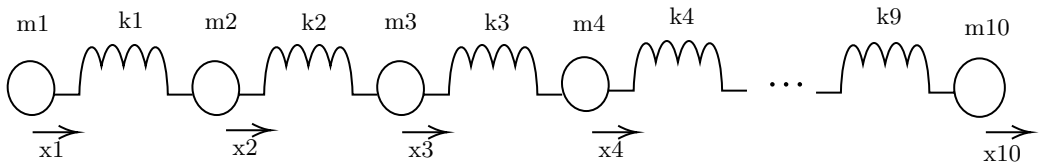
Phys 222 - HW 2

Issar Amro

Prob 1 - Network Laplacian

10 one dimensional coupled harmonic oscillators:



1. The Lagrangian functions for each mass are the following

$$\mathcal{L}_1 = \frac{1}{2}m_1\dot{x}_1^2 - \frac{1}{2}k_1(x_1 - x_2)^2$$

$$\mathcal{L}_2 = \frac{1}{2}m_2\dot{x}_2^2 - \frac{1}{2}k_2(x_2 - x_3)^2 + \frac{1}{2}k_1(x_1 - x_2)^2$$

$$\mathcal{L}_3 = \frac{1}{2}m_3\dot{x}_3 - \frac{1}{2}k_3(x_3 - x_4)^2 + \frac{1}{2}k_2(x_2 - x_3)^2$$

$$\vdots$$

$$\mathcal{L}_{10} = \frac{1}{2}m_{10}\dot{x}_{10} + \frac{1}{2}k_9(x_9 - x_{10})^2$$

Applying the Euler-Lagrange equation on each Lagrangian function, we get 10 equations of motion for the 10 masses:

$$m_1\ddot{x}_1 = k_1(x_1 - x_2)$$

$$m_2\ddot{x}_2 = k_2(x_2 - x_3) - k_1(x_1 - x_2)$$

$$m_3\ddot{x}_3 = k_3(x_3 - x_4) - k_2(x_2 - x_3)$$

$$\vdots$$

$$m_{10}\ddot{x}_{10} = -k_9(x_9 - x_{10})$$

2. The equations of motion can be written in the form $\ddot{x} = -Lx$. Taking all the masses to be equal such that $m_1 = m_2 = \cdots = m$, the Laplacian matrix is then:

$$L = \begin{pmatrix} \frac{k_1}{m} & -\frac{k_1}{m} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -\frac{k_1}{m} & \frac{(k_1+k_2)}{m} & -\frac{k_2}{m} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -\frac{k_2}{m} & \frac{(k_2+k_3)}{m} & -\frac{k_3}{m} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -\frac{k_3}{m} & \frac{(k_3+k_4)}{m} & -\frac{k_4}{m} & 0 & 0 & 0 & 0 & 0 \\ \vdots & & & & & & & & & \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -\frac{k_9}{m} & \frac{k_9}{m} \end{pmatrix}$$

Taking all spring constants to be equal i.e $k_1 = k_2 = \cdots = k$

$$L = \frac{k}{m} \begin{pmatrix} 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 2 & -1 & 0 & 0 & 0 & 0 & 0 \\ \vdots & & & & & & & & & \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 \end{pmatrix}$$

Applying the L matrix on the identity vector v_1, we get the 0 vector since the sum of the entries of each row in L is = 0. Hence the eigenvalue λ_1 corresponding to the eigenvector v_1 is = 0.
Finding the eigenvalues of L using sympy

```
k, m = symbols('k m')
L = sp.zeros(10,10)
L[0] = [k/m,-k/m,0,0,0,0,0,0,0,0]
L[9] = [0,0,0,0,0,0,0,0,-k/m,k/m]
for i in range(1, 9):
    row = []
    for j in range(0,i-1):
        row.append(0)
    row.append(-k/m)
    row.append(2*k/m)
    row.append(-k/m)
    for o in range(0, 10 - len(row)):
        row.append(0)
    L[i] = row
eigenvalues = L.eigenvals()
eigenvectors = L.eigenvects()
fiedler_vector = eigenvectors[1][2][0]
```

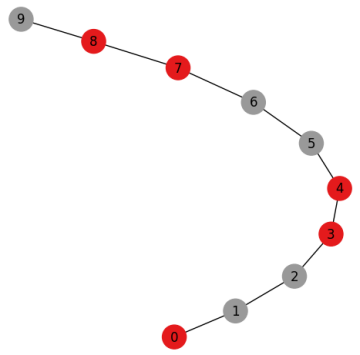
we get the second eigenvalue $\lambda_2 = \frac{2k}{m}$ corresponding to the eigenvector

$$v_2 = [-1\ 1\ 1 - 1 - 1\ 1\ 1 - 1 - 1\ 1]$$

which is the Fiedler vector with half of its entries positive and the other half negative, as expected (since $v1 \cdot v2 = 0$). We also can see how the first (zero) eigenvalue corresponds to the unit vector. Since L is a positive semi-definite symmetric matrix, the equation $\ddot{x} + Lx = 0$ has a complex solution of the form $x(t) = Ae^{i\sqrt{\lambda}t} + Be^{-i\sqrt{\lambda}t}$ which describes an oscillatory behavior.

• Building a newtork consisting of these oscillators where each 2 are coupled and using the Fiedler eigenvector to identify communities in the network, I got this graph

```
G = nx.Graph()
n = L.shape[0]
G.add_nodes_from(range(n))
for i in range(n):
    for j in range(i + 1, n):
        if L[i, j] != 0:
            G.add_edge(i, j)
community_assignment = [0 if val < 0 else 1 for val in fiedler_vector]
pos = nx.spring_layout(G)
plt.figure(figsize=(5, 5))
nx.draw(G, pos, with_labels=True,
node_color=community_assignment, cmap=plt.cm.Set1, node_size=500)
```

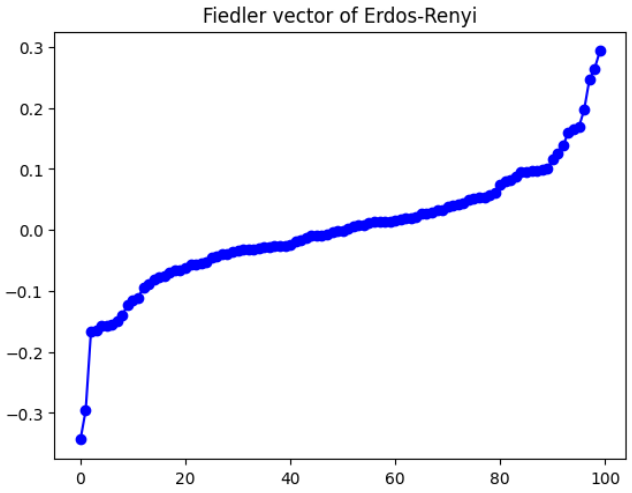


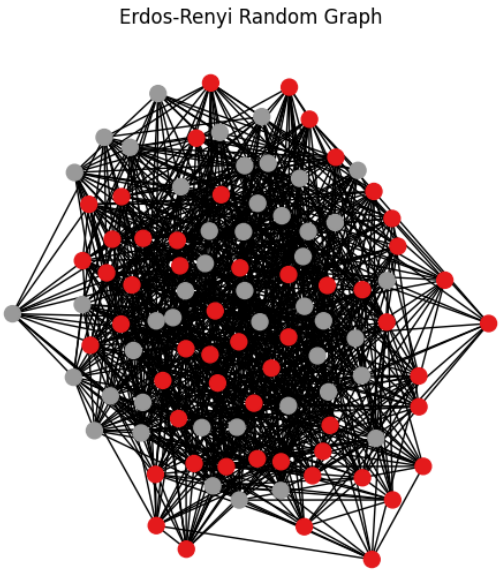
We see how every 2 consecutive nodes are connected by an edge signifying their coupling and are color coded based on the community they belong to. The red nodes/oscillators move in phase with each other and out of phase with the grey ones, they all have the same magnitude of the amplitude of oscillations and the same frequency $\frac{2k}{m}$ but oscillate along an opposite direction to that of the grey ones.

3. Erdos-Renyi random network: I generated a random network with (N = 100, p = 0.2). After obtaining the Laplacian matrix of the network, I generated its eigenvalues and got the

eigenvector corresponding to the second eigenvalue, the Fiedler vector. Setting a threshold to be 0, the values in the Fiedler vector below the threshold belong to one community and the values lying above the threshold belong to the second community. Color coding the nodes based on their community assignment results in this graph

```
randgraph = nx.erdos_renyi_graph(100, 0.2)
L1 = nx.laplacian_matrix(randgraph, nodelist=None, weight='weight').toarray()
eig1, eigv1 = np.linalg.eig(L1)
plt.plot(sorted(eigv1[1]), marker='o', linestyle='-', color='b')
community_assignment1 = [0 if val < 0 else 1 for val in eigv1[1]]
pos1 = nx.spring_layout(randgraph)
plt.figure(figsize=(5, 5))
nx.draw(randgraph, pos1, with_labels=False, node_color=community_assignment1,
cmap=plt.cm.Set1, node_size=100)
```





I used networkx built-in functions, I got some properties of this network

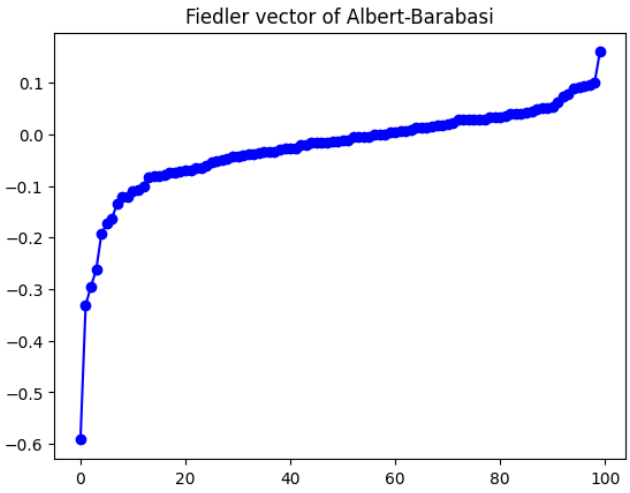
```
degree_centrality1 = nx.degree_centrality(randgraph)
print("Average Path Diff = ", nx.average_shortest_path_length(randgraph))
print("Average Clustering = ", nx.average_clustering(randgraph))
print("Node Degree Centrality:", degree_centrality1)
```

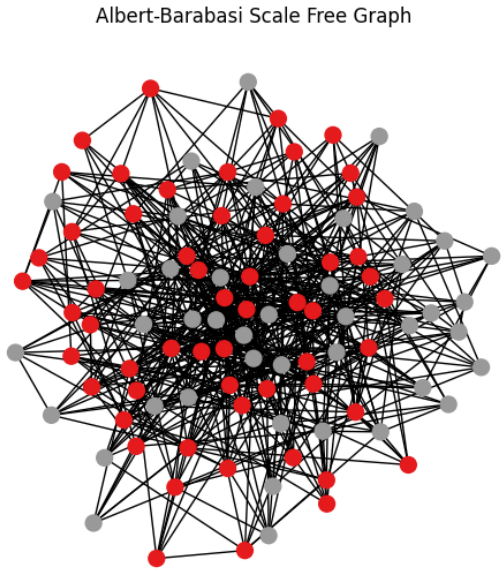
- Average Path Diff = 1.7995959595959596
- Clustering Coefficient = 0.2033974012439553
- The degree centrality of each node, for the first 6 nodes as an example:

$$[0 : 0.21; 1 : 0.17; 2 : 0.22; 3 : 0.21; 4 : 0.17; 5 : 0.22]$$

Albert-Barabasi Scale Free Network: Doing the same steps as for the random graph,

```
saclefreegraph = nx.barabasi_albert_graph(100, 7)
L2 = nx.laplacian_matrix(saclefreegraph, nodelist=None, weight='weight').toarray()
eig2, eigv2 = np.linalg.eig(L2)
plt.plot(sorted(eigv2[1]), marker='o', linestyle='-', color='b')
community_assignment2 = [0 if val < 0 else 1 for val in eigv2[1]]
pos2 = nx.spring_layout(saclefreegraph)
plt.figure(figsize=(5, 5))
nx.draw(saclefreegraph, pos2, with_labels=False, node_color=community_assignment2,
cmap=plt.cm.Set1, node_size=100)
```





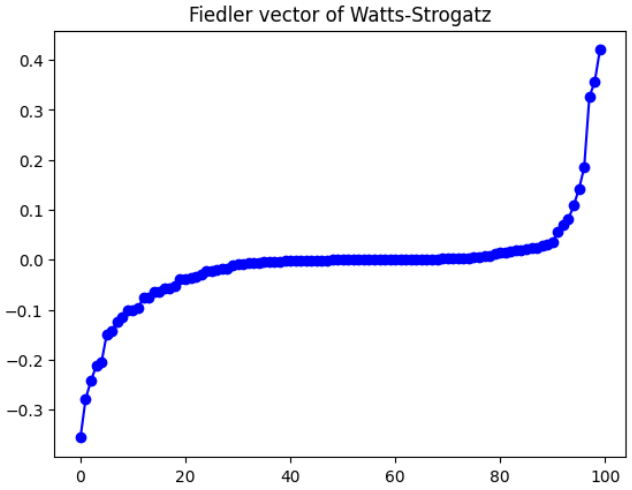

```
degree_centrality2 = nx.degree_centrality(saclefreegraph)
print("av path diff = ", nx.average_shortest_path_length(saclefreegraph))
print("av clustering = ", nx.average_clustering(saclefreegraph))
print("Degree Centrality:", degree_centrality2)
```

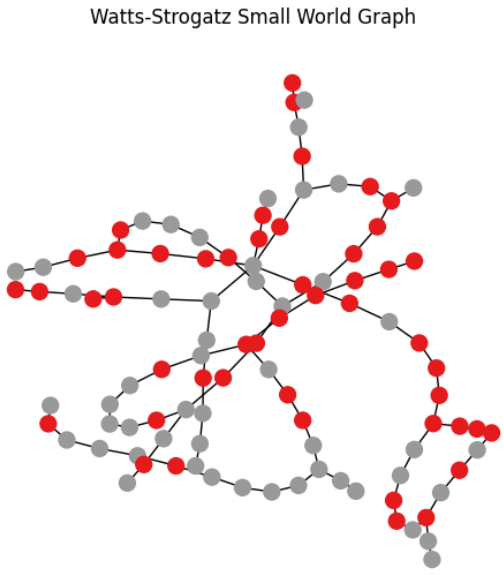
- Average Path Diff = 2.0238383838383838
- Clustering Coefficient = 0.2114080759191525
- The degree centrality of each node example:

$$[0 : 0.5; 1 : 0.05; 2 : 0.24; 3 : 0.23; 4 : 0.23; 5 : 0.15]$$

Watts-Strogatz Small World Network:

```
smallworld = nx.watts_strogatz_graph(100, 3, 0.2, seed=None)
L3 = nx.laplacian_matrix(smallworld, nodelist=None, weight='weight').toarray()
eig3, eigv3 = np.linalg.eig(L3)
plt.plot(sorted(eigv3[1]), marker='o', linestyle='-', color='b')
community_assignment3 = [0 if val < 0 else 1 for val in eigv3[1]]
pos3 = nx.spring_layout(smallworld)
plt.figure(figsize=(5, 5))
nx.draw(smallworld, pos3, with_labels=False, node_color=community_assignment3,
cmap=plt.cm.Set1, node_size=100)
```






```
degree_centrality3 = nx.degree_centrality(smallworld)
```

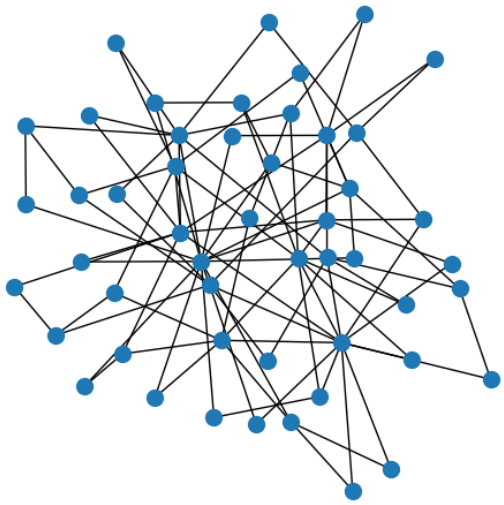
```
print("av path diff= ", nx.average_shortest_path_length(smallworld))
print("av clustering = ", nx.average_clustering(smallworld))
print("Degree Centrality:", degree_centrality3)
```

- Average Path Diff = 16.76060606060606
- Clustering Coefficient = 0
- The degree centrality of each node example:

$$[0 : 0.03; 1 : 0.01; 2 : 0.03; 3 : 0.02; 4 : 0.02; 5 : 0.02]$$

The clustering coefficient being 0 here may be due to the random rewiring of the edges; for moderate or big p values, the network becomes more random and loses its initial clustering.
note. Doing the same for the three networks setting N = 1000 now, we notice that for the random network the values of the average path difference, clustering coefficient, and node degree centrality don't vary much. For the scale free graph, the value of the average path difference increases while the clustering coefficient drops (significantly) and the node degree centrality drops (on average). For the small world, the average path difference increases significantly while the node degree centrality drops significantly (on average) all while the clustering coefficient remains 0.
Now, generating a scale free network with (N = 50, m = 2)

```
scalefree = nx.barabasi_albert_graph(50, 2)
L = nx.laplacian_matrix(scalefree, nodelist=None, weight='weight').toarray()
eig, eigv = np.linalg.eig(L)
pos = nx.spring_layout(scalefree)
plt.figure(figsize=(5, 5))
nx.draw(scalefree, pos, with_labels=False, node_size=100)
```

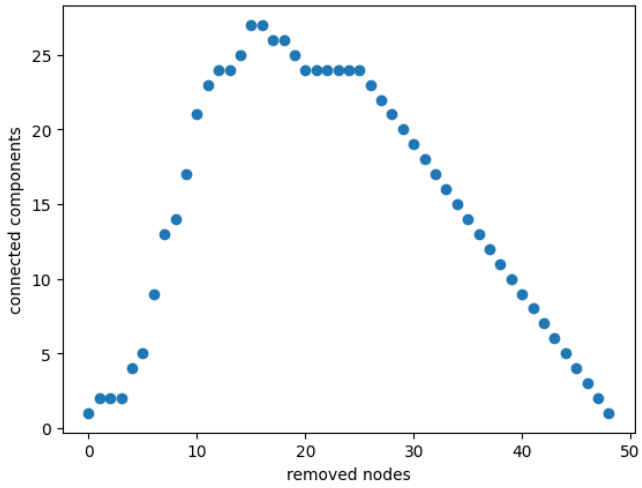


Noting that the number of 0 eigenvalues for a given scale free network corresponds to number of connected components in the network.

Removing nodes based on their degree in descending order:

```
degrees = nx.degree(scalefree)
sorted_nodes = sorted(degrees, key=lambda x: x[1], reverse=True)
nb_of_connected_nodes = []
for node, _ in sorted_nodes:
    scalefree.remove_node(node)
    if not scalefree:
        break
    L = nx.laplacian_matrix(scalefree, nodelist=None, weight='weight').toarray()
    eig, eigv = np.linalg.eig(L)
    nb_of_connected_nodes.append(len(eig[eig < 1e-10]))
```

The number of connected components started by 1 (the initial network) and kept growing (after removing 1 node at a time) reaching a maximum then decreased 1 value at a time (all the nodes are now disconnected) until it reached 1 (the last node). Plotting the number of connected components as function of the number of removed nodes:

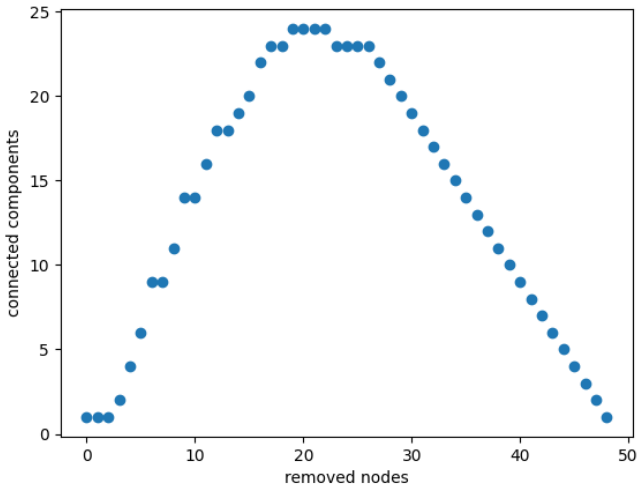


Removing nodes based on their centrality in descending order:

```
degree_centrality = nx.degree_centrality(scalefree)
sorted_nodes = sorted(degree_centrality.items(), key=lambda x: x[1], reverse=True)
nb_of_connected_nodes = []
for node, _ in sorted_nodes:
    scalefree.remove_node(node)
    if not scalefree:
        break
```

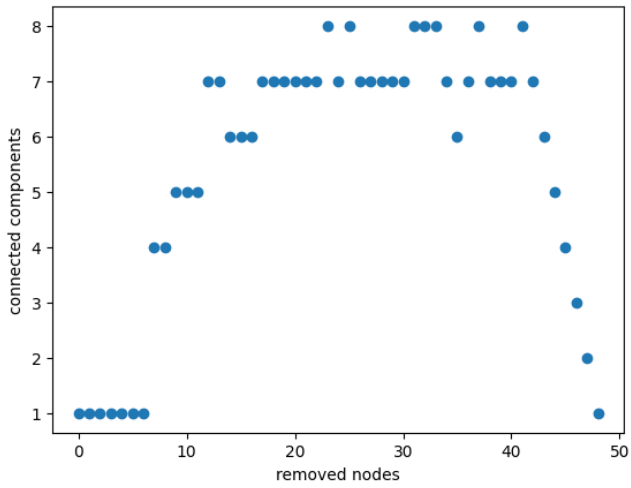
```
    L = nx.laplacian_matrix(scalefree, nodelist=None, weight='weight').toarray()
    eig, eigv = np.linalg.eig(L)
    nb_of_connected_nodes.append(len(eig[eig < 1e-10]))
```

We notice a similar behavior as for when removing the nodes based on their centrality but now the curve is smoother:



Removing nodes randomly:

```
N = 50
nb_of_connected_nodes = []
for i in range(50):
    if len(scalefree) > 0:
        random_number = np.random.choice(list(scalefree.nodes()), 1, replace=False)[0]
        scalefree.remove_node(random_number)
    if not scalefree:
        break
    L = nx.laplacian_matrix(scalefree, nodelist=None, weight='weight').toarray()
    eig, eigv = np.linalg.eig(L)
nb_of_connected_nodes.append(len(eig[eig < 1e-10]))
```



Here, it seems that the number of connected nodes tends to not change much after each node removal, and oscillates until reaching a threshold where all nodes become disconnected then starts decreasing 1 value at a time until reaching 1.
The most efficient way in terms of regularity/predictability seems to be removing nodes based on the nodes' centrality since the curve it gave us is smooth and easy to predict.

Prob 2 - Tight-Binding

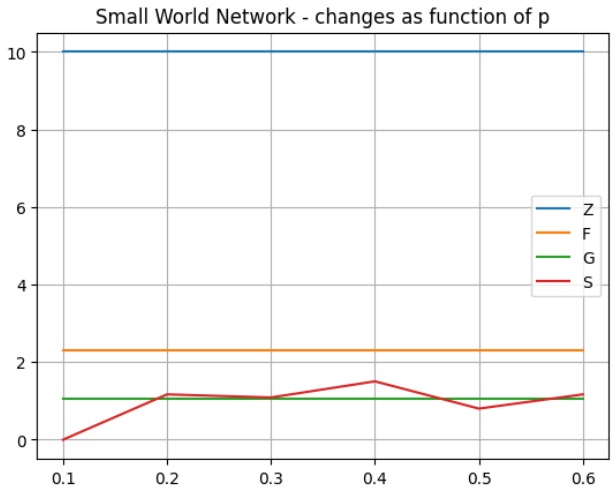
The energy levels of the system are given by the eigenvalues of the adjacency matrix of the network and are of the form $E_j = \alpha + \beta\lambda_j$. Taking $\alpha = 0, \beta = -1$, they become $E_j = -\lambda_j$ which means, the Hamiltonian now is proportional to the adjacency matrix such that

$$H = -D$$

$\beta = -1$ having the meaning of $\frac{1}{K_b T}$ gives us $K_b T = -1$.
• The partition function expression is given by $Z = tr(e^{\beta H}) = tr(e^{-H}) = tr(e^{D})$.
• The entropy expression is given by $S = -\Sigma_i p_i ln(p_i)$ where p_i is the probability distribution for the degree of each node, for Watts-Strogatz graph, this probability arises from the probability of rewiring (p). For Erdos-Renyi graph, it arises from the probability of edge creation (p). And for random regular graph, it is related to the degree of each node (d) divided by the number of nodes ($\frac{d}{N}$) which represents the degree distribution probability.
• The Helmholtz free energy can be found using $F = -KTln(z) = ln(Z)$.
• The Gibbs free energy can be found using the discrete method $G = N\mu = N\frac{dF}{dN} = N(F(N) - F(N-1))$.
Now, we generate the three graphs for different p values, and use their adjacency matrices to recover Z, G, and F. We recover the degree distribution by generating the degree distribution histogram array and dividing the elements by the number of nodes.

Watts-Strogatz small world graph:

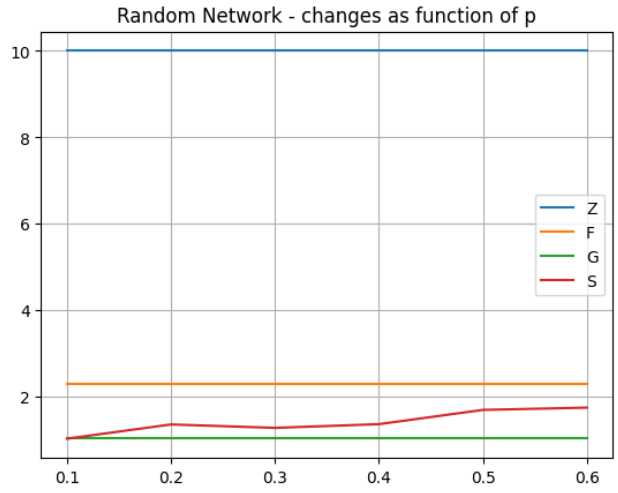
```
p_vals = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6]
N_vals = [9, 10]
S = []
F = []
G = []
Z = []
for p in p_vals:
  F1 = []
  for N in N_vals:
     smallworld = nx.watts_strogatz_graph(N, 5, p, seed=None)
     A = nx.adjacency_matrix(smallworld, nodelist=None, dtype=None, weight='weight').toarray()
     Z1 = np.trace(np.exp(A))
     F1.append(np.log(Z1))
  Z.append(Z1)
  F.append(F1[1])
  G.append((F1[1] - F1[0])*N_vals[1])
  degree_histogram = nx.degree_histogram(smallworld)
  prob_distribution = [count / N for count in degree_histogram if count > 0]
  S.append(-np.sum([pr * np.log(pr) for pr in prob_distribution if pr > 0]))
```



Erdos-Renyi random graph:

```
p_vals = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6]
N_vals = [9, 10]
S = []
```

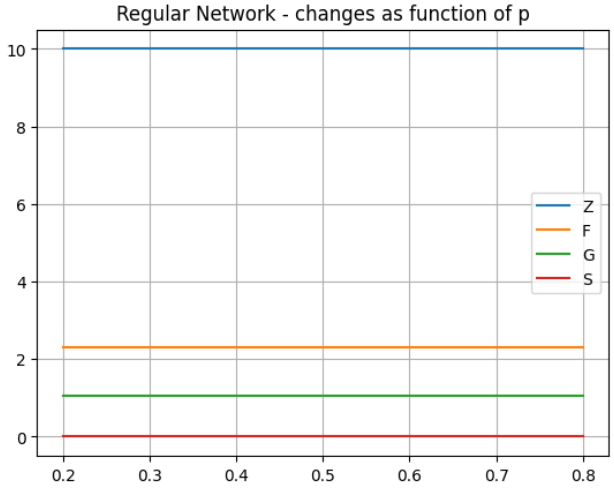
```
F = []
G = []
Z = []
for p in p_vals:
  F1 = []
  for N in N_vals:
     Egraph = nx.erdos_renyi_graph(N, p)
     A = nx.adjacency_matrix(Egraph, nodelist=None, dtype=None, weight='weight').toarray()
     Z1 = np.trace(np.exp(A))
     F1.append(np.log(Z1))
  Z.append(Z1)
  F.append(F1[1])
  G.append((F1[1] - F1[0])*N_vals[1])
  degree_histogram = nx.degree_histogram(Egraph)
  prob_distribution = [count / N for count in degree_histogram if count > 0]
  S.append(-np.sum([pr * np.log(pr) for pr in prob_distribution if pr > 0]))
```



Regular graph:

```
d_vals = [2, 4, 6, 8]
N_vals = [9, 10]
p_vals = np.zeros(len(d_vals))
# p = d/N
N = 10
S = []
F = []
G = []
Z = []
```

```
for d in d_vals:
  F1 = []
  for N in N_vals:
     Rgraph = nx.random_regular_graph(d, N)
     A = nx.adjacency_matrix(Rgraph, nodelist=None, dtype=None, weight='weight').toarray()
     Z1 = np.trace(np.exp(A))
     F1.append(np.log(Z1))
  Z.append(Z1)
  F.append(F1[1])
  G.append((F1[1] - F1[0])*N_vals[1])
  degree_histogram = nx.degree_histogram(Rgraph)
  prob_distribution = [count / N for count in degree_histogram if count > 0]
  S.append(-np.sum([pr * np.log(pr) for pr in prob_distribution if pr > 0]))
for i in range(len(d_vals)):
   p_vals[i] = d_vals[i]/N
```



We notice that, for the three graphs, the values of Z, F, and G are fixed for all p values since these quantities depend only on the number of nodes. S, however, depends on the degree probability of the nodes and changes as we change p. What is peculiar about the S quantity is it being 0 for the regular network for all values of p. The regular graph generator takes as input the degree of each node and generates a network with N number of modes with the same degree each. This renders the problem deterministic rather than probabilistic and i.e. at each p value, we are certain of the state (degree) of each node, and hence entropy is zero.

Prob 3 - KBZ Universality Class And The Wigner Semi Circle

1. After the orthogonal transformation, X becomes

$$X = \begin{bmatrix} x_{11} & x_{12} \\ x_{12} & x_{22} \end{bmatrix} = \begin{bmatrix} \lambda_1 \cos\theta^2 + \lambda_2 \sin\theta^2 & \cos\theta \sin\theta(\lambda_1 - \lambda_2) \\ \cos\theta \sin\theta(\lambda_2 - \lambda_1) & \lambda_1 \sin\theta^2 + \lambda_2 \cos\theta^2 \end{bmatrix}$$

The Jacobian matrix is give by

$$J = \begin{bmatrix} \frac{\partial x_{11}}{\partial \lambda_1} & \frac{\partial x_{22}}{\partial \lambda_1} & \frac{\partial x_{12}}{\partial \lambda_1} \\ \frac{\partial x_{11}}{\partial \lambda_2} & \frac{\partial x_{22}}{\partial \lambda_2} & \frac{\partial x_{12}}{\partial \lambda_2} \\ \frac{\partial x_{11}}{\partial \theta} & \frac{\partial x_{22}}{\partial \theta} & \frac{\partial x_{12}}{\partial \theta} \end{bmatrix}$$

Evaluating it for the values we got after the orthogonal transformation, we get

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

Introducing the symbols $(\lambda_1, \lambda_2, \theta)$ and finding the determinant symbolically, we get $det(J) = \lambda_1 - \lambda_2$.

```
theta, l1, l2 = sp.symbols('theta l1 l2')

J = sp.Matrix([[sp.cos(theta)**2, sp.sin(theta)**2, sp.cos(theta)*sp.sin(theta)],
               [sp.sin(theta)**2, sp.cos(theta)**2, - sp.cos(theta)*sp.sin(theta)],
               [2*sp.sin(theta)*sp.cos(theta)*(l2 - l1),
               2*sp.sin(theta)*sp.cos(theta)*(l1 - l2),
               sp.cos(theta)**2*(l1- l2) + sp.sin(theta)**2*(l2-l1)]])

dete = J.det()
print(dete)

expression = l1*sp.sin(theta)**6 + 3*l1*sp.sin(theta)**4*sp.cos(theta)**2 +
3*l1*sp.sin(theta)**2*sp.cos(theta)**4 + l1*sp.cos(theta)**6 - l2*sp.sin(theta)**6
- 3*l2*sp.sin(theta)**4*sp.cos(theta)**2 - 3*l2*sp.sin(theta)**2*sp.cos(theta)**4 -
l2*sp.cos(theta)**6

simplified_expression = sp.simplify(expression)
```

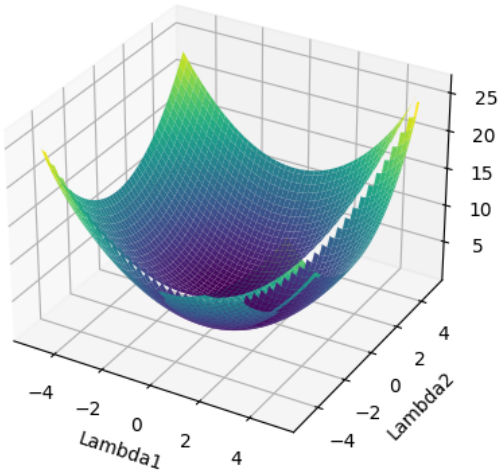
2. For the system we are studying, having 2 eigenvalues, the Hamiltonian is given by

$$H = \frac{1}{2}(\lambda_1^2 + \lambda_2^2) - ln|\lambda_1 - \lambda_2|$$

The Hamiltonian represents the effective potential term where the first, quadratic, term is the

familiar potential energy term which is a paraboloid in 3D. The second term is a logarithmic correction to the potential energy that shifts the paraboloid downwards. 3D plotting the Hamiltonian as a function of the eigenvalues, setting their range [-5,5], we get

```
def hamiltonian(lambda1, lambda2):
    return 0.5 * (lambda1**2 + lambda2**2) - np.log(np.abs(lambda1 - lambda2))
lambda1 = np.linspace(-5, 5, 100)
lambda2 = np.linspace(-5, 5, 100)
lambda1, lambda2 = np.meshgrid(lambda1, lambda2)
H = hamiltonian(lambda1, lambda2)
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
surface = ax.plot_surface(lambda1, lambda2, H, cmap='viridis')
```



The eigenvalues represent the energy states of the system, and we would expect the behavior of the system to be around its equilibrium state / ground state of the minimum potential energy. From the plot, we see that the region around the minimum is the region where $\lambda_1 \approx \lambda_2$ and their values are around 0.
note. The 2 eigenvalues cannot be equal or we will get an undefined value for the logarithm.

3. Generating an Erdos-Renyi random network with N = 1000 and p = 0.5, calculating $r = \sqrt{Np(1-p)}$, we get $r = 15.8113883$. I found the eigenvalues of the adjacency matrix of the network and stored them in an array. Almost all the eigenvalues lie in the range [-2r, 2r] according to Wigner's semi-circle law. And within this range, the density of the eigenvalues is of the form

$$\rho(\lambda) = \frac{\sqrt{4-\lambda^2}}{2\pi}$$

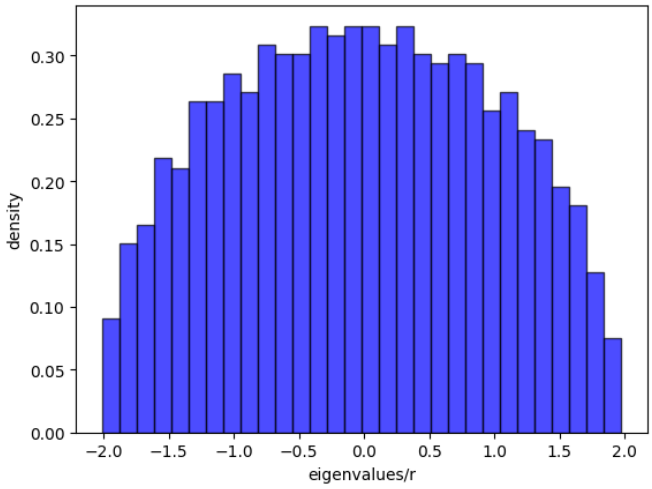
To recover this equation of the density and verify Wigner's semi-circle law, we use the polynomial

in λ that we can get from the density expression

$$\rho^2 = a + b\lambda^2$$

where $a = \frac{1}{\pi^2}$ and $b = \frac{1}{4\pi^2}$. The aim is to recover these coefficients of this polynomial using the data from the network.
First, I divided the eigenvalues array by r to represent the eigenvalues whose concentration is on [-2,2] on a density histogram getting

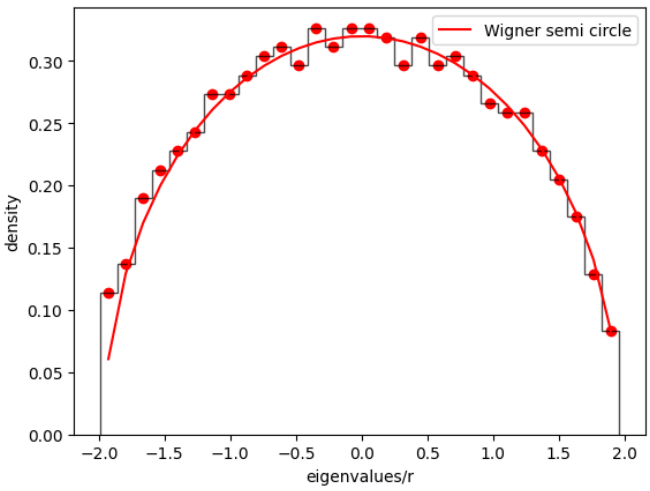
```
N = 1000
p = 0.5
Egraph = nx.erdos_renyi_graph(N, p)
pos = nx.spring_layout(Egraph)
plt.figure(figsize=(5, 5))
nx.draw(Egraph, pos)
plt.title("Erdos-Renyi Network")
plt.show()
A = nx.adjacency_matrix(Egraph).toarray()
eigs = np.linalg.eigvals(A)
# r = 15.8113883 for N = 1000
eigs = eigs / 15.8113883
for i in eigs:
    if i > 30 or i < -30:
        eigs = np.delete(eigs, np.where(eigs == i))
```



This is exactly what we expected. Now, to verify the law, we store the y values (the densities) and the bin edges of the histogram, the x values (the scaled eigenvalues) are recovered by finding the bin midpoints. So now, having ρ^2 and λ^2, I used the least square fitting to find the coefficients

of the quadratic polynomial and got $(a = 0.1, b = -0.02)$ which matches the actual coefficients. Plotting the curve of ρ as function of the scaled eigenvalues, we get

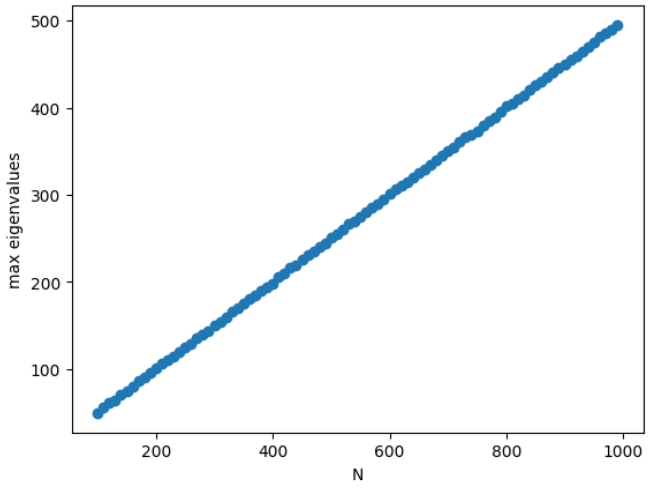
```
y_values, bin_edges, _ = plt.hist(eigs, bins=30, density=True, alpha=0.7, color='blue',
edgecolor='black', histtype="step")
x_values = (bin_edges[1:] + bin_edges[:-1]) / 2
x = x_values
y = y_values**2
order = 2
n = len(x)
v = np.zeros((n, order+1))
for i in range(n):
    row = []
    for j in range(order+1):
        row.append(x[i]**2)
    v[i] = row
exp = []
for i in range(order, -1, -1):
    exp.append(i)
v = v ** exp
v_new = np.transpose(v) @ v
y_matrix = np.transpose(v) @ y
coef_matrix = np.linalg.solve(v_new, y_matrix)
p = 0
for i in range(order):
    p += coef_matrix[len(coef_matrix) - (i+1)]* x**(2*i)
```



Hence, the scaled eigenvalues do follow the Wigner's semi-circle law distribution.

4. I found the maximum eigenvalues of the adjacency matrix of Erdos Renyi graphs corresponding to different number of nodes and got a linear relation between λ_{max} and N such that $\lambda_{max} = aN$

```
N_vals = np.arange(100, 1000, 10)
p = 0.5
max_eig = []
for N in N_vals:
    Egraph = nx.erdos_renyi_graph(N, p)
    A = nx.adjacency_matrix(Egraph).toarray()
    eigs = np.linalg.eigvals(A)
    max_eig.append(np.max(eigs))
```



Prob 5 - Dynamics

1. Given the Lorenz system:

$$\dot{x} = \sigma(y - x)$$

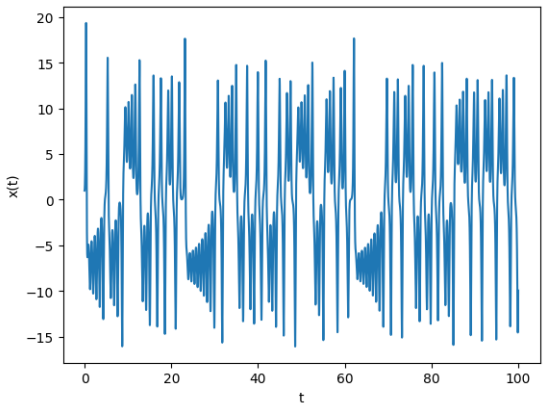
$$\dot{y} = rx - y - xz$$

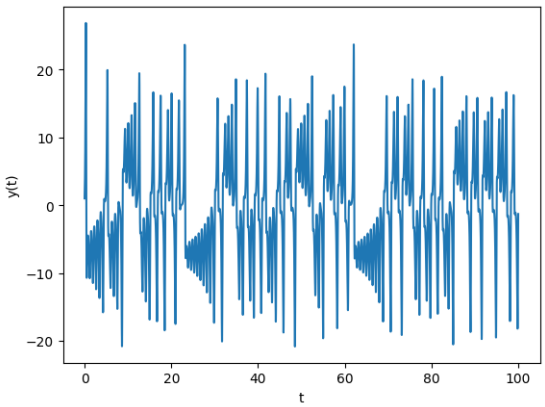
$$\dot{z} = xy - bz$$

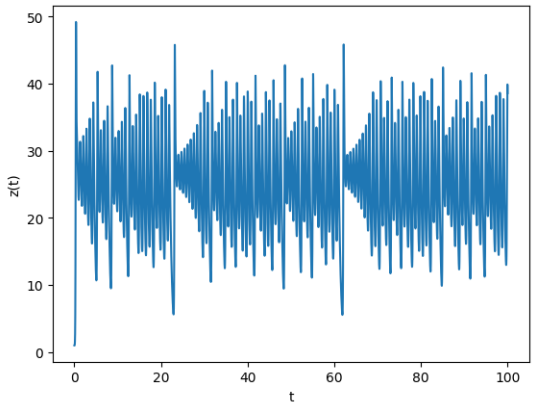
Taking (σ, b, r) = (10, 2, 28) and the initial conditions $(x_0, y_0, z_0) = (1, 1, 1)$, RK4 method returns these solutions:

```
h = 0.001
t = np.arange(0, 100, h)
x = np.zeros(len(t))
y = np.zeros(len(t))
z = np.zeros(len(t))
```

```
sigma = 10
b = 2
r = 28
x[0] = 1
y[0] = 1
z[0] = 1

for i in range(0, len(t) -1):
    k1 = h*sigma*(y[i] - x[i])
    k2 = h* sigma*(y[i] - (x[i] + 0.5*k1))
    k3 = h* sigma*(y[i] - (x[i] + 0.5*k2))
    k4 = h* sigma*(y[i] - (x[i] + k3))
    x[i+1] = x[i] + (k1 + 2*k2 + 2*k3 + k4)/6

    k1_ = h*(r*x[i] - y[i] - x[i]*z[i])
    k2_ = h*(r*x[i] - (y[i] + 0.5*k1_) - x[i]*z[i])
    k3_ = h*(r*x[i] - (y[i] + 0.5*k2_) - x[i]*z[i])
    k4_ = h*(r*x[i] - (y[i] + k3_) - x[i]*z[i])
    y[i+1] = y[i] + (k1_ + 2*k2_ + 2*k3_ + k4_)/6

    k1__ = h*(x[i]*y[i] - b*z[i])
    k2__ = h*(x[i]*y[i] - b*(z[i] + 0.5*k1__))
    k3__ = h*(x[i]*y[i] - b*(z[i] + 0.5*k2__))
    k4__ = h*(x[i]*y[i] - b*(z[i] + k3__))
    z[i+1] = z[i] + (k1__ + 2*k2__ + 2*k3__ + k4__)/6
```







2. I used Newton-Raphson method for a system of non-linear coupled equations:

$$f_1 = y - x = 0$$

$$f_2 = rx - y - xz = 0$$

$$f_3 = xy - bz = 0$$

To solve for the roots, one must find difference column vector

$$\Delta v = \begin{pmatrix} x_{i+1} - x_i \\ y_{i+1} - y_i \\ z_{i+1} - z_i \end{pmatrix}$$

Such that $J\Delta v = -f$ where J is the Jacobian matrix. Hence, $\Delta v = -J^{-1}f$ and we can now use the recurrence relation

$$(x_{i+1}, y_{i+1}, z_{i+1}) = (x_i, y_i, z_i) + \Delta v$$

Fixing the above values of (σ, b, r) where r $>$ 1, I computed $\sqrt{b(r-1)} \approx 7$ and $r - 1 = 27$ and then started my initial guess by doing 1 bisection on the interval near the values I computed.
I got the fixed points, depending on the initial interval I introduced, as follows:

```
def F(x, y, sigma):
    return sigma * y - sigma * x

def G(x, y, z, r):
    return r * x - y - x * z

def H(x, y, z, b):
    return x * y - b * z

def J(x, y, z, sigma, r, b):
```

```
    return np.array([[-sigma, sigma, 0],
                     [r - z, -1, -x],
                     [y, x, -b]])

def Newton_System(J, F, G, H, x0, y0, z0, p, k):
    root = np.zeros((k, 3))
    root[0] = [x0, y0, z0]
    n = 0
    tol = 0.5 * 10**(-p + 1)
    error = tol + 1
    f = np.zeros(3)
    sigma = 10
    b = 2
    r = 28
    while (error > tol and n < k - 1):
        f[0] = F(root[n][0], root[n][1], sigma)
        f[1] = G(root[n][0], root[n][1], root[n][2], r)
        f[2] = H(root[n][0], root[n][1], root[n][2], b)
        delta = -np.linalg.inv(J(root[n][0], root[n][1], root[n][2], sigma, r, b)) @ f
        root[n+1] = root[n] + delta
        error = np.linalg.norm(root[n+1] - root[n]) / np.linalg.norm(root[n+1])
        n = n + 1
    if n >= k:
        print("No convergence")
    else:
        final_r = root[len(root) - 1]
    return final_r
```

• fixed point 1 = (0,0,0)
• fixed point 2 = (7.34846923, 7.34846923, 27)
• fixed point 3 = (-7.34846923, -7.34846923, 27)
which is exactly what I calculated them to be.
Now, to study the stability at these fixed points, we find the determinant of the Jacobian matrix at these points: The Jacobian matrix is

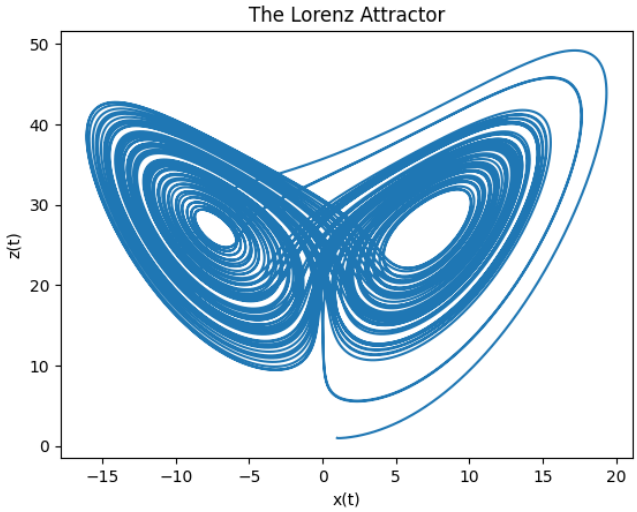
$$J = \begin{pmatrix} -\sigma & \sigma & 0 \\ r - z & -1 & -x \\ y & x & -b \end{pmatrix}$$

Finding it at fixed point 1, we get $det(J)_1 = 539.9$. At fixed point 2, we get $det(J)_2 = -1080$ and at fixed point 3, we get $det(J)_3 = -1080$. Hence fixed points 2,3 are stable and fixed point 1 is

unstable.

```
det1 = np.linalg.det(J(-7.34846923, -7.34846923, 27, sigma, r, b))
det2 = np.linalg.det(J(7.34846923, 7.34846923, 27, sigma, r, b))
det3 = np.linalg.det(J(0, 0, 0, sigma, r, b))
```

To check these stable points visually, I plotted $z(t)$ as function of $x(t)$ getting the famous "Lorenz Attractor":



Other than being beautiful, this graph shows that the stable fixed point 2 for (x = 7.34846923, z = 27) and the stable fixed point 3 (x = -7.34846923, z = 27) are the "butterfly wings" regions where oscillations occur about them, emphasizing their stability. Fixed point 1 (x = 0, z = 0) is the point of intersection of the trajectories orbiting the stable points and is obviously unstable.

3. Taking 2 arbitrary initial conditions (0, 1, 2) and $(0 + \delta_{0_1}, 1 + \delta_{0_2}, 2 + \delta_{0_3})$ where I took $\delta_0 = [0.5 \quad 0.5 \quad 0.5]$.

Solving the system for these 2 initial conditions, we take the last points in the 2 solutions at the last value for time (in my case $t_f = 100$) and find their difference getting a new vector $\delta(t_f) = [\delta_1 \quad \delta_2 \quad \delta_3]$. The Liapunov exponent is hence found by taking the logarithm on both sides of this equation $|\delta(t_f)| = |\delta_0|e^{\lambda t_f}$ getting

$$\lambda = \frac{1}{t_f} ln \left(\frac{|\delta(t_f)|}{|\delta_0|} \right)$$

In my case, I found $\delta(t_f)$ to be [4.58773876 14.11353406 6.03284588] and hence, $\lambda \approx 0.02917$. The Liapunov exponent being positive is a signature of chaos, so it was something expected.

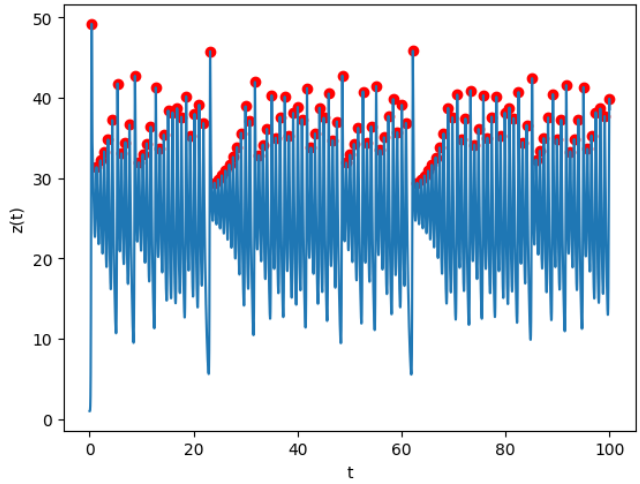
```
h = 0.001
t = np.arange(0, 100, h)
x1 = np.zeros(len(t))
x2 = np.zeros(len(t))
y1 = np.zeros(len(t))
```

```
y2 = np.zeros(len(t))
z1 = np.zeros(len(t))
z2 = np.zeros(len(t))
sigma = 10
b = 2
r = 28
delta0 = np.zeros(3)
delta0[0] = 0.5
delta0[1] = 0.5
delta0[2] = 0.5
x1[0] = 0
x2[0] = 0 + delta0[0]
y1[0] = 1
y2[0] = 1 + delta0[1]
z1[0] = 2
z2[0] = 2 + delta0[2]

for i in range(0, len(t) -1):
    k1 = h*sigma*(y1[i] - x1[i])
    k2 = h* sigma*(y1[i] - (x1[i] + 0.5*k1))
    k3 = h* sigma*(y1[i] - (x1[i] + 0.5*k2))
    k4 = h* sigma*(y1[i] - (x1[i] + k3))
    x1[i+1] = x1[i] + (k1 + 2*k2 + 2*k3 + k4)/6

    k11 = h*sigma*(y2[i] - x2[i])
    k22 = h* sigma*(y2[i] - (x2[i] + 0.5*k11))
    k33 = h* sigma*(y2[i] - (x2[i] + 0.5*k22))
    k44 = h* sigma*(y2[i] - (x2[i] + k33))
    x2[i+1] = x2[i] + (k11 + 2*k22 + 2*k33 + k44)/6

    k1_ = h*(r*x1[i] - y1[i] - x1[i]*z1[i])
    k2_ = h*(r*x1[i] - (y1[i] + 0.5*k1_) - x1[i]*z1[i])
    k3_ = h*(r*x1[i] - (y1[i] + 0.5*k2_) - x1[i]*z1[i])
    k4_ = h*(r*x1[i] - (y1[i] + k3_) - x1[i]*z1[i])
    y1[i+1] = y1[i] + (k1_ + 2*k2_ + 2*k3_ + k4_)/6

    k1_1 = h*(r*x2[i] - y2[i] - x2[i]*z2[i])
    k2_2 = h*(r*x2[i] - (y2[i] + 0.5*k1_1) - x2[i]*z2[i])
    k3_3 = h*(r*x2[i] - (y2[i] + 0.5*k2_2) - x2[i]*z2[i])
```

```
    k4_4 = h*(r*x2[i] - (y2[i] + k3_3) - x2[i]*z2[i])
    y2[i+1] = y2[i] + (k1_1 + 2*k2_2 + 2*k3_3 + k4_4)/6

    k1__ = h*(x1[i]*y1[i] - b*z1[i])
    k2__ = h*(x1[i]*y1[i] - b*(z1[i] + 0.5*k1__))
    k3__ = h*(x1[i]*y1[i] - b*(z1[i] + 0.5*k2__))
    k4__ = h*(x1[i]*y1[i] - b*(z1[i] + k3__))
    z1[i+1] = z1[i] + (k1__ + 2*k2__ + 2*k3__ + k4__)/6

    k1__1 = h*(x2[i]*y2[i] - b*z2[i])
    k2__2 = h*(x2[i]*y2[i] - b*(z2[i] + 0.5*k1__1))
    k3__3 = h*(x2[i]*y2[i] - b*(z2[i] + 0.5*k2__2))
    k4__4 = h*(x2[i]*y2[i] - b*(z2[i] + k3__3))
    z2[i+1] = z2[i] + (k1__1 + 2*k2__2 + 2*k3__3 + k4__4)/6

delta = np.zeros(3)
delta[0] = x1[len(x2) - 1] - x2[len(x1) - 1]
delta[1] = y1[len(x2) - 1] - y2[len(x1) - 1]
delta[2] = z1[len(x2) - 1] - z2[len(x1) - 1]
print(delta)

liapunov = (1/t[len(t) - 1]) * np.log(np.linalg.norm(delta) / np.linalg.norm(delta0))
print(liapunov)
```

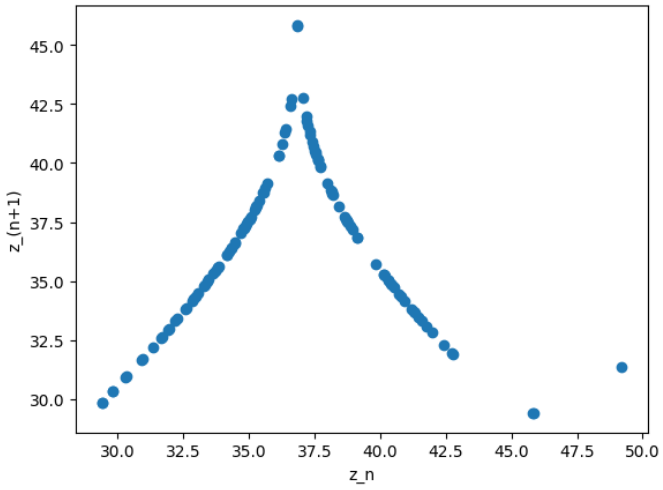
4. To find the local maxima of z(t), I developed an algorithm that compares each value of z with the one before and after it, if the middle value is bigger than both, it is stored as a local maximum. The algorithm gave me this plot with an array identifying all the local maxima of z as function of time:

```
z_max = []
t_max = []
for i in range(len(z) - 3):
    if z[i] < z[i+1] and z[i+1] > z[i+2]:
        z_max.append(z[i+1])
        t_max.append(t[i+1])
```



Taking the z series and using it to produce 2 arrays: 1 shifted to the right (removing its first element) - denoting it by z_{n+1} and the other shifting it to the left (removing its last element) - denoting it by z_n. Plotting z_{n+1} as function of z_n, we get this graph:

```
z_n1 = z_max[1:]
z_n = z_max[:len(z_max)-1]
```



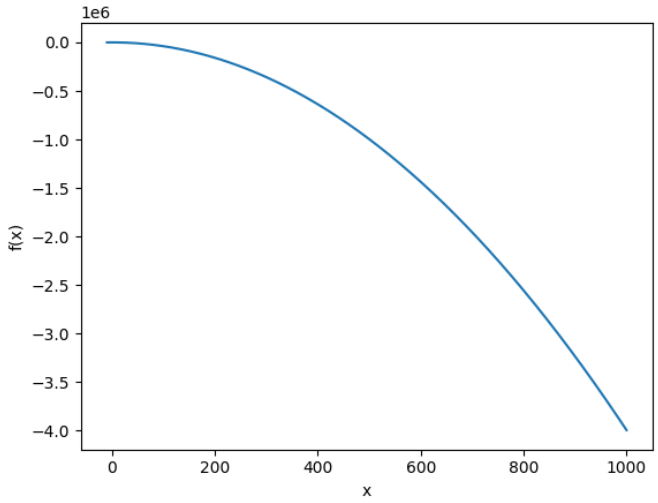
Prob 6 - Logistic Map

The logistic map is given by

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

1. Plotting $f(x)$ as function of x, for $\mu = 4$, we get a graph of a decreasing function

```
x = np.arange(-10, 1000, 0.05)
mu = 4
y = mu*x*(1-x)
```



2. I implemented a function that takes an initial value (x_0), a μ value a number of iterations (m), and returns an array of m "random" numbers. The function starts with the initial value and apply the logistic map recurrence relation m times, each time generating a random number. For instance, taking ($x_0 = 0.5, \mu = 3.5, m = 5$), I get this array of 5 random numbers:

$$[0.875, 0.3828125, 0.826934814453125, 0.5008976948447526, 0.87499717950388]$$

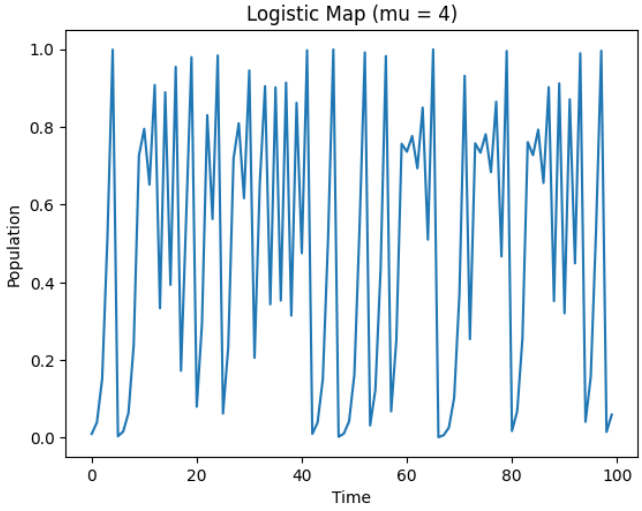
To make things even more random, I tried taking the initial condition from a numpy random number generator on an interval $[0, 1, 3)$.

```
def LogisticMap_rand(x0, mu, n):
    rand_values = []
    x = x0
    for i in range(n):
        x = mu*x*(1-x)
        rand_values.append(x)
    return rand_values
x0 = np.random.rand(1) * (3-0.1) + 0.5
print(LogisticMap_rand(x0, 3.5, 5))
```

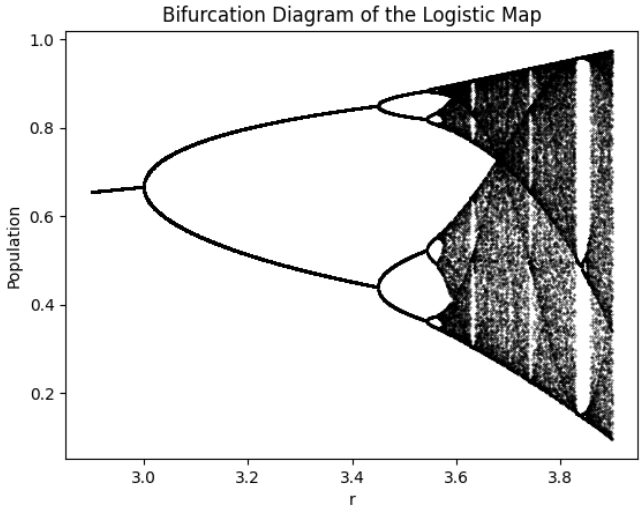
Solving the logistic map, i.e. starting by an initial condition and applying the recurrence relation function for $\mu = 4$, $x_0 = 0.4$ and 100 iterations, where the x values represent the population and the iterations play the role of time steps. I got this solution graph

```
def logistic_map(x0, mu, m):
    population = [x0]
    for i in range(m - 1):
        x0 = mu * x0 * (1 - x0)
        population.append(x0)
```

```
    return population
x0 = 0.4
mu = 4
iterations = 100
```



3. Generating the bifurcation diagram of the logistic map: I implemented a function that takes an array of r values, an initial condition x_0, and 2 values for "transient" iterations and overall iterations. It applies the logistic map on the parameters with the transient iterations, allowing the system to settle then takes the last value of these iterations as the initial condition for the logistic map with the overall iterations. The results are now stored in an array as the y values of the bifurcation diagram and the r array as the x values. Getting for $r = [2.9, 3.9]$ and $x_0 = 0.4$:



Now, to keep track of the doubling of the roots, I truncated the y values for each r value up to 3 decimal digits and converted the array into a set such that only the unique values (up to 3

decimal digits) are stored. I found the first instances of doubling, which corresponded to this first r values with (2, 4, 8, 16, 32, 64, 128) unique roots and got these unique population values to be, respectively, corresponding to these r values: (3, 3.45, 3.545, 3.565, 3.569, 3.57, 3.571).
Recovering $\delta = \frac{\Delta_n}{\Delta_{n+1}}$ I got these values (4.739, 5, 4.1) which verified the convergence to the first Feigenbaum constant (4.669201..). For the α value, I found the population values at the last occurrence of the 2 roots (0.44, 0.85) and the last occurrence of the lower 4 roots (0.36, 0.51) getting $\alpha = \frac{d_n}{d_{n+1}} = 2.739$ which is close to the second Feigenbaum constant (2.502907..).
The code is as follows:

```
x0 = 0.4
def truncate_number_strict(lst, decimal_places):
    factor = 10.0 ** decimal_places
    return [int(element * factor) / factor for element in lst]

def bifurcation_diagram(r_values, x0, transient_iterations, iterations):
    bifurcation_data = []
    uni = []
    len_s = []
    for r in r_values:

        transient_population = logistic_map(x0, r, transient_iterations)
        final_population = logistic_map(transient_population[-1], r, iterations)
        bifurcation_data.append(final_population)
        s = set(truncate_number_strict(final_population, 3))
        uni.append({r:s})
        len_s.append({r:len(s)})
    return bifurcation_data, uni, len_s

r_values = [i/1000 for i in range(2900, 3900)]
transient_iterations = 10000
population_values = bifurcation_diagram(r_values, x0, transient_iterations, iterations)
bifuraction_data, uni, len_s = bifurcation_diagram(r_values, x0, transient_iterations,
iterations)

for i in range(len(r_values)):
    plt.plot([r_values[i]] * len(bifuraction_data[i]), bifuraction_data[i], 'k.',
    markersize=0.6)
```

Prob 7 - What Do 1D Maps Have To Do With Science?

The Rossler system is given by

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

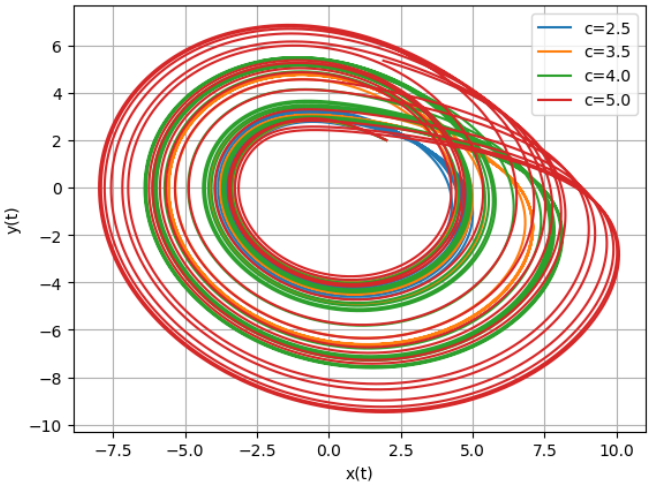
$$\dot{y} = x + ay$$

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

with a = b = 0.2 **1.** Taking c = [2.5, 3.5, 4, 5], solving the Rossler system and plotting the y versus the x solutions we get

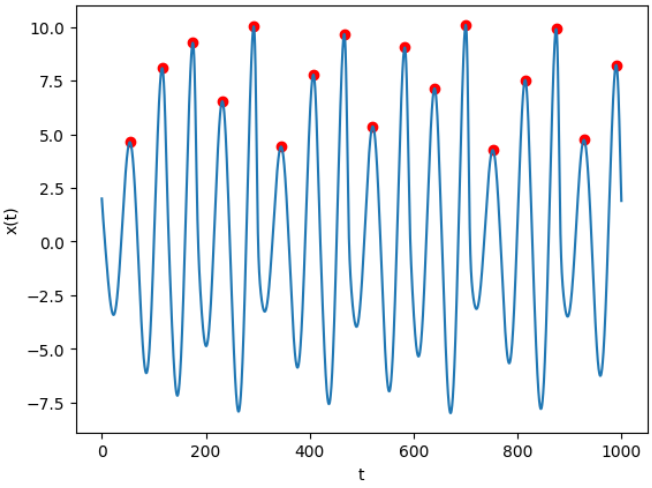
```
t = np.arange(0,1000, 0.01)
x = np.zeros(len(t))
y = np.zeros(len(t))
z = np.zeros(len(t))
x[0] = 2
y[0] = 2
z[0] = 2
a = 0.2
b = 0.2
c = [2.5, 3.5, 4, 5]
def RK4_Rossler(x, y, z, a, b, c):
    h = 0.001
    for i in range(0, len(t)-1):
        k1 = h* (- y[i] - z[i])
        k2 = h* (- y[i] - z[i])
        k3 = h* (- y[i] - z[i])
        k4 = h* (- y[i] - z[i])
        x[i+1] = x[i] + (k1 + 2*k2 + 2*k3 + k4)/6

        k1_ = h* (x[i] + a*y[i])
        k2_ = h* (x[i] + a*(y[i] + 0.5*k1_))
        k3_ = h* (x[i] + a*(y[i] + 0.5*k2_))
        k4_ = h* (x[i] + a*(y[i] + k3_))
        y[i+1] = y[i] + (k1_ + 2*k2_ + 2*k3_ + k4_)/6

        k1__ = h* (b + z[i] * (x[i] - c))
        k2__ = h* (b + (z[i] + 0.5*k1__) * (x[i] - c))
        k3__ = h* (b + (z[i] + 0.5*k2__) * (x[i] - c))
        k4__ = h* (b + (z[i] + k3__) * (x[i] - c))
        z[i+1] = z[i] + (k1__ + 2*k2__ + 2*k3__ + k4__)/6
```

```
    return x, y, z
```

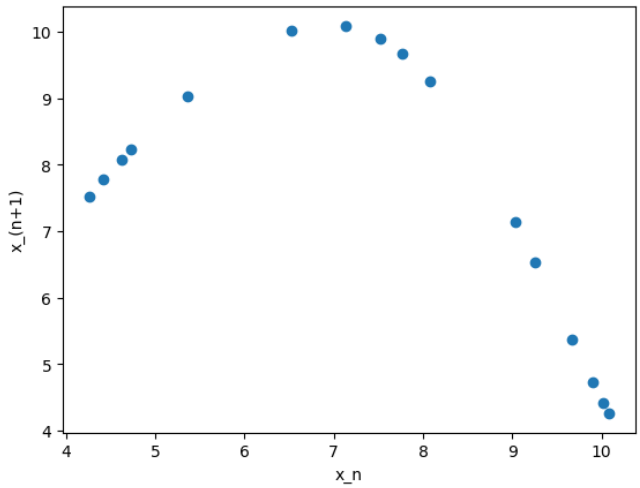


2. Using the same method the find the x maxima for c = 5 and plot x_{n+1} as function of x_n as for the Lorenz system,

```
x5,y5,z5 = RK4_Rossler(x, y, z, 0.2, 0.2, 5)
x_max = []
t_max = []
for i in range(len(z) - 3):
    if x5[i] < x5[i+1] and x5[i+1] > x5[i+2]:
        x_max.append(x5[i+1])
        t_max.append(t[i+1])
```

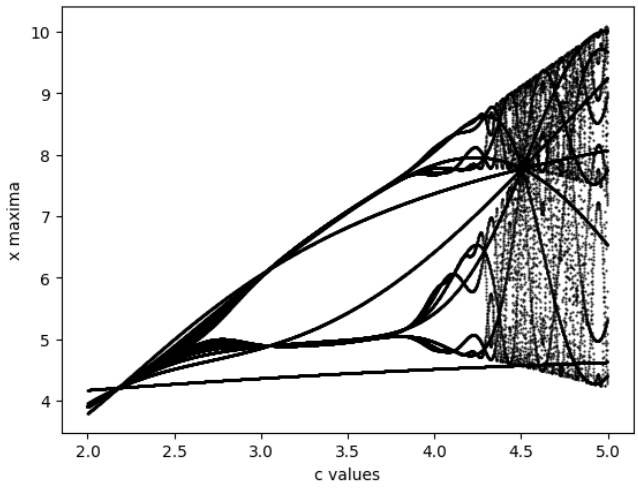



```
x_n1 = x_max[1:]
x_n = x_max[:len(x_max)-1]
```



3. I implemented a function that stores the x maxima for a range of c values [2,5], then plotted these maxima for each c values, getting this bifurcation graph

```
c_vals = np.arange(2, 5, 0.001)
def rossler_diagram(c_vals, RK4_Rossler, x, y, z, a, b):
    max_values = []
    for c in c_vals:
     xs, ys, zs = RK4_Rossler(x, y, z, a, b, c)
     x_o = []
     for i in range(len(xs) - 3):
        if xs[i] < xs[i+1] and xs[i+1] > xs[i+2]:
            x_o.append(xs[i+1])
     max_values.append(x_o)
    return max_values
data = rossler_diagram(c_vals, RK4_Rossler, x, y, z, a, b)
for i in range(len(c_vals)):
    plt.plot([c_vals[i]]*len(data[i]), data[i], 'k.', markersize=1)
```



This "Rossler Bifurcation Diagram" exhibits similar behavior as the bifurcation diagram of the logistic map. We notice root doubling events around c = 2.6 and then around c = 3.8 and so on.