Documentation for the `cmp.py` script

1 Introduction

The program in this project is written in the open source programming language Python, using the standard library, along with the NumPy and Matplotlib packages. As such those two packages are all you really need to run the software.

It is however recommended that you install Anaconda, which is a Python distribution that also includes both NumPy and Matplotlib, along with a wealth of other packages useful for scientific computing. The Jupyter package, especially, is useful, as it allows the user to write a bunch of small scripts (called cells) in a single file (called a notebook), and run the cells individually and quickly, while sharing the workspace between cells (i.e., variables persist between cells). A quick intro to Jupyter notebooks can be found at this link.

2 Installation

As recommended we will install the Anaconda distribution. Download the software from here: `https://www.anaconda.com/download/`. Install the software, and when asked if you want to include Anaconda to the path (or something along those lines) say: "Yes". This will allow you to start Python from your command line/terminal with the command `python`.

When Anaconda has been installed, you need the files from the project, these can be found on GitHub, at `https://github.com/NikolaiNielsen/Bachelor` (Or a direct download link).

Extract the code from the archive. The files `cmp.py`, `lattices.py`, `scattering.py`, `band_structure.py` are all necessary, while `test.ipynb` includes some code examples.

To run the code you can do one of a couple of things:

- Start Jupyter notebook (run the command `jupyter notebook` from the command line/terminal), navigate to the directory where the code is located, start the test notebook (`test.ipynb`) and write your code there.
- Navigate, using your command line/terminal, to the directory where the code is located and run the command `python`. This creates an interactive python session where you can type in each command manually.

The reason why you need to navigate to the folder containing the code is because otherwise Python will not know where to look for the package, when you run the first line in the usage section. Python first checks the current working directory and next any other folders added to `$PYTHONPATH`. If your current working directory is the directory containing the code, then python can correctly import the package.

3 Usage

The first commands you need to type is

```
1 from cmp import *
2 %matplotlib notebook
```

The first line will import the 4 main functions, `Lattices`, `Reciprocal`, `Scattering` and `Band_structure`, along with the numpy package (accessed as `np`) and pyplot package (a subpackage of matplotlib, accessed as `plt`). The second line is only needed when using Jupyter notebooks, and makes the plots interactive (this line may need to be executed twice for it to work properly).

Now you can start using the 4 functions to plot things. The functions take the following arguments:

3.1 `Lattice()`

The desired crystal structure can be specified in two ways:

- The name of the Bravais Lattice Type can be specified as the `lattice_name` argument, accepting a string value (see table 1 and 2 in the thesis for available names).
- The lattice vectors (`a1`, `a2`, `a3`) can be specified as 3 ndarrays, containing 3 elements each. The `basis` can be specified as an ndarray, containing n rows and 3 columns.

There is also a list of optional arguments that can be passed to the Lattice function:

- `colors`: a list/tuple with n strings specifying the colors of each of the atoms in the basis. A single string can also be given, if only one color is desired. Defaults to plotting
- `sizes`: a list/tuple of n numbers specifying the relative size of the atoms. 2 is the default value.
- `grid_type`: either "latticevectors" or "axes". "latticevectors" plots gridlines along the lattice vectors. "axes" tries to plot gridlines parallel to the x, y and z-axes (requires a crystal structure that can be specified as a orthorhombic lattice with basis). Defaults to the preffered type of gridlines for a given Bravais lattice type
- `max_`: a list/tuples specifying the maximum amount of lattice points to plot in each direction of the lattice vectors. Defaults to [2,2,2].
- `lim_type`: either "individual", "sum" and "proper". Provides 3 different ways of limiting the plotted crystal structure: The default (and recommended) is "proper", which calculates the 8 lattice points given by linear combinations of the lattice-vectors and the coefficients specified in `min_` and `max_`. This forms a parallelipiped, where a orthorhombic plot-box is calculated to fit these 8 vertices inside.
- `indices`: a list/tuple of 3 elements, corresponding to the Miller indices h, j and l. If specified this plots the corresponding (family of) lattice planes. For the Lattice function, this argument defaults to None, meaning no planes are plotted.
- `unit_type`: Either "primitive" and "conventional". Specifies the type of unit cell to plot. "primitive" plots only the lattice vectors up to `max_`, while "conventional" fills out the plot box. Default is "primitive", but when specifying `lattice_name` it defaults to the preferred value for that lattice.
- `verbose`: True/False boolean value. Makes the functions print a lot of information helpful for debugging. Only partially implemented. Defaults to False
- `returns`: True/False boolean value. If true the function returns the figure and axes handles. Defaults to False

3.2 `Reciprocal()`

This function is actually just a copy of the `Lattice` function, with a default argument for indices of (1,1,1). Otherwise everything is identical.

3.3 `Scattering()`

This function simulates neutron scattering on a simple cubic lattice with a basis. It plots two figures in one window. One holds the crystal, a representation of the incoming neutron beam, the detection screen and the points as detected by this screen. The second figure shows just the detection plane and the points detected by it, along with the Miller indices giving rise to this particular scattering event. The alpha value of the points on the screen corresponds to the normalized intensity of the scattering events.

The function needs 3 pieces of information to work properly. The basis, incoming wave vector and scattering lengths for the atoms in the basis.

The basis can be specified in the following ways:

- `lattice_name`: a string, specifying the Bravais Lattice Type (currently it only accepts "simple cubic", "bcc" and "fcc")
- `basis`: an ndarray, with n rows and 3 columns, specifying the absolute position of the atoms within the unit cell (assuming lattice spacing of 1)

The two other requires arguments are

- `k_in`: an ndarray with 3 numbers.. The script simulate a neutron beam incident on the top of the material. As such, the z-component should be negative. If positive, the script will flip the sign. If the argument `normalize` is True, then `k_in` is defined in units of $2\pi/a$ (with $a = 1$ in the program).
- `form_factor`: an ndarray with n numbers, where n is the number of atoms in the basis. Due to normalization of the intensity, only the relative values of elements in `form_factor` matter. As such it is recommended to input values near unity to minimize any rounding error. By default all scattering lengths are equal

Other optional arguments that can be passed are as follows:

- `highlight`: a list/tuple of 3 elements. Corresponds to the Miller indices h, j and l. This will highlight the given scattering event associated with these Miller indices (if present). Defaults to None
- `show_all`: True/False boolean. If set to true this will plot all outgoing wave vectors (with lengths equal to their wavelength), and lines going from the scattering point out to the points on the detection plane. Defaults to False
- `normalize`: True/False boolean. If true, the incident wave vector is defined in units of $2\pi/a$ with $a = 1$. (in other words, `k_in` is multiplied by 2π). Defaults to True.
- `verbose`: True/False boolean. Prints information useful in debugging. Also only partially implemented. Defaults to False
- `returns`: True/False boolean. If True the program returns the figure handle and the two axes handles. Defaults to False

3.4 `Band_structure()`

The band structure function takes the following main arguments:

- `V0`: A float. The potential strength, in units of $E_0 = \hbar^2 k_0^2/m$. For the harmonic potential `V0`=1 is the high potential limit, and any value above it will not provide the desired result. For the Dirac potential there is no bound on `V0` other than the floating point accuracy and the eigenvalue-finding algorithm (it breaks down around `V0` $= 10^{15}$). Defaults to 0
- `n_k`: An integer. The number of points (in each direction) to include. It is suggested that this is an odd number to correctly get the points with $k_x = 0$ and $k_y = 0$. Defaults to 51

- `G_range`: A list of integers. The list of allowed coefficients for m_1 and m_2, where $\mathbf{G} = m_1\mathbf{b}_1 + m_2\mathbf{b}_2$. Should be symmetrical about 0, like `G_range = [-2, -1, 0, 1, 2]`. Defaults to [-3, -2, -1, 0, 1, 2, 3].

- `potential`: "harmonic", "dirac" or a function handle. specifies the potential. If "harmonic" or "dirac" is *not* specified, but instead a function is given, the program will use this function to calculate the potential matrix. The function provided needs to take two arguments: `coeff` and `V0`, where `coeff` is a list of 2 integers: [m_1, m_2] and `V0` of course is V_0/E_0.

There are also the following optional arguments:

- `contour`: True/False boolean. If True, the program plots the Fermi surface contour in the main figure as well as the surface plot of the dispersion relation. Defaults to False

- `edges`: True/False boolean. If True, the program plots the edges of the dispersion relation in another colour (black). Defaults to False.

- `returns`: True/False boolean. If True the program returns the figure handle and the two axes handles. Defaults to False

4 Examples

Say we want to plot an fcc with the conventional unit cell. This can be done in two ways:

```
Lattice(lattice_name='fcc')
```

Or

```
Lattice(a1=np.array([1, 0, 0]),
        a2=np.array([0, 1, 0]),
        a3=np.array([0, 0, 1]),
        basis=np.array([[0, 0, 0],
                        [0.5, 0.5, 0],
                        [0.5, 0, 0.5],
                        [0, 0.5, 0.5]]))
```

If we want to plot a simple cubic lattice with the (1,1,1) family of lattice planes we write

```
Reciprocal(lattice_name="simple cubic",
           indices=(1, 1, 1))
```

If we want scattering on a bcc lattice, with k_in at normal incidence (magnitude of $2\pi/a$), while higlighting the $(0, 0, 2)$ planes and plotting all information (with all form factors being equal, which is the default):

```
Scattering(lattice_name='bcc',
           k_in=np.array([0, 0, -1]),
           highlight=(0,0,2),
           show_all=True)
```

If we want to plot the band structure of a medium strength harmonic potential (say $V_0/E_0 = 0.3$), with 101 points in each direction we write

```
Band_structure(V0=0.3,
               n_k=101,
               potential='harmonic')
```

5 Glossary

`np` is the numpy package. This is the standard python package for doing numerical calculations.

A `list` is a comma separated list of elements, enclosed in square brackets:

```
a = [1,2,3]
```

A `tuple` is like a list, but enclosed in parentheses:

```
b = (1,2,3)
```

An `ndarray` is the multidimensional array type specified by numpy. This corresponds to the default Matlab arrays. It is invoked by passing a list/tuple of lists/tuples to the `np.array()` function. Say I want to create a unit vector in the x-direction. Then I write

```
x = np.array([1,0,0])
```

If I want to specify the basis for a bcc lattice (conventional unit cell) I need to pass a list of lists. Each sub-list is then a row in the resulting array. Do note the double square brackets: ([[],[]])

```
basis = np.array([[0,0,0],
                  [0.5, 0.5, 0.5]])
```