Final Exam Report

Ali Abolhassanzadeh Mahani
97110863

1 Problem 1

1.1 Introduction

I made the `BlumeCapel` class with init function that takes values of J, D, and length L, and creates a system with random spins with side length L. In each `metropolis()` time step, the system gives all of the cells (ammortized) a chance to change spin. since we have 2 options to change to, we take one randomly and then use the metropolis `decision()` method to see if we will change the spin. For the system to reach equilibrium, I devised a method called `eqaulize()` that evolves the system 20 time steps in each iteration, and saves the energy of the system. Then, it finds the auto-correlation of the energies and if the it's lower than a threshold, the system is equalized.

I made the `get_data()` method, to take data from the equalized system in each `step` time steps. I also made the `simulate()` method, that simulated the system to various values of J and D and returns the collected data. I then save the data to a `.npy` file in the `data/` subdirectory and use the `analysis.py` file to draw the plats.

1.2 Part A

I plotted the change in each variable, over various D, for different values of J. (Fig 1, 2, 3)

1.3 Part B

for $L = \{20, 40\}$, the runtime was too high and couldn't complete the data collection in time. So I could only plot for $L = 20$ in Fig 4

2 Problem 2

2.1 Theory

The leonard-Jones potential is as follows:

$$U_{ij} = 4\epsilon\left(\left(\frac{\sigma}{r_{ij}}\right)^{12} - \left(\frac{\sigma}{r_{ij}}\right)^{6}\right),\ r_{ij} \equiv \|\vec{r}_i - \vec{r}_j\| \tag{1}$$

The reduced units is as follows:

$$\begin{aligned} E' &= \frac{E}{\epsilon}, \\ L' &= \frac{L}{\sigma}, \text{L = length}, \\ m' &= \frac{m}{m_{arg}}, \\ T' &= k_B T \end{aligned} \tag{2}$$

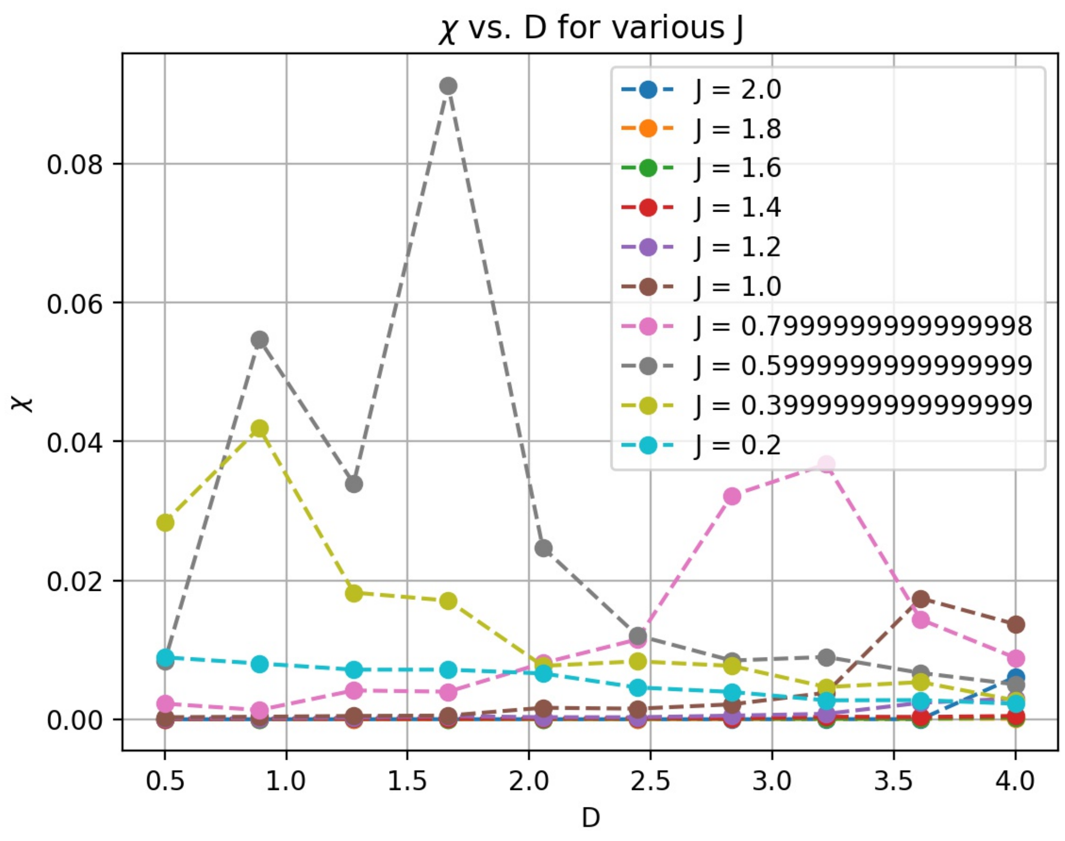

Figure 1: We can see that the critical value of D changes with J

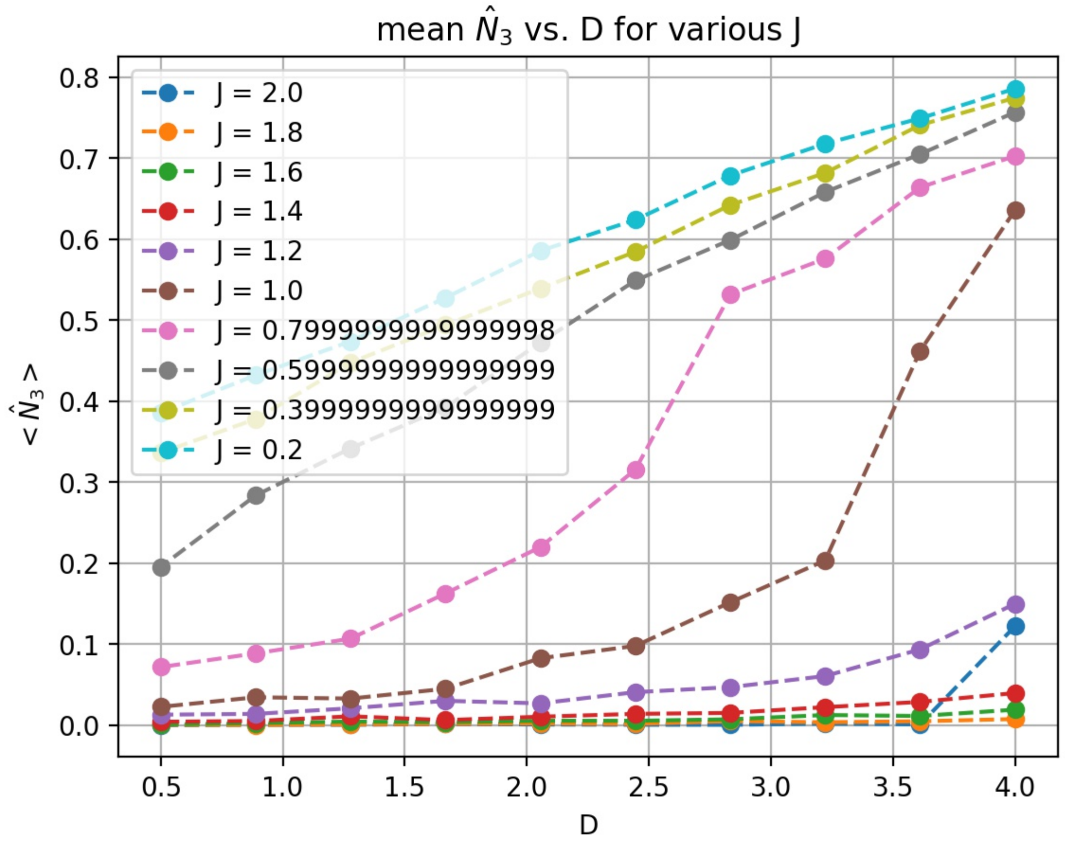

Figure 2: We can see that the critical value of D changes with J

The values for the constanst is as follows:

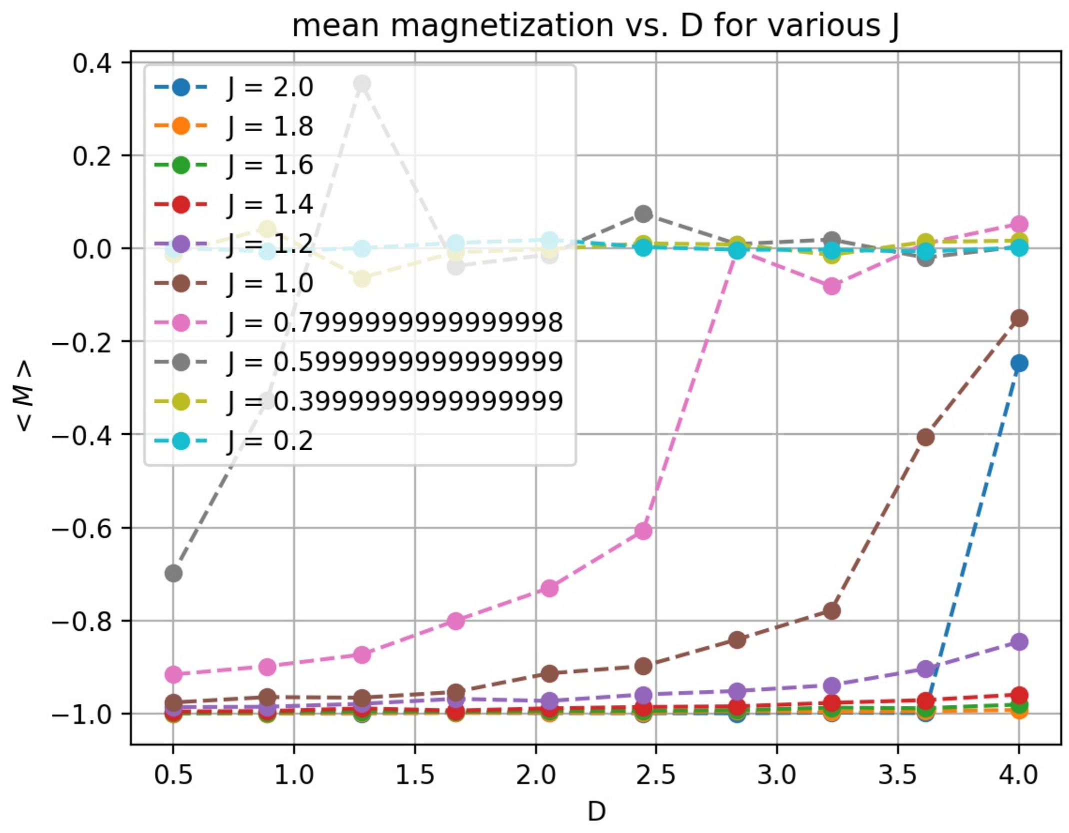

Figure 3: We can see that the critical value of D changes with J

$$
\begin{aligned}
\epsilon &\simeq 15.03 \times 10^{-22} J, \\
\sigma &\simeq 3.419 \times 10^{-10} m, \\
m_{arg} &\simeq 6.63 \times 10^{-26} kg \\
k_B &\simeq 1.38 \times 10^{-23} \frac{J}{K}
\end{aligned}
\tag{3}
$$

Now I show the relation for density, perssure, temperature, and time in Table 1

$f = \alpha f'$		
α	param	value
density ρ	$\frac{m_{arg}}{\sigma^3}$	4142.97
time τ	$\sqrt{\frac{m_{arg}\sigma^2}{\epsilon}}$	1.63×10^{-12}
temp. T	$\frac{\epsilon}{k_B}$	117.8
pressure P	$\frac{\epsilon}{\sigma^3}$	9.94×10^7

Table 1: The coefficients that relate the SI to the reduced unit system.

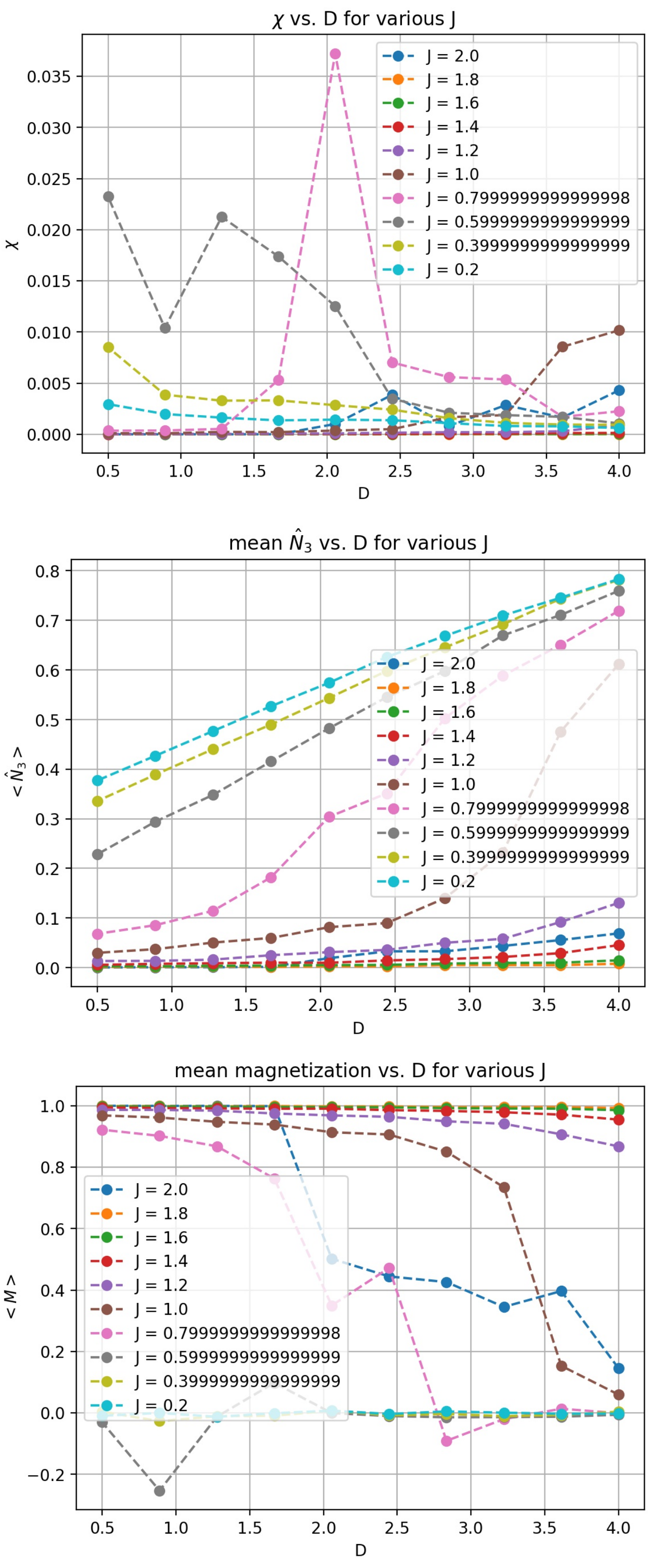

Figure 4: Plots for problem 1 with length 20

2.2 Psudocode

Choosing $L = 7$ in reduced units gives $\rho = 0.36$ for 128 particles. And having max value of velocities to me 1.5 gives a temperature of about $80K$.

```
 def __init__():
x, y, z ← positions // uniformly distributed inside the box
self.L = 7 // size of the box L * L * L
self.volume = self.L ** 3
self.accel ← accel()
v_x, v_y, v_z = // random velocity with max velocity around 1.5
stabilize()

def accel(x, y, z):
// return acceleration for each direction considering periodic boundaries

def stabilize():
// take mean of the velocity of the particles as center of mass velocity
// substract all the velocities by that value

def timestep():
// verlet method to evolve
h = 0.001
x, y, z += v_x, v_y, v_z * h + 0.5 * self.accel * h ** 2 // update position
v_x, v_y, v_z += self.accel * h * 0.5 // update speed partially
self.accel = accel() // update self.accel
v_x, v_y, v_z += self.accel * 0.5 * h //completely update veolcities
// set periodic boundary conditions that if a particle moves out, comes in from the
other side

def temp():
// take sum of square of velocities and divide by 3 * 127

def pressure():
return (128 * temp() - sum(F_ij dot r_ij) * N * 3 / 2) / V // where V is the volume
of the box
```

There's also a fully functional code for this in `p2/` directory. The only necessity is to specify the initial velocity and the positions. One can just change the value of `self.dim` to 3 and the code is ready to simulate the 3-D version of Argon atoms.

3 Problem 3

Group members: Ali Abolhassanzadeh Mahani, Sina Moammar, Shabnaz Ghaffari

In order to simulate this problem, we tried modifying Sina's MD simulation code since it was faster than the others. In this code, we chnaged the initial position of the particles, the side length, and other parameters. We added matrices for neghibors and changed the potential to be of the form of the problem at hand.

This `LangevinHarmonic` class has several features. the `render()` method, evolves the system and creates several files with the same name and different extensions. The `.info` file, stores general data about the parameters of the system in *JSON* format. The `.data` file, stores the values for temperature, pressure, mean neighbors, etc., and the `.traj` file, stores the velocity and position of the particles in every recorded time frame. These values are written in *binary* to take less space and be more efficient.

Then there's the `DataAnalysis` class, that reads the data from the generated files, and compiles the data. Then, it returns a `data` object that stores all the data from the files and contains methods to `animate` the system, return `mean_neighbors`, etc. .

3.1 Part A

This section of the problem is pretty much straightforward. I used the `eval()` built-in function in python to calculate the values for the side of the system for various values of area percentage. Then, I stored them in a dictionary and printed them into the console.

The results are below:

```
{0.4: 26.95353967742715, 0.7: 20.374960839824098, 0.8: 19.059030682889404,
0.9: 17.9690264516181, 1.0: 17.04691527687798}
```

3.2 Part B

For this section, we took the values derived in the previous section, and simulated the problem for 6000 time steps to reach equilibrium. Then, we plotted the mean neighbors over time in a graph. (Fig 5)

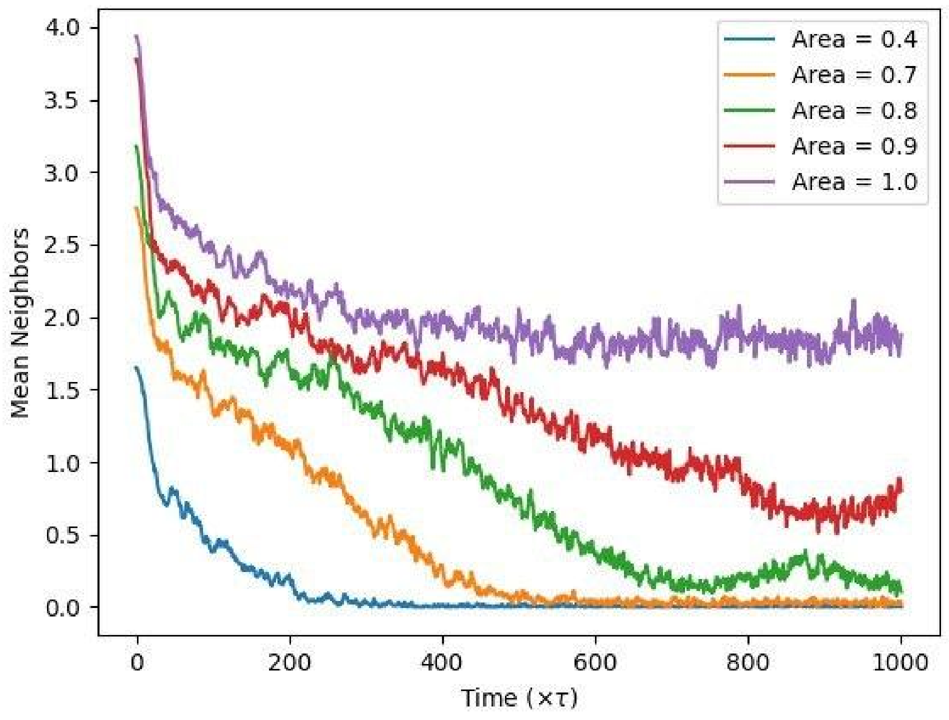

Figure 5: We can see that all the systems reach equilibrium at about 4 time units.

3.3 Part C

In this section, in order to make more accurate conclusions, we made ensembles of size 10 for each value of area percentage and took the mean and stdev (as error). The plot is available in Fig 6

3.4 Part D

This phase transition is know as a *first-order phase transition driven by fluctuations*. With increasing temperature, the critical value of area percentage decreases.

4 Problem 4

I implemented the *euler* and *backward euler* in the file `eulers.py`. Then, in `main.py`, I import them and use them to calculate the results. Both methods store and return an array for `time`, `x`, `x_dot`,

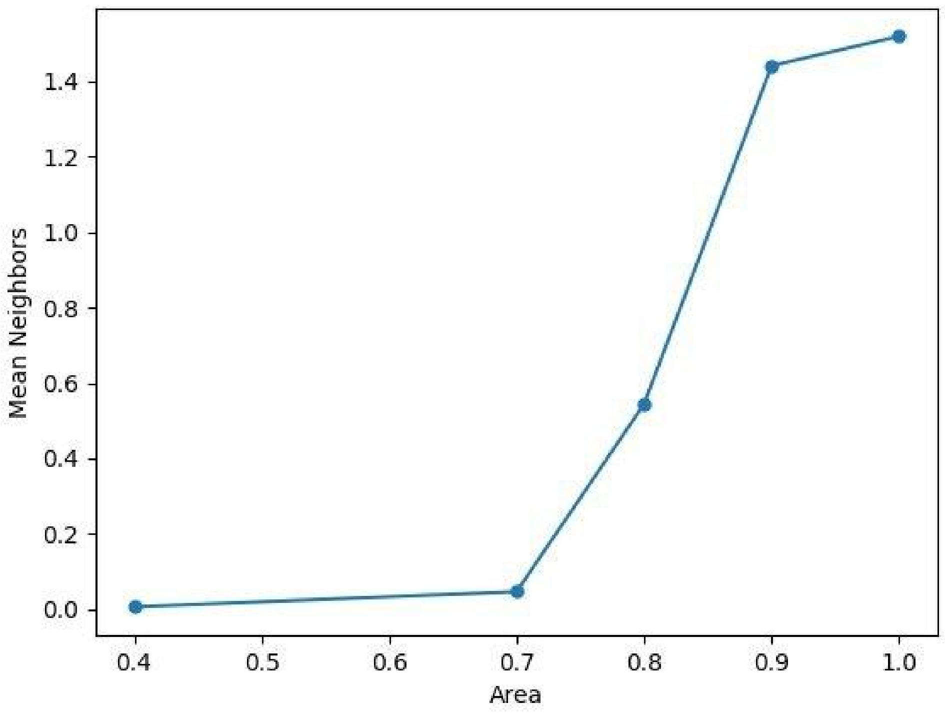

Figure 6: Phase transition accures at about 1.0

and `count` which is the l ength of the previous arrays.

4.1 Part A

I calculated the curves using *euler* method for the given time steps. The results are in Fig 7 blow:

4.2 Part B

Did the same as part A using the *backward euler* method. Results in Fig 8

4.3 Part C

The analytic result of the integral is:

$$y = \frac{1}{20}\left(e^{-t} - e^{-21t}\right) \tag{4}$$

Since the algorithms take the count and the ending time for them isn't exactly 4, in order to find the right value for error, I calculate the error by taking the real value for the last time of each algorithm, and substract *that* from the resulted value of that algorithm.

You can see the values of runtime and error for each value of h and for euler and backward euler in Table 2

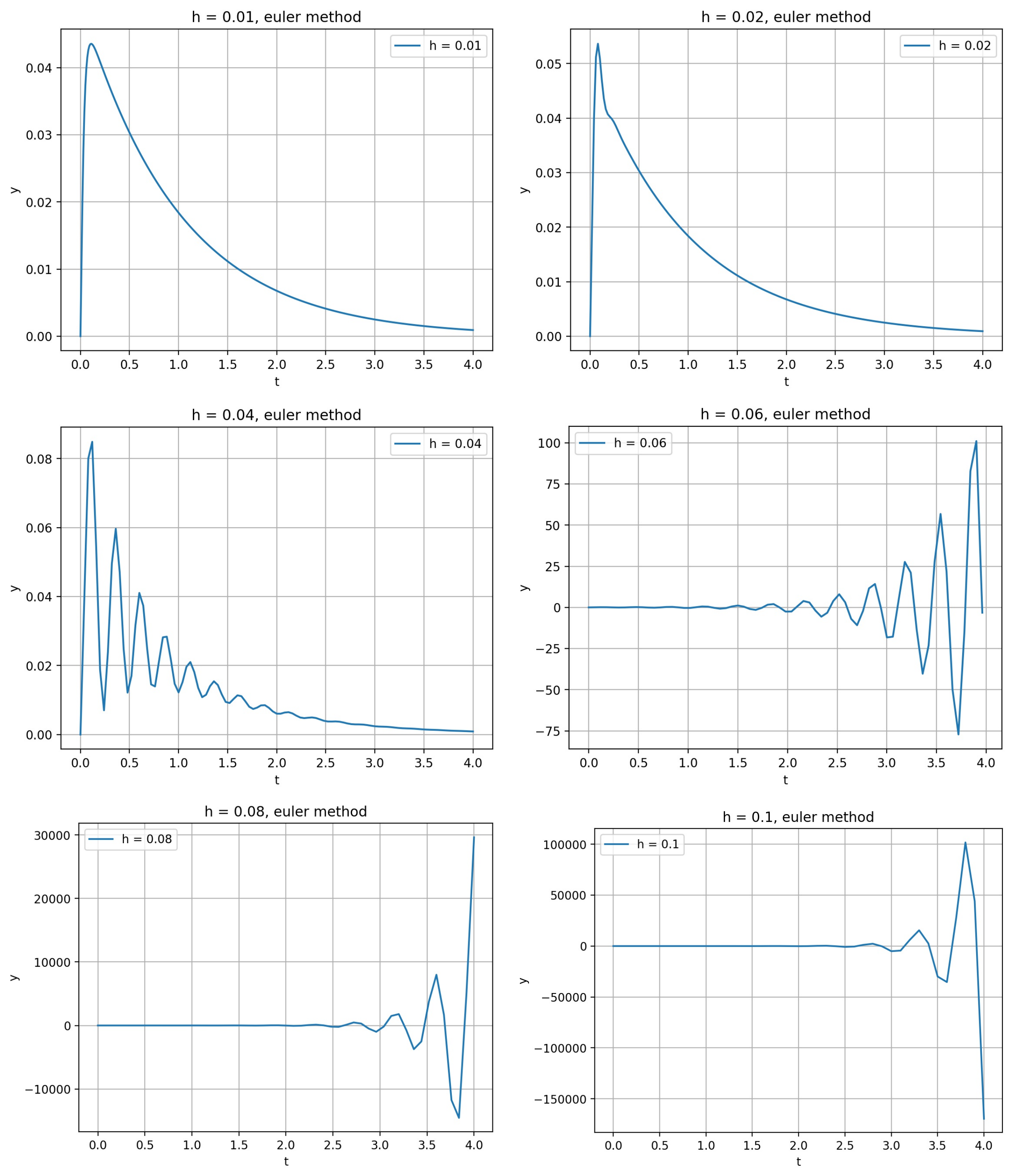

Figure 7: The integral curve for various values of h using the euler method. It can be seen that from a value of h, the results become *unstable*

	euler		backward euler	
h	runtime	error	runtime	error
0.01	0.001356363	-6.81×10^{-7}	0.00118112	-2.28×10^{-7}
0.02	0.000642061	-1.35×10^{-6}	0.00055789	-4.54×10^{-7}
0.04	0.000321388	-9.26×10^{-6}	0.00028133	-9.028×10^{-7}
0.06	0.000210523	-3.227	0.00018715	-1.399×10^{-6}
0.08	0.000161647	29622	0.00014257	-1.78×10^{-6}
0.1	0.000129699	-169639	0.00011444	-2.2575

Table 2: data for euler and backward euler. As one can see, the backward euler method gives much more accurate results while the runtime is almost the same.

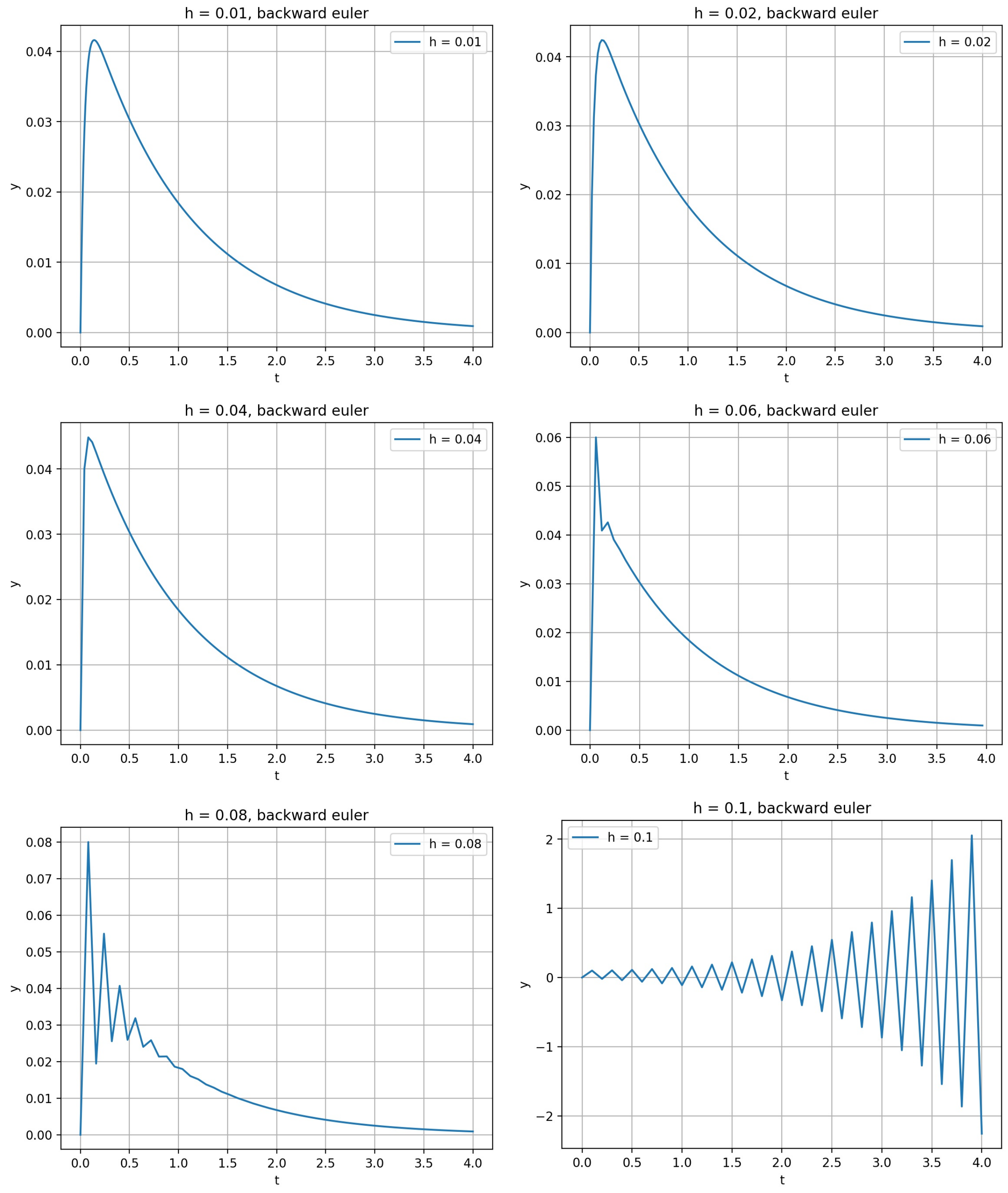

Figure 8: The integral curve for various values of h using the backward euler method. It can be seen that from a value of h, the results become *unstable*

4.4 Part D

I used the the following algorithm:

$$\begin{aligned} k &= \dot{y}(y_n, t)h \\ \dot{y}_{n+1} &= \dot{y}(y_n + k, t + \frac{h}{2}) \\ y_{n+1} &= y_n + \frac{\dot{y}_{n+1} + \dot{y}_n}{2}h \\ \dot{y}_{n+1} &= \dot{y}(y_{n+1}, t + h) \end{aligned} \tag{5}$$

The values for runtime and error are in Table 3

h	runtime	error
0.01	0.002925872	2.579×10^{-6}
0.02	0.001465082	5.903×10^{-6}
0.04	0.000742435	1.653×10^{-5}
0.06	0.000485897	4.276×10^{-5}
0.08	0.000373840	0.000156
0.1	0.000296592	-2.01672

Table 3: data for `my_method`. As one can see, my method gives accurate and more stable results that the euler method, while the runtime is about 10 times that of euler.

5 Problem 5

I made a `System` class that takes the values for `r`, and makes the system. the `system_play()` method randomly matches the players into pairs. Then, these pairs play with each other and if their strategies differ, they both change their strategy for the next step. The `next_step()` method plays the system for one step and updates the value of Q. Note that in `__init__()`, `r` players have fixed strategies.

5.1 Part A

Here I plot the values for Q and the strategy of player 50 in the system. (Fig 9)

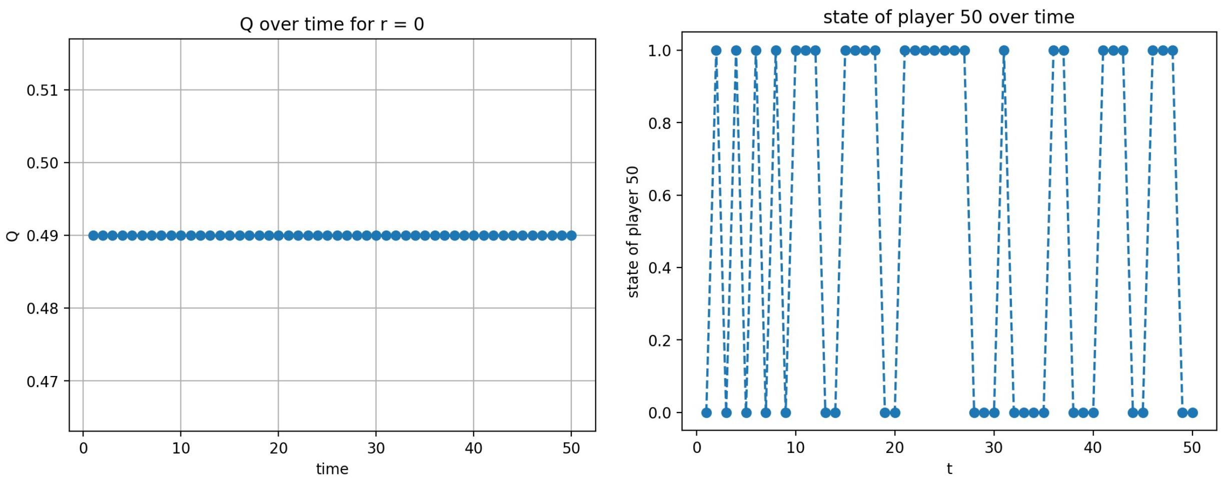

Figure 9: As one can see the system is dynamic and the states changes but the value for Q is fixed. This is due to the dynamic of the players, since if their strategies differ, they *both* change strategy results in no change in Q

5.2 Part B

Here I plot the values of Q for the systems with `r` fixed players, varied from 0 to 10 for 50 time steps. (Fig 10)

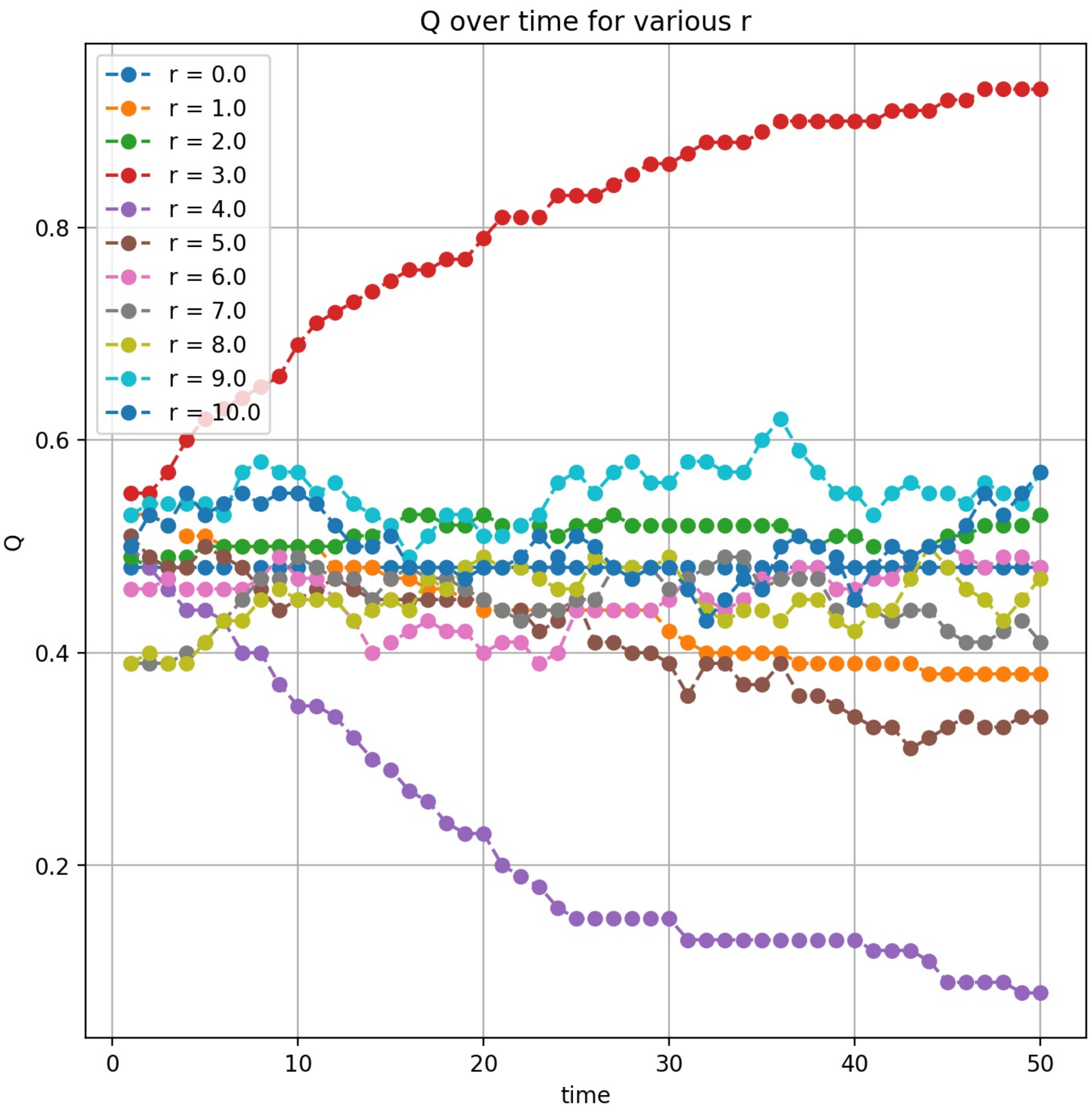

Figure 10: Now since some of the players are fixated on their strategies, Q varies with time. We see that for intermediate values of r, the system is divergent.