System of interacting spins: Code documentation

Michael Negus
Diamond Light Source Ltd.
m.negus@warwick.ac.uk

September 16, 2016

Contents

1 Introduction

This documents provides the documentation for the code I have produced during my summer placement at Diamond Light Source Ltd in 2016. The aim of my project was to test a range of heuristic solvers for their effectiveness at solving physics problems. The code here is part of what the solvers were tested against, which describes a system of interacting spins on a lattice.

There are two separate pieces of code: One describes a two-dimensional lattice of two-dimensional spin vectors, the other describes three-dimensional lattice of three-dimensional spin vectors. Although the two-dimensional lattice is what the solvers were tested against, the three-dimensional lattice is much more interesting, and in fact you can reproduce everything the two-dimensional lattice does using it. As a result, this document mainly focuses on the three-dimensional code, not only because it is much better written than the two-dimensional code.

If you have any questions, please do not hesitate to email me at: m.negus@warwick.ac.uk

1.1 What you need to read

You will definitely need to read section **2** in order to install the relevant packages.

Most likely, you want to use the code to create plots of some of the interesting materials, like $FeBO_3$. In that case, you also need to read section **3**, which will give you the bare essentials of the code and how to plot some given example models.

If you are interested in creating your own models, then section **4** gives a description of what each module does. You will need to understand all of these in order to know enough to create your own models.

If, for some reason, you want to look at the two-dimensional code, then the appendix **A** gives that, although there is a lot less to say about it than the three-dimensional code.

1.2 How you should use this document

This document is documentation for my code, but does not make any sense without the code! When this document mentions a module, class, function etc, you should have that piece of code open next to you, so you can check to see if you get what this document is talking about. You won't understand anything from just reading this document, and then looking at the code later. Dive in!

1.3 What you should know

Using this code does not necessarily require any knowledge of physics, however it certainly helps! If you don't, fear not, as when I started writing this code I had pretty much zero experience with condensed matter physics. I recommend reading my report, which you can find on https://github.com/MNegus/SpinModel3D, which gives an explanation of the background to this code.

2 Installation guide

2.1 Getting the code

The three-dimensional is available as a GitHub repository at https://github.com/MNegus/SpinModel3D, clone the master. "model3D" is the package for the three-dimensional code. The two-dimensional code is at https://github.com/MNegus/SpinModel2D, with the package being "model2D".

2.2 Operating system

The code was written on a Red Hat Enterprise version of Linux. The code is yet to be tested on different operating systems other than Linux, although theoretically it could work. The main trouble will be installing the Ipopt solver, which the three-dimensional code relies on. If you want to make your life as easy as possible, I recommend using a Linux machine.

2.3 Python

Having Python 2.7 installed is essential, which if you are using a Linux machine, will already be installed. A scientific distribution of Python, such as Anaconda, is recommended as it will have NumPy already installed. If not, then you definitely need to install NumPy.

2.4 Pyomo

Pyomo is the key package to this code. It is a Python-based algebraic modelling language and provides the interface with the Ipopt solver. Installation instructions are provided on the Pyomo website here: http://www.pyomo.org/installation/. When running the code, you need to ensure Pyomo is in your Python path. Installing Pyomo into your science distribution, such as Anaconda, is the easiest way to do this.

2.5 Ipopt

Ipopt is a solver used by both the three-dimensional and two-dimensional code. Installing it is not the most simple thing in the world, but their website gives step-by-step instructions which are very helpful: http://www.coin-or.org/Ipopt/documentation/node10.html.

I recommend installing in a "software" directory in your local area, and then ensuring you have

```
your_software_dir/CoinIpopt/build/bin
```

in your PATH.

Third party code has to be installed, as said in the installation instructions for Ipopt. Luckily, there are scripts installed for the essentials, which are ASL, Blas, Lapack, Metis and Mumps. Other third party code can be installed, but it is not essential and makes things more complicated.

2.6 Two-dimensional code only

If you want to use the two-dimensional code, then you also need to install SciPy and pyOpt. SciPy will already be installed in a scientific distribution of Python.

pyOpt is an optimisation package for Python, and there are installation instructions here: http://www.pyopt.org/install.html. Once installed, make sure where you installed it is in your PATH.

3 Getting started

This section gives an overview on how to work the code, using pre-written models to help you get straight into plotting some results.

3.1 Importing relevant modules

First of all you'll need to make sure you've installed the relevant packages as described in section **2**. You'll want the package "model3D" in the directory where your Python scripts are, or somewhere in your PATH.

Start by creating a new Python file. At the top, you'll want to import some of the modules, so put:

```
import model3D.model as model
import model3D.vectors as vectors
import model3D.plotting as plotting
import model3D.examples as examples
```

So `model` is the module containing the class `SpinModel`, which is the object which all the information regarding the three-dimensional models lies. `vectors` contains the class `PyomoVector`, which is used for any vectors in code. `plotting` is the module used to plot the lattice and `examples` is a module which contains some functions to make example models.

3.2 Creating models

Next, you want to create a model. The hard work has been done for you, so all you have to do is call a function. However, it is important you understand how to use vectors. The class `PyomoVector` in `vectors` is used to define a vector. For example, if you want to define the vector $\mathbf{v} = (1, 0, 1)$, then you can define it in terms of Cartesian components by calling:

```
my_vector = vectors.PyomoVector(cart_comp=(1, 0, 1))
```

Similarly, you can define the same vector in terms of spherical polar components. Its length is ≈ 1.41, the azimuth angle is 0 and the polar angle is ≈ 0.79, so you can define the vector as:

```
my_vector = vectors.PyomoVector(sph_comp=(1.41, 0, 0.79))
```

The usual operations on vectors can be done too, for more details, look at the code and the rest of this document. The `vectors` module also has some constants for commonly used vectors, such as the unit vectors in the x, y and z directions.

3.2.1 Simple cubic

This is the model for a cubic lattice, with only spin-coupling interactions of each spin with its closest neighbours, and a uniform magnetic field. Basically, you can create a ferromagnet or an anti-ferromagnet.

The function `examples.simple_cubic_model` is called to create the model. The first term, `grid_dimen`, is a tuple giving the dimensions of the grid. `spin_coup_term` sets the coupling term for the spins with their neighbours, and `mag_vector` is a `PyomoVector` for the magnetic field.

So, if you wanted a ferromagnet (where `spin_coup_term` is positive), and a magnetic field along the x-axis, you could write:

```
my_model = examples.simple_cubic_model((4, 4, 4), spin_coup_term=1.0,
↪  mag_vector=PyomoVector(cart_comp=(0.1, 0.0, 0.0)))
```

3.2.2 $FeBO_3$

This is a model for the weak ferromagnet $FeBO_3$, where each spin couples to its nearest neighbours, which are in the plane above and below it. There is spin coupling, Dzyaloshinskii-Moriya interactions, strong anisotropy keeping the spins in the plane and an overall magnetic field.

The function `examples.febo3_model` is called to create the model. The first argument, `grid_dimen`, is a tuple giving the dimensions of the grid. The other argument is `mag_vector`, which is the `PyomoVector` of the magnetic field.

So, if you wanted to create a $10 \times 10 \times 10$ grid, with a magnetic field along the y-axis, you could write:

```
my_model = examples.febo3_model((10, 10, 10),
↪  mag_vector=PyomoVector(cart_comp=(0.0, 100.0, 0.0)))
```

3.2.3 Skyrmions

This is a model for a single-layer hexagonal lattice where magnetic skyrmions can exist. Skyrmions appear due to the magnetic field and the Dzyaloshinskii-Moriya interaction of the spins. The model is such that the spins are given random initial conditions, which results in different patterns of skyrmions each time.

The function `examples.skyrmion_model` is used to create the model. It only takes one input, which is `grid_dimen`, the dimensions of the grid in a tuple. As this is a single-layer lattice, only two dimensions are needed.

So, if you wanted to create a 40×40 grid, you could write:

```
my_model = examples.skyrmion_model((40, 40))
```

3.3 Solving the models

Once you have defined your model, you will need to solve it. This is where the Ipopt solver is called and a solution to the model is found. All you need to do is call the `.solve()` method on your model, for example:

```
my_model.solve()
```

It is worth noting that this can sometimes take a long time, so do not be worried if the code appears to have done nothing in a few minutes.

3.4 Plotting

Once you have solved the model, you will no-doubt want to do some plotting so you can see the results! The results are plotted using matplotlib on a three-dimensional grid.

To plot a model, all you have to do is call the `.plot()` method on your model, like this:

```
my_model.plot()
```

This will plot the entire grid, where all of the spin vectors are blue. There are some ways you can customise how you plot. If you want to restrict which Cartesian components you plot, you can use the `x_limits`, `y_limits` and `z_limits` key-word arguments. These are tuples which give the limits of what you plot. For example, if you only wanted to plot the model with x-components between 1 and 10, you could write:

```
my_model.plot(x_limits=(1, 10))
```

You also have the option to toggle whether or not you want to plot the points where the spins are as little circles, by writing `plot_points=True` or `False`. These can be useful in small plots to see where the layers are, but can make things hard to see when the plots are large.

You can also plot a magnetic field vector, by passing in a vector you want into the `mag_vector` argument. All this does is plot a vector at the top of the grid, which can help you to see what direction the magnetic field points in.

Finally, the colours of the spin vectors can be changed. If you look in the source code for plotting, you will see the class `SpinColors`, listed here:

```
class SpinColors(Enum):
    'Enum class for the different color schemes for the spins'
    plain = 1
    alternating = 2
    z_gradient = 3
    xy_gradient = 4
```

By default, plots are set to `plain`, which plots all of the spin vectors in blue. To change this, you need to use the key-word argument `spin_colors`. So if you wanted `plain`, you would write:

```
my_model.plot(spin_colors=plotting.SpinColors.plain)
```

`alternating` means each lattice layer is plotted in an alternate colour.
`z_gradient` means that there is a gradient of colours the spins are plotted as depending on their z-component, so is useful for seeing which direction they are pointing in. This is the recommended colour scheme when plotting the skyrmions.
`xy_gradient` is similar to `z_gradient`, except the colours are now related to the rotation in the x-y plane rather than the z-direction. So this is useful to see which direction the spins are pointing in the plane. This is the recommended colour scheme when plotting $FeBO_3$.

3.5 Saving

You may wish the save the results of a model so you can plot the same model multiple times, without having to solve it again. This can be done using the `model.save_model_results` function. This function takes three arguments. First, is a string containing the parent directory you want the files to be saved in, second is a string for the name of the directory where the results are stored, and lastly you need to give the results of the model, by calling the `.get_model_results()` method on your model. So for example, you can call:

```
model.save_model_results(parent_dir, dir_name, my_model.get_model_results())
```

Then, if you want to plot this result, you can call `plotting.plot_saved_model` function. The first arguments is the directory where the results are saved in, and the rest of the arguments are the same as the `.plot` method on your model. So continuing from the above example, you could write:

```
plotting.plot_saved_model(parent_dir + '/' + dir_name)
```

Saving and then plotting is especially useful looking at smaller parts of a large model.

4 Module explanation

This section gives a description about what each module does, and how the other modules use it. If you want to use the code to create your own models, I advise reading through this and the code to get an understanding on each module, as they are all used when defining a model for yourself.

4.1 `grid`

A key distinction to make in this code is between “grid” space and “real” space. The actual computation is done using a grid, equivalent to an array. The spins are stored in a three-dimensional grid, so this module defines how those grids are indexed.

The class `GridCoord` is used to represent what coordinate on the grid a spin exists. The grid is indexed by coordinates i, j and k, so each point on the grid has a specific grid-coordinate. These coordinates are represented by `GridCoord`, and this class is initialised by specifying the i, j and k variables. So for example:

```
my_coord = GridCoord(1, 2, 3)
```

The methods in the `GridCoord` class are mainly overriding Python operations so they make sense in the context. For example, the equals operator is overridden, so when equating two `GridCoord`’s, returns `True` if their i, j and k components are the same.

`GridCoord.to_tuple()` returns the instance of `GridCoord` expressed as a tuple, where the first, second and third elements of the tuple are the i, j and k values, respectively.

`GridCoord.adjacent_nbrs()` returns a list of `GridCoords` closest neighbouring coordinates in grid space.

`CoordList` is a class for lists of `GridCoords`. This only really exists to override some methods in the standard Python lists, so that functions such as `in` work when applied to `GridCoords`. Most likely, you won’t need to use this class. But if you have a list of `GridCoords`, and you want to do operations like `in` and `append`, then use a `CoordList` to be safe.

4.2 `vectors`

`vectors` is the module which deals in "real" space. The key class in this module is `PyomoVector`. The reason why it has the word "Pyomo" in it, is due to the fact that Pyomo does not deal exactly with floats, rather it deals with "Pyomo objects". Although these objects are initialised using standard Python floats, when the solver is running they are not. Hence, functions like *cos* and *sin* need to be the Pyomo version of *cos* and *sin*, rather than math or numpy. Hence, the class ensures the calculations conform to Pyomo.

As described in section **3.2**, `PyomoVector` is initialised either by specifying the Cartesian components or the spherical polar components. If you want to initialise with Cartesian, you could write:

```
my_vector = PyomoVector(cart_comp=(1, 2, 3))
```

where the elements of `cart_comp` are the x, y and z components. If you want to initialise with spherical polar components, you could write:

```
my_vector = PyomoVector(sph_comp=(1, np.pi, 0.5 * np.pi)
```

where the elements of `sph_comp` are the length, azimuth angle and polar angle (given in radians).

The components that weren't specified are calculated from the ones that are. Meaning if you initialised using Cartesian, then the spherical ones are calculated and can still be accessed, and vice versa.

An important thing about these vectors is you should not change the component by direct access to their class variables. If you want to set any of the components, use the `set_` functions. These will ensure the components are self consistent. For example, if you wanted to change the z component of a vector to 4, you could write:

```
my_vector.set_z(4)
```

Many of the other methods of the class are overriding Python operations on the class. This means addition, subtraction and multiplication by scalar quantities can all be done with vectors.

```
vector_1 = PyomoVector(cart_comp=(1, 2, 3))
vector_2 = PyomoVector(cart_comp=(4, 5, 6))

# All operations methods are valid
vector_1 + vector_2
vector_1 - vector_2
4.5 * vector_1
vector_1 * 3.6
-vector_1
+vector_1
```

The equals comparison between two `PyomoVector`s works as expected; if their components are equal. The `abs` operator returns the length of the vector.

The methods `scalar_product` and `vector_product` are self explanatory, they return the scalar and vector product of two `PyomoVector`s.

`normalise` returns a `PyomoVector` parallel to the current `PyomoVector`, with its `length` specified by the argument `new_length`.

`cart_tuple` and `sph_tuple` return the Cartesian and spherical components of the vector as tuples.

The class `Spin` inherits all of the methods from `PyomoVector`. Basically, it is a unit length `PyomoVector`. Subsequently, it is initialised only by the azimuth and polar angles. It is this `Spin` class which represents the variables in the model.

At the bottom of the script, you will find a list of constants. These are commonly used vectors which are defined there so that they can be imported into other modules to save you writing out the same vectors multiple times.

4.3 `shell`

This module only contains the class `Shell`. The module imports from `grid` and `vectors`, so understanding of those modules is important before using this module.

A `Shell` object is used to describe all of the interactions that a spin has. The first argument when initialising a `Shell` is `centre_coord`, which is the `GridCoord` of the spin in question. The next argument is `initial_spin`, which is where a `Spin` object is given to indicate what the initial direction of the spin in question will be when passed into the solver. For example, you could put `initial_spin=Spin(np.pi, 0.5 * np.pi)`

The next argument is `mag_vector`, which is a `PyomoVector` giving the external magnetic field vector that spin experiences. Finally, the argument `aniso_term` gives the term for the single-ion-anisotropy at that spin. This is given in a tuple, the first argument being the strength (a float) and the second argument the direction, given as a `PyomoVector`. If the strength is positive, the spin will want to be parallel or anti-parallel to the direction vector. If the strength is negative, the spin will want to be in the plane perpendicular to the direction vector.

Just having the `Shell` initialised means the external forces on the spin are there, and the spin is initialised, but at the moment the spin does not experience any interactions with other spins. These are then manually added. To do this, you need to call the `add_coupling` method. The first argument of this method is the `GridCoord` of the spin you want to couple with. The term `spin_coup_term` gives the spin coupling term (a float), i.e. positive indicates the spins want to be parallel, negative indicates the spins want to be anti-parallel. The term `dm_vector` gives the Dzyaloshinskii-Moriya vector for the interaction between the two spins. Both of these are zero by default, so if they aren't specified, then there will not be an interaction of that kind.

You can repeatedly use the `add_coupling` method until all of the spins you want to couple with have been listed. Note the method `adjacent_nbrs` may come in useful for speeding this process up, but remember this gives a list of adjacent neighbours in grid space, they may not be the closest neighbours in real space (as in $FeBO_3$).

The other two methods are used to retrieve the terms for spin-coupling and Dzyaloshinskii-Moriya vectors, these exist due to the classes not working nicely with Python lists. It is unlikely you will need to use these methods, they are mainly there as they are used in `model.SpinModel`.

4.4 `model`

This module is where the magic happens. The class `SpinModel` is where you define your model of interacting spins. You don't need to understand how most of the code in this module works in order to use it, but I'll give an overview.

You start off by initialising your `SpinModel`. There are three arguments when initialising a `SpinModel`:

`param_array` - This is a three-dimensional numpy array of `Shell` objects. Each `Shell` is centred around a spin, so this array gives the description of all of the interactions you want involved in your model, and how many spins you want. This can seem quite a difficult and complicated thing to create, and it is worth looking at the models in `examples` to see how it is done there. The point I want to make is, `for` loops are your friend. Most of the time, you will have some interaction that every spin will experience, so many `Shell`s will be similar. To help grasp the concept, I'll explain how you'll construct a `param_array` for a $5 \times 5 \times 5$ simple cubic ferromagnet, where each spin only interacts with its closest neighbours. An example piece of code is here:

```python
param_array_list = []
for i in range(5):
    i_row = []
    for j in range(5):
        j_row = []
        for k in range(5):
            centre_coord = GridCoord(i, j, k)
            shell = Shell(centre_coord, initial_spin=(random(), random())
            for coord in centre_coord.adjacent_nbrs():
                shell.add_coupling(coord, spin_coup_term=1.0)
            j_row.append(shell)
        i_row.append(j_row)
    param_array_list.append(i_row)
param_array = np.array(param_array_list)
```

The first line defines the empty list to which the `Shells` are stored in. Then the three `for` loops are so we are looping over a three-dimensional grid, and with each indentation, have a new "row" to add to the list. `centre_coord` is the coordinate the `Shell` is centred on, which is at position (i, j, k) in grid space. The `Shell` is then initialised to be centered on this coordinate, with a random initial spin (initial spin choice is down to you). Then, the adjacent neighbours are looped over to add the spin coupling too. Note this could be done outside of a loop, any way you want. Then, the terms are appended to the list. By the end, `param_array_list` is a three-dimensional list of `Shells`. The last line is important as it converts this list into a numpy array, which is essential to the code working.

Defining the `param_array` is by far the most complicated part of defining a model. Be sure to look at the examples to get a better idea of how it works.

`boundary` - This indicates the boundary condition on the grid. What this boils down to is: Are the spins you have defined in your model on their own in space? That is, when you define your three-dimensional grid of spins, are they the only thing near them? If they are, then the spins on the outer edge of the grid do not couple to anything when they "look" outside of the grid, and then you define `boundary='zero'`. On the other hand, you may want the spins you have defined to be part of a large, periodic lattice. In which case, when a spin "looks" outside the grid, it actually looks back into itself as you have a periodicity. In this case, you define `boundary='periodic'`. The default is `'zero'`, due to the fact this does indeed reflect what happens physically when you have nano-scale materials. The other case, although it does produce good results, isn't actually how physical systems work. But the results are interesting, so use it as you please.

`basis_vectors` - This is a tuple of three `PyomoVectors`, which define the basis vectors of the lattice. This defines your model as a Bravais-lattice, and by default is a cubic lattice. This does not really come into play until you start plotting your model, but can be important before then. For example, with the skyrmion model, the direction of the Dzyaloshinskii-Moriya relies on the real-space vector positions of the spins, which depends on the basis vectors.

The initialise method then sets up what is called a `ConcreteModel`, which is a Pyomo object for an optimisation model. This involves defining the variables and objective function of the model. First, the variables are the spins, which can be expressed in terms of their azimuth angle and polar angle. They are indexed by the i, j and k coordinates. So a set of points, called a `RangeSet`, are defined for each of these coordinates. Then the azimuth and polar variables are defined, indexed by these points. Then, the Hamiltonian is defined to be the objective function which needs to be minimised. The definition of the Hamiltonian is listed in the code. The code is well commented, so a complete explanation to how it is defined is not given here (partly because I don't have time).

Once the initialisation method is finished, next the model needs to be solved. This is done by calling the `.solve()` method on the model. This simply calls the Ipopt solver, which will minimise the Hamiltonian, and return the result. These results are stored within the SpinModel object.

The method get_model_results gets the results of the solver. It returns two tuples: The first tuple contains three arrays: the Cartesian components of the positions of the spins (determined by the basis vectors), where the second tuple contains three arrays: the Cartesian components of the vectors of the spins.

To plot this specific SpinModel, the .plot() method needs to be called on the SpinModel. A description on how plotting works was given in section **3.4**, so it will not be repeated here. Also, all the .plot() function does is call a function from the plotting module, so a description on how that works should be saved until then.

There are two other non-private functions within this module. The first is coord_to_vector. This converts a GridCoord into a PyomoVector, depending on the basis vectors of the lattice. The first argument is basis_vectors, a tuple containing the three basis vectors. The second argument is coord, the GridCoord that wants to be converted. The function returns the PyomoVector equivalent of the GridCoord.

The other non-private function is save_model_results. This is used for saving the results of a model, and has already been described in section **3.4**.

4.5 plotting

Finally, the last module is plotting. This module contains the function needed to plot your model, which is plot_spin_model, although you will usually not need to call this function directly. Instead, it is either called on the .plot method of SpinModel, or you call plot_saved_model. Either way, these have been explained in sections **3.4** and **3.5**.

The function plot_spin_model works by taking in two tuples: One contains the Cartesian coordinates of the positions of the vectors, namely input_pos_arrays. The other contains the Cartesian components of the vectors of the spins, namely input_comps_arrays. The formatting of these lists follows the output of the method on SpinModel, get_model_results, so if you want to call this function directly, the easiest way is using that method.

The rest of the arguments for this function give the options for how the grid should be plotted, which was described in section **3.4**. This function does a lot of boring stuff with matplotlib to make the 3D look somewhat nice, which is easier said than done. It's worth noting the 3D plotting won't show if called from command line, you need to use something like DAWN or Spyder in order to get a nice plot for it.

5 Creating your own models

If you've understood everything in sections **3** and **4**, then you should in theory be able to create your own models! The biggest hurdle is understanding what the separate classes do, and then defining the param_array how you want it. Start off with some simple models and then work your way up. Refer back to the examples if you are stuck, and if you can't figure something out, don't be afraid to email me.

A Two-dimensional models

This code was written before the three-dimensional code, almost like a warm up. If you're familiar with the three-dimensional code, you'll see a lot of similarities with it. It's a lot more rough around the edges.

The main difference is this code was written to be compatible with pyOpt and SciPy, as well as Pyomo. As a result, there's a lot of different syntax. A quick overview of the modules will be given here. The code is heavily commented, so there won't be much line-by-line detail.

A.1 spin_solve_2D

The class SpinProblem was used to define a model. It has many initial arguments, but the key one is grid_dimen, which is a tuple giving the dimensions of the two-dimensional grid. boundary gives the boundary condition, as in section **4.4** but in two-dimensions. start_spins is a two-dimensional grid of floats, which represent the starting angle of the spins. The spins are parameterised by a single float value in the range 0 to 2π. The basis vectors are given using the Vector class, defined at the top of the module, which is simply a class that contains the x and y coordinates. The rest of the arguments describe the interactions that take place in the problem.

The interesting point here is you can choose which solver you want to solve the problem. These solvers are available in pyOpt and SciPy already, and if you've already installed Ipopt then you can call that with Pyomo. If you want an idea which solvers are good or bad at solving these problems, see my report.

To plot a solved model, call the .plot() method on your SpinProblem object.

A.2 spin_plot

This is the module in charge of plotting the results of a SpinProblem. As in the three-dimensional case, you probably won't ever call the plot_spins function directly, and instead call it via a SpinProblem object, or from some results you've saved earlier.

There aren't any options as to things like colours or how big the grid you plot is, so it's mainly just a manner of calling a function and being done with it.

The plot_from_file function is for plotting spins saved in a csv file. As far as I can see, I've appeared to have lost the function that actually saved the result of a solver as a csv file, but this should not be too hard as they are just a two-dimensional array of floats. See it as homework for anyone who is keen enough to have actually read this far down the document!

B Three-dimensional code listing

B.1 grid.py

```python
'Contains the class for 3D grid coordinates'


class GridCoord(object):
    'Class for coordinates of the points of the grid where spins are.'
    def __init__(self, i, j, k):
        self.i = i
        self.j = j
        self.k = k

    def l1_distance(self, grid_coord):
        '''Calculates the distance between itself and another grid_coord
        according to the L1 metric'''
        return abs(self.i - grid_coord.i)\
            + abs(self.j - grid_coord.j)\
            + abs(self.k - grid_coord.k)

    def to_tuple(self):
        'Converts to a tuple'
        return (self.i, self.j, self.k)

    def __eq__(self, grid_coord):
        'Overloads the == function to compare coordinates'
        if isinstance(grid_coord, GridCoord):
            return self.is_equal(grid_coord)
        return NotImplemented

    def is_equal(self, grid_coord):
        'Prints True if self is the same coordinate as grid_coord'
        return self.i == grid_coord.i and self.j == grid_coord.j \
            and self.k == grid_coord.k

    def adjacent_nbrs(self):
        'Returns the GridCoords of the adjacent neighbours to grid_coord'
        return (GridCoord(self.i - 1, self.j, self.k),
                GridCoord(self.i + 1, self.j, self.k),
                GridCoord(self.i, self.j - 1, self.k),
                GridCoord(self.i, self.j + 1, self.k),
                GridCoord(self.i, self.j, self.k - 1),
                GridCoord(self.i, self.j, self.k + 1))


class CoordList(list):
    '''Class for a list of grid coordinates, overriding some methods'''
    def in_list(self, coord):
        '''Returns True if the GridCoord object coord is in the list'''
        for item in self:
            if coord == item:
                return True
        return False

    def __contains__(self, item):
        if isinstance(item, GridCoord):
            return self.in_list(item)
        else:
            return list.__contains__(self, item)

    def append(self, item):
        '''Appends an item to a list only if it isn't already in it'''
        if item in self:
            raise Exception("Item already in list")
        else:
            super(CoordList, self).append(item)

    def index(self, item):
        '''Returns the index in the list which the item is stored'''
        for i, element in enumerate(self):
            if element == item:
                return i
        raise ValueError(str(item) + ' is not in list')
```

B.2 vectors.py

```
'Class for a Pyomo vector and a Spin object which is derived from PyomoVector'

from __future__ import division
from pyomo.environ import sqrt, value, cos, sin, acos, atan

import numpy as np


class PyomoVector(object):
    '''Generic vector object which stores both the Cartesian and Spherical
    components of the vector'''
    def __init__(self, cart_comp=None, sph_comp=None):
        '''Initialised the vector either in Cartesian components or Spherical
        components. If both are given, only looks at cart_comp'''
        if cart_comp is not None:
            self.x = cart_comp[0]
            self.y = cart_comp[1]
            self.z = cart_comp[2]

            self.length = sqrt(self.x ** 2 + self.y ** 2 + self.z ** 2)
            if value(self.length) == 0:
                self.azi = 0.0
                self.pol = 0.0
            else:
                self.pol = acos(self.z / self.length)
                if value(self.x) != 0:
                    self.azi = atan(self.y / self.x)
                else:
                    self.azi = 0.0
        elif sph_comp is not None:
            self.length = sph_comp[0]
            self.azi = sph_comp[1]
            self.pol = sph_comp[2]

            self.x = self.length * cos(self.azi) * sin(self.pol)
            self.y = self.length * sin(self.azi) * sin(self.pol)
            self.z = self.length * cos(self.pol)
        else:
            raise Exception('''Must give either Cartesian or Spherical
                            components''')

    def _config_sph(self):
        'Configures the spherical components from the Cartesian'
        self.length = sqrt(self.x ** 2 + self.y ** 2 + self.z ** 2)
        if value(self.length) == 0:
            self.azi = 0.0
            self.pol = 0.0
        else:
            self.pol = acos(self.z / self.length)
            if value(self.x) != 0:
                self.azi = atan(self.y / self.x)
            else:
                self.azi = 0.0

    def _config_cart(self):
        'Configures the Cartesian components from the spherical'
        self.x = self.length * cos(self.azi) * sin(self.pol)
        self.y = self.length * sin(self.azi) * sin(self.pol)
        self.z = self.length * cos(self.pol)

    def set_x(self, comp_value):
        'Sets the x-component to the specified value'
        self.x = comp_value
        self._config_sph()

    def set_y(self, comp_value):
        'Sets the y-component to the specified value'
        self.y = comp_value
        self._config_sph()

    def set_z(self, comp_value):
        'Sets the z-component to the specified value'
        self.z = comp_value
        self._config_sph()

    def set_length(self, comp_value):
        'Sets the length component to the specified value'
        self.length = comp_value
        self._config_cart()

    def set_azi(self, comp_value):
        'Sets the azi component to the specified value'
        self.azi = comp_value
        self._config_cart()

    def set_pol(self, comp_value):
        'Sets the pol component to the specified value'
        self.pol = comp_value
        self._config_cart()

    def add(self, vector):
        'Returns the addition of itself with another PyomoVector'
        return PyomoVector(cart_comp=(self.x + vector.x,
                                      self.y + vector.y,
                                      self.z + vector.z))

    def __add__(self, vector):
        'Overrides the addition operator'
        if isinstance(vector, PyomoVector):
            return self.add(vector)
        return NotImplemented

    def multiply(self, scalar):
        'Returns the result of the vector being multiplied by a scalar'
        return PyomoVector(cart_comp=(scalar * self.x,
                                      scalar * self.y,
                                      scalar * self.z))

    def __mul__(self, scalar):
        'Overrides the multiplication operator for scalars'
        return self.multiply(scalar)
```

```
def __rmul__(self, scalar):
    'Overrides the reverse multiplication for scalars'
    return self * scalar

def __sub__(self, vector):
    'Overrides the subtraction of vectors'
    if isinstance(vector, PyomoVector):
        neg_vector = -1 * vector
        return self + neg_vector
    return NotImplemented

def __pos_(self):
    'Overrids the "+" operand in front of a vector'
    return self

def __neg__(self):
    'Overrides the "-" operand in front of a vector'
    return -1 * self

def __abs__(self):
    'Overrides the abs() function'
    return self.length

def scalar_product(self, vector):
    'Returns the scalar product of itself with another PyomoVector'
    return self.x * vector.x + self.y * vector.y + self.z * vector.z

def vector_product(self, vector):
    'Returns the vector product of itself with another PyomoVector'
    x_comp = self.y * vector.z - self.z * vector.y
    y_comp = self.z * vector.x - self.x * vector.z
    z_comp = self.x * vector.y - self.y * vector.x
    return PyomoVector(cart_comp=(x_comp, y_comp, z_comp))

def normalise(self, new_length):
    'Returns a vector parallel to itself with given length'
    if value(abs(self)) == 0:
        return self
    else:
        factor = new_length / abs(self)
        return self * factor

def cart_tuple(self):
    'Returns the Cartesian components in a tuple'
    return (value(self.x), value(self.y), value(self.z))

def sph_tuple(self):
    'Returns the Spherical components in a tuple'
    return (value(self.length), value(self.azi), value(self.pol))

def __eq__(self, vector):
    'Overides the equals method'
    if isinstance(vector, PyomoVector):
            return self.x == vector.x and\
                self.y == vector.y and\
                self.z == vector.z
        return NotImplemented


class Spin(PyomoVector):
    '''Object for a spin, which is a unit vector and defined by an azimuth and
    polar angle.'''
    def __init__(self, azi, pol):
        'Azimuth and polar angles passed in to define the spin.'
        PyomoVector.__init__(self, sph_comp=(1.0, azi, pol))

# Unit vectors in x, y, z directions
X_UNIT = PyomoVector(cart_comp=(1.0, 0.0, 0.0))
Y_UNIT = PyomoVector(cart_comp=(0.0, 1.0, 0.0))
Z_UNIT = PyomoVector(cart_comp=(0.0, 0.0, 1.0))

# Zero vector
ZERO_VECTOR = PyomoVector(cart_comp=(0.0, 0.0, 0.0))

# Standard cubic lattice basis vectors
CUBIC_BASIS_VECTORS = (X_UNIT, Y_UNIT, Z_UNIT)

# Hexagonal lattice basis vectors
HEX_BASIS_1 = X_UNIT
HEX_BASIS_2 = PyomoVector(cart_comp=(np.cos(np.radians(120)),
                                     np.sin(np.radians(120)),
                                     0.0))
HEX_BASIS_3 = Z_UNIT
HEX_BASIS_VECTORS = (HEX_BASIS_1, HEX_BASIS_2, HEX_BASIS_3)
```

B.3 shell.py

```
'''Class for storing the coupling parameters of each spin with its
neighbours, stored in shells'''

from model3D.grid import CoordList
from model3D.vectors import PyomoVector, Spin, ZERO_VECTOR


class Shell(object):
    '''Class for a shell around a grid point storing the coupling parameters of
    the spin at that point with its neighbouring spins'''
    def __init__(self, centre_coord, initial_spin=Spin(0.0, 0.0),
                 mag_vector=ZERO_VECTOR, aniso_term=(0.0, ZERO_VECTOR)):
        '''The coordinate of the centre of the shell is given'''
        self.centre = centre_coord  # Grid coord of the centre of the shell
        self.initial_spin = initial_spin
        self.mag_vector = mag_vector  # Magnetic field vector at the centre

        self.aniso_strength = aniso_term[0]
        self.aniso_direction = aniso_term[1].normalise(1.0)
        self.coord_list = CoordList([])  # List of the coordinates for coupling
        self.spin_coup_dict = {}  # Dict for values of spin coupling params
        self.dm_vector_dict = {}  # Dict for the DM-vector when coupling with
                                  # each coord

    def add_coupling(self, coord, spin_coup_term=0.0, dm_vector=ZERO_VECTOR):
        '''Adds a coordinate of a spin and coupling terms for the centre spin
        to interact with'''
        if coord in self.coord_list:
            for key_coord in self.spin_coup_dict.keys():
                if key_coord == coord:
                    self.spin_coup_dict[key_coord] = spin_coup_term

            for key_coord in self.dm_vector_dict.keys():
                if key_coord == coord:
                    self.dm_vector_dict[key_coord] = dm_vector
        else:
            self.coord_list.append(coord)
            self.spin_coup_dict[coord] = spin_coup_term
            self.dm_vector_dict[coord] = dm_vector

    def get_spin_coup(self, coord):
        '''Returns the spin coupling term at coord'''
        coord_index = self.coord_list.index(coord)
        return self.spin_coup_dict[self.coord_list[coord_index]]

    def get_dm_vector(self, coord):
        '''Returns the Dzyaloshinskii-Moriya interaction vector at coord'''
        coord_index = self.coord_list.index(coord)
        return self.dm_vector_dict[self.coord_list[coord_index]]
```

B.4 model.py

```
'Model for three dimensional spins on a three dimensional lattice'

from __future__ import division

import os
from pyomo.environ import Var, Objective, value, ConcreteModel, RangeSet, \
                          Reals, minimize
from pyomo.opt import SolverFactory
import numpy as np

from model3D.vectors import PyomoVector, Spin, CUBIC_BASIS_VECTORS, ZERO_VECTOR
from model3D.grid import GridCoord
from model3D.plotting import plot_spin_model, SpinColors


def _init_rule_azi(pyomo_model, i, j, k):
    'Initialisation rule for the azimuth angles'
    spin = pyomo_model.param_array[value(i), value(j), value(k)].initial_spin
    return value(spin.azi)
```

```python
def _init_rule_pol(pyomo_model, i, j, k):
    'Initialisation rule for the polar angles'
    spin = pyomo_model.param_array[value(i), value(j), value(k)].initial_spin
    return value(spin.pol)


def _get_bound_coord(pyomo_model, input_coord):
    '''Returns the coordinate corresponding to the boundary condition for
     the input coordinate'''
    if input_coord in pyomo_model.coord_array:
        # If the coordinate is in the grid, return the coordinate
        return input_coord
    elif pyomo_model.boundary == 'zero':
        return 'zero'  # Returning 'zero' indicates there is "no spin" here
    elif pyomo_model.boundary == 'periodic':
        # Puts required data for coordinate, points and length of the grid in
        # lists in order to loop over them
        component_list = [input_coord.i, input_coord.j, input_coord.k]
        grid_points_list = [pyomo_model.grid_i_points,
                            pyomo_model.grid_j_points,
                            pyomo_model.grid_k_points]
        grid_no_list = [pyomo_model.grid_i_no,
                        pyomo_model.grid_j_no,
                        pyomo_model.grid_k_no]

        # Finds the corresponding grid point for all three coordinates
        for p in range(3):
            if value(component_list[p]) < 0:
                while not component_list[p] in grid_points_list[p]:
                    component_list[p] += grid_no_list[p]
            elif value(component_list[p]) >= grid_no_list[p]:
                while not component_list[p] in grid_points_list[p]:
                    component_list[p] -= grid_no_list[p]

        return GridCoord(component_list[0],
                         component_list[1],
                         component_list[2])


def _get_spin(pyomo_model, input_coord):
    'Returns the spin corresponding to the position given by input_coord'
    bound_coord = _get_bound_coord(pyomo_model, input_coord)
    if bound_coord == 'zero':
        return PyomoVector(cart_comp=(0.0, 0.0, 0.0))
    else:
        return Spin(pyomo_model.azi[bound_coord.i,
                                    bound_coord.j,
                                    bound_coord.k],
                    pyomo_model.pol[bound_coord.i,
                                    bound_coord.j,
                                    bound_coord.k])


def _nbr_coup_sum(pyomo_model, input_coord):
    '''Returns the sum of the interactions the spin at input_coord has with its
    neighbours that are listed in its shell, including the spin coupling and
    Dzyaloshinskii-Moriya exchange'''
    input_spin = _get_spin(pyomo_model, input_coord)
    param_shell = pyomo_model.param_array[value(input_coord.i),
                                          value(input_coord.j),
                                          value(input_coord.k)]

    nbr_spin_list = []  # List to store the neighbouring spins
    coup_const_list = []  # List to store the spin coupling constants
    dm_vector_list = []  # List to store the DM vectors for each of the spins

    for nbr_coord in param_shell.coord_list:
        nbr_spin_list.append(_get_spin(pyomo_model, nbr_coord))
        coup_const_list.append(param_shell.get_spin_coup(nbr_coord))
        dm_vector_list.append(param_shell.get_dm_vector(nbr_coord))

    return sum(coup_const_list[n] *
               input_spin.scalar_product(nbr_spin_list[n]) +
               dm_vector_list[n].scalar_product(
                   input_spin.vector_product(nbr_spin_list[n]))
               for n in range(len(nbr_spin_list)))


def _hamiltonian(pyomo_model):
    'The Hamiltonian function for the given model to be minimized by Pyomo'
    total_nbr_coup = 0.0  # Contribution from spin coupling with neighbours
    total_mag = 0.0  # Contribution from coupling of magnetic field and spins
    total_aniso = 0.0  # Contribution from coupling with the anisotropy
    for coord in pyomo_model.coord_array:
        # Spin at the input coordinate location
        input_spin = _get_spin(pyomo_model, coord)

        # Shell for coupling parameters
        param_shell = pyomo_model.param_array[value(coord.i),
                                              value(coord.j),
                                              value(coord.k)]

        # Adds sum of scalar products of nearest neighbours
        total_nbr_coup += _nbr_coup_sum(pyomo_model, coord)

        # Adds scalar product of the spin with the magnetic field
        mag_vector = param_shell.mag_vector
        total_mag += input_spin.scalar_product(mag_vector)

        # Adds the coupling with the anisotropy
        total_aniso += param_shell.aniso_strength \
            * (input_spin.scalar_product(param_shell.aniso_direction)) ** 2

    return -total_nbr_coup - total_mag - total_aniso


def coord_to_vector(basis_vectors, coord):
    'Returns the vector that the GridCoord coord represents in real space'
    return coord.i * basis_vectors[0]\
        + coord.j * basis_vectors[1]\
        + coord.k * basis_vectors[2]


def save_model_results(parent_dir, save_dir_name, spins_list):
    '''Saves the model in the parent directory, parent_dir. Results list is a
    list in the format of the output of SpinModel.get_model_results'''
    # Creates a directory to save the results in
    save_dir = parent_dir + '/' + save_dir_name
    if not os.path.exists(parent_dir + '/' + save_dir_name):
        os.mkdir(parent_dir + '/' + save_dir_name)

    # Gets the results to save into the directory
    xyz_list = ['x', 'y', 'z']  # Characters used for saving file names
    for m in range(3):
        np.save(save_dir + '/' + xyz_list[m] + '_pos',
                spins_list[0][m])
        np.save(save_dir + '/' + xyz_list[m] + '_spin_comps',
                spins_list[1][m])


class SpinModel(object):
    'Class for creating the model of a 3D lattice of 3D spins'
    def __init__(self,  param_array, boundary='zero',
                 basis_vectors=CUBIC_BASIS_VECTORS):
        '''
        Input parameters required to create the model:
        * boundary - String for the boundary condition, either "periodic" or
          "zero"
        * param_array - A numpy array of shells, which will have one shell per
          spin on the grid, and the shell is a Shell object which stores the
          parameters needed for coupling that particular spin
        * basis_vectors - The basis vectors of the lattice
        '''
        self.basis_vectors = basis_vectors

        # Defining the Pyomo model
        self.pyomo_model = ConcreteModel()

        # ======================================================================
        # Storing parameters as model objects
        # ======================================================================
        self.pyomo_model.boundary = boundary

        self.pyomo_model.param_array = param_array

        # ======================================================================
        # Derived objects from the parameters
        # ======================================================================
        # The number of points in the i, j and k directions in grid space
        self.pyomo_model.grid_i_no = param_array.shape[0]
        self.pyomo_model.grid_j_no = param_array.shape[1]
        self.pyomo_model.grid_k_no = param_array.shape[2]
        # Creates an array of all the coordinates in the grid
        self.pyomo_model.coord_array = np.array([param_shell.centre
                                                 for param_shell
                                                 in param_array.flatten()])

        # Ranges for the grid points
        self.pyomo_model.grid_i_points =\
            RangeSet(0, self.pyomo_model.grid_i_no - 1)
        self.pyomo_model.grid_j_points =\
            RangeSet(0, self.pyomo_model.grid_j_no - 1)
        self.pyomo_model.grid_k_points =\
            RangeSet(0, self.pyomo_model.grid_k_no - 1)

        # Variables for the azimuth and polar angles of the spins
        self.pyomo_model.azi = Var(self.pyomo_model.grid_i_points,
                                   self.pyomo_model.grid_j_points,
                                   self.pyomo_model.grid_k_points,
                                   domain=Reals,
                                   bounds=(0.0, 2 * np.pi),
                                   initialize=_init_rule_azi)

        self.pyomo_model.pol = Var(self.pyomo_model.grid_i_points,
                                   self.pyomo_model.grid_j_points,
                                   self.pyomo_model.grid_k_points,
                                   domain=Reals,
                                   bounds=(0.0, np.pi),
                                   initialize=_init_rule_pol)

        # Objective function to minimize
        self.pyomo_model.OBJ = Objective(rule=_hamiltonian, sense=minimize)

        # Creates the results variables
        self.results_array_azi = np.zeros((value(self.pyomo_model.grid_i_no),
                                           value(self.pyomo_model.grid_j_no),
                                           value(self.pyomo_model.grid_k_no)))
        self.results_array_pol = np.copy(self.results_array_azi)
        self.results_obj = 0.0

    def solve(self):
        'Solves the given problem using the ipopt solver'
        # Solves the model
        opt = SolverFactory("ipopt")
        opt.solve(self.pyomo_model)

        for input_coord in self.pyomo_model.coord_array:
            i, j, k = input_coord.i, input_coord.j, input_coord.k
            self.results_array_azi[i, j, k] =\
                value(self.pyomo_model.azi[i, j, k])
            self.results_array_pol[i, j, k] =\
                value(self.pyomo_model.pol[i, j, k])

        self.results_obj = value(self.pyomo_model.OBJ)
    def get_model_results(self):
        'Returns arrays containing the positions and components of the spins'
        coord_array = self.pyomo_model.coord_array  # To reduce length of lines

        # Arrays to store x, y and z positions of the spins
        x_pos, y_pos, z_pos = [], [], []
        for coord in coord_array:
            # Converts coord into vector in real space
            pos_vector = coord_to_vector(self.basis_vectors, coord)
            x_pos.append(pos_vector.x)
            y_pos.append(pos_vector.y)
            z_pos.append(pos_vector.z)

        # Converts the lists into numpy arrays
        x_pos = np.array(x_pos)
        y_pos = np.array(y_pos)
        z_pos = np.array(z_pos)

        # Components of the vectors of the spins
        x_spin_comps = np.zeros(x_pos.size)
        y_spin_comps = np.copy(x_spin_comps)
        z_spin_comps = np.copy(x_spin_comps)

        # Fills the spin component arrays
        for index, coord in enumerate(coord_array):
            spin = _get_spin(self.pyomo_model, coord)
            x_spin_comps[index] = value(spin.x)
            y_spin_comps[index] = value(spin.y)
            z_spin_comps[index] = value(spin.z)

        return ((x_pos, y_pos, z_pos),
                (x_spin_comps, y_spin_comps, z_spin_comps))

    def plot(self, spin_colors=SpinColors.plain, plot_points=True,
             x_limits=(-np.inf, np.inf), y_limits=(-np.inf, np.inf),
             z_limits=(-np.inf, np.inf), mag_vector=ZERO_VECTOR):
        'Passes the required data to lattice_3D_plot for plotting'
        spin_results = self.get_model_results()
        plot_spin_model(spin_results[0], spin_results[1],
                        spin_colors=spin_colors, plot_points=plot_points,
                        x_limits=x_limits, y_limits=y_limits,
                        z_limits=z_limits, mag_vector=mag_vector)
```

B.5 plotting.py

```
'Function for plotting a 3D lattice'

from enum import Enum
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import numpy as np

from model3D.vectors import ZERO_VECTOR, value
```

```
class SpinColors(Enum):
    'Enum class for the different color schemes for the spins'
    plain = 1
    alternating = 2
    z_gradient = 3
    xy_gradient = 4


def plot_spin_model(input_pos_arrays, input_comps_arrays,
                    spin_colors=SpinColors.plain, plot_points=True,
                    x_limits=(-np.inf, np.inf), y_limits=(-np.inf, np.inf),
                    z_limits=(-np.inf, np.inf), mag_vector=ZERO_VECTOR):
    'Plots the spins of a given SpinModel object'
    # Creates figure and axes
    fig = plt.figure()
    axes = fig.add_subplot(111, projection='3d')

    # Removing the points that are not within the x, y, z limits
    no_points = len(input_pos_arrays[0])

    x_pos, y_pos, z_pos, = [], [], []
    x_spin_comps, y_spin_comps, z_spin_comps = [], [], []

    for n in range(no_points):
        if x_limits[0] <= input_pos_arrays[0][n] <= x_limits[1] and\
                y_limits[0] <= input_pos_arrays[1][n] <= y_limits[1] and\
                z_limits[0] <= input_pos_arrays[2][n] <= z_limits[1]:
            x_pos.append(input_pos_arrays[0][n])
            y_pos.append(input_pos_arrays[1][n])
            z_pos.append(input_pos_arrays[2][n])

            x_spin_comps.append(input_comps_arrays[0][n])
            y_spin_comps.append(input_comps_arrays[1][n])
            z_spin_comps.append(input_comps_arrays[2][n])

    # Converting lists into numpy arrays
    x_pos, y_pos, z_pos = np.array(x_pos), np.array(y_pos), np.array(z_pos)
    x_spin_comps, y_spin_comps, z_spin_comps = np.array(x_spin_comps),\
        np.array(y_spin_comps),\
        np.array(z_spin_comps)

    # Scatter plot for the grid points where the spins are
    if plot_points:
        axes.scatter(x_pos, y_pos, z_pos, color='blue')
        # axes.scatter(x_pos[::2], y_pos[::2], z_pos[::2], color='red')
        # axes.scatter(x_pos[1::2], y_pos[1::2], z_pos[1::2], color='green')

    # Plots the spin vectors
    if spin_colors == SpinColors.plain:
        # All the spins are plotted with the same colour'
        axes.quiver(x_pos, y_pos, z_pos,
                    x_spin_comps, y_spin_comps, z_spin_comps,
                    length=0.4, arrow_length_ratio=0.5, pivot='tail',
```

```python
                   linewidth=3.0)
    elif spin_colors == SpinColors.alternating:
        # The spins have alternating colours for each lattice layer
        axes.quiver(x_pos[::2], y_pos[::2], z_pos[::2],
                    x_spin_comps[::2], y_spin_comps[::2], z_spin_comps[::2],
                    length=0.4, arrow_length_ratio=0.5, pivot='tail',
                    linewidth=3.0, color='green')
        axes.quiver(x_pos[1::2], y_pos[1::2], z_pos[1::2],
                    x_spin_comps[1::2], y_spin_comps[1::2], z_spin_comps[1::2],
                    length=0.4, arrow_length_ratio=0.5, pivot='tail',
                    linewidth=3.0)
    elif spin_colors == SpinColors.z_gradient:
        # The spins have a gradient of colours depending on their z-component
        for p, z_comp in enumerate(z_spin_comps):
            color = (0, 0.5 * (1 - z_comp), 0.5 * (1 + z_comp))
            axes.quiver(x_pos[p], y_pos[p], z_pos[p],
                        x_spin_comps[p], y_spin_comps[p], z_spin_comps[p],
                        length=0.4, arrow_length_ratio=0.5, pivot='tail',
                        linewidth=3.0, color=color)

    elif spin_colors == SpinColors.xy_gradient:
        for p in range(len(x_pos)):
            color = (0.5 * (1 - x_spin_comps[p]), 0.5 * (1 + y_spin_comps[p]),
                     0)
            axes.quiver(x_pos[p], y_pos[p], z_pos[p],
                        x_spin_comps[p], y_spin_comps[p], z_spin_comps[p],
                        length=0.4, arrow_length_ratio=0.5, pivot='tail',
                        linewidth=3.0, color=color)

    # Determines how the graph should be scaled
    max_range = np.array([x_pos.max() - x_pos.min(),
                          y_pos.max() - y_pos.min(),
                          z_pos.max() - z_pos.min()]).max() / 2.0
    axes.set_xlim(x_pos.mean() - max_range, x_pos.mean() + max_range)
    axes.set_ylim(y_pos.mean() - max_range, y_pos.mean() + max_range)
    axes.set_zlim(z_pos.mean() - max_range, z_pos.mean() + max_range)

    # Draws the magnetic field vector
    axes.quiver([x_pos.mean()], [y_pos.mean()],
                [z_pos.max() + 0.1 * max_range], [value(mag_vector.x)],
                [value(mag_vector.y)], [value(mag_vector.z)],
                length=0.5 * max_range, arrow_length_ratio=0.3, pivot='tail',
                linewidth=3.0)

    # Setting axis labels
    axes.set_xlabel('x')
    axes.set_ylabel('y')
    axes.set_zlabel('z')

    plt.show()


def plot_saved_model(save_dir, spin_colors=SpinColors.plain, plot_points=True,
                     x_limits=(-np.inf, np.inf), y_limits=(-np.inf, np.inf),
                  z_limits=(-np.inf, np.inf), mag_vector=ZERO_VECTOR):
    'Plots the results that are saved in save_dir'
    load_pos_arrays, load_comps_arrays = [], []
    # Gets the results to save into the directory
    xyz_list = ['x', 'y', 'z']  # Characters used for saving file names
    for m in range(3):
        load_pos_arrays.append(np.load(
            save_dir + '/' + xyz_list[m] + '_pos.npy'))
        load_comps_arrays.append(np.load(
            save_dir + '/' + xyz_list[m] + '_spin_comps.npy'))

    plot_spin_model(load_pos_arrays, load_comps_arrays,
                    spin_colors=spin_colors, plot_points=plot_points,
                    x_limits=x_limits, y_limits=y_limits, z_limits=z_limits,
                    mag_vector=mag_vector)
```

B.6 examples.py

```
'Test lattice models'

from __future__ import division

import numpy as np
from numpy.random import random

from model3D.model import SpinModel, coord_to_vector
from model3D.shell import Shell
from model3D.vectors import PyomoVector, Spin, Z_UNIT, HEX_BASIS_VECTORS,\
            ZERO_VECTOR
from model3D.grid import GridCoord


def skyrmion_model(grid_dimen):
    '''Returns a model for a two-dimensional grid containing Skyrmions, with
    dimensions given by the tuple grid_dimen'''
    param_array_list = []  # List to store shells around each spin

    #  Creates list of shells for parameters
    for i in range(grid_dimen[0]):
        i_row = []
        for j in range(grid_dimen[1]):
            #  Shell for coordinate (i, j, k)
            shell = Shell(GridCoord(i, j, 0),
                          initial_spin=Spin(random() * 2 * np.pi,
                                            random() * np.pi),
                          mag_vector=1.1 * Z_UNIT,
                          aniso_term=(0.5, Z_UNIT))

            #  Vector of the position of the spin
            pos_vector = coord_to_vector(HEX_BASIS_VECTORS, GridCoord(i, j, 0))

            #  List of neighbours in the plane
            nbr_list = [GridCoord(i - 1, j, 0), GridCoord(i + 1, j, 0),
                        GridCoord(i, j - 1, 0), GridCoord(i, j + 1, 0)]
            #  Adds the coupling with the neighbouring spins
            for coord in nbr_list:
                nbr_vector = coord_to_vector(HEX_BASIS_VECTORS, coord)
                unit_path_vector = (nbr_vector - pos_vector).normalise(1)
                dm_vector = Z_UNIT.vector_product(unit_path_vector)
                shell.add_coupling(coord, spin_coup_term=1.0,
                                   dm_vector=dm_vector)

            i_row.append([shell])
        param_array_list.append(i_row)

    #  Converts to a numpy array
    param_array = np.array(param_array_list)

    return SpinModel(param_array, basis_vectors=HEX_BASIS_VECTORS)


def febo3_model(grid_dimen, mag_vector=ZERO_VECTOR):
    '''Returns a model for a three-dimensional lattice of FeBO_3, with
    dimensions given by the tuple grid_dimen, and a magnetic field vector,
    mag_vector'''
    basis_vectors = (HEX_BASIS_VECTORS[0], HEX_BASIS_VECTORS[1],
                     (1 / 3) * HEX_BASIS_VECTORS[0] +
                     (2 / 3) * HEX_BASIS_VECTORS[1] +
                     (1 / 6) * Z_UNIT)
    param_array_list = []  # List to store shells around each spin

    # Initial spin vectors
    init_spin_1 = Spin(random() * 2 * np.pi, random() * np.pi)
    init_spin_2 = Spin(random() * 2 * np.pi, random() * np.pi)

    #  Creates list of shells for parameters
    for i in range(grid_dimen[0]):
        i_row = []
        for j in range(grid_dimen[1]):
            j_row = []
            for k in range(grid_dimen[2]):
                shell = None
                if j < grid_dimen[1] % 2:
                    shell = Shell(GridCoord(i, j, k),
                                  initial_spin=init_spin_1,
                                  aniso_term=(-100, Z_UNIT),
                                  mag_vector=mag_vector)
                else:
                    shell = Shell(GridCoord(i, j, k),
                                  initial_spin=init_spin_2,
                                  aniso_term=(-100, Z_UNIT),
                                  mag_vector=mag_vector)

                #  Coupling for spins below
                shell.add_coupling(GridCoord(i, j, k - 1),
                                   spin_coup_term=-10.3,
                                   dm_vector=PyomoVector(cart_comp=(0, 0,
                                                                      -0.5)))
                shell.add_coupling(GridCoord(i + 1, j, k - 1),
                                   spin_coup_term=-10.3,
                                   dm_vector=PyomoVector(cart_comp=(0, 0,
                                                                    -0.5)))
                shell.add_coupling(GridCoord(i, j + 1, k - 1),
                                   spin_coup_term=-10.3,
                                   dm_vector=PyomoVector(cart_comp=(0, 0,
                                                                    -0.5)))

                #  Coupling for spins above
                shell.add_coupling(GridCoord(i, j, k + 1),
                                   spin_coup_term=-10.3,
                                   dm_vector=-PyomoVector(cart_comp=(0, 0,
                                                                     -0.5)))
                shell.add_coupling(GridCoord(i - 1, j, k + 1),
                                   spin_coup_term=-10.3,
                                   dm_vector=-PyomoVector(cart_comp=(0, 0,
                                                                     -0.5)))
                shell.add_coupling(GridCoord(i, j - 1, k + 1),
                                   spin_coup_term=-10.3,
                                   dm_vector=-PyomoVector(cart_comp=(0, 0,
                                                                     -0.5)))

                j_row.append(shell)
            i_row.append(j_row)
        param_array_list.append(i_row)

    #  Converts to a numpy array
    param_array = np.array(param_array_list)

    return SpinModel(param_array, basis_vectors=basis_vectors)


def simple_cubic_model(grid_dimen, spin_coup_term=1.0, mag_vector=ZERO_VECTOR):
    '''Returns a model for a three-dimensional, cubic lattice, with spins
    interacting with their closest neighbours by the spin coupling term, and an
    applied magnetic field given by mag_vector'''
    param_array_list = []  # List to store shells around each spin

    #  Creates list of shells for parameters
    for i in range(grid_dimen[0]):
        i_row = []
        for j in range(grid_dimen[1]):
            j_row = []
            for k in range(grid_dimen[2]):
                #  Shell for coordinate (i, j, k)
                shell = Shell(GridCoord(i, j, k),
                              initial_spin=Spin(0,
                                                0),
                              mag_vector=mag_vector)

                #  Adds coupling with adjacent neighbours
                for coord in GridCoord(i, j, k).adjacent_nbrs():
                    shell.add_coupling(coord, spin_coup_term=spin_coup_term)
                j_row.append(shell)
            i_row.append(j_row)
        param_array_list.append(i_row)

    # Converts to a numpy array
    param_array = np.array(param_array_list)

    return SpinModel(param_array)
```

C Two-dimensional code listing

C.1 spin_solve_2D.py

```
'Python script for creating a 2D model of interacting spins'


import pyomo.environ
from pyomo.opt import SolverFactory
import time
import scipy.optimize
import numpy as np
import pyOpt
from spin_plot import plot_spins


class Vector(object):
    'Class for a 2D vector in Cartesian space'
    def __init__(self, x, y):
        self.x = x
        self.y = y


def _init_spins_rule(pyomo_model, i, j):
    'Returns the spin (i,j) in the initial configuration'
    return pyomo_model.start_spins[i, j]


def _get_bound_coord(input_coord, problem):
    'Gets the coordinate relating input_coord depending on boundary conditions'
    if input_coord.i in range(problem.row_no) and\
       input_coord.j in range(problem.col_no):
        return input_coord
    elif problem.boundary == 'zero':
        return None  # Indicates there is "no spin" here
    elif problem.boundary == 'periodic':

        # Matches the point back into the defined grid
        if input_coord.i < 0:
            while input_coord.i not in range(problem.row_no):
                input_coord.i += problem.row_no
        elif input_coord.i > problem.row_no - 1:
            while input_coord.i not in range(problem.row_no):
                input_coord.i -= problem.row_no

        if input_coord.j < 0:
            while input_coord.j not in range(problem.col_no):
                input_coord.j += problem.col_no
        elif input_coord.j > problem.col_no - 1:
            while input_coord.j not in range(problem.row_no):
                input_coord.j -= problem.col_no
        return input_coord


def _get_spin(input_coord, problem, is_pyomo, spin_array):
    '''Returns the spin at input_coord.
    If is_pyomo is True, it returns the Pyomo spin variable.
    If is_pyomo is False, then it will get the spin from spin_array which
    is a 2D numpy array'''
    bound_coord = _get_bound_coord(input_coord, problem)
    if is_pyomo:
        return problem.model.s[bound_coord.i, bound_coord.j]
    else:
        return spin_array[bound_coord.i, bound_coord.j]


def _adj_sum(input_coord, problem, is_pyomo, spin_array):
    '''Calculates the sum of the scalar products of the adjacent spins to the
    spin at input_coord. If is_pyomo is True it returns this as the sum of the
    pyomo variables, else it returns the floatof the sum from the numpy array
    containg spins'''
    input_spin = _get_spin(input_coord, problem, is_pyomo, spin_array)
    # List of the spin values in the adjacent coordinates to input_coord
    adj_spin_list = [_get_spin(Coord(input_coord.i-1, input_coord.j), problem,
                               is_pyomo, spin_array),
                     _get_spin(Coord(input_coord.i+1, input_coord.j), problem,
                               is_pyomo, spin_array),
                     _get_spin(Coord(input_coord.i, input_coord.j-1), problem,
                               is_pyomo, spin_array),
                     _get_spin(Coord(input_coord.i, input_coord.j+1), problem,
                               is_pyomo, spin_array)]

    # Sum of the scalar product of spins
    if is_pyomo:
        return sum(pyomo.environ.cos(input_spin - adj_spin)
                   for adj_spin in adj_spin_list)
    else:
        return sum(np.cos(input_spin - adj_spin)
                   for adj_spin in adj_spin_list)


def _diag_sum(input_coord, problem, is_pyomo, spin_array):
    '''Calculates the sum of the scalar products of the closest diagonal spins
    to the spin at input_coord. If is_pyomo is True it returns this as the sum
    of the pyomo variables, else it returns the float of the sum from the numpy
    array containing spins'''
    input_spin = _get_spin(input_coord, problem, is_pyomo, spin_array)
    # List of the spin values in the closest diagonal coords to the input_coord
    diag_spin_list = [_get_spin(Coord(input_coord.i-1, input_coord.j-1),
                                problem, is_pyomo, spin_array),
                      _get_spin(Coord(input_coord.i+1, input_coord.j-1),
                                problem, is_pyomo, spin_array),
                      _get_spin(Coord(input_coord.i-1, input_coord.j+1),
                                problem, is_pyomo, spin_array),
                      _get_spin(Coord(input_coord.i+1, input_coord.j+1),
                                problem, is_pyomo, spin_array)]
    # Sum of the scalar product of spins
    if is_pyomo:
        return sum(pyomo.environ.cos(input_spin - diag_spin)
                   for diag_spin in diag_spin_list)
    else:
        return sum(np.cos(input_spin - diag_spin)
                   for diag_spin in diag_spin_list)


def _python_hamiltonian(problem, spin_array):
    '''Returns the Hamiltonian using standard Python code (as opposed to Pyomo
    objects), applied to the given numpy array: spin_array'''
    tot_adj = 0.0  # Total of the adjacent scalar products
    tot_diag = 0.0  # Total of the diagonal scalar products
    tot_mag = 0.0  # Total of the magnetic field interaction terms
    tot_aniso = 0.0  # Total of the terms from single-ion-anisotropy

    is_pyomo = False  # Indicates this is not for Pyomo

    for input_coord in problem.coord_array:
        # Loops over all the coordinates in the grid and adds to totals
        input_spin = _get_spin(input_coord, problem, is_pyomo, spin_array)
        tot_adj += _adj_sum(input_coord, problem, is_pyomo, spin_array)
        tot_diag += _diag_sum(input_coord, problem, is_pyomo, spin_array)
        tot_mag += np.cos(problem.mag_ang - input_spin)
        tot_aniso += (np.sin(problem.aniso_ang - input_spin))**2

    return -problem.adj_coup*tot_adj - problem.diag_coup*tot_diag \
        - problem.mag_strength*tot_mag \
        + problem.aniso_strength*tot_aniso


class Coord(object):
    'Class for a coordinate object i.e. i - row number, j - column number'
    def __init__(self, i, j):
        self.i = i
        self.j = j


class SpinProblem(object):
    '''Class which takes in the parameters of an interacting spin problem and
    solves it with a choice of solver'''
    def __init__(self, grid_dimen, boundary="periodic", start_spins=None,
                 bas_vec1=(1.0, 0.0), bas_vec2=(0.0, 1.0), adj_coup=1.0,
                 diag_coup=0.0, aniso=(0.0, 0.0), mag_fiel=(0.0, 0.0)):
```

```
    '''
    Input explanation:
    * grid_dimen  - Tuple containing dimensions of grid.
    I.e. (m,n) -> m rows, n columns
    * boundary    - Boundary conditions of the grid. Either "periodic"
    or "zero"
    * start_spins - Initial condition for the spins. If None, then all
    spins will be 0.0. Else they are given in a numpy array
    * bas_vec1    - First basis vector of the lattice given as a tuple
    * bas_vec2    - Second basis vector of the lattive given as a tuple
    * adj_coup    - Coupling term between adjacent neighbouring spins
    * diag_coup    - Coupling term between spins on the closest
    * aniso       - Tuple representing single-ion-anisotropy.
    First term is strength, second is angle (in radians) that it points in
    * mag_fiel    - Tuple representing magnetic field. First term is
    strength,second is angle (in radians) that it points in
    '''
    # ___Stores the input parameters as class variables___

    self.row_no = grid_dimen[0]
    self.col_no = grid_dimen[1]
    self.boundary = boundary

    if start_spins is None:
        self.start_spins = np.zeros((self.row_no, self.col_no))
    else:
        self.start_spins = start_spins

    self.bas_vec1 = Vector(x=bas_vec1[0], y=bas_vec1[1])
    self.bas_vec2 = Vector(x=bas_vec2[0], y=bas_vec2[1])

    self.adj_coup = adj_coup
    self.diag_coup = diag_coup

    self.aniso_strength = aniso[0]
    self.aniso_ang = aniso[1]

    self.mag_strength = mag_fiel[0]
    self.mag_ang = mag_fiel[1]

    # Variables to store the solver data we want
    self.results_array = None  # Array to store optimised spins
    self.solve_time = None  # Time the solver took to solve the  problem
    self.result_obj = None  # The resulting objective function

    # Derived variables
    self.no_spins = self.row_no * self.col_no  # Total number of spins
    # Array to hold then Coord objects in for the grid points
    coord_array_2d = np.array([[Coord(i, j) for j in range(self.col_no)]
                               for i in range(self.row_no)])
    self.coord_array = coord_array_2d.flatten()

def scipy_solve(self, solver):
    '''Solves the problem using a SciPy solver specified by the input,
    where "solver" is a string, i.e. solver = "Nelder-Mead"'''
    # Clearing the solution variables
    self.results_array = None
    self.result_obj = None
    self.solve_time = None

    start_spins_1d = self.start_spins.flatten()  # Array of intial spins

    solver_output = None  # Variable that stores the solver output
    time_1, time_2 = 0.0, 0.0  # Variables used for timing the solvers

    def _1d_hamiltonian(spin_array_1d):
        '''Gives a 1D numpy array of the spins as inputs and returns the
        Hamiltonian'''
        spin_array_2d = spin_array_1d.reshape((self.row_no, self.col_no))
        return _python_hamiltonian(self, spin_array_2d)

    if solver == "Diff_Evo":
        # Differential evolution requires different conditions
        bounds = [(0, 2*np.pi) for i in range(self.no_spins)]

        time_1 = time.time()
        solver_output = scipy.optimize.differential_evolution(
            _1d_hamiltonian, bounds)
        time_2 = time.time()
    else:
        # Any optimize.minimize solver
        time_1 = time.time()
        solver_output = scipy.optimize.minimize(
            _1d_hamiltonian, start_spins_1d, method=solver)
        time_2 = time.time()

    # Spin results
    results_1d = solver_output.x
    self.results_array = results_1d.reshape((self.row_no, self.col_no))

    # Objective function
    self.result_obj = solver_output.fun

    # Solve time
    self.solve_time = time_2 - time_1

def pyopt_solve(self, solver):
    '''Solves the problem using a pyOpt solver specified by the input,
    where "solver" is a pyOpt object, i.e. solver = pyOpt.SOLVOPT()'''
    # Clearing the solution variables
    self.results_array = None
    self.result_obj = None
    self.solve_time = None

    def opt_func(spin_vars):
        'The optimisation function used by pyOpt'
        # List of the variables used for spins
        spin_list = [spin_vars[i] for i in range(self.no_spins)]
```

```
            # 2D numpy array for the spin list
            spin_array = np.asarray(spin_list).reshape((self.row_no,
                                                        self.col_no))

            obj_fun = _python_hamiltonian(self, spin_array)  # Objective func

            constr_list = []  # List of constraints)

            fail = 0  # Indicates the success of the solver

            return obj_fun, constr_list, fail

        # Sets up the pyOpt optimization
        opt_prob = pyOpt.Optimization('Spin system', opt_func)

        for input_coord in self.coord_array:
            # Adds the variables for the spins as pyOpt variables
            spin_name = 's[' + str(input_coord.i) + ', ' \
                    + str(input_coord.j) + ']'
            this_start_spin = self.start_spins[input_coord.i, input_coord.j]
            opt_prob.addVar(spin_name, lower=0.0, upper=2*np.pi,
                            value=this_start_spin)

        opt_prob.addObj('obj_fun')  # Tells pyOpt what the objective function

        solver(opt_prob)  # Tells the solver to solve the optimization problem

        # Set of solutions for the spin
        var_set = opt_prob.solution(0).getVarSet()
        results_list = [var_set[i].value for i in range(self.no_spins)]
        self.results_array = np.asarray(results_list).reshape((self.row_no,
                                                               self.col_no))

        # Objective function
        pyopt_result_obj = opt_prob.solution(0).getObj(0)
        self.result_obj = pyopt_result_obj.value

        # Solve time
        opt_prob.solution(0).write2file('pyOpt_output.txt')  # Output file

        # Searches the output file for the time measured by pyOpt
        for line in open('pyOpt_output.txt').readlines():
            if line.find('Total Time:') != -1:
                line_split = line.split()
                self.solve_time = line_split[line_split.index("Time:") + 1]
                break

    def pyomo_solve(self, solver):
        '''Solves the problem using a Pyomo solver specified by the input,
        where "solver" is a string, i.e. solver = "ipopt"'''
        # Clearing the solution variables
        self.results_array = None
        self.result_obj = None
```

```
self.solve_time = None

# Setting up the Pyomo model object
self.model = pyomo.environ.ConcreteModel()

# Range sets for the grid points
self.model.row_points = pyomo.environ.RangeSet(0, self.row_no - 1)
self.model.col_points = pyomo.environ.RangeSet(0, self.col_no - 1)

# Initialisation of the spins
self.model.start_spins = self.start_spins

# Pyomo variable for the spins
self.model.s = pyomo.environ.Var(self.model.row_points,
                                 self.model.col_points,
                                 domain=pyomo.environ.NonNegativeReals,
                                 initialize=_init_spins_rule)

def _pyomo_hamil(pyomo_model):
    'The Hamiltonian function using the pyomo objects'
    tot_adj = 0.0  # Total of the adjacent scalar products
    tot_diag = 0.0  # Total of the diagonal scalar products
    tot_mag = 0.0  # Total of the magnetic field interaction terms
    tot_aniso = 0.0  # Total of the terms from single-ion-anisotropy

    for input_coord in self.coord_array:
        input_spin = _get_spin(input_coord, self, is_pyomo=True,
                               spin_array=None)
        tot_adj += _adj_sum(input_coord, self, is_pyomo=True,
                            spin_array=None)
        tot_diag += _diag_sum(input_coord, self, is_pyomo=True,
                              spin_array=None)
        tot_mag += pyomo.environ.cos(self.mag_ang - input_spin)
        tot_aniso += (pyomo.environ.sin(self.aniso_ang -
                                        input_spin)) ** 2

    return -self.adj_coup*tot_adj - self.diag_coup*tot_diag - \
        self.mag_strength*tot_mag + self.aniso_strength*tot_aniso

# Objective function (the Hamiltonian)
self.model.OBJ = pyomo.environ.Objective(rule=_pyomo_hamil,
                                         sense=pyomo.environ.minimize)

# Solves the model
opt = SolverFactory(solver)
opt.solve(self.model, logfile=solver+'.txt')

# Creates the results array
self.results_array = np.zeros((self.row_no, self.col_no))
for i in self.model.row_points:
    for j in self.model.col_points:
        self.results_array[i, j] = \
                    pyomo.environ.value(self.model.s[i, j])
        self.result_obj = pyomo.environ.value(self.model.OBJ)
        # Searches the log file to find the solving time
        line_str = "Total CPU secs in IPOPT (w/o function evaluations)   ="
        if solver == 'ipopt':
            for line in reversed(open(solver+'.txt').readlines()):
                if line.find(line_str) != -1:
                    line_split = line.split()
                    self.solve_time = line_split[line_split.index("=") + 1]
                    break

    def plot(self, plot_name=None):
        'Plots the results of the optimization problem'
        # Lists that store the x and y positions in real space of the spins
        x_pos, y_pos = [], []
        # Lists that store the components of the vectors of the spins
        x_spin_comps, y_spin_comps = [], []

        for spin_coord in self.coord_array:
            # Loops over the spins and fills the lists
            x_pos.append(spin_coord.i * self.bas_vec1.x +
                         spin_coord.j * self.bas_vec2.x)
            y_pos.append(spin_coord.i * self.bas_vec1.y +
                         spin_coord.j * self.bas_vec2.y)
            x_spin_comps.append(np.cos(self.results_array[spin_coord.i,
                                                          spin_coord.j]))
            y_spin_comps.append(np.sin(self.results_array[spin_coord.i,
                                                          spin_coord.j]))

        plot_spins((x_pos, y_pos, x_spin_comps, y_spin_comps))
```

C.2 spin_plot.py

```
'Functions for plotting a 2D lattice of interacting spins'

import csv
import math
import matplotlib.pyplot as plt


def plot_spins(input_list, plot_name=None):
    '''Plots the configuration of spins, where input_list =
    (x_pos, y_pos, x_spin_comps, y_spin_comps), where x_pos and y_pos are
    arrays indicating the points where the spins are, and x_spin_comps and
    y_spin_comps are the components of the vectors of the spins at
    those points'''
    x_pos, y_pos = input_list[0], input_list[1]
    x_spin_comps, y_spin_comps = input_list[2], input_list[3]

    plt.figure()
    axes = plt.gca()
    plt.gca().set_aspect('equal', adjustable='box')

    axes.quiver(x_pos, y_pos, x_spin_comps, y_spin_comps)
    axes.plot(x_pos, y_pos, 'or')
    axes.set_xlim([min(x_pos) - 1, max(x_pos) + 1])
    axes.set_ylim([min(y_pos) - 1, max(y_pos) + 1])
    axes.set_xticks([])
    axes.set_yticks([])

    if plot_name is None:
        plt.draw()
        plt.show()
    else:
        plt.title(plot_name)
        plt.draw()
        fig_name = plot_name + ".png"
        plt.savefig(fig_name)
        plt.show()


def plot_from_file(filename, plot_name=None, bas_vec_1=(1.0, 0.0),
                   bas_vec_2=(0.0, 1.0)):
    '''Plots spins given from a csv file, depending on basis vectors.
    Default is square lattice'''
    grid = []
    with open(filename, 'r') as data:
        rows = csv.reader(data)
        grid = [[float(cell) for cell in row] for row in rows]

    x_pos, y_pos, x_spin_comps, y_spin_comps = [], [], [], []
    row_no = len(grid)
    col_no = len(grid[0])
    for i in range(row_no):
        for j in range(col_no):
            x_pos.append(i*bas_vec_1[0] + j*bas_vec_2[0])
            y_pos.append(i*bas_vec_1[1] + j*bas_vec_2[1])
            x_spin_comps.append(math.cos(grid[i][j]))
            y_spin_comps.append(math.sin(grid[i][j]))

    plot_spins((x_pos, y_pos, x_spin_comps, y_spin_comps), plot_name)
```