PHYS 222 - Homework II

Fahed Abu Shaer

October 2023

1 Network Laplacian

1) For 4 coupled oscillator,

$$\begin{aligned}
T_1 &= \frac{1}{2}m\dot{x_1}^2 \\
T_2 &= \frac{1}{2}m\dot{x_2}^2 \\
T_3 &= \frac{1}{2}m\dot{x_3}^2 \\
T_4 &= \frac{1}{2}m\dot{x_4}^2
\end{aligned}$$

$$\begin{aligned}
U_1 &= \frac{1}{2}k(x_1 - x_2)^2 \\
U_2 &= \frac{1}{2}k(x_2 - x_1)^2 + \frac{1}{2}k(x_2 - x_3)^2 \\
U_3 &= \frac{1}{2}k(x_3 - x_2)^2 + \frac{1}{2}k(x_3 - x_4)^2 \\
U_4 &= \frac{1}{2}k(x_4 - x_3)^2
\end{aligned}$$

Finding a pattern and generalizing for N coupled harmonic oscillator,

$$\begin{aligned}
T_N &= \sum_{i=1}^{N} \frac{1}{2}m_i\dot{x_i}^2 \\
U_N &= \sum_{i=1}^{N} \frac{1}{2}k(x_i - x_{i-1})^2 + \frac{1}{2}k(x_i - x_{i+1})^2
\end{aligned}$$

Taking the boundary conditions $x_0 = x_1$ and $x_N = x_{N+1}$. The Lagrangian is

$$L = T - U = \sum_{i=1}^{N} \frac{1}{2}m_i\dot{x_i}^2 - \sum_{i=1}^{N} \frac{1}{2}k(x_i - x_{i-1})^2 + \frac{1}{2}k(x_i - x_{i+1})^2$$

So, we can get the equations of motion by taking derivatives of the Lagrangian.

$$\begin{aligned}
\frac{\partial}{\partial t}\left(\frac{\partial L}{\partial \dot{x_i}}\right) &= \frac{\partial}{\partial t}(m_i\dot{x_i}) \\
&= m_i\dot{x_i} \\
\frac{\partial L}{\partial x_i} &= -k(x_i - x_{i-1}) - k(x_i - x_{i+1}) \\
&= -k(-x_{i-1} + 2x_i - x_{i+1})
\end{aligned}$$

The equations of motion are given by the Euler-Lagrange equation.

$$\begin{aligned}
\frac{\partial}{\partial t}\left(\frac{\partial L}{\partial \dot{x_i}}\right) &= \frac{\partial L}{\partial x_i} \\
m_i\dot{x_i} &= -k(x_i - x_{i-1}) - k(x_i - x_{i+1}) \\
&= -k(-x_{i-1} + 2x_i - x_{i+1}) \\
\dot{x_i} &= -\frac{k}{m_i}(-x_{i-1} + 2x_i - x_{i+1})
\end{aligned}$$

2) The equations of motion can be written in the form $\ddot{x} = -Lx$, where $L = -x_{i-1} + 2x_i - x_{i+1}$. I constructed the Laplacian of the system symbolically, and I found the eigenvalues and eigenvectors of the system.

```
m = syp.Symbol('m')
k = syp.Symbol('k')
N = 10
def f(i,j):
    if i == 0 and j == 0 or i == N-1 and j == N-1:
        return 1 * (k/m)
    elif i == j:
        return 2 * (k/m)
    elif i+1 == j or i-1 == j:
        return -1 * (k/m)
    else:
        return 0
L = syp.Matrix(N, N, f)
l = L.eigenvals()
v = L.eigenvects()
display(L)
display(l)
display(v)
```

The Laplacian of the system and its eigenvalues and vectors are,

$$L = \begin{bmatrix} \frac{k}{m} & -\frac{k}{m} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -\frac{k}{m} & \frac{2k}{m} & -\frac{k}{m} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -\frac{k}{m} & \frac{2k}{m} & -\frac{k}{m} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -\frac{k}{m} & \frac{2k}{m} & -\frac{k}{m} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -\frac{k}{m} & \frac{2k}{m} & -\frac{k}{m} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -\frac{k}{m} & \frac{2k}{m} & -\frac{k}{m} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -\frac{k}{m} & \frac{2k}{m} & -\frac{k}{m} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -\frac{k}{m} & \frac{2k}{m} & -\frac{k}{m} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -\frac{k}{m} & \frac{2k}{m} & -\frac{k}{m} \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -\frac{k}{m} & \frac{k}{m} \end{bmatrix}$$

$$\lambda = \left\{ 0,\ \frac{2k}{m},\ \frac{k\left(\frac{3}{2} - \frac{\sqrt{5}}{2}\right)}{m},\ \frac{k\left(2 - \sqrt{\frac{5}{2} - \frac{\sqrt{5}}{2}}\right)}{m},\ \frac{k\left(2 - \sqrt{\frac{\sqrt{5}}{2} + \frac{5}{2}}\right)}{m},\ \frac{k\left(\frac{5}{2} - \frac{\sqrt{5}}{2}\right)}{m},\ \frac{k\left(\frac{\sqrt{5}}{2} + \frac{3}{2}\right)}{m},\ \frac{k\left(\frac{\sqrt{5}}{2} + \frac{5}{2}\right)}{m}, \ldots \right\}$$

$$\lambda_i, v_i = \left[\left(0,\ 1,\ \left[\begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \right] \right),\ \left(\frac{2k}{m},\ 1,\ \left[\begin{bmatrix} -1 \\ 1 \\ 1 \\ -1 \\ -1 \\ 1 \\ 1 \\ -1 \\ -1 \\ 1 \end{bmatrix} \right] \right),\ \left(\frac{k\left(\frac{3}{2} - \frac{\sqrt{5}}{2}\right)}{m},\ 1,\ \left[\begin{bmatrix} 1 \\ -\frac{1}{2} + \frac{\sqrt{5}}{2} \\ 0 \\ \frac{1}{2} - \frac{\sqrt{5}}{2} \\ -1 \\ -1 \\ \frac{1}{2} - \frac{\sqrt{5}}{2} \\ 0 \\ -\frac{1}{2} + \frac{\sqrt{5}}{2} \\ 1 \end{bmatrix} \right] \right), \ldots \right]$$

This verifies that $\lambda_1 = 0$ and its associated eigenvector is $v_1 = I$. Also, we can see that the Fiedler vector is the eigenvector associated with the smallest nonzero eigenvalue which is $\lambda_2 = \frac{2k}{m}$, and the vector is an equal mixture of -1 and 1 $v_2 = [-1, 1, 1, -1, -1, 1, 1, -1, -1, 1]$. Each eigenvalue corresponds to a different normal mode of the coupled oscillators. The first eigenvalue $\lambda_1 = 0$ corresponds to the transnational mode with frequency $\omega = 0$ where all the nodes, masses, move in the same direction. Whereas the second eigenvalue $\lambda_2 = \frac{2k}{m}$ corresponds to the first vibrational mode with frequency $\omega = \sqrt{\frac{2k}{m}}$ where half of the nodes move in the positive direction and the other half moves in the negative direction forming two different communities. This is illustrated in the eigenvectors of the system.

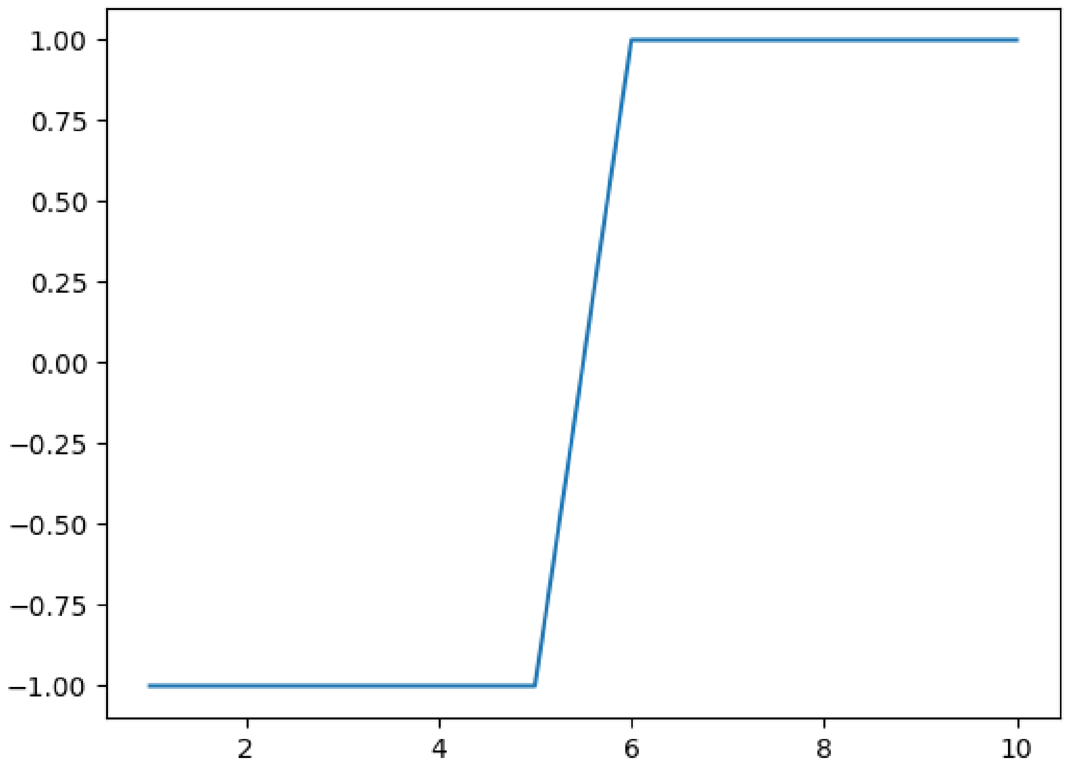

Figure 1: The Fiedler Vector Splitting into Two Communities

3) I generated an Erdös-Rényi random network and I found its adjacency matrix, Laplacian, the Fiedler vector, the nodes' centralities, clustering coefficient, and average path length.

```
def Random_Network(N):
    print("Random Network - Erdos-Renyi Construction for", N, "nodes")
    G = nx.erdos_renyi_graph(N, 0.3)
    nx.draw_spring(G, with_labels = True)
    A = nx.to_numpy_array(G)
    plt.show()
    print("This is the adjacency matrix:")
    print(A)
    print()

    d = list(nx.degree(G))
    D = np.zeros((N,N))
    for i in range(N):
        D[i][i] = d[i][1]
    print("This is the degree matrix:")
    print(D)
    print()

    L = D - A
    print("This is the Laplacian of the network:")
    print(L)
    print()

    if nx.is_connected(G) == True:
        v = nx.fiedler_vector(G)
        v.sort()
        print("This is the Fielder vector of the network:")
        print(v)
        plt.plot(np.arange(1, N+1), v)
        plt.show()
        print("This is the average path length:")
        l = nx.average_shortest_path_length(G)
        print(l)
        print()

    print("This is the betweenness centralities of the network:")
    C = nx.betweenness_centrality(G)
    print(C)
    print()

    print("This is the clustering coefficient of the network:")
    c = nx.clustering(G)
    print(c)
    print()
```

These are the graph and results that I got for 20 nodes.

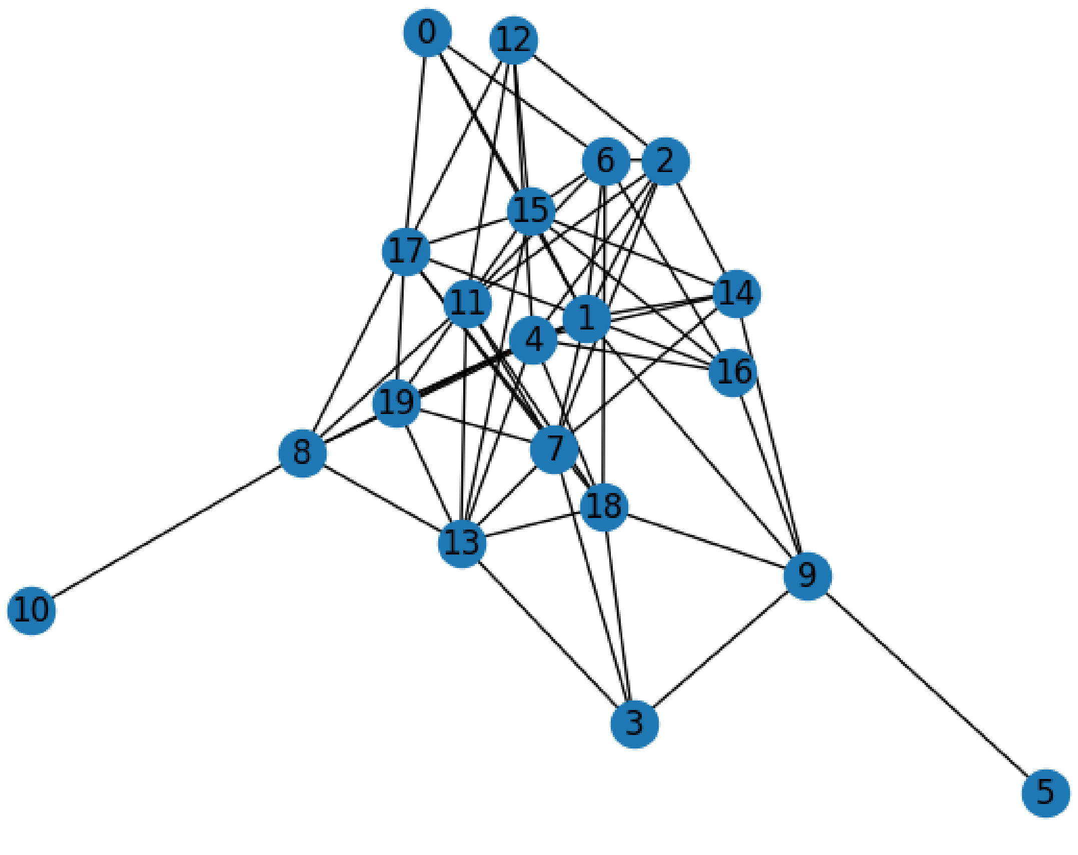

Figure 2: Random Network with 20 Nodes

```
This is the adjacency matrix:
[[0. 1. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 1. 0. 0.]
 [1. 0. 1. 0. 1. 0. 1. 1. 1. 1. 0. 0. 0. 0. 1. 1. 1. 1. 0. 1.]
 [0. 1. 0. 0. 1. 0. 1. 1. 0. 0. 0. 1. 1. 0. 1. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 1. 0. 1. 0. 0. 0. 1. 0. 0. 0. 0. 1. 0.]
 [0. 1. 1. 0. 0. 0. 0. 0. 1. 0. 0. 0. 1. 1. 1. 0. 1. 0. 1. 1.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [1. 1. 1. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 1. 1. 0. 1. 0.]
 [0. 1. 1. 1. 0. 0. 0. 0. 0. 0. 0. 1. 0. 1. 1. 0. 0. 1. 0. 1.]
 [0. 1. 0. 0. 1. 0. 0. 0. 0. 0. 1. 1. 0. 1. 0. 0. 0. 1. 0. 0.]
 [0. 1. 0. 1. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 1. 0. 1. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0. 0. 1. 1. 1. 0. 0. 0. 1. 1. 0. 0. 0. 0. 1. 0.]
 [0. 0. 1. 0. 1. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 1. 0. 1. 0. 0.]
 [0. 0. 0. 1. 1. 0. 0. 1. 1. 0. 0. 1. 0. 0. 0. 1. 0. 0. 1. 1.]
 [0. 1. 1. 0. 1. 0. 0. 1. 0. 1. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
 [1. 1. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 1. 1. 1. 0. 1. 1. 0. 1.]
 [0. 1. 0. 0. 1. 0. 1. 0. 0. 1. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
 [1. 1. 0. 0. 0. 0. 0. 1. 1. 0. 0. 0. 1. 0. 0. 1. 0. 0. 1. 1.]
 [0. 0. 0. 1. 1. 0. 1. 0. 0. 1. 0. 1. 0. 1. 0. 0. 0. 1. 0. 0.]
 [0. 1. 0. 0. 1. 0. 0. 1. 0. 0. 0. 0. 0. 1. 0. 1. 0. 1. 0. 0.]]

This is the degree matrix:
[[ 4.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0. 12.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  7.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  4.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  9.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  1.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  7.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  8.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  6.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  6.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  1.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  7.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  5.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  8.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  6.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  9.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  5.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  8.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  7.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  6.]]

This is the Laplacian of the network:
[[ 4. -1.  0.  0.  0.  0. -1.  0.  0.  0.  0.  0.  0.  0.  0. -1.  0. -1.  0.  0.]
 [-1. 12. -1.  0. -1.  0. -1. -1. -1. -1.  0.  0.  0.  0. -1. -1. -1. -1.  0. -1.]
 [ 0. -1.  7.  0. -1.  0. -1. -1.  0.  0.  0. -1. -1.  0. -1.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  4.  0.  0.  0. -1.  0. -1.  0.  0.  0. -1.  0.  0.  0.  0. -1.  0.]
 [ 0. -1. -1.  0.  9.  0.  0.  0. -1.  0.  0.  0. -1. -1. -1.  0. -1.  0. -1. -1.]
 [ 0.  0.  0.  0.  0.  1.  0.  0.  0. -1.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [-1. -1. -1.  0.  0.  0.  7.  0.  0.  0.  0. -1.  0.  0.  0. -1. -1.  0. -1.  0.]
 [ 0. -1. -1. -1.  0.  0.  0.  8.  0.  0.  0. -1.  0. -1. -1.  0.  0. -1.  0. -1.]
 [ 0. -1.  0.  0. -1.  0.  0.  0.  6.  0. -1. -1.  0. -1.  0.  0.  0. -1.  0.  0.]
 [ 0. -1.  0. -1.  0. -1.  0.  0.  0.  6.  0.  0.  0.  0. -1.  0. -1.  0. -1.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0. -1.  0.  1.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0. -1.  0.  0.  0. -1. -1. -1.  0.  0.  7. -1. -1.  0.  0.  0.  0. -1.  0.]
 [ 0.  0. -1.  0. -1.  0.  0.  0.  0.  0.  0. -1.  5.  0.  0. -1.  0. -1.  0.  0.]
 [ 0.  0.  0. -1. -1.  0.  0. -1. -1.  0.  0. -1.  0.  8.  0. -1.  0.  0. -1. -1.]
 [ 0. -1. -1.  0. -1.  0.  0. -1.  0. -1.  0.  0.  0.  0.  6. -1.  0.  0.  0.  0.]
 [-1. -1.  0.  0.  0.  0. -1.  0.  0.  0.  0.  0. -1. -1. -1.  9. -1. -1.  0. -1.]
 [ 0. -1.  0.  0. -1.  0. -1.  0.  0. -1.  0.  0.  0.  0.  0. -1.  5.  0.  0.  0.]
 [-1. -1.  0.  0.  0.  0.  0. -1. -1.  0.  0.  0. -1.  0.  0. -1.  0.  8. -1. -1.]
 [ 0.  0.  0. -1. -1.  0. -1.  0.  0. -1.  0. -1.  0. -1.  0.  0.  0. -1.  7.  0.]
 [ 0. -1.  0.  0. -1.  0.  0. -1.  0.  0.  0.  0.  0. -1.  0. -1.  0. -1.  0.  6.]]

This is the Fielder vector of the network:
[-0.6612082  -0.14621478 -0.03099813 -0.02711154 -0.02251598
-0.0220166 -0.01843407 -0.01601889 -0.01223839 -0.00985297
-0.00653824 -0.00589808 -0.00313882 -0.00263287  0.01628778
0.02182194  0.03009937  0.04522358  0.15779743  0.71358746]

This is the average path length:
1.8263157894736841

This is the betweenness centralities of the network:
{0: 0.0014619883040935672, 1: 0.17163742690058476, 2: 0.02199480181936322,
3: 0.011208576998050682, 4: 0.06972059779077322, 5: 0.0,
6: 0.034600389863547756, 7: 0.04571150097465887, 8: 0.11286549707602339,
9: 0.1257309941520468, 10: 0.0, 11: 0.03947368421052631,
12: 0.010136452241715398, 13: 0.05565302144249512, 14: 0.02426900584795321,
15: 0.05756985055230669, 16: 0.015107212475633526, 17: 0.05363872644574398,
18: 0.06052631578947367, 19: 0.006822612085769979}

This is the clustering coefficient of the network:
{0: 0.8333333333333334, 1: 0.36363636363636365, 2: 0.47619047619047616, 3: 0.5,
4: 0.2777777777777778, 5: 0, 6: 0.38095238095238093, 7: 0.35714285714285715,
8: 0.26666666666666666, 9: 0.2, 10: 0, 11: 0.3333333333333333,
12: 0.3, 13: 0.35714285714285715, 14: 0.4666666666666667, 15: 0.3333333333333333,
16: 0.5, 17: 0.32142857142857145, 18: 0.23809523809523808, 19: 0.6}
```

This is the Fiedler vector graph, you can clearly see that the network is split into two distinct communities.

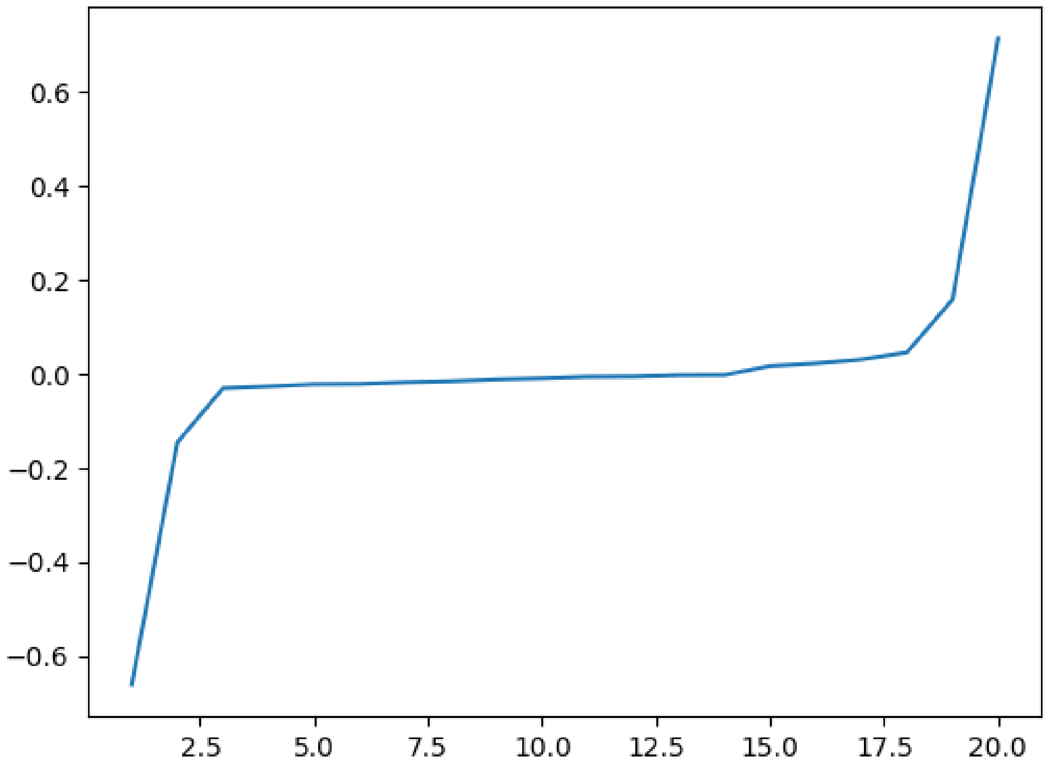

Figure 3: Fiedler Vector Graph of a Random Network of 20 Nodes

Then I did the same things for the Albert-Barabási scale-free network.

```
def Scale_Free(N):
    print("Scale-Free Network - Albert and Barabsi Construction of", N, "nodes")
    F = nx.barabasi_albert_graph(N, 5)
    nx.draw_spring(F, with_labels = True)
    A = nx.to_numpy_array(F)
    plt.show()
    print("This is the adjacency matrix:")
    print(A)
    print()

    d = list(nx.degree(F))
    D = np.zeros((N,N))
    for i in range(N):
        D[i][i] = d[i][1]
    print("This is the degree matrix:")
    print(D)
    print()

    L = D - A
    print("This is the Laplacian of the network:")
    print(L)
    print()

    if nx.is_connected(F) == True:
        v = nx.fiedler_vector(F)
        v.sort()
        print("This is the Fielder vector of the network:")
        print(v)
        plt.plot(np.arange(1, N+1), v)
        plt.show()

        print("This is the average path length:")
        l = nx.average_shortest_path_length(F)
        print(l)
        print()

    print("This is the betweenness centralities of the network:")
    C = nx.betweenness_centrality(F)
    print(C)
    print()

    print("This is the clustering coefficient of the network:")
    c = nx.clustering(F)
    print(c)
    print()
```

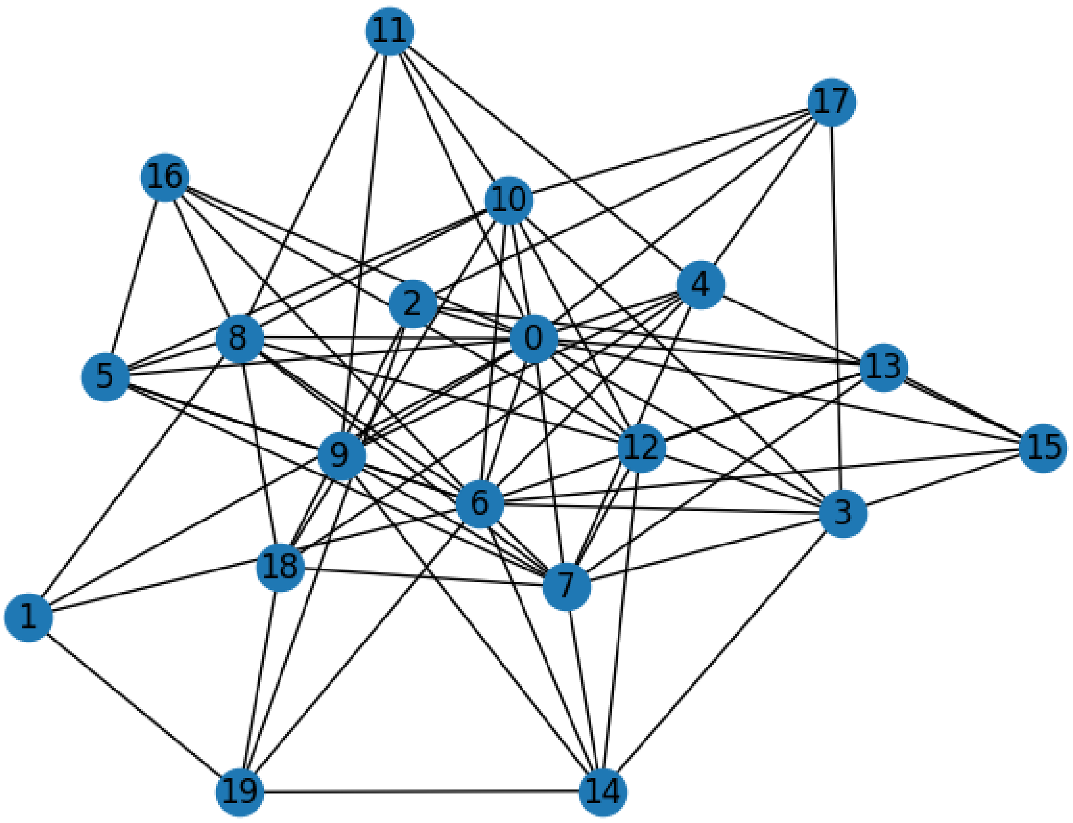

Figure 4: Scale-Free Network with 20 Nodes

```
This is the adjacency matrix:
[[0. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 0. 1. 1. 1. 0. 0.]
 [1. 0. 0. 0. 0. 0. 1. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
 [1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 1. 1. 1.]
 [1. 0. 0. 0. 0. 0. 1. 1. 0. 0. 1. 0. 1. 0. 1. 1. 0. 1. 0. 0.]
 [1. 0. 0. 0. 0. 0. 1. 1. 0. 1. 0. 1. 0. 0. 0. 1. 0. 1. 1. 0.]
 [1. 0. 0. 0. 0. 0. 1. 1. 1. 1. 1. 0. 0. 0. 0. 0. 1. 0. 0. 0.]
 [1. 1. 0. 1. 1. 1. 0. 1. 1. 1. 1. 0. 0. 1. 1. 1. 1. 0. 0. 1.]
 [1. 0. 0. 1. 1. 1. 1. 0. 1. 1. 0. 0. 1. 1. 1. 0. 0. 0. 1. 0.]
 [1. 1. 0. 0. 0. 1. 1. 1. 0. 0. 1. 1. 1. 0. 0. 0. 1. 0. 1. 0.]
 [1. 0. 0. 0. 1. 1. 1. 1. 0. 0. 0. 1. 0. 0. 1. 0. 0. 0. 0. 0.]
 [1. 0. 0. 1. 0. 1. 1. 0. 1. 0. 0. 1. 1. 0. 0. 0. 0. 1. 1. 0.]
 [1. 0. 0. 0. 1. 0. 0. 0. 1. 1. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [1. 0. 0. 1. 0. 0. 0. 1. 1. 0. 1. 0. 0. 1. 1. 0. 1. 0. 0. 0.]
 [1. 0. 1. 0. 0. 0. 1. 1. 0. 0. 0. 0. 1. 0. 0. 1. 0. 0. 0. 0.]
 [0. 0. 0. 1. 0. 0. 1. 1. 0. 1. 0. 0. 1. 0. 0. 0. 0. 0. 0. 1.]
 [1. 0. 0. 1. 1. 0. 1. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]
 [1. 0. 0. 0. 0. 1. 1. 0. 1. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]
 [1. 0. 1. 1. 1. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 1. 0. 1. 0. 0. 1. 1. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
 [0. 1. 1. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 1. 0.]]

This is the degree matrix:
[[16.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  4.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  5.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  8.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  8.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  7.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0. 14.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0. 11.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0. 10.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  7.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  9.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  5.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  8.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  6.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  6.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  5.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  5.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  5.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  6.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  5.]]

This is the Laplacian of the network:
[[16. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1.  0. -1. -1. -1.  0.  0.]
 [-1.  4.  0.  0.  0.  0. -1.  0. -1.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0. -1.]
 [-1.  0.  5.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0. -1.  0.  0.  0. -1. -1. -1.]
 [-1.  0.  0.  8.  0.  0. -1. -1.  0.  0. -1.  0. -1.  0. -1. -1.  0. -1.  0.  0.]
 [-1.  0.  0.  0.  8.  0. -1. -1.  0. -1.  0. -1.  0.  0.  0. -1.  0. -1. -1.  0.]
 [-1.  0.  0.  0.  0.  7. -1. -1. -1. -1. -1.  0.  0.  0.  0.  0. -1.  0.  0.  0.]
 [-1. -1.  0. -1. -1. -1. 14. -1. -1. -1. -1.  0.  0. -1. -1. -1. -1.  0.  0. -1.]
 [-1.  0.  0. -1. -1. -1. -1. 11. -1. -1.  0.  0. -1. -1. -1.  0.  0.  0. -1.  0.]
 [-1. -1.  0.  0.  0. -1. -1. -1. 10.  0. -1. -1. -1.  0.  0.  0. -1.  0. -1.  0.]
 [-1.  0.  0.  0. -1. -1. -1. -1.  0.  7.  0. -1.  0.  0. -1.  0.  0.  0.  0.  0.]
 [-1.  0.  0. -1.  0. -1. -1.  0. -1.  0.  9. -1. -1.  0.  0.  0.  0. -1. -1.  0.]
 [-1.  0.  0.  0. -1.  0.  0.  0. -1. -1. -1.  5.  0.  0.  0.  0.  0.  0.  0.  0.]
 [-1.  0.  0. -1.  0.  0.  0. -1. -1.  0. -1.  0.  8. -1. -1.  0. -1.  0.  0.  0.]
 [-1.  0. -1.  0.  0.  0. -1. -1.  0.  0.  0.  0. -1.  6.  0. -1.  0.  0.  0.  0.]
 [ 0.  0.  0. -1.  0.  0. -1. -1.  0. -1.  0.  0. -1.  0.  6.  0.  0.  0.  0. -1.]
 [-1.  0.  0. -1. -1.  0. -1.  0.  0.  0.  0.  0.  0. -1.  0.  5.  0.  0.  0.  0.]
 [-1.  0.  0.  0.  0. -1. -1.  0. -1.  0.  0.  0. -1.  0.  0.  0.  5.  0.  0.  0.]
 [-1.  0. -1. -1. -1.  0.  0.  0.  0.  0. -1.  0.  0.  0.  0.  0.  0.  5.  0.  0.]
 [ 0.  0. -1.  0. -1.  0.  0. -1. -1.  0. -1.  0.  0.  0.  0.  0.  0.  0.  6. -1.]
 [ 0. -1. -1.  0.  0.  0. -1.  0.  0.  0.  0.  0.  0.  0. -1.  0.  0.  0. -1.  5.]]

This is the Fielder vector of the network:
[-0.25386772 -0.21896678 -0.17455678 -0.16966309 -0.16734418
-0.12333848 -0.11089381 -0.10601751 -0.09823481 -0.07593571
-0.0572971  -0.04528459 -0.02838385 -0.01471877 -0.00844505
0.02623561  0.20431798  0.35643013  0.50030848  0.56565604]
This is the average path length:
1.6105263157894736

This is the betweenness centralities of the network:
{0: 0.18499721526037313, 1: 0.004873294346978557, 2: 0.02017543859649123,
3: 0.025835421888053465, 4: 0.03489278752436647, 5: 0.007755499860763018,
6: 0.12997772208298525, 7: 0.050730994152046786, 8: 0.04610832637148427,
9: 0.017446393762183234, 10: 0.037280701754385956, 11: 0.004970760233918128,
12: 0.02276524644945697, 13: 0.013596491228070173, 14: 0.01827485380116959,
15: 0.003801169590643275, 16: 0.0019005847953216374, 17: 0.009161793372319687,
18: 0.02315510999721526, 19: 0.02066276803118908}
```

```
This is the clustering coefficient of the network:
{0: 0.36666666666666664, 1: 0.6666666666666666, 2: 0.3, 3: 0.5357142857142857,
4: 0.42857142857142855, 5: 0.7142857142857143, 6: 0.37362637362637363,
7: 0.4727272727272727, 8: 0.4888888888888889, 9: 0.6190476190476191,
10: 0.4444444444444444, 11: 0.6, 12: 0.5,
13: 0.5333333333333333, 14: 0.5333333333333333, 15: 0.7,
16: 0.8, 17: 0.5, 18: 0.26666666666666666, 19: 0.3}
```

This is the Fiedler graph. It is harder to infer the communities from this graph, or even know if the network splits into communities or no. Probably it's because I used a small number of nodes in this simulation.

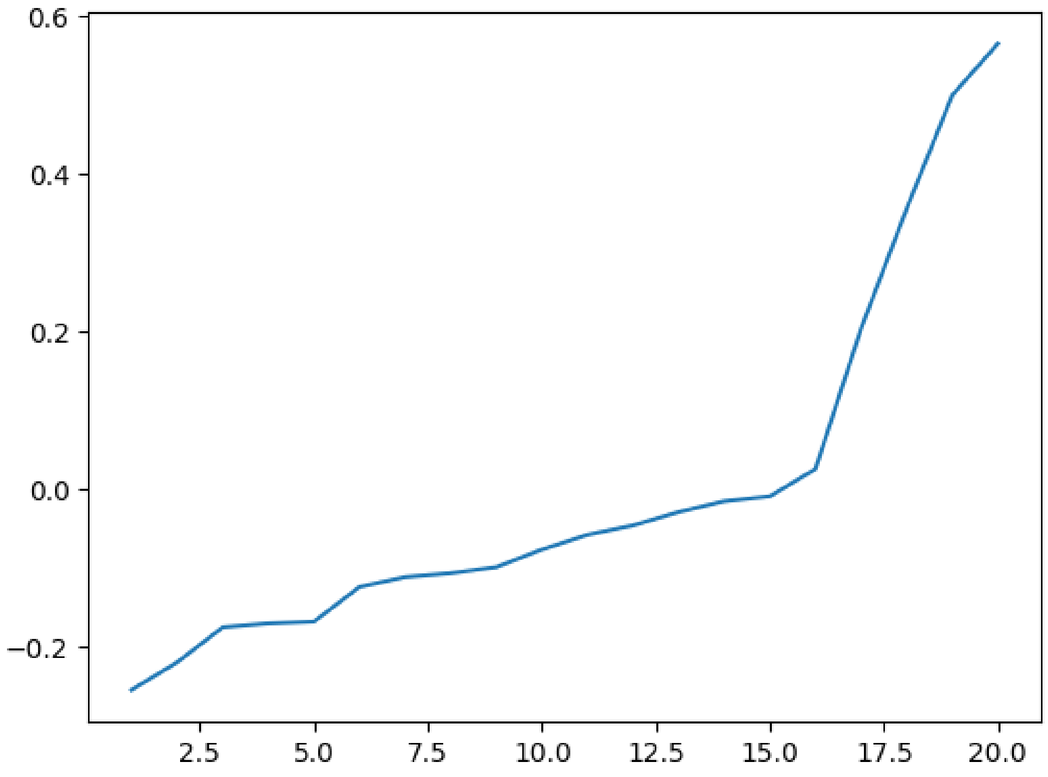

Figure 5: Fiedler Vector Graph for Scale-Free Network of 20 Nodes

Lastly, I applied all of this again on a Watts-Strogatz small-world network.

```python
def Small_World(N):
    print("Small-World Network - Watts and Strogatz's Construction with", N, "nodes")
    W = nx.watts_strogatz_graph(N, 10, 0.5)
    nx.draw_spring(W, with_labels = True)
    A = nx.to_numpy_array(W)
    plt.show()
    print("This is the adjacency matrix:")
    print(A)
    print()

    d = list(nx.degree(W))
    D = np.zeros((N,N))
    for i in range(N):
        D[i][i] = d[i][1]
    print("This is the degree matrix:")
    print(D)
    print()

    L = D - A
```

```
print("This is the Laplacian of the network:")
print(L)
print()

if nx.is_connected(W) == True:
    v = nx.fiedler_vector(W)
    v.sort()
    print("This is the Fielder vector of the network:")
    print(v)
    plt.plot(np.arange(1, N+1), v)
    plt.show()

    print("This is the average path length:")
    l = nx.average_shortest_path_length(W)
    print(l)
    print()

print("This is the betweenness centralities of the network:")
C = nx.betweenness_centrality(W)
print(C)
print()

print("This is the clustering coefficient of the network:")
c = nx.clustering(W)
print(c)
print()
```

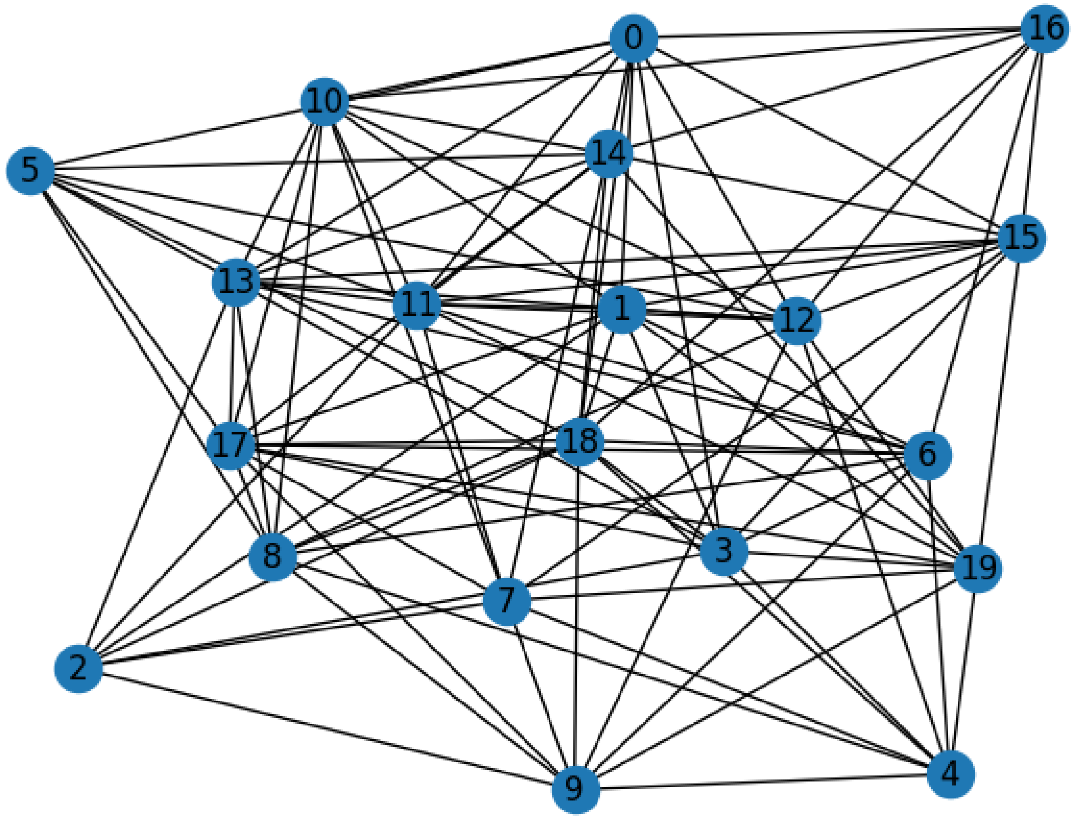

Figure 6: Small-World Network with 20 Nodes

```
This is the adjacency matrix:
[[0. 1. 0. 1. 0. 1. 0. 0. 0. 0. 1. 1. 1. 1. 1. 1. 1. 0. 1. 0.]
 [1. 0. 1. 1. 0. 0. 1. 0. 0. 0. 1. 1. 1. 0. 0. 1. 0. 1. 1. 1.]
 [0. 1. 0. 1. 0. 0. 0. 1. 0. 1. 0. 1. 0. 1. 0. 0. 0. 0. 1. 0.]
 [1. 1. 1. 0. 1. 0. 1. 0. 0. 0. 0. 0. 0. 1. 0. 1. 0. 1. 1. 1.]
 [0. 0. 0. 1. 0. 0. 1. 1. 1. 1. 0. 0. 1. 0. 0. 0. 0. 0. 1. 1.]
 [1. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 1. 1. 1. 1. 0. 0. 1. 1. 0.]
 [0. 1. 0. 1. 1. 0. 0. 0. 1. 1. 0. 1. 0. 1. 0. 0. 1. 1. 1. 0.]
 [0. 0. 1. 0. 1. 0. 0. 0. 0. 1. 1. 1. 0. 0. 1. 1. 0. 1. 0. 1.]
 [0. 0. 0. 0. 1. 1. 1. 0. 0. 1. 1. 0. 1. 1. 0. 0. 0. 1. 1. 0.]
 [0. 0. 1. 0. 1. 0. 1. 1. 1. 0. 0. 0. 1. 0. 0. 0. 0. 1. 1. 1.]
 [1. 1. 0. 0. 0. 0. 0. 1. 1. 0. 0. 1. 1. 1. 1. 0. 1. 1. 0. 0.]
 [1. 1. 1. 0. 0. 1. 1. 1. 0. 0. 1. 0. 1. 1. 1. 1. 0. 0. 0. 1.]
 [1. 1. 0. 0. 1. 1. 0. 0. 1. 1. 1. 1. 0. 1. 0. 1. 1. 0. 0. 1.]
 [1. 0. 1. 1. 0. 1. 1. 0. 1. 0. 1. 1. 1. 0. 1. 1. 0. 1. 0. 0.]
 [1. 0. 0. 0. 0. 1. 0. 1. 0. 0. 1. 1. 0. 1. 0. 1. 1. 1. 1. 1.]
 [1. 1. 0. 1. 0. 0. 0. 1. 0. 0. 0. 1. 1. 1. 1. 0. 1. 0. 0. 1.]
 [1. 0. 0. 0. 0. 0. 1. 0. 0. 0. 1. 0. 1. 0. 1. 1. 0. 0. 1. 0.]
 [0. 1. 0. 1. 0. 1. 1. 1. 1. 1. 1. 0. 0. 1. 1. 0. 0. 0. 1. 1.]
 [1. 1. 1. 1. 1. 1. 1. 0. 1. 1. 0. 0. 0. 0. 1. 0. 1. 1. 0. 0.]
 [0. 1. 0. 1. 1. 0. 0. 1. 0. 1. 0. 1. 1. 0. 1. 1. 0. 1. 0. 0.]]

This is the degree matrix:
[[11.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0. 11.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  7.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0. 10.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  8.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  8.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0. 10.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  9.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  9.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  9.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0. 10.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0. 12.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0. 12.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0. 12.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0. 11.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0. 10.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  7.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0. 12.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0. 12.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0. 0. 10.]]

This is the Laplacian of the network:
[[11. -1.  0. -1.  0. -1.  0.  0.  0.  0. -1. -1. -1. -1. -1. -1. -1.  0.  -1.  0.]
 [-1. 11. -1. -1.  0.  0. -1.  0.  0.  0. -1. -1. -1.  0.  0. -1.  0. -1.  -1. -1.]
 [ 0. -1.  7. -1.  0.  0.  0. -1.  0. -1.  0. -1.  0. -1.  0.  0.  0.  0.  -1.  0.]
 [-1. -1. -1. 10. -1.  0. -1.  0.  0.  0.  0.  0.  0. -1.  0. -1.  0. -1.  -1. -1.]
 [ 0.  0.  0. -1.  8.  0. -1. -1. -1. -1.  0.  0. -1.  0.  0.  0.  0.  0.  -1. -1.]
 [-1.  0.  0.  0.  0.  8.  0.  0. -1.  0.  0. -1. -1. -1. -1.  0.  0. -1.  -1.  0.]
 [ 0. -1.  0. -1. -1.  0. 10.  0. -1. -1.  0. -1.  0. -1.  0.  0. -1. -1.  -1.  0.]
 [ 0.  0. -1.  0. -1.  0.  0.  9.  0. -1. -1. -1.  0.  0. -1. -1.  0. -1.   0. -1.]
 [ 0.  0.  0.  0. -1. -1. -1.  0.  9. -1. -1.  0. -1. -1.  0.  0.  0. -1.  -1.  0.]
 [ 0.  0. -1.  0. -1.  0. -1. -1. -1.  9.  0.  0. -1.  0.  0.  0.  0. -1.  -1. -1.]
 [-1. -1.  0.  0.  0.  0.  0. -1. -1.  0. 10. -1. -1. -1. -1.  0. -1. -1.   0.  0.]
 [-1. -1. -1.  0.  0. -1. -1. -1.  0.  0. -1. 12. -1. -1. -1. -1.  0.  0.   0. -1.]
 [-1. -1.  0.  0. -1. -1.  0.  0. -1. -1. -1. -1. 12. -1.  0. -1. -1.  0.   0. -1.]
 [-1.  0. -1. -1.  0. -1. -1.  0. -1.  0. -1. -1. -1. 12. -1. -1.  0. -1.   0.  0.]
 [-1.  0.  0.  0.  0. -1.  0. -1.  0.  0. -1. -1.  0. -1. 11. -1. -1. -1.  -1. -1.]
 [-1. -1.  0. -1.  0.  0.  0. -1.  0.  0.  0. -1. -1. -1. -1. 10. -1.  0.   0. -1.]
 [-1.  0.  0.  0.  0.  0. -1.  0.  0.  0. -1.  0. -1.  0. -1. -1.  7.  0.  -1.  0.]
 [ 0. -1.  0. -1.  0. -1. -1. -1. -1. -1. -1.  0.  0. -1. -1.  0.  0. 12.  -1. -1.]
 [-1. -1. -1. -1. -1. -1. -1.  0. -1. -1.  0.  0.  0.  0. -1.  0. -1. -1.  12.  0.]
 [ 0. -1.  0. -1. -1.  0.  0. -1.  0. -1.  0. -1. -1.  0. -1. -1.  0. -1.   0. 10.]]

This is the Fielder vector of the network:
[-0.53912347 -0.25017065 -0.20161295 -0.19736308 -0.19707415
-0.1759899  -0.08648019 -0.06936522 -0.06431446  0.0025416
 0.03915323  0.04983783  0.0781307   0.09551346  0.12034997
  0.12061163  0.18029432  0.35143961 0.36347563  0.38014609]
This is the average path length:
1.4736842105263157

This is the betweenness centralities of the network:
{0: 0.022363779819920167, 1: 0.027568922305764406, 2: 0.011767845539775366,
3: 0.024201707973637798, 4: 0.014438874965190751, 5: 0.009182678919521025,
6: 0.02785667873387171, 7: 0.02481203007518797, 8: 0.017481203007518795,
9: 0.019813422445001387, 10: 0.02294161329249048, 11: 0.0363919056901513,
12: 0.051046597976422525, 13: 0.03902116402116401, 14: 0.028815093288777495,
15: 0.020790401930752808, 16: 0.009565580618212196, 17: 0.03940638633621089,
18: 0.05543024227234753, 19: 0.023419660261765524}

This is the clustering coefficient of the network:
{0: 0.6, 1: 0.509090909090909, 2: 0.42857142857142855, 3: 0.5111111111111111,
4: 0.5357142857142857, 5: 0.6428571428571429, 6: 0.4666666666666667,
7: 0.4722222222222222, 8: 0.5833333333333334, 9: 0.5277777777777778,
10: 0.5333333333333333, 11: 0.5151515151515151, 12: 0.45454545454545453,
13: 0.5, 14: 0.5454545454545454, 15: 0.5777777777777777, 16: 0.5238095238095238,
17: 0.48484848484848486, 18: 0.4393939393939394, 19: 0.5333333333333333}
```

This is the Fiedler graph. We can see that the network splits into two communities roughly.

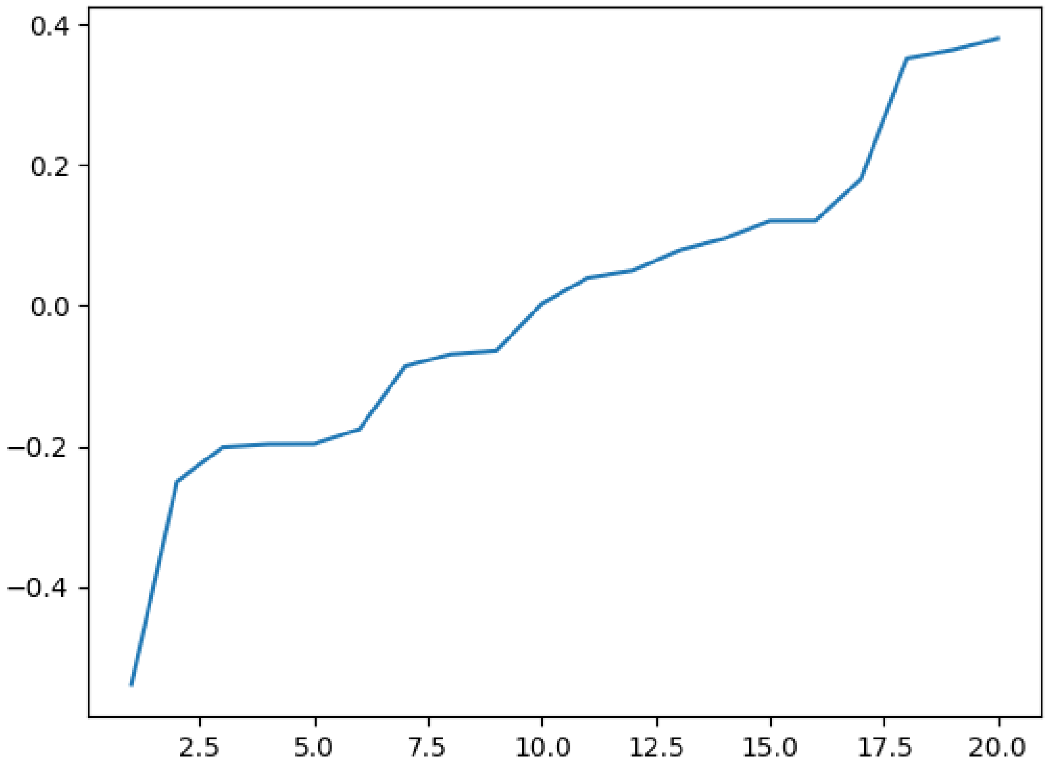

Figure 7: Fiedler Vector Graph for Small-World Network of 20 Nodes

Lastly, I created an algorithm to remove nodes iterative based on their degree, their centrality, and randomly.

```
def remove_degree(G, pos):
    d = dict(nx.degree(G))
    highd = max(d, key = d.get)
    G.remove_node(highd)
    nx.draw(G, with_labels = True, pos = pos)
    plt.show()

def remove_centerality(G, pos):
    c = dict(nx.betweenness_centrality(G))
    highc = max(c, key = c.get)
    G.remove_node(highc)
    nx.draw(G, with_labels = True, pos = pos)
    plt.show()

def remove_randomly(G, pos):
    n = list(G.nodes)
    randn = rand.choice(n)
    G.remove_node(randn)
    nx.draw(G, with_labels = True, pos = pos)
    plt.show()
```

I applied those three methods on a scale-free network with 100 nodes. I created three animations titled "Centerality Remover", "Degree Remover", "Random Remover" in the folder which shows how the network is evolving under the respective removal covering the same interval of time. As can be seen, the fastest way to lose connectivity is the removal of the nodes with the highest degree, the most connections. The second fastest is the removal of the nodes with highest betweenness centrality. Lastly, the worst way to lose connectivity is removing nodes randomly. So, in case of a spread, the most efficient way to prevent further spreading is to remove the nodes with large number of connections and links, hence the highest degrees.

2 Tight-Binding

Given that the tight-binding Hamiltonian can be expressed as a function of the adjacency matrix $\hat{H} = \hat{\alpha}I + \hat{\beta}A$, so the energy levels of the system is the eigenvalues of the adjacency matrix $E_j = \hat{\alpha} + \beta\hat{\lambda}_j$. For the conditions $\hat{\alpha} = 0$, $\hat{\beta} = -1$, and $k_bT = 1$, I generated different networks and I calculated the statistical mechanical properties of the system, partition function, entropy, Helmholtz free energy, and Gibbs free energy, where the properties are given by,

$$\begin{aligned} Z &= tr(exp\beta\hat{H}) \\ S &= -p_j log p_j \\ F &= -k_bTlog(Z) \\ G &= N\left(F(N+1) - F(N)\right) \end{aligned}$$

First, I generated the Watts Strogatz Small World network with $n = 12$, $k = 3$, and $p = 0.1$. Then, I found its eigenvalues, degree distribution, degree distribution probability, partition function, entropy, Helmholtz free energy, and Gibbs free energy. Then I varied the probabilities of the network and plotted the statistical characteristics as a function of probability.

```
n = 12
k = 3
p = 0.1
G = nx.watts_strogatz_graph(n, k, p)
pos = nx.spring_layout(G)
nx.draw(G, with_labels = True, pos = pos)
plt.show()
l = nx.to_numpy_array(G)
eig = np.linalg.eigvals(l)
print("Eigenvalues of the Network:")
print(eig)
deg_dis = nx.degree_histogram(G)
print("Degree Distribution:", deg_dis)

Z = np.trace(np.exp(((-1) * l)))
print("Partition Function:", Z)

prob = np.divide(deg_dis, n)
print("Probabilities of each degree:", prob)

S = - np.sum(np.fromiter((p * np.log(p) for p in prob if p != 0), dtype = float))
print("Entropy:", S)

F = - np.log(Z)
print("Helmholtz Free Energy:", F)

g = nx.watts_strogatz_graph(n+1, k, p)
l1 = nx.to_numpy_array(g)
Z1 = np.trace(np.exp(((-1) * l1)))
F1 = - np.log(Z1)
print("Helmholtz Free Energy:", F1)
G = n * (F - F1)
print("Gibbs Free Energy:", G)
```

This is the result for this network,

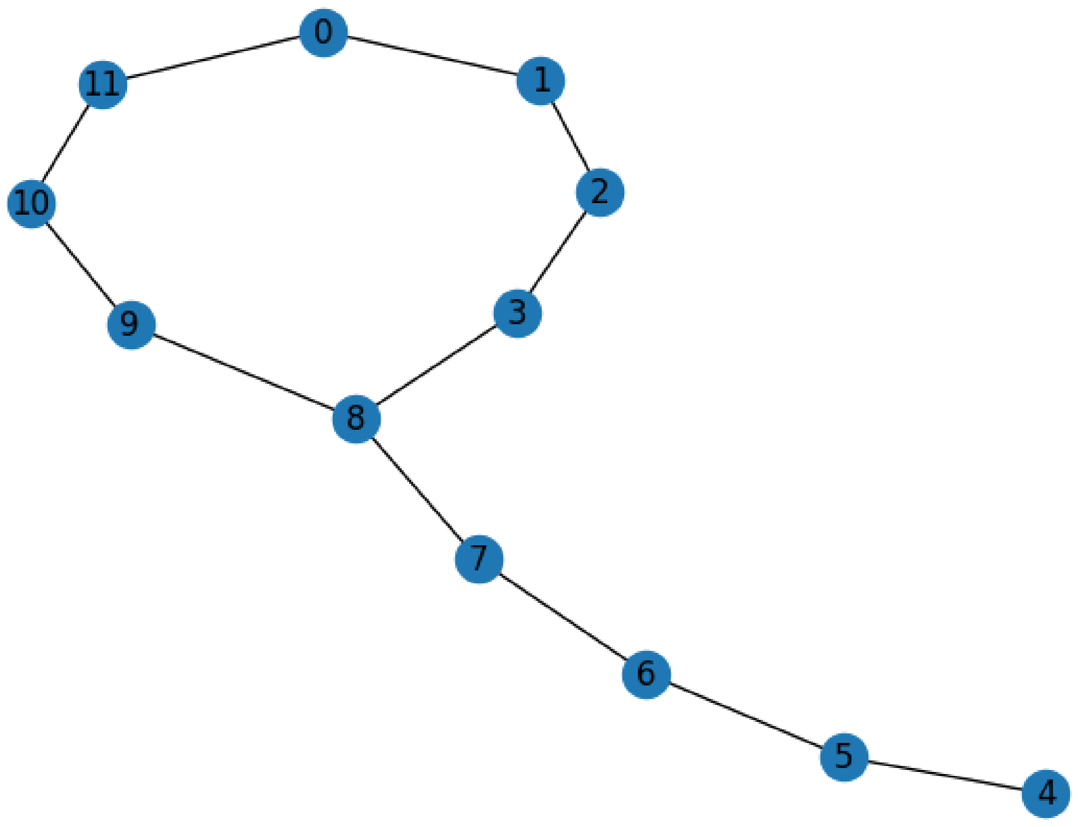

Figure 8: Small World Network with n = 12, k = 3, and p = 0.1

```
Eigenvalues of the Network:
[-2.13577921e+00+0.00000000e+00j -1.73205081e+00+0.00000000e+00j
 -1.41421356e+00+0.00000000e+00j -6.62153447e-01+0.00000000e+00j
  2.13577921e+00+0.00000000e+00j  6.62153447e-01+0.00000000e+00j
  1.73205081e+00+0.00000000e+00j  1.41421356e+00+0.00000000e+00j
  7.53421539e-18+1.49373945e-17j  7.53421539e-18-1.49373945e-17j
 -1.41421356e+00+0.00000000e+00j  1.41421356e+00+0.00000000e+00j]
Degree distribution: [0, 1, 10, 1]
Partition Function: 12.0
Probabilities of each degree: [0.         0.08333333 0.83333333 0.08333333]
Entropy: 0.5660857389596289
Helmholtz Free Energy: -2.4849066497880004
Gibbs Free Energy: 0.9605124920824366
```

Then, I varied p and plotted Z, S, F, and G as a function of p.

```
N = 10
z = np.zeros(N)
S = np.zeros(N)
F = np.zeros(N)
G = np.zeros(N)
p = np.zeros(N)
for i in range(1, N + 1):
    p[i-1] = i/N
    Z[i-1], S[i-1], F[i-1], G[i-1] = small_world_stat(n = 12, k = 3, p = i/N)
plt.plot(p, Z, label = "Z")
plt.title("Partition Function")
plt.show()
plt.plot(p, S, label = 'S')
plt.title("Entropy")
plt.show()
plt.plot(p, F, label = 'F')
plt.title("Helmoholtz Free Energy")
plt.show()
plt.plot(p, G, label = 'G')
plt.title("Gibbs Free Energy")
plt.show()
```

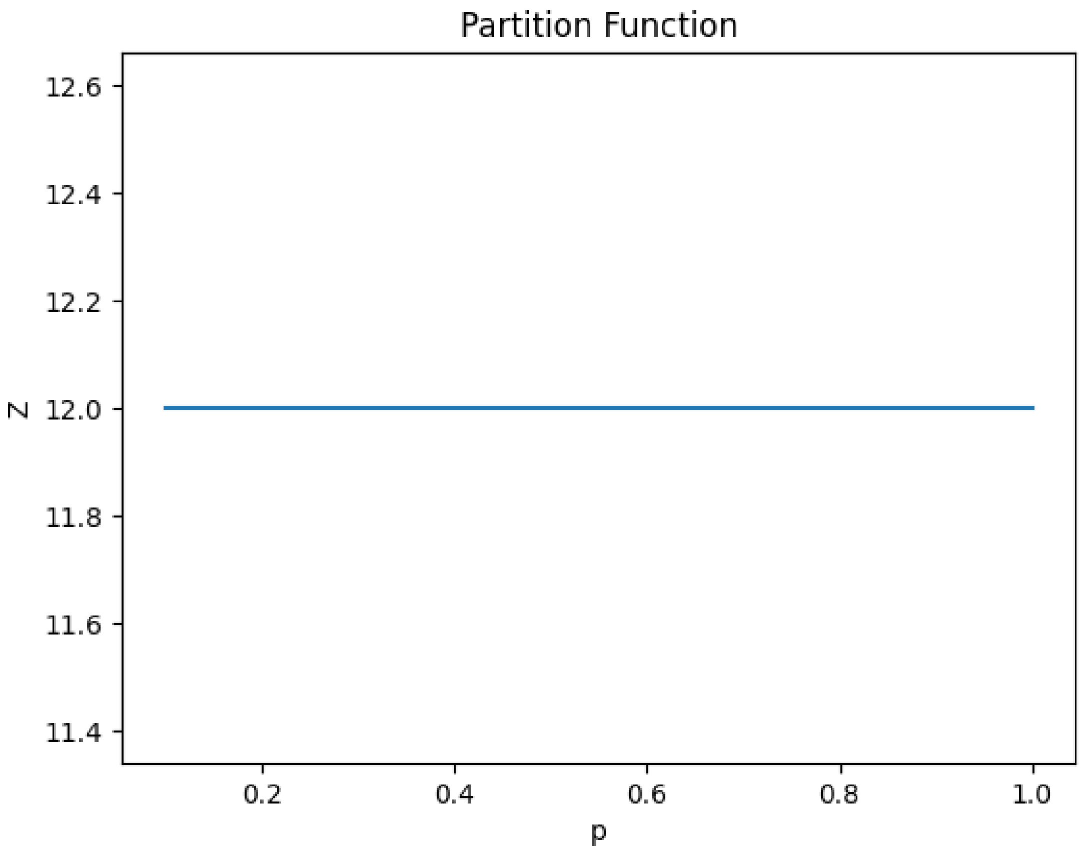

Figure 9: Partition Function as a Function of Probability

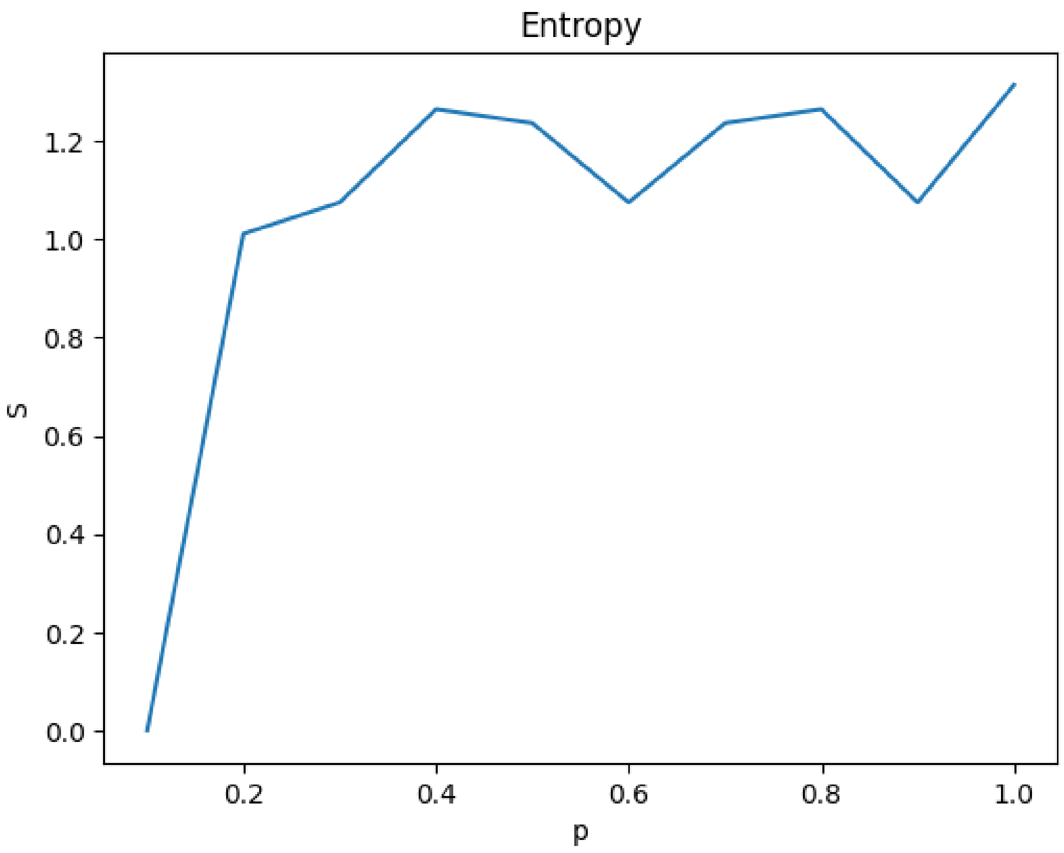

Figure 10: Entropy as a Function of Probability

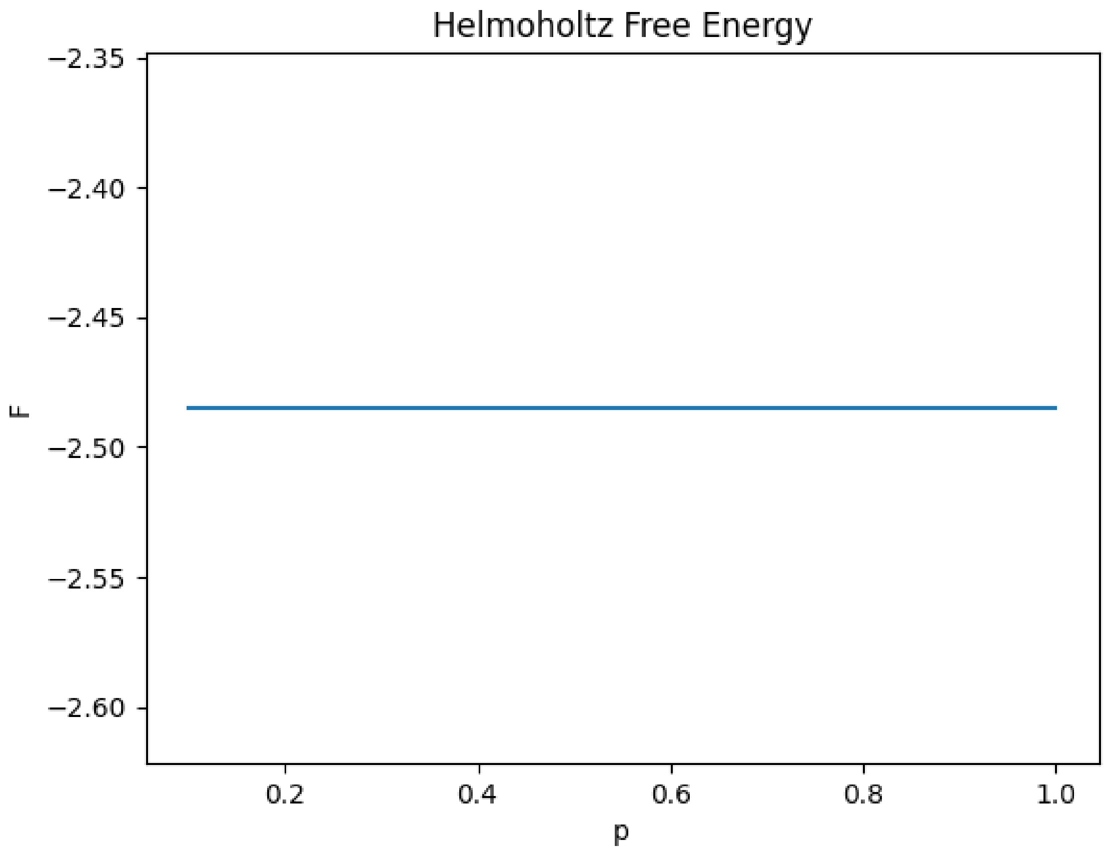

Figure 11: Helmholtz Free Energy as a Function of Probability

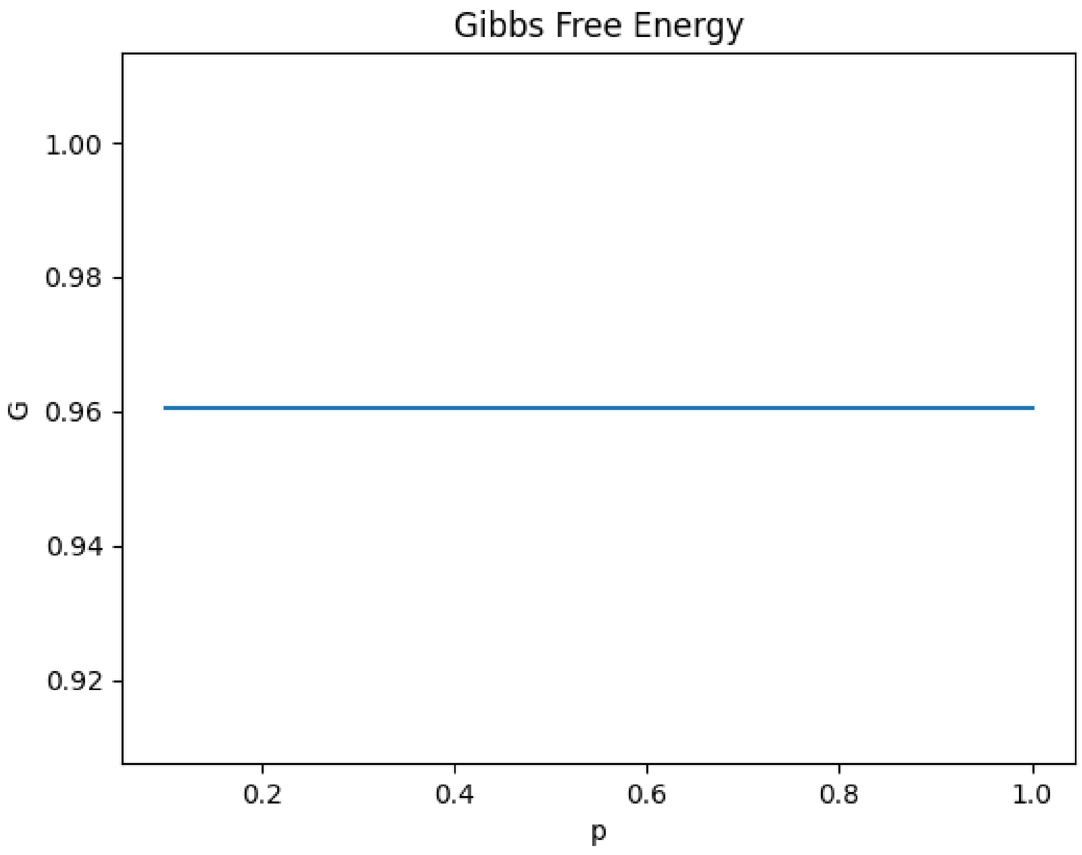

Figure 12: Gibbs Free Energy as a Function of Probability

From what can be observed, the only thing that is changing as the probability is changing is the entropy which coincides with statistical mechanics.
I applied the same thing again on a Erdös-Rényi random network.

```
n = 12
p = 0.1
G = nx.erdos_renyi_graph(n, p)
pos = nx.spring_layout(G)
nx.draw(G, with_labels = True, pos = pos)
plt.show()

l = nx.to_numpy_array(G)
eig = np.linalg.eigvals(l)
print("Eigenvalues of the Network:")
print(eig)

deg_dis = nx.degree_histogram(G)
print("Degree distribution:", deg_dis)

Z = np.trace(np.exp(((-1) * l)))
print("Partition Function:", Z)

prob = np.divide(deg_dis, n)
print("Probabilities of each degree:", prob)

S = - np.sum(np.fromiter((p * np.log(p) for p in prob if p != 0), dtype = float))
print("Entropy:", S)

F = - np.log(Z)
print("Helmholtz Free Energy:", F)

g = nx.erdos_renyi_graph(n+1, p)


l1 = nx.to_numpy_array(g)
eig1 = np.linalg.eigvals(l1)

Z1 = np.trace(np.exp(((-1) * l1)))

F1 = - np.log(Z1)

G = n * (F - F1)
print("Gibbs Free Energy:", G)
```

This is what I got,

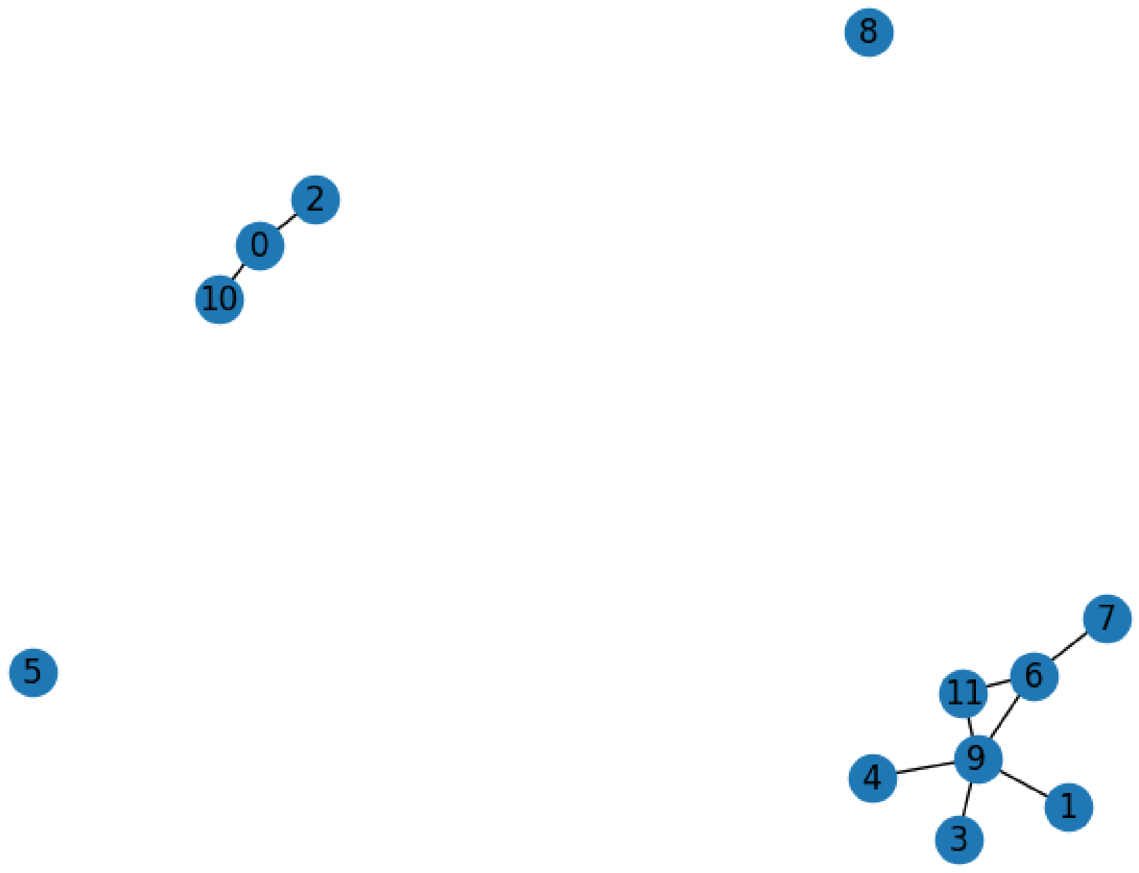

Figure 13: Erdös Rényi Random Graph n = 12 and p = 0.1

```
Eigenvalues of the Network:
[ 1.41421356e+00 -1.41421356e+00  2.59440037e+00  9.15882801e-01
  1.30933376e-16 -2.12202128e+00 -1.38826190e+00  8.85313927e-17
 -4.41555067e-17 -3.09527021e-18  0.00000000e+00  0.00000000e+00]
Degree distribution: [2, 6, 2, 1, 0, 1]
Partition Function: 12.0
Probabilities of each degree: [0.16666667 0.5        0.16666667 0.08333333
0.         0.08333333]
Entropy: 1.3579778549873245
Helmholtz Free Energy: -2.4849066497880004
Gibbs Free Energy: 0.9605124920824366
```

Then, I varied the probabilities and plotted them.

```
N = 10
z = np.zeros(N)
S = np.zeros(N)
F = np.zeros(N)
G = np.zeros(N)
p = np.zeros(N)
for i in range(1, N + 1):
    p[i-1] = i/N
    result = random_stat(n=12, p=i/N)
    Z[i-1], S[i-1], F[i-1], G[i-1] = result
plt.plot(p, Z, label = "Z")
plt.title("Partition Function")
plt.xlabel('p')
plt.ylabel('Z')
plt.show()
plt.plot(p, S, label = 'S')
plt.title("Entropy")
plt.xlabel('p')
plt.ylabel('S')
plt.show()
plt.plot(p, F, label = 'F')
plt.title("Helmoholtz Free Energy")
plt.xlabel('p')
plt.ylabel('F')
plt.show()
plt.plot(p, G, label = 'G')
plt.title("Gibbs Free Energy")
plt.xlabel('p')
plt.ylabel('G')
plt.show()
```

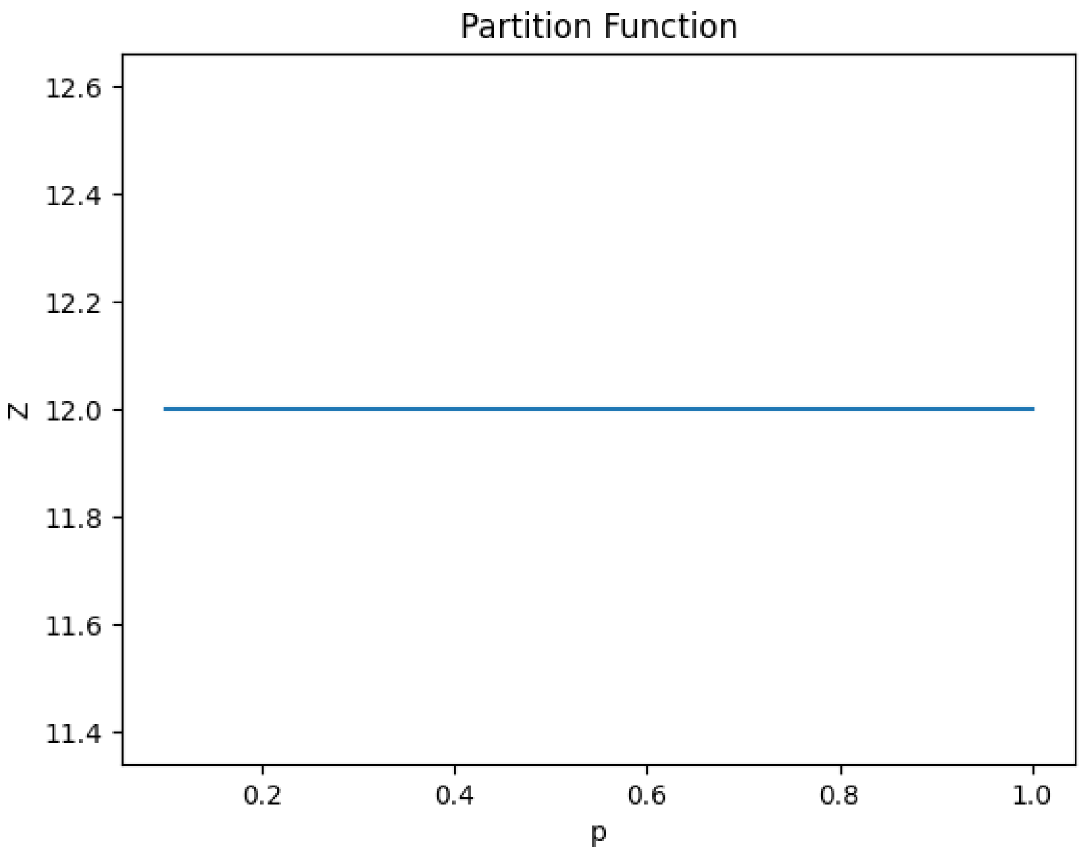

Figure 14: Partition Function as a Function of Probability

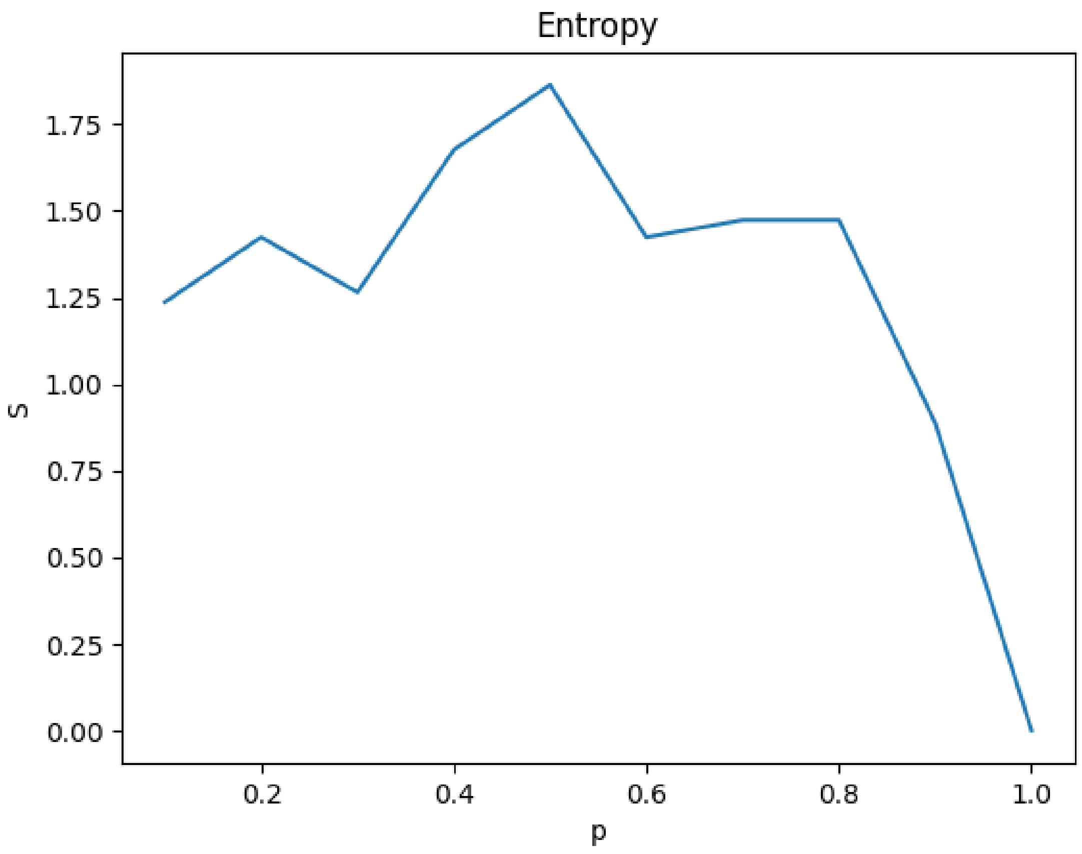

Figure 15: Entropy as a Function of Probability

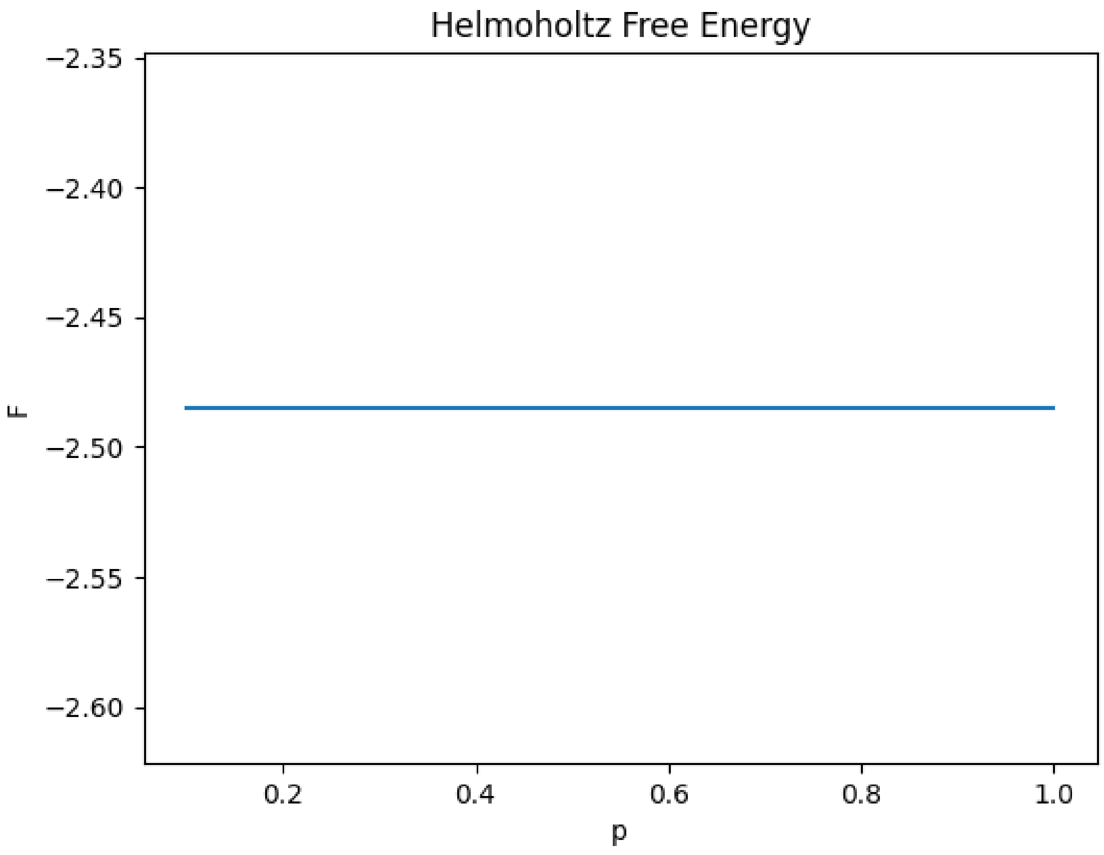

Figure 16: Helmholtz Free Energy as a Function of Probability

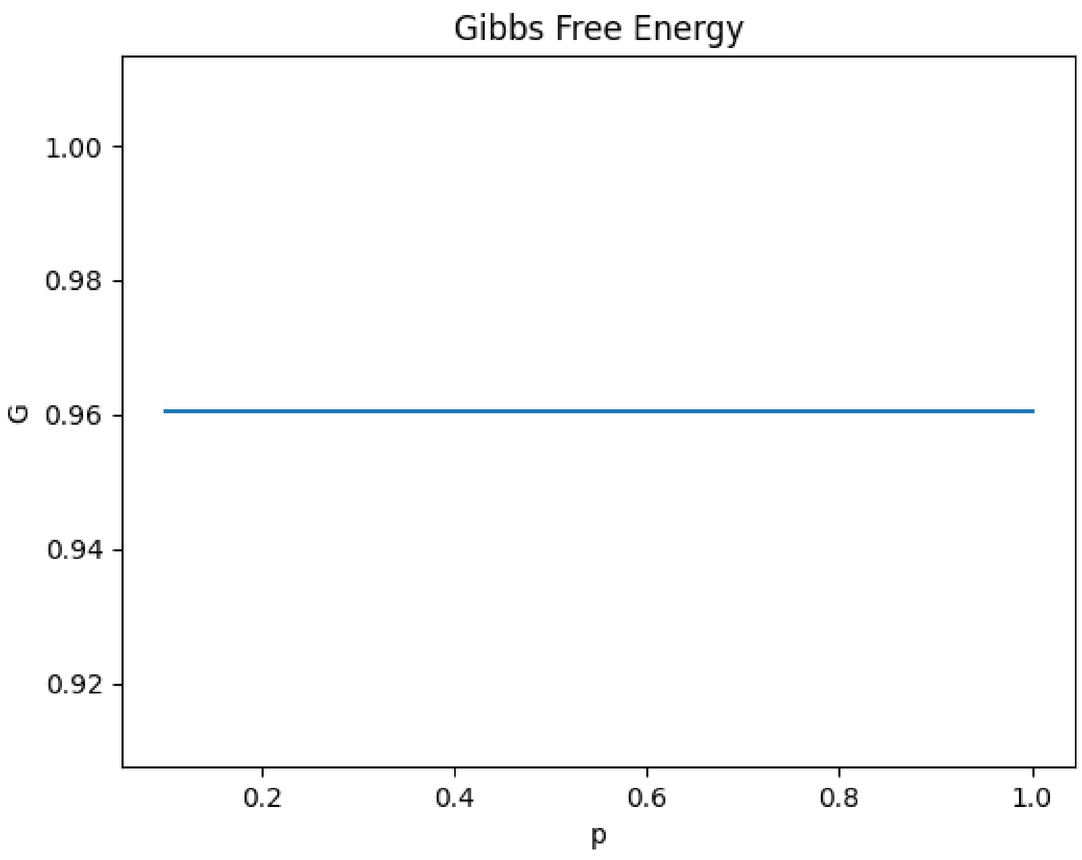

Figure 17: Gibbs Free Energy as a Function of Probability

I obtain similar behavior to the small world network, so I have the same conclusion.
Lastly, I found the statistical variables in a regular network of $d = 3$ and $n = 12$.

```
d = 3
n = 12
G = nx.random_regular_graph(d, n)
pos = nx.spring_layout(G)
nx.draw(G, with_labels = True, pos = pos)
plt.show()

l = nx.to_numpy_array(G)
eig = np.linalg.eigvals(l)
print("Eigenvalues of the Network:")
print(eig)

deg_dis = nx.degree_histogram(G)
print("Degree distribution:", deg_dis)

Z = np.trace(np.exp(((-1) * l)))
print("Partition Function:", Z)

prob = np.divide(deg_dis, n)
print("Probabilities of each degree:", prob)

S = - np.sum(np.fromiter((p * np.log(p) for p in prob if p != 0), dtype = float))
print("Entropy:", S)

F = - np.log(Z)
print("Helmholtz Free Energy:", F)

g = nx.random_regular_graph(d, n+2)
```

```
l1 = nx.to_numpy_array(g)
eig1 = np.linalg.eigvals(l1)

Z1 = np.trace(np.exp(((-1) * l1)))

F1 = - np.log(Z1)

G = n * (F - F1)
print("Gibbs Free Energy:", G)
```

This is the results I got,

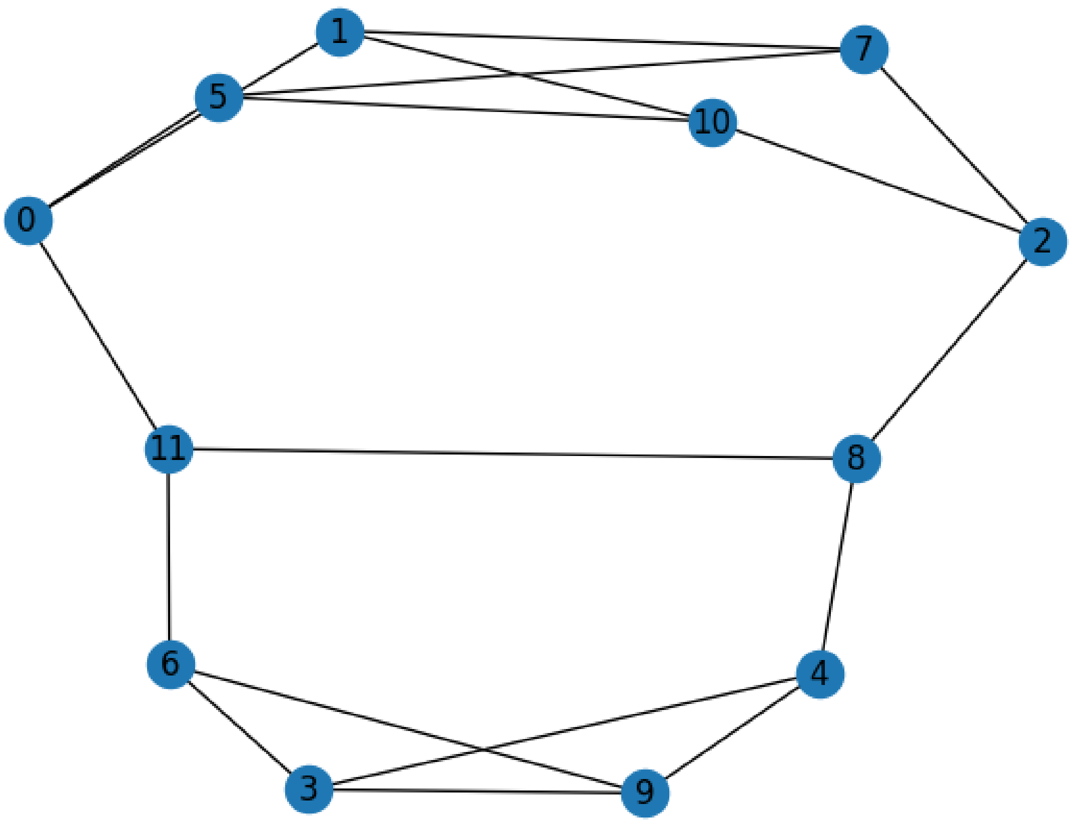

Figure 18: Regular Network n = 12 and d = 3

```
Eigenvalues of the Network:
[-2.89121985e+00  3.00000000e+00  2.65544238e+00  1.27841361e+00
  1.21075588e+00  3.17430608e-01 -1.86619826e+00 -1.70462437e+00
 -1.00000000e+00 -1.00000000e+00  5.80852991e-18 -2.98491875e-17]
Degree distribution: [0, 0, 0, 12]
Partition Function: 12.0
Probabilities of each degree: [0. 0. 0. 1.]
Entropy: -0.0
Helmholtz Free Energy: -2.4849066497880004
Gibbs Free Energy: 1.8498081579270966
```

3 KPZ Universality Class and the Wigner Semi Circle

1) To find the determinant of the Jacobian, I constructed the Jacobian symbolically, and then I computed the determinant of the Jacobian symbolically.

```python
# constructing the symbols
x11 = syp.Symbol("x_11")
x22 = syp.Symbol("x_22")
x12 = syp.Symbol("x_12")
l1 = syp.Symbol("\lambda_1")
l2 = syp.Symbol("\lambda_2")
theta = syp.Symbol("\\theta")

# constructing the matracies
O = syp.Matrix([[syp.cos(theta), -syp.sin(theta)], [syp.sin(theta), syp.cos(theta)]])
L = syp.Matrix([[l1, 0], [0, l2]])
X = (O @ L) @ syp.transpose(O)
x11 = X[0, 0]
x22 = X[1, 1]
x12 = X[0, 1]

# constructing the Jacobian and finding the determinant
J = syp.Matrix([[x11.diff(l1), x22.diff(l1), x12.diff(l1)],
                [x11.diff(l2), x22.diff(l2), x12.diff(l2)],
                [x11.diff(theta), x22.diff(theta), x12.diff(theta)]])
display(X, J)
d = J.det()
det = syp.simplify(d)
display(det)
```

and this is the Jacobian, unsimplified determinant, and simplified determinant that I got.

$$\mathrm{J} = \begin{bmatrix} \cos^2(\theta) & \sin^2(\theta) & \sin(\theta)\cos(\theta) \\ \sin^2(\theta) & \cos^2(\theta) & -\sin(\theta)\cos(\theta) \\ -2\lambda_1\sin(\theta)\cos(\theta)+2\lambda_2\sin(\theta)\cos(\theta) & 2\lambda_1\sin(\theta)\cos(\theta)-2\lambda_2\sin(\theta)\cos(\theta) & -\lambda_1\sin^2(\theta)+\lambda_1\cos^2(\theta)+\lambda_2\sin^2(\theta)-\lambda_2\cos^2(\theta) \end{bmatrix}$$

$$\det(\mathrm{J}) = \lambda_1\sin^6(\theta)+3\lambda_1\sin^4(\theta)\cos^2(\theta)+3\lambda_1\sin^2(\theta)\cos^4(\theta)+\lambda_1\cos^6(\theta)-\lambda_2\sin^6(\theta)-3\lambda_2\sin^4(\theta)\cos^2(\theta)-3\lambda_2\sin^2(\theta)\cos^4(\theta)-\lambda_2\cos^6(\theta) = \lambda_1-\lambda_2$$

2) This Hamiltonian can be interpreted as the potential energy of the system, λ_i^2 with a shifted term, $ln(|\lambda_i - \lambda_j|)$, and I plotted the Hamiltonian as a function of λ_1 and λ_2 taking $\lambda_i = [-50, 50]$ and $\lambda_j = [-50, 50]$.

```python
lam1 = syp.Symbol("\lambda_1")
lam2 = syp.Symbol("\lambda_2")
def H(lam1, lam2):
    return (1/2) * ((lam1 ** 2) - syp.log(syp.Abs(lam1 - lam2)))
plot3d(H(lam1, lam2), (lam1, -50, 50), (lam2, -50, 50))
```

This is the graph I got,

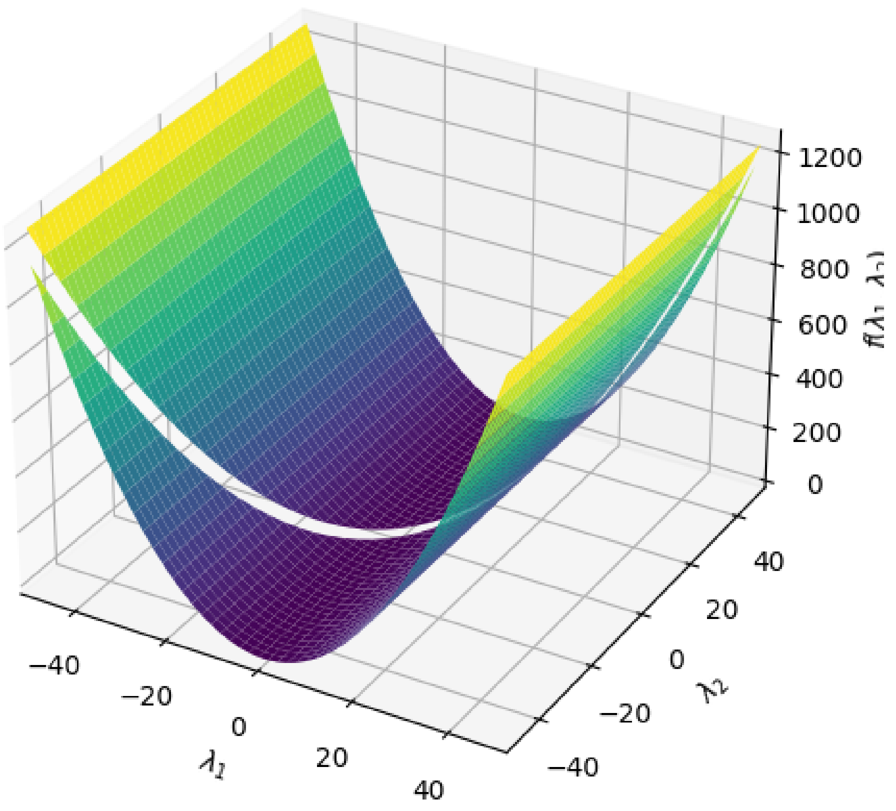

Figure 19: The Hamiltonian as a Function of the Eigenvalues

It looks as expected, at small eigenvalues, it's parabolic, and at larger eigenvalues, it's logarithmic. Also, as expected the curve is centered at zero and there is a discontinuity when $\lambda_i = \lambda_j$ and the logarithm is not defined.

3) The spectral density of an Erdös-Rényi random network follows Wigner's semicircle law where the eigenvalues of the network lies mostly in the range $[-2r,\ 2r]$, r being $r = \sqrt{Np(1-p)}$ and p is the probability of rewiring. So to see the spectral density of the random network, I generated a Er:os-Rényi random network with $N = 1000$ nodes and $p = 0.3$ rewiring probability. I found its adjacency matrix and its eigenvalues. Then, I found the density of the eigenvalues and plotted them.

```
N = 1000
p = 0.3
G = nx.erdos_renyi_graph(N, p)

r = np.sqrt(N*p*(1-p))
A = nx.to_numpy_array(G)
lam = np.linalg.eigvals(A)
n, l = np.histogram(lam, bins = 100, range = (-2 * r, 2 * r), density = True)
x = (1/2)*(l[1:] + l[:-1])
y = n**2

plt.hist(lam, bins=100, range= (-2 * r, 2 * r), color = 'r', density = True)
plt.title("Spectral Density")
plt.xlabel("$\lambda$")
plt.ylabel("Density")
plt.show()
```

This is the histogram of the density that I got,

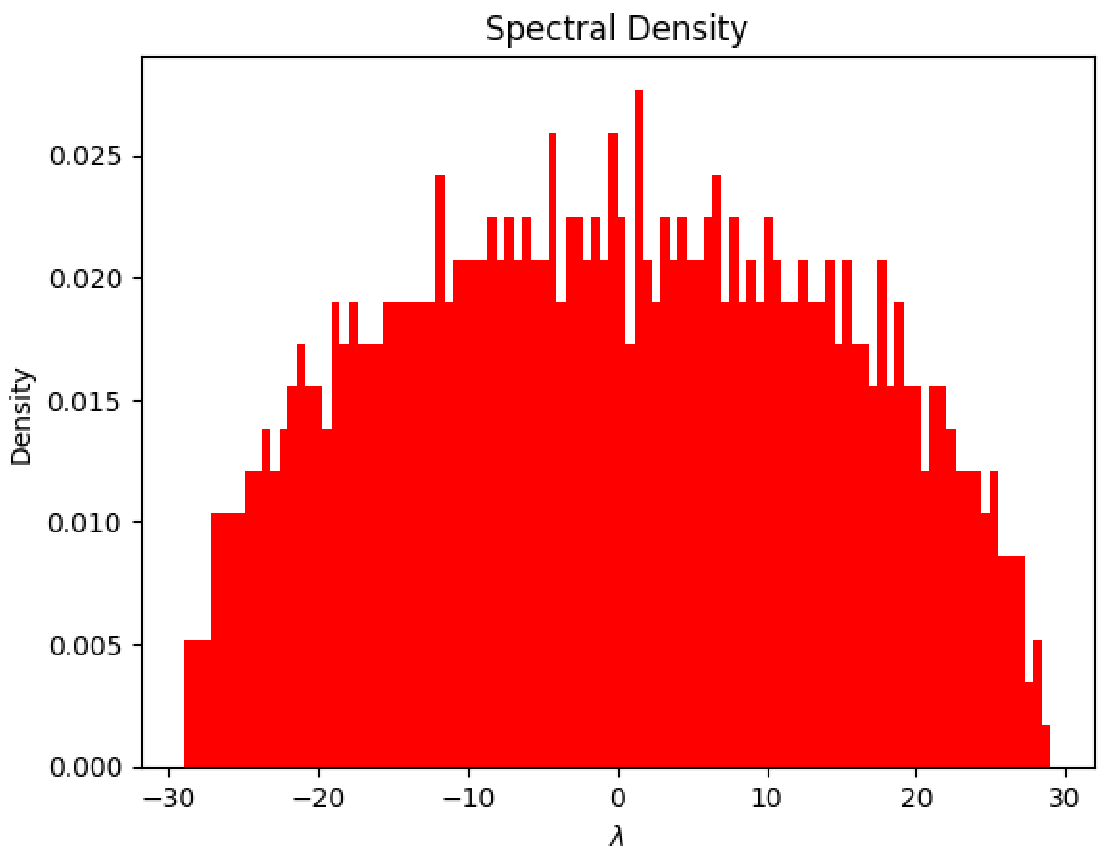

Figure 20: Spectral Density Histogram

The density function of Wigner's semicircle law is given by $\rho(\lambda) = \frac{\sqrt{(4-\lambda_2)}}{2\pi}$, which can be written as $\rho^2(\lambda) = \left(\frac{1}{2\pi}\right)\left(4-\lambda^2\right)$, so the square of the density is $\rho^2 = a + b\lambda^2$. So, I used a regression program that I wrote in the early stages of this course to fit the spectral density data.

```
def regression(x, y, n):
    def Vandermond(x, y, n):
        V = np.empty([len(x), n + 1])
        for i in range(n + 1):
            V[:, i] = x ** i
        return V

    def LeastSquare(V, y):
        Vt = np.transpose(V)
        A = Vt @ V
        Y = Vt @ y
        b = np.linalg.solve(A, Y)
        return b

    def fitpolynomial(x, b, n):
        Ps = np.zeros(len(x))
        for j in range(len(x)):
            p = 0
            for i in range(n + 1):
                p += b[i] * x[j] ** i
            Ps[j] = p
        return Ps

    V = Vandermond(x, y, n)
```

```
        b = LeastSquare(V, y)
        Ps = fitpolynomial(x, b, n)
        for i in range(len(x) - 1):
            xs = np.linspace(x[i], x[i+1], 50)
            p = fitpolynomial(xs, b, n)
            plt.plot(xs,p, color = 'k')

    plt.scatter(l[1:], n**2)
    plt.title("Spectral Density")
    plt.xlabel("$\lambda$")
    plt.ylabel("Density")
    regression(x, y, 2)
    plt.show()
```

and I obtained this graph showing the spectral density and the fit.

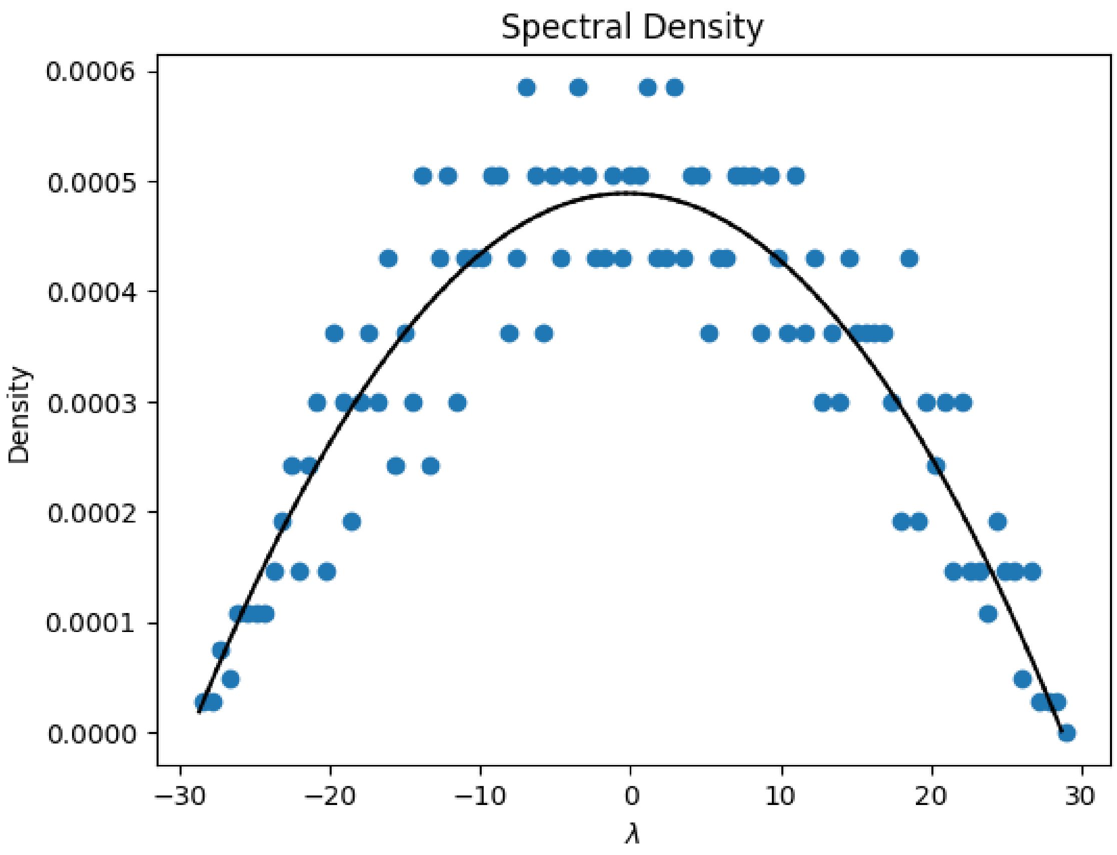

Figure 21: The Spectral Density with its Appropriate Fit

This verifies that the spectral density of an Erdös-Rényi random network follows Wigner's semicircle law.

4) To find different maximum eigenvalues, I generated multiple random networks with the same number of nodes and rewiring probability, and I computed their eigenvalues and plotted them in a histogram.

```
    N = 1000
    p = 0.3
    maxl = np.zeros(1000)
    for i in range(1000):
        G = nx.erdos_renyi_graph(N, p)
        A = nx.to_numpy_array(G)
        lam = np.linalg.eigvals(A)
        maxl[i] = np.max(lam)
    plt.hist(maxl, bins=100, color = 'r')
    plt.show()
```

This is the graph that I got,

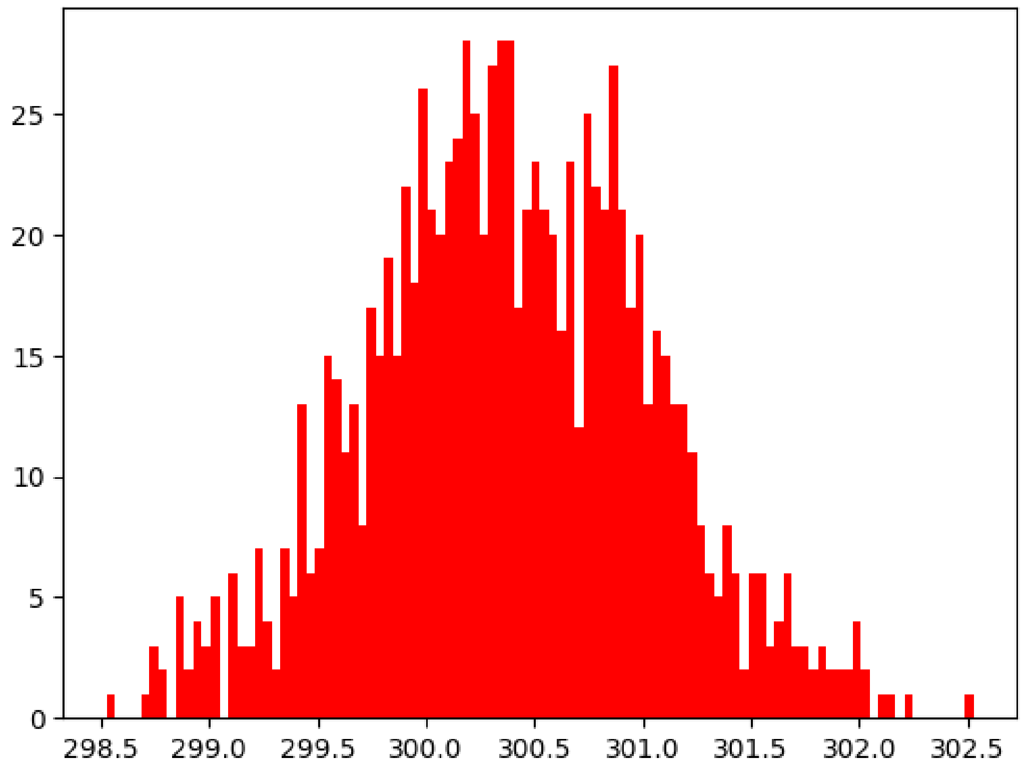

Figure 22: Maximum Eigenvalues of Different Random Networks

This confirmed the notion that the top eigenvalues follow the Tracy-Widom distribution.

4 Dynamics

1) We have the following system of differential equations

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

I applied Runge-Kutta 4 to solve the system and obtain the values of x, y, z as a function of t. I used the standard parameter values $\sigma = 10,\ r = 28,\ b = \frac{8}{3}$ with initial conditions $x_0 = y_0 = z_0 = 0.1 and t_0 = 0$

```
N = 10000
t = np.zeros(N)
x = np.zeros(N)
y = np.zeros(N)
z = np.zeros(N)
t[0] = 0
x[0] = 0.1
y[0] = 0.1
z[0] = 0.1
h = 0.01
sig = 10
r = 28
b = 8/3
for i in range(1, len(t)):
    kx1 = h * sig * (y[i-1]-x[i-1])
    kx2 = h * sig * ((y[i-1]) - (x[i-1] + (kx1/2)))
    kx3 = h * sig * ((y[i-1]) - (x[i-1] + (kx2/2)))
```

```
        kx4 = h * sig * ((y[i-1]) - (x[i-1] + (kx3)))
        x[i] = x[i-1] + (1/6) * (kx1 + 2 * kx2 + 2 * kx3 + kx4)

        ky1 = h * (r * x[i-1] - y[i-1] - x[i-1] * z[i-1])
        ky2 = h * (r * (x[i-1]) - (y[i-1] + (ky1/2)) - ((x[i-1]) * (z[i-1])))
        ky3 = h * (r * (x[i-1]) - (y[i-1] + (ky2/2)) - ((x[i-1]) * (z[i-1])))
        ky4 = h * (r * (x[i-1]) - (y[i-1] + (ky3)) - ((x[i-1]) * (z[i-1])))
        y[i] = y[i-1] + (1/6) * (ky1 + 2 * ky2 + 2 * ky3 + ky4)

        kz1 = h * (x[i-1] * y[i-1] - b * z[i-1])
        kz2 = h * ((x[i-1])*(y[i-1]) - b * (z[i-1] + (kz1/2)))
        kz3 = h * ((x[i-1])*(y[i-1]) - b * (z[i-1] + (kz2/2)))
        kz4 = h * ((x[i-1])*(y[i-1]) - b * (z[i-1] + (kz3)))
        z[i] = z[i-1] + (1/6) * (kz1 + 2 * kz2 + 2 * kz3 + ky4)

        t[i] = t[i-1] + h
    plt.figure(figsize= (40, 40))
    plt.plot(t, x, label = "x(t)")
    plt.plot(t, y, label = "y(t)")
    plt.plot(t, z, label = "z(t)")
    plt.legend()
    plt.show()
    plt.figure(figsize= (40, 40))
    plt.plot(x, z, 'k')
    plt.show()
```

and I plotted $x(t),\ y(t),\ and z(t)$ as a function of t, z as a function of x, and z as a function of $x$and y.

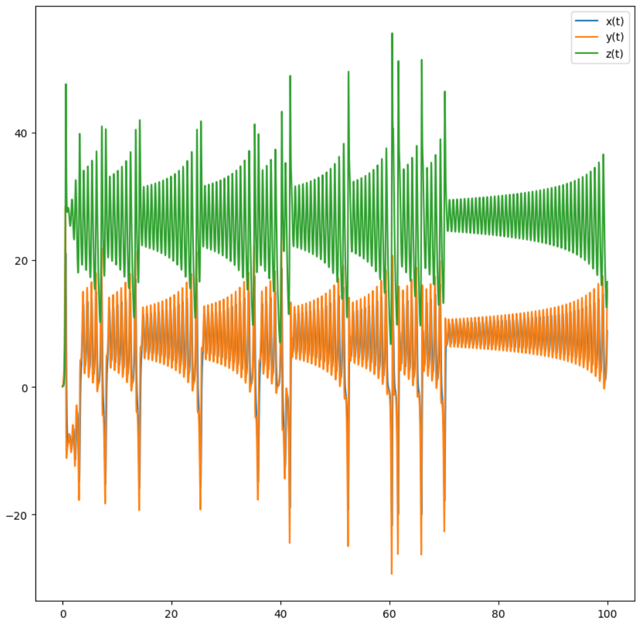

Figure 23: $x(t),\ y(t),\ and z(t)$ as a function of t

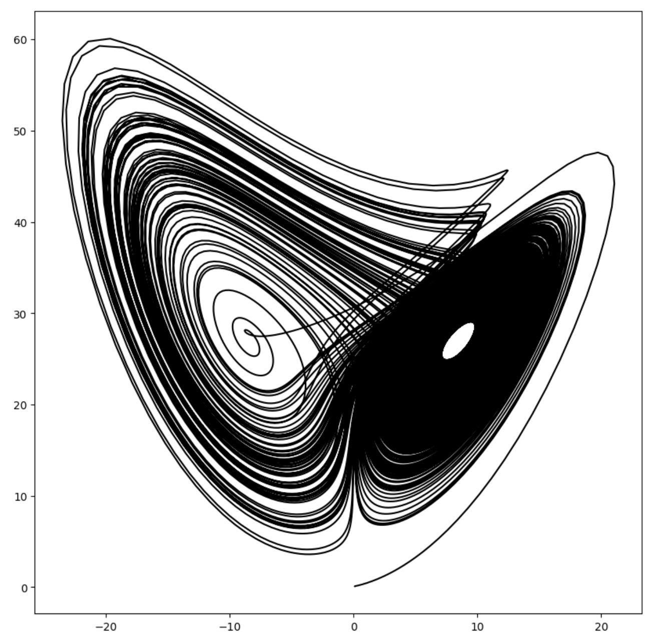
Figure 24: z as a function of x

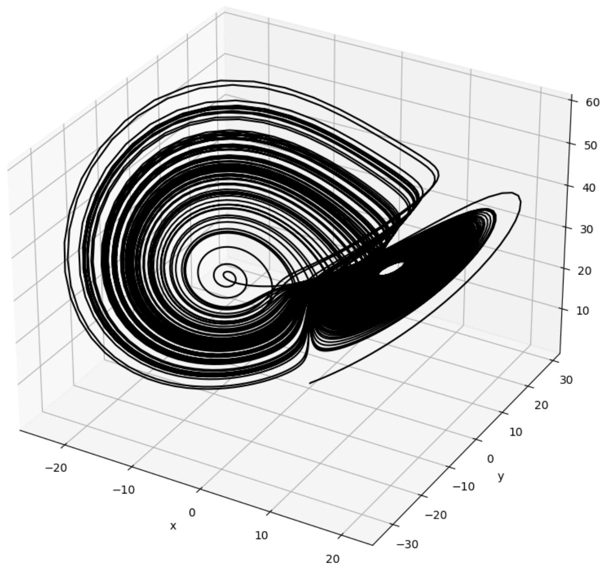

Figure 25: z as a function of $x$and y

2) I used Newton Raphson Method to find the fixed points of the system and I plotted them while varying r. I discovered that the fixed points have a behavior that is called pitchfork bifurcation.

```python
def F(x, y, z, r, b, sig):
    return np.array([[sig * (y-x)], [r*x - y - x * z], [x * y - b * z]])

def J(x, y, z, r, b, sig):
    j = np.zeros((3, 3))
    j[0, :] = np.array([-sig, sig, 0])
    j[1, :] = np.array([r - z, -1, -x])
    j[2, :] = np.array([y, x, -b])
    return j
```

```python
def NewtonRaphson(x0, y0, z0, r, b, sig):
    x = np.array([x0, y0, z0])
    for i in range(100):
        xs = x[0]
        ys = x[1]
        zs = x[2]
        f = F(xs, ys, zs, r, b, sig)
        j = J(xs, ys, zs, r, b, sig)
        xs = np.linalg.solve(j, f)
        x = x - xs.flatten()
        i += 1
    return x

r = np.linspace(0, 4, 200)
xs = []
ys = []
zs = []
rs = []
for i in range(len(r)):
    x1 = NewtonRaphson(0.1, 0.1, 0.1, r[i], 8/3, 10)
    x2 = NewtonRaphson(10, 10, 10, r[i], 8/3, 10)
    x3 = NewtonRaphson(-6, -6, -6, r[i], 8/3, 10)
    xs.extend([x1[0], x2[0], x3[0]])
    ys.extend([x1[1], x2[1], x3[1]])
    zs.extend([x1[2], x2[2], x3[2]])
    rs.extend([r[i]] * 3)
plt.scatter(rs, xs, c = 'k', marker= '.', s = 1)
plt.show()
plt.scatter(rs, ys, c = 'k', marker= '.', s = 1)
plt.show()
plt.scatter(rs, zs, c = 'k', marker= '.', s = 1)
plt.show()
```

and these are the graphs I obtained,

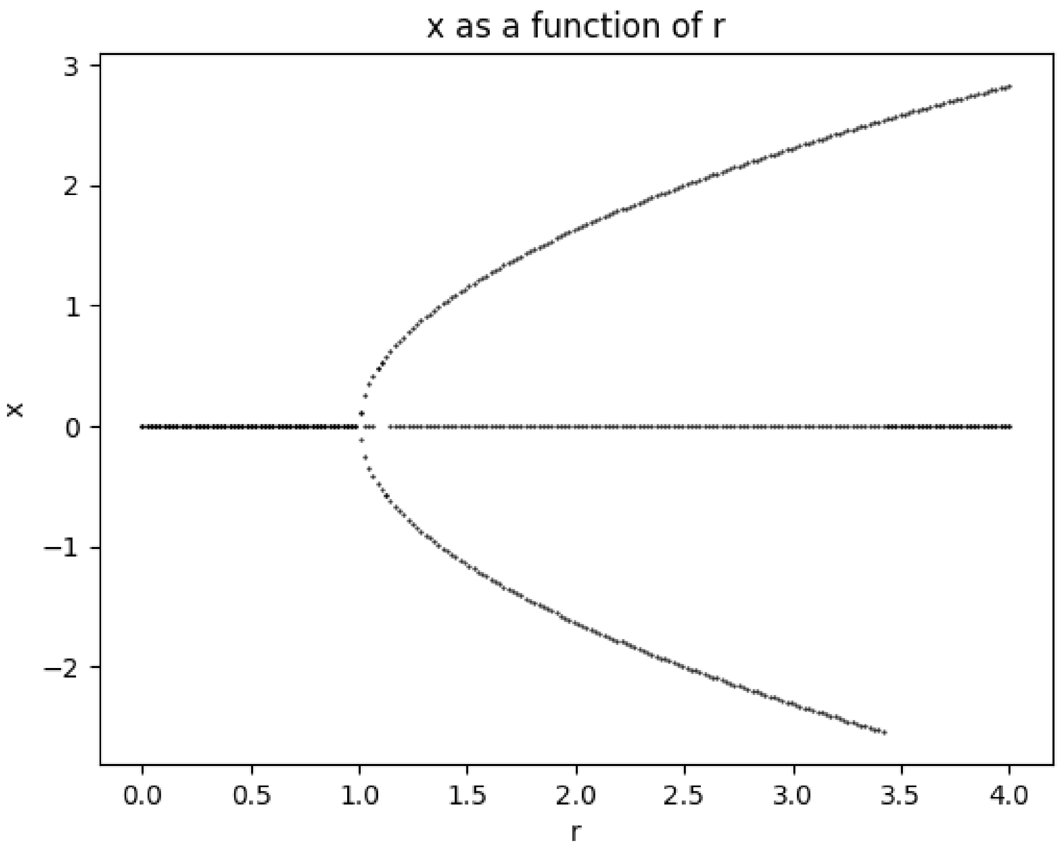

Figure 26: x roots as a function of r

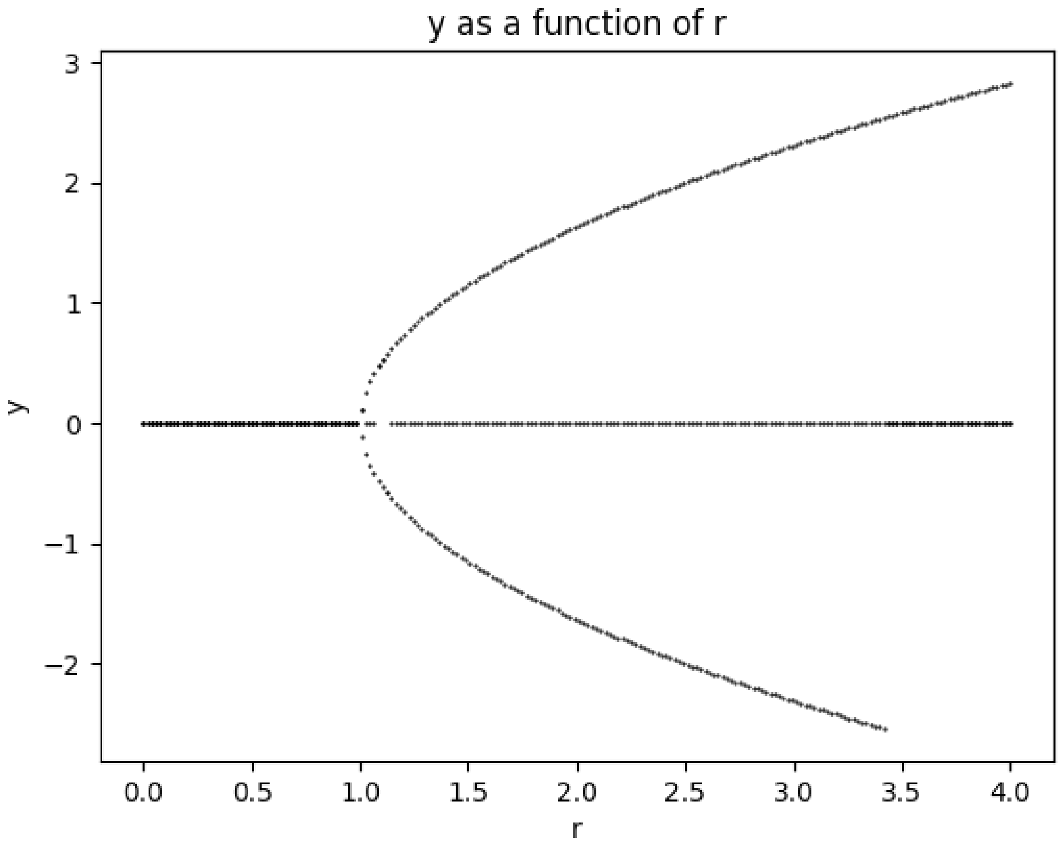

Figure 27: y roots as a function of r

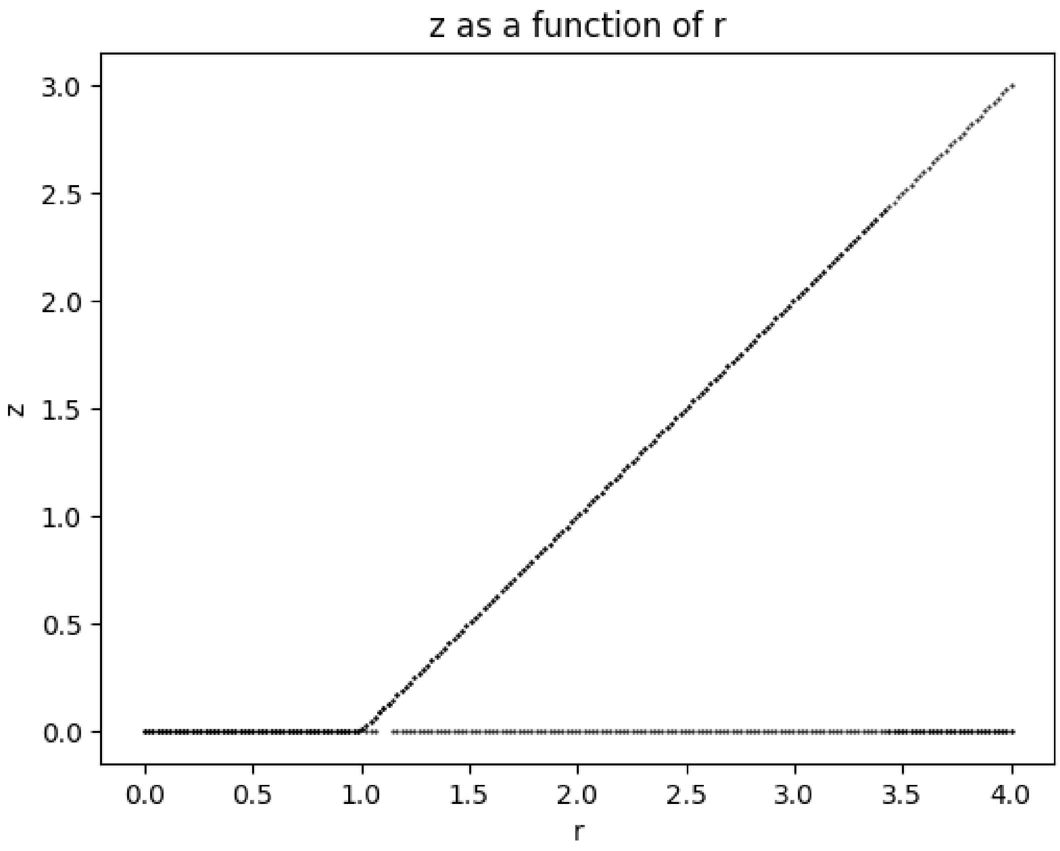

Figure 28: z roots as a function of r

Then, I evaluated the stability of these points by checking the eigenvalues of the Jacobian at those points.

```
def stability(x, r, b, sig):
j = J(x[0], x[1], x[2], r, b, sig)
l = np.linalg.eigvals(j)
print(l)
if np.any(np.iscomplex(l)):
    print("Complex Eigenvalues: Center Oscillatory Point")
elif np.all(np.real(l) > 0):
    print("Positive Real Eigenvalues: Unstable Point")
elif np.all(np.real(l) < 0):
    print("Negative Real Eigenvalues: Stable Point")
else:
    print("Mixed Real Eigenvalues: Saddle Point")

x = NewtonRaphson(0.1, 0.1, 0.1, 2, 8/3, 10)
print(x)
stability(x, 2, 8/3, 10)
print()

x = NewtonRaphson(10, 10, 10, 2, 8/3, 10)
print(x)
stability(x, 2, 8/3, 10)
print()

x = NewtonRaphson(-6, -6, -6, 2, 8/3, 10)
print(x)
stability(x, 2, 8/3, 10)
```

For $\sigma = 8/3,\ b = 10,$ and $r = 2$, I got the following fixed points and their corresponding stability,

Fixed Point	Eigenvalues of the Jacobian	Type of Eigenvalues	Type of Fixed Point
$[0, 0, 0]$	$[-11.84, 0.84, -2.66]$	Real Mixed	Saddle Point
$[1.63, 1.63, 1]$	$[-11.23, -1.21 + 1.8i, -1.21 - 1.8i]$	Imaginary	Center
$[-1.63, -1.63, 1]$	$[-11.23, -1.21 + 1.8i, -1.21 - 1.8i]$	Imaginary	Center

3) To find the Liapunov exponent, I considered two very close initial conditions $\mathbf{x_{10}} = (0.5, 0.6, 0.7)$ and $\mathbf{x_{20}} = (0.6, 0.7, 0.8)$. I calculated the distance between them $||\delta_0|| = \sqrt{\Sigma(x_{10,i} - x_{20,i})^2}$. I let the points evolve over time according to the equations that govern the system. I then picked the two points at a later time $\mathbf{x_{11}}$ and $\mathbf{x_{21}}$ and calculated the distance between them $||\delta(t)||$. The expression of the Liapunov exponent is given by $||\delta(t)|| = ||\delta_0||e^{\lambda t}$. I have the distances and time, so I solved for λ, and I got,

$$\lambda = \frac{1}{t}\frac{||\delta(t)||}{||\delta_0||}$$

and I wrote a code that would calculate λ for any given two initial points.

```
def liapunov_exponent(x10, x20):
    d0 = np.sqrt(np.sum((x10 - x20) ** 2))
    t1, x1, y1, z1 = Lorenz_Attractor(N = 10000, h = 0.01, x0 = x10[0], y0 = x10[1],
                    z0 = x10[2], t0 = 0, sig = 10, r = 28, b = 8/3)
    t2, x2, y2, z2 = Lorenz_Attractor(N = 10000, h = 0.01, x0 = x20[0], y0 = x20[1],
                    z0 = x20[2], t0 = 0, sig = 10, r = 28, b = 8/3)

    x11 = np.array([x1[9999], y1[9999], z1[9999]])
    x22 = np.array([x2[9999], y2[9999], z2[9999]])
    dt = np.sqrt(np.sum((x11 - x22) ** 2))

    lam = (1/t[9999]) * np.log(dt/d0)

    return lam
```

```
x10 = np.array([0.5, 0.6, 0.7])
x20 = np.array([0.6, 0.7, 0.8])
lam = liapunov_exponent(x10, x20)
print(f"For x10 = {x10} and x20 = {x20}: Lambda is {lam}")
```

For $\mathbf{x_{10}} = (0.5, 0.6, 0.7)$ and $\mathbf{x_{20}} = (0.6, 0.7, 0.8)$, $\lambda = 0.04244068476563179$

4) I found the maximum of z and I plotted z_{n+1} as a function of z_n.

```
x0 = np.array([0.5, 0.6, 0.7])
t, x, y, z = Lorenz_Attractor(N = 1000000, h = 0.001, x0 = x0[0], y0 = x0[1],
                              z0 = x0[2], t0 = 0, sig = 10, r = 28, b = 8/3)

indices = []
max_z = []
max_t = []
for i in range(1, len(x) - 1):
    if z[i] > z[i-1] and z[i] > z[i+1]:
        indices.append(i)
        max_z.append(z[i])
        max_t.append(t[i])

plt.plot(t, z)
plt.xlabel("t")
plt.ylabel("z")
plt.scatter(max_t, max_z, color = 'r')
plt.show()

plt.scatter(max_z[:-1], max_z[1:])
plt.xlabel("$z_n$")
plt.ylabel("$z_{n+1}$")
plt.show()
```

and these are the graphs I obtained,

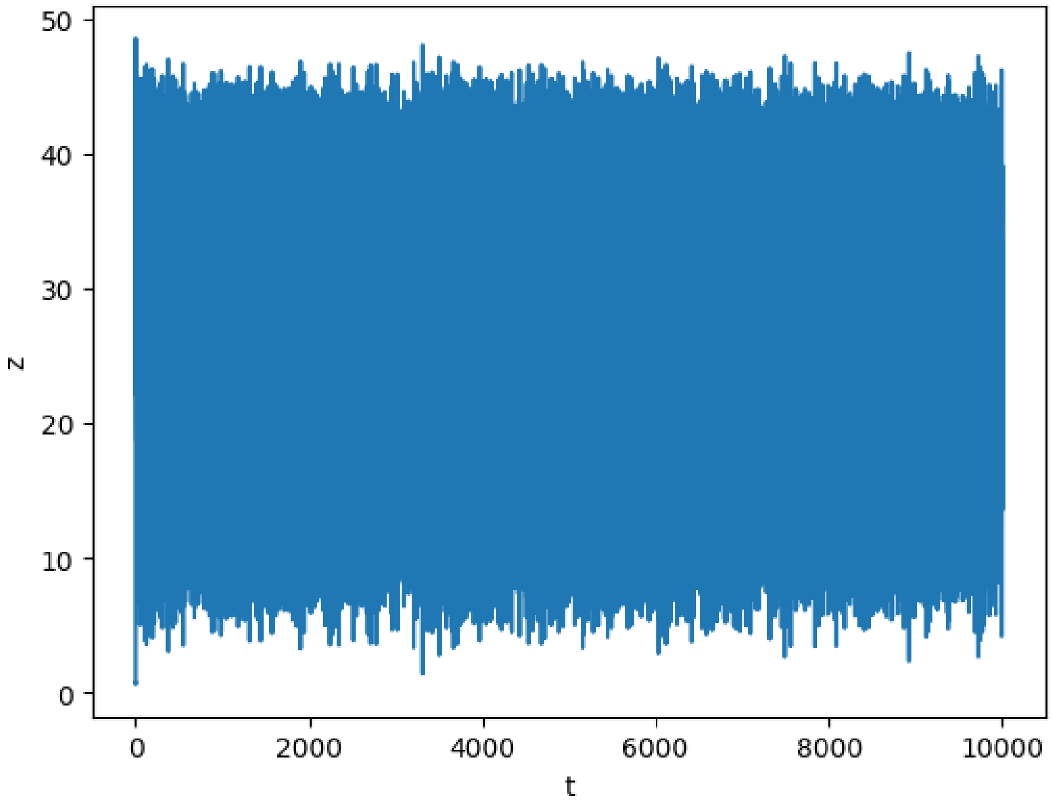

Figure 29: z as a function of t

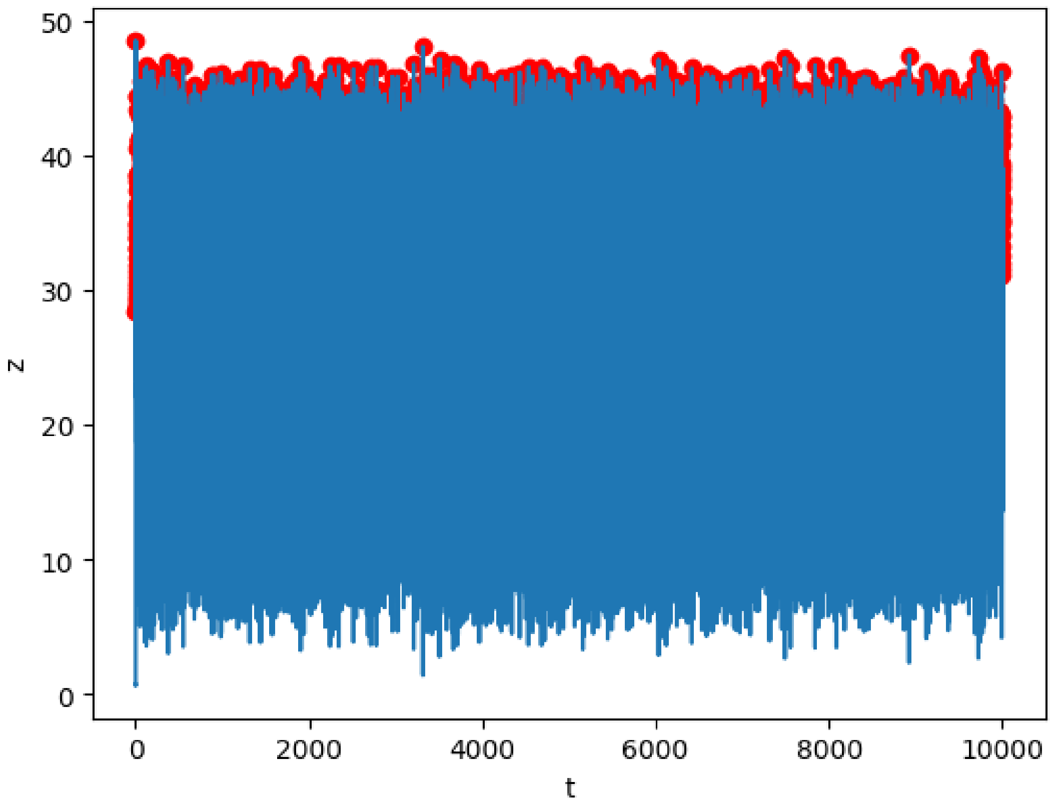

Figure 30: z as a function of t with the maximums highlighted

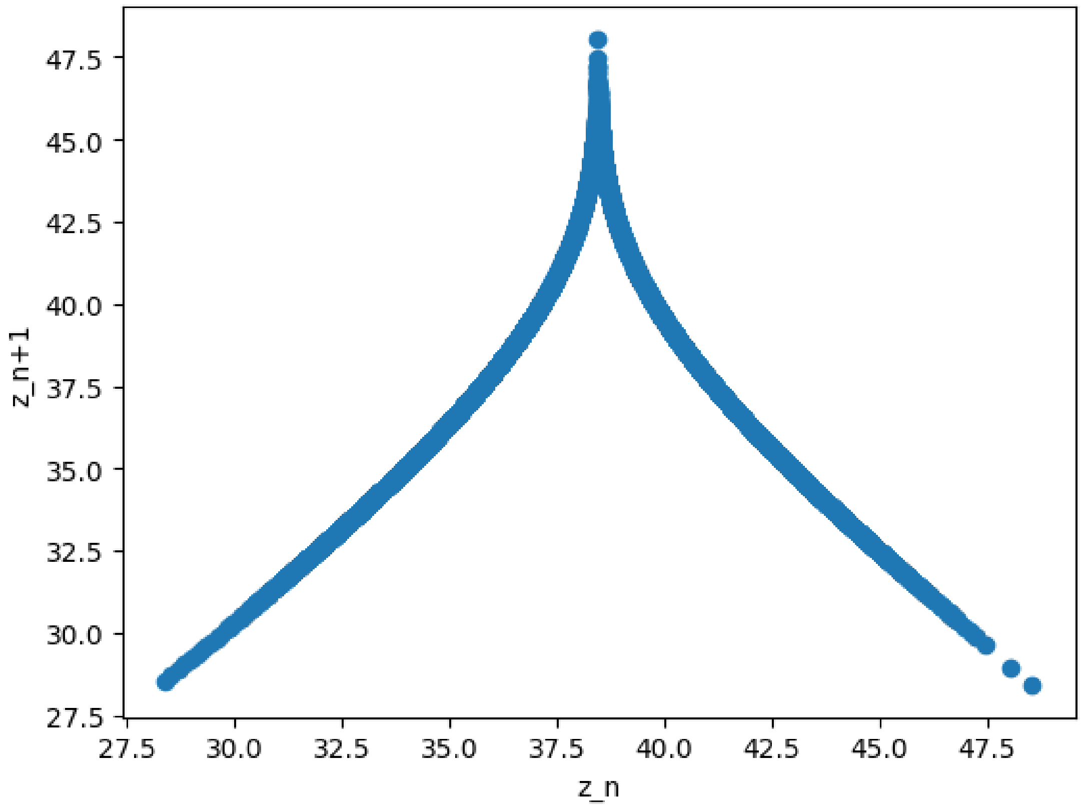

Figure 31: z_{n+1} as a function of z_n

Lastly, I wrote an algorithm that would determine of the limit cycle was stable or not based on the derivative of z $|f'(z) > 1$.

```
dz = np.gradient(z)
dzs = np.sqrt(np.sum(dz ** 2))
if dzs > 1:
    print("The limit cycle is unstable")
elif dzs < 1:
    print("The limit cycle is stable")
```

The system I was studying had an unstable limit cycle.

5 Logistic Map

1) Given the equation of the logistic map,

$$x_{n+1} = f(x_n) = \mu x(1-x)$$

I plotted $f(x)$ as a function of x for $\mu = 1$.

```
N = 100
mu = 1
x = np.linspace(0, 1.2, N)
plt.plot(x, mu * x * (1-x))
plt.grid()
plt.show()
```

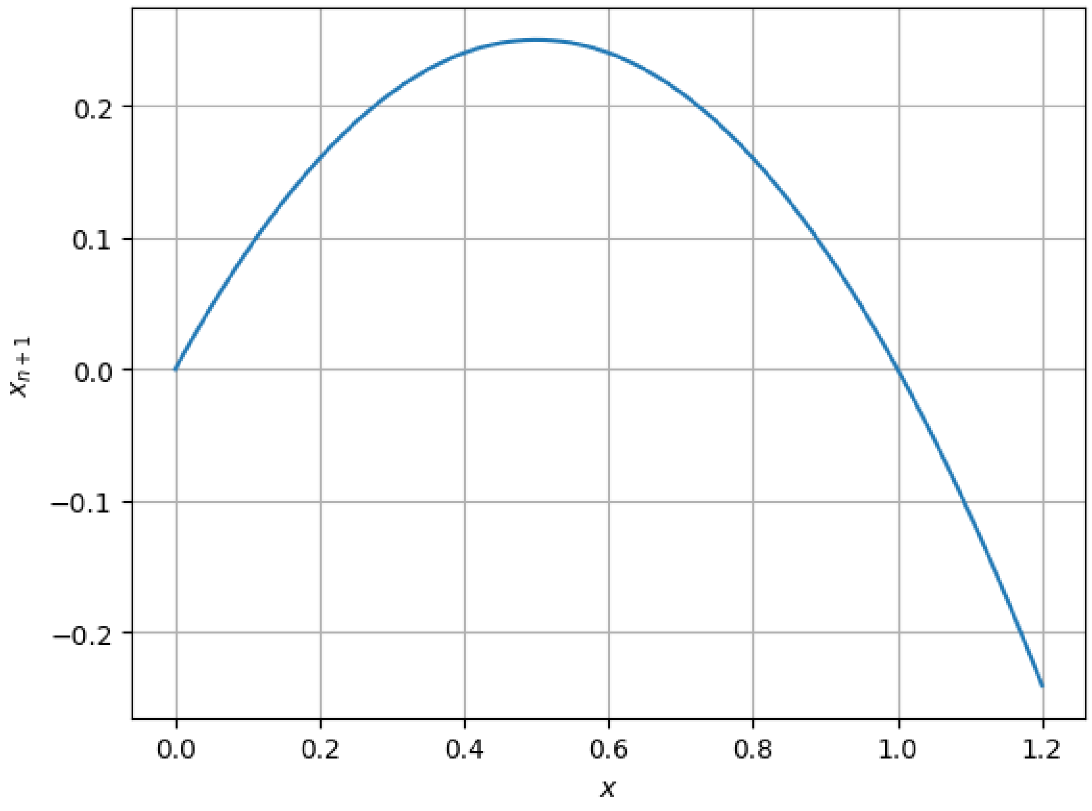

Figure 32: $f(x)$ as a function of x for $\mu = 1$

2) Then, I solved the logistic map, and for large μ, the function behaved in a ultra chaotic way. Then, I took those solutions and I plotted them against μ to get the logistic map.

```
def logistic_equation(mu, x0, N):
    x = np.zeros(N)
    x[0] = x0
    for i in range(N - 1):
        x[i+1] = mu * x[i] * (1 - x[i])
    return x

N = 100
mu = np.linspace(0, 4.0, N)
mus = []
values = []
for i in range(len(mu)):
    x = logistic_equation(mu[i], 0.8, N)
    mus.extend([mu[i]] * (N - 10))
    values.extend(x[10:])
plt.scatter(mus, values, c = 'k', marker= '.', s = 1)
plt.show()
```

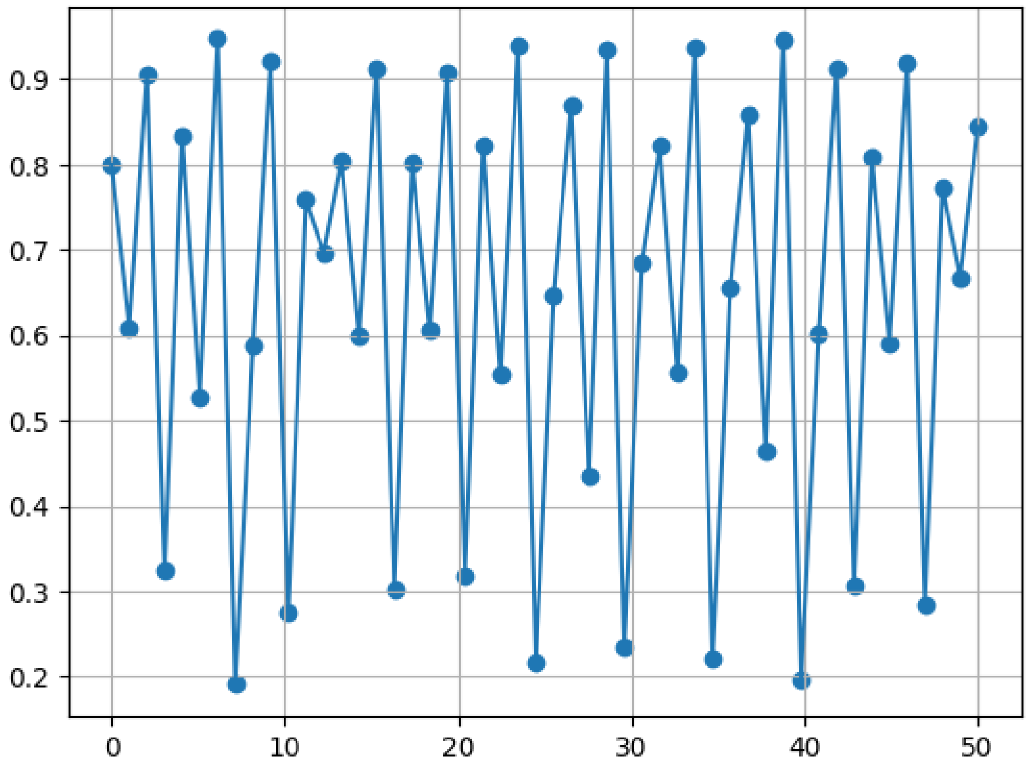

Figure 33: Solution of the Map for $\mu = 3.8$

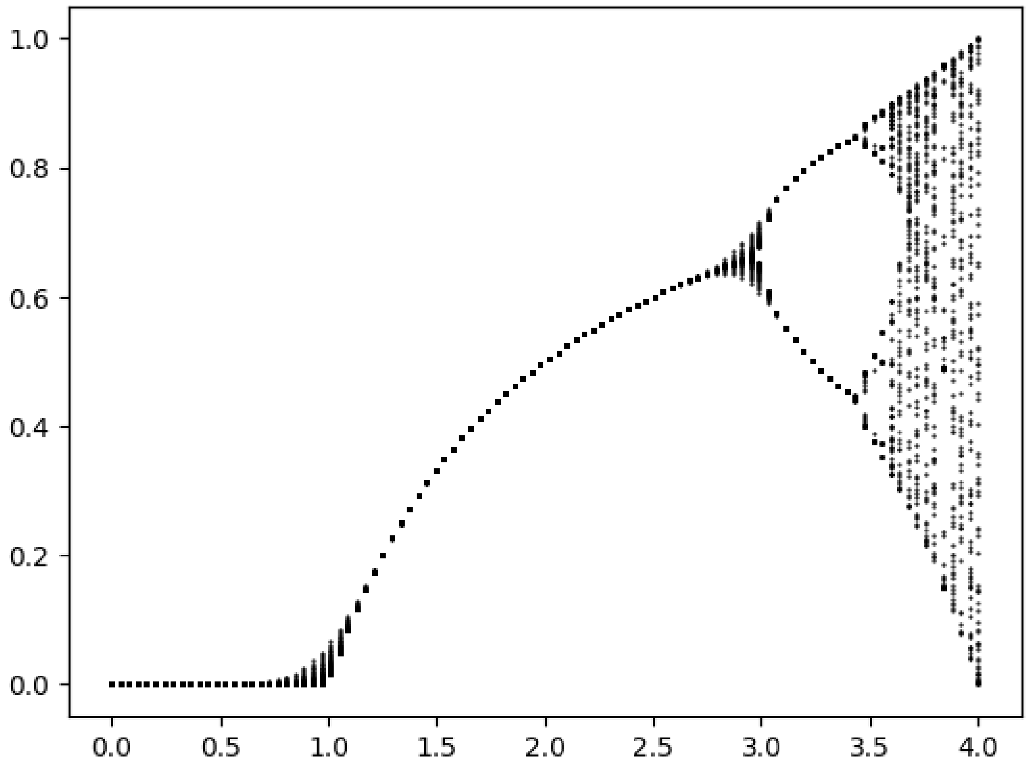

Figure 34: Logistic Map

I then used the chaotic regime for high μ and I created a random number generator that can generate for any desired order.

```
def random_number(order):
    x = logistic_equation(4, 0.8, 100)
    r = rand.choice(x) * order
    return r
```

6 What Do ID Maps Have to Do With Science?

1) The Rossler system is defined as

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

Taking $a = b = 0.2$ and varying $c = [2.5, 3.5, 4, 5]$ with initial conditions $x_0 = y_0 = z_0 = 1$, I solved the Rossler System using Runge-Kutta 4 and I plotted y against x and the system in 3D.

```
def Rossler_System(a, b, c, N, h, t0, x0, y0, z0):
t = np.zeros(N)
x = np.zeros(N)
y = np.zeros(N)
z = np.zeros(N)
t[0] = t0
x[0] = x0
y[0] = y0
z[0] = z0
for i in range(1, len(t)):
    kx1 = h * (- y[i-1] - z[i-1])
    kx2 = h * (- (y[i-1]) - (z[i-1]))
    kx3 = h * (- (y[i-1]) - (z[i-1]))
    kx4 = h * (- (y[i-1]) - (z[i-1]))
    x[i] = x[i-1] + (1/6) * (kx1 + 2 * kx2 + 2 * kx3 + kx4)

    ky1 = h * (x[i-1] + a * y[i-1])
    ky2 = h * ((x[i-1]) + a * (y[i-1] + (ky1/2)))
    ky3 = h * ((x[i-1]) + a * (y[i-1] + (ky2/2)))
    ky4 = h * ((x[i-1]) + a * (y[i-1] + (ky3)))
    y[i] = y[i-1] + (1/6) * (ky1 + 2 * ky2 + 2 * ky3 + ky4)

    kz1 = h * (b + z[i-1] * (x[i-1] - c))
    kz2 = h * (b + (z[i-1] + (kz1/2)) * ((x[i-1]) - c))
    kz3 = h * (b + (z[i-1] + (kz2/2)) * ((x[i-1]) - c))
    kz4 = h * (b + (z[i-1] + (kz3)) * ((x[i-1]) - c))
    z[i] = z[i-1] + (1/6) * (kz1 + 2 * kz2 + 2 * kz3 + kz4)
    t[i] = t[i-1] + h
return t, x, y, z

a, b = 0.2, 0.2
c = [2.5, 3.5, 4, 5]
t0 = 0
x0, y0, z0 = 1, 1, 1
h = 1e-2
N = 100000
for i in range(len(c)):
    t, x, y, z = Rossler_System(a, b, c[i], N, h, t0, x0, y0, z0)
    print(f"a = {a}, b = {b}, c = {c[i]}")
    plt.plot(y,x)
    plt.xlabel("y")
    plt.ylabel("x")
    plt.title(f"a = {a}, b = {b}, c = {c[i]}")
    plt.show()

    ax = plt.axes(projection = "3d")
    ax.set_xlabel("x")
    ax.set_ylabel("y")
    ax.set_zlabel("z")
    ax.plot3D(x, y, z)
    plt.show()
```

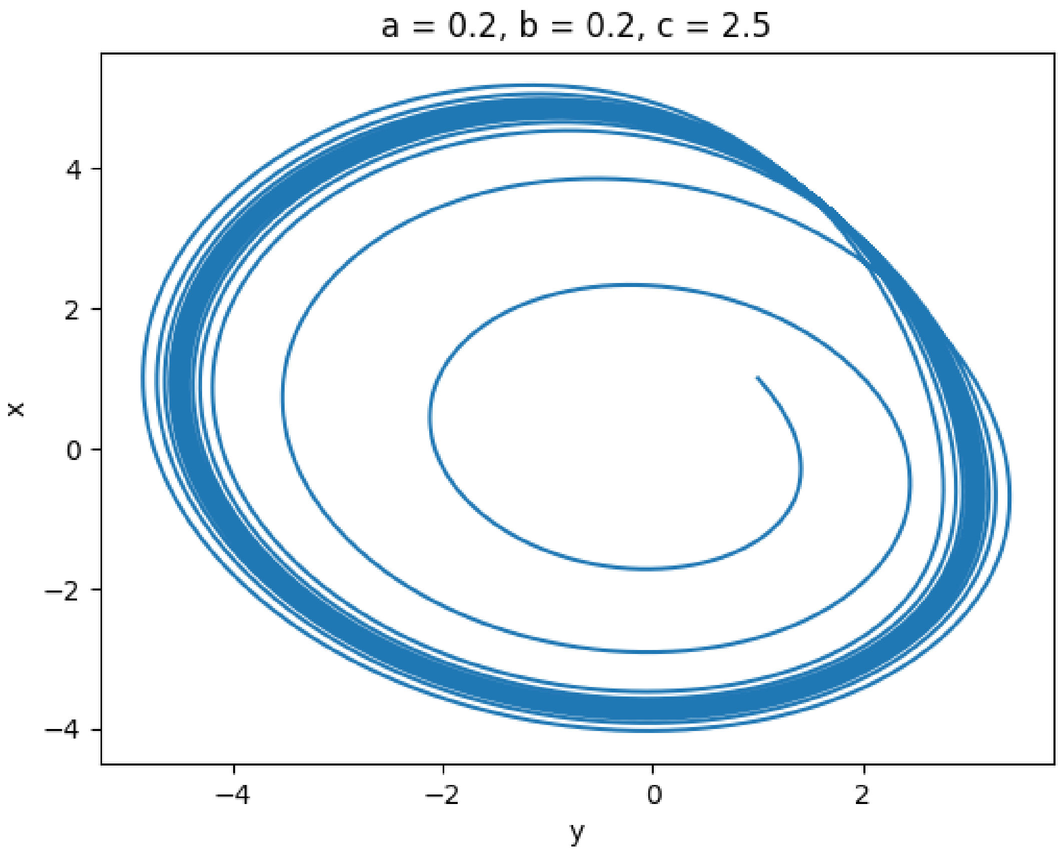

Figure 35: y versus x c = 2.5

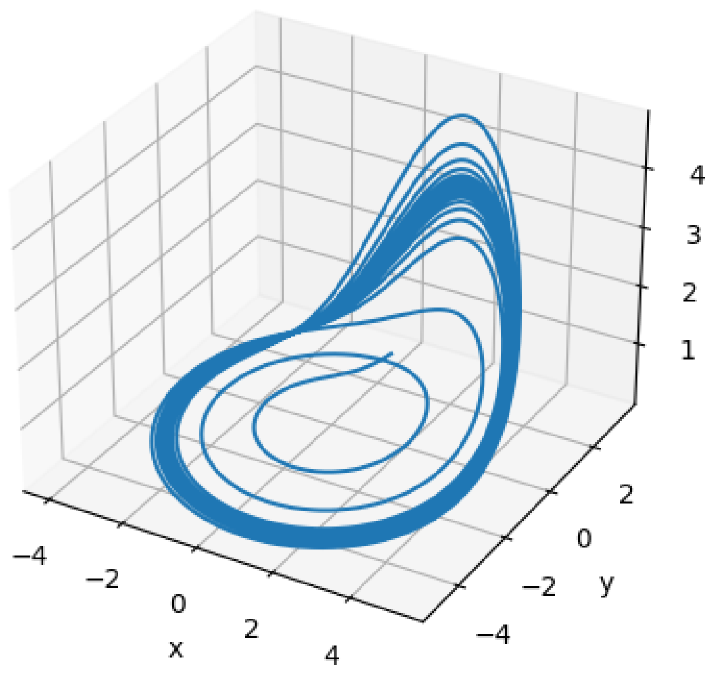

Figure 36: c = 2.5

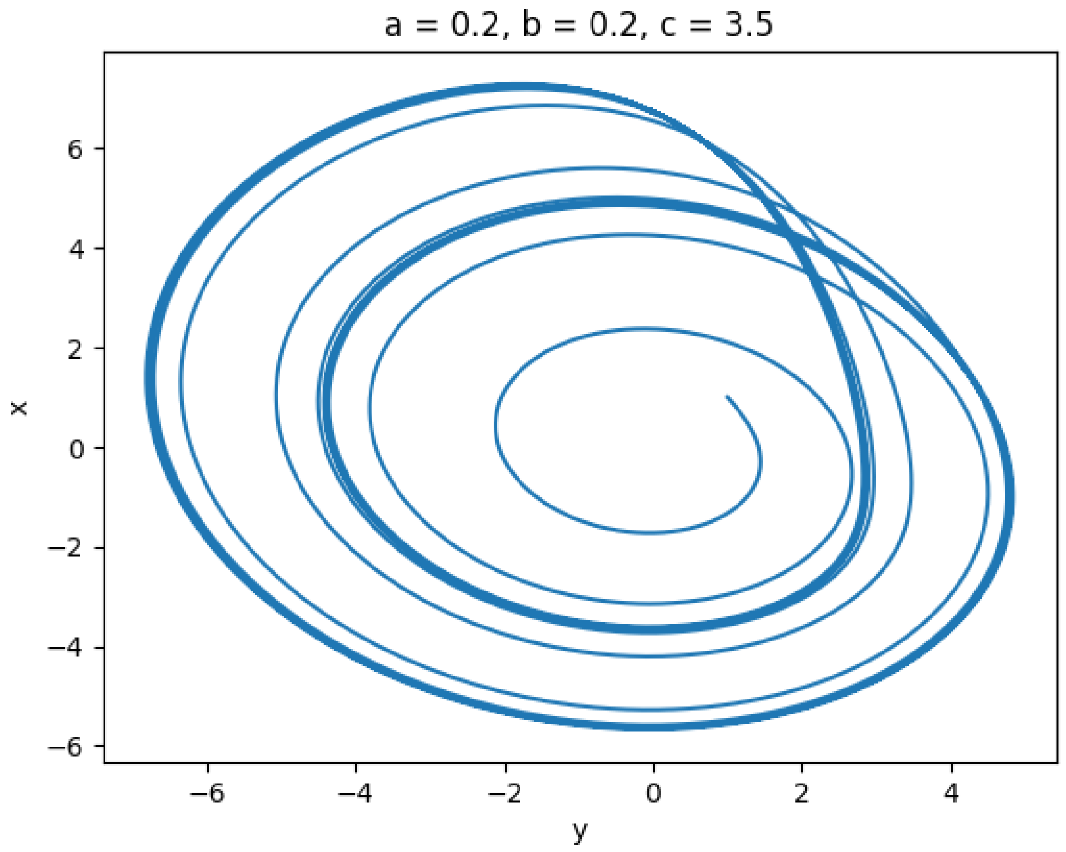

Figure 37: y versus x c = 3.5

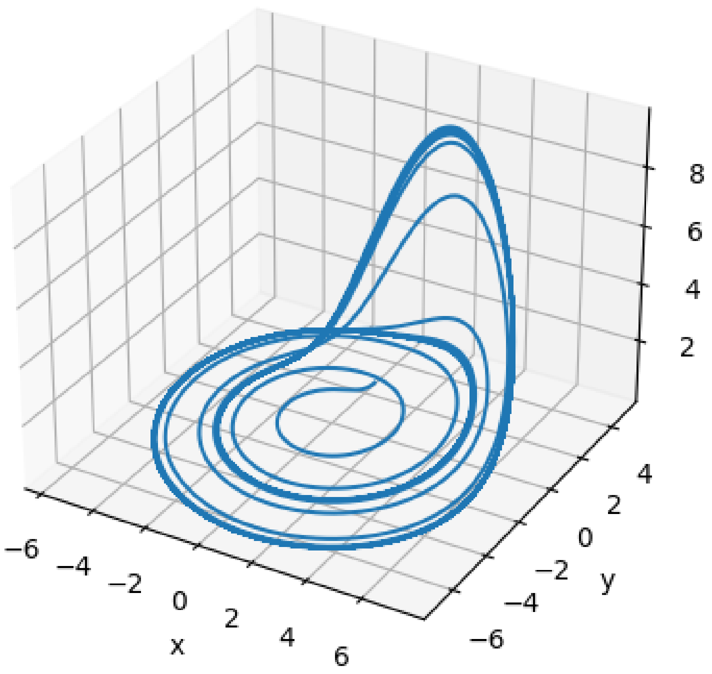

Figure 38: c = 3.5

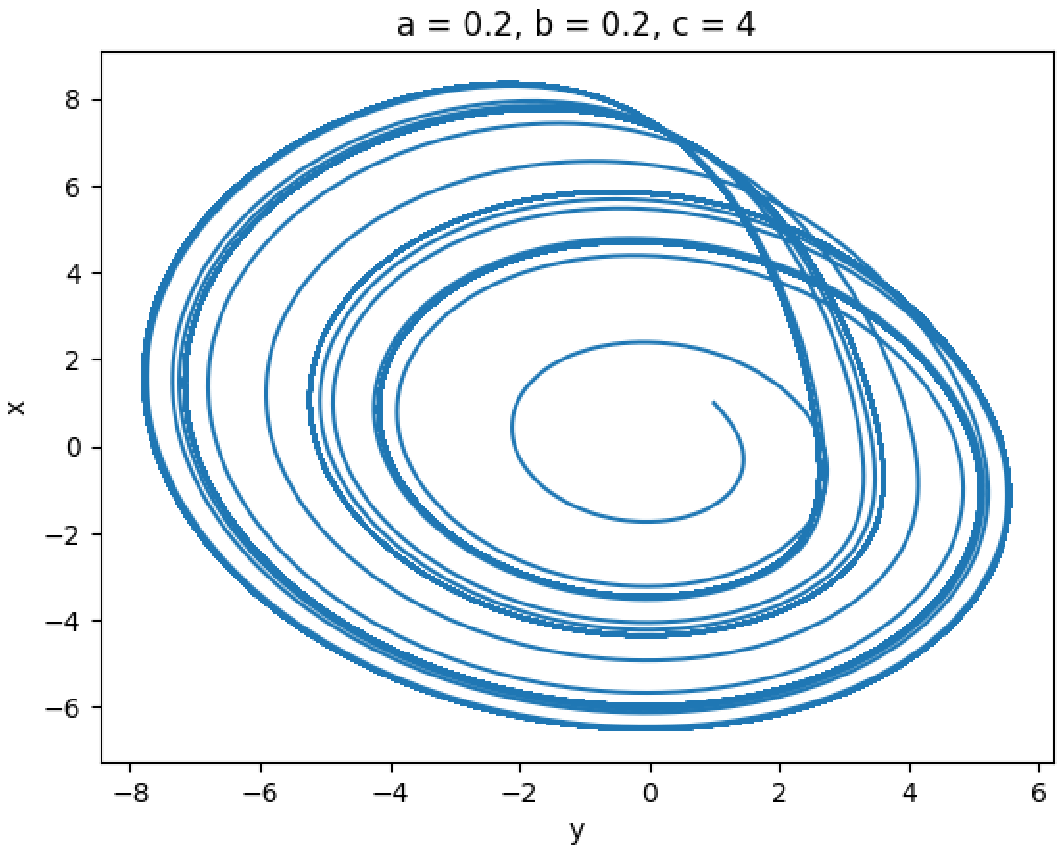

Figure 39: y versus x c = 4

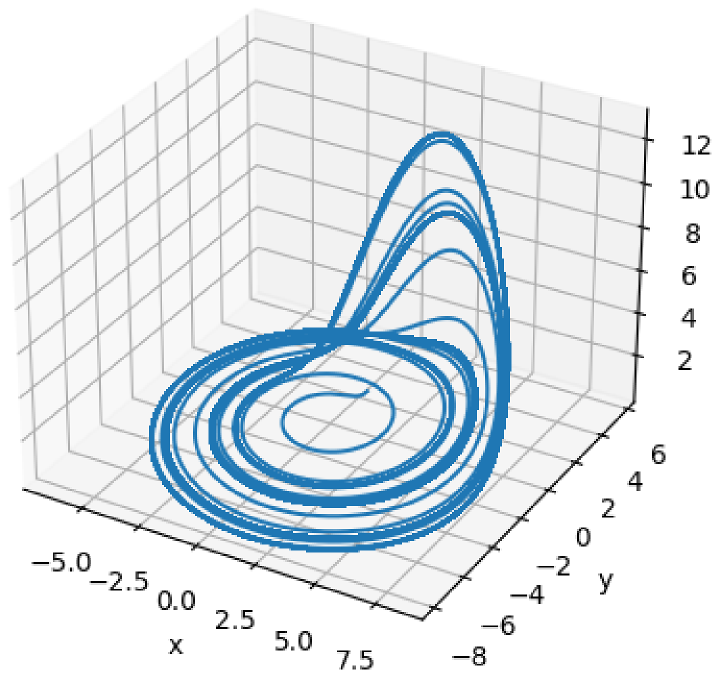

Figure 40: c = 4

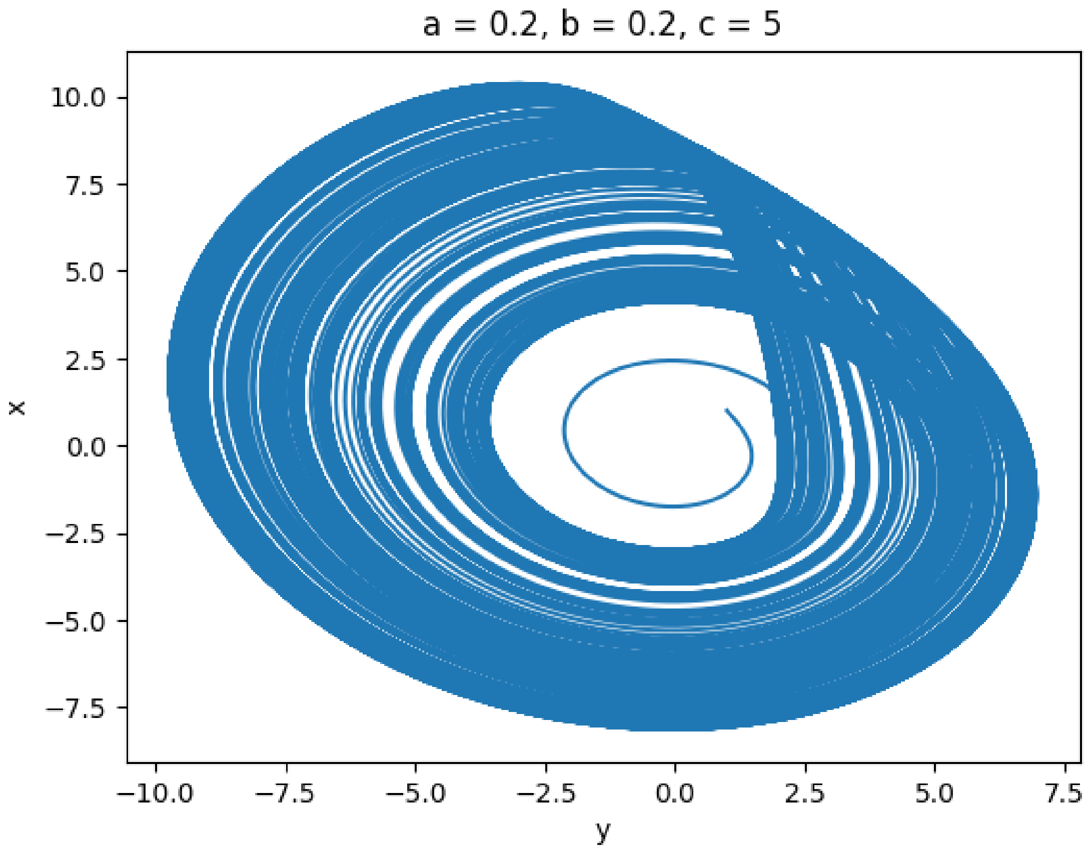

Figure 41: y versus x c = 5

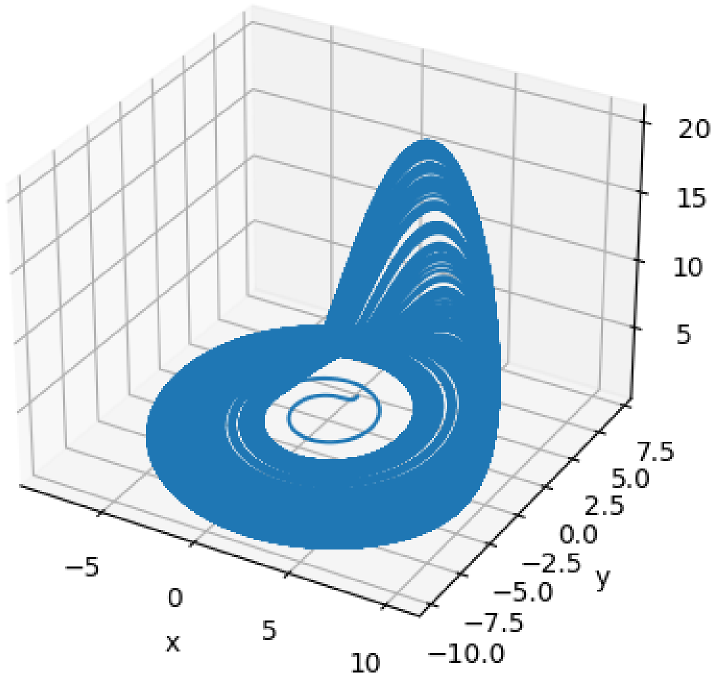

Figure 42: c = 5

As expected, when $c = 2.5$, the loop goes once before closing. When $c = 3.5$, the loop goes twice before closing. When $c = 4$, the system starts to get chaotic. Lastly, when $c = 5$, the system goes in full chaos.

2) For $c = 5$, I calculated the successive maxima of x for initial conditions $x_0 = y_0 = z_0 = 2$ and $t_0 = 0$, and I plotted the each maximum x_n and a function of the next x_{n+1}.

```python
a, b, c = 0.2, 0.2, 5
N = 10000
h = 1e-2
t0, x0, y0, z0 = 0, 2, 2, 2
t, x, y, z = Rossler_System(a, b, c, N, h, t0, x0, y0, z0)
plt.plot(t, x)
plt.xlabel("t")
plt.ylabel("x")
indices = []
max_x = []
max_t = []
for i in range(1, len(x) - 1):
    if x[i] > x[i-1] and x[i] > x[i+1]:
        indices.append(i)
        max_x.append(x[i])
        max_t.append(t[i])
xs = x[indices[0]: indices[1] + 1]
ts = t[indices[0]: indices[1] + 1]
plt.scatter(max_t, max_x, color = 'r')
plt.show()
plt.scatter(max_x[:-1], max_x[1:])
plt.xlabel("$x_n$")
plt.ylabel("$x_{n+1}$")
plt.show()
```

These are the graphs that I got,

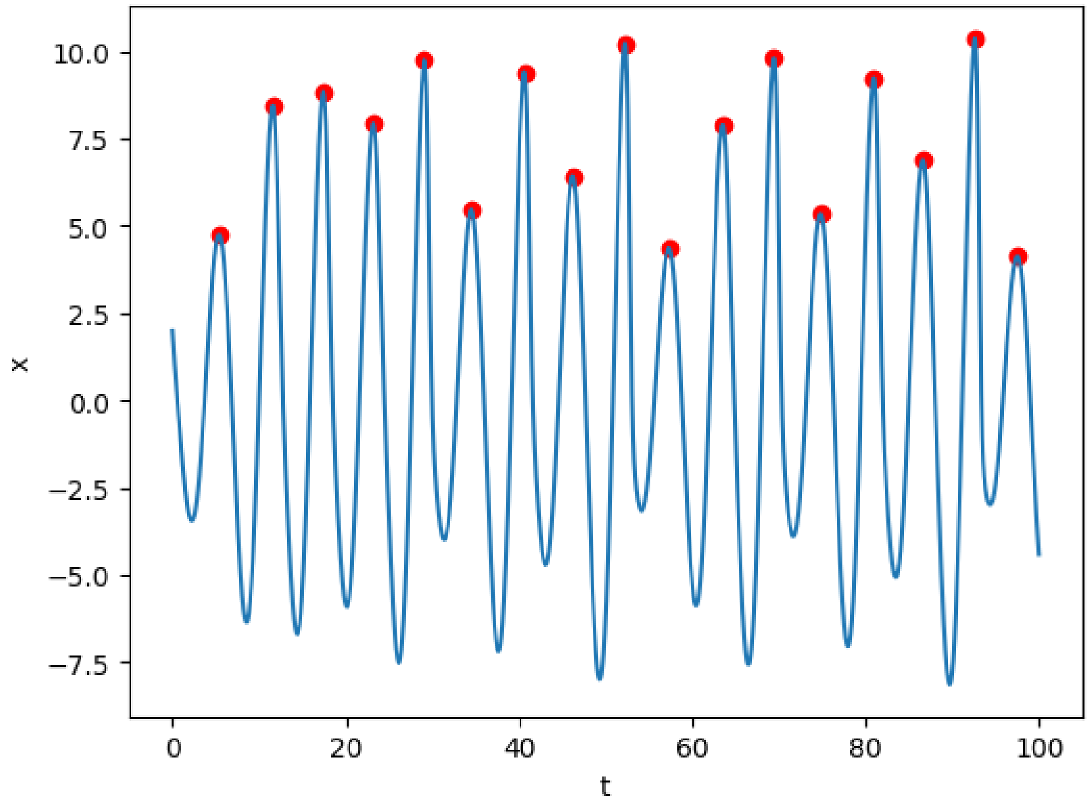

Figure 43: x as a Function of t With the Maxima Highlighted

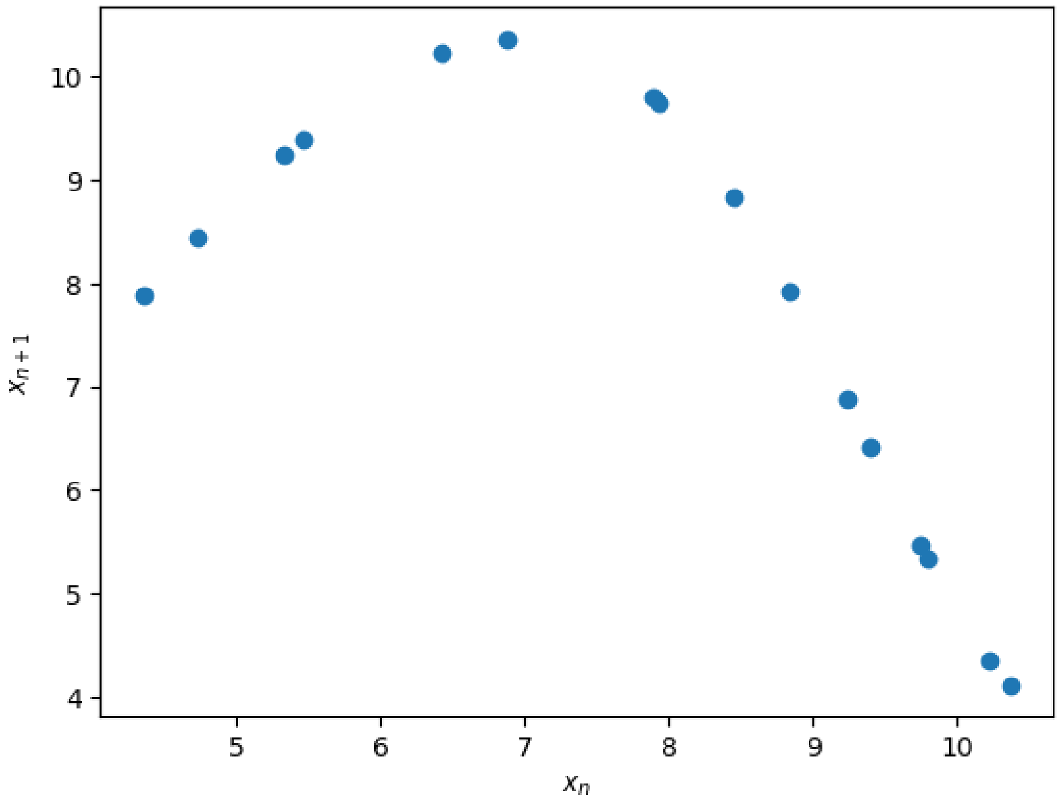

Figure 44: x_{n+1} as a function of x_n

The curve obtained by plotting each maximum as a function of the next looks like the curve of a logistic equation! Therefore, if we plot the maxima as we vary c we might get a fig tree-like pattern.

3) So, I tried to find the maxima for different values of c, and I plotted them as a function of c.

```
def maximum(x, t):
    indices = []
    max_x = []
    max_t = []
    for i in range(1, len(x) - 1):
        if x[i] > x[i-1] and x[i] > x[i+1]:
            indices.append(i)
            max_x.append(x[i])
            max_t.append(t[i])
    return max_x, max_t

a, b = 0.2, 0.2
c = [2.5, 3.5, 4, 5]
t0 = 0
x0, y0, z0 = 2, 2, 2
h = 1e-2
N = 100000
n = 100
c = np.linspace(2.5, 5, n)
cs = []
values = []
for i in range(len(c)):
    t, x, y, z = Rossler_System(a, b, c[i], N, h, t0, x0, y0, z0)
    x_max, t_max = maximum(x, t)
    cs.extend([c[i]] * (n - 10))
    values.extend(x_max[10:n])
plt.scatter(cs, values, c = 'k', marker= '.', s = 1)
plt.show()
```

I got this graph,

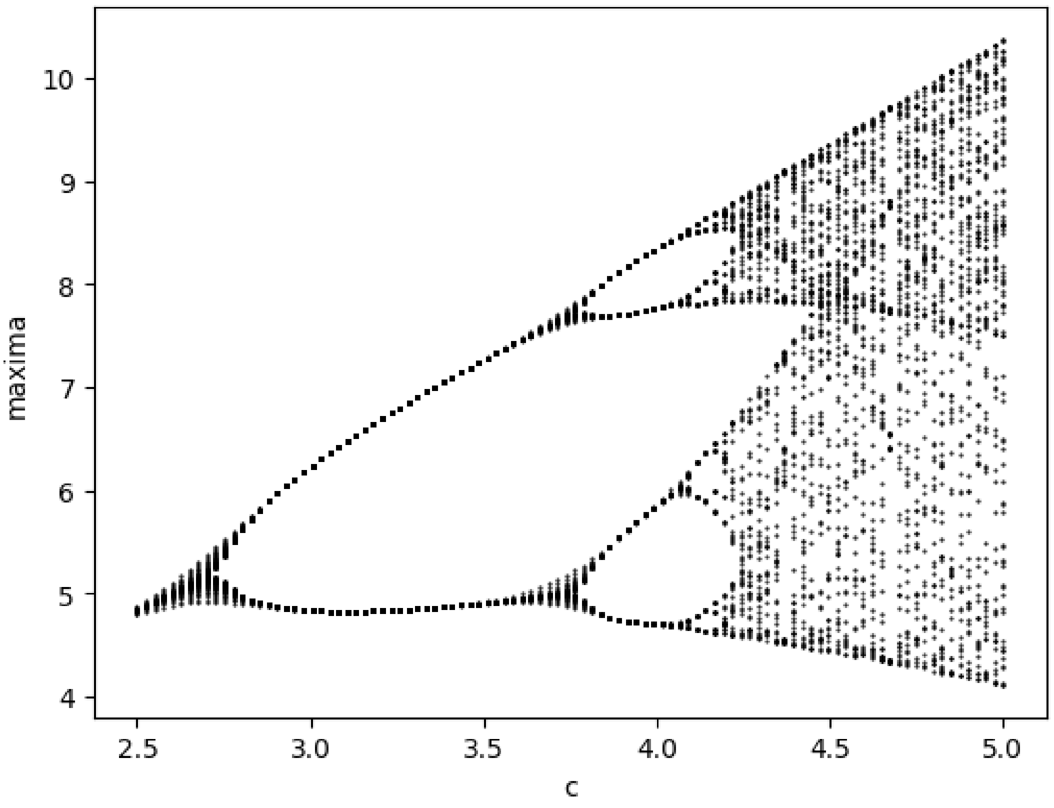

Figure 45: The Maxima as a Function of c

This is a fig tree-like pattern! As we expected from plotting the maxima against each other!

7 SVD and Lagrangian Coherent Structures

1) Given the flow is described by the stream-function:

$$\psi(x,y,t) = Asin(\pi f(t))sin(\pi y)$$
$$f(x,t) = a(t)x^2 + b(t)x$$
$$a = \epsilon sin(\omega t)$$
$$b = 1 - 2\epsilon sin(\omega t)$$

The constants are given to be $A = 0.25$, $\epsilon = 0.1$, $\omega = \frac{2\pi}{10}$, and the domain is defined over $[0,2] \times [0,1]$. The velocity field is defined as,

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

So, I generated the field for 200 time frames.

```
x = np.linspace(0, 2, N)
y = np.linspace(0, 1, M)
X,Y = np.meshgrid(x,y)
t = np.linspace(0, 100, frame)
u = np.empty((len(t), len(X), len(Y)))
v = np.empty((len(t), len(X), len(Y)))

for i in range(frame):
    u[i, :, :] = du(X, Y, t[i], A, e, w)
    v[i, :, :] = dv(X, Y, t[i], A, e, w)
```

2) I then stacked the time frames into a matrix of the dimensions of $(2 \times N \times M) \times t$).

```
D = np.zeros((2 * N * M, frame))
for i in range(frame):
    D[:N * M, i] = u[i,:,:].reshape(N * M)
    D[N * M:, i] = v[i,:,:].reshape(N * M)
```

3) I decomposed my data using the singular value decomposition.

```
U,S,V = np.linalg.svd(D, full_matrices= False)
plt.scatter(np.arange(len(S**2)), S**2)
plt.show()
```

Then, I plotted the singular values to see what the dominant ones, and I found that 2 values dominates.

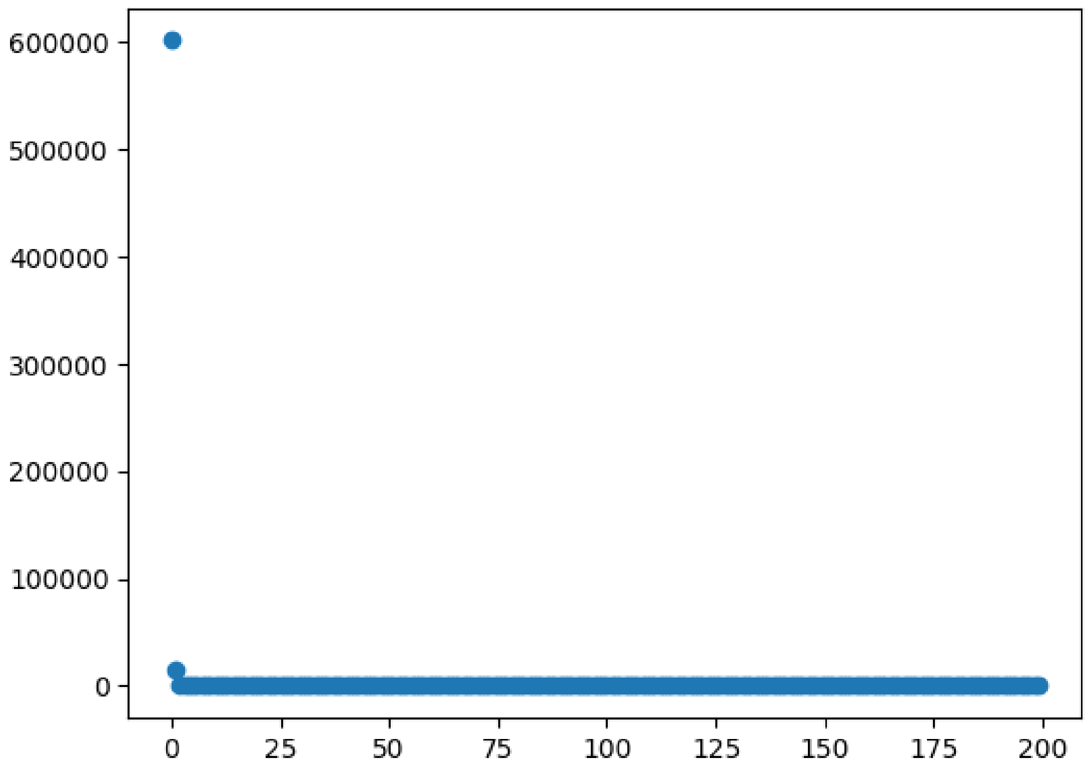

Figure 46: Singular Values

4) I used the first 5 singular values and I generated an animation that shows the evolution of the velocity fields. The animation is titled "u and v".

```
A= U[:, :6] * S[5] @ V[:6]
U = np.zeros((frame, len(X), len(Y)))
V = np.zeros((frame, len(X), len(Y)))
for i in range(frame):
    U[i, :, :] = A[:N*M, i].reshape(N, M)
    V[i, :, :] = A[N*M:, i].reshape(N, M)

fig, axs = plt.subplots(1, 2, figsize=(12, 5))

cax1 = fig.add_axes([0.06, 0.1, 0.02, 0.8])
cax2 = fig.add_axes([0.94, 0.1, 0.02, 0.8])

def animate(i):
    axs[0].clear()
    axs[1].clear()
```

```
        axs[0].set_title("u")
        axs[1].set_title('v')

        axs[0].set_xlim(X.min(), X.max())
        axs[0].set_ylim(Y.min(), Y.max())

        axs[1].set_xlim(X.min(), X.max())
        axs[1].set_ylim(Y.min(), Y.max())

        c1 = axs[0].pcolormesh(X, Y, U[i, :, :], shading='auto', cmap='viridis')
        c2 = axs[1].pcolormesh(X, Y, V[i, :, :], shading='auto', cmap='viridis')

        fig.colorbar(c1, cax=cax1)
        fig.colorbar(c2, cax=cax2)

    ani = FuncAnimation(fig, animate, frames=frame, interval=50, blit=False)
    ani.save("u and v.gif")
```

5) I calculate the finite time lyapunov exponent of the system. I first calculated the Jacobian of the velocity field and using the SVD I decomposed the Jacobian and found the eigenvalues of the Jacobian which is related to the singular values. Then, I found the FTLE of the flow and generated the Lagrangian coherent structure.

```
    FTLE = np.zeros((N, M))
    for i in range(N):
        for j in range(M):
            Lambda = np.zeros(frame)
            for k in range(frame):
                du_dx, du_dy = np.gradient(u[k, :, :], axis=(1, 0))
                dv_dx, dv_dy = np.gradient(v[k, :, :], axis=(1, 0))
                J = np.array([[du_dx[i, j], du_dy[i, j]],
                              [dv_dx[i, j], dv_dy[i, j]]])
                _, S, _ = np.linalg.svd(J)
                Lambda[k] = np.max(S)
            FTLE[i, j] = (1/200) * np.log(np.max(Lambda))

    plt.contourf(X, Y, FTLE, levels=100, cmap='jet_r')
    plt.colorbar(label='FTLE')
    plt.xlabel('X')
    plt.ylabel('Y')
    plt.title('Coherent Structures')
    plt.tight_layout()
    plt.show()
```

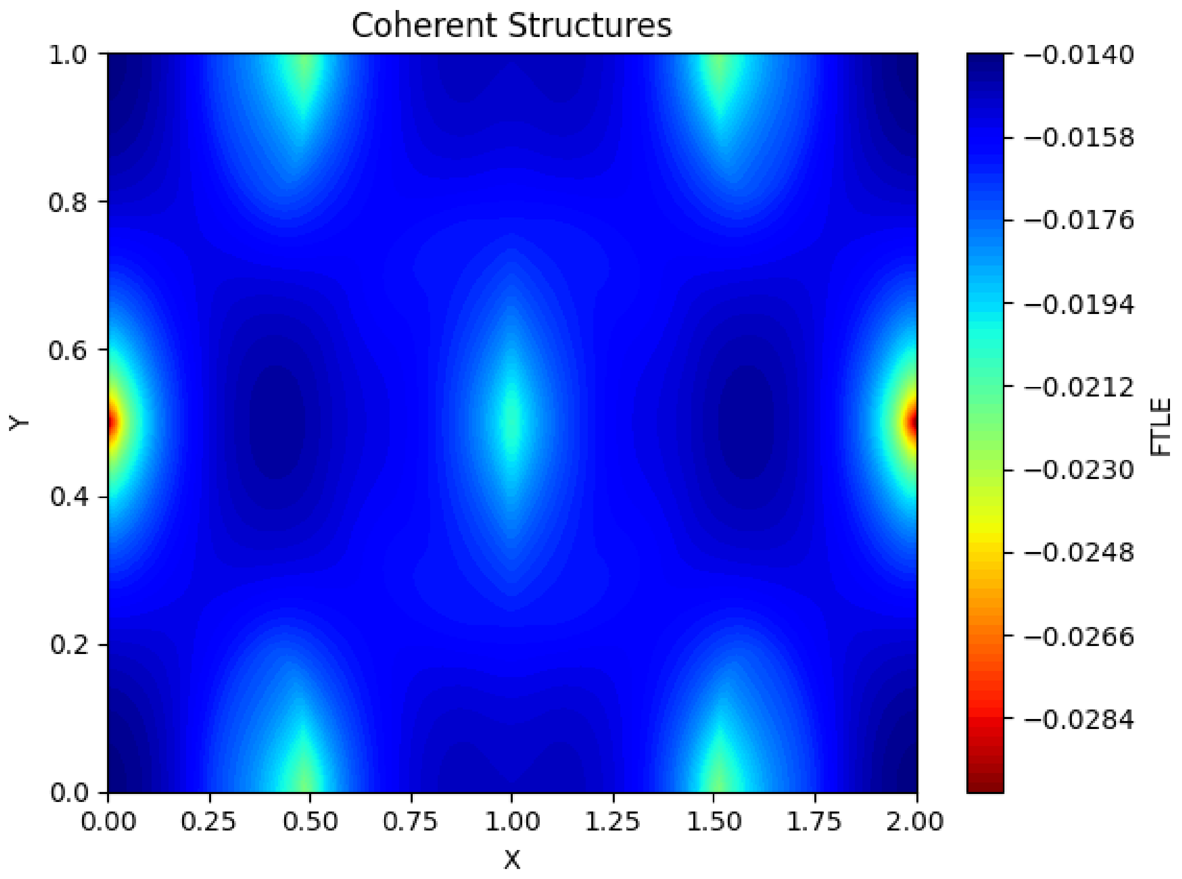

Figure 47: Coherent Structure of the Flow