HW2 - Page Rank

Computational linear algebra for large scale problems

Arash Daneshvar

s314415

Exercise 1

In [9]:

```
import numpy as np

A = np.array([
    [0  , 0  , 1, 1/2],
    [1/3, 0  , 0,   0],
    [1/3, 1/2, 0, 1/2],
    [1/3, 1/2, 0,   0]])

# Compute eigenvalues and eigenvectors
eigenvalues, eigenvectors = np.linalg.eig(A)

# Find the index of the eigenvalue equal to 1
index = np.where(np.isclose(eigenvalues, 1))[0][0]

# Get the corresponding eigenvector
eigenvector = eigenvectors[:, index]

# Normalize the eigenvector so that the sum of its elements is 1
eigenvector_normalized = eigenvector / np.sum(eigenvector)

# Extract the real part of the eigenvector
eigenvector_real = np.real(eigenvector_normalized)

print("Real part of the eigenvector:")
print(eigenvector_real)
```

```
Real part of the eigenvector:
[0.38709677 0.12903226 0.29032258 0.19354839]
```

After adding page 5:

In [2]:
```python
import numpy as np

A = np.array([
    [0  , 0  , 1/2, 1/2, 0],
    [1/3, 0  ,   0,   0, 0],
    [1/3, 1/2,   0, 1/2, 1],
    [1/3, 1/2,   0,   0, 0],
    [0  ,   0, 1/2,   0, 0]])

# Compute eigenvalues and eigenvectors
eigenvalues, eigenvectors = np.linalg.eig(A)

# Find the index of the eigenvalue equal to 1
index = np.where(np.isclose(eigenvalues, 1))[0][0]

# Get the corresponding eigenvector
eigenvector = eigenvectors[:, index]

# Normalize the eigenvector so that the sum of its elements is 1
eigenvector_normalized = eigenvector / np.sum(eigenvector)

# Extract the real part of the eigenvector
eigenvector_real = np.real(eigenvector_normalized)

print("Real part of the eigenvector:")
print(eigenvector_real)
```

```
Real part of the eigenvector:
[0.24489796 0.08163265 0.36734694 0.12244898 0.18367347]
```

Exercise 2

In [1]:
```python
import numpy as np

# Define the matrix A
A = np.array([
    [0  , 0  , 1, 1/2, 0, 0],
    [1/3, 0  , 0,   0, 0, 0],
    [1/3, 1/2, 0, 1/2, 0, 0],
    [1/3, 1/2, 0,   0, 0, 0],
    [0  , 0  , 0,   0, 0, 0],
    [0  , 0  , 0,   0, 0, 0]])

# Perform row reduction to find the rank of A
rank_A = np.linalg.matrix_rank(A)

# Number of components (subwebs)
num_components = 3

# Output the results
print(f"Matrix A:\n{A}")
print(f"Rank of A: {rank_A}")
print(f"Number of components (subwebs): {num_components}")
print(f"dim(V1(A)) {'equals' if rank_A == num_components else 'exceeds' i
```

```
Matrix A:
[[0.         0.         1.         0.5        0.         0.        ]
 [0.33333333 0.         0.         0.         0.         0.        ]
 [0.33333333 0.5        0.         0.5        0.         0.        ]
 [0.33333333 0.5        0.         0.         0.         0.        ]
 [0.         0.         0.         0.         0.         0.        ]
 [0.         0.         0.         0.         0.         0.        ]]
Rank of A: 4
Number of components (subwebs): 3
dim(V1(A)) exceeds the number of components in the web.
```

Exercise 3

In [23]:

```
import numpy as np

# Define a sample column-stochastic matrix A
A = np.array([
    [0, 1, 0, 0, 1/3],
    [1, 0, 0, 0, 0],
    [0, 0, 0, 1, 1/3],
    [0, 0, 1, 0, 1/3],
    [0, 0, 0, 0, 0]
])

# Verify that A is column-stochastic (each column sums to 1)
print(f"Column sums of A: {A.sum(axis=0)}")

# Compute the eigenvalues and eigenvectors of matrix A
eigenvalues, eigenvectors = np.linalg.eig(A)

# Print the eigenvalues and eigenvectors
print(f"Eigenvalues of A: {eigenvalues}")
print(f"Eigenvectors of A:\n{eigenvectors}")

# Identify the index of the eigenvalue that is (approximately) 1
eigenvalue_1_index = np.where(np.isclose(eigenvalues, 1))[0]

# Extract the eigenvectors corresponding to the eigenvalue 1
eigenvectors_1 = eigenvectors[:, eigenvalue_1_index]

# Output the eigenvectors corresponding to eigenvalue 1
print(f"Eigenvectors corresponding to eigenvalue 1:\n{eigenvectors_1}")

# Dimension of the eigenspace for eigenvalue 1
dimension_V1_A = eigenvectors_1.shape[1]
print(f"Dimension of V1(A): {dimension_V1_A}")
```

```
Column sums of A: [1. 1. 1. 1. 1.]
Eigenvalues of A: [ 1. -1.  1. -1.  0.]
Eigenvectors of A:
[[ 0.70710678 -0.70710678  0.          0.          0.        ]
 [ 0.70710678  0.70710678  0.          0.         -0.28867513]
 [ 0.         -0.          0.70710678 -0.70710678 -0.28867513]
 [ 0.         -0.          0.70710678  0.70710678 -0.28867513]
 [ 0.          0.          0.          0.          0.8660254 ]]
Eigenvectors corresponding to eigenvalue 1:
[[0.70710678 0.        ]
 [0.70710678 0.        ]
 [0.         0.70710678]
 [0.         0.70710678]
 [0.         0.        ]]
Dimension of V1(A): 2
```

Exercise 4

In [24]:

```
import numpy as np

# Define the substochastic matrix A'
A_prime = np.array([[0, 0, 0, 1/2],
                    [1/3, 0, 0, 0],
                    [1/3, 1/2, 0, 1/2],
                    [1/3, 1/2, 0, 0]])

# Compute the eigenvalues and eigenvectors of A'
eigenvalues, eigenvectors = np.linalg.eig(A_prime)

# Find the Perron (largest positive) eigenvalue and corresponding eigenve
perron_index = np.argmax(eigenvalues.real)
perron_eigenvalue = eigenvalues[perron_index].real
perron_eigenvector = eigenvectors[:, perron_index].real

# Ensure non-negativity
perron_eigenvector = np.abs(perron_eigenvector)

# Scale the Perron eigenvector so that its components sum to one
scaled_perron_eigenvector = perron_eigenvector / np.sum(perron_eigenvecto

# Output the results
print(f"Perron eigenvalue: {perron_eigenvalue}")
print(f"Non-negative Perron eigenvector:\n{perron_eigenvector}")
print(f"Scaled Perron eigenvector:\n{scaled_perron_eigenvector}")
```

```
Perron eigenvalue: 0.5613532393351085
Non-negative Perron eigenvector:
[0.37479335 0.22255348 0.79557628 0.42078293]
Scaled Perron eigenvector:
[0.20664504 0.12270648 0.43864676 0.23200172]
```

Exercise 8

In [29]:
```python
import numpy as np

# Define the first column-stochastic matrix A
A = np.array([
    [0.5, 0.3, 0.2],
    [0.5, 0.7, 0.8]
])

# Define the second column-stochastic matrix B
B = np.array([
    [0.4, 0.6],
    [0.4, 0.3],
    [0.2, 0.1]
])

# Compute the product of A and B
C = np.dot(A, B)

# Print the resulting matrix C
print("Matrix A:")
print(A)

print("\nMatrix B:")
print(B)

print("\nProduct Matrix C = AB:")
print(C)

# Verify that C is column-stochastic
# Check non-negativity
non_negative = np.all(C >= 0)
print(f"\nAll entries non-negative: {non_negative}")

# Check column sums
column_sums = np.sum(C, axis=0)
print(f"Column sums of C: {column_sums}")

is_column_stochastic = np.allclose(column_sums, 1)
print(f"C is column-stochastic: {is_column_stochastic}")
```

```
Matrix A:
[[0.5 0.3 0.2]
 [0.5 0.7 0.8]]

Matrix B:
[[0.4 0.6]
 [0.4 0.3]
 [0.2 0.1]]

Product Matrix C = AB:
[[0.36 0.41]
 [0.64 0.59]]

All entries non-negative: True
Column sums of C: [1. 1.]
C is column-stochastic: True
```

Exercise 11

In [32]:

```
import numpy as np

# Define column-stochastic matrix A
A = np.array([
    [0  , 0  , 1/2, 1/2, 0],
    [1/3, 0  ,   0,   0, 0],
    [1/3, 1/2,   0, 1/2, 1],
    [1/3, 1/2,   0,   0, 0],
    [0  ,   0, 1/2,   0, 0]])

# Define the teleportation matrix S
n = 5
S = np.ones((n, n)) / n

# Define the damping factor
m = 0.15

# Calculate the Google matrix M
M = m * S + (1 - m) * A

# Calculate the eigenvalues and eigenvectors of M
eigenvalues, eigenvectors = np.linalg.eig(M)

# Find the index of the eigenvalue equal to 1
index = np.where(np.isclose(eigenvalues, 1))[0][0]

# Get the corresponding eigenvector
eigenvector = eigenvectors[:, index]

# Normalize the eigenvector so that the sum of its elements is 1
eigenvector_normalized = eigenvector / np.sum(eigenvector)

# Extract the real part of the eigenvector
eigenvector_real = np.real(eigenvector_normalized)

print("Real part of the eigenvector:")
print(eigenvector_real)
```

```
Real part of the eigenvector:
[0.23714058 0.09718983 0.34889409 0.13849551 0.17827999]
```

Exercise 12

In [35]:
```python
import numpy as np

# Define column-stochastic matrix A
A = np.array([
    [0  , 0  , 1/2, 1/2, 0, 1/5],
    [1/3, 0  ,   0,   0, 0, 1/5],
    [1/3, 1/2,   0, 1/2, 1, 1/5],
    [1/3, 1/2,   0,   0, 0, 1/5],
    [0  ,   0, 1/2,   0, 0, 1/5],
    [0  ,   0,   0,   0, 0,   0]])

# Define the teleportation matrix S
n = 6
S = np.ones((n, n)) / n

# Define the damping factor
m = 0.15

# Calculate the Google matrix M
M = m * S + (1 - m) * A

# Calculate the eigenvalues and eigenvectors of M
eigenvalues, eigenvectors = np.linalg.eig(M)

# Find the index of the eigenvalue equal to 1
index = np.where(np.isclose(eigenvalues, 1))[0][0]

# Get the corresponding eigenvector
eigenvector = eigenvectors[:, index]

# Normalize the eigenvector so that the sum of its elements is 1
eigenvector_normalized = eigenvector / np.sum(eigenvector)

# Extract the real part of the eigenvector
eigenvector_real = np.real(eigenvector_normalized)

print("Real part of the eigenvector:")
print(eigenvector_real)
```

```
Real part of the eigenvector:
[0.23121207 0.09476009 0.34017174 0.13503312 0.17382299 0.025     ]
```

In [36]:
```python
# List of values
values = [0.23121207, 0.09476009, 0.34017174, 0.13503312, 0.17382299, 0.0

# List of page names
page_names = ['Page 1', 'Page 2', 'Page 3', 'Page 4', 'Page 5', 'Page 6']

# Create a dictionary mapping each value to a corresponding page name
page_ranking = {page_names[i]: values[i] for i in range(len(values))}

# Sort the dictionary by values in descending order
sorted_page_ranking = dict(sorted(page_ranking.items(), key=lambda item:

# Print the sorted dictionary
print(sorted_page_ranking)
```

```
{'Page 3': 0.34017174, 'Page 1': 0.23121207, 'Page 5': 0.17382299, 'Page
4': 0.13503312, 'Page 2': 0.09476009, 'Page 6': 0.025}
```

Exercise 13

In [6]:

```
import numpy as np

# Given matrix A
A = np.array([
    [0, 0, 1, 1/2, 0, 0],
    [1/3, 0, 0, 0, 0, 0],
    [1/3, 1/2, 0, 1/2, 0, 0],
    [1/3, 1/2, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 1],
    [0, 0, 0, 0, 1, 0]
])

# Define the teleportation matrix S
n = 6
S = np.ones((n, n)) / n

# Define the damping factor
m = 0.15

# Calculate the Google matrix M
M = m * S + (1 - m) * A

# Calculate the eigenvalues and eigenvectors of M
eigenvalues, eigenvectors = np.linalg.eig(M)

# Find the index of the eigenvalue equal to 1
index = np.where(np.isclose(eigenvalues, 1))[0][0]

# Get the corresponding eigenvector
eigenvector = eigenvectors[:, index]

# Normalize the eigenvector so that the sum of its elements is 1
eigenvector_normalized = eigenvector / np.sum(eigenvector)

# Extract the real part of the eigenvector
eigenvector_real = np.real(eigenvector_normalized)

print("Real part of the eigenvector:")
print(eigenvector_real)
```

```
Real part of the eigenvector:
[0.24543378 0.09453957 0.19197442 0.13471889 0.16666667 0.16666667]
```

In [8]:

```
# List of values
values = [0.24543378, 0.09453957, 0.19197442, 0.13471889, 0.16666667, 0.1

# List of page names
page_names = ['Page 1', 'Page 2', 'Page 3', 'Page 4', 'Page 5', 'Page 6']

# Create a dictionary mapping each value to a corresponding page name
page_ranking = {page_names[i]: values[i] for i in range(len(values))}

# Sort the dictionary by values in descending order
sorted_page_ranking = dict(sorted(page_ranking.items(), key=lambda item:

# Print the sorted dictionary
print(sorted_page_ranking)
```

```
{'Page 1': 0.24543378, 'Page 3': 0.19197442, 'Page 5': 0.16666667, 'Page
6': 0.16666667, 'Page 4': 0.13471889, 'Page 2': 0.09453957}
```

Exercise 14

In [43]:
```
import numpy as np

# Define the matrix M from the previous exercise
M = np.array([
    [0.03, 0.03, 0.455, 0.455, 0.03],
    [0.31333333, 0.03, 0.03, 0.03, 0.03],
    [0.31333333, 0.455, 0.03, 0.455, 0.88],
    [0.31333333, 0.455, 0.03, 0.03, 0.03],
    [0.03, 0.03, 0.455, 0.03, 0.03]
])

# Define the initial guess x0 not too close to the actual eigenvector
x0 = np.array([1, 0, 0, 0, 0])

# Placeholder for the actual eigenvector q
q = np.array([0.20664504, 0.12270648, 0.43864676, 0.23200172, 0.0])  # Ex

# Function to compute the 1-norm difference
def norm_difference(M, x0, q, k):
    xk = x0
    differences = []
    ratios = []
    for i in range(k):
        xk = M @ xk
        difference = np.linalg.norm(xk - q, 1)
        differences.append(difference)
        if i > 0:
            ratios.append(differences[i] / differences[i-1])
    return differences, ratios

# Calculate the differences and ratios for k = 1, 5, 10, 50
k_values = [1, 5, 10, 50]
differences, ratios = norm_difference(M, x0, q, max(k_values))

# Extract the required values
results = {k: (differences[k-1], ratios[k-2] if k > 1 else None) for k in

# Calculate c
c = 1 - 2 * np.min(M)

# Calculate the second largest eigenvalue
eigenvalues = np.linalg.eigvals(M)
second_largest_eigenvalue = sorted(eigenvalues, reverse=True)[1]

print({"Norms", "Ratios"}, results)
print("c", c)
print("Second largest eigenvalue", second_largest_eigenvalue)
```

```
{'Norms', 'Ratios'} {1: (0.6039169300000001, None), 5: (0.422910421290361
8, 1.0603507186943075), 10: (0.41785543595143537, 1.0021216076525001), 5
0: (0.41755108035978783, 1.0000000009654022)}
c 0.94
Second largest eigenvalue (0.2858814853553465+0j)
```

Exercise 16

In [56]:

```
import numpy as np

# Define the matrix A
A = np.array([
    [0, 0.5, 0.5],
    [0, 0, 0.5],
    [1, 0.5, 0]
])

# Define the matrix S
S = np.ones((3, 3)) / 3

# Define a dense sampling of m values in the range [0, 1)
m_values = np.linspace(0, 1, 1000, endpoint=False)

# Function to check if a matrix is diagonalizable
def is_diagonalizable(matrix):
    eigenvalues, eigenvectors = np.linalg.eig(matrix)
    rank = np.linalg.matrix_rank(eigenvectors)
    return rank == matrix.shape[0]

# Define the matrix M
M = (1 - m) * A + m * S

# If any value of m in the range [0, 1) is found where M is not diagonali
if len(non_diagonalizable_m) > 0:
    print(f"M is not diagonalizable for all values of 0 <= m < 1")
else:
    print("M is diagonalizable for all values of 0 <= m < 1")
```

```
M is not diagonalizable for all values of 0 <= m < 1
```