HW1$_P hys222$

rah120

November 2023

1 Question1:Network Laplacian

```
[h]
import numpy as np
import matplotlib.pyplot as plt
import sympy as sp
import scipy
import networkx as nx
from sklearn.cluster import KMeans

k, m= sp.symbols('k m')
N= 10
D= sp.zeros(N,N)
A= sp.zeros(N,N)

for i in range(N):
    if i == 0:
        A[i, i] = -1 *(k/m)
        D[i, i+1]= 1 *(k/m)
    elif i==N-1:
        A[i , i]= -1 * (k/m)
        D[i, i-1]= 1 * (k/m)
    else:
        A[i , i] = -2 * (k/m)
        D[i, i-1]= 1 * (k/m)
        D[i, i+1]= 1 *(k/m)

L= D+A
eigenvalues = L.eigenvals()
a=L.eigenvects()
v1= a[0][2]

# assuming that v2 is the second smallest eigenvector
v2= a[1][2]
v2= sp.Matrix(a[1][2])
```

```
print(v2)
partition_ids = [1 if v > 0 else 0 for v in v2]
G = nx.path_graph(10)
pos = nx.spring_layout(G)
nx.draw(G, pos, node_color=partition_ids, with_labels=True, cmap=plt.cm.RdYlBu)
plt.show()
#It showed that the first one is bue, 2nd and 3rd are red, etc.
#Interpretation: the pattern can be interpreted as 2 nodes having same color are strongly co
# also it can suggest that the amplitude of oscillations of 2 different colors are different
#A random network
#Adjacency matrices
#first, we generate the networks
#Random Network
N=10
m= 9
E_R= nx.erdos_renyi_graph(N, m)
A1 = nx.adjacency_matrix(E_R).toarray()

#SCALE_FREE NETWORK
barabasi= nx.barabasi_albert_graph(N, m)
A2= nx.adjacency_matrix(barabasi).toarray()

#A SMALL WOLRD NETWORK
p = 1/(N-1)  # Probability of rewiring each edge
W_S= nx.watts_strogatz_graph(N, m, p)
A3= nx.adjacency_matrix(W_S).toarray()

#Laplacians and inferring communities
#Random Network
L1 = nx.laplacian_matrix(E_R).toarray()
eigvl, eigvec = np.linalg.eigh(L1)
# Extract the Fiedler vector (second smallest eigenvector)
fiedler_vector1 = eigvec[:, 1]
# Infer communities based on the sign of the Fiedler vector
communities1 = [0 if x >= 0 else 1 for x in fiedler_vector1]
centrality1 = nx.betweenness_centrality(E_R)
avg_path_length1 = nx.average_shortest_path_length(E_R)
print("Average path length1:", avg_path_length1)
clustering_coefficients1 = nx.clustering(E_R)
for node, cc in clustering_coefficients1.items():
    print(f"Node {node}: Clustering Coefficient = {cc}")
#2
L2 = nx.laplacian_matrix(barabasi).toarray()
eigvl, eigvec = np.linalg.eigh(L2)
fiedler_vector2 = eigvec[:, 1]
communities2 = [0 if x >= 0 else 1 for x in fiedler_vector2]
```

```
centrality2 = nx.betweenness_centrality(barabasi)
avg_path_length2 = nx.average_shortest_path_length(barabasi)
print("Average path length2:", avg_path_length2)
clustering_coefficients2 = nx.clustering(barabasi)
for node, cc in clustering_coefficients2.items():
    print(f"Node {node}: Clustering Coefficient = {cc}")
#L3
L3 = nx.laplacian_matrix(W_S).toarray()
eigvl, eigvec = np.linalg.eigh(L3)
fiedler_vector3 = eigvec[:, 1]
communities3 = [0 if x >= 0 else 1 for x in fiedler_vector3]
centrality3 = nx.betweenness_centrality(W_S)
avg_path_length3 = nx.average_shortest_path_length(W_S)
print("Average path length3:", avg_path_length3)
clustering_coefficients3 = nx.clustering(W_S)
for node, cc in clustering_coefficients3.items():
    print(f"Node {node}: Clustering Coefficient = {cc}")

#different sizes of N
N_list= [50, 100, 150]
for N1 in N_list:
   m= 9
   E_R= nx.erdos_renyi_graph(N1, m)
   A1 = nx.adjacency_matrix(E_R).toarray()

#SCALE_FREE NETWORK
   barabasi= nx.barabasi_albert_graph(N1, m)
   A2= nx.adjacency_matrix(barabasi).toarray()

#A SMALL WOLRD NETWORK
   p = 1/(N1-1)  # Probability of rewiring each edge
   W_S= nx.watts_strogatz_graph(N1, m, p)
   A3= nx.adjacency_matrix(W_S).toarray()

#Laplacians and inferring communities
#Random Network
   L1 = nx.laplacian_matrix(E_R).toarray()
   eigvl, eigvec = np.linalg.eigh(L1)
# Extract the Fiedler vector (second smallest eigenvector)
   fiedler_vector1 = eigvec[:, 1]
# Infer communities based on the sign of the Fiedler vector
   communities1 = [0 if x >= 0 else 1 for x in fiedler_vector1]
   centrality1 = nx.betweenness_centrality(E_R)
   avg_path_length1 = nx.average_shortest_path_length(E_R)
   print("Average path length1:", avg_path_length1)
   clustering_coefficients1 = nx.clustering(E_R)
```

```
   for node, cc in clustering_coefficients1.items():
     print(f"Node {node}: Clustering Coefficient = {cc}")
#2
   L2 = nx.laplacian_matrix(barabasi).toarray()
   eigvl, eigvec = np.linalg.eigh(L2)
   fiedler_vector2 = eigvec[:, 1]
   communities2 = [0 if x >= 0 else 1 for x in fiedler_vector2]
   centrality2 = nx.betweenness_centrality(barabasi)
   avg_path_length2 = nx.average_shortest_path_length(barabasi)
   print("Average path length2:", avg_path_length2)
   clustering_coefficients2 = nx.clustering(barabasi)
   for node, cc in clustering_coefficients2.items():
      print(f"Node {node}: Clustering Coefficient = {cc}")
#L3
   L3 = nx.laplacian_matrix(W_S).toarray()
   eigvl, eigvec = np.linalg.eigh(L3)
   fiedler_vector3 = eigvec[:, 1]
   communities3 = [0 if x >= 0 else 1 for x in fiedler_vector3]
   centrality3 = nx.betweenness_centrality(W_S)
   avg_path_length3 = nx.average_shortest_path_length(W_S)
   print("Average path length3:", avg_path_length3)
   clustering_coefficients3 = nx.clustering(W_S)
   for node, cc in clustering_coefficients3.items():
      print(f"Node {node}: Clustering Coefficient = {cc}")

# for random networks, I didn't notice a change. In
scale free networks networks, the clustering coefficients are decreasing slightly as expecte
# increase in path length, but unaffected clustering coefficients as N increases for small w

#Loss of connectivity
num_nodes = 100
scale_free_network = nx.barabasi_albert_graph(num_nodes, m=5)
title = 'trial'
strategy = 'centrality'
def plot_network(scale_free_network, title):
    pos = nx.spring_layout(scale_free_network)
    nx.draw(scale_free_network, pos, with_labels=True)
    plt.title(title)
# Simulate targeted attacks
def targeted_attack(scale_free_network, strategy):
    original_nodes = set(scale_free_network.nodes())
    components_sizes = []

    while len(scale_free_network.nodes()) > 0:
        if strategy == "degree":
            node_to_remove = max(scale_free_network.degree(), key=lambda x: x[1])[0]
```

```
        elif strategy == "centrality":
            node_to_remove = max(nx.eigenvector_centrality(scale_free_network),key=nx.eigenv
        elif strategy == "random":
            node_to_remove = np.random.choice(list(scale_free_network.nodes()))

        scale_free_network.remove_node(node_to_remove)
        components = list(nx.connected_components(scale_free_network))
        components_sizes.append([len(comp) for comp in components])

        # Plot the network and pause for a short duration
        plt.clf()
        plot_network(scale_free_network, f"Remaining Nodes: {len(scale_free_network.nodes())
        plt.pause(0.5)  # Adjust the duration as needed

    return original_nodes, components_sizes

# Main function
def main():
    num_nodes = 100
    G = nx.barabasi_albert_graph(num_nodes, m=5)
    strategies = ["degree", "centrality", "random"]

    for strategy in strategies:
        original_nodes, components_sizes = targeted_attack(G.copy(), strategy)

    plt.show()

main()

#The best strategy is by attacking the degree because removing the highest degrees can disru
```

1. It showed that the first one is blue, 2nd and 3rd are red, etc. Interpretation: the pattern can be interpreted as 2 nodes having same color are strongly coupled or moving phase; while 2 of different colors ca suggest that they are moving out of phase and no strong coupling. Also it can suggest that the amplitude of oscillations of 2 different colors are different.
3.For random networks, I didn't notice a change. In scale free networks networks, the clustering coefficients are decreasing slightly as expected and the average length is increasing too increase in path length, but unaffected clustering coefficients as N increases for small worlds. The best strategy is by attacking the degree because removing the highest degrees can disrupt the network faster than others.
Note: I couldn't plot the loss of connectivity at the end because it is an animation.

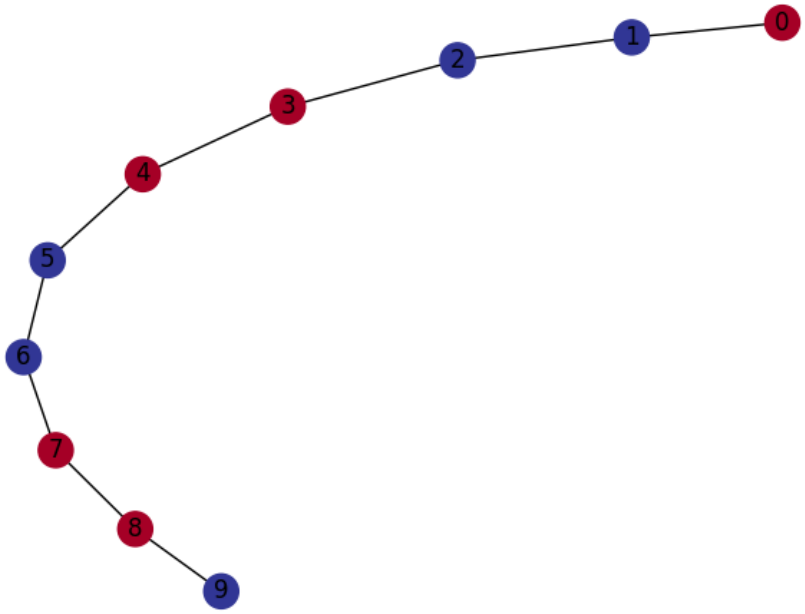

Figure 1: x(t) colored

2 Q2: Tight-binding

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

#small world
N=12
k=3
p= 0.03
W_S= nx.watts_strogatz_graph(N, k, p)

pos = nx.spring_layout(W_S)
nx.draw(W_S, pos, with_labels=False, node_size=30)
plt.title("Small-World Network (Watts-Strogatz Model)")
plt.show()

#random network
p2= 0.4
E_R= nx.erdos_renyi_graph(N, p= p2)
pos = nx.spring_layout(E_R)
nx.draw(E_R, pos, with_labels=True, node_size=100, node_color='skyblue', font_size=8)
plt.title("Random Erdős-Rényi Graph")
plt.show()

#regular network
p3= 0.6 #not used
regular= nx.random_regular_graph(4, N)
pos = nx.spring_layout(regular)
nx.draw(regular, pos, with_labels=False, node_size=30)
plt.title("Random Regular Network")
plt.show()

def functions(title):
    if title == 'random network':
        b= E_R
    elif title== 'small world':
        b= W_S
    else:
        b= regular
    A1= nx.adjacency_matrix(b).toarray()
    eig = np.linalg.eigvalsh(A1)
    Z = np.sum(np.exp(eig))
    E= [-1 * i for i in eig]
    print(E)
```

```
    BoltzF= 1
    p= [ (np.exp(np.clip(- 1* BoltzF * i, -700, 700))/ Z) for i in E]
    S= -1 *sum([i * np.log(i) for i in p])
    print(S)
    F= - (1/BoltzF) * np.log(Z)
    print(F)
    G= F + (1/BoltzF) * np.log(N)
    print(G)

functions('random network')
functions('small world')
functions('regular')

#or
def functions(title):
    if title == 'random network':
        b= E_R
    elif title== 'small world':
        b= W_S
    else:
        b= regular
    A1= nx.adjacency_matrix(b).toarray()
    eig = np.linalg.eigvalsh(A1)
    Z = np.sum(np.exp(eig))
    E= [-1 * i for i in eig]
    print(E)
    BoltzF= 1
    p= dict(b.degree())
    S= -1 *sum([i/N * np.log(i/N) for i in p.values()])
    print(S)
    F= - (1/BoltzF) * np.log(Z)
    print(F)
    G= F + (1/BoltzF) * np.log(N)
    print(G)

functions('random network')
functions('small world')
functions('regular')
```

I computed p in two different ways; the first one is like we get it in statistical mechanics using boltzman's factor(I can put the usual value of betta)/. In the 2nd one, I used the formual p= Ni/N I computed p in two different ways; the first one is like we get it in statistical mechanics using boltzman's factor(I can put the usual value of betta)/. In the 2nd one, I used the formual p= Ni/N

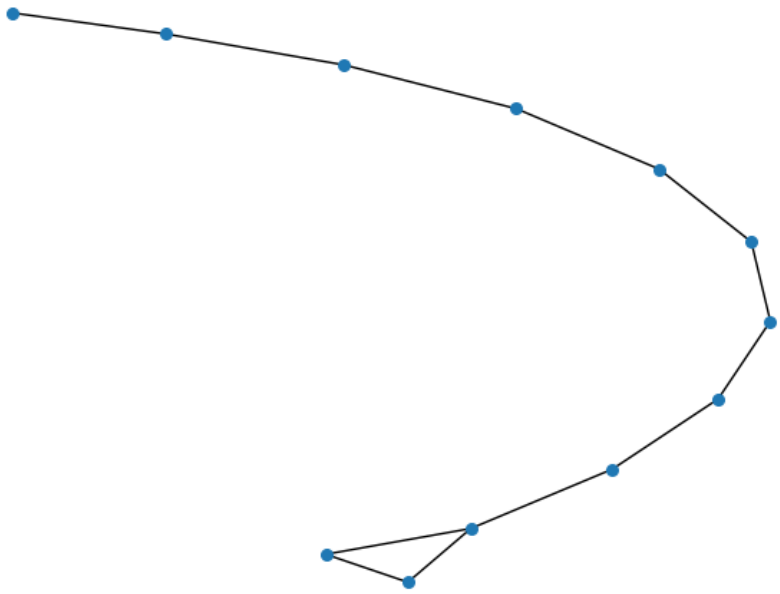

Figure 2: small world network

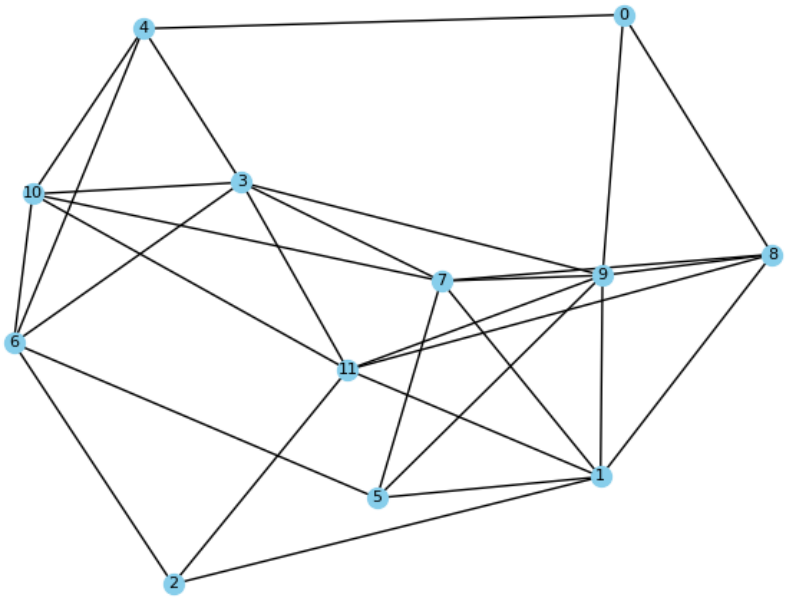

Figure 3: random network

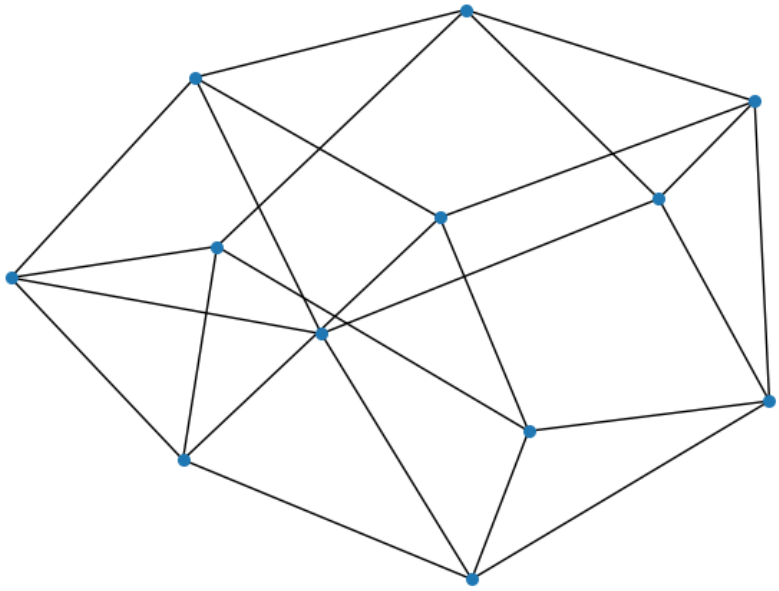

Figure 4: regular network

3 Q3: KPZ Universality Class and the Wigner Semi Circle

```
    import numpy as np
import matplotlib.pyplot as plt
import sympy as sp
import networkx as nx
import random
from scipy.stats import semicircular


x11, x12, x22, theta, l1, l2 = sp.symbols("x11 x12 x22 theta l1 l2")
X= sp.Matrix([x11, x12, x12, x22])
X= X.reshape(2, 2)
O = sp.Matrix([sp.cos(theta), - sp.sin(theta), sp.sin(theta), sp.cos(theta)])
O= O.reshape(2,2)
OT= O.transpose()
l= sp.Matrix([l1, 0, 0, l2])
l= l.reshape(2,2)
X1= O @ l @ OT
x11= X1[0, 0]
x12= X1[0, 1]
x22= X1[1, 1]
J= sp.Matrix([sp.diff(x11, l1), sp.diff(x22, l1), sp.diff(x12, l1), sp.diff(x11, l2), sp.dif
J= J.reshape(3, 3)
print(J)
det= sp.det(J)
det1= sp.simplify(det)
print(det1)

#2
N= 100
random_integers= sorted([random.uniform(-10, 10) for i in range(N)])
Hamiltonian= 0
b= 0
Hamiltonian_list= []
for i in  range(len(random_integers)):
    j = i+ 1
    while j <= N-1:
        Hamiltonian += (1/2) * random_integers[i]**2 -(np.log(abs(random_integers[i]- rando
        j +=1
    Hamiltonian_list.append(Hamiltonian)
plt.plot(random_integers, Hamiltonian_list)
plt.plot(random_integers, Hamiltonian_list)
plt.show()
```

```
#I can interpret it as lamda squared goes for the kinetic energy, and log goes for potential
#If we thought that the nodes should be connected as our goal from the question, kinetic ene
#in order for the potential to be high and so the eigenvalues are not centered around a valu
#3
N1= 10000
p= 0.5
E_R= nx.erdos_renyi_graph(N1, p)
A= nx.adjacency_matrix(E_R).toarray()
eig = np.linalg.eigvalsh(A)
r= (N1 * p * (1-p))**(1/2)
random_integers1= sorted([random.uniform(- 2* r, 2* r) for i in range(N1)])
density= [((4 - i**2)*(1/2))/ (2 * np.pi) for i in eig]
#print(density)
def remove_outliers(data):
    q1 = np.percentile(data, 25)
    q3 = np.percentile(data, 75)
    iqr = q3 - q1
    lower_bound = q1 - 1.5 * iqr
    upper_bound = q3 + 1.5 * iqr
    return data[(data >= lower_bound) & (data <= upper_bound)]
# well removing the outlier above code, I took it in stat231
new_eig = remove_outliers(eig)
plt.hist(new_eig, bins=50, density=True, alpha=0.6, label='Eigenvalue Histogram')
hist, edges = np.histogram(new_eig, bins=50, density=True)
bin_midpoints = (edges[1:] + edges[:-1]) / 2
plt.plot(bin_midpoints, hist, 'r--', linewidth=2, label='Bin Edges')
plt.show()

#4
lambda_max= []
N2= sorted([random.randint(500, 1000) for i in range(10)])
for i in range(10):
    E_R1= nx.erdos_renyi_graph(N2[i], p)
    A1= nx.adjacency_matrix(E_R1).toarray()
    eig = np.linalg.eigvalsh(A1)
    r1= max(list(eig))
    lambda_max.append(r1)
    plt.plot(lambda_max, color='blue', linestyle='dashed')

    plt.title(f'Tracy-Widom Distribution for N={N}')
    plt.xlabel('Eigenvalue')
    plt.ylabel('Density')
    plt.legend()
    plt.show()
```

```
print(lambda_max)
```

2. I can interpret it as lambda squared goes for the kinetic energy, and log goes for potential interacting as a repulsive or attractive force If we thought that the nodes should be connected as our goal from the question, kinetic energy should be smaller than the potential energy; therefore, eigenvalues should be separated by a large distance in order for the potential to be high and so the eigenvalues are not centered around a value. However, if eigenvalues are close to each other, potential is negligible which suggest that they do converge to one lambda.

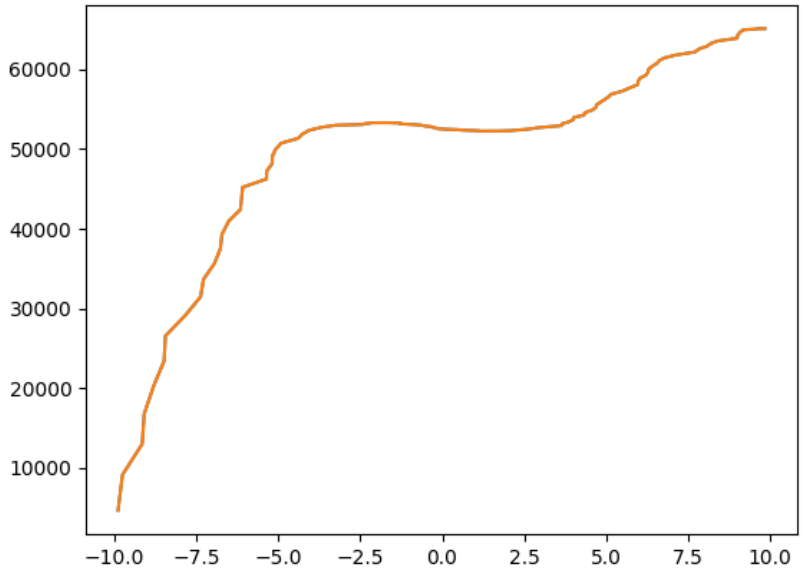

Figure 5: Hamiltonian

4. The distribution of Tracy-widom is also applied to the distribution of length of the longest increasing subsequence (Ulam Problem). T-W applied to study the fluctuations in the height of the interface in certain growth proceses (KPZ).

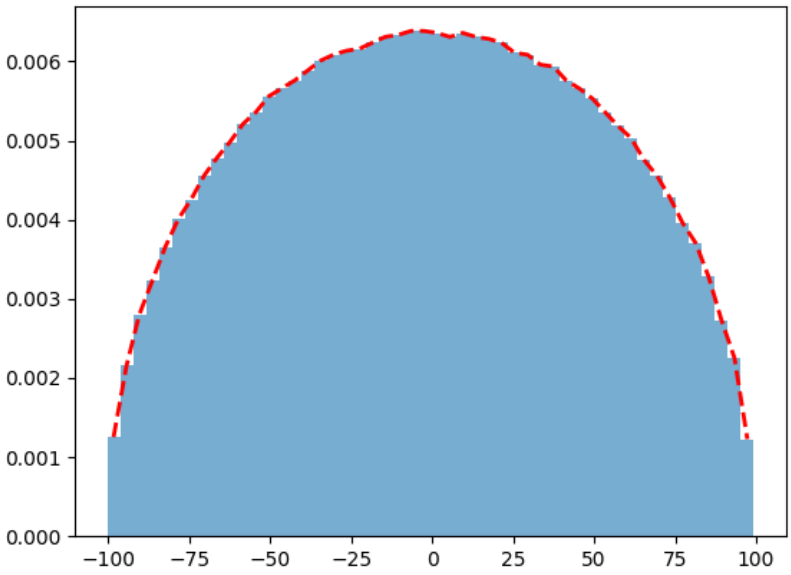

Figure 6: Lagrange for a Specific Data set

4 Q5:Dynamics

```
import numpy as  np
import matplotlib.pyplot as plt
import sympy as sp
import networkx as nx
import random
from scipy.optimize import fsolve
import scipy

r, s, b, x, y, z= sp.symbols('r s b x y z')
s=10
r= 28
b= 8/3
def f1(x, y, z, s):
    return s * (y - x)
def f2(x, y, z, r):
    return r*x - y - x * z
def f3(x, y, z, b):
    return x* y - b * z

def RK4(f1, f2, f3, x0, y0, z0, t0, tn, h):

    t_values = [t0]
    x_values = [x0]
    y_values = [y0]
    z_values = [z0]
    while t_values[-1] <= tn:
        t_n = t_values[-1]
        x_n = x_values[-1]
        y_n = y_values[-1]
        z_n = z_values[-1]

        k1x = h * f1(x_n, y_n, 0, s)
        k1y = h * f2(y_n, x_n, z_n, r)
        k1z = h * f3(x_n, y_n, z_n, b)

        k2x = h * f1(x_n + 0.5 * k1x, y_n , 0, s)
        k2y = h * f2(x_n, y_n + 0.5 * k1y, z_n, r)
        k2z = h * f3(x_n, y_n, z_n + 0.5 * k1z, b)

        k3x = h * f1(x_n + 0.5 * k2x, y_n, z_n, s)
        k3y = h * f2(x_n, y_n + 0.5 * k2y, z_n, r)
        k3z = h * f3(x_n, y_n, z_n + 0.5 * k2z, b)

        k4x = h * f1(x_n + k3x, y_n, z_n, s)
```

```
        k4y = h * f2(x_n, y_n + k3y, z_n, r)
        k4z = h * f3(x_n, y_n, z_n + k3z, b)

        x_n1 = x_n + (k1x + 2 * k2x + 2 * k3x + k4x) / 6
        y_n1 = y_n + (k1y + 2 * k2y + 2 * k3y + k4y) / 6
        z_n1 = z_n + (k1z + 2 * k2z + 2 * k3z + k4z) / 6

        t_n1 = t_n + h

        t_values.append(t_n1)
        x_values.append(x_n1)
        y_values.append(y_n1)
        z_values.append(z_n1)

    return t_values, x_values, y_values, z_values
print(RK4(f1, f2, f3, 2, 1, 3, 0, 10, 0.1))
t_values, x_values, y_values, z_values = RK4(f1,f2, f3, 1, 0, 0, 0, 10, 0.01)

plt.plot(t_values, x_values)
plt.plot(t_values, y_values, 'red')
plt.plot(t_values, z_values, "green")
plt.figure()
plt.plot(x_values, z_values)
plt.show()

#2
def lorenz_system(initial_conditions, s, r, b):
    x, y, z= initial_conditions
    return[f1(x, y, z, s), f2(x, y, z, r), f3(x, y, z, b)]
fixed_points= fsolve(lorenz_system, [1, 2, 3], args=(s, 4, b))
print(fixed_points)

    #constructing the Jacobian
r_list= np.arange(2, 40)
for i in r_list:
    fixed_points1= fsolve(lorenz_system, [i, i+1, i+2], args=(s, i, b))
    point = {x: fixed_points1[0], y: fixed_points1[1], z: fixed_points1[2]}
    print(point)
    J= sp.Matrix([sp.diff(f1(x, y, z, s), x).subs(point), sp.diff(f1(x, y, z, s), y).subs(po
    J= J.reshape(3, 3)
    eigenvalues, eigv= scipy.linalg.eig(np.array(J).astype(float))
    print(eigenvalues)
    if all(eig.real < 0 for eig in eigenvalues) and all(eig.imag ==0 for eig in eigenvalues)
        print('stable')
    elif eigenvalues.real[0]>0 and eigenvalues.real[1]>0 and eigenvalues.real[2]>0 and all(e
        print('unstable')
```

```
    elif any(eig.imag !=0 for eig in eigenvalues):
        print('oscillatory')
    else:
        print('saddle')

#3
#X1:
x10 = 1
y10= 2
z10 = 3

#X2:
x20= 2
y20= 3
z20= 4
t_inf= 100000
delta_0= ((x20 - x10)**2 + (y20 - y10)**2 + (z20- z10)**2)**(1/2)
t_values1, x_values1, y_values1, z_values1 = RK4(f1,f2, f3, x10, y10, z10, 0, t_inf, 0.01)
x10_inf= x_values1[-1]
y10_inf= y_values1[-1]
z10_inf= z_values1[-1]

t_values2, x_values2, y_values2, z_values2 = RK4(f1,f2, f3, x20, y20, z20, 0, t_inf, 0.01)
x20_inf= x_values2[-1]
y20_inf= y_values2[-1]
z20_inf= z_values2[-1]

delta_inf= ((x20_inf - x10_inf)**2 + (y20_inf - y10_inf)**2 + (z20_inf- z10_inf)**2)**(1/2)
liaponauv_exponent= (1/t_inf) * np.log(delta_inf/delta_0)
print(liaponauv_exponent)

t_values, x_values, y_values, z_values = RK4(f1, f2, f3, 1, 2, 3, 0, 1000, 0.001)
z_maxima= []
for j in range(len(z_values)-1):
       if z_values[j] > z_values[j-1] and z_values[j]> z_values[j+1]:
              z_maxima.append(z_values[j])
z_n= z_maxima[:-1:2]
z_nplus = z_maxima[1::2]

plt.scatter(z_n, z_nplus, label= f'Lorenz map')
plt.xlabel('z_n')
plt.ylabel('z_n+1')
plt.show()
```

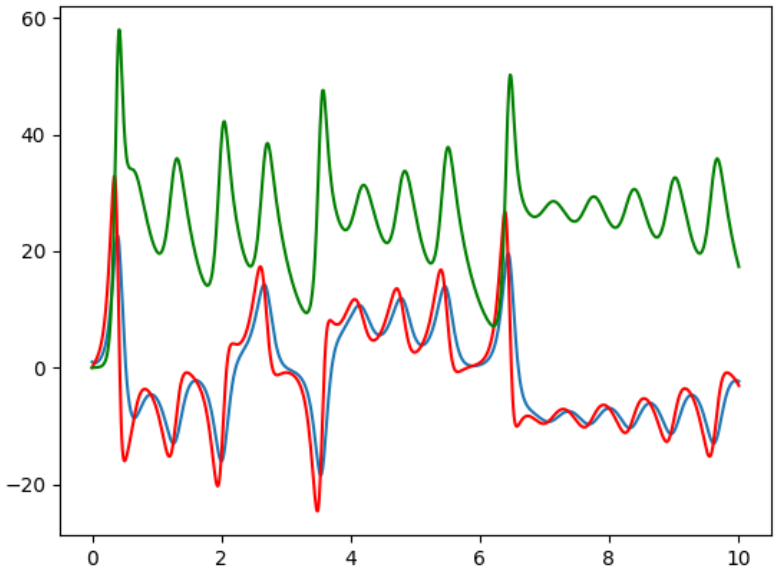

Figure 7: x(t), y(t), z(t)

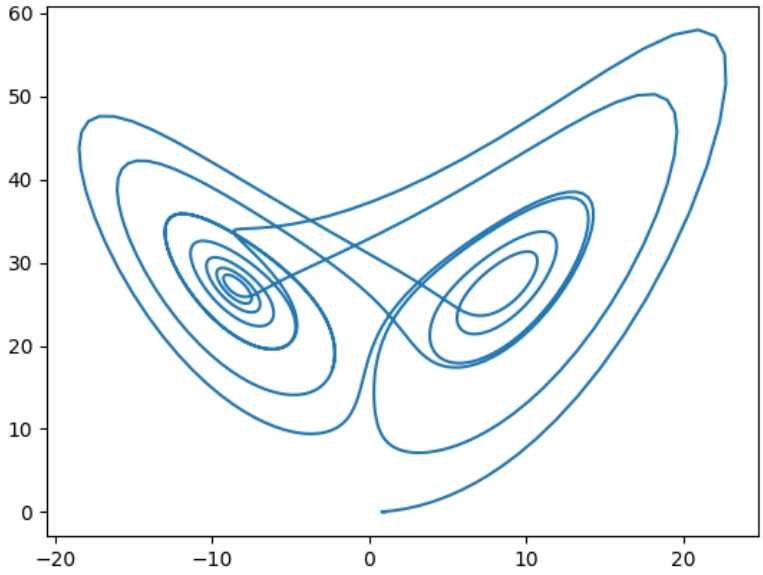

Figure 8: Lorenz System Attractor

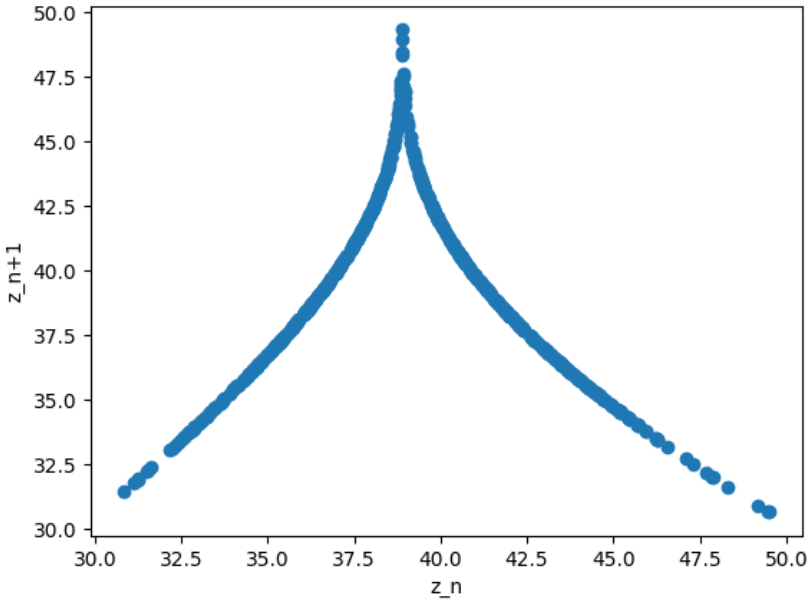

Figure 9: Lorenz Map

5 Q6: Logistic Map

```
import numpy as  np
import matplotlib.pyplot as plt
import sympy as sp
import networkx as nx
import random

#1
u, x, x0, iterations= sp.symbols('u x x0 iterations')
f= u * x * ( x -1)
u_substitution = f.subs({u: 1})
function_f = sp.lambdify(x, u_substitution, 'numpy')
x= np.linspace(-20, 20, 100)
f_values= function_f(x)
plt.plot(x, f_values)
plt.show()

#2
def logistic_map(u, x0, iterations):
    x=[x0]
    for i in range(40):
        x_plus= u * x0* ( x0 -1)
        x.append(x_plus)
    return x

random_number_generator= logistic_map(u, x0, iterations)

#3
def delta_alpha(logistic_map, u_list):
    maxima = []
    minima=[]
    for u in u_list:
        x_values = logistic_map(u, 0.5, 100)
        if np.max(x_values) != np.min(x_values):
           maxima.append(np.max(x_values))
           minima.append(np.min(x_values))
    doubling_index = np.argmax(np.diff(maxima))
    if doubling_index == len(u_list) - 1:
        return np.nan, np.nan

    delta = (u_list[doubling_index + 1] - u_list[doubling_index])/(u_list[doubling_index]- u

    alpha = (maxima[doubling_index]- minima[doubling_index])/ (maxima[doubling_index - 1]- m

    return delta, alpha
```

```
print(delta_alpha(logistic_map, np.arange(2.5, 20, 0.1)))
```

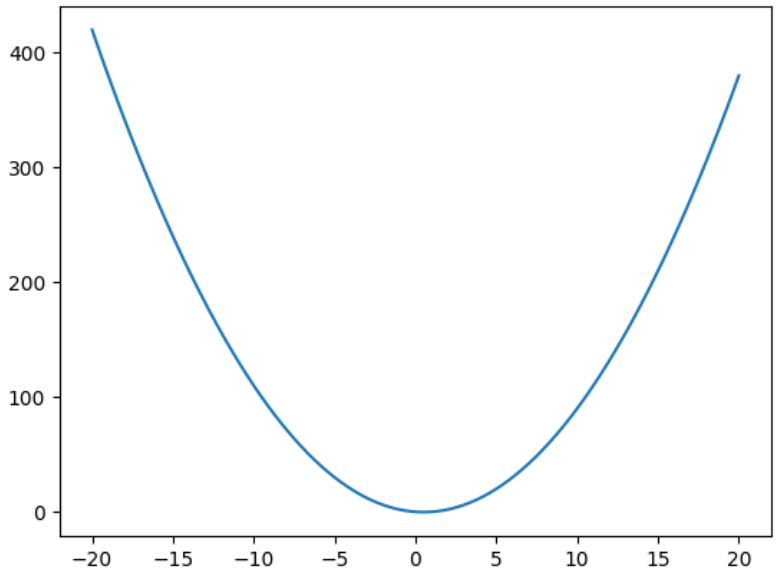

Figure 10: f(x)

As expected, if we changed the range of u-list from 4 to 20, both delta and alpha will decrease.

6 Q7: What do 1D maps have to do with Science?

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

a, b, i, x, y, z= sp.symbols('a b i x y z')
a=0.2
b= 0.2
c = [2.5, 3.5, 4, 5]
def f1(y, z):
            return - y - z
def f2(y, x):
            return x + a * y
def f3(x, z, i):
            return b + z * (x- i)
def RK4(f1, f2, f3, x0, y0, z0, t0, tn, h, i):
            t_values = [t0]
            x_values = [x0]
            y_values = [y0]
            z_values = [z0]
            while t_values[-1] <= tn:
                  t_n = t_values[-1]
                  x_n = x_values[-1]
                  y_n = y_values[-1]
                  z_n = z_values[-1]

                  k1x = h * f1(y_n, z_n)
                  k1y = h * f2(y_n, x_n)
                  k1z = h * f3(x_n, z_n, i)

                  k2x = h * f1(y_n, z_n)
                  k2y = h * f2(y_n + 0.5 * k1y, x_n)
                  k2z = h * f3(x_n , z_n + 0.5 * k1z, i)

                  k3x = h * f1(y_n, z_n)
                  k3y = h * f2(y_n + 0.5 * k2y, x_n)
                  k3z = h * f3(x_n, z_n + 0.5 * k2z, i)

                  k4x = h * f1(y_n, z_n)
                  k4y = h * f2(y_n + k3y, x_n)
                  k4z = h * f3(x_n, z_n + k3z, i)
```

```
                x_n1 = x_n + (k1x + 2 * k2x + 2 * k3x + k4x) / 6
                y_n1 = y_n + (k1y + 2 * k2y + 2 * k3y + k4y) / 6
                z_n1 = z_n + (k1z + 2 * k2z + 2 * k3z + k4z) / 6

                t_n1 = t_n + h

                t_values.append(t_n1)
                x_values.append(x_n1)
                y_values.append(y_n1)
                z_values.append(z_n1)

        return t_values, x_values, y_values, z_values
for i in c:
    t_values, x_values, y_values, z_values = RK4(f1, f2, f3, 1, 0, 0, 0, 1000, 0.001, i)
    #plt.plot(y_values, x_values, label=f'c={i}')
    #plt.show()

#2
t_values, x_values, y_values, z_values = RK4(f1, f2, f3, 1, 0, 0, 0, 1000, 0.001, 5)
x_maxima= []
for j in range(len(x_values)-1):
       if x_values[j] > x_values[j-1] and x_values[j]> x_values[j+1]:
               x_maxima.append(x_values[j])
x_n= x_maxima[:-1:2]
x_nplus = x_maxima[1::2]

#plt.scatter(x_n, x_nplus)
#plt.show()

#3
c2 = []
maxima2 = []
c1 = np.linspace(2.5, 6, 200)

for i in range(len(c1)):
    t_values, x_values, y_values, z_values = RK4(f1, f2, f3, 2, 2, 2, 0, 500, 0.001, c1[i])
    x_maxima = []

    for j in range(1, len(x_values)-1):
        if x_values[j] > x_values[j-1] and x_values[j] > x_values[j+1]:
            x_maxima.append(x_values[j])

    maxima2.extend(x_maxima[10:100])
    c2.extend([c1[i]] * (100-10))
c2 = c2[:len(maxima2)]
plt.scatter(c2, maxima2, marker='.', s=1, c='red', label='Bifurcation')
```

```
plt.xlabel('c')
plt.ylabel('Maxima of x')
plt.show()
```

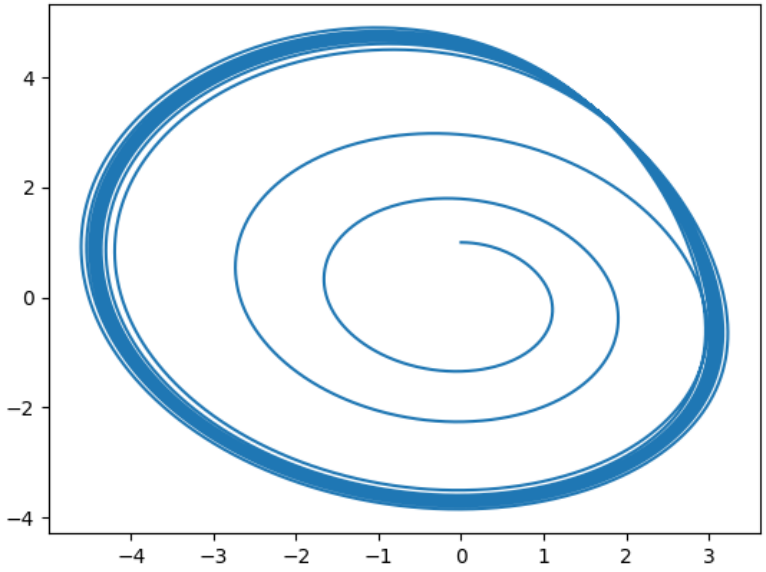

Figure 11: y versus x for c = 2.5

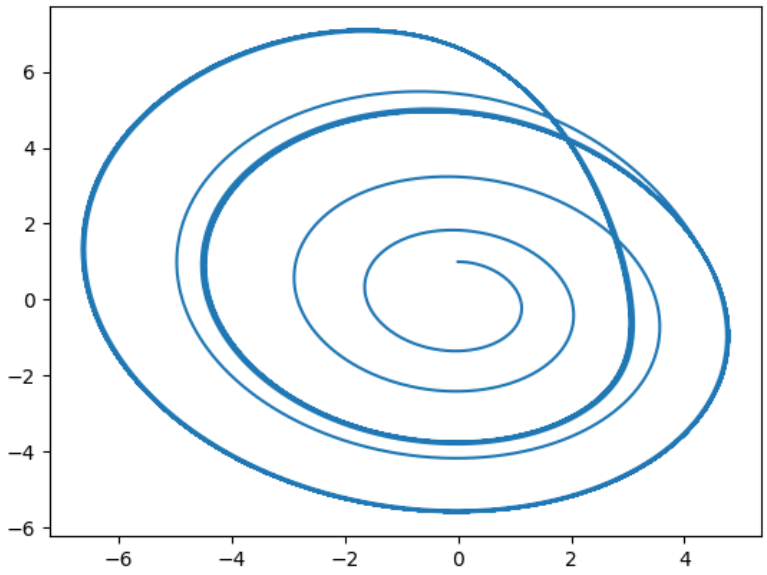

Figure 12: y versus x for c= 3.5

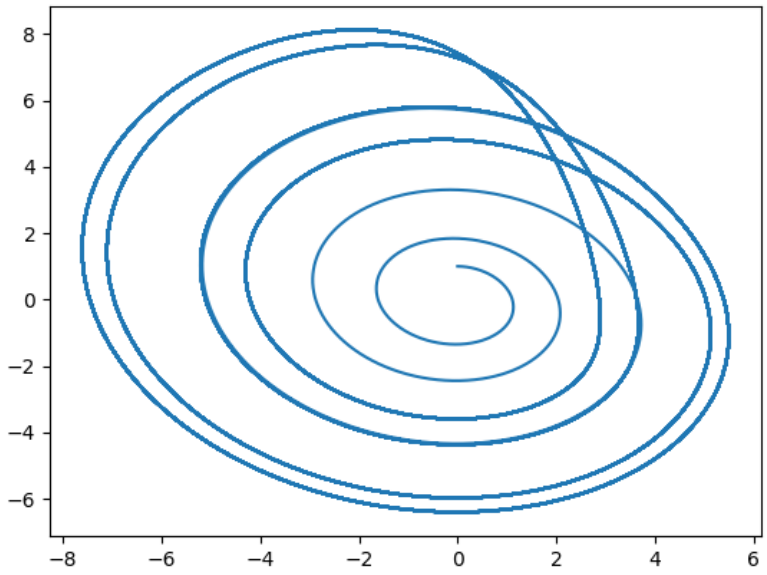

Figure 13: y versus x for c= 4

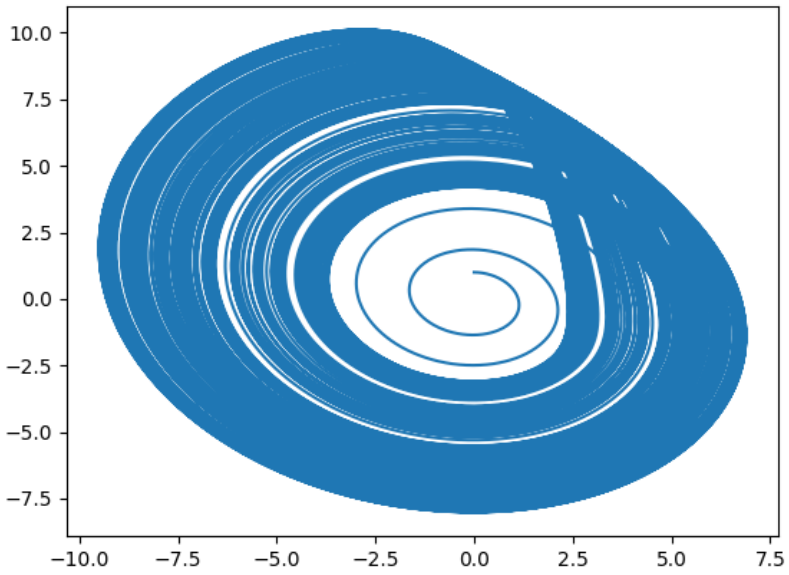

Figure 14: y versus x for c = 5

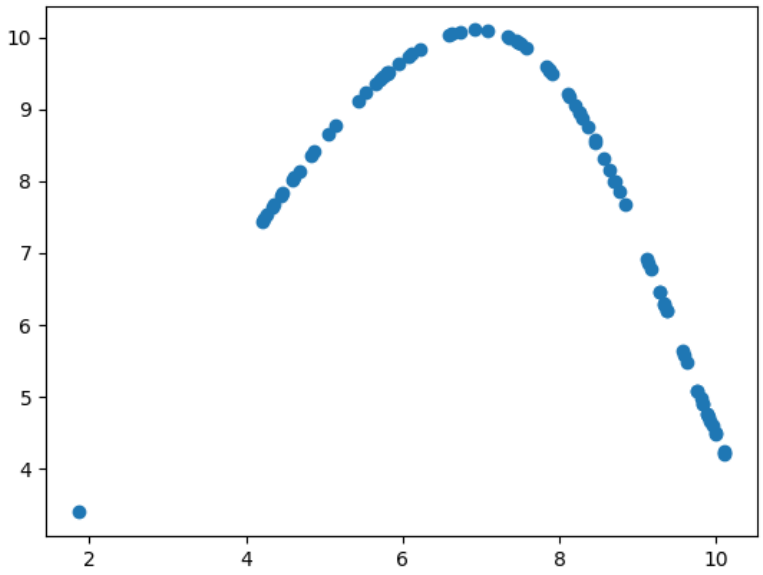

Figure 15: xn+1 versus xn

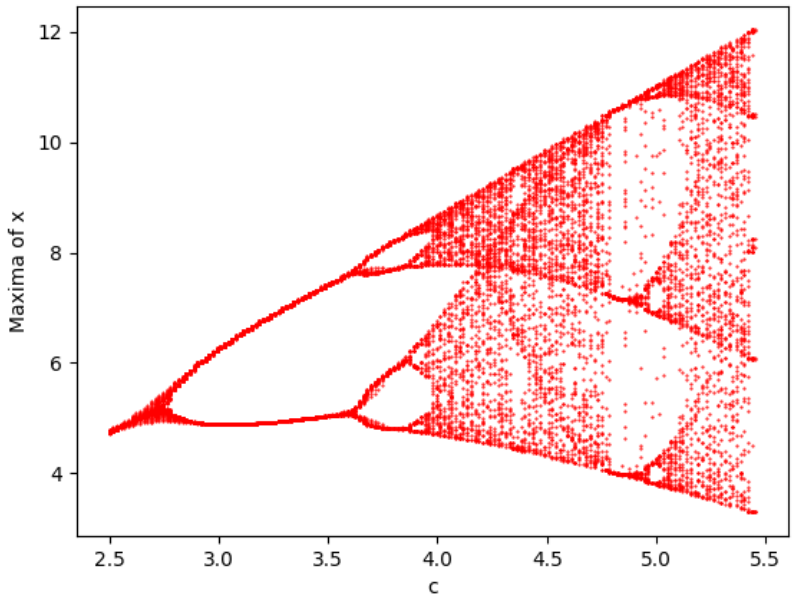

Figure 16: Bifurcation Diagram

7 Q8: SVD and Lagragian coherent Structures

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

x, y, t = sp.symbols('x y t')

A = 0.25
e = 0.1
w = (2 * sp.pi) / 10
domain_x = [0, 2]
domain_y = [0, 1]

def psie(x, y, t):
    return A * sp.sin(sp.pi * f(x, t)) * sp.sin(sp.pi * y)

def f(x, t):
    return a(t) * x **2 + b(t) * x

def a(t):
    return e * sp.sin(w * t)

def b(t):
    return 1 - 2 * e * sp.sin(w * t)

def u(x, y, t):
    return - (sp.pi * A * sp.sin(sp.pi * f(x, t)) * sp.cos(sp.pi * y))
def v(x, y, t):
    return A * sp.cos(sp.pi * f(x, t)) * sp.sin(sp.pi * y) * sp.pi * (2 * a(t) * x + b(t))



def u_v_generator(psie, f, a, b, t0, tn, mesh_x, mesh_y):
    t_values = np.arange(t0, tn + 1)  # Array of time values from t0 to tn
    mesh_u = np.zeros_like(mesh_x)
    mesh_v = np.zeros_like(mesh_y)

    # Define symbolic variables for x, y, and t
    sym_x, sym_y, sym_t = sp.symbols('sym_x sym_y sym_t')

    # Define symbolic expressions for u and v
    u_expr = u(sym_x, sym_y, sym_t)
    v_expr = v(sym_x, sym_y, sym_t)

    # Lambdify the expressions
```

```
    u_func = sp.lambdify((sym_x, sym_y, sym_t), u_expr, 'numpy')
    v_func = sp.lambdify((sym_x, sym_y, sym_t), v_expr, 'numpy')

    for t_val in t_values:
        # Evaluate u and v at each point in the mesh
        mesh_u += u_func(mesh_x, mesh_y, t_val)
        mesh_v += v_func(mesh_x, mesh_y, t_val)
    D = np.zeros((2, mesh_x.shape[0], mesh_x.shape[1], len(t_values)))
    for i, t_val in enumerate(t_values):
        D[0, :, :, i] = u_func(mesh_x, mesh_y, t_val)
        D[1, :, :, i] = v_func(mesh_x, mesh_y, t_val)
    return mesh_u, mesh_v, D

# Create a mesh of points
mesh_x, mesh_y = np.meshgrid(np.linspace(domain_x[0], domain_x[1], 100), np.linspace(domain_

# Call the modified u_v_generator function
mesh_u, mesh_v, D = u_v_generator(psie, f, a, b, 0, 200, mesh_x, mesh_y)

print(mesh_u)
print(mesh_v)
print(len(mesh_u))

#2
print(D)

#3
U, Sigma, Vt = np.linalg.svd(D, full_matrices=False)

Sigma_matrix = np.diagflat(Sigma)

print(Sigma_matrix)

#4
k=2
highest_singular_values = Sigma[:k]
Vt_reshaped = Vt.reshape(-1, 201, 200)
temporal_basis = Vt_reshaped[:k, :, :]
for i in range(2):
    plt.plot(np.arange(len(temporal_basis[0])), temporal_basis[i], 'r', label=f'Singular Val

plt.xlabel('Time Step')
plt.ylabel('Temporal Basis Value')
plt.title('Evolution of Temporal Basis for Highest Singular Values')
plt.legend()
plt.show()
```

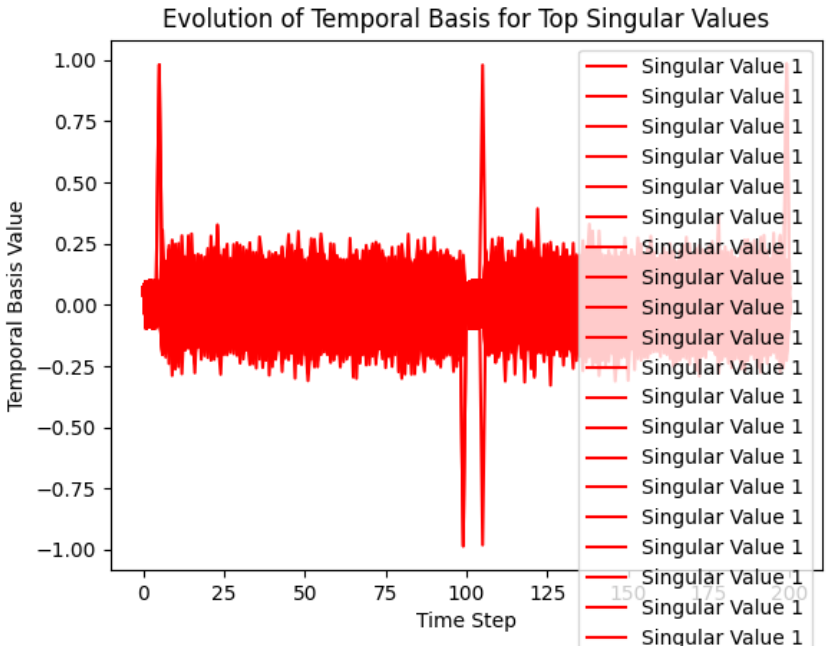

Figure 17: Evolution of Temporal Basis for Highest Singular Values