that the notebook is saved with the output for all cells and pushed to your repository – you don't want to lose easy points.

Please note that although the main objective of this coursework was to find a magnetic skyrmion, running `my-research/magentic-skyrmion.ipynb` Jupyter notebook might not result in one even if you implemented all methods correctly. Metropolis-Hastings Monte Carlo simulation algorithm includes randomness, and we might end up in a local minimum equilibrium state in the energy landscape that does not correspond to a magnetic skyrmion. If you wonder what a magnetic skyrmion in a $T(O)$ crystallographic class you are simulating looks like, we show one in Fig. 7. If you found a skyrmion, you would see a similar configuration in your visualisation of the lattice of spins.

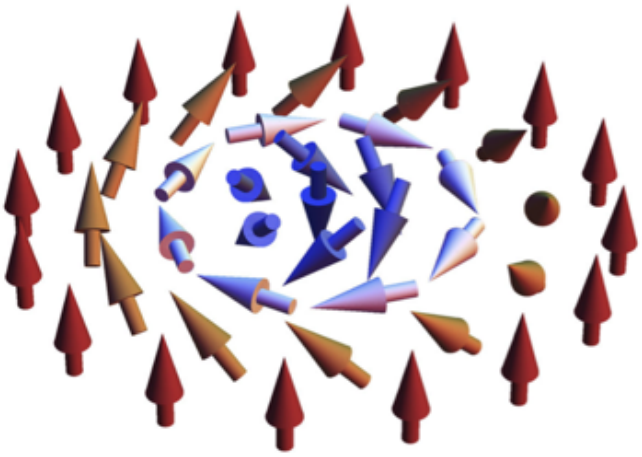

Figure 7: **Magnetic skyrmion**. An example of a magnetic skyrmion in a $T(O)$ crystallographic class. The central spin is pointing in the opposite direction to the surrounding spins.

Points

The maximum number of points for each task is given in Table 1.

Task	points
Task 1 – `mean`	5
Task 2 – `__abs__`	6
Task 3 – `zeeman`	6
Task 4 – `anisotropy`	7
Task 5 – `exchange`	8
Task 6 – `dmi`	12
Task 7 – `drive`	14
Task 8 – `plot`	12
Task 9 – optimisation	10
Task 10 – documentation	8
Task 11 – packaging	8
Task 12 – magnetic skyrmion	4
Total	100

Table 1: **Maximum number of points for each task.**

Important bits

- **GitHub username:** Before accepting the assessment using the GitHub Classroom link, you must ensure that your GitHub username is `esemsc-YOURIMPERIALUSERNAME`. For instance, if your Imperial username is `ab1234`, then your GitHub username must be `esemsc-ab1234`. If for some reason you cannot change your GitHub username, email Marijan with the explanation and to discuss the next steps.
- **The deadline** is **Friday, 18 October 2024, 16:30 BST**. Please ensure everything is committed and pushed to your assessment repository before the deadline.
- **You must submit the Academic Integrity Declaration form with your submission**. Submitting this form is not optional and the deadline to submit it is the same as the deadline for the coursework. We strongly advise you to familiarise yourself with the form before you begin working on your coursework to ensure you understand the guidelines and policies regarding academic integrity in this assessment. **Use the following link**: https://forms.office.com/e/wSs46LtiwP
- Academic Misconduct is taken very seriously and dealt with according to the Academic Misconduct Policy and Procedure.

- Declare all references (articles, webpages, books, etc.) you consulted and used in `references.md`. For each reference, clearly explain what idea or code you have taken. In addition, cite appropriately in your code using comments whenever necessary.
- There will be **optional Q&A sessions** on Wednesday (16 October) and Thursday (17 October) at 16:45 BST. In person/live attendance of these sessions is not mandatory – they will be streamed and recorded.
- All questions related to this assessment **must be posted to the "Assessment Two" channel** so that all students can see the answers and potentially benefit from them. Only if your question or concern is of a personal nature, you can send it directly to the teaching staff.
- All **deadline extensions** due to mitigating circumstances must be discussed and agreed with the PGT Senior Tutor, James Percival.
- **Backup, backup, backup!** Commit and push to your repository as often as possible. This will help you ensure you have the backup of your work on GitHub and allow you to practice `git` and GitHub. Only the final submitted version will be marked, so don't be shy pushing to your repo any intermediate (unpolished steps). The loss of work due to a technical issue or a mistake is normally not a mitigating circumstance.
- You **cannot change any class or function signatures**, except for the `plot` function in `Spins` class. You can add as many keyword arguments as you like to the `plot` function, but ensure that the default values of keyword arguments are the ones you would like to see your plots with.
- You are free to introduce additional functions or classes if they help you with the implementation of tasks.
- Nothing is perfect. Although we invested considerable effort to ensure this assessment document does not contain typos or mistakes, some are probably still be there. If you believe you spotted one, please let us know on the Assessment Two channel as soon as possible, and we will clarify it for you.
- If you get an email from GitHub telling you that the workflow is failing, this is because we set up GitHub Actions in your repository, and all tests are run each time you push. To pass that workflow, you need to ensure your code is correctly packaged and all tests are passing.
- **This is your project** and you can do what you think is necessary as long as the basic requirements are being followed. Therefore, if you believe that doing something we have not explicitly asked you to do would benefit your project, demonstrate your knowledge and skills, and possibly increase your mark, do it.

References

1. Skuse, B. The third pillar. *Physics World* **32,** 40. `https://dx.doi.org/10.1088/2058-7058/32/3/33` (Mar. 2019).
2. Parkin, S. & Yang, S.-H. Memory on the racetrack. *Nature Nanotechnology* **10,** 195–198. `https://doi.org/10.1038/nnano.2015.41` (Mar. 2015).
3. Zhang, X., Ezawa, M. & Zhou, Y. Magnetic skyrmion logic gates: conversion, duplication and merging of skyrmions. *Scientific Reports* **5.** `https://doi.org/10.1038/srep09400` (Mar. 2015).
4. Yokouchi, T. *et al.* Pattern recognition with neuromorphic computing using magnetic field–induced dynamics of skyrmions. *Science Advances* **8.** ISSN: 2375-2548. `http://dx.doi.org/10.1126/sciadv.abq5652` (Sept. 2022).
5. Li, W., Liu, Y., Qian, Z. & Yang, Y. Evaluation of Tumor Treatment of Magnetic Nanoparticles Driven by Extremely Low Frequency Magnetic Field. *Scientific Reports* **7.** `https://doi.org/10.1038/srep46287` (Apr. 2017).
6. Rößler, U. K., Bogdanov, A. N. & Pfleiderer, C. Spontaneous skyrmion ground states in magnetic metals. *Nature* **442,** 797–801. `https://doi.org/10.1038/nature05056` (Aug. 2006).
7. Mühlbauer, S. *et al.* Skyrmion Lattice in a Chiral Magnet. *Science* **323,** 915–919. `https://doi.org/10.1126/science.1166767` (Feb. 2009).
8. Yu, X. Z. *et al.* Real-space observation of a two-dimensional skyrmion crystal. *Nature* **465,** 901–904. ISSN: 1476-4687. `http://dx.doi.org/10.1038/nature09124` (June 2010).

9. Fert, A., Reyren, N. & Cros, V. Magnetic skyrmions: advances in physics and potential applications. *Nature Reviews Materials* **2.** https://doi.org/10.1038/natrevmats.2017.31 (June 2017).
10. Nagaosa, N. & Tokura, Y. Topological properties and dynamics of magnetic skyrmions. *Nature Nanotechnology* **8,** 899–911. https://doi.org/10.1038/nnano.2013.243 (Dec. 2013).
11. Wiesendanger, R. Nanoscale magnetic skyrmions in metallic films and multilayers: a new twist for spintronics. *Nature Reviews Materials* **1.** https://doi.org/10.1038/natrevmats.2016.44 (June 2016).