Linear Algebra

UE23CS241B

4th Semester, Academic Year 2023

Date: 04/04/2025

Name: Pranav Hemanth	SRN: PES1UG23CS433	Section: G

Linear Algebra Assignment 7-14

```
# Project 9: Orthogonal matrices and 3D graphics import numpy as np import matplotlib.pyplot as plt from mpl_toolkits.mplot3d import Axes3D import math import itertools from scipy.io import loadmat
```

```
print("\nSubtask 1-4\n") def
rotation(theta_x, theta_y, theta_z):
# Rotation matrix around the x-axis
    Rx = np.array([[1, 0, 0],
        [0, np.cos(theta_x), -np.sin(theta_x)],
[0, np.sin(theta_x), np.cos(theta_x)]])
    # Rotation matrix around the y-axis
    Ry = np.array([[np.cos(theta_y), 0, -np.sin(theta_y)],
        [0, 1, 0],
        [np.sin(theta_y), 0, np.cos(theta_y)]])
    # Rotation matrix around the z-axis
    Rz = np.array([[np.cos(theta_z), -np.sin(theta_z), 0],
        [np.sin(theta_z), np.cos(theta_z), 0],
        [0, 0, 1]])
    # Combined rotation matrix
rotmat = Rz @ Ry @ Rx     return
rotmat
```

Subtask 1-4

```
print("\nSubtask 5\n")
# Define cube vertices
Vertices = np.array([[1, 1, 1],
[-1, 1, 1],
[1, -1, 1],
[1, 1, -1],
[-1, -1, 1],
[-1, 1, -1],
[1, -1, -1],
[-1, -1, -1]])
# Define adjacency matrix (Edges)
Edges = np.zeros((8, 8))
Edges[0, 1] = 1
Edges[0, 2] = 1
Edges[0, 3] = 1
Edges[1, 4] = 1
Edges[1, 5] = 1
Edges[2, 4] = 1
Edges[2, 6] = 1
Edges[3, 5] = 1
Edges[3, 6] = 1
Edges[4, 7] = 1
Edges[5, 7] = 1
Edges[6, 7] = 1
Edges = Edges + Edges.T # Make the matrix symmetric
```

Subtask 5

```
print("\nSubtask 6\n") # Define rotation angles theta_x = np.pi / 3 # 60 degrees theta_y = np.pi / 4 # 45 degrees theta_z = np.pi / 6 # 30 degrees # Generate the rotation matrix rotmat = rotation(theta_x, theta_y, theta_z)
```

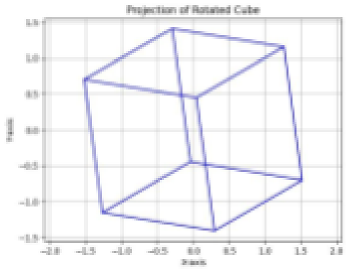
Subtask 6

```
print("\nSubtask 7\n")
# Rotate the vertices
VertRot = Vertices @ rotmat.T # Transpose the rotation matrix
```

Subtask 7

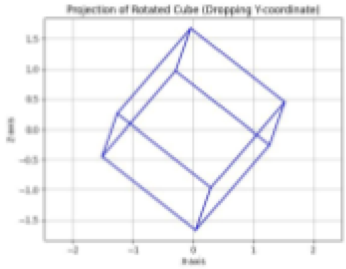
```
print("\nSubtask 8\n") # Create a new
figure window plt.figure()
plt.axis('equal') plt.title('Projection
of Rotated Cube') # Draw the projection
of the cube for j in range(8):
    for k in range(j + 1, 8): # Start with j + 1 to avoid repeating lines
if Edges[j, k] == 1:
        # Draw lines connecting the vertices (projecting by dropping the last coordinate)
plt.plot([VertRot[j, 0], VertRot[k, 0]], [VertRot[j, 1], VertRot[k, 1]], 'b-')
plt.xlabel('X-axis') plt.ylabel('Y-axis') plt.grid() plt.show()
```

```
Subtask 8
```



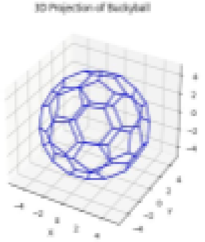
```
print("\nSubtask 9\n") # Create a new figure window plt.figure()
plt.axis('equal') plt.title('Projection of Rotated Cube (Dropping
Y-coordinate)') # Draw the projection of the cube by dropping the
Y-coordinate for j in range(8):     for k in range(j + 1, 8):
if Edges[j, k] == 1:
        # Draw lines connecting the vertices (projecting by dropping the Y coordinate)             plt.plot([VertRot[j, 0],
VertRot[k, 0]], [VertRot[j, 2], VertRot[k, 2]], 'b-') # Use the Z coordinate instead of Y plt.xlabel('X-axis') plt.ylabel('Z-axis')
# Update label to reflect the projection plt.grid() plt.show()
```

```
Subtask 9
```



```
print("\nSubtask 10-12\n") def
distance(a, b):
    '''Calculates the straight line distance between two points a and b.'''
return np.linalg.norm(np.array(a) - np.array(b)) def makecoords():
    '''Generate a list of coordinates for the buckyball.'''
phi = 0.5 * (1 + math.sqrt(5))     c1 = (0, 1, 3 * phi)
c2 = (2, (1 + 2 * phi), phi)     c3 = (1, 2 + phi, 2 * phi)
combos1 = list(itertools.product((1, -1), repeat=2))     for
i in range(len(combos1)):
        combos1[i] = (1,) + combos1[i]     combos23 =
list(itertools.product((1, -1), repeat=3))     coords = []
for i in combos1:
        coords.append(np.array(c1) * np.array(i))
for i in combos23:
        coords.append(np.array(c2) * np.array(i))
coords.append(np.array(c3) * np.array(i))
    # Permutation matrices
    P1 = np.array([[0, 0, 1], [1, 0, 0], [0, 1, 0]])
P2 = np.array([[0, 1, 0], [0, 0, 1], [1, 0, 0]])     for
i in coords[:]:
        coords.append(P1 @ i)
coords.append(P2 @ i)
return coords def
makeadjmat(coords):
    '''Make a 60x60 adjacency matrix for the coordinates.'''
D = np.zeros((60, 60))     for i in range(len(coords)):
for j in range(len(coords)):             if
distance(coords[i], coords[j]) == 2.0:                 D[i][j]
= 1     return D def rotation(theta_x, theta_y, theta_z):
    '''Create a rotation matrix based on the specified angles.'''
rot_x = np.array([[1, 0, 0],
    [0, np.cos(theta_x), -np.sin(theta_x)],     [0,
np.sin(theta_x), np.cos(theta_x)]])     rot_y =
np.array([[np.cos(theta_y), 0, np.sin(theta_y)],
    [0, 1, 0],
    [-np.sin(theta_y), 0, np.cos(theta_y)]])     rot_z =
np.array([[np.cos(theta_z), -np.sin(theta_z), 0],
    [np.sin(theta_z), np.cos(theta_z), 0],
    [0, 0, 1]])     return rot_z @ rot_y @ rot_x # Combined
rotation matrix def plot_buckyball(coords, edges, rotmat):
    '''Plot the 3D projection of the buckyball.'''     fig =
plt.figure()     ax = fig.add_subplot(111, projection='3d')
ax.set_title('3D Projection of Buckyball')     # Apply the rotation
matrix to the coordinates     rotated_coords = [np.dot(rotmat,
vertex) for vertex in coords]     num_vertices = len(coords)     for
j in range(num_vertices):
        for k in range(j + 1, num_vertices):
if edges[j, k] == 1:
                ax.plot([rotated_coords[j][0], rotated_coords[k][0]],
[rotated_coords[j][1], rotated_coords[k][1]],
[rotated_coords[j][2], rotated_coords[k][2]], 'b-')
ax.set_xlabel('X')     ax.set_ylabel('Y')     ax.set_zlabel('Z')
plt.show() if __name__ == "__main__":
    # Task 10: Generate coordinates for the buckyball
coords = makecoords()
    # Generate the adjacency matrix
edges = makeadjmat(coords)
    # Find and print the number of vertices     num_vertices =
len(coords)     print("Number of vertices in the array Vertices2:",
num_vertices)
    # Define rotation angles (in radians)
theta_x = np.pi / 3     theta_y = np.pi /
4     theta_z = np.pi / 6
    # Generate the rotation matrix     rotmat =
rotation(theta_x, theta_y, theta_z)     # Task 11: Plot
the 3D projection of the buckyball
plot_buckyball(coords, edges, rotmat)
```

```
Subtask 10-11

Number of vertices in the array Vertices: 60
```



```
print("\nSubtask 13\n") data_v = loadmat("/content/v.mat") data_f =
loadmat("/content/f.mat") # Extract the vertices (v) and faces (f) from the
loaded data v = data_v['v'] # Ensure the key matches the variable name in the
.mat file f = data_f['f'] # Ensure the key matches the variable name in the
.mat file
# Variables: v, f
print("Vertices (v):")
print(v)
print("\nFaces (f):")
print(f)
```

```
Subtask 13

Vertices (v):
[[ 58.64743805 111.18914032   6.92400026]
 [ 60.88143921 106.30313873  14.13500023]
 [ 67.99643707 114.31414032   8.17200089]  ...
 [ 87.3354187   43.88992691   1.28999996]
 [ 97.1484375   53.10213852   1.47399998]
 [ 88.33444214  45.02913666   3.69500017]]

Faces (f):
[[   1    2    3]
 [   4    5    6]
 [   7    8    9]  ...
 [1312 1313 1314]
 [1315 1316 1317]
 [1318 1319 1320]]
```

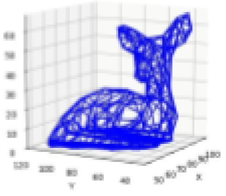
```
print("\nSubtask 14\n") mFaces, nFaces = f.shape # Get the number of
rows and columns in f
# Output the dimensions of f print("\nDimensions of the
face matrix f,") print("Number of faces (mFaces):",
mFaces) print("Number of vertices per face (nFaces):",
nFaces)
```

```
Subtask 14


Dimensions of the face matrix f,
Number of faces (mFaces): 440
Number of vertices per face (nFaces): 3
```

```
print("\nSubtask 15\n") # Get the number of faces mFaces =
f.shape[0] # Generate the 3D model fig = plt.figure() ax =
fig.add_subplot(111, projection='3d')
ax.set_box_aspect([1, 1, 1]) # Set aspect ratio to equal
# Loop through each face and plot the edges for
j in range(mFaces):
# Draw lines between the vertices of each face     ax.plot([v[f[j, 0] - 1, 0], v[f[j, 1] - 1,
0]], [v[f[j, 0] - 1, 1], v[f[j, 1] - 1, 1]],     [v[f[j, 0] - 1, 2], v[f[j, 1] - 1, 2]],
color='b') # Edge between vertex 1 and 2     ax.plot([v[f[j, 0] - 1, 0], v[f[j, 2] - 1, 0]],
[v[f[j, 0] - 1, 1], v[f[j, 2] - 1, 1]],     [v[f[j, 0] - 1, 2], v[f[j, 2] - 1, 2]], color='b')
# Edge between vertex 1 and 3     ax.plot([v[f[j, 1] - 1, 0], v[f[j, 2] - 1, 0]], [v[f[j, 1] -
1, 1], v[f[j, 2] - 1, 1]],
```

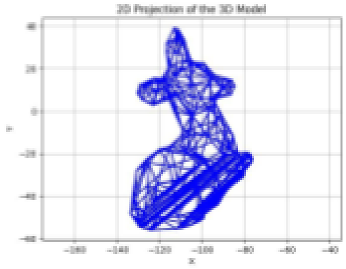
```
    [v[f[j, 1] - 1, 2], v[f[j, 2] - 1, 2]], color='b') # Edge between vertex 2 and 3
# Set labels for the axes
ax.set_xlabel('X') ax.set_ylabel('Y')
ax.set_zlabel('Z')
# Set the viewpoint (azimuth, elevation) ax.view_init(elev=10,
azim=210) # You can change the angles here
# Show the 3D plot plt.show()
```

```
Subtask 15
```



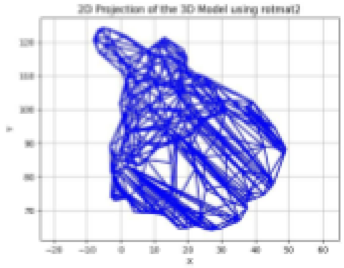
```
print("\nSubtask 16\n") theta1 = np.pi / 3 #
Rotation around x-axis theta2 = np.pi / 4 #
Rotation around y-axis theta3 = np.pi #
Rotation around z-axis
# Generate the rotation matrix rotmat =
rotation(theta1, theta2, theta3)
# Transform the coordinates of the vertices with the rotation matrix
VertRot = v @ rotmat.T # Apply rotation
# Create a new figure window for the 2D projection
plt.figure() plt.axis('equal') plt.title("2D
Projection of the 3D Model")
# Plot the 2D projection by connecting the edges defined in f for
j in range(f.shape[0]):
    plt.plot([VertRot[f[j, 0] - 1, 0], VertRot[f[j, 1] - 1, 0]],
[VertRot[f[j, 0] - 1, 1], VertRot[f[j, 1] - 1, 1]], color='b')
plt.plot([VertRot[f[j, 0] - 1, 0], VertRot[f[j, 2] - 1, 0]],     [VertRot[f[j,
0] - 1, 1], VertRot[f[j, 2] - 1, 1]], color='b')     plt.plot([VertRot[f[j, 1] -
1, 0], VertRot[f[j, 2] - 1, 0]],
    [VertRot[f[j, 1] - 1, 1], VertRot[f[j, 2] - 1, 1]], color='b')
# Set labels for the axes
plt.xlabel('X')
plt.ylabel('Y')
plt.grid() plt.show()
```

```
Subtask 16
```



```
print("\nSubtask 17\n")
# Define the new rotation angles theta1 =
-np.pi / 3 # Rotation around x-axis theta2 =
0 # No rotation around y-axis theta3 = np.pi
/ 4 # Rotation around z-axis
# Generate the rotation matrix rotmat2 =
rotation(theta1, theta2, theta3)
# Rotate the vertices vRot = v @
rotmat2.T # Apply rotation
# Project to the XY plane vPrj = vRot[:, :2] # Keep only the
first two coordinates # Create a new figure window for the 2D
projection plt.figure() plt.axis('equal') plt.title("2D
Projection of the 3D Model using rotmat2") # Plot the 2D
projection by connecting the edges defined in f for j in
range(f.shape[0]):
    plt.plot([vPrj[f[j, 0] - 1, 0], vPrj[f[j, 1] - 1, 0]],
[vPrj[f[j, 0] - 1, 1], vPrj[f[j, 1] - 1, 1]], color='b')
plt.plot([vPrj[f[j, 0] - 1, 0], vPrj[f[j, 2] - 1, 0]],     [vPrj[f[j,
0] - 1, 1], vPrj[f[j, 2] - 1, 1]], color='b')     plt.plot([vPrj[f[j,
1] - 1, 0], vPrj[f[j, 2] - 1, 0]],     [vPrj[f[j, 1] - 1, 1],
vPrj[f[j, 2] - 1, 1]], color='b')
# Set labels for the axes
plt.xlabel('X')
plt.ylabel('Y')
plt.grid() plt.show()
```

```
Subtask 17
```



```
# Project 10: Discrete dynamical systems, linear transformations of the plane, and the Chaos Game import numpy as np import matplotlib.pyplot as plt
```

```
print("\nSubtask 1\n")
# Generate linearly spaced values
t = np.linspace(0, 2 * np.pi, 4) #
Remove the fourth element t =
np.delete(t, 3)
# Define the vertices of the equilateral triangle v
= np.array([np.cos(t), np.sin(t)])
```

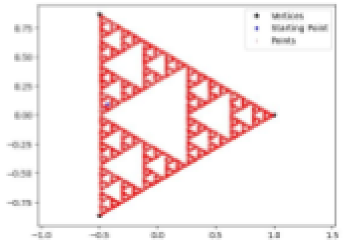
```
Subtask 1
```

```
print("\nSubtask 2\n") # Define the linear transformation matrix
T = np.array([[0.5, 0], [0, 0.5]])
# Define the starting point with random values between -0.5 and 0.5 x = np.random.rand(2, 1) - 0.5
```

```
Subtask 2
```

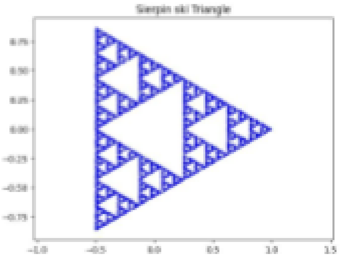
```
print("\nSubtask 3\n")
# Define the vertices of the triangle t = np.linspace(0, 2 * np.pi, 4)[:-1] #
Generate t and remove the last element v = np.array([np.cos(t), np.sin(t)])
# Define the linear transformation matrix
T = np.array([[0.5, 0], [0, 0.5]])
# Define the starting point x =
np.random.rand(2, 1) - 0.5
# Number of iterations
Num = 10000
# Create an array to store all points points
= np.zeros((2, Num))
# Initial point
current_point = x #
Iterative process
for i in range(Num):
    # Store the current point     points[:,
i] = current_point.flatten()
    # Choose a random vertex
    vertex = v[:, np.random.choice(3)]
    # Apply the linear transformation and update the point
current_point = T @ (current_point + vertex.reshape(2, 1))
# Plot the vertices of the triangle
plt.plot(v[0, :], v[1, :], 'k*', label='Vertices')
# Plot the starting point plt.plot(x[0, 0], x[1, 0], 'b.', label='Starting
Point') # Plot all the points obtained during the iterations
plt.plot(points[0, :], points[1, :], 'r.', markersize=1, label='Points')
plt.axis('equal') plt.legend() plt.show()
```

```
Subtask 3
```



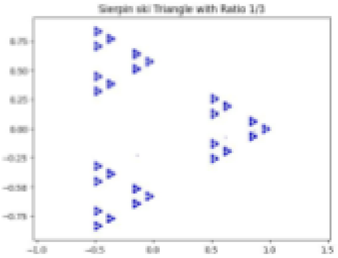
```
print("\nSubtask 4\n")
# Define the vertices of the triangle t = np.linspace(0, 2 * np.pi, 4)[:-1] #
Generate t and remove the last element v = np.array([np.cos(t), np.sin(t)])
# Define the linear transformation matrix
T = np.array([[0.5, 0], [0, 0.5]])
# Define the starting point x =
np.random.rand(2, 1) - 0.5
# Number of iterations
Num = 10000
# Create an array to store all points points
= np.zeros((2, Num + 1))
# Initial point points[:,
0] = x.flatten() #
Iterative process for j in
range(Num):
    k = np.random.randint(0, 3) # Random integer from 0 to 2
    # Perform the transformation     current_point = points[:, j]
# Get the current point     transformed_point = T @
(current_point - v[:, k]) + v[:, k]     points[:, j + 1] =
transformed_point
# Plot the points plt.plot(points[0, :], points[1, :],
'b.', markersize=1) plt.axis('equal') plt.title('Sierpin
ski Triangle') plt.show()
```

```
Subtask 4
```



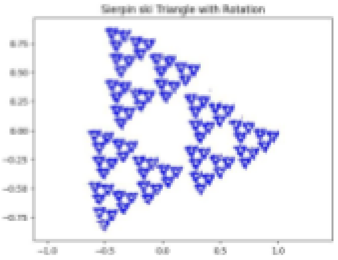
```
print("\nSubtask 5\n")
# Define the vertices of the triangle t = np.linspace(0, 2 * np.pi, 4)[:-1] #
Generate t and remove the last element v = np.array([np.cos(t), np.sin(t)])
# Define the new linear transformation matrix with a ratio of 1/3
T = np.array([[1/3, 0], [0, 1/3]])
# Define the starting point x =
np.random.rand(2, 1) - 0.5
# Number of iterations
Num = 10000
# Create an array to store all points points
= np.zeros((2, Num + 1))
# Initial point points[:,
0] = x.flatten() #
Iterative process for j in
range(Num):
    k = np.random.randint(0, 3) # Random integer from 0 to 2
    # Perform the transformation     current_point = points[:, j]
# Get the current point     transformed_point = T @
(current_point - v[:, k]) + v[:, k]     points[:, j + 1] =
transformed_point
# Plot the points plt.plot(points[0, :], points[1, :],
'b.', markersize=1) plt.axis('equal') plt.title('Sierpin
ski Triangle with Ratio 1/3') plt.show()
```

```
Subtask 5
```



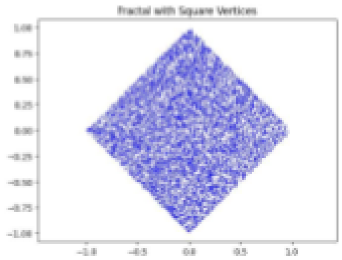
```
print("\nSubtask 6\n")
# Define the vertices of the triangle t = np.linspace(0, 2 * np.pi, 4)[:-1] #
Generate t and remove the last element v = np.array([np.cos(t), np.sin(t)])
# Define the rotation angle and transformation matrix T theta
= np.pi / 18
T = 0.5 * np.array([[np.cos(theta), -np.sin(theta)],
[np.sin(theta), np.cos(theta)]])
# Define the starting point x =
np.random.rand(2, 1) - 0.5
# Number of iterations
Num = 10000
# Create an array to store all points points
= np.zeros((2, Num + 1))
# Initial point points[:,
0] = x.flatten() #
Iterative process for j in
range(Num):
    k = np.random.randint(0, 3) # Random integer from 0 to 2
    # Perform the transformation     current_point = points[:, j]
# Get the current point     transformed_point = T @
(current_point - v[:, k]) + v[:, k]     points[:, j + 1] =
transformed_point
# Plot the points
plt.plot(points[0, :], points[1, :], 'b.', markersize=1)
plt.axis('equal') plt.title('Sierpin ski Triangle with
Rotation') plt.show()
```

```
Subtask 6
```



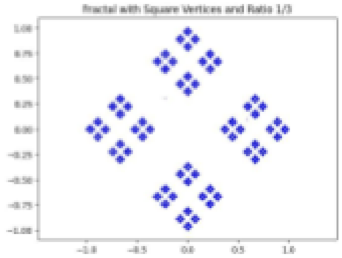
```
print("\nSubtask 7\n")
# Define the vertices of the square t = np.linspace(0, 2 * np.pi, 5)[:-1] #
Generate t and remove the last element v = np.array([np.cos(t), np.sin(t)])
# Define the linear transformation matrix
T = np.array([[0.5, 0], [0, 0.5]])
# Define the starting point x =
np.random.rand(2, 1) - 0.5
# Number of iterations
Num = 10000
# Create an array to store all points points
= np.zeros((2, Num + 1))
# Initial point points[:,
0] = x.flatten() #
Iterative process for j in
range(Num):
    k = np.random.randint(0, 4) # Random integer from 0 to 3 (for 4 vertices)
    # Perform the transformation     current_point = points[:, j]
# Get the current point     transformed_point = T @
(current_point - v[:, k]) + v[:, k]     points[:, j + 1] =
transformed_point
# Plot the points plt.figure() plt.plot(points[0, :],
points[1, :], 'b.', markersize=1) plt.axis('equal')
plt.title('Fractal with Square Vertices') plt.show()
```

```
Subtask 7
```



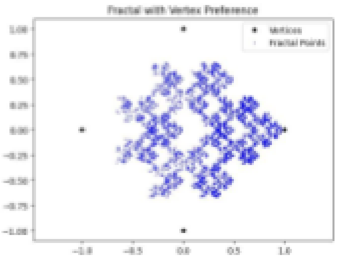
```
print("\nSubtask 8\n")
# Define the vertices of the square t = np.linspace(0, 2 * np.pi, 5)[:-1] #
Generate t and remove the last element v = np.array([np.cos(t), np.sin(t)])
# Define the new linear transformation matrix with a ratio of 1/3
T = np.array([[1/3, 0], [0, 1/3]])
# Define the starting point x =
np.random.rand(2, 1) - 0.5
# Number of iterations
Num = 10000
# Create an array to store all points points
= np.zeros((2, Num + 1))
# Initial point points[:,
0] = x.flatten() #
Iterative process for j in
range(Num):
    k = np.random.randint(0, 4) # Random integer from 0 to 3 (for 4 vertices)
    # Perform the transformation     current_point = points[:, j]
# Get the current point     transformed_point = T @
(current_point - v[:, k]) + v[:, k]     points[:, j + 1] =
transformed_point
# Plot the points plt.figure() plt.plot(points[0, :],
points[1, :], 'b.', markersize=1) plt.axis('equal')
plt.title('Fractal with Square Vertices and Ratio 1/3')
plt.show()
```

```
Subtask 8
```



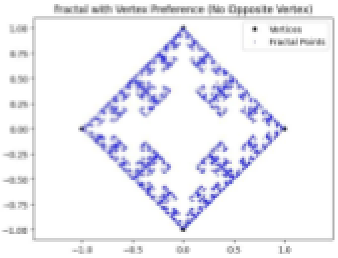
```
print("\nSubtask 9\n")
# Define the vertices of the square using complex exponentials t = np.linspace(0,
2 * np.pi, 5)[:-1] # Generate t and remove the last element v = np.exp(1j * t) #
Complex exponential for the vertices
# Define the linear transformation matrix
T = np.array([[0.5, 0], [0, 0.5]])
# Define the starting point x =
np.random.rand(2, 1) - 0.5
# Number of iterations
Num = 5000
# Create an array to store all points points
= np.zeros((2, Num + 1))
# Initial point points[:,
0] = x.flatten()
# Variable to keep track of the previous vertex index k1
= -1
# Iterative process
for j in range(Num):
    k = np.random.randint(0, 4) # Random integer from 0 to 3
# Ensure the same vertex is not selected twice in a row     if
k >= k1:
        k = (k + 1) % 4 # Move to the next vertex if k >= k1
    # Get the current vertex     w =
np.array([np.real(v[k]), np.imag(v[k])])
    # Perform the transformation     current_point = points[:, j] # Get the current
point     transformed_point = T @ (current_point - w) + w     points[:, j + 1] =
transformed_point     # Update the previous vertex index     k1 = k # Plot the
vertices and the points plt.figure() plt.plot(np.real(v), np.imag(v), 'k*',
label='Vertices') # Plot vertices plt.plot(points[0, :], points[1, :], 'b.',
markersize=1, label='Fractal Points') plt.axis('equal') plt.title('Fractal with
Vertex Preference') plt.legend() plt.show()
```

```
Subtask 9
```



```
print("\nSubtask 10\n")
# Define the vertices of the square using complex exponentials t = np.linspace(0,
2 * np.pi, 5)[:-1] # Generate t and remove the last element v = np.exp(1j * t) #
Complex exponential for the vertices
# Define the linear transformation matrix
T = np.array([[0.5, 0], [0, 0.5]])
# Define the starting point x =
np.random.rand(2, 1) - 0.5
# Number of iterations
Num = 5000
# Create an array to store all points points
= np.zeros((2, Num + 1))
# Initial point points[:,
0] = x.flatten()
# Variable to keep track of the previous vertex index
k1 = 0 w = np.array([np.real(v[k1]), np.imag(v[k1])])
# Iterative process
for j in range(Num):
    k = np.random.randint(0, 4) # Random integer from 0 to 3
# Ensure the new vertex is not opposite to the previous one
if (k != (k1 + 2) % 4) and ((k1 != (k + 2) % 4)):         w =
np.array([np.real(v[k]), np.imag(v[k])])
transformed_point = T @ (points[:, j] - w) + w         points[:,
j + 1] = transformed_point         k1 = k # Update the previous
vertex index     else:
        points[:, j + 1] = points[:, j] # If opposite, repeat the current point
# Plot the vertices and the points plt.figure() plt.plot(np.real(v), np.imag(v),
'k*', label='Vertices') # Plot vertices plt.plot(points[0, :], points[1, :], 'b.',
markersize=1, label='Fractal Points') plt.axis('equal') plt.title('Fractal with
Vertex Preference (No Opposite Vertex)') plt.legend() plt.show()
```

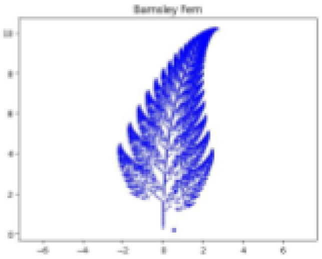
```
Subtask 10
```



```
print("\nSubtask 11\n") # Define the transformation matrices and translation vectors
T1 = np.array([[0.85, 0.04], [-0.04, 0.85]])
T2 = np.array([[-0.15, 0.28], [0.26, 0.24]])
T3 = np.array([[0.2, -0.26], [0.23, 0.22]])
T4 = np.array([[0, 0], [0, 0.16]])
Q1 = np.array([0, 1.64])
Q2 = np.array([-0.028, 1.05])
Q3 = np.array([0, 1.6])
Q4 = np.array([0, 0])
P1 = 0.85
P2 = 0.07
P3 = 0.07
```

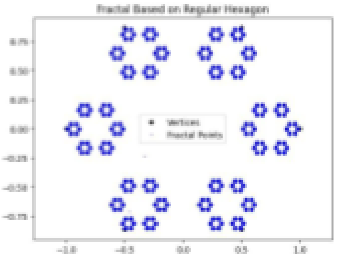
```
P4 = 0.01
Num = 15000
# Initialize the starting point x =
np.zeros((2, Num + 1)) x[:, 0] =
np.random.rand(2) # Starting point
# Plot the initial point
plt.figure() plt.plot(x[0, 0],
x[1, 0], 'b.')
# Iterative process to generate the fern for
j in range(Num):
    r = np.random.rand()
if r <= P1:
        x[:, j + 1] = T1 @ x[:, j] + Q1
elif r <= P1 + P2:
        x[:, j + 1] = T2 @ x[:, j] + Q2
elif r <= P1 + P2 + P3:
        x[:, j + 1] = T3 @ x[:, j] + Q3
else:
        x[:, j + 1] = T4 @ x[:, j] + Q4
# Plot the fractal plt.plot(x[0, :], x[1, :],
'b.', markersize=1) plt.axis('equal')
plt.title('Barnsley Fern') plt.show()
```

```
Subtask 11
```



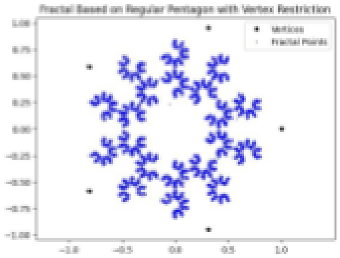
```
print("\nSubtask 12\n")
# Define the vertices of a regular hexagon t = np.linspace(0, 2 * np.pi, 7)[:-1] #
Generate 6 vertices and remove the last point v = np.array([np.cos(t), np.sin(t)]) #
Calculate vertices
# Define the linear transformation matrix
T = np.array([[0.25, 0], [0, 0.25]])
# Define the number of iterations
Num = 15000
# Initialize the starting point x = np.zeros((2, Num + 1))
x[:, 0] = np.random.rand(2) - 0.5 # Random starting point
# Create an array to store all points
points = np.zeros((2, Num + 1)) points[:,
0] = x[:, 0]
# Iterative process to generate the fractal for
j in range(Num):
    k = np.random.randint(0, 6) # Randomly choose one of the 6 vertices
points[:, j + 1] = T @ (points[:, j] - v[:, k]) + v[:, k]
# Plot the vertices and the points plt.figure()
plt.plot(v[0, :], v[1, :], 'k*', label='Vertices') # Plot vertices of hexagon plt.plot(points[0,
:], points[1, :], 'b.', markersize=1, label='Fractal Points') # Plot fractal points
plt.axis('equal') plt.title('Fractal Based on Regular Hexagon') plt.legend() plt.show()
```

```
Subtask 12
```



```
print("\nSubtask 13\n")
# Define the vertices of a regular pentagon t =
np.linspace(0, 2 * np.pi, 6)[:-1] # Generate 5 vertices v =
np.array([np.cos(t), np.sin(t)]) # Calculate vertices
# Define the linear transformation matrix
T = np.array([[2/5, 0], [0, 2/5]])
# Define the number of iterations
Num = 10000
# Initialize the starting point x = np.zeros((2, Num + 1))
x[:, 0] = np.random.rand(2) - 0.5 # Random starting point
# Create an array to store all points
points = np.zeros((2, Num + 1)) points[:,
0] = x[:, 0]
# Initialize previous vertex index
prev_vertex_index = np.random.randint(0, 5) #
Iterative process to generate the fractal for
j in range(Num):
    # Randomly choose one of the 5 vertices but not the same as the previous one
current_vertex_index = prev_vertex_index     while current_vertex_index ==
prev_vertex_index:          current_vertex_index = np.random.randint(0, 5)
    # Apply the transformation     w = v[:,
current_vertex_index]     points[:, j + 1] = T @
(points[:, j] - w) + w
    # Update the previous vertex index     prev_vertex_index = current_vertex_index # Plot the
vertices and the points plt.figure() plt.plot(v[0, :], v[1, :], 'k*', label='Vertices') # Plot
vertices of pentagon plt.plot(points[0, :], points[1, :], 'b.', markersize=1, label='Fractal
Points') # Plot fractal points plt.axis('equal') plt.title('Fractal Based on Regular Pentagon with
Vertex Restriction') plt.legend() plt.show()
```

```
Subtask 13
```



```
# Project 11: Projections, eigenvectors, Principal Component Analysis, and face recognition algorithms import numpy as np from PIL import Image import os import numpy as np from PIL import Image import os import matplotlib.pyplot as plt
```

```
print("\nSubtask 1\n")
# Parameters Database_Size = 30
database_path = "/content/database"



# Initialize list to store image vectors
P = []
# Reading images from the database for
j in range(1, Database_Size + 1):
    image_path = os.path.join(database_path, f'person{j}.pgm')
image = Image.open(image_path)     image_array = np.array(image)
# Get dimensions of the image     m, n = image_array.shape
    # Reshape the image array to a column vector
image_vector = image_array.reshape(m * n, 1)
    P.append(image_vector)
# Convert list to numpy array (matrix)
P = np.hstack(P)
# Print out the variables for verification
print(f"Database Size: {Database_Size}") print(f"Image
dimensions (m, n): ({m}, {n})") print(f"P matrix shape:
{P.shape}")
```

```
Subtask 1

Database Size: 30
Image dimensions (m, n): (112, 92)
P matrix shape: (10304, 30)
```

```
print("\nSubtask 2\n") #
Compute the mean face mean_face
= np.mean(P, axis=1)
# Reshape the mean face back to the original image dimensions mean_face_image =
mean_face.reshape(m, n)
# Display the mean face image
plt.imshow(mean_face_image, cmap='gray')
plt.title('Mean Face') plt.axis('off') #
Hide axis plt.show()
```

```
Subtask 2
```

```
print("\nSubtask 3\n") #
Compute the mean face mean_face
= np.mean(P, axis=1)
# Convert P to double (float64 in numpy)
P = P.astype(np.float64)
# Subtract the mean face from each column of P mean_face_column =
mean_face.reshape(-1, 1)
P = P - mean_face_column @ np.ones((1, Database_Size)) #
Print the first column of P to verify subtraction
print(P[:, 0])
```

```
Subtask 3

[-38.8        -38.3        -42.73333333 ... -14.56666667 -14.4
 -16.33333333]
```

```
print("\nSubtask 4\n")
# Compute the covariance matrix P^T * P
PTP = P.T @ P
# Compute the eigenvalues and eigenvectors of P^T * P
Values, Vectors = np.linalg.eig(PTP)
# Compute the actual eigenvectors of the covariance matrix
EigenVectors = P @ Vectors
# Normalize the eigenvectors
EigenVectors = EigenVectors / np.linalg.norm(EigenVectors, axis=0) # Display the first few eigenvalues for verification print("Eigenvalues:", Values)
```

```
Subtask 4

Eigenvalues: [9.69069623e+07 4.87292722e+07 4.37900456e+07 3.34756826e+07
 2.46521889e+07 2.04782347e+07 1.63378623e+07 1.60131047e+07  1.41415577e+07 1.32446059e+07 5.46447125e-09 1.11004779e+07  1.06792480e+07 9.84547441e+06 9.27154963e+06 8.44340124e+06  7.68223964e+06 3.30696241e+06 3.42829390e+06 3.52946256e+06  3.76629399e+06 4.17775602e+06 7.12503632e+06
6.92200481e+06
 4.74323775e+06 4.98622750e+06 5.37363278e+06 5.60944036e+06
 6.08634551e+06 6.37529704e+06]
```

```
print("\nSubtask 5\n")
# Display the set of eigenfaces
eigenfaces = [] for j in range(1,
Database_Size):
    eigenface = EigenVectors[:, j] + mean_face
eigenface_image = eigenface.reshape(m, n)
eigenfaces.append(eigenface_image)
# Concatenate the eigenfaces horizontally
EigenFaces = np.hstack(eigenfaces)
# Display the eigenfaces
plt.figure(figsize=(15, 5))
plt.imshow(EigenFaces, cmap='gray')
plt.title('Eigenfaces')
plt.axis('off') # Hide axis plt.show()
```

```
Subtask 5
```

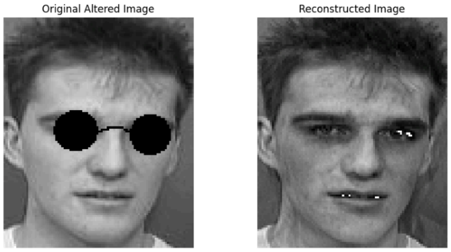
```
print("\nSubtask 6\n")
# Compute the Products matrix
Products = EigenVectors.T @ EigenVectors
# Print the Products matrix to verify orthogonalit  y
print("Products matrix:")
print(Products)
# Check if Products matrix is diagonal
is_diagonal = np.allclose(Products, np.diag(np.diagonal(Products)))
print(f"Is Products matrix diagonal? {is_diagonal}")
```

```
Subtask 6

Products matrix:
[illegible]
```

```
 [ 1.87436872e-15  8.48279780e-16  1.00000000e+00  6.33174069e-16
  -1.02348685e-15 -5.63785130e-17  2.63677968e-16  1.77375475e-16
   3.37837397e-16 -9.28077060e-17 -2.59362082e-01 -2.32452946e-16
   3.28730099e-16 -2.41126563e-16  1.35308431e-16  1.38777878e-16
  -5.76795556e-17 -1.98398156e-15 -1.21430643e-16 -2.86229374e-17
   4.29344060e-16  4.15466272e-16 -1.80411242e-16 -4.62303806e-16
   1.76941795e-16 -3.17454396e-16  2.25514052e-16 -1.97758476e-16
  -1.12973866e-16 -1.29887420e-16]
 [-2.60425362e-16 -2.55654872e-16  6.33174069e-16  1.00000000e+00
   1.61329283e-16 -1.10068205e-15 -2.09901541e-16  2.23562488e-16
   2.82759927e-16 -3.55618313e-16  2.21955321e-02  5.58580959e-16
   1.43982049e-16 -2.89698820e-16  3.73832909e-16 -5.30825384e-16
  -2.98372438e-16  5.06539255e-16  9.85322934e-16 -9.21138166e-16
  -7.11236625e-16 -2.41560244e-16  7.85938155e-16  8.58688121e-17
  -2.06432094e-16  4.11129464e-16  1.42247325e-16  7.19910243e-17
  -1.46584134e-16 -4.46691295e-17]
 [-3.95516953e-16  7.02563008e-17 -1.02348685e-15  1.61329283e-16
   1.00000000e+00  1.74339709e-16 -8.74300632e-16 -3.74700271e-16
   1.47451495e-17 -2.81675724e-16 -6.93256420e-02 -4.32813507e-16
  -2.49800181e-16 -4.75747913e-16  2.77555756e-17 -2.15105711e-16
  -1.69135539e-16  2.02962647e-16  2.41777084e-17 -3.48679419e-16
   1.79110199e-16  1.82145965e-17 -5.49039980e-16  2.16840434e-16
  -1.42247325e-16  1.37043155e-16 -2.25514052e-17  2.48065457e-16
  -1.70761842e-16 -8.95034304e-17]
 [-2.07299455e-16  2.46330734e-16 -5.63785130e-17 -1.10068205e-15
   1.74339709e-16  1.00000000e+00 -3.96384314e-16  1.11022302e-16
  -2.75821033e-16 -7.28583860e-17  1.70553510e-01 -9.62771529e-17
   2.32452946e-16  7.19910243e-17 -3.85975973e-16  2.09034179e-16
   8.50014503e-17 -2.09251019e-16  8.82540568e-17 -2.44596010e-16
  -4.27175656e-16  1.60461922e-16  2.46764414e-16  8.86877377e-17
   5.06539255e-16  7.80625564e-18 -3.55618313e-16 -2.67581096e-16
  -1.42247325e-16 -3.13600396e-16]
 [-6.19621542e-17  2.99239800e-16  2.63677968e-16 -2.09901541e-16
  -8.74300632e-16 -3.96384314e-16  1.00000000e+00  3.69582837e-15
  -2.04697370e-16 -2.65412692e-16  4.15913906e-03 -2.07678926e-16
  -4.34981912e-16 -3.55618313e-16  4.33680869e-17 -1.85615412e-16
   4.17200996e-16 -7.25114413e-16  7.87564458e-16  4.07660017e-17
  -5.16080234e-16 -8.80372164e-17  5.20417043e-18 -6.28620420e-16
```

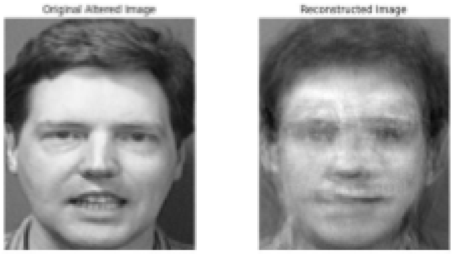
```
print("\nSubtask 7\n")
# Define image dimensions
m, n = 112, 92
# Read the altered image
altered_image_path ="/content/database/person30altered1.pgm"
image_read = Image.open(altered_image_path)
image_array = np.array(image_read)
U = image_array.reshape(m * n, 1)
# Compute the norms of the eigenvectors
Products = EigenVectors.T @ EigenVectors
NormsEigenVectors = np.diag(Products)
# Compute the projection coefficients
W = EigenVectors.T @ (U.astype(np.float64) - mean_face.reshape(-1, 1))
W = W / NormsEigenVectors.reshape(-1, 1) # Ensure proper division
# Reconstruct the image from the projection
U_approx = EigenVectors @ W + mean_face.reshape(-1, 1)
# Print shapes for debugging
print("Shape of U_approx:", U_approx.shape)
print("Expected shape:", (m * n, 1))
# Ensure the shape matches for reshaping
if U_approx.shape[0] == m * n and U_approx.shape[1] == 1:
    image_approx = U_approx.reshape(m, n).astype(np.uint8)
else:
    raise ValueError(f"Cannot reshape array of size {U_approx.size} into shape ({m}, {n})")
# Display the original altered image and the reconstructed image
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.imshow(image_array, cmap='gray')
plt.title('Original Altered Image')
plt.axis('off') # Hide axis
plt.subplot(1, 2, 2)
plt.imshow(image_approx, cmap='gray')
plt.title('Reconstructed Image')
plt.axis('off') # Hide axis
plt.show()
```

```
Subtask 7

Shape of U_approx: (10304, 1)
Expected shape: (10304, 1)
```



```
print("\nSubtask 8\n") #
Define image dimensions
m, n = 112, 92
# Read the altered image altered_image_path
="/content/database/person31.pgm" image_read =
Image.open(altered_image_path) image_array =
np.array(image_read)
U = image_array.reshape(m * n, 1)
# Compute the norms of the eigenvectors
Products = EigenVectors.T @ EigenVectors
NormsEigenVectors = np.diag(Products) #
Compute the projection coefficients
W = EigenVectors.T @ (U.astype(np.float64) - mean_face.reshape(-1, 1))
W = W / NormsEigenVectors.reshape(-1, 1) # Ensure proper division
# Reconstruct the image from the projection
U_approx = EigenVectors @ W + mean_face.reshape(-1, 1)
# Print shapes for debugging print("Shape of U_approx:",
U_approx.shape) print("Expected shape:", (m * n, 1)) #
Ensure the shape matches for reshaping if U_approx.shape[0]
== m * n and U_approx.shape[1] == 1:     image_approx =
U_approx.reshape(m, n).astype(np.uint8) else:
    raise ValueError(f"Cannot reshape array of size {U_approx.size} into shape ({m}, {n})")
# Display the original altered image and the reconstructed image
plt.figure(figsize=(10, 5)) plt.subplot(1, 2, 1)
plt.imshow(image_array, cmap='gray') plt.title('Original Altered
Image') plt.axis('off') # Hide axis plt.subplot(1, 2, 2)
plt.imshow(image_approx, cmap='gray') plt.title('Reconstructed
Image') plt.axis('off') # Hide axis plt.show()
```

```
Subtask 8

Shape of U_approx: (10304, 1)
Expected shape: (10304, 1)
```



```
print("\nSubtask 9\n") # # Recognition and approximation of a new face (person31.pgm) import numpy as np from PIL import Image import matplotlib.pyplot as plt
# Define image dimensions
```

```
m, n = 112, 92
# Read the new image (person31.pgm) new_image_path
="/content/database/person31.pgm" image_read =
Image.open(new_image_path) image_array =
np.array(image_read)
U = image_array.reshape(m * n, 1)
# Compute the norms of the eigenvectors
Products = EigenVectors.T @ EigenVectors
NormsEigenVectors = np.diag(Products) #
Compute the projection coefficients
W = EigenVectors.T @ (U.astype(np.float64) - mean_face.reshape(-1, 1))
W = W / NormsEigenVectors.reshape(-1, 1) # Ensure proper division
# Reconstruct the image from the projection
U_approx = EigenVectors @ W + mean_face.reshape(-1, 1)
114 # Print shapes for debugging print("Shape of U_approx:",
U_approx.shape) print("Expected shape:", (m * n, 1)) #
Ensure the shape matches for reshaping if U_approx.shape[0]
== m * n and U_approx.shape[1] == 1:     image_approx =
U_approx.reshape(m, n).astype(np.uint8) else:
    raise ValueError(f"Cannot reshape array of size {U_approx.size} into shape ({m}, {n})")
# Display the original new image and the reconstructed image
plt.figure(figsize=(10, 5)) plt.subplot(1, 2, 1)
plt.imshow(image_array, cmap='gray') plt.title('Original New
Image') plt.axis('off') # Hide axis plt.subplot(1, 2, 2)
plt.imshow(image_approx, cmap='gray') plt.title('Reconstructed
Image') plt.axis('off') # Hide axis plt.show()
# Variables: image_read, U, NormsEigenVectors, W, U_approx
```

```
Subtask 9

Shape of U_approx: (10304, 1)
Expected shape: (10304, 1)
```

Original New Image

Reconstructed Image

```
# Project 12: Matrix eigenvalues and the Google's PageRank algorithm import numpy as np import matplotlib.pyplot as plt import scipy.io import networkx as nx from scipy.sparse import csr_matrix from matplotlib.pylab import eig
```

```
print("\nSubtask 1\n")
# Load the network data
# Load the adjacency matrix from the file 'AdjMatrix.mat' data
= scipy.io.loadmat("/content/AdjMatrix.mat")



AdjMatrix =csr_matrix(data['AdjMatrix']) # Check the
sparsity of the matrix num_elements = AdjMatrix.shape[0]
* AdjMatrix.shape[1] num_non_zero_elements =
AdjMatrix.nnz nnzAdjMatrix = num_non_zero_elements /
num_elements print(f"Sparsity of AdjMatrix:
{nnzAdjMatrix:.4f}")
```

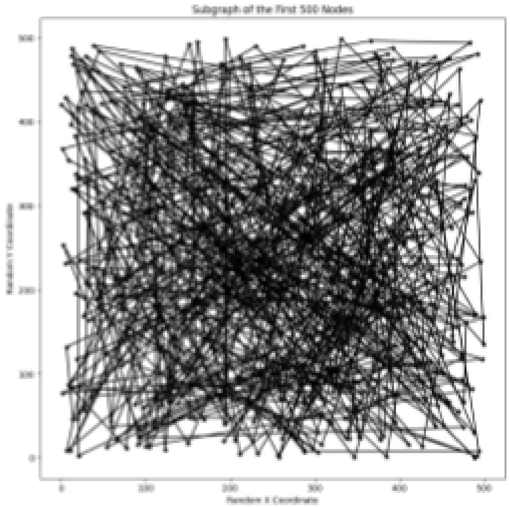
```
Subtask 1

Sparsity of AdjMatrix: 0.0015
```

```
print("\nSubtask 2\n") # Check the dimensions of the matrix m, n = AdjMatrix.shape print(f"Dimensions of AdjMatrix: {m} x {n}")
```

```
Subtask 2

Dimensions of AdjMatrix: 8297 x 8297
```

```
print("\nSubtask 3\n")
# Create a smaller submatrix and plot the network
NumNetwork = 500
AdjMatrixSmall = AdjMatrix[:NumNetwork, :NumNetwork].toarray() # Extract submatrix
# Generate random coordinates for the nodes #np.random.seed(0) # For
reproducibility coordinates = np.random.rand(NumNetwork, 2) * NumNetwork #
Random coordinates
# Plot the graph plt.figure(figsize=(10, 10))
plt.plot(coordinates[:, 0], coordinates[:, 1], 'k-*')
plt.title('Subgraph of the First 500 Nodes')
plt.xlabel('Random X Coordinate') plt.ylabel('Random Y
Coordinate') plt.show() # Variables
print(f"AdjMatrixSmall shape: {AdjMatrixSmall.shape}")
print(f"Coordinates shape: {coordinates.shape}")
print(f"NumNetwork: {NumNetwork}")
```

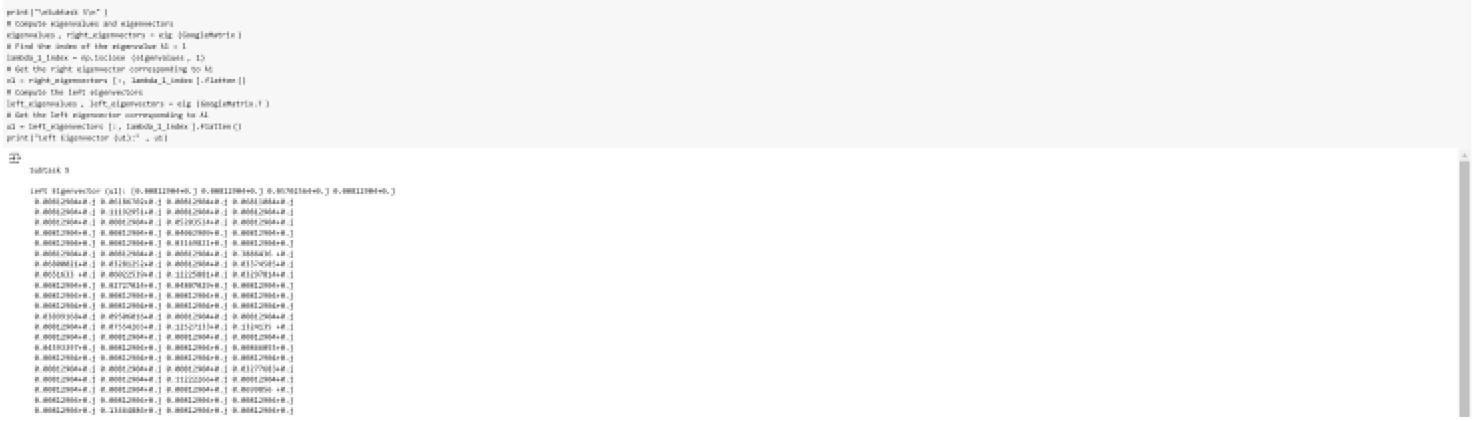
```
Subtask 3
```



```
AdjMatrixSmall shape: (500, 500)
Coordinates shape: (500, 2)

NumNetwork: 500
```

```
print("\nSubtask 4\n") #
Compute the Google Matrix
alpha = 0.15
GoogleMatrix = np.zeros((NumNetwork, NumNetwork))
# Check the amount of links originating from each webpage
NumLinks = np.sum(AdjMatrixSmall, axis=1) for i in
range(NumNetwork):     if NumLinks[i] != 0:
        GoogleMatrix[i, :] = AdjMatrixSmall[i, :] / NumLinks[i]
else:
        GoogleMatrix[i, :] = 1.0 / NumNetwork
GoogleMatrix = (1 - alpha) * GoogleMatrix + alpha * np.ones((NumNetwork,
NumNetwork)) / NumNetwork
# Compute the vectors w0, w1, w2, w3, w5, w10 w0
= np.ones(NumNetwork) / np.sqrt(NumNetwork) w1 =
w0 @ GoogleMatrix w2 = w1 @ GoogleMatrix w3 = w2
@ GoogleMatrix w10 = w0 @ (GoogleMatrix ** 10) w5
= w0 @ (GoogleMatrix ** 5) deltaw = w10 - w5
print("Difference δw:", np.linalg.norm(deltaw))
```

```
Subtask 4

Difference δw: 0.2791[illegible]
```



```
print("\nSubtask 5\n")
# Compute eigenvalues and eigenvectors
eigenvalues , right_eigenvectors = eig (GoogleMatrix )
# Find the index of the eigenvalue λ1 = 1
lambda_1_index = np.isclose (eigenvalues , 1)
# Get the right eigenvector corresponding to λ1
u1 = right_eigenvectors [:, lambda_1_index ].flatten ()
# Compute the left eigenvectors
left_eigenvalues , left_eigenvectors = eig (GoogleMatrix.T )
# Get the left eigenvector corresponding to λ1
u1 = left_eigenvectors [:, lambda_1_index ].flatten()
print("Left Eigenvector (u1):" , u1)
```

```
Subtask 5

Left Eigenvector (u1): [illegible]
```

```
print("\nSubtask 6\n")
# Normalize u1 to have all positive components
u1 = np.abs(u1) / np.linalg.norm(u1, 1)
```

```
Subtask 6
```

```
print("\nSubtask 7\n")
MaxRank, PageMaxRank = np.max(u1), np.argmax(u1)
print(f"MaxRank: {MaxRank}, PageMaxRank: {PageMaxRank}")
```

```
Subtask 7

MaxRank: [illegible], PageMaxRank: 27
```

```
print("\nSubtask 8\n")
MostLinks = np.sum(AdjMatrixSmall, axis=0) # Sum of columns
MaxLinks, PageMaxLinks = np.max(MostLinks), np.argmax(MostLinks)
print(f"MostLinks: {MostLinks}, MaxLinks: {MaxLinks}, PageMaxLinks: {PageMaxLinks}")
```

```
Subtask 8


MostLinks: [  0.   0.  31.   0.   0.  20.   0.  40.   0.  15.   0.   0.   0.   0.
  35.   0.   0.   0.  22.   0.   0.   0.  22.   0.   0.   0.   0. 122.  35.  23.   0.  14.  34.  20.  43.  24.   0.  14.  23.   0.   0.   0.   0.   0.   0.   0.   0.   0.  19.  32.   0.   0.   0.  40.  56.  65.   0.   0.   0.   0.  20.   0.   0.   1.   0.   0.   0.   0.   0.   0.   0.
  16.   0.   0.  64.   0.   0.   0.   0.  43.   0.   0.   0.   0.   0.  55.   0.   0.  16.   0.   0.   0.  17.  27.  31.   0.   0.   0.   0.   0.   0.   0.   0.   0.  21.   0.   0.  11.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.  19.   0.   0.
  20.   0.  28.   0.   0.  35.   0.   0.   0.   0.  22.   0.   0.   0.  29.   0.   0.  15.   0.   0.  20.  32.  44.  18.  18.   1.   0.   0.   0.   0.   0.   0.   0.  14.   0.   0.   0.  12.  24.   0.   0.  10.   0.   0.   0.  16.   0.   0.  35.   0.   0.   0.  31.  27.   0.   0.   0.   0.
   0.   0.   0.   0.  27.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.  28.   0.   0.   0.   0.   0.   0.   0.   0.   0.  94.   0.  21.   0.   0.  18.   0.   0.   0.   0.  16.   0.  21.  10.  36.   0.   7.   0.  22.   0.   0.   0.  20.   0.   0.   0.   0.   0.   0.  52.   0.   0.
  17.   0.   0.   0.  22.   0.  38.   0.   0.  21.   0.  25.  13.  32.   0.  24.   0.   0.   0.   0.   0.   0.   0.   0.   0. 122.   0.  13.  29.   0.   0.   0.   0.   0.  19.   0.  70.   0.   0.   8.  19.   0.   0.   0.  20.   0.   0.   0.   0.   0.   0.   0.  15. 108.  22.  13.  24.   0.
  25.   0.  18.   0.  29.   0.  34.  15.   0.  26.   1.   0.   0.  26.   0.  15.   0.  21.  41.   0.   7.  39.   0.   0.   0.   0.   0.   0.   9.   0.  13.   0.   0.   0.   7.  20.   0.  13.   0.   0.   0.   0.   3.   0.  35.  15.   8.   0.   0.   0.   0.   0.  43.   0.  12.   7.   0.   0.
  11.  18.   0.   0.   0.   0.  15.   0.  45.  21.  11.  20.   0.   0.   0.  28.  17.  21.   0.   0.   0.   0.   0.   0.   0.   0.   0.  10.  13.  33.   5.   0.  19.   0.  24.   0.   0.   0.   0.   0.  50.   6.   0.  21.  14.  45.   0.   0.   0.   0.   0.   0.   0.  36.   0.  15.   0.   3.
   0.  10.  16.   0.   0.   4.   0.  22.  22.  15.   0.  21.  24.  12.   0.  18.   0.   0.   0.  43.  15.   0.   0.   0.   0.   0.   0.  18.   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.
  23.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.  18.   0.   0.
   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.], MaxLinks: 122.0, PageMaxLinks: 27
```

```
print("\nSubtask 9\n") are_equal = PageMaxRank == PageMaxLinks print(f"Is the highest ranking webpage the same as the page with the most hyperlinks?{are_equal}")
# Q1: What is the number of hyperlinks pointing to the webpage MaxRank? print(f"Number of hyperlinks pointing to the webpage MaxRank:{MostLinks[PageMaxRank]}")
```

```
Subtask 9

Is the highest ranking webpage the same as the page with the most hyperlinks?True
Number of hyperlinks pointing to the webpage MaxRank:122.0
```

```
# Project 13: Social networks, clustering, and eigenvalue problems import numpy as np import matplotlib.pyplot as plt import scipy.io from scipy.linalg import eigh from scipy.io import loadmat
```

```
print("\nSubtask 1\n")
# 1. Simple graph: Define adjacency matrix
AdjMatrix = np.array([[0, 1, 1, 0],
[1, 0, 0, 1],
[1, 0, 0, 1], [0, 1, 1,
0]]) print("Adjacency
Matrix:") print(AdjMatrix)
```

```
Subtask 1

Adjacency Matrix:
[[0 1 1 0]
 [1 0 0 1]
 [1 0 0 1]
 [0 1 1 0]]
```

```
print("\nSubtask 2\n") # 2. Find the row sums of the matrix AdjMatrix RowSums = np.sum(AdjMatrix, axis=1) print("\nRow Sums:") print(RowSums)
```

```
Subtask 2


Row Sums:
[2 2 2 2]
```

```
print("\nSubtask 3\n") # 3. Compute the Laplacian of the graph LaplaceGraph = np.diag(RowSums) - AdjMatrix print("\nLaplacian Matrix:") print(LaplaceGraph) # Check if LaplaceGraph is singular test_vector = np.ones(len(LaplaceGraph)) singularity_check = LaplaceGraph @
 test_vector print("\nSingularity Check (Laplacian * ones):") print(singularity_check)
```

```
Subtask 3


Laplacian Matrix:
[[ 2 -1 -1  0]
 [-1  2  0 -1]
 [-1  0  2 -1]
 [ 0 -1 -1  2]]

Singularity Check (Laplacian * ones):
[0. 0. 0. 0.]
```

```
print("\nSubtask 4\n")
# 4. Find eigenvalues and eigenvectors using the eig function
D, V = np.linalg.eig(LaplaceGraph)
```

```
Subtask 4
```

```
print("\nSubtask 5\n")
d, ind = np.argsort(D), np.argsort(D)
D = np.diag(D[ind]) V = V[:, ind]
print("\nEigenvalues (sorted):")
print(np.diag(D))
print("\nEigenvectors (sorted):")
print(V)
```

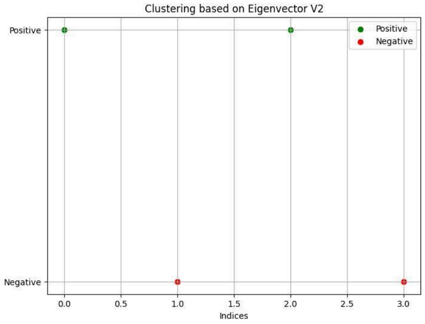
```
Subtask 5


Eigenvalues (sorted):
[-2.22044605e-16  2.00000000e+00  2.00000000e+00  4.00000000e+00]

Eigenvectors (sorted):
[[ 5.00000000e-01  4.08248290e-01  7.07106781e-01 -5.00000000e-01]
 [ 5.00000000e-01 -5.77350269e-01  4.80181756e-16  5.00000000e-01]
 [ 5.00000000e-01  5.77350269e-01 -1.77321568e-16  5.00000000e-01]
 [ 5.00000000e-01 -4.08248290e-01 -7.07106781e-01 -5.00000000e-01]]
```

```
print("\nSubtask 6\n")
# 6. Identify the second smallest eigenvalue and its corresponding eigenvector second_smallest_eigenvalue =
D[1, 1]
V2 = V[:, 1]
# Ensure V2 has a positive first entry if V2[0] < 0:     V2 = -V2
print("\nSecond Smallest Eigenvalue:") print(second_smallest_eigenvalue)
print("\nEigenvector corresponding to the second smallest eigenvalue (V2):")
print(V2)
```

```
Subtask 6


Second Smallest Eigenvalue:
1.9999999999999991

Eigenvector corresponding to the second smallest eigenvalue (V2):
[ 0.40824829 -0.57735027  0.57735027 -0.40824829]
```

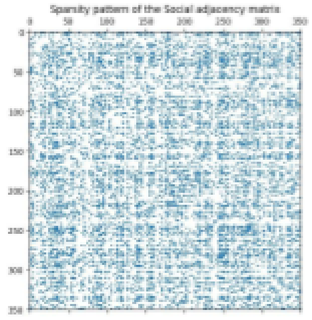
```
print("\nSubtask 7\n")
# 7. Separate the elements of the eigenvector V2
pos = [] neg = [] for j in range(len(V2)):     if
V2[j] > 0:
        pos.append(j)
else:
        neg.append(j) print("\nPositive
Indices (V2 > 0):") print(pos)
print("\nNegative Indices (V2 <= 0):")
print(neg)
# Optional: Visualize the clusters plt.figure(figsize=(8, 6))
plt.scatter(pos, [1]*len(pos), color='green', label='Positive')
plt.scatter(neg, [0]*len(neg), color='red', label='Negative')
plt.yticks([0, 1], ['Negative', 'Positive'])
plt.title('Clustering based on Eigenvector V2')
plt.xlabel('Indices') plt.legend() plt.grid() plt.show()
```

```
Subtask 7


Positive Indices (V2 > 0):
[0, 2]

Negative Indices (V2 <= 0):
[1, 3]
```



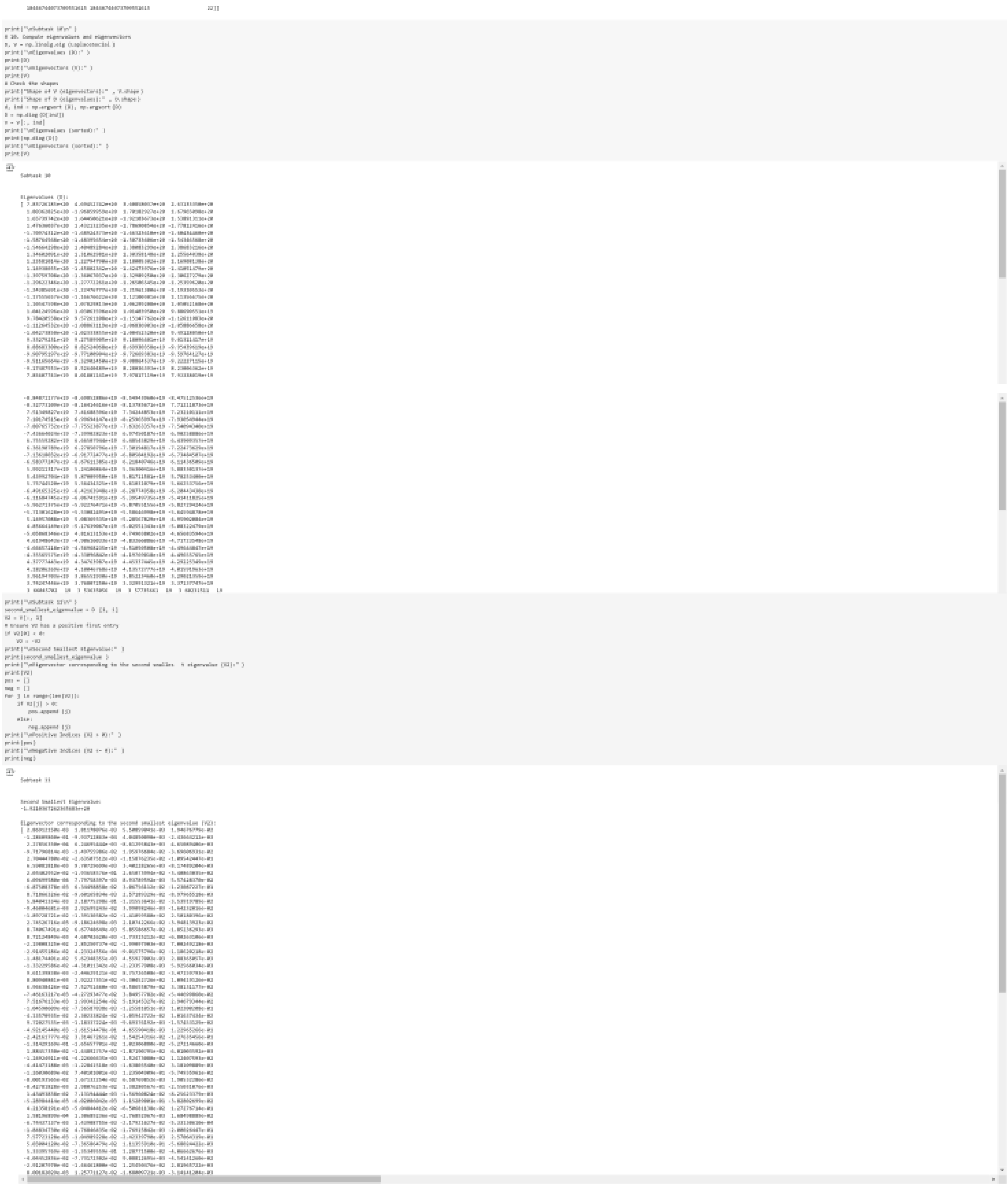
```
print("\nSubtask 8\n")
# 8. Load the data
data = loadmat("/content/social.mat") Social = data['Social']
print("Loaded Social adjacency matrix with shape:", Social.shape)
# Spy plot of the Social matrix plt.figure(figsize=(8, 6))
plt.spy(Social, markersize=1) plt.title('Sparsity pattern of
the Social adjacency matrix') plt.show()
```

```
Subtask 8

Loaded Social adjacency matrix with shape: (351, 351)
```



```
print("\nSubtask 9\n") # 9. Define DiagSocial and LaplaceSocial
DiagSocial = np.sum(Social, axis=1) LaplaceSocial = np.diag(DiagSocial) - Social print("\nDiagonal matrix DiagSocial:") print(DiagSocial) print("\nLaplacian matrix LaplaceSocial:") print(LaplaceSocial)
```

```
Subtask 9


Diagonal matrix DiagSocial:
[42 19 25 36 50  5  7  2 29  5 28  5  4  8 38 17 47  4 29 49 28 17  7 23
  8 43 25 48 30 20 43 47  6 39 34 46 40 50 27 32 15 46 33 31 24 42 10  8
  6 44 26  5 24 47  6 24 50  7 56 11  4  4 15 24  9 21  7 22 23  5 40 36
  8  5  9  4  4 45  4 23  5 35 46  9  8  7 56 34 49  9 48 42  6 44 33 34
 29 24 50  4  8  4 21 43 28  6 21 26  5  8 23  3 27 51 42 48 18  7 31 50
 46 45 25  4  4  7 22 52 41  4  5 24 16  8 20  3 34  4 46 46 43 27 43 32
  4 42 42  4  5  4 26  7 20  7 51 26 32 48 42 55 48  6  6  7 28  6  7  4
 29 33 19 39  4  6  6 47 32 45 45 22 30 39 19  5 27 44 30 34 43  5  5 48
 19 26 35  7 50 31  6 31  6 34 51  7  6 29 32  5  4 18 29  4  5 38 32 11
 43 24 46 54 34 48  6  5  5 18 49 14 47 22 28 46 46 22  6  7 45  5  5  6
  6 16 23  6 41 43 49 50 40 48 41 44 23 31 27 40 46 52  5  5  5 27 25 44
 47  6  4 36 16 43  8  5  7 44 55 41  3 29 50 34 39 25 44 50  8 47 19  6
 21 46 40 34 22  6 48 28 41 29 18  3  5  6 52 44  8  6  5 18  6 42 44 45
  8  6 25 33 51 20  5 32 41  5 20  5 49 45 15 17 23  4  5  9  7 50 36 51
 46  6 47 33  6 30  5  5 48  9  8 53 21 19 22]

Laplacian matrix LaplaceSocial:
[[                  42                    0                    0 ...
                     0                    0                    0]  [                    0                   19                    0 ...
  18446744073709551615 18446744073709551615                    0]  [                    0                    0                   25 ...
                     0                    0                    0]  ...
 [                    0 18446744073709551615                    0 ...
                    21                    0 18446744073709551615]  [                    0 18446744073709551615                    0 ...
                     0                   19 18446744073709551615]  [                    0                    0                    0 ...
```



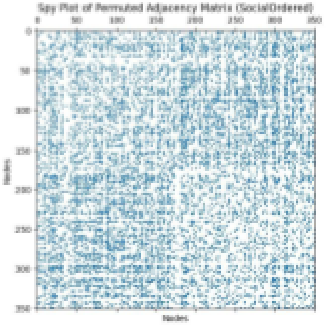
```
print("\nSubtask 12\n")
# Create the order based on positive and negative indices order
= pos + neg # Combine the positive and negative indices m, n =
Social.shape # Get the shape of the Social matrix iden =
np.eye(m) # Identity matrix of size m
# Create the permutation matrix P
P = np.zeros((m, m)) for j in
range(m):     for k in range(m):
        P[j, k] = iden[order[j], k]
# Permute the adjacency matrix
SocialOrdered = P @ Social @ P.T # Using matrix multiplication print("Shape
of SocialOrdered:", SocialOrdered.shape)
```

```
Subtask 12
```

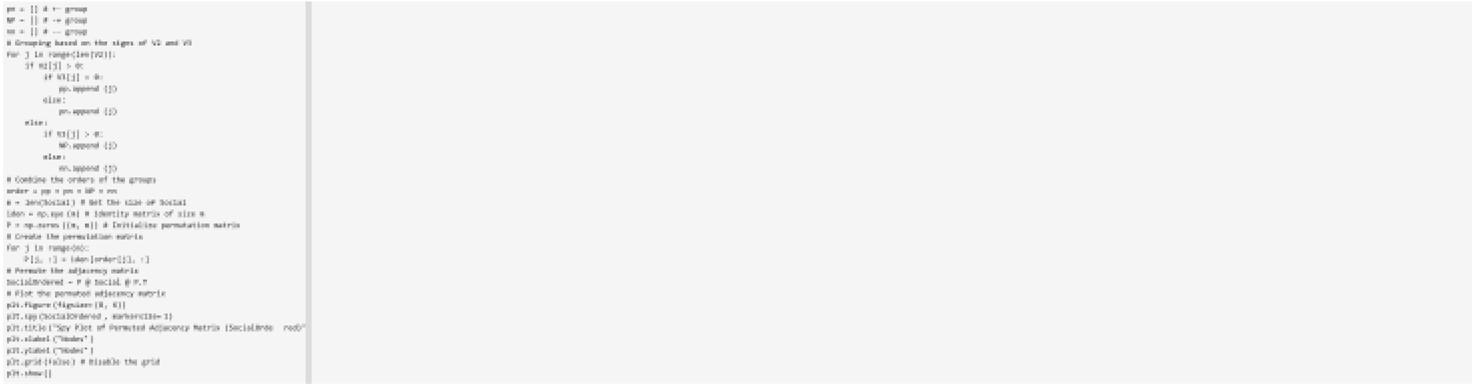
```
Shape of SocialOrdered: (351, 351)
```

```
print("\nSubtask 13\n") # Plot the permuted adjacency matrix plt.figure(figsize=(8, 6)) plt.spy(SocialOrdered, markersize=1) # Using a smaller marker size for better visibility plt.title("Spy Plot of Permuted Adjacency Matrix (SocialOrdered)") plt.xlabel("Nodes") plt.ylabel("Nodes") plt.grid(False) # Disable the grid
plt.show()
```

```
Subtask 13
```

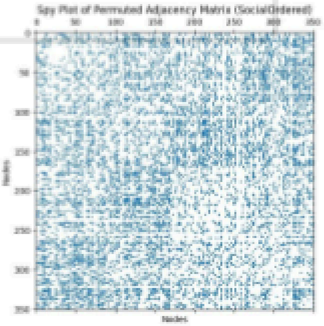


```
print("\nSubtask 14\n")
# Explore the third smallest eigenvalue for clustering V3
= V[:, 2] # Get the third eigenvector
if V3[0] < 0: # Ensure V3 has a positive first entry
    V3 = -V3 # Initialize lists
for the groups pp = [] # ++ group
```



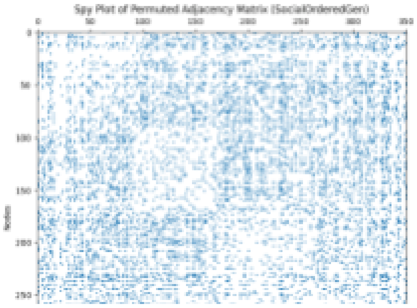
```
pm = [] # +- group
MP = [] # -+ group
mm = [] # -- group
# Grouping based on the signs of V2 and V3
for j in range(len(V2)):
    if V2[j] > 0:
        if V3[j] > 0:
            pp.append (j)
        else:
            pm.append (j)
    else:
        if V3[j] > 0:
            MP.append (j)
        else:
            mm.append (j)
# Combine the orders of the groups
order = pp + pm + MP + mm
n = len(Social) # Get the size of Social
Iden = np.eye (n) # Identity matrix of size n
P = np.zeros ((n, n)) # Initialize permutation matrix
# Create the permutation matrix
for j in range(n):
    P[j, :] = Iden[order[j], :]
# Permute the adjacency matrix
SocialOrdered = P @ Social @ P.T
# Plot the permuted adjacency matrix
plt.figure(figsize=(8, 6))
plt.spy(SocialOrdered , markersize= 1)
plt.title("Spy Plot of Permuted Adjacency Matrix (SocialOrde  red)"
plt.xlabel("Nodes")
plt.ylabel("Nodes")
plt.grid(False) # Disable the grid
plt.show()
```

```
Subtask 14
```



```
print("\nSubtask 15\n")
# Step 15: Fiedler vector procedure iteratively for clusters import
numpy as np
import matplotlib.pyplot as plt
# Assuming 'Social' is your adjacency matrix, and 'pos' and 'neg' are your positive and negative indices
# Define SocialPos and SocialNeg based on the positive and negative indices
SocialPos = Social[np.ix_(pos, pos)]
SocialNeg = Social[np.ix_(neg, neg)]
# Calculate the Laplacian for the positive group rowsumpos
= np.sum(SocialPos, axis=1)
DiagSocialPos = np.diag(rowsumpos)
LaplaceSocialPos = DiagSocialPos - SocialPos
# Eigen decomposition for positive group DPos
, VPos = np.linalg.eig(LaplaceSocialPos) d,
ind = np.argsort(DPos), np.argsort(DPos)
DPos = np.diag(DPos[ind])
VPos = VPos[:, ind]
V2Pos = VPos[:, 1] # Second smallest eigenvector for positive group
# Group positive nodes posp
= [] # Positive group posn =
[] # Negative group for j in
range(len(V2Pos)):     if
V2Pos[j] > 0:
        posp.append(pos[j]) # Append original index
else:
        posn.append(pos[j]) # Append original index #
Calculate the Laplacian for the negative group
rowsumneg = np.sum(SocialNeg, axis=1)
DiagSocialNeg = np.diag(rowsumneg)
LaplaceSocialNeg = DiagSocialNeg - SocialNeg
# Eigen decomposition for negative group DNeg
, VNeg = np.linalg.eig(LaplaceSocialNeg) d,
ind = np.argsort(DNeg), np.argsort(DNeg)
DNeg = np.diag(DNeg[ind])
VNeg = VNeg[:, ind]
V2Neg = VNeg[:, 1] # Second smallest eigenvector for negative group
# Group negative nodes negp
= [] # Positive group negn =
[] # Negative group for j in
range(len(V2Neg)):     if
V2Neg[j] > 0:
        negp.append(neg[j]) # Append original index
else:
        negn.append(neg[j]) # Append original index #
Generate the final order for the permutation ordergen
= posp + posn + negp + negn # Create the permutation
matrix m = len(Social) # Assuming the size of Social
iden = np.eye(m) # Identity matrix of size m
P = np.zeros((m, m)) # Initialize permutation matrix
# Create the permutation matrix for
j in range(m):
    P[j, :] = iden[ordergen[j], :] # Filling the permutation matrix based on ordergen #
Permute the adjacency matrix
SocialOrderedGen = P @ Social @ P.T # Permutation of the Social matrix
# Plot the permuted adjacency matrix
plt.figure(figsize=(10, 8)) plt.spy(SocialOrderedGen,
markersize=1)
plt.title("Spy Plot of Permuted Adjacency Matrix (SocialOrderedGen)")
plt.xlabel("Nodes") plt.ylabel("Nodes") plt.grid(False) # Disable grid
for clarity plt.show()
```

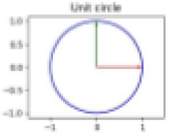
```
Subtask 15
```



```
# Project 14: Singular Value Decomposition and image compression import numpy as np import matplotlib.pyplot as plt
```

```
print("\nSubtask 1\n")
# Task 1: Plotting the unit circle and basis vectors t = np.linspace(0, 2 * np.pi, 100) X = np.array([np.cos(t), np.sin(t)]) plt.subplot(2, 2, 1) plt.plot(X[0, :], X[1, :], 'b') plt.quiver(0, 0, 1, 0, color='r', angles='xy', scale_units='xy', scale=1) plt.quiver(0, 0, 0, 1, color='g', angles='xy',
 scale_units='xy', scale=1) plt.axis('equal') plt.title('Unit circle')
plt.show()
```

```
Subtask 1
```

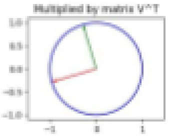


```
print("\nSubtask 2\n") A = np.array([[2, 1], [-1, 1]]) U, S, V = np.linalg.svd(A) print("U:\n", U) print("S:\n", S) print("V:\n", V) # Verify orthogonality print("U' * U:\n", np.dot(U.T, U)) print("V' * V:\n", np.dot(V.T, V))
```

```
Subtask 2

U:
 [[-0.95709203  0.28978415]
 [ 0.28978415  0.95709203]] S:
 [2.30277564 1.30277564] V:
 [[-0.95709203 -0.28978415]
 [-0.28978415  0.95709203]] U' * U:
 [[1.00000000e+00 1.77671996e-17]
 [1.77671996e-17 1.00000000e+00]] V' * V:
 [[ 1.00000000e+00 -2.42191841e-17]
 [-2.42191841e-17  1.00000000e+00]]
```

```
print("\nSubtask 3\n") VX = np.dot(V.T, X) plt.subplot(2, 2, 2) plt.plot(VX[0, :], VX[1,
:], 'b') plt.quiver(0, 0, V[0, 0], V[0, 1], color='r', angles='xy', scale_units='xy',
scale=1) plt.quiver(0, 0, V[1, 0], V[1, 1], color='g', angles='xy', scale_units='xy',
scale=1) plt.axis('equal')
plt.title('Multiplied by matrix V^T') plt.show()
```

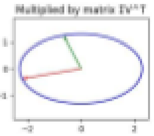
```
Subtask 3
```



```
print("\nSubtask 4\n") S_matrix = np.diag(S) SVX = np.dot(S_matrix, VX) plt.subplot(2, 2, 3) plt.plot(SVX[0, :], SVX[1, :], 'b') plt.quiver(0, 0, S[0] * V[0, 0], S[1] * V[0, 1], color='r', angles='xy', scale_units='xy', scale=1) plt.quiver(0, 0, S[0] * V[1, 0], S[1] * V[1, 1], color='g', angles='xy', scale_units='xy',
 scale=1) plt.axis('equal') plt.title('Multiplied by matrix ΣV^T')
```

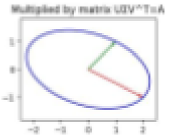
```
plt.show()
```

```
Subtask 4
```



```
print("\nSubtask 5\n") AX =
np.dot(U, SVX) plt.subplot(2, 2,
4) plt.plot(AX[0, :], AX[1, :],
'b')
plt.quiver(0, 0, U[0, 0] * S[0] * V[0, 0] + U[0, 1] * S[1] * V[0, 1], U[1, 0] *
S[0] * V[0, 0] + U[1, 1] * S[1] * V[0, 1], color='r', angles='xy',
scale_units='xy', scale=1)
plt.quiver(0, 0, U[0, 0] * S[0] * V[1, 0] + U[0, 1] * S[1] * V[1, 1], U[1, 0] *
S[0] * V[1, 0] + U[1, 1] * S[1] * V[1, 1], color='g', angles='xy',
scale_units='xy', scale=1) plt.axis('equal') plt.title('Multiplied by matrix
UΣV^T=A') plt.show()
```

```
Subtask 5
```



```
print("\nSubtask 6\n")
# Modification example for U and V (this is just a random example, modifications need to be chosen carefully)
U1 = U V1 = V.T print("U1 * S * V1.T:\n", np.dot(U1,
np.dot(S_matrix, V1.T)))
```

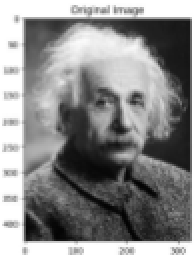
```
Subtask 6

U1 * S * V1.T:
 [[ 2.  1.]
 [-1.  1.]]
```

```
print("\nSubtask 7\n") Av1 =
np.dot(A, V.T[:, 0]) Av2 = np.dot(A,
V.T[:, 1]) print("Av1:\n", Av1)
print("σ1 * u1:\n", S[0] * U[:, 0])
print("Av2:\n", Av2) print("σ2 *
u2:\n", S[1] * U[:, 1])
# Numerical check print("A * V - U * S:\n", np.dot(A, V.T) -
np.dot(U, S_matrix))
```

```
Subtask 7
Av1:
 [-2.2039682   0.66730788] σ1 * u1:  [-2.2039682   0.66730788] Av2:
 [0.37752373 1.24687618] σ2 * u2:  [0.37752373 1.24687618] A * V - U * S:
 [[-8.88178420e-16  2.77555756e-16]
 [ 3.33066907e-16  2.22044605e-16]]
```

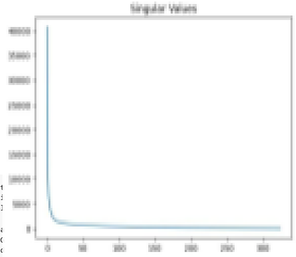
```
print("\nSubtask 8-11\n") import
cv2
# Load the image
ImJPG = cv2.imread("/content/einstein.jpg",cv2.IMREAD_GRAYSCALE)
plt.figure() plt.imshow(ImJPG, cmap='gray') plt.title('Original
Image') plt.show() # Singular Value Decomposition
UIm, SIm, VIm = np.linalg.svd(ImJPG.astype(np.float64), full_matrices=False) print(UIm)
# Plot Singular Values plt.figure()
plt.plot(np.arange(len(SIm)), SIm)
plt.title('Singular Values') plt.show()
# Image compression using truncated SVD
for k in [50, 100, 150]:
    ImJPG_comp = np.dot(UIm[:, :k], np.dot(np.diag(SIm[:k]), VIm[:k, :]))     plt.figure()
plt.imshow(ImJPG_comp, cmap='gray')     plt.title(f'Compressed Image with {k} Singular Values')
plt.show()     pct = 1 - (np.size(UIm[:, :k]) + np.size(VIm[:k, :]) * np.size(np.diag(SIm[:k])))
/np.size(ImJPG)     print(f'Compression percentage for {k} singular values: {pct:.3f}')
```

```
Subtask 8-11
```

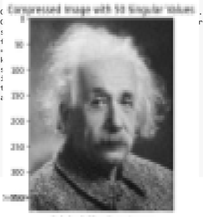


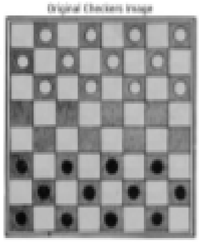
```
[[-0.01973233  0.00558642 -0.02123928 ... -0.00261544 -0.05072357
  -0.03953517]
 [-0.01978487  0.00571134 -0.02162803 ...  0.0073873  -0.03997558
  -0.03325746]
 [-0.01991589  0.00594026 -0.02212336 ... -0.00760312 -0.0560467
  -0.09563301]  ...
 [-0.01950539 -0.00595175 -0.04791957 ...  0.00209835  0.06150357
   0.04754879]  [-0.02000929 -0.00796643 -0.0507054  ... -0.012451   -0.0317809
```

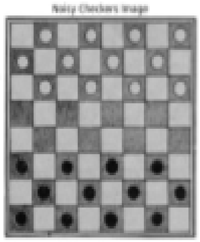
```
                                                              -0.05879769]
[-0.0581[illegible] -0.0587[illegible] -0.05[illegible] ... -0.0[illegible] -0.05[illegible]
  0.0[illegible]]
```

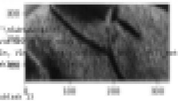


```
print
#12 i
matpl
cv2
# Loa
ImJPG
# Add
m, n
ImJPG                                 .random.rand(m, n) - 0.5)
ImJPG                                 re values are within valid range
# Dis
plt.f
cmap=
Check
plt.s
plt.i
plt.t
plt.a
```







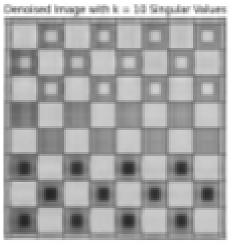


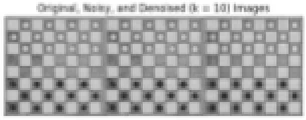
```
print("\n[illegible]
# Compre[illegible]
U[illegible], S[illegible], V[illegible]                                 matrices=False)
# Verif[illegible]
```

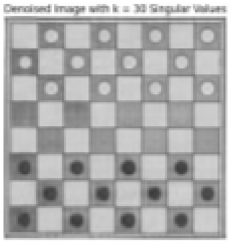
```
Subtask 3)
Compression percentage for 150 singular values: -7750.91%
```

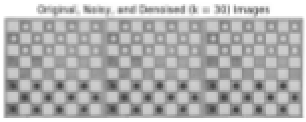
```
print("\nSubtask 14\n")
# Function to approximate the image with k singular values def
approximate_image(U, S, V, k):
    return np.dot(U[:, :k], np.dot(np.diag(S[:k]), V[:k, :])) #
Approximations with k = 10, k = 30, k = 50 singular values ks =
[10, 30, 50] for k in ks:
    ImJPG_approx = approximate_image(UIm, SIm, VIm, k)
plt.figure()     plt.imshow(ImJPG_approx, cmap='gray')
plt.title(f'Denoised Image with k = {k} Singular Values')
plt.axis('off')     plt.show()
    # Compare the images to the initial noisy image     plt.figure()
plt.imshow(np.hstack((ImJPG, ImJPG_Noisy, ImJPG_approx)), cmap='gray')
plt.title(f'Original, Noisy, and Denoised (k = {k}) Images')
plt.axis('off')     plt.show()
```

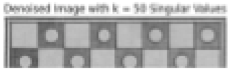
```
Subtask 14
```


Denoised Image with k = 10 Singular Values


Original, Noisy, and Denoised (k = 10) Images


Denoised Image with k = 30 Singular Values


Original, Noisy, and Denoised (k = 30) Images


Denoised Image with k = 50 Singular Values