PHYS 310 Homework 2

Khalil El Achi

Contents

1 Network Laplacian

1.1

The Lagrangian associated with $N = 10$ coupled harmonic oscillators is written as follows. Assuming the oscillators have one degree of freedom along the x - direction:

$$\begin{aligned} \mathcal{L} &= KE - PE \\ &= \frac{1}{2}\sum_{i=1}^{N} m_i \dot{x}_i^2 - \frac{1}{2}\sum_{i=1}^{N-1} k_i (x_{i+1} - x_i)^2 \end{aligned}$$

To get the equations of motions, apply the Euler-Lagrangian equation along the x degree of freedom where:

$$\begin{aligned} \frac{d}{dt}\left(\frac{\partial \mathcal{L}}{\partial \dot{x}_i}\right) - \frac{\partial L}{\partial x_i} &= 0 \\ \sum_{i=1}^{N} m_i \ddot{x}_i - \sum_{i=1}^{N} k_i (x_{i+1} - x_i) &= 0 \end{aligned}$$

2 Tight-binding

To find the thermodynamical properties of a network depending on which states it is in, the following calculations are carried.
The hamiltonian of the system is defined by the adjacency matrix

$$H = \alpha I + \beta A$$

The associated energies are computed through the eigenvalues of the matrix A

$$E_i = \alpha + \beta \lambda_i$$

Subseuqently the problem revolves around computing the following properties of the system depending on the arrangment of the nodes of the network:

- Partition Function: $Z = tr(e^{\beta H}) = \sum_j e^{\beta * E_j}$
- Entropy $S = -\sum_j p_j \log(p_j)$
- Helmholtz Free Energy $F = -\frac{\log Z}{\beta}$
- Gilbert Free Energy $G = F + \frac{\langle E \rangle}{\beta}$

The properties will be computed for the small world, random, and regular models. These will simply change the adjacency matrix generated. The Network package will be used in python to generate the adjacency matrix. Whilst taking the approximation that $\alpha = 0$ and $\beta = \frac{1}{K_B T} = -1$.

The general code that will used to generate the graphs of the partition function, Helmholtz free energy, entropy, and Gilbert free energy as a function of the probabilities is the following:

```
import numpy as np
import matplotlib.pyplot as plt
import networkx as nx
import scipy.sparse
# Define the temperature and other constants
k_B = 1.0 # Boltzmann constant
T = 1.0 # Temperature

# Function to calculate the partition function
def calculate_partition_function(beta, energy_levels):
return np.sum(np.exp(-beta * energy_levels))

# Function to calculate the entropy
def calculate_entropy(probabilities):
return -np.sum(probabilities * np.log(probabilities))

# Function to calculate the Helmholtz free energy
def calculate_helmholtz_free_energy(beta, partition_function):
return -1 / beta * np.log(partition_function)

# Function to calculate the Gibbs free energy
def calculate_gibbs_free_energy(helmholtz_free_energy, average_energy):
return helmholtz_free_energy + 1 / beta * average_energy


for i, p in enumerate(p_values):

       adjacency_matrix = ## The code for the adjancey matrix will be given
           ↪ below

       eigenvalues = np.linalg.eigvals(beta * adjacency_matrix)

       probabilities = np.exp(-beta * (eigenvalues - alpha))

       partition_function= calculate_partition_function(beta, eigenvalues)
       partition_function_values[i] = partition_function
       entropy_values[i] = calculate_entropy(probabilities)
       helmholtz_free_energy_values[i] = calculate_helmholtz_free_energy(beta,
           ↪ partition_function)

       average_energy = np.mean(eigenvalues)

       gibbs_free_energy_values[i] = calculate_gibbs_free_energy(
           ↪ helmholtz_free_energy_values[i], average_energy)
```

```
plt.figure(figsize=(12, 8))
plt.grid()
plt.subplot(2, 2, 1)
plt.plot(p_values, partition_function_values)
plt.title('Partition Function Z')

plt.subplot(2, 2, 2)
plt.plot(p_values, entropy_values)
plt.title('Entropy S')

plt.subplot(2, 2, 3)
plt.plot(p_values, helmholtz_free_energy_values)
plt.title('Helmholtz Free Energy H')

plt.subplot(2, 2, 4)
plt.plot(p_values, gibbs_free_energy_values)
plt.title('Gibbs Free Energy G')

plt.tight_layout()
plt.show()
```

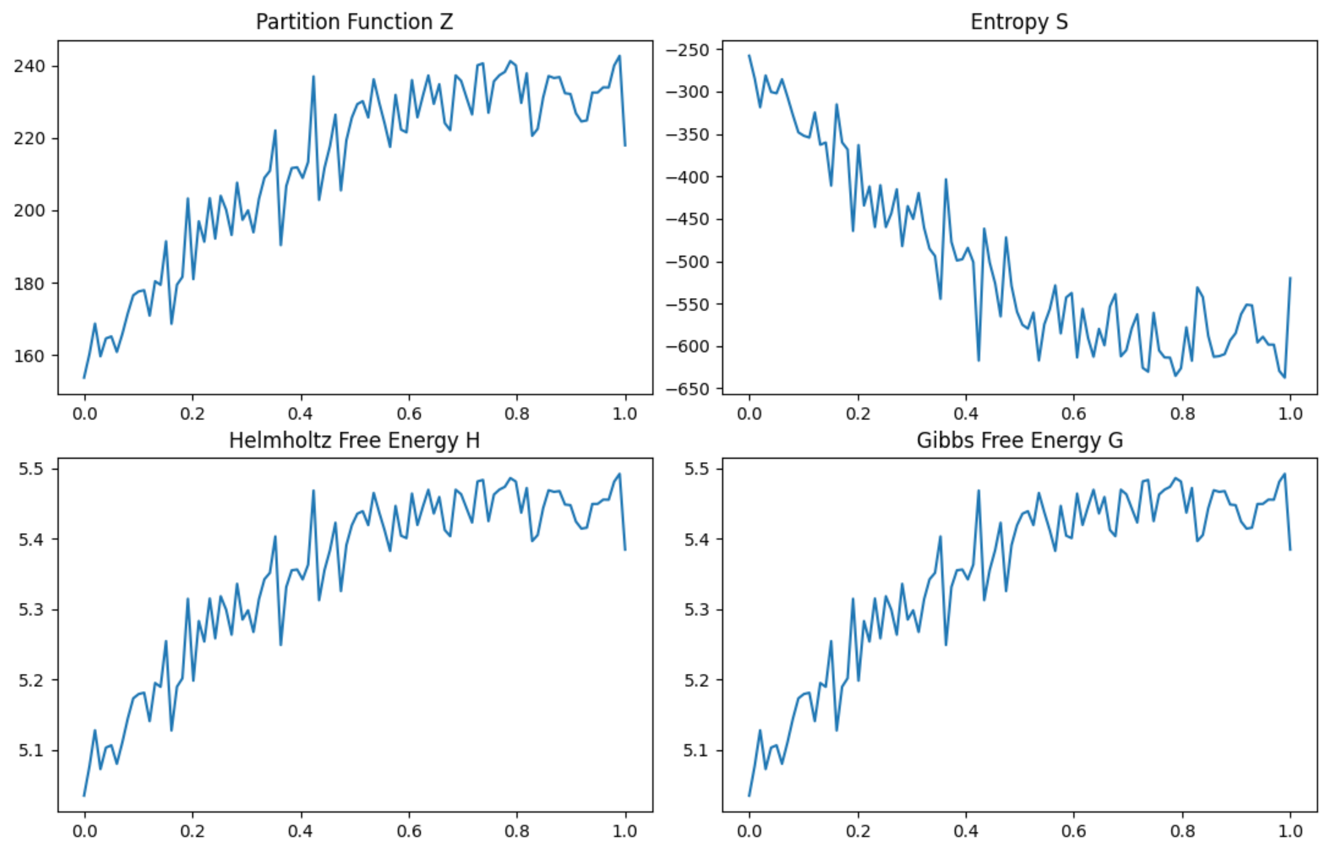
What will change throughout the code is how the adjacency matrix is generated for the different models.

2.1 Small-World Network or Watts-Strogatz Model

The code that will generate the adjacency matrix A for the small world model:

```
def generate_small_world(n, k, p):
        #Where n is the number of nodes and k is the number of nearest neighbors,
           ↪  and p is the probabilities which will be iterated over
        G = nx.watts_strogatz_graph(n, k, p)
        A = nx.adjacency_matrix(G).todense()

return A

adjacency_matrix = generate_small_world(n, k, p) # placed in the for loop
```

The outcome of the code:



2.2 Random Network or Erdos–Renyi Model

The code that will generate the adjacency matrix A for the random network:

```
def generate_random_network(n, p):
        #Same parameters are used in the small world
        G = nx.erdos_renyi_graph(n, p)
        A = nx.adjacency_matrix(G).todense()

        return A

adjacency_matrix = generate_random_network(n, p) # placed in the for loop
```

The code generates the following graphs:

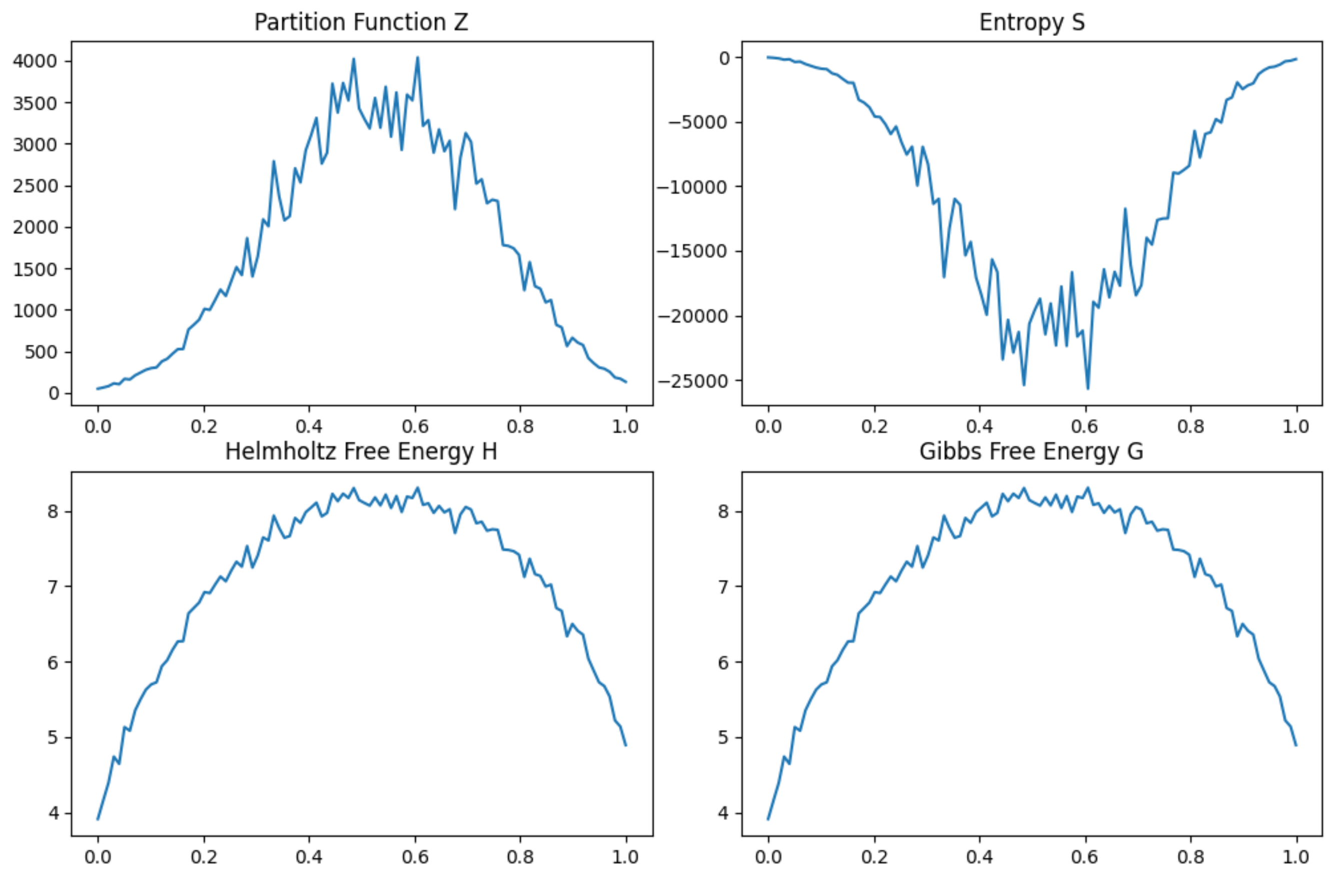


Figure 1: The Thermodynamic Properties of the Random Model

2.3 Regular Network

For Regular networks, the probabilities of neighboring nodes remains the same and the properties remain constant.
The adjacency matrix will be produced as such:

```python
def generate_regular_lattice(rows, cols): # for any number of equal rows and
    ↪ columns
        G = nx.grid_2d_graph(rows, cols)
        A = nx.adjacency_matrix(G).todense()

        return A
```

The code outputs a constant measure of the properties that remains constant throughout the network's p.

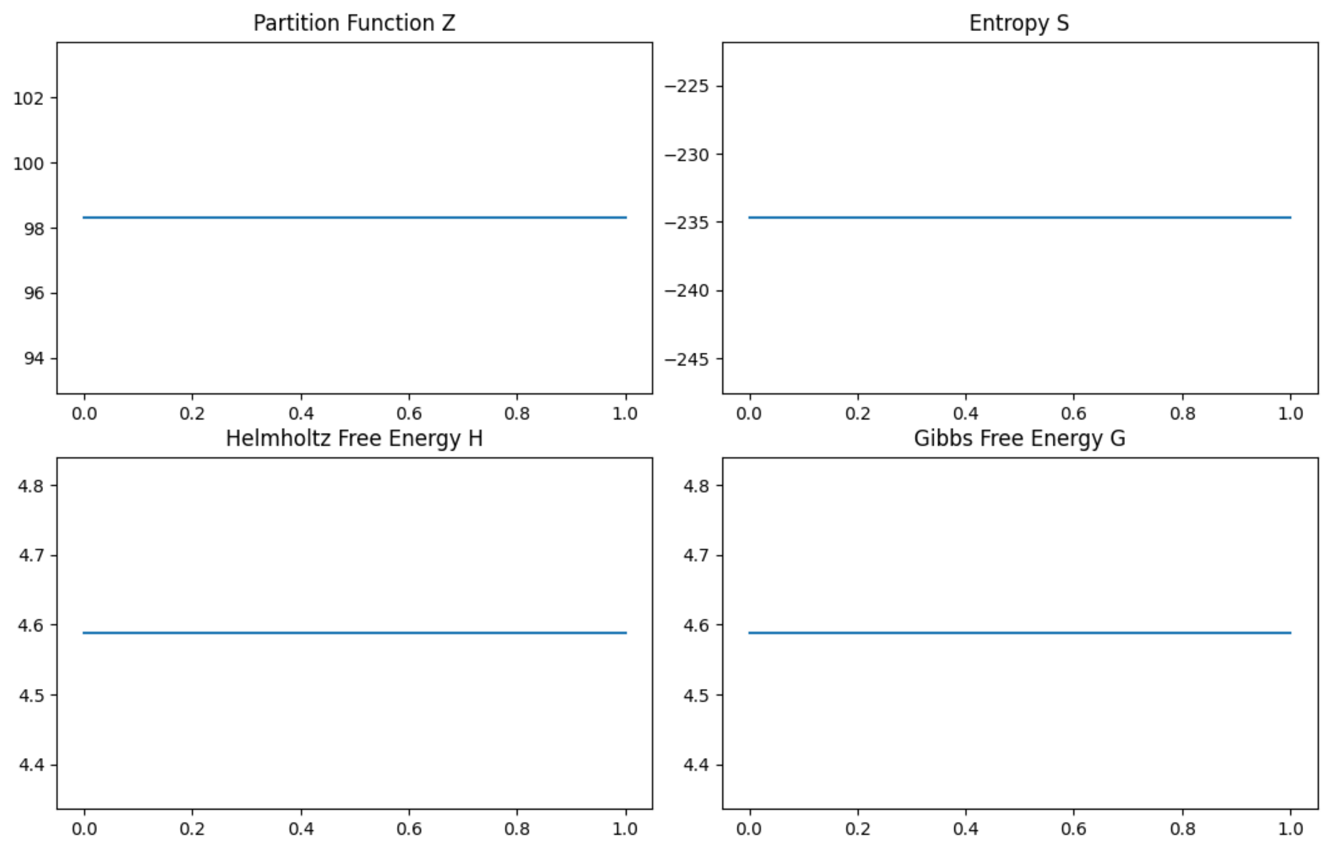


Figure 2: The Thermodynamic Properties of the Regular Model

3 KPZ Universality Class and the Wigner Semi Circle

4 Further Notes

Explored thoroughly !

5 Dynamics

5.1

The Lorenz system of the following function:

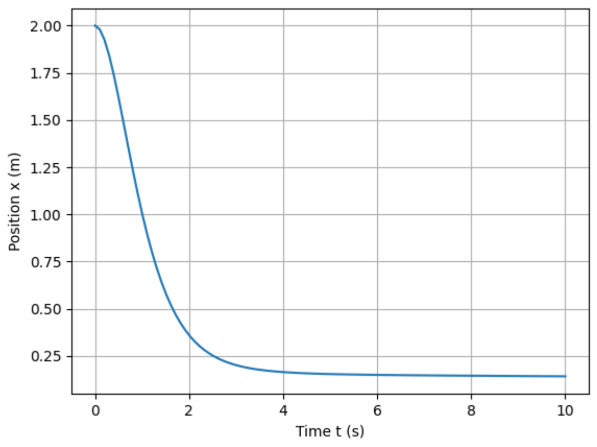
$$\begin{aligned}\dot{x} &= \sigma(y - x)\\ \dot{y} &= rx - y - xz\\ \dot{z} &= xy - bz\end{aligned}$$

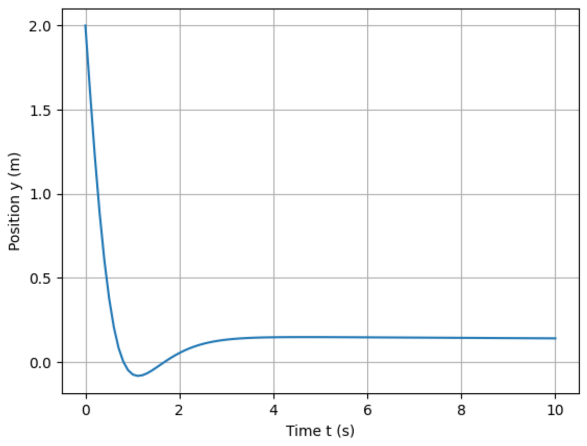
The following code will be used to construct an RK4 approximation to plot x, y, and z in function of time. To assist the code the following parameters were taken: $\sigma,\ b,\ r = 1$ and the initial conditions of $x, y, z = 2$.

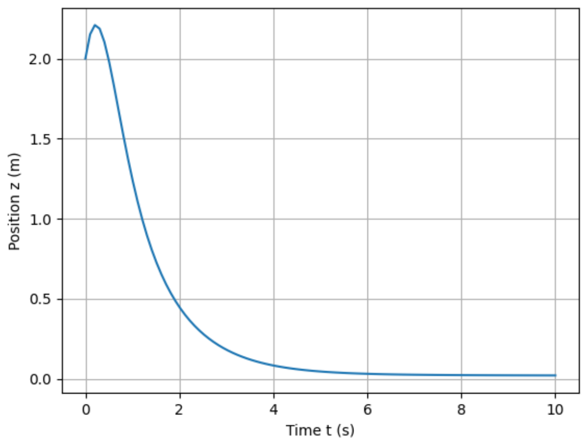
```
import numpy as np
import scipy as sp
import math
import matplotlib.pyplot as plt
import sympy as sy

# RK 4 Method
# x' = sigma( y -x )
# y' = rx - y - xz
# z' = xy - bz
sigma = 1
r = 1
b =1
def X(x,y,z):
        return sigma*(y - x)
def Y(x,y,z):
        return r*x - y - x*z
def Z(x,y,z):
        return x*y - b*z

t=np.linspace(0,10,100)
xpoints=[]
ypoints=[]
zpoints=[]
h=t[1]-t[0]
x=2
y=2
z=2
for i in range(0,len(t)):
        xpoints.append(x)
        ypoints.append(y)
        zpoints.append(z)

        m1 = h*X(x,y,z)
        k1 = h*Y(x,y,z)
        l1 = h*Z(x,y,z)

        m2 = h* X(x+m1/2,y+k1/2,z+l1/2)
        k2 = h* Y(x+m1/2,y+k1/2,z+l1/2)
        l2 = h* Z(x+m1/2,y+k1/2,z+l1/2)

        m3 = h* X(x+m2/2,y+k2/2,z+l2/2)
        k3 = h* Y(x+m2/2,y+k2/2,z+l2/2)
        l3 = h* Z(x+m2/2,y+k2/2,z+l2/2)
```

```
            m4 = h* X(x+m3,y+k3,z+l3)
            k4 = h* Y(x+m3,y+k3,z+l3)
            l4 = h* Z(x+m3,y+k3,z+l3)

            x += (m1 + 2*m2 + 2*m3 + m4)/6
            y += (k1 + 2*k2 + 2*k3 + k4)/6
            z += (l1 + 2*l2 + 2*l3 + l4)/6

    plt.plot(t,xpoints)
    plt.xlabel('Time t (s)')
    plt.ylabel('Position x (m)')
    plt.grid()
    plt.show()
    plt.figure
    plt.plot(t,ypoints)
    plt.xlabel('Time t (s)')
    plt.ylabel('Position y (m)')
    plt.grid()
    plt.show()
    plt.figure
    plt.plot(t,zpoints)
    plt.xlabel('Time t (s)')
    plt.ylabel('Position z (m)')
    plt.grid()
    plt.show()
```

The code outputs the following graphs that shows the evolution of x, y, and z in time space:


Position x (m)
Time t (s)
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0
2
4
6
8
10


Position y (m)
Time t (s)
2.0
1.5
1.0
0.5
0.0
0
2
4
6
8
10


Position z (m)
Time t (s)
2.0
1.5
1.0
0.5
0.0
0
2
4
6
8
10

6 Logistic Map

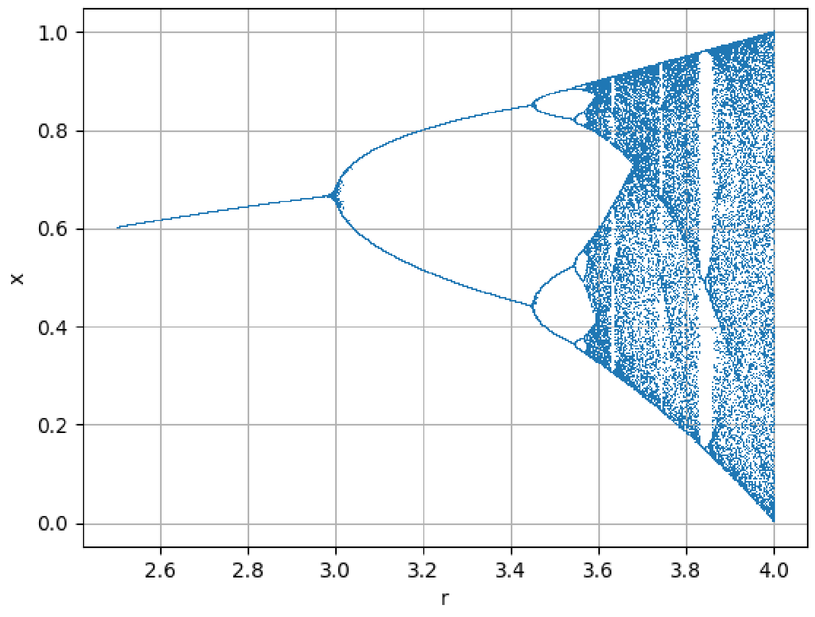
6.1

The Logistic Map is represented by the following iterative function:

$$x_{n+1} = rx(1 - x_n)$$

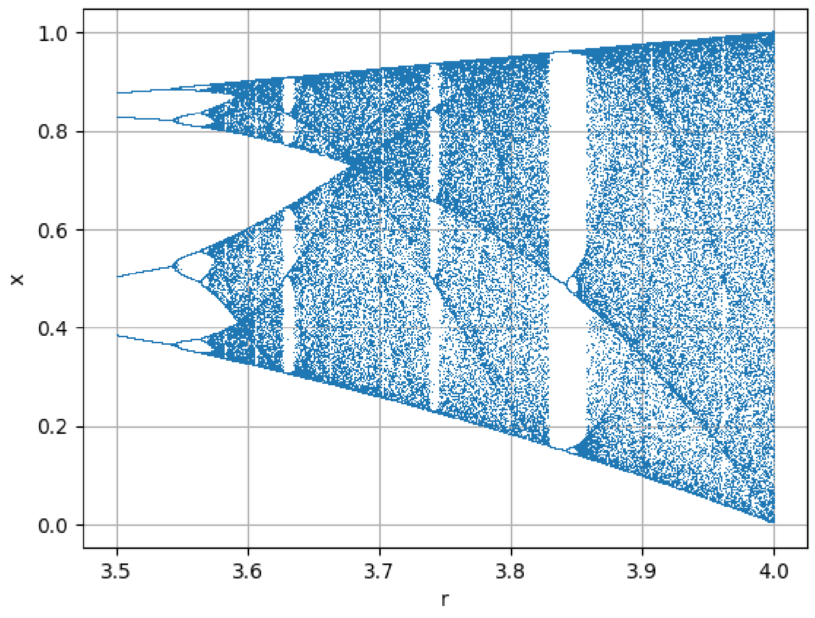
To find the chaotic regime, arbitrarily iterate over values of r from 2.5 to 4, the logistic map will be executed as follows:

```
import numpy as np
import scipy as sp
import math
import matplotlib.pyplot as plt
import sympy as sy

xpoints = [] # Storing the x points
R = np.linspace(3.5,4,100000)

for r in R:
        x = np.random.random()
        for j in range(100):
                x = r*x*(1-x)

        for j in range(100):
                x = r*x*(1-x)
        xpoints.append(x)

plt.plot(R,xpoints, ls='',marker = ',')
plt.grid()
plt.xlabel("r")
plt.ylabel("x")
plt.show()
```



6.2

The obvious chaotic regime is seen to be around the values of r between 3.5 and 4. To zoom in to that regime and to find its constituents, the zoomed in figure is as such:

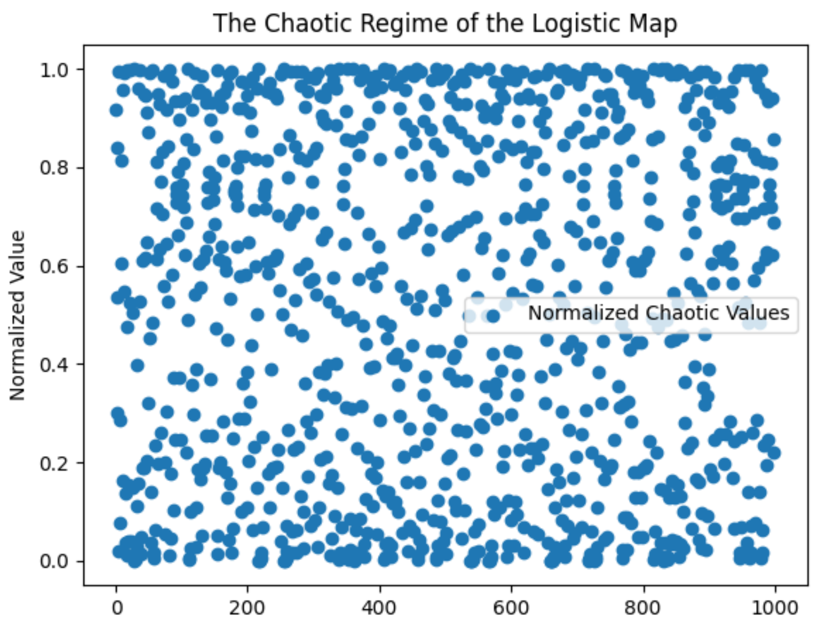


The values of function around r=4 show proper chaos that the values can be used as a random number generator.

To do that, first one needs to normalize the values.
The Chaotic regime is developed by:

```
x_chaotic = []
x=np.random.random()
r=4
for i in range(1000):
x = r * x * (1-x)
x_chaotic.append(x)

x_chaotic_normalized = (x_chaotic - np.min(x_chaotic)) / (np.max(
    ↪ x_chaotic) - np.min(x_chaotic))

plt.scatter(range(1000),x_chaotic_normalized, label='Normalized Chaotic
    ↪ Values')
plt.title('The Chaotic Regime of the Logistic Map')
plt.ylabel('Normalized Value')
plt.legend()
plt.show()
```

The given plot is of the following:



Randomly picking a value from these x values will generate a random number that is normalized between 0 and 1.

6.3

7 What do 1D maps have to do with Science?

7.1

The Rossler system that constitutes the following PDEs:

$$\dot{x} = -y - z$$

$$\dot{y} = x + ay$$

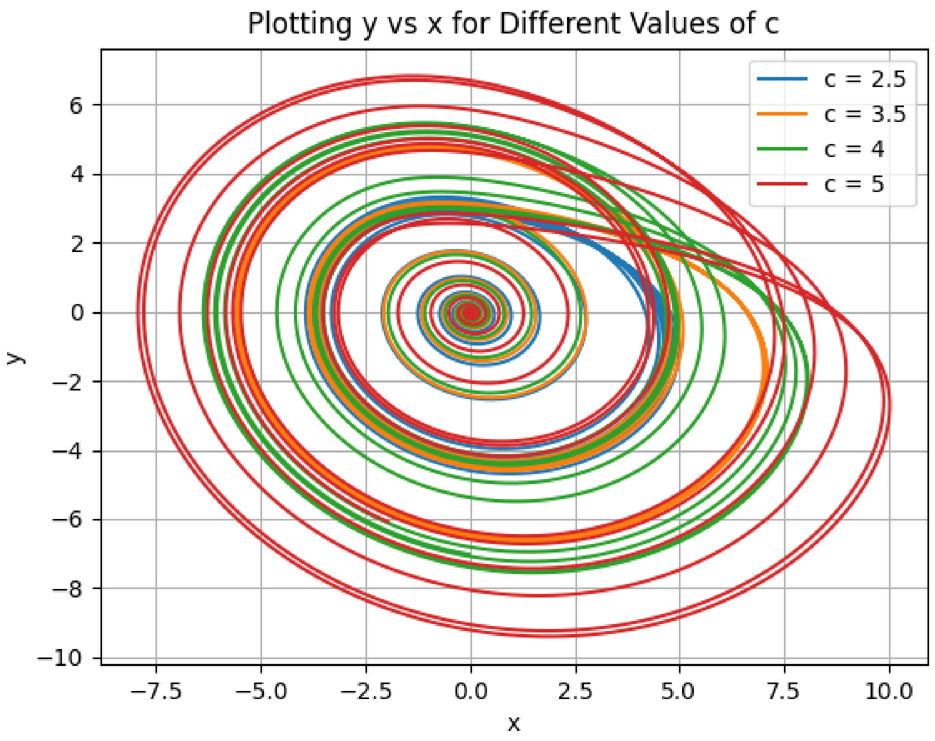
$$\dot{z} = b + z(x - c)$$

The system will be solved using scipy package and $solve_ivp$ function

```
import numpy as np
import math
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp
import sympy as sy

#
# x' = - y - z
# y' = x + ay
# z' = b + z(x-c)
a = b = 0.2
c = [2.5,3.5,4,5]

def rossler(t,xyz,c):
       x,y,z = xyz
       X = - y - z
       Y = x + a*y
       Z = b + z*(x-c)
       return [X,Y,Z]
#Initial conditions
xyz0 = [1,1,1]

def solve_rossler(c):
       # Initial conditions
       xyz0 = [0, 0, 0]
       t=(0,100)
       # Solve the system
       sol = solve_ivp(rossler, t, xyz0, args=(c,), dense_output=True)
       return sol

for c in c:
       sol = solve_rossler(c)
```

```
            t_eval = np.linspace(0, 100, 10000)
            xyz = sol.sol(t_eval)
            plt.plot(xyz[0], xyz[1], label=f'c = {c}')

    plt.title('Plotting y vs x for Different Values of c')
    plt.xlabel('x')
    plt.ylabel('y')
    plt.legend()
    plt.grid()
    plt.show()
```

The code outputs the evolution of y with respect to x for different values of c. The evolution is a bunch of circles evolving around the origin. The greater is c is the bigger and wider the circular path is.



7.2

To find the successive maxima of x for c = 5, the following function is written:

```
    def find_maxima(t, y):
            maxima_indices = (y[:-2] < y[1:-1]) & (y[2:] < y[1:-1])
            return t[1:-1][maxima_indices], y[1:-1][maxima_indices]
```

```
        sol = solve_rossler(5)
        maxima_t, maxima_x = find_maxima(sol.t, sol.y[0])

        plt.figure(figsize=(8, 6))
        plt.plot(maxima_x[:-1], maxima_x[1:], '.', markersize=2)
        plt.title('Successive Maxima Plot for c = 5')
        plt.xlabel('$x_n$')
        plt.ylabel('$x_{n+1}$')
        plt.grid()
        plt.show()
```

The outcome of the code didn't show the logistic map but it showed something similar I found online for the progression of y_{n+1} vs. y_n

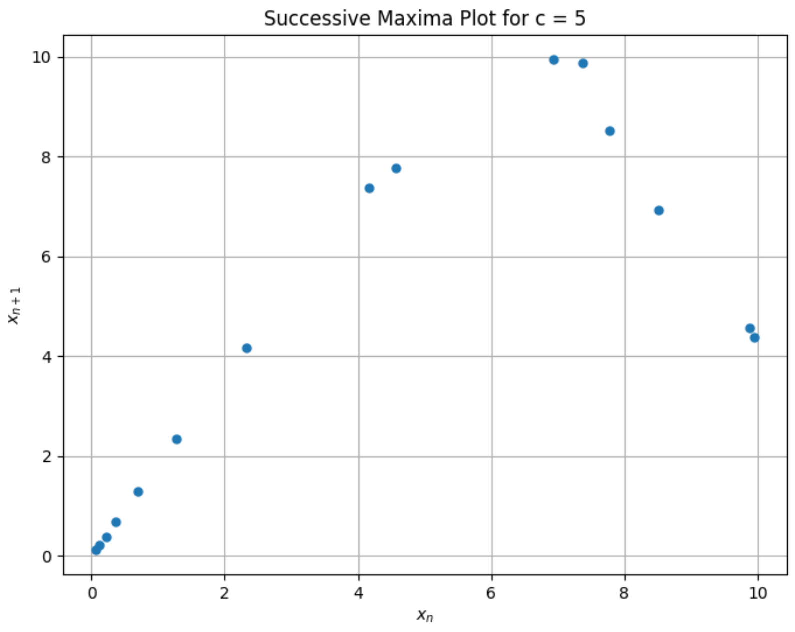


Figure 3: Evolution of x_{n+1} vs. x_n for $c = 5$

7.3

To find the Maxima representing the period variation of the system, as we vary c, the following code is done:

```
def find_maxima(c):
        sol = solve_rossler(c)
        t_eval = np.linspace(0, 100, 10000)
        xyz = sol.sol(t_eval)
        xn = xyz[0]
        maxima_indices = np.where((xn[:-2] < xn[1:-1]) & (xn[1:-1] > xn[2:]))[0]
            ↪ + 1
        maxima_values = xn[maxima_indices]

        return maxima_indices, maxima_values


c_values = np.linspace(2.5, 3.8, 100)

for c in c_values:
        maxima_indices, maxima_values = find_maxima(c)
        plt.scatter([c] * len(maxima_indices), maxima_values, color='b')

plt.title('Maxima versus c in the Rossler System')
plt.xlabel('c')
plt.ylabel('Maxima')
plt.grid()
plt.show()
```

The fig tree diagram showing the Maxima as c is varied is the following:

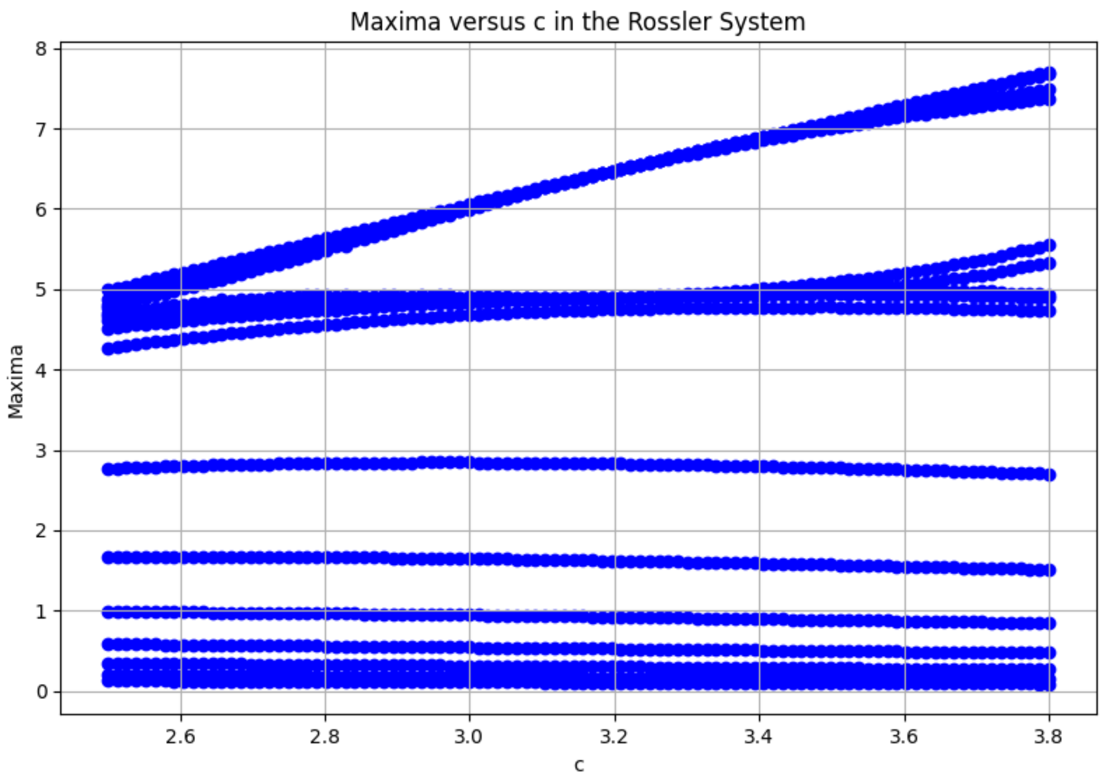


Figure 4: The evolution of the Maxima vs. c

8 SVD and Lagragian coherent Structures

To solve the following system:

$$\psi(x, y, t) = A \sin(\pi f(t)) \sin(\pi y)$$

$$f(x, t) = a(t)x^2 + b(t)x$$

$$a = \epsilon \sin(\omega t)$$

$$b = 1 - 2\epsilon \sin(\omega t)$$

Taking the following parameters:

$$A = 0.25,\ \epsilon = 0.1,\ \omega = \frac{2\pi}{10}$$

The velocity field is defined over:

$$u = -\frac{\partial \psi}{\partial y}$$

$$v = -\frac{\partial \psi}{\partial x}$$

8.1

The code that will compute the evolution of the velocity fields in 200 time frames and in the domain $x \in [0, 2]$ and $y \in [0, 1]$

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.linalg import svd
from scipy.integrate import odeint
from matplotlib.animation import FuncAnimation

A = 0.25
epsilon = 0.1
omega = 2 * np.pi / 10
x_range = np.linspace(0, 2, 100)
y_range = np.linspace(0, 1, 50)
t_frames = 200
dt = 0.1

u_frames = np.zeros((t_frames, len(y_range), len(x_range)))
v_frames = np.zeros((t_frames, len(y_range), len(x_range)))

for t in range(t_frames):
        a = epsilon * np.sin(omega * t * dt)
        b = 1 - 2 * epsilon * np.sin(omega * t * dt)
        f = lambda x: a * x**2 + b * x
```

```
            psi = A * np.sin(np.pi * f(x_range)) * np.sin(np.pi * y_range[:,
                ↪ None])

            u_frames[t] = -1 * np.gradient(psi, axis=1) # Partial derivative
                ↪ with respect to y
            v_frames[t] = np.gradient(psi, axis=0) # Partial derivative with
                ↪ respect to x

    sample_frame_index = 20
    plt.quiver(x_range, y_range, u_frames[sample_frame_index, :, :], v_frames
        ↪ [sample_frame_index, :, :])
    plt.title('Velocity field for 200 time frames')
    plt.xlabel('x')
    plt.ylabel('y')
    plt.show()
```

The velocity field scatter of x and y in time frames 200:

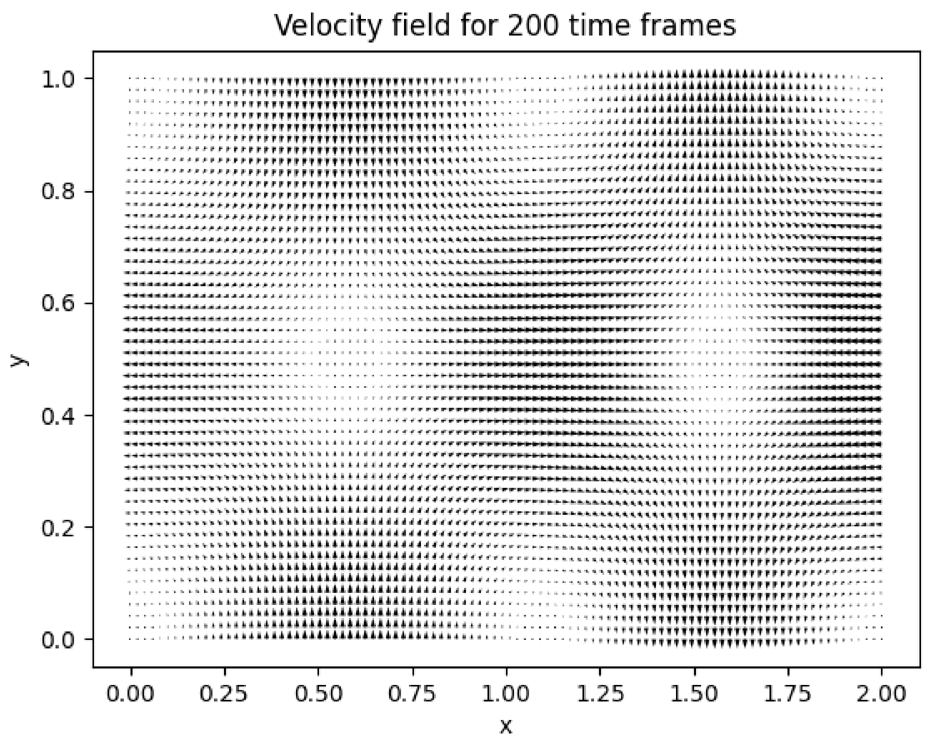


Figure 5: Y and X position for 200 frames in the domain [0,2] and [0,1]

8.2

To reshape the vectors in a D matrix with dimensions (2 × N × M) × t.

```
        D = np.stack((u_frames, v_frames), axis=0).reshape((2, len(y_range), len(
            ↪ x_range), t_frames), order='F')

        D = D.reshape((2 * len(x_range) * len(y_range), t_frames), order='F')
```

8.3

To perform the SVD, we utilize the built-in function svd from Scipy:

```
        from scipy.linalg import svd
        U, S, Vt = svd(D)
```

8.4

To find the highest singular values:

```
        plt.plot(S)
        plt.title('Singular Values')
        plt.xlabel('Index')
        plt.ylabel('Magnitude')
        plt.show()
```

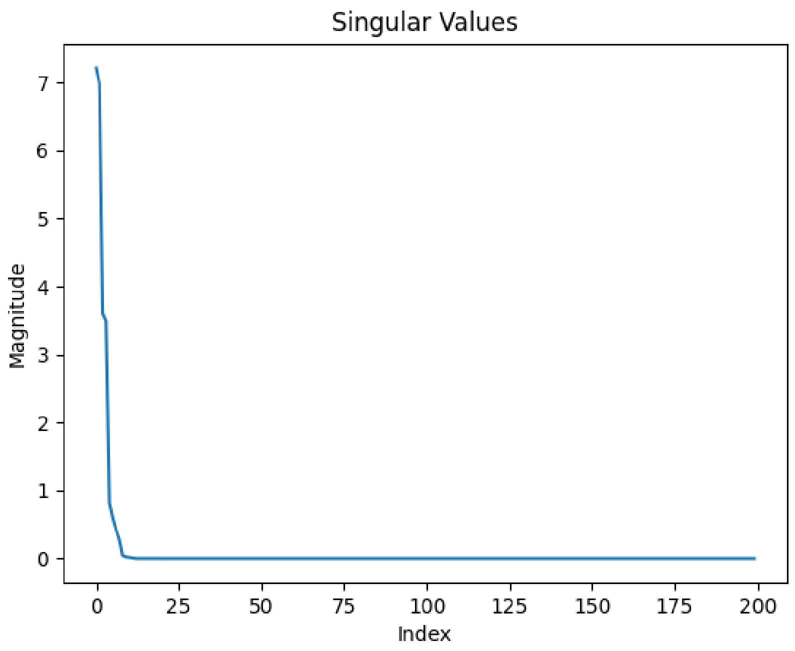


Figure 6: The highest Singular Values and their index

8.5

The simulation that computes the finite time lyapunov exponent and animates the flow:

```
num_modes = 5
num_frames = 200
N=100
M=50
temporal_basis = U[:, :num_modes]

FTLE = np.log(S[:num_modes]).reshape((num_modes, 1)) / num_frames

from matplotlib.animation import FuncAnimation
fig, ax = plt.subplots()
def update(frame):
        ax.clear()
        ax.quiver(x_range, y_range, D[:N * M, frame].reshape((M, N)), D[N
           ↪ * M:,  frame].reshape((M, N)))
        ax.set_title(f'Frame {frame}')

ani = FuncAnimation(fig, update, frames=num_frames, interval=50)
plt.show()
```

The video showing the animation of the phase portrait will be uploaded in GitHub under "Q8."

9 PDEs

To solve the Turning reaction diffusion model of the following system of equations:

$$\frac{\partial a}{\partial t} = d_a \nabla^2 a + F(a, b)$$
$$\frac{\partial b}{\partial t} = d_b \nabla^2 b + G(a, b)$$

Where d_a and d_b are the diffusion coefficient of a and b respectively. By setting a certain equation to the diffusion equation $F(a, b)$ and $F(a, b)$, by doing the finite element to find the diffusion patterns in both the reactions.
The equations can be written in this format to achieve:

- Spots:
$$F(a, b) = u - v * a + a^2 b$$
$$G(a, b) = -a^2 b$$

- Strips
$$F(a, b) = u - v * a - a^2 b$$
$$G(a, b) = +a^2 b$$

Where u and v are arbitrary numbers that represent different diffusion parameters.
The following code is implemented in MATLAB to achieve a simulation of the diffusion of both u and v. The video will be recorded and attached in GitHub.

```
da = 0.1;
db = 0.05;
u = 0.2;
v = 0.1;

L = 40;
Nx = 100;
Nt = 5000;
dt = 0.01;

dx = L / (Nx - 1);
x = linspace(0, L, Nx);
[X, Y] = meshgrid(x, x);


a = 1 - 0.1 * rand(Nx, Nx);
b = 0.1 * rand(Nx, Nx);


for t = 1:Nt
       laplace_a = del2(a, dx);
       laplace_b = del2(b, dx);
       % For spots
       f_spots = u - v * a + a.^2 .* b;
       g_spots = - a.^2 .* b;
       % For strips
       f_strips = u - v * a - a.^2 .* b;
       g_strips = a.^2 .* v;

       dadt = da * laplace_a + f_strips;
       dbdt = db * laplace_b + g_strips;

       a = a + dt * dadt;
       b = b + dt * dbdt;
       % To plot the simulation
       if mod(t, 100) == 0
              subplot(1, 2, 1);
              contourf(X, Y, a, 20, 'LineStyle', 'none');
              title(['a at t = ' num2str(t * dt)]);

              subplot(1, 2, 2);
```

```
            contourf(X, Y, b, 20, 'LineStyle', 'none');
            title(['b at t = ' num2str(t * dt)]);
            pause(0.001)
            hold on
        end
end
```

The simulation video shows the creation of spots in u and v, with very quick diffusion for v and slow diffusion in u. I tried to play with the parameters to fix the simulation but MATLAB kept crashing and my laptop couldn't handle the simulation. Unfortunately as well I couldn't achieve the diffusion of strips even after changing the additional terms in the differential equation of $F(a,b)$ and $G(a,b)$. I will include the outcome in the videos titled "Q9 Strips" and "Q9 Spots."