Research software engineering in Julia

Read the README.md file on [GitHub](https://github.com/Noldolino/Julia-Programming-Project) on how to run the code.

This is a project report of the assignment of the course “research software engineering in Julia”. The goal of this project was to write code of a current topic related to programming in Julia to get hands on experience. Additionally, the goal was to work with Github to learn clean programming practices as well as implementing tricks used in the course for optimizing code like using parallelism. All the written code is stored in Github under <https://github.com/Noldolino/Julia-Programming-Project> .

For my assignment, I used a python code named “Noldnold.py” from my colleague Hugo Gerlitz in the Fleischhauer AG in RPTU Kaiserslautern.

The goal was it to rewrite this python code in Julia and use parallelism to optimize the runtime of the now Julia code.

Here is a brief and simplified explanation of the topic of the physics that is being calculated:

The program is used to calculate theoretical one particle physics, where we consider a single particle in a 2D lattice. Under the influence of an external magnetic field the translation invariance will break, which makes it possible to define a unit cell of the new system. A Hamilton operator in form of a matrix is used to describe the system, which has to be diagonalized, to calculate the eigenenergy of the system, but this matrix can become arbitrarily large depending on our parameters. Therefore, by fourier-transforming we can decouple the kx- and ky modes, this results in many small Hamilton operators, which can be more easily diagonalized.

These smaller Matrices can be analytically calculated, so their form is generally known, meaning they can be numerically calculated.

The code will diagonalize the Matrices which will give different eigenvalues, which are the eigenenergies of the states. If an input is calculated, it will be saved in a dictionary to save up on compute time. The eigenvalues will be plotted over the kx and ky states, which will result in a 3D-plot. The probability of the particle sitting on one of the sites in the corresponding unit cell will be calculated and plotted. Lastly, I implemented Threading to use parallelism for faster runtime.

Firstly, I adapted parts of the python code to make it easier for me to rewrite it in Julia. I removed all the self. functions and wrote all the variables as global variables. This was, so I didn’t have to simulate the object-based coding with structs. The edited file is called “Noldnoldmodified.py”.

I then rewrote the whole python code in Julia, the file is called “Noldnoldmodified.py translated to Julia.jl”, where I had to import some packages like LinearAlgebra, SparseArrays etc. The uses of the imported packages are all documented in the file.

In the global variables the most noteworthy parameters are p: the magnetic field, q: the amount of lattice sites in a unit cell – p and q should be coprime to one another. x\_unitcells and y\_unitcells dictate the size of the lattice, while kx and ky are the position in the lattice – kx and ky should be between 0 and x\_unitcells/y\_unitcells. The band\_index dictates the corresponding band – this should be between 1 and q.

In OneParticle are all the variables defined- as in given a value.

Hamiltonian defines a Hamilton operator which is a matrix on a certain point of the lattice (for simplicity I will say “on a certain point” even though we are in a reciprocal k-space). kx and ky give the position in this lattice. Programming this was pretty straightforward.

Energy\_line is the first actual calculation. Here the Hamiltonian matrices of one kx site and all its ky sites will be diagonalized; the eigenvalues and vectors will then be saved in a dictionary.

Energy\_spectrum will diagonalize the remaining kx with all their ky Hamiltonian matrices. The values will be stored in a folder. This folder named "./dictionaries/one\_particle/" is the only thing that has to be considered for the code to run. The user has to make sure that this folder exists to run the code. If a combination of inputs was already calculated, the stored values will be used instead of recalculating them again.

Plot\_3d will, as the name suggest, plot the eigenenergies against kx and ky, which will create energy bands. The amount of energy bands should be equal to q.

Plot\_state will calculate the probability distribution of the particle to be in one of the sites of the unit cells. This probability changes depending on which energy band one is looking at. The inputs are nux, nuy -which can be between kx and ky – and band\_index which will be between 0 and q.

The user can then give their input in the last part the code, which runs the functions and plots and also measures the runtime with @time.

Lastly, I rewrote the functions energy\_line, energy\_spectrum and plot\_3d with @threads, which I put separately in Github with “Julia rewrite with parallelism.jl”. I also tested the runtime of all of these in separate files with @benchmark under the name “energy\_line benchmark testing.jl”, “energy\_spectrum benchmark testing.jl” and “3dplot benchmark testing.jl”. Obviously these three files can only be executed after “Julia rewrite with parallelism.jl” was executed once. I could observe a runtime increase of 20% with energy\_spectrum\_parallel”. The other two functions could not be improved with a small sample of lattices ranging from 100x100 to 300x300.

The Julia code itself on the other hand is about 4 times as fast as the python code with about 15 seconds for lattice sizes in the range of 100x100 to 300x300 for my laptop compared to about one minute with the python code.

This project as a beginner gave me a good introduction into Julia without being too overwhelming. There are a lot of possible improvements to be done, for example using threads for object-oriented coding or one of the many other optimization tricks that were covered in the lecture. The code could also be expanded in the future to use parallelism to compute more than one lattice at a time, instead of optimizing the diagonalization steps itself.