Overview of Programs for Total Enumeration Analysis

Nathaniel D. Hoffman

May 13, 2020

Abstract

The purpose of this document is to explain the programs I wrote to analyze Heusler structures in the Spring of 2020. Most of these programs will, I believe, work well with arbitrary structures, and they all can be extended to any form of model which uses simple scalar multipliers to describe nearest and next-nearest neighbor interactions. I have also documented each of my programs within the code itself.

Contents

1 Python Setup

1.1 Dependencies

I ran all of the `python` code using Python 3.7.5, although it should run fine on Python 3.8 and future versions as long as the dependencies are also updated. At time of writing, the dependencies are listed in the `requirements.txt` file:

```
scipy==1.4.1
plotly==4.6.0
numpy==1.18.2
pandas==1.0.3
```

This should be all of the `python` libraries required to run all of the code I have written. Scipy and Numpy deal with all of the mathematical methods of analysis, Plotly is used for all of the plotting, and Pandas is primarily used to organize, import, and export large datasets. Again, I believe this code will run on future versions of these libraries. However, the `plotly` library must be installed specifically as the specified version (or a later one). By default, `pip3 install plotly` will currently install a previous version of the library which is rather outdated.

1.2 Installing and Running

The advent of `python` Virtual Environments makes it very easy to install just the packages required for this specific set of tools while keeping your main `python` distribution unchanged. It also allows for simple control of dependency versions and does not require administrator access to set up.

However, the original installation of `virtualenv` must be performed first. The following code will install it at a `user` access level.

```
python3 -m pip install --user virtualenv
```

Next, navigate to the directory with the analysis tools. The following command will initialize a virtual environment in a folder called `env`:

```
python3 -m venv env
```

To activate this virtual environment, simply source the activation script:

```
source env/bin/activate
```

Alternative sourcing scripts for `tcsh` or the `fish` shells are also included in this folder. To exit the virtual environment, use the command

```
deactivate
```

While in a virtual environment, we can install the required packages using the command

```
pip3 install -r requirements.txt
```

All of the `python` programs I have written can then be run in the virtual environment using

```
python3 <program name>.py
```

While I have set up all of the `python` files with a shebang (`#!`), these will not point to a virtual environment. They can only be run without the `python3` command if the packages have been installed on the system version of `python` (at least at the `user` level). This can be achieved by running

```
pip3 install -r --user requirements.txt
```

After this, it should be possible to run all of the `python` programs with

```
./<program name>.py
```

They will also run using the `python3` command.

2 Analysis.py

Running the program with the `-h` or `--help` argument will display a short manual for the required inputs of this program:

```
python3 Analysis.py --help
```

2.1 Input Files

There are two files which must be included (in the following order) in order to perform any analysis. These are simple an `XYZ` file (which uses the typical syntax and formatting obtained from `mkpearson`) and an `struct_enum.out` file, which is obtained from running the `enum.x` program. If these required files have been provided, the program by itself will not do anything. The user must also specify at least one argument and a supplementary file (or number) as described in the following sections.

2.2 The `--map` Argument

This command simply generates a heatmap-like graph to visualize lowest energy states using the "J-K" model. It requires an additional `CSV` file which contains rows and columns of configuration IDs (matching those given in `struct_enum.out`) and is generated using the program `EnergyMinimization_v2.f90`. Running

```
python3 Analysis.py XYZ struct_enum.out --map <filename>.csv
```

will produce and display the desired plot using `plotly`'s usual `HTML` format. These plots can be saved as `HTML` files from the web browser or as images using the camera icon in the plot window.

The `--map` argument is outdated, since we realized it was possible to condense the search over J and K values. This is discussed in the next subsection.

2.3 The `--rad` Argument

Like the `--map` argument, this also takes a single `CSV` file. However, this file provides the data from a radial sweep over J and K values. This can be generated from the `EMin_v3.f90` program (the method of generation is

discussed in the corresponding section). The file will have columns of J, K, θ, and the configuration ID, in that order. Degenerate configurations will have rows which are identical in every column except for the ID. I don't believe the rows actually have to be in an order of increasing θ, so in the future, if someone were to use some sort of asynchronous aggregation to generate the file, this program should still work. Also like the `--map` argument, a plot can be generated using

```
python3 Analysis.py XYZ struct_enum.out --rad <filename>.csv
```

This will generate a plot which looks more like a pie chart. Degenerate structures will show up in different colors within the same pie slice (the radial position is meaningless) while the angular position will indicate the J and K value used.

2.4 The `--draw` Argument

The final argument does not take an additional file, but instead a numerical configuration ID which can be found in either of the plots generated from the method above or simply from the `struct_enum.out` file. Running

```
python3 Analysis.py --draw ####
```

will produce a 3D crystal structure plot which can be rotated and zoomed. By default, it will contain just the lattice points described in the `XYZ` file, but there is a section of commented-out code which can be uncommented to plot a single repetition of the structure in every direction (expanding from a unit cell to a $3 \times 3 \times 3$ cluster of unit cells).

3 `EnergyMinimization_v2.f90`

This file contains the source code for a `FORTRAN` program to generate the square plots of energy minimizing configurations for a range of J and K values. It does not aggregate degeneracies, which is why the later version, `EMin_v3.f90` is recommended. By default, this program will generate a 201×201 (this is hard-coded but can be modified by changing the declaration of the `STEPS` variable) grid of energy-minimizing configuration IDs in a `CSV` file (it will send them to `stdout` but they should be piped to a `CSV` file for use in the `Analysis.py` program). It will also use a diagonal interaction matrix,

described in the variable `intmatrix`. This is a three-dimensional array of size $N \times N \times 2$, where N is the number of species. By default, I initialize it with $N = 4$ because I have set it up to also run fine on $N < 4$ without requiring any changes to the interaction matrix. The interaction matrix can be thought of as having two layers, the first for the nearest-neighbor interactions and the second for the next-nearest neighbor interactions. In our default J-K model, our Hamiltonian can be written as

$$H = \sum_{<i,j>} J\delta_{\sigma_i,\sigma_j} + \sum_{\ll i,j \gg} K\delta_{\sigma_i,\sigma_j} \tag{1}$$

where $< i, j >$ and $\ll i, j \gg$ correspond to the set of nearest and next-nearest neighbor pairs of lattice points respectively, and σ_i refers to the elemental species of the atom located at the ith lattice point. The interaction matrix for this Hamiltonian can be written as

$$M_{ij1} = \begin{pmatrix} J & 0 & 0 & 0 \\ 0 & J & 0 & 0 \\ 0 & 0 & J & 0 \\ 0 & 0 & 0 & J \end{pmatrix} \tag{2}$$

$$M_{ij2} = \begin{pmatrix} K & 0 & 0 & 0 \\ 0 & K & 0 & 0 \\ 0 & 0 & K & 0 \\ 0 & 0 & 0 & K \end{pmatrix} \tag{3}$$

In this way, it is clear to see that the on-diagonal terms of the matrix correspond to the particles interaction with like-species while the M_{ij} element of each matrix corresponds to the interaction between the ith and jth species.

During this project, we also developed an alternative Hamiltonian which only applied to the four-species structures:

$$H = \sum_{<i,j>} J\delta_{\chi_i,\chi_j} + \sum_{\ll i,j \gg} K\delta_{\sigma_i,\sigma_j} \tag{4}$$

where χ_i now refers to a grouping of species. Specifically for this model, we say that species 1 and 2 are in the same group while species 3 and 4 are in a different group. This adds a J-like (nearest-neighbor only) interaction term between species 1 and 2 and between species 3 and 4, which changes the first

layer of the matrix to be

$$M_{ij1} = \begin{pmatrix} J & J & 0 & 0 \\ J & J & 0 & 0 \\ 0 & 0 & J & J \\ 0 & 0 & J & J \end{pmatrix} \tag{5}$$

and leaves M_{ij2} unchanged.

This program (as well as `EMin_v3.f90`) takes two files as input, the first being the `struct_enum.out` file generated by `enum.x` and the second being a file which I typically name `INT` which holds data that describes which lattice positions are nearest and next-nearest neighbors. This file can be generated using `Generate_Interactions.py`. Because this program is relatively less useful than `EMin_v3.f90`, I have chosen to not document it or provide a compiled version and instead document the more up-to-date version.

4 EMin_v3.f90

This code functions almost identically to `EnergyMinimization_v2.f90` except it produces data in the radial format used by `Analysis.py` and also accounts for degeneracies. It generates 201 (again, this can be changed by modifying the `STEPS` variable) θ values for positions equally spaced around a unit circle. Values for J and K are generated using $J = \cos(\theta)$ and $K = \sin(\theta)$. Note that this actually results in the opposite axis assignment to that created by `EnergyMinimization_v2.f90`. This program takes an interaction file, `INT`, described in the previous section, and can be run using

```
EnergyMinimization.x struct_enum.out INT >> output.csv
```

This will generate data for the diagonal J-K model, while

```
EnergyMinimization_Widom.x struct_enum.out INT >> output.csv
```

will generate data for the block-diagonal model mentioned in the previous section. In general, I use the suffix `_Widom` to describe data generated with that model, omitting the suffix for the diagonal model.

5 Generate_Interactions.py

This is a simple program that generates a `CSV` file containing information about the interactions between particular lattice sites in a crystal. It takes the standard `XYZ` file as an argument and generates three columns of data. The first and second column refer to the ith and jth lattice point while the third is 1 for a nearest-neighbor interaction and 2 for a next-nearest-neighbor interaction. The program itself generates these with periodic boundary conditions in mind by using a $3 \times 3 \times 3$ block of unit cells and measuring the bond lengths between each lattice point in each cell to each lattice point in the center cell. It then takes the two smallest (non-zero) distances and outputs all pairs of lattice points which are that distance away from each other. The program can be run using

```
python3 Generate_Interactions.py XYZ >> INT
```

6 JK_Map.f90

This program is very similar to the minimization programs described above, but rather than finding the structure with the minimal energy, it lists how many J and K-like bonds are present in every configuration listed in the `struct_enum.out` file. It will output three columns of data corresponding to J, K, and a string of numbers corresponding to the configurations given in `struct_enum.out`. For Heusler structures, these will be strings of 16 numbers ranging between 0 and 3 (for each species).

I again include two compiled versions of this program corresponding to the two models discussed. They can be run using

```
JK_Map<_Widom>.x struct_enum.out INT >> JKCONFIG<.csv>
```

7 JK_Scatter.py

This is a very short and simple program which takes the map file generated by `JK_Map.f90` as its only input and draws a scatter plot of the configurations and their corresponding locations in J-K space. There will typically be a lot of degenerate configurations in this regard, so only one configuration will be displayed at each point if there are multiple configurations with the

same J and K values. This program will also take the first value at each of these degenerate points and output a condensed version of the output from `JK_Map.f90` which only includes one of each degenerate point. The program is executed by

```
python3 JK_Scatter.py JKCONFIG >> JKCONFIG_Reduced<.csv>
```

8 GenXYZ.py

This program generates a group of directories containing `XYZ` files for various configurations given in an input file generated by `JK_Map.f90` (or a reduced version from `JK_Scatter.py`). A description of this program can be viewed by running it with the `-h` or `--help` argument. It takes three files, an original `XYZ` file, an output directory, and a `JKCONFIG` file as mentioned above:

```
python3 GenXYZ.py XYZ <output directory> JKCONFIG
```

This program will generate subdirectories in the `<output directory>` which each contain a new `XYZ` file corresponding to each of the configurations in `JKCONFIG`. The filenames will be generated by converting the configuration string into a number `UID` where `UID` is the base-10 representation of the configuration string if that string were thought of as a base-N number, with N being the number of species. For instance, in the two-species case, it will pretend the configuration string is a binary number and convert it to decimal. In the three-species case, it will pretend it is a base-3 number. The output directory will be filled with subdirectories named `variation_<UID>`.

9 JK_fitter.py

The final program I created compares the J-K model to `VASP` simulations. The user inputs two files, one being a `JKCONFIG` file from the programs above and the second being a file containing two columns, the first being the name of the variation file (generated by `GenXYZ.py`) and the second being the energy from `VASP`:

```
python3 JK_fitter.py JKCONFIG ENERGY_DATA
```

will create two plots. The first is a 3D representation of the data being fit and the second is a parity plot of the data. I did not write a specific program to create the `ENERGY_DATA` file, but it's not hard to run a short script to aggregate the data. This program will also output three numbers, J, K, and an energy offset which are the results from fitting the data to a plane described by

$$E = Jx + Ky + E_{\text{offset}} \tag{6}$$

where x and y are the coordinates of a structure in J-K space and E_{offset} is some energy offset.

10 A Typical Workflow

I will now walk through the typical workflow for analyzing a particular structure described by an `XYZ` file. No other files are needed at first, as they will all be generated by this program. For clarity, assume all files are stored in the same directory.

```
cat XYZ | xyz2enum.sh -m ‘‘AlCo’’ -n 2 -s ‘‘0/1 0/1’’ >> enum.in
```

At the bottom of `enum.in`, we will have to add some lines to restrict concentrations (this is not required but usually what we want to do):

```
printf "1 1 2\n1 1 2\n" >> enum.in
```

This will set the concentration to 50% of each element.

```
enum.x enum.in
```

This will generate `struct_enum.out`.

```
python3 Generate_Interactions.py XYZ >> INT
EnergyMinimization.x struct_enum.out INT >> 2sp_degen_rad.csv
python3 Analysis.py XYZ struct_enum.out --rad 2sp_degen_rad.csv
```

This would display a radial plot of the energy-minimizing structures. We could look at the $B2$ structure (which I happen to know the enumeration ID of) with

```
python3 Analysis.py XYZ struct_enum.out --draw 59
```

So far we have operated entirely in the realm of mathematical theory. Now let us compare this model to simulations done in VASP to see how well this model works.

```
JK_Map.x struct_enum.out INT >> JKCONFIG
python3 JK_Scatter.py JKCONFIG >> JKCONFIG_REDUCED
mkdir vars
python3 GenXYZ.py XYZ vars JKCONFIG_REDUCED
cd vars
for d in variations.*; do cd $d; xyz2pos XYZ -mag co=3; qvasp; cd -; done
cd ..
```

We are now back in the main directory. Once the VASP simulations are complete (along with relaxation, if so desired), we can quickly grab the necessary data with a short bash script (shamelessly copied from a script written by Dr. Widom):

```
for name in variation*; do
    E=`awk '/E0=/{print 1.*$5}' $name/output | tail -1`;
    printf "%s %10.2f\n" $name $E >> ENERGY_DATA;
done
```

Finally, we can visualize this data with

```
python3 JK_fitter.py JKCONFIG ENERGY_DATA
```

Again, all of these analyses should work with any arbitrary XYZ file, not just Heusler structures and any kind of J-K-like model.