System of interacting spins: Code documentation

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

September 14, 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 materials.

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. Dive in!

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 do give 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 **A**. 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 now 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. As before, the first term, `grid_dimen`, is a tuple giving the dimensions of the grid. The second term 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 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.

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 two 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 `GridCoord`s closest neighbouring coordinates in grid space.

`CoordList` is a class for lists of `GridCoord`s. This only really exists to override some methods in the standard Python lists, so that functions such as `in` work when applied to `GridCoord`s. Most likely, you won't need to use this class. But if you have a list of `GridCoord`s, 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 main class to 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 specifies that it conforms 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 may 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 may 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 manually change the components. 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 `PyomoVectors` 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 `PyomoVectors`.

`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 merely 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 defined. 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:

```
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 `Shell`s 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 `Shell`s. 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 spin 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 an 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!