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:

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

1.2

To find the Fiedler Vector, it represents the second eigenvalue of the Netowrk's Laplacian which is calculated via:

$$L = D - A$$

Where D is the degree matrix and A is the Adjacency Matrix.
The following code generates the adjacency and degree matrix for $N = 10$ coupled harmonic oscillators. and computes the Network's Laplacian along with its perspective eigenvalues and vectors. Where v_1 is indeed the identity vector of ones.
The second eigenvector v_2 is graphed with respect to the Node index or network partition. And indeed we see half f the values corresponding to negative weights and the other half to positive weights.

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.linalg import eigh
from scipy.integrate import odeint

# Parameters
N = 10
m = 1
k = 1
A = np.diag(np.ones(N - 1), k=1) + np.diag(np.ones(N - 1), k=-1)

D = np.diag(np.sum(A, axis=1))

L = D - A
```

```
eigenvalues, eigenvectors = eigh(L)

v2 = eigenvectors[:, 1]

# Visualize the Fiedler vector
plt.plot(v2, 'o-', label='Fiedler vector')
plt.xlabel('Node Index')
plt.ylabel('Fiedler Vector Value')
plt.legend()
plt.grid()
plt.show()
```

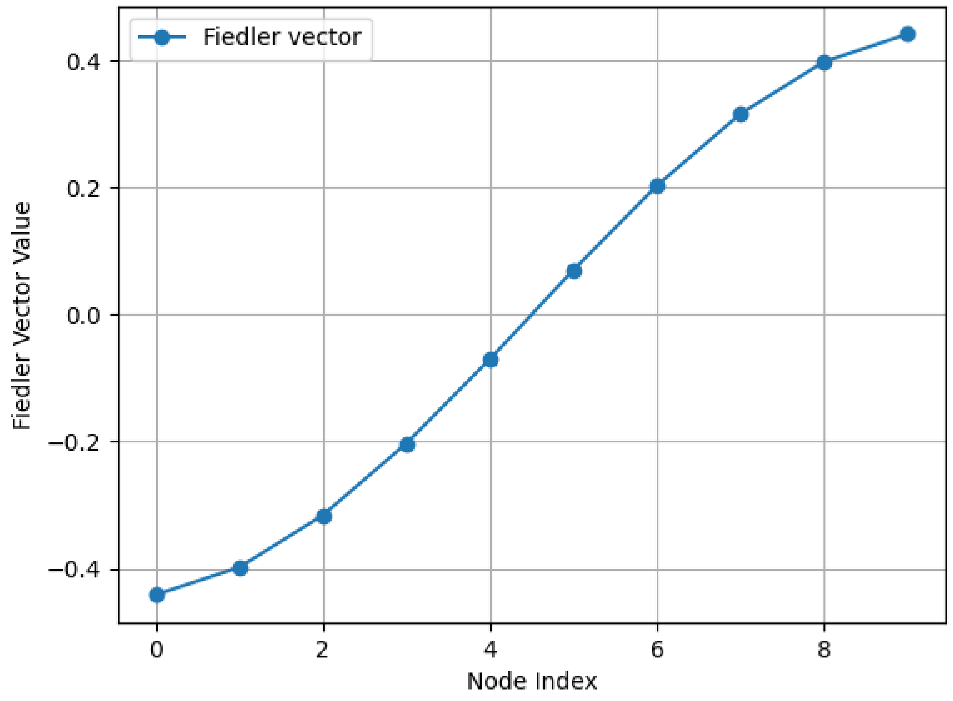


Figure 1: The second eigenvalue with respect to the Node/Network Index

1.3 1.3

To generate the adjacency matricies and to find their perspective Laplacians, the Networkx package is utilized in Python. The following functions can generate the matricies and compute their Laplacians:

```
import networkx as nx
import numpy as np
import matplotlib.pyplot as plt
```

```
def generate_random_network(N, p):
        G = nx.erdos_renyi_graph(N, p)
        return G

def generate_scale_free_network(N, m):
        G = nx.barabasi_albert_graph(N, m)
        return G

def generate_small_world_network(N, k, p):
        G = nx.watts_strogatz_graph(N, k, p)
        return G

# Function to calculate Laplacian matrix
def calculate_laplacian_matrix(G):
        L = nx.laplacian_matrix(G).toarray()
        return L
```

To get the properties of the network, further function in Networkx are utilized:

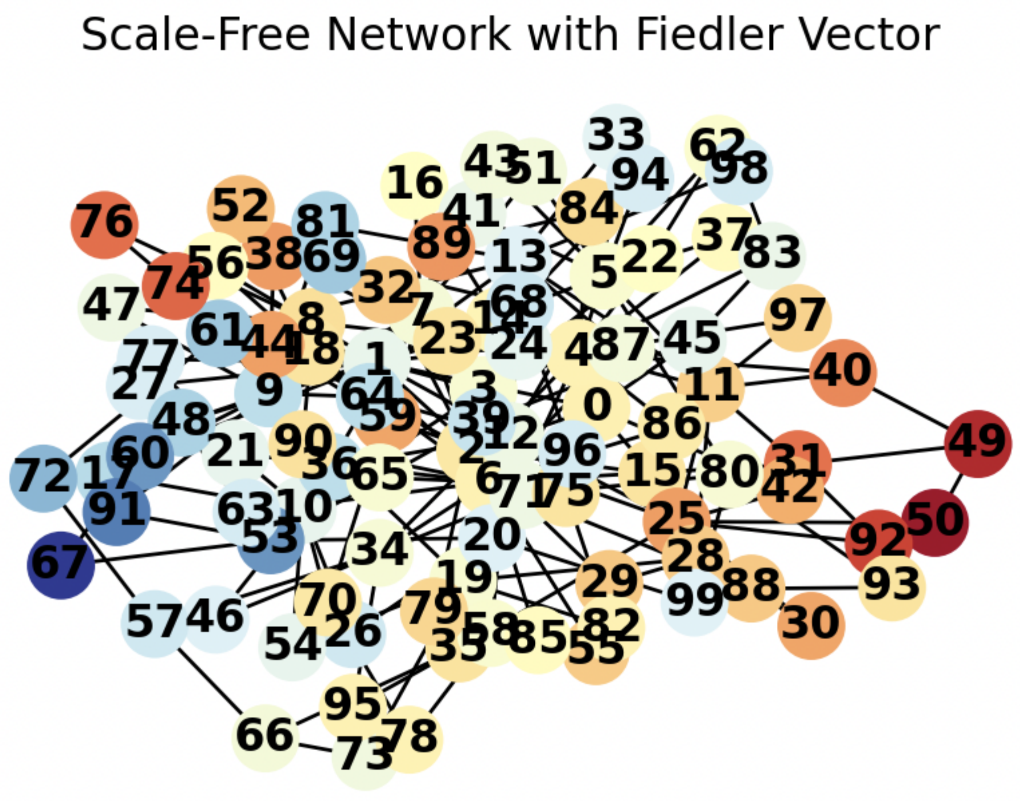
```
def infer_communities(G):
        L = calculate_laplacian_matrix(G)
        eigenvalues, eigenvectors = np.linalg.eigh(L)
        fiedler_vector = eigenvectors[:, 1]

def compute_node_properties(G):
        degree_centrality = nx.degree_centrality(G)
        betweenness_centrality = nx.betweenness_centrality(G)
        eigenvector_centrality = nx.eigenvector_centrality(G)
        clustering_coefficient = nx.clustering(G)
        average_path_length = nx.average_shortest_path_length(G)
        return degree_centrality, betweenness_centrality, eigenvector_centrality,
           ↪  clustering_coefficient, average_path_length
```

Then to perform a scale-free network and compute their properties, utilize the following code:

```
N = 100
m_scale_free = 2

# Generate Scale Free Network networks
scale_free_network = generate_scale_free_network(N, m_scale_free)
fiedler_vector_scale_free = infer_communities(scale_free_network)

# Compute node properties
degree_centrality, betweenness_centrality, eigenvector_centrality,
   ↪ clustering_coefficient, average_path_length = compute_node_properties(
   ↪ scale_free_network)
```

```
print(compute_node_properties(scale_free_network))

nx.draw(scale_free_network, with_labels=True, font_weight='bold', node_color=
    ↪ fiedler_vector_scale_free, cmap=plt.cm.RdYlBu)
plt.title('Scale-Free Network with Fiedler Vector')
```

Then the scale-free network will look something like this:



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.

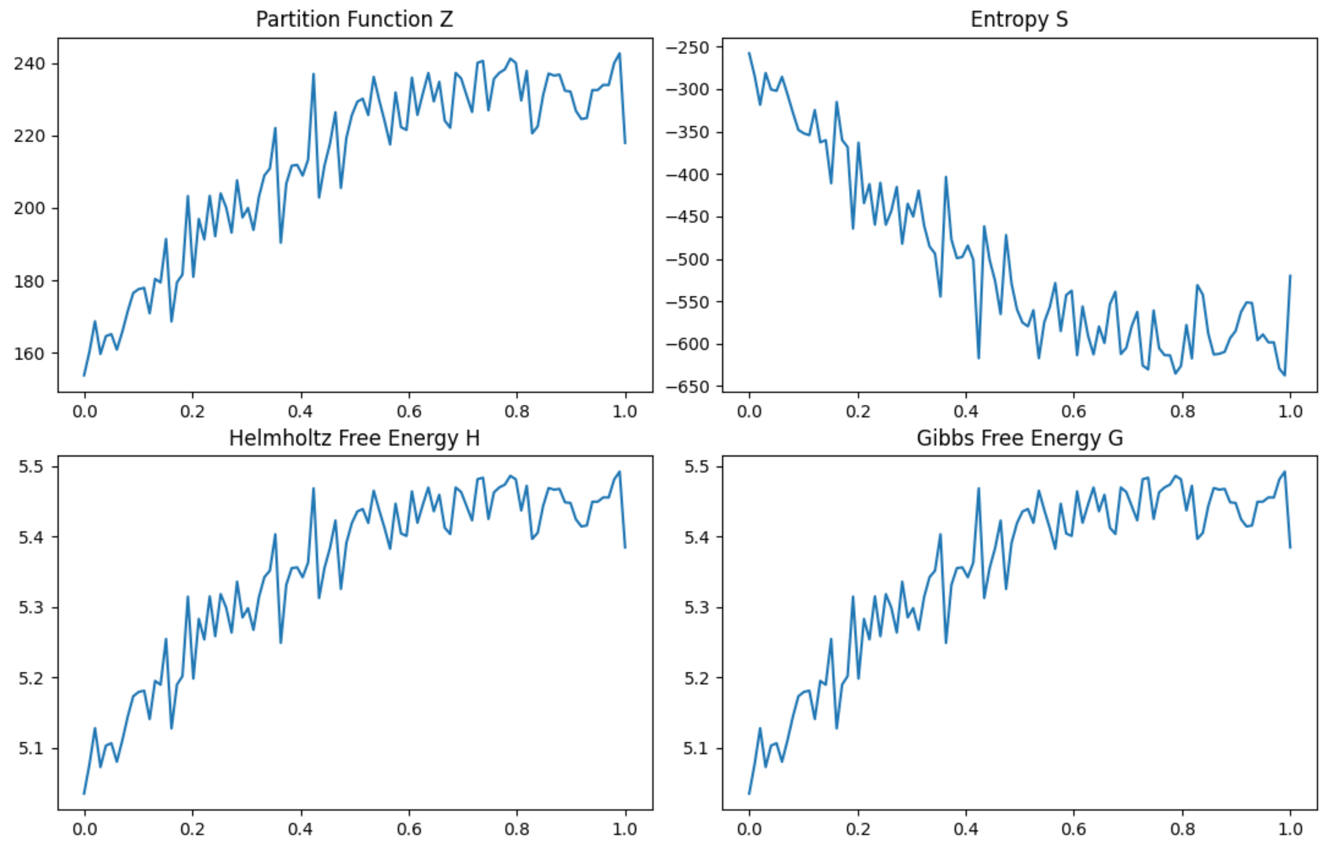
2.1 Small-World Network or Watts-Strogatz Model

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:

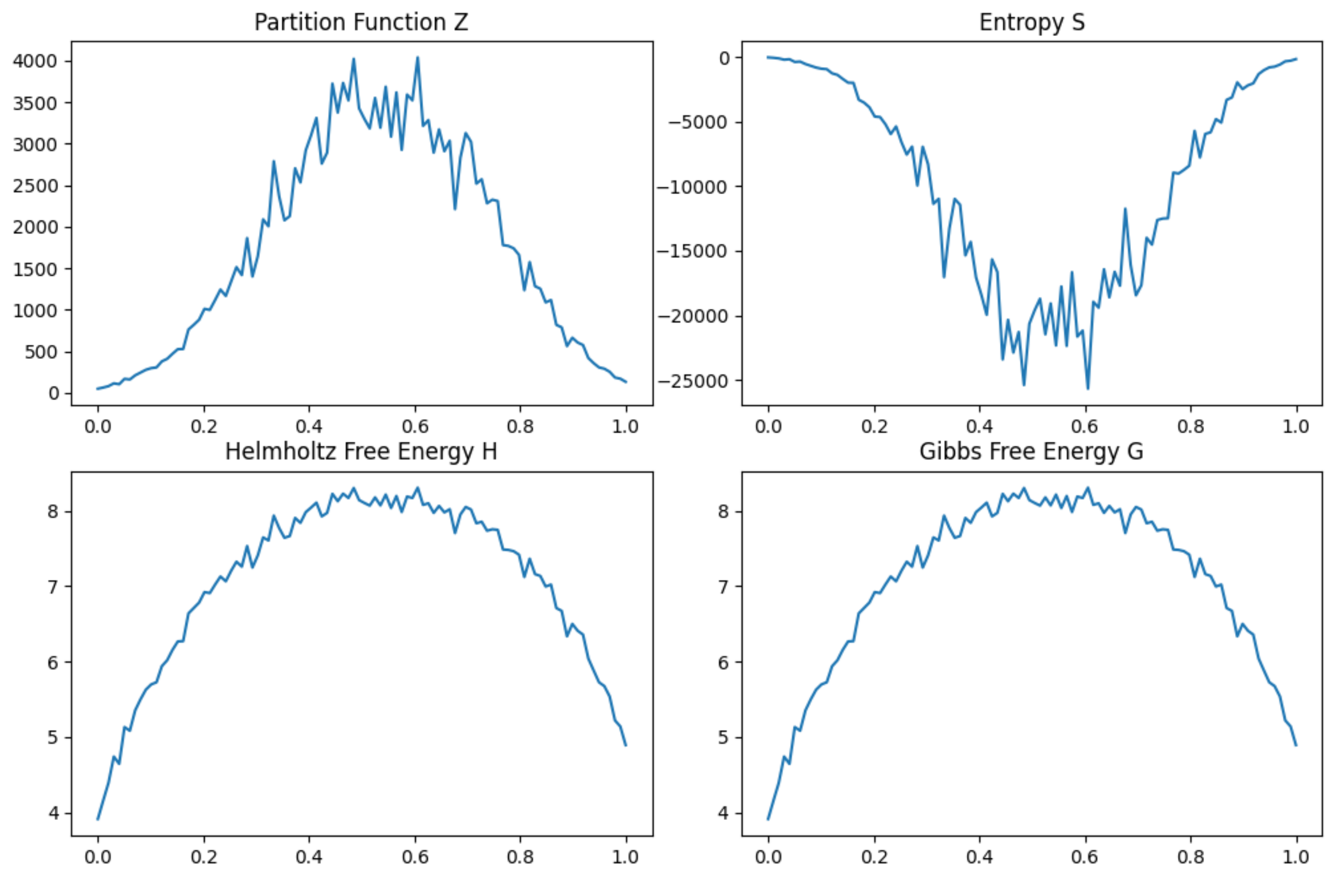


Figure 2: The Thermodynamic Properties of the Random Model

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:

```
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.

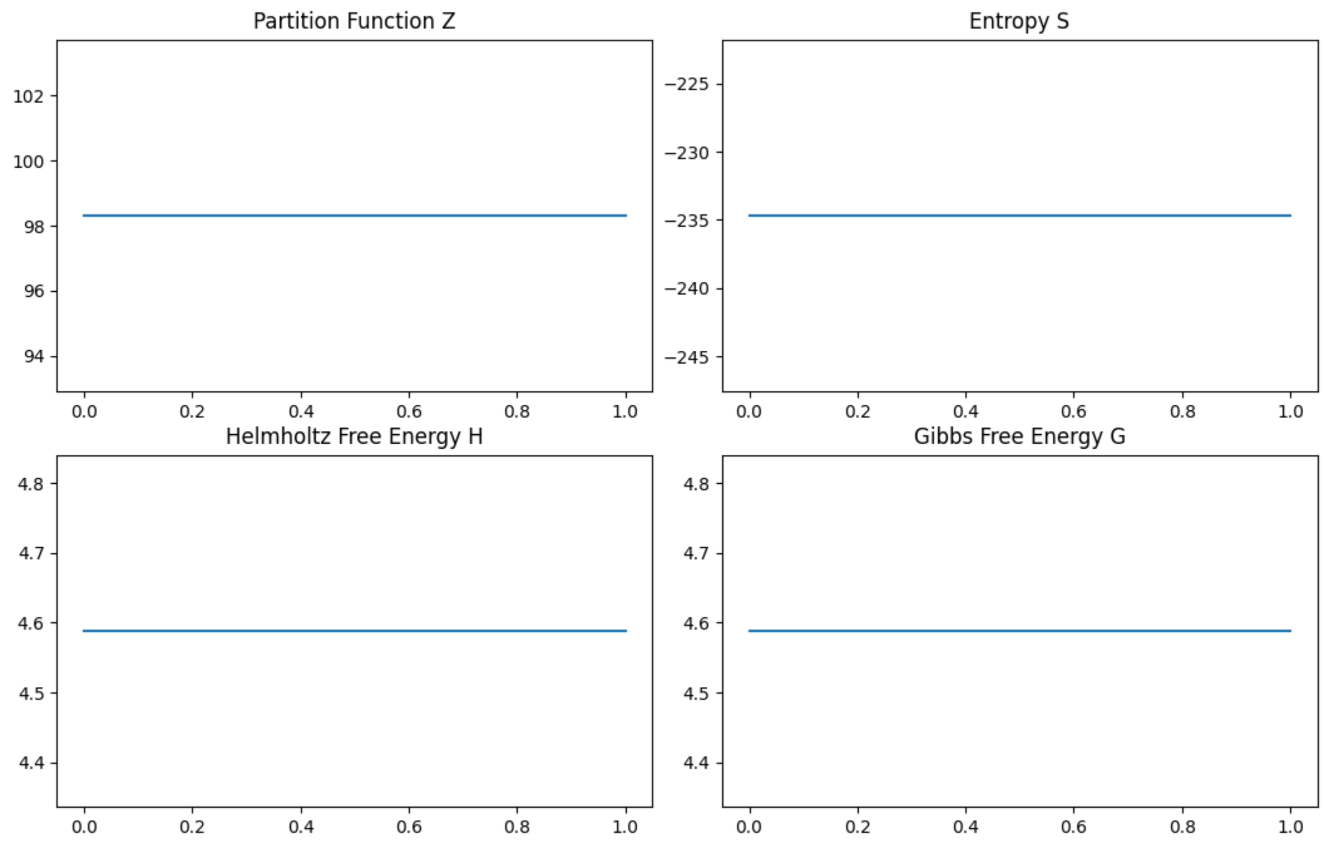


Figure 3: The Thermodynamic Properties of the Regular Model

3 KPZ Universality Class and the Wigner Semi Circle

3.1

First to be able to get the Jacobian matrix, we start from the trasnformation equations:

$$X = \begin{bmatrix} x_{11} & x_{12} \\ x_{12} & x_{22} \end{bmatrix}$$

Combine the eigenvalues in a Lambda Matrix

$$\Lambda = \begin{pmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{pmatrix}$$

Then X can be written in terms of the rotation matrix and the eigenvalues matrix in the form of $X = O\Lambda O^T$
Then X is in the form of:

$$X = \begin{bmatrix} x_{11} & x_{12} \\ x_{12} & x_{22} \end{bmatrix} = \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix} \begin{pmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{pmatrix} \begin{pmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{pmatrix}$$

To find the Jacobian factor, a trick can be done would be to differentiate the large X matrix with respect to λ_1, λ_2, and θ respectively. Then the components of the final matrix will be the

terms found in the Jacobian factor.
Thus the operation proceeds as follows:

$$\frac{\partial X}{\partial \lambda_1} = O\frac{\partial \Lambda}{\partial \lambda_1}O^T = \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{pmatrix} = \begin{pmatrix} \cos^2(\theta) & \sin(\theta)\cos(\theta) \\ \sin(\theta)\cos(\theta) & \sin^2(\theta) \end{pmatrix}$$

Similarly, the same procedure is done for the other terms:

$$\frac{\partial X}{\partial \lambda_2} = O\frac{\partial \Lambda}{\partial \lambda_2}O^T = \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix} \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{pmatrix} = \begin{pmatrix} \sin^2(\theta) & -\sin(\theta)\cos(\theta) \\ -\sin(\theta)\cos(\theta) & \cos^2(\theta) \end{pmatrix}$$

$$\frac{\partial X}{\partial \theta} = \begin{pmatrix} (2\cos\theta\sin\theta)(\lambda_1-\lambda_2) & \cos^2\theta(\lambda_1-\lambda_2) - \sin^2\theta(\lambda_1-\lambda_2) \\ \cos^2\theta(\lambda_1-\lambda_2) - \sin^2\theta(\lambda_1-\lambda_2) & -2\cos\theta\sin\theta)(\lambda_1-\lambda_2) \end{pmatrix}$$

MATLAB symbolic package can be used to quickly compute the derivatives quickly:

```
>> syms t1 t2 t
>> X = [cos(t) -sin(t);sin(t) cos(t)] * [t1 0;0 t2] * [cos(t) sin(t);-sin
   ↪ (t) cos(t)]

X =

[ t1*cos(t)^2 + t2*sin(t)^2, t1*cos(t)*sin(t) - t2*cos(t)*sin(t)]
[t1*cos(t)*sin(t) - t2*cos(t)*sin(t), t2*cos(t)^2 + t1*sin(t)^2]

>> diff(X,t1)

ans =

[ cos(t)^2, cos(t)*sin(t)]
[cos(t)*sin(t), sin(t)^2]

>> diff(X,t2)

ans =

[ sin(t)^2, -cos(t)*sin(t)]
[-cos(t)*sin(t), cos(t)^2]

>> diff(X,t)

ans =

[ 2*t2*cos(t)*sin(t) - 2*t1*cos(t)*sin(t), t1*cos(t)^2 - t2*cos(t)^2 - t1
   ↪ *sin(t)^2 + t2*sin(t)^2]
[t1*cos(t)^2 - t2*cos(t)^2 - t1*sin(t)^2 + t2*sin(t)^2, 2*t1*cos(t)*sin(t
   ↪ ) - 2*t2*cos(t)*sin(t)]
```

Now the Jacobian can be easily written as

$$\begin{pmatrix} \frac{\partial x_{11}}{\partial \lambda_1} & \frac{\partial x_{22}}{\partial \lambda_1} & \frac{\partial x_{12}}{\partial \lambda_1} \\ \frac{\partial x_{11}}{\partial \lambda_2} & \frac{\partial x_{22}}{\partial \lambda_2} & \frac{\partial x_{12}}{\partial \lambda_2} \\ \frac{\partial x_{11}}{\partial \theta} & \frac{\partial x_{22}}{\partial \theta} & \frac{\partial x_{12}}{\partial \theta} \end{pmatrix} = \begin{pmatrix} \cos^2\theta & \sin^2\theta & \cos\theta\sin\theta \\ \sin^2\theta & \cos^2\theta & -\cos\theta\sin\theta \\ (2\cos\theta\sin\theta)(\lambda_1-\lambda_2) & -(2\cos\theta\sin\theta)(\lambda_1-\lambda_2) & \cos^2\theta(\lambda_1-\lambda_2)-\sin^2\theta(\lambda_1-\lambda_2) \end{pmatrix}$$

The determinant can be found with MATLAB's syms:

```
>> J = [cos(t)^2 sin(t)^2 cos(t)*sin(t);sin(t)^2 cos(t)^2 -cos(t)*sin(t)
   ↪ ;(2*cos(t)*sin(t))*(t1-t2) (2*cos(t)*sin(t))*(t1-t2) (cos(t)^2 -
   ↪ sin(t)^2)*(t1-t2)]

J =

[ cos(t)^2, sin(t)^2, cos(t)*sin(t)]
[ sin(t)^2, cos(t)^2, -cos(t)*sin(t)]
[2*cos(t)*sin(t)*(t1 - t2), 2*cos(t)*sin(t)*(t1 - t2), (cos(t)^2 - sin(t)
   ↪ ^2)*(t1 - t2)]

>> D = det(J)

D =

t1*cos(t)^6 - t2*cos(t)^6 + t1*sin(t)^6 - t2*sin(t)^6 - t1*cos(t)^2*sin(t
   ↪ )^4 - t1*cos(t)^4*sin(t)^2 + t2*cos(t)^2*sin(t)^4 + t2*cos(t)^4*sin
   ↪ (t)^2

>> simplify(D)

ans =

((cos(4*t) + 1)*(t1 - t2))/2
```

Which gives the determinant of J = $(\lambda_1 - \lambda_2)(\frac{1}{2} + \frac{\cos(4\theta)}{2})$ I checked my answer multiple times but it gave the same answer. For the sake of the question I'll approximate the determinant of the Jacobian as $\lambda_1 - \lambda_2$. Then the probability can be written as:

$$e^{\frac{1}{2}(\lambda_1^2+\lambda_2^2)} det|J(\lambda_1,\lambda_2,\theta)| d\lambda_1 d\lambda_2 d\theta = e^{-\frac{\beta}{2}\sum_{i=1}^N \lambda_i^2} \prod_{1\le i<j\le N} = \exp(-\frac{\beta}{2}\sum_{i=1}^N \lambda_i^2 + \beta \log|\lambda_1 - \lambda_2|)$$

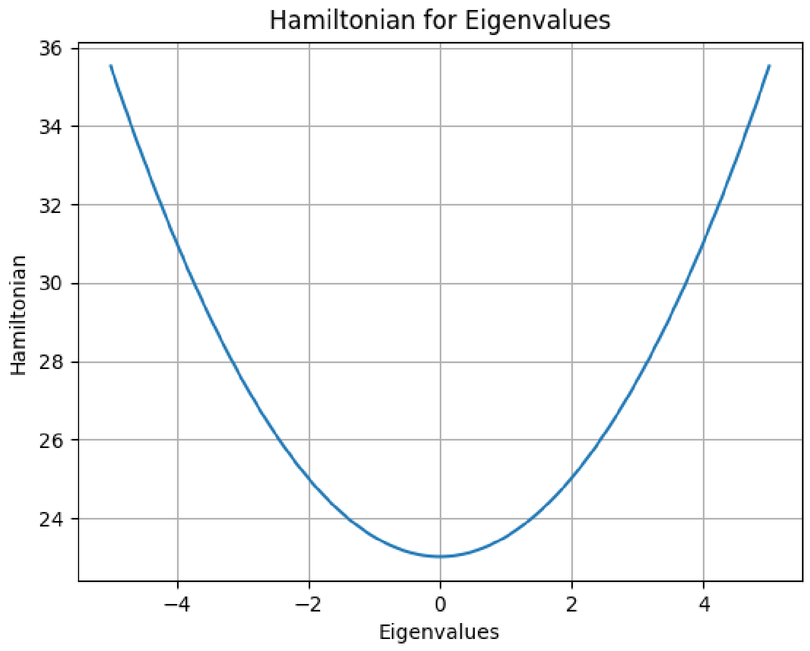
3.2

The Hamiltonian is then written as:

$$\hat{H} = \frac{1}{2}\sum_{i=1}^N \lambda_i^2 - \sum_{i=1}^N\sum_{j=1}^N \log|\lambda_i - \lambda_j|$$

T interpret the terms of the Hamiltonian: The first term $\frac{1}{2}\sum_{i=1}^{N}\lambda_i^2$ represents the kinetic energy term. It penalizes large values of eigenvalues, indicating a preference for smaller eigenvalues. This term favors small eigenvalues. The eigenvalues are expected to be centered around zero or have a tendency to be small.
The second term $\sum_{i=1}^{N}\sum_{j=1}^{N}\log|\lambda_i - \lambda_j|$ is the potential energy term. It penalizes eigenvalues that are close to each other, promoting a spread or repulsion between the eigenvalues. This term penalizes eigenvalues that are close to each other. As a result, the eigenvalues are expected to repel each other, leading to a spread across the eigenvalue space.
To graph the Hamiltonian as a function of the eigenvalues, the following Python code has been executed:

```
import numpy as np
import matplotlib.pyplot as plt
def hamiltonian(eigenvalues):
      N = len(eigenvalues)
      kinetic_energy = 0.5 * np.sum(eigenvalues**2)
       potential_energy = -np.sum(np.log(np.abs(np.subtract.outer(
          ↪ eigenvalues, eigenvalues) +  1e-10)))
       return kinetic_energy + potential_energy

eigenvalues_range = np.linspace(-5, 5, 100)

hamiltonian_values = [hamiltonian(np.array([eig])) for eig in
   ↪ eigenvalues_range]

plt.plot(eigenvalues_range, hamiltonian_values)
plt.xlabel('Eigenvalues')
plt.ylabel('Hamiltonian')
plt.title('Hamiltonian for Eigenvalues')
plt.grid()
plt.show()
```



The Hamiltonian exhibits predictable parabolic behavior.

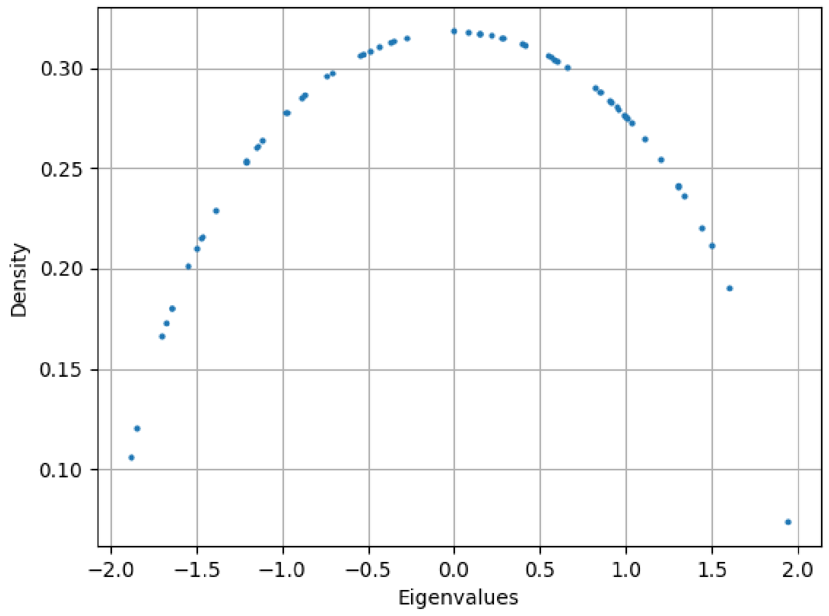
3.3

To find the densities as per the variation of the eigenvalues, the following code is executed:

```
import numpy as np
import matplotlib.pyplot as plt

def erdos_renyi_spectral_density(eigenvalues, r):
       density = np.sqrt(4 - eigenvalues**2) / (2 * np.pi)
       density[np.abs(eigenvalues) > 2 * r] = 0
       return density

# Parameters
N = 1000
p = 0.2
r = np.sqrt(p * N * (1 - p))

eigenvalues = np.random.uniform(-2 * r, 2 * r, N)

density = erdos_renyi_spectral_density(eigenvalues, r)

plt.plot(eigenvalues, density, 'o', markersize=2)
plt.xlabel('Eigenvalues')
plt.ylabel('Density')
plt.grid()
```

```
        plt.show()
```

The code outputs the following graph that shows a maxima around zero, and the densities diminishing as it approaches the boundary of -2r and 2r.



4 Further Notes

Explored thoroughly !

5 Dynamics

5.1

The Lorenz system of the following function:

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

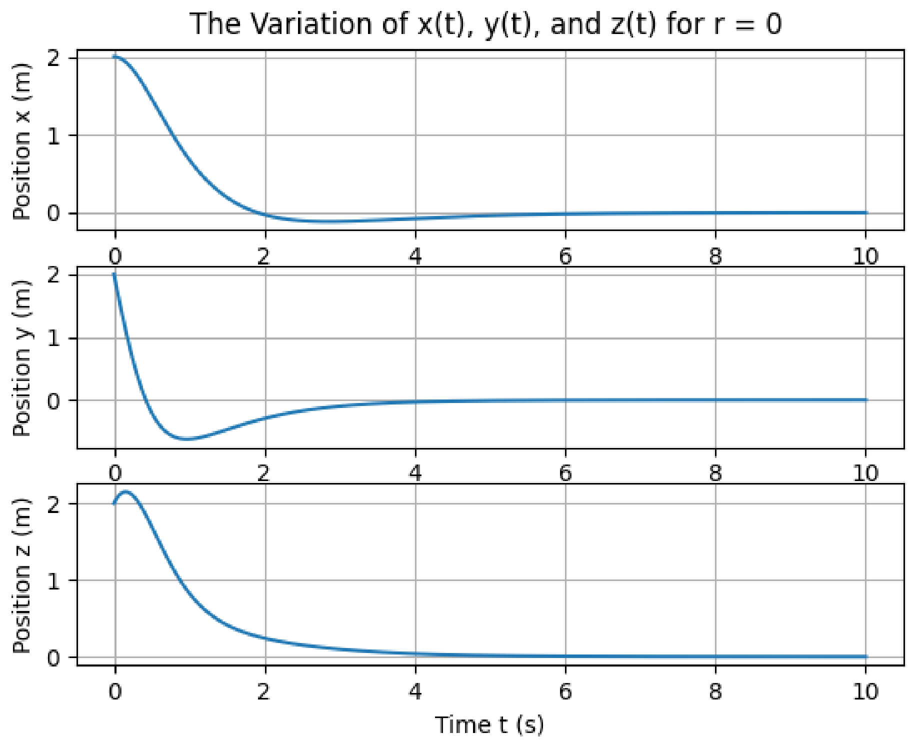
The following code will be used to construct an RK4 approximation to plot x, y, and z in function of time. To assist the code the following parameters were taken: $\sigma,\ b,\ r = 1$ and the initial conditions of $x, y, z = 2$.

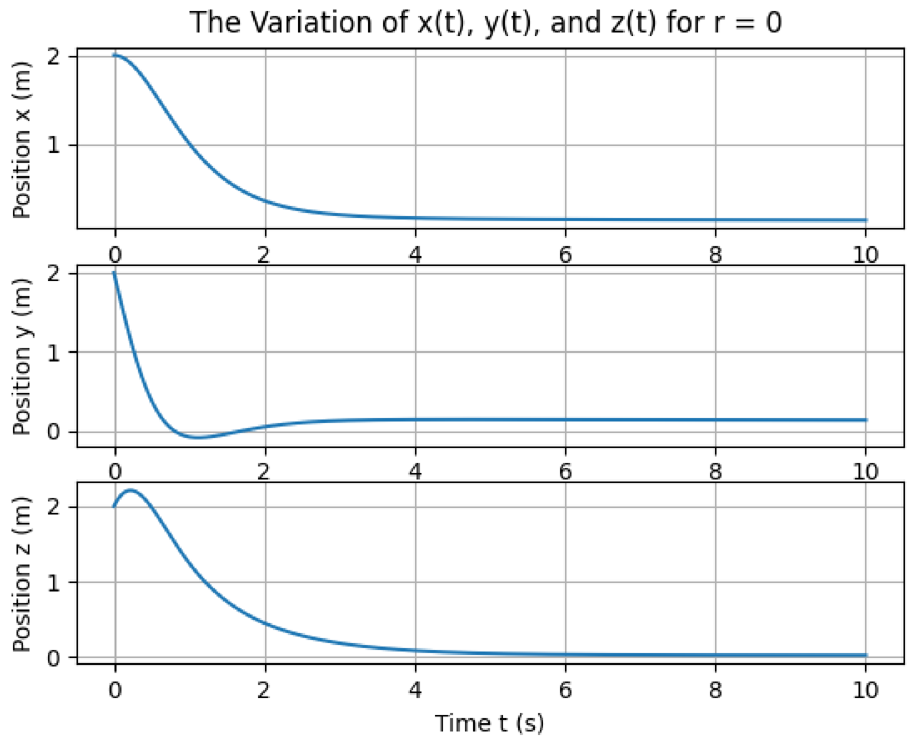
```
        import numpy as np
        import scipy as sp
        import math
```

```
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.subplot(3,1,1)
plt.plot(t,xpoints)
plt.xlabel('Time t (s)')
plt.ylabel('Position x (m)')
plt.grid()
plt.subplot(3,1,2)
plt.plot(t,ypoints)
plt.xlabel('Time t (s)')
plt.ylabel('Position y (m)')
plt.grid()
plt.subplot(3,1,3)
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. This will be done for three perspective values of the bifurcation coefficient r. It will be done for $r = 0,\ 1,\ 5$


The Variation of x(t), y(t), and z(t) for r = 0
Position x (m)
Position y (m)
Position z (m)
Time t (s)


The Variation of x(t), y(t), and z(t) for r = 0
Position x (m)
Position y (m)
Position z (m)
Time t (s)

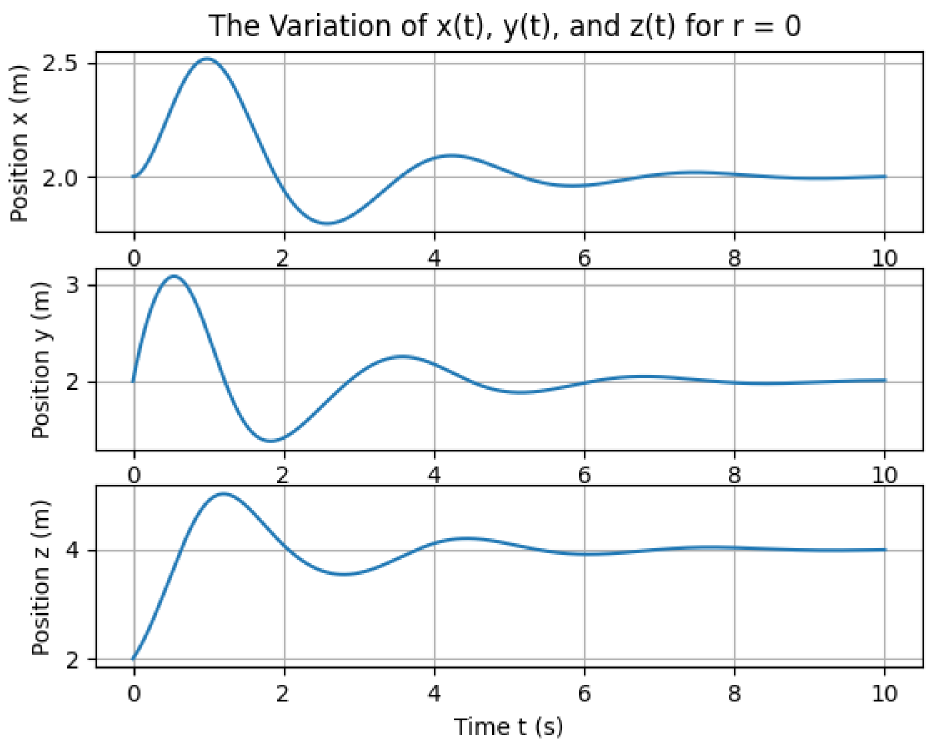


Figure 4: The solution of the Lorenz System done via RK4

If one wants to visualize the Lorenz attractor in 3d projection with the evolution of x, y, and z. The code is executed as such:

```
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
n=10000
s = 10
cmap = plt.cm.winter
for i in range(0, n - s, s):
       ax.plot(xpoints[i:i + s + 1], ypoints[i:i + s + 1], zpoints[i:i + s + 1],
          ↪  color=cmap(i / n), alpha=0.4)

plt.grid()
plt.show()
```

Here the literature suggests to choose these parameters: $\sigma = 10, b = 8/3, r = 28$

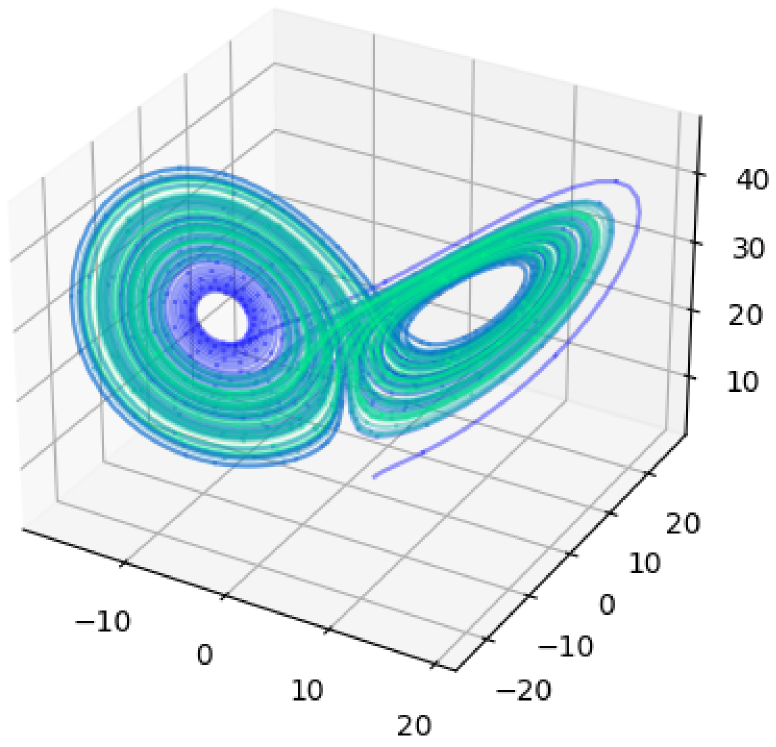


Figure 5: The Lorenz Atrractor plotted in 3d Projection

5.2

To find the fixed points, we import fsolve from scipy.optimize, the code will thus compute the two fixed points numerically.
The code that will execute the operation:

```
from scipy.optimize import fsolve
sigma = 10
r = 28
b =8/3
def fixed_points_equations(xyz, sigma, r, b):
        x, y, z = xyz
        eq1 = sigma * (y - x)
        eq2 = r * x - y - x * z
        eq3 = x * y - b * z
        return [eq1, eq2, eq3]
```

```
initial_guess = [(0, 0, 0), (1, 1, 1)] # Guesses
fixed_points = [fsolve(fixed_points_equations, guess, args=(x, y, z)) for guess
    ↪ in initial_guess]

print(fixed_points)
```

The code outputs the two fixed points: $(x = 0, y = 0, z = 0)$ and $(x = 2.008566218225481, y = 2.008566218225481, z = 1.0100387285512362)$
These values are consistent even when the parameters σ, b, r are changed.
Setting the bifurcation parameter $r > r_c > 1$, a pair of fixed points points other than the origin emerge. To do so set each derivative to zero.

- $$\frac{dx}{dt} = 0 \rightarrow \sigma(y - x) = 0 \rightarrow x = y$$

- $$\frac{dz}{dt} = 0 \rightarrow x^2 - bz = 0 \rightarrow z = \frac{x^2}{b}$$

- $$\begin{aligned} \frac{dy}{dt} &= 0 \\ x(r-1) - xz &= 0 \;(\div x) \\ (r-1) - z &= 0 \\ -\frac{x^2}{b} + (r-1) &= 0 \;(\times - b) \\ x^2 - b(r-1) &= 0 \end{aligned}$$

  The resulting quadratic equation results in:

  $$x^{\pm} = \pm\sqrt{b(r-1)}$$

  This corresponds to the following two fixed points:

  $$x^* = y^* = \pm\sqrt{b(r-1)},\; z = r - 1$$

  To test for stability we compute the Jacobian of the matrix and plug in one of the fixed points.

  $$\begin{pmatrix} \frac{d\dot{x}}{dx} & \frac{d\dot{x}}{dy} & \frac{d\dot{x}}{dz} \\ \frac{d\dot{y}}{dx} & \frac{d\dot{y}}{dy} & \frac{d\dot{y}}{dz} \\ \frac{d\dot{z}}{dx} & \frac{d\dot{z}}{dy} & \frac{d\dot{z}}{dz} \end{pmatrix} = \begin{pmatrix} -\sigma & \sigma & 0 \\ r - z & -1 & -x \\ y & x & -b \end{pmatrix} = \begin{pmatrix} -\sigma & \sigma & 0 \\ 1 & -1 & -\sqrt{b(r-1)} \\ \sqrt{b(r-1)} & \sqrt{b(r-1)} & -b \end{pmatrix}$$

  To get the stability of the fixed points, take the special values of the parameters σ, b, r and compute the eigenvalues of the Jacobian with the np.linalg.eigvals() function:

```
import sympy as sp
import numpy as np
# Define the symbolic variables

sigma = 10
b = 8/3
r = 28
A = np.array([
[-sigma, sigma, 0],
[1, -1, -np.sqrt(b*(r-1))],
[np.sqrt(b*(r-1)), np.sqrt(b*(r-1)), -b]
])

eigenvalues = np.linalg.eigvals(A)

print("Eigenvalues:", eigenvalues)
```

The code outputs the following eigenvalues: Eigenvalues: [-13.85457791 +0.j], [0.09395562+10.19450522j], and [0.09395562-10.19450522j].
One real and a pair of imaginary eigenvalues. The pitchfork bifurcation usually results in a stable fixed point around the center and a pair of unstable fixed points when the bifurcation parameter is above the bifurcation point. Similar to this case when $r > r_c$ *or* $r > 1$.

5.3

To get the Liaponov exponent, we first pick an initial point with coordinates $(1, 1, 1)$. Then we pick a point direct next to it with an additional factor $(1 + 1e - 6, 1 + 1e - 6, 1 + 1e - 6)$. Then we attempt to linearize the equation as such:

$$\delta = \delta_0 e^{\lambda t}$$

$$\ln(\delta) = \lambda t + \ln(\delta)$$

One can plot the variation of the distance between the two points varies with each iteration, then the slope of the linear fit would the exponent. Note that this exponent is unique to the initial conditions, iteration, and parameters chosen. The code to compute the exponent is as such:

```
    def calculate_lyapunov_exponent(initial_condition, t_max, dt):
        t_values, x0 = lorenz(sigma, rho, beta, t_max, dt)
        x1 = initial_condition + 1e-6 * np.random.rand(3)
        distances = np.linalg.norm(x0 - x1, axis=1)
        fit = np.polyfit(t_values, np.log(distances), 1)
        return fit[0]

    initial_condition = np.array([1.0, 1.0, 1.0])
    lyapunov_exponent = calculate_lyapunov_exponent(initial_condition, t_max,
      ↪ dt)
```

For the following conditions: $\sigma = 10, r = 28, b = 8/3, x0 = y0 = z0 = 1$, the code outputs the following Lyaponov exponent: -0.0031681728464301416

5.4

To find the Lorenz Map the following code is executed were the $find_peaks$ function is utilized to find the maxima of z by their perspective indices in the vector. Then a vector is constructed with the maximum points of z and the point directly after it in the vector as such:

```
from scipy.signal import find_peaks
t_values = np.linspace(0, 50, 5000)
z_values = states[:, 2]

maxima_indices, _ = find_peaks(z_values)
#print(maxima_indices)

z_n = z_values[maxima_indices]
z_n_plus_1 = z_values[maxima_indices + 1]

plt.figure(figsize=(8, 6))
plt.plot(z_n, z_n_plus_1, 'o', markersize=2)
plt.title('Lorenz Map')
plt.xlabel('z_n')
plt.ylabel('z_{n+1}')
plt.grid()
plt.show()
```

The output is the following graph. Unfortunately, it doesn't look close to what the literature suggests. I tried varying the parameters, initial conditions, and time scales but the plot looks identical just with more dots on the plot.

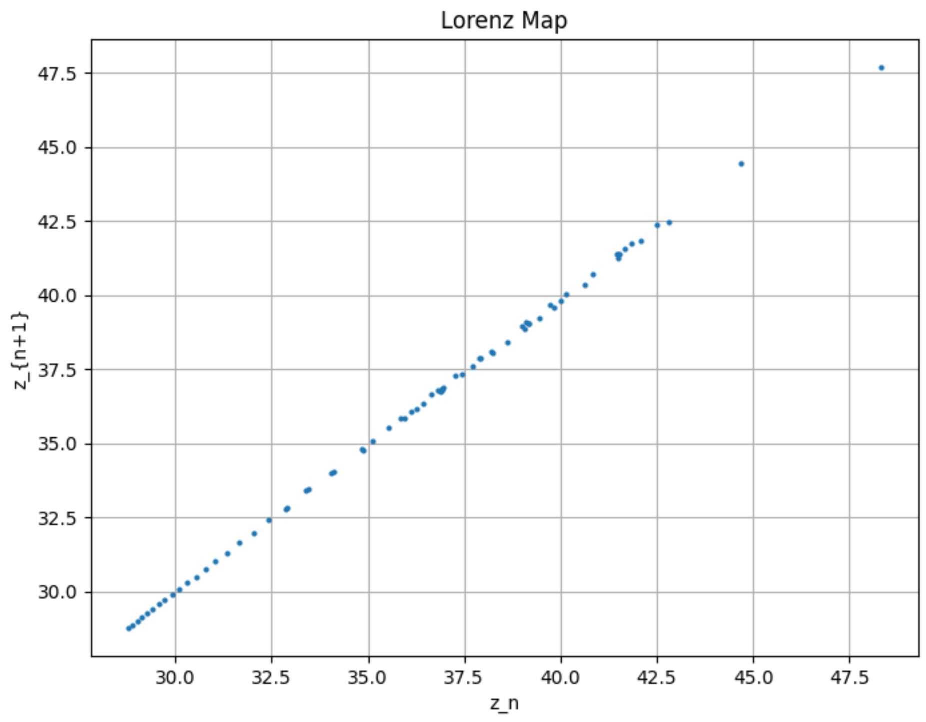


Figure 6: Lorenz Map $\rightarrow z_{n+1}$ vs. z_n

6 Logistic Map

6.1

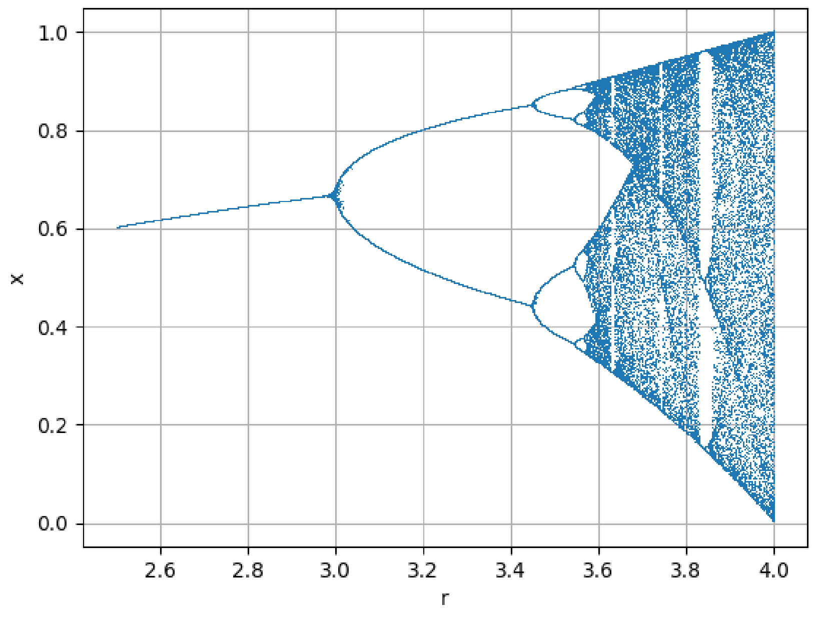
The Logistic Map is represented by the following iterative function:

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

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

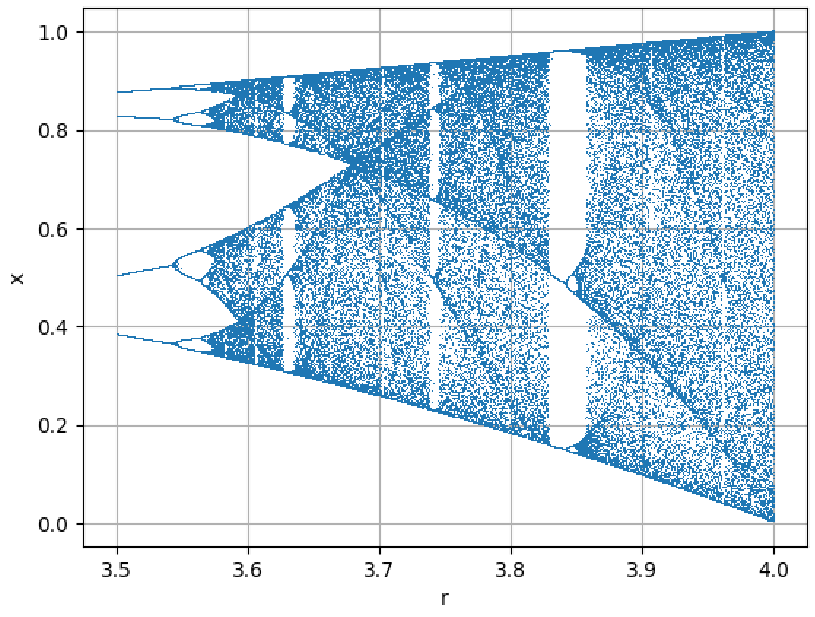
```
import numpy as np
import scipy as sp
import math
import matplotlib.pyplot as plt
import sympy as sy

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

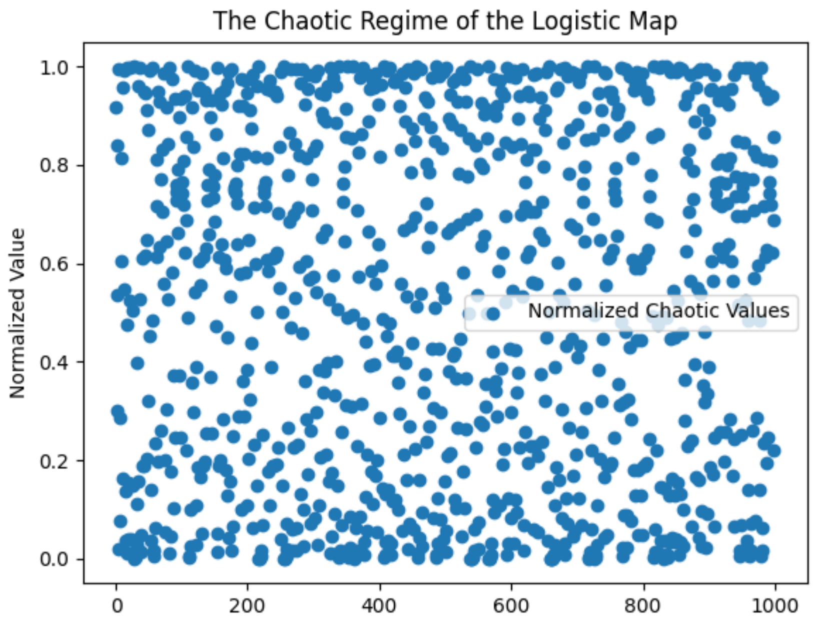
The obvious chaotic regime is seen to be around the values of r between 3.5 and 4. To zoom in to that regime and to find its constituents, the zoomed in figure is as such:



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

To compute the $\delta = \frac{\Delta_{n+1}}{\Delta_n}$ and $\alpha = \frac{d_{n+1}}{d_n}$ values, we require to know at which value of r the period doubles. The following code will compute at what values of r we experience the first period doubling:

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

xpoints = np.zeros((1000,3))
bifurcation_points = []
R = np.linspace(2.85,3.15,1000)
for i in range(len(R)):
        xpoints[i,0] = R[i]
        x = np.random.random()
        x_points = []
        for j in range(500):
                x = R[i]*x*(1-x)
                x_points.append(x)
                if round(x_points[197],2) != round(x_points[198],2):
                bifurcation_points.append(R[i])
                print(bifurcation_points)
                break

xpoints[i][1]= x_points[197]
xpoints[i][2] = x_points[198]
print(xpoints)
plt.scatter(xpoints[:,0],xpoints[:,1],s=1)
plt.scatter(xpoints[:,0], xpoints[:,2],s=1)
plt.show()
```

Which shows that at $r = 2.9794294294294295$ The first period doubling takes place. Unfortunately I couldn't figure out how to write a code that will break when another period doubling takes place. Thus x will oscillate between 4 different values rather than 2. The if statement wouldn't work to break when 4 different values of x are detected.
Another way to do this, this code will only graph one of the converging values of x_n as we iterate over r. It will look for period doubling and break the loop.

```
import numpy as np
import matplotlib.pyplot as plt

def logistic_map(x, r):
        return r * x * (1 - x)

def find_period_doubling(r_values, x0, iterations):
        period_doubling_points = []

        for r in r_values:
                x = x0
                for _ in range(iterations):
                        x = logistic_map(x, r)
                        x_old = x
                        for _ in range(iterations):
                                x = logistic_map(x, r)
                                if np.isclose(x, x_old, rtol=1e-5, atol=1e-8):
                                        period_doubling_points.append((r, x))
                                        break

        return np.array(period_doubling_points)

r_values = np.linspace(2.5, 4.0, 1000)
x0 = 0.5
iterations = 1000

period_doubling_points = find_period_doubling(r_values, x0, iterations)

plt.plot(period_doubling_points[:, 0], period_doubling_points[:, 1], 'bo',
    ↪ markersize=2)
plt.xlabel('r')
plt.ylabel('x')
plt.title('Period Doubling in the Logistic Map')
plt.show()
```

The perspective graph is given by:

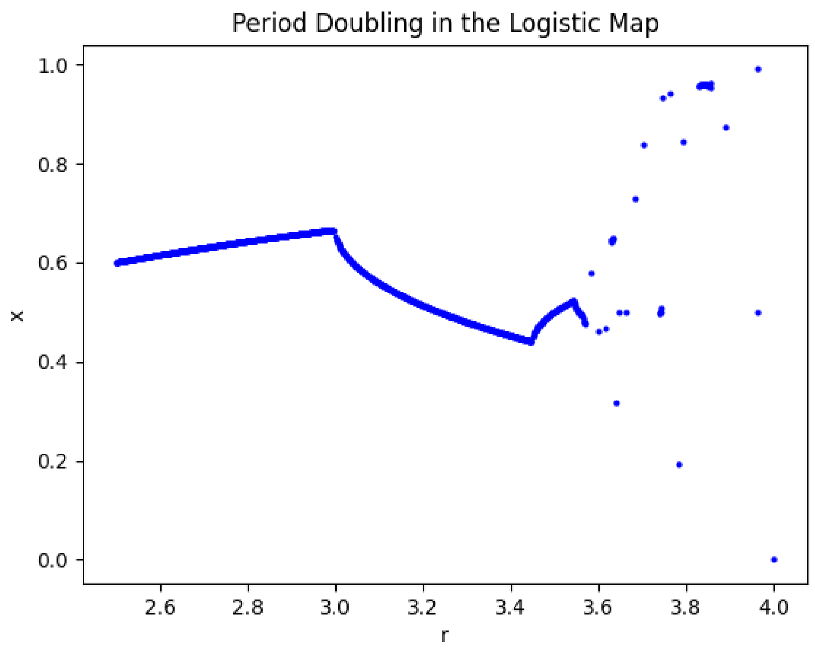


Figure 7: Logistic Graph with Period Doubling Cutoff

Thus analytically from the graph at these points a period doubling happens: $x_1 = 2.99$, $x_2 = 3.455$, and $x_3 = 3.548$. Then $\delta = \frac{x_3 - x_2}{x_2 - x_1} = 0.2$

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

7.1

The Rossler system that constitutes the following PDEs:

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

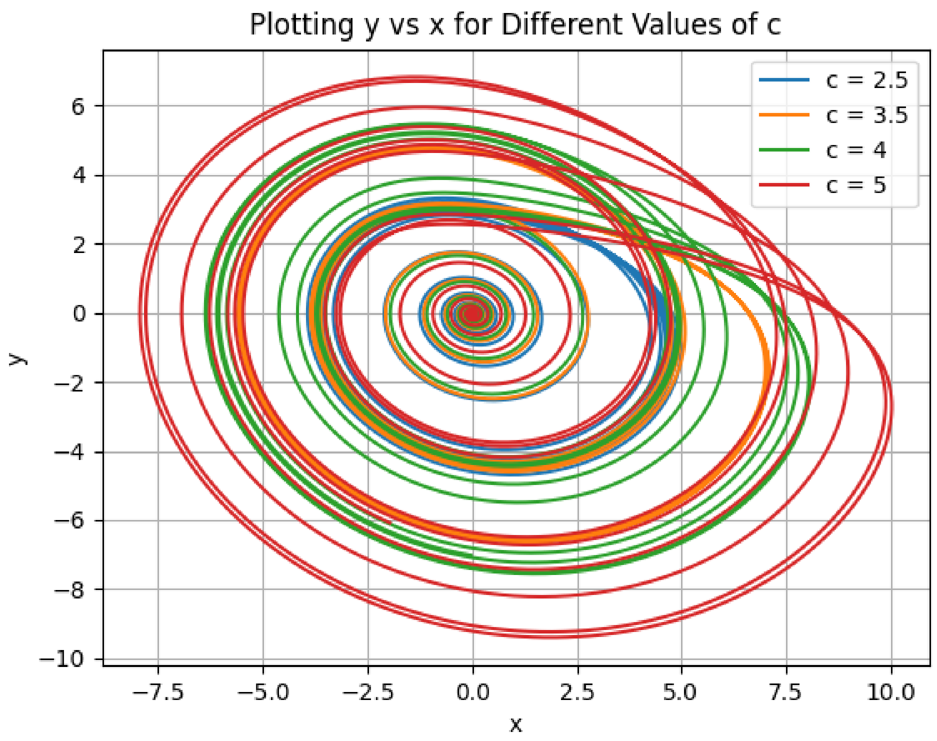
$$\dot{y} = x + ay$$

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

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

```python
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

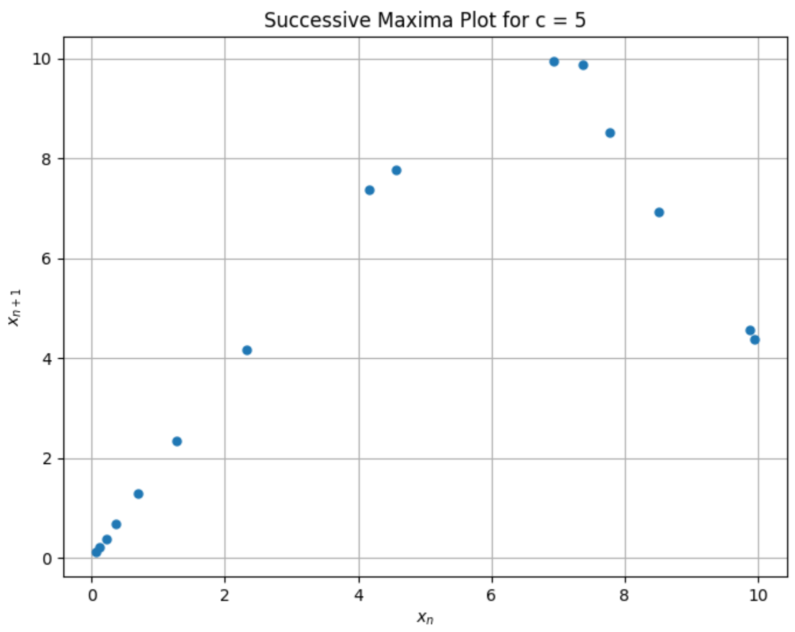


Figure 8: Evolution of x_{n+1} vs. x_n for $c = 5$

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:

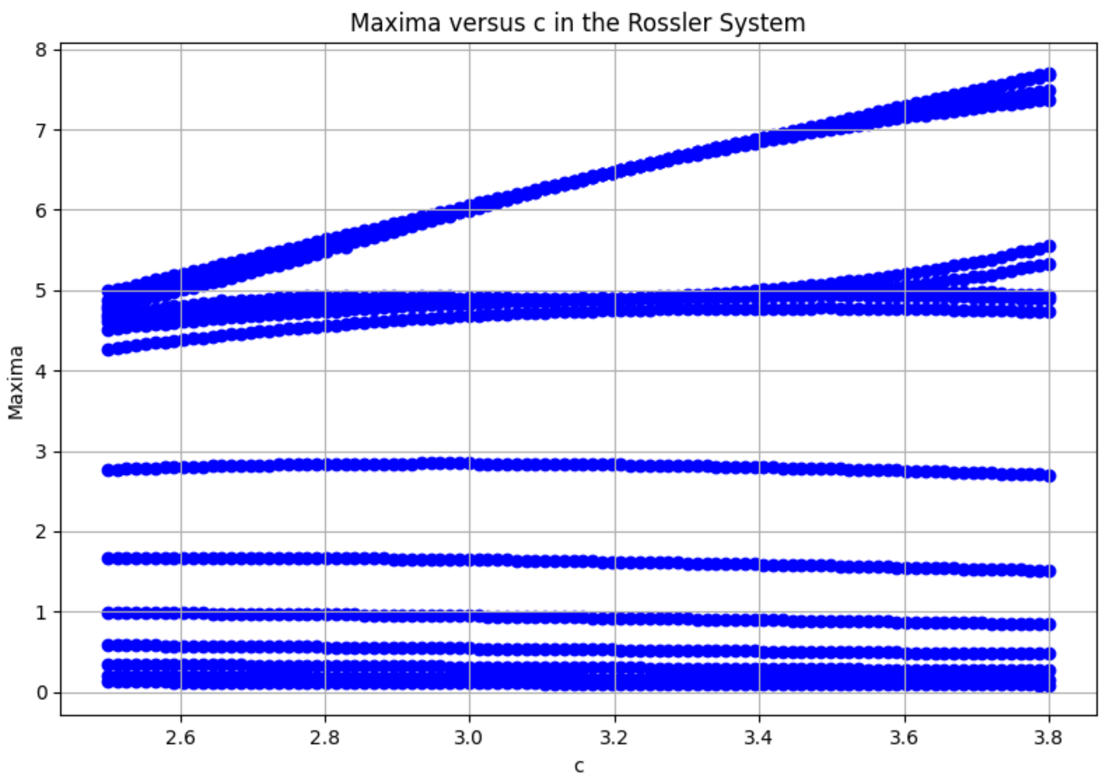


Figure 9: The evolution of the Maxima vs. c

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:

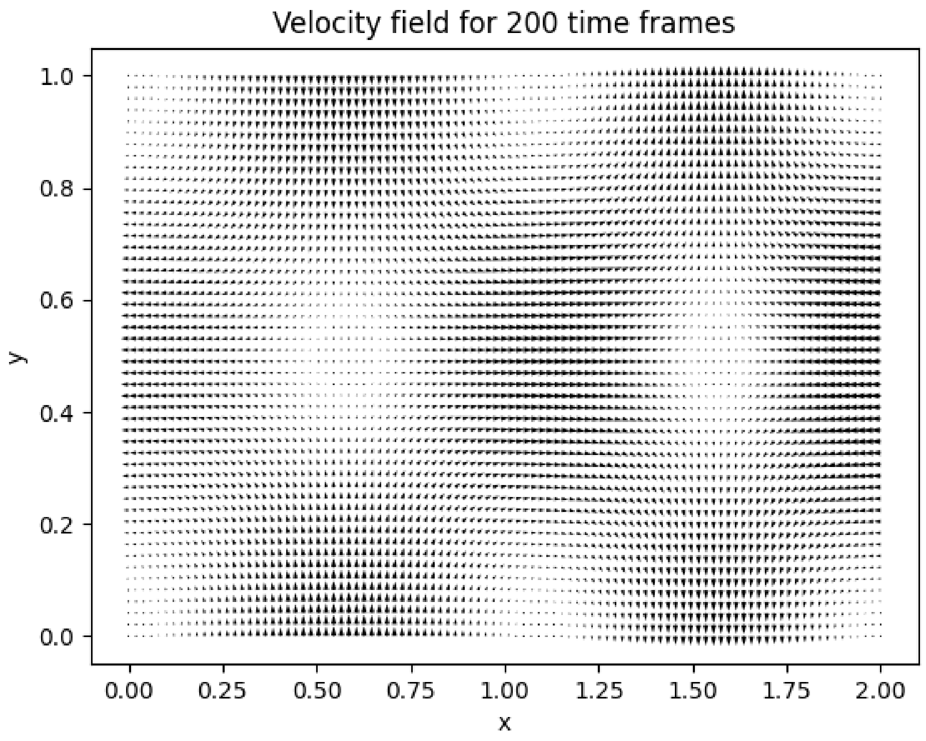


Figure 10: Y and X position for 200 frames in the domain [0,2] and [0,1]

8.2

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

```
        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()
```

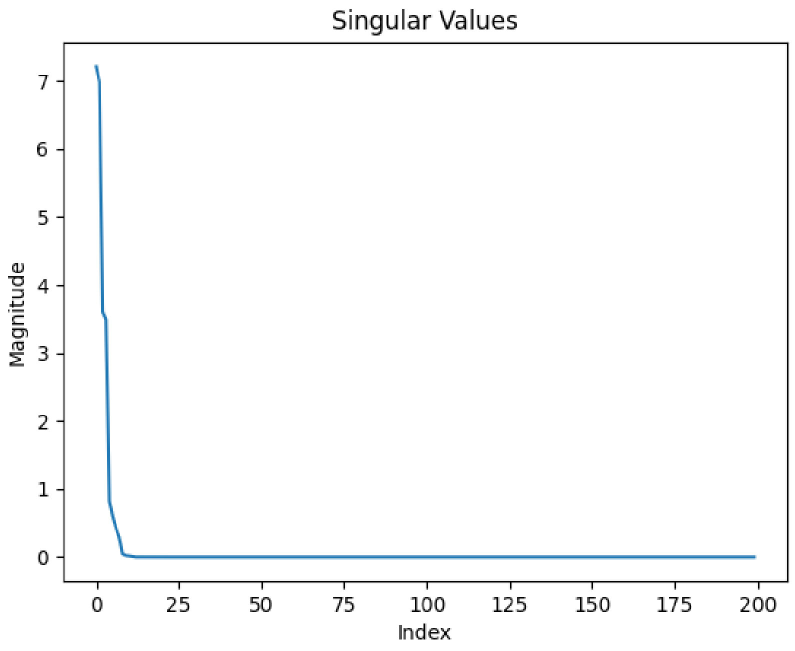


Figure 11: The highest Singular Values and their index

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."