Week 3

0. Table of Contents

1. Goals

The main aim of this week is to begin reproducing a simple active matter system in code. To this end, a suitable environment to work in must be found and utilised. For now we are primarily sticking to python.

We have gained access to the Bristol supercomputers BlueCrystal4 and BluePebble (I have access to the former, my lab partner has access to the latter). We will attempt to apply the scripts to the supercomputers.

In Week 2 it was suggested that I would go more in depth in statistical analysis. This has been put aside for now, in the interest of getting a feel for how ABP simulations work in practice.

2. Configuring BlueCrystal4

I already have access to BlueCrystal4 (BC4) due to one of my other modules. For that particular module, I am programming in C/C++; as such, I have already configured (installed and loaded) modules compatible with it. These modules are generally installed on a user instance, and loaded using the following bash command: `module load <path>`. For example, for loading the necessary files for Intel's C/C++ compiler (icc), I use the following: `module load languages/intel/2020-u4`.

The issue that arises is that, at least initially, we have decided to work in Python. While I plan to attempt to translate (and speed up) our later codes in C, it is preferable to agree on a shared language while we get our bearings (and Python is easier, for all its faults). I therefore need use both C and python modules; hopefully this will not cause conflicts as long as they are not loaded simultaneously. The loading can easily be done in the `.bashrc` script on the user home instance, which will keep the module loaded for any and all processes. This is inadvised in the BC4 user manual, however - like I said, conflicts can occur. It is much healthier and safer to instead load modules in the `.sh` script that will send a request to the job queue. This will be elaborated on later.

You load/add modules with `module load` or `module add`. I have used the former for C and the latter for Python, though strictly to differentiate the two better. Below is a quick list of the different module commands I use:

```
module avail #Lists available modules; can combine with grep to search for a specific one

module load languages/intel/2020-u4 #Loads icc for C/C++
module add languages/anaconda3/2022.11-3.9.13-tensorflow-2.11 #Loads anaconda tensor flow
module add languages/anaconda3/2020-3.8.5 #Loads full anaconda package
curl https://pyenv.run | bash #Loads python environment (non-anaconda alternative)

#NOTE: only one of these should be done for any instance!

module list #Lists loaded modules
```

Tested a quick "Hello, World!" program just to make sure that python runs correctly; it does, at least for now.

3. Small Github Digression

We have added a shared file regarding various commands for Github and Quarto. It can now be found here, as well as in the landing page, currently under the name 'Commands Information'.

4. Persistent Exclusion Process Code

The code models particles moving across various lattice sites, with a 'tumble' probability. This is the probability that the particle will change its direction at every iteration (or time 'tick'). This means that the tumble variable is inversely proportional to the **persistence length** of the environment.

Note that this is a (relatively) simple algorithm: every particle has the same chance to change direction (in a more accurate model they would more likely follow a probability distribution instead).Nonetheless, it's a good start for getting a feel for one of the core topics of this project: the way **persistence length influences collective behaviour**. In this case, the **clustering** phenomena is heavily affected.

Below are some GIF files with varying tumble speeds P_{tumble} applied to the model. Note that the animation runs at 6 frames per second, and computes 50 frames total.

$P_{tumble} = 0.0005$

(Placeholder text: `.gif` can obviously not be rendered in pdf form. Consult website for better layout.)

The persistence length is so small here that almost all particles form into clusters.

$P_{tumble} = 0.001$

(Placeholder text: `.gif` can obviously not be rendered in pdf form. Consult website for better layout.)

The clustering is still noticeable here, but there are many more free particles roaming already (with a persistence length twice as big as the previous one).

$P_{tumble} = 0.01$

(Placeholder text: `.gif` can obviously not be rendered in pdf form. Consult website for better layout.)

Clusters are far less frequent, but still noticeable. This is a tenfold increase in persistence length compared to the previous case.

$P_{tumble} = 0.1$

(Placeholder text: `.gif` can obviously not be rendered in pdf form. Consult website for better layout.)

Barely any clusters form here, and when they do, it's only for a few frames.

$P_{tumble} = 0.33$

(Placeholder text: `.gif` can obviously not be rendered in pdf form. Consult website for better layout.)

There are no noticeable clusters.

Note that if $P_{tumble} = 1$, there is no autonomous activity within the system. In other words, particles exercise simple inertial movement in frictionless environment.

5. Code Examination

We have started looking at the code. It's hard to standardise everything we have talked about into notes format; most of it is commented on the repository (formalised as in-line comments and docstrings). As the status of this repository is unknown at this time (whether it will be included in the final project submissions, that is; it is, after all, an example supplied by the supervisor, which we are analysing to get a better understaning of the topic), I will summarise to the best of my ability. (Note that this will be vastly incomplete, as we have only discussed a few scripts, and only partially!).

`lattice.py`

- establishes a lattice class
  - call each instance that fits within a class an **object**
- establishes various attributes for the class
  - `Nsites`: the total amount of lattice sites within system
  - 'Nparticles': the total amount of particles moving around the system (with the stipulation that only one particle can fit in a lattice site at a given time)
  - `connectivity=4`: we are unsure what this is as of yet; operating under the assumption that this is the amount of neighbours a lattice site is connected to (and thus the amount of places a particle occupying a lattice site can jump to, depending on its orientation)
- `set_square_connectivity` subfunction
  - requires values of `Nx` and `Ny` (number of sites along x axis and number of sites along y axis) to rectangularly fit the number of total sites (`Nx*Ny==Nsites`)
  - creates a 'neighbor table' using 2D numpy arrays, with each row being an individual lattice site and each column being an individual neighbour (along collectivity)
  - this is flattened into a one-dimensional array, ran through `construct_square_neighbor_table` (discussed below), then the result is obtained in both a one-dimensional (flattened) version and in a restored two-dimensional version

`construct_square_neighbor_table`

- takes the neighbor table from `lattice.py` as outlined above
- the fastest way to store data in C is through a one dimensional array called with pointers, as it ensures that all the data is contiguously stored in the same memory address area
  - **the indexing is of note (and may be useful later)**

- i,j lattice sites
    - i goes through Nx positions (rows)
    - j goes through Ny positions (columns)
- index: `i * Ny + j`

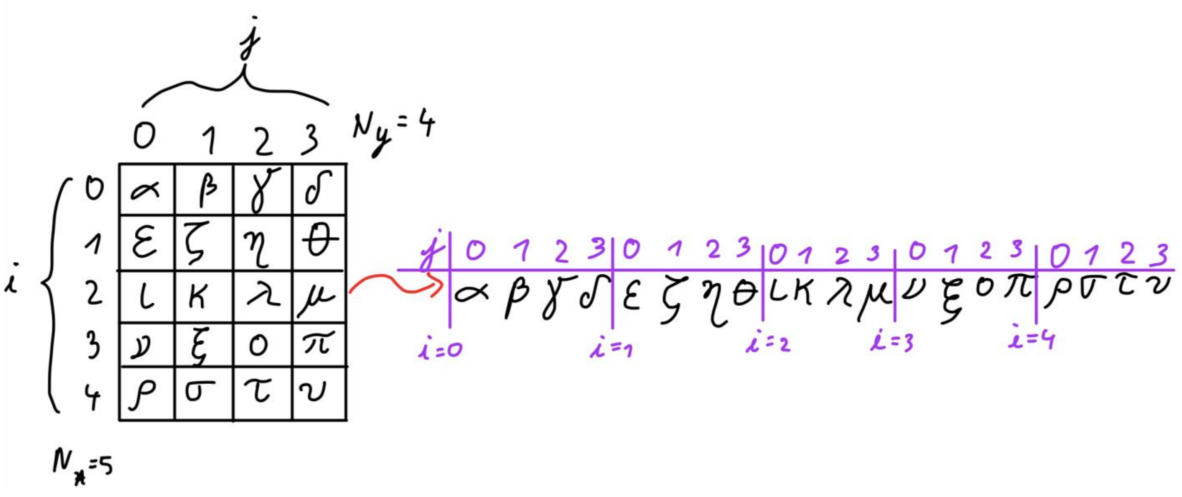
j
0 1 2 3
$N_y = 4$
i
0 1 2 3 4
α β γ δ
ε ζ η θ
ι κ λ μ
ν ξ o π
ρ σ τ υ
$N_x=5$
j
0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3
$\alpha\ \beta\ \gamma\ \delta\ \varepsilon\ \zeta\ \eta\ \theta\ \iota\ \kappa\ \lambda\ \mu\ \nu\ \xi\ o\ \pi\ \rho\ \sigma\ \tau\ \upsilon$
i=0
i=1
i=2
i=3
i=4

Figure 1: 2D to 1D array sketch example