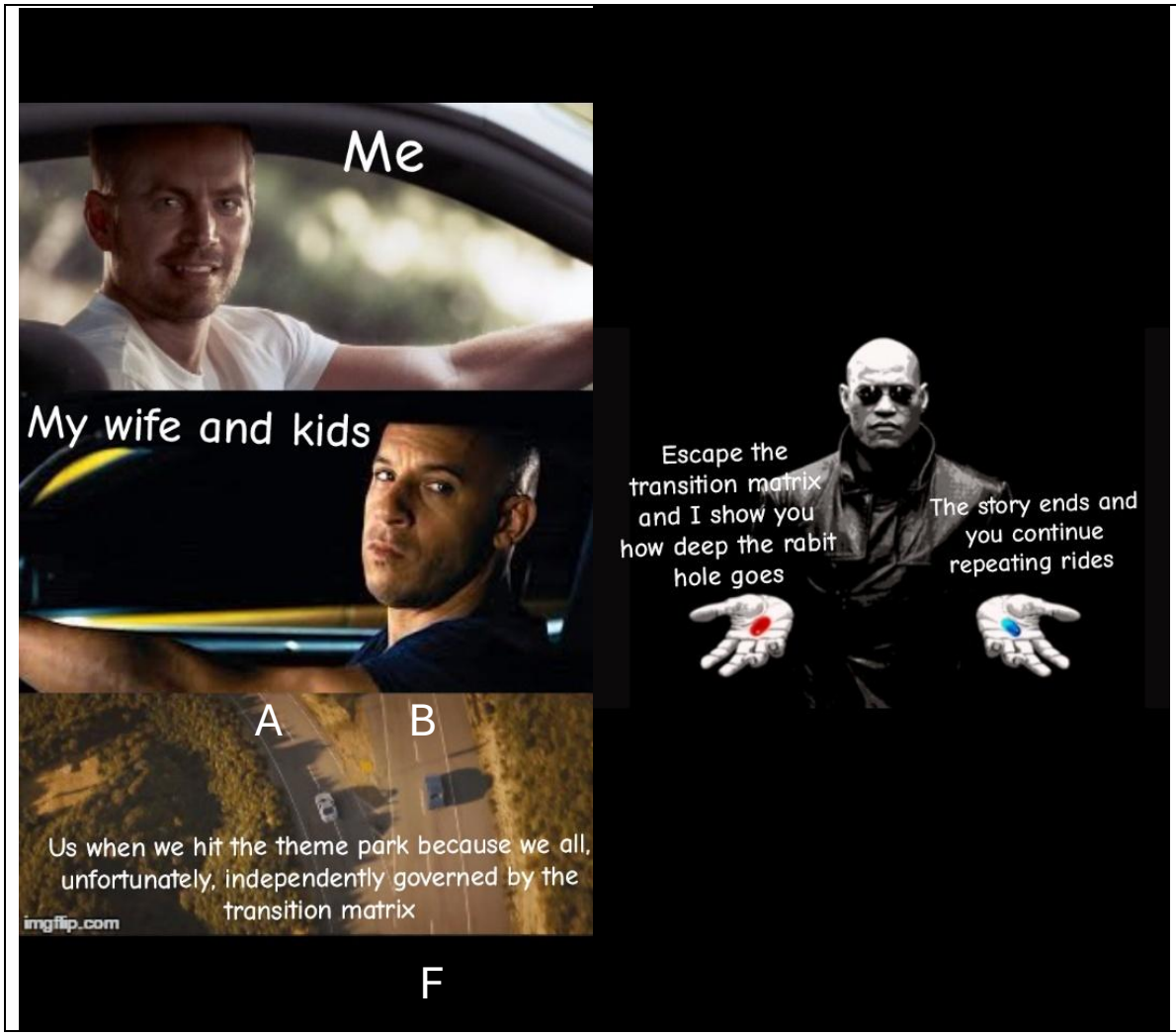
EMTH211-25S2 Assignment 1 - 2025

Names	Theodore Reardon	Joel Rogers
Student ID	47757113	14939437
Signature		

Meme:



Question 1: Theme Park of Doom

(a) What is the long-term probability distribution for the locations of the visitors? Give your answer in terms of percentages.

The transition matrix models the movement between rides (A to J, left to right, top to bottom) that visitors can make. Visitors can either stay on the same ride or move to an adjacent ride, with the probability of moving to a new ride being three times higher than re-riding the same ride.

- $P(Same) = \frac{1}{(1+(3\times adj))}$
- $P(Adjacent) = \frac{3}{(1+(3\times adj))}$

$$P = \begin{bmatrix} 0.14 & 0.43 & 0 & 0 & 0 & 0.43 & 0 & 0 & 0 & 0 \\ 0.3 & 0.1 & 0.3 & 0 & 0 & 0.3 & 0 & 0 & 0 & 0 \\ 0 & 0.3 & 0.1 & 0.3 & 0.3 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.43 & 0.14 & 0.43 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.3 & 0.3 & 0.1 & 0.3 & 0 & 0 & 0 & 0 \\ 0.23 & 0.23 & 0 & 0 & 0.23 & 0.08 & 0 & 0.23 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0.14 & 0.43 & 0 & 0.43 \\ 0 & 0 & 0 & 0 & 0 & 0.23 & 0.23 & 0.08 & 0.23 & 0.23 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.43 & 0.143 & 0.43 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0.3 & 0.3 & 0.3 & 0.1 \end{bmatrix}$$

To find the steady state of the transition matrix P, we solve for the vector $\pi = [\pi_1, \pi_2, \ldots, \pi_{10}]$ such that:

$$\pi P = \pi$$

With the constraint:

$$\sum_{i=1}^{10} \pi_i = 1$$

This is equivalent to solving the system:

$$\pi(P - I) = 0$$

$$\pi = \begin{bmatrix} 0.0745 \\ 0.1064 \\ 0.1064 \\ 0.0745 \\ 0.1064 \\ 0.1383 \\ 0.0745 \\ 0.1383 \\ 0.0745 \\ 0.1064 \end{bmatrix}$$

Ride	Percentage of visitors
A	7.45%
B	10.64%
C	10.64%
D	7.45%
E	10.64%
F	13.83%
G	7.45%
H	13.83%
I	7.45%
J	10.64%

(b) The visitors won't spend infinite time at the theme park, so perhaps the long-term distribution isn't the important thing. If all the visitors start at ride A, where are they in the following steps? How many steps does it take to get the relative error with respect to the long-term distribution (measured with the infinity norm) down below 1%? Present a plot of your results; you may find plt.stackplot useful.

Initial State:

$$\boldsymbol{v}^{(0)} = [1,0,0,0,0,0,0,0,0,0]$$

Compute the state distribution at each step k:

$$\boldsymbol{v}^{(k)} = \boldsymbol{v}^{(0)}$$

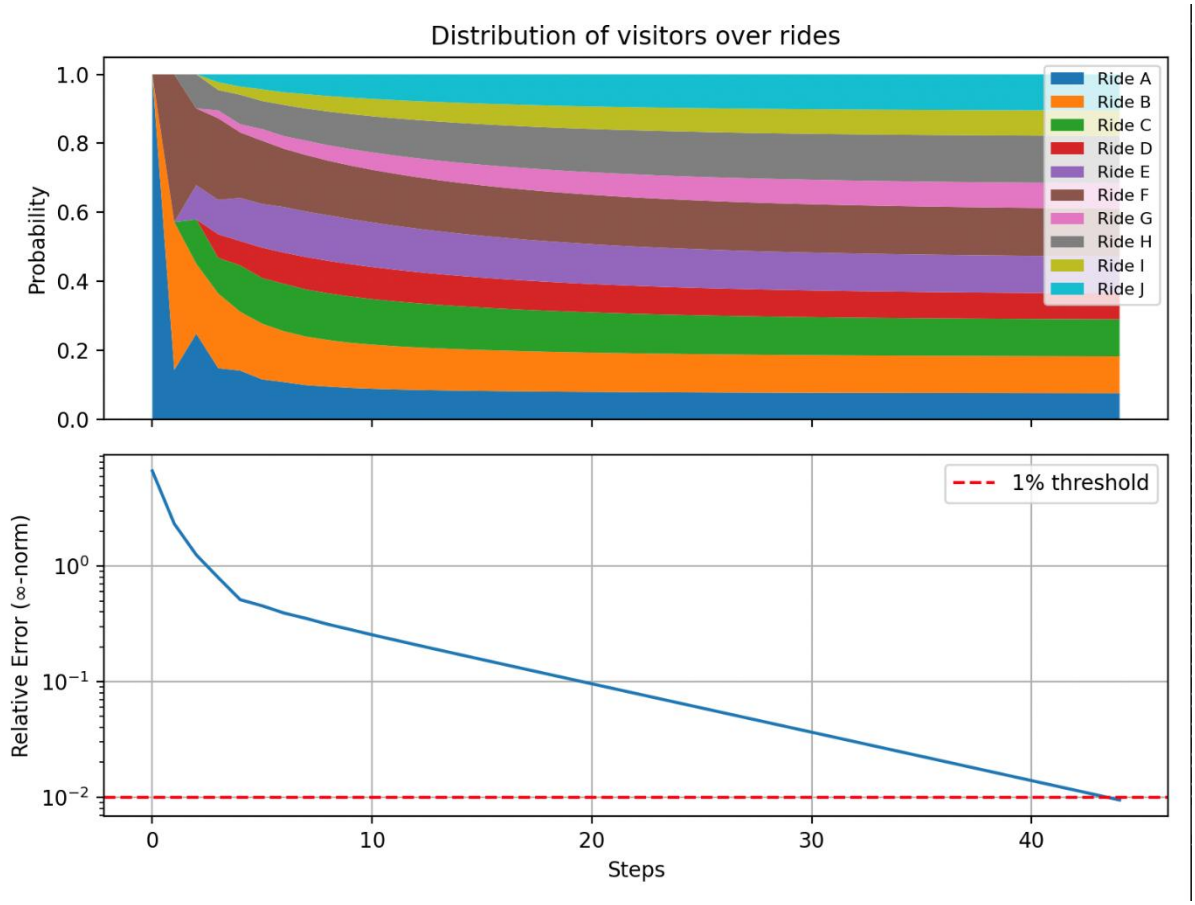
Compare with the stationary distribution π:

$$error(k) = max_i \left| \frac{v_i^{(k)} - \pi_i}{\pi_i} \right|$$

We require $error(k) < 0.01$

Iterate until the error condition is met, Compute $\boldsymbol{v}^{(k)}$for increasing k until the error drops below 1%

Plot:



(c) What would change if the visitors were 5 times as likely to go to adjacent rides instead of 3 times as likely? What would happen if the park disallowed repeating rides immediately? Present predicted results.

i) Visitors being 5 times more likely to go to adjacent rides would mean that the initial dispersion of visitors throughout the park would be much faster compared to the previous 3 times more likely. As the chance of re-rides has majorly decreased. This has the effect of visitors spending more time on the well-connected rides and less on the more isolated rides.

Ride	Percentage of visitors
A	7.33%
B	10.67%
C	10.67%
D	7.33%
E	10.67%
F	14.00%
G	7.33%
H	14.00%
I	7.33%
J	10.67%

ii) If the park disallowed repeating rides this would increase mobility and lead to a more uniform distribution. When immediate repeats are disallowed, all self-loops are set to zero and outgoing probabilities are scaled up proportionally, accelerating the movement between rides and further flattening the long-term distribution.

Ride	**Percentage of visitors**
A	7.14%
B	10.71%
C	10.71%
D	7.14%
E	10.71%
F	14.29%
G	7.14%
H	14.29%
I	7.14%
J	10.71%

iii) In both cases the system converges more rapidly to steady state, and therefore the resulting visitor distribution becomes more evenly spread across the park, with central or highly connected rides gaining more traction amongst the visitors.

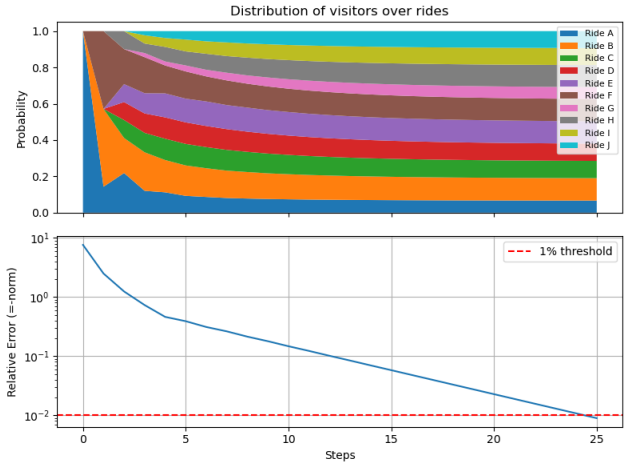
(d) You have enough money to either build one bridge and one path, or three land-based paths. What could you change to improve the evenness of the ride distribution? Describe the change and present predicted results.

$$variance_{original} = 0.1383 - 0.0745 = 0.0638$$

Option 1: Build one bridge and one path

Add bridge between (E and I)

Add path between (B to D)



$$\pi = \begin{bmatrix} 0.066 \\ 0.1226 \\ 0.0943 \\ 0.0943 \\ 0.1226 \\ 0.1226 \\ 0.066 \\ 0.1226 \\ 0.0943 \\ 0.0943 \end{bmatrix}$$

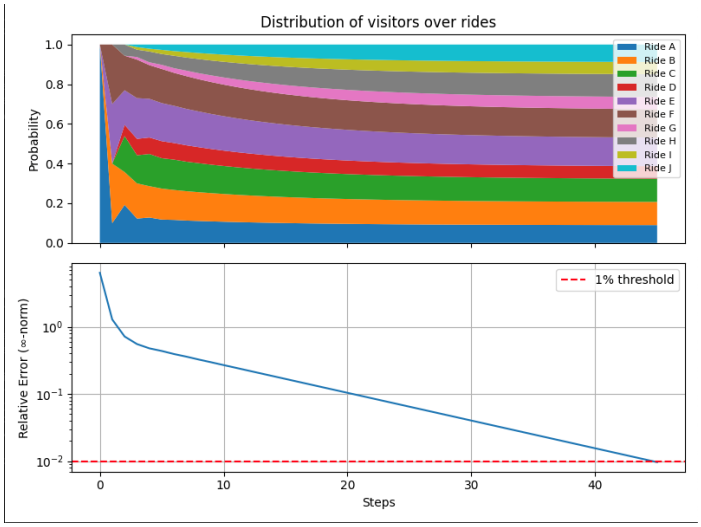
$$variance_1 = 0.1226 - 0.066 = 0.0566$$

Option 2: Build 3 Land based paths

Add path (A to E)

Add path (B to E)

Add path (C to F)



$$\pi = \begin{bmatrix} 0.0893 \\ 0.1161 \\ 0.1161 \\ 0.0625 \\ 0.1429 \\ 0.1429 \\ 0.0625 \\ 0.1161 \\ 0.0625 \\ 0.0893 \end{bmatrix}$$

$$varience_2 = 0.1429 - 0.0625 = 0.0804$$

$$variance_1 < variance_{original}$$

$$variance_2 > variance_{original}$$

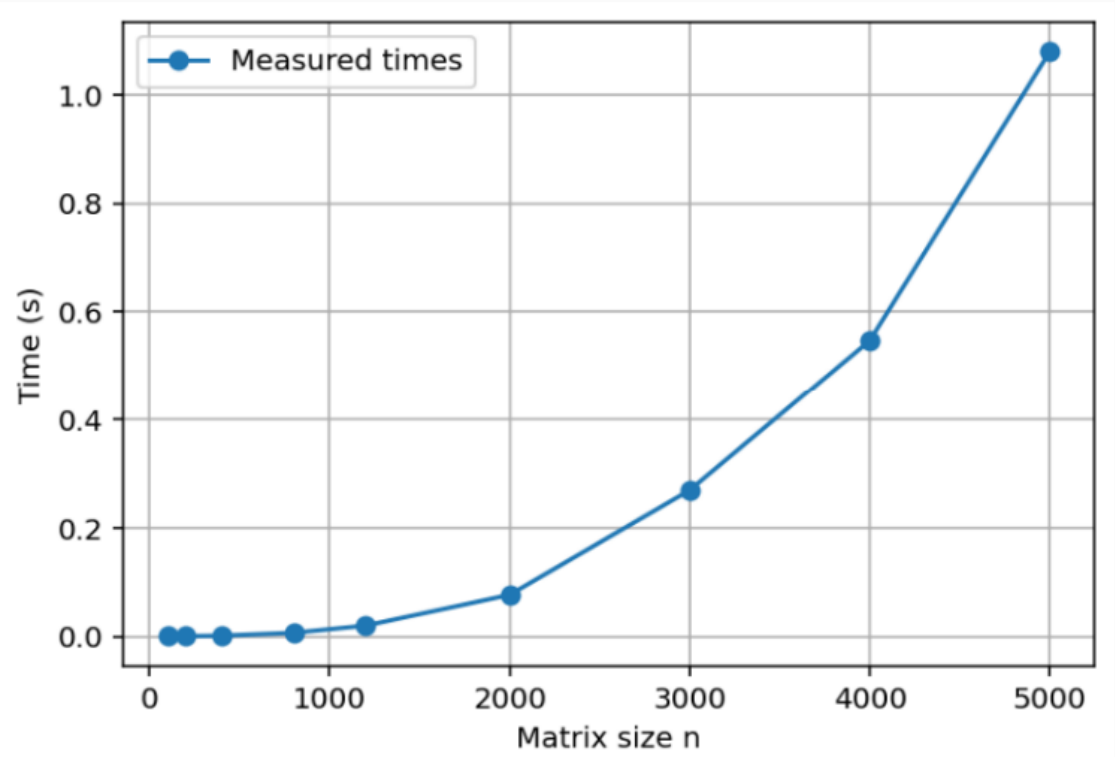
As the variance of option 1, adding a bridge and a path, has decreased compared to the original variance, but option 2 has increased the variance, option 1 is the better option for decreasing the distribution throughout the park.

(e) Would you recommend to an employer that a Markov chain model be used in a situation like this? Discuss the advantages and limitations of Markov chains for this problem. This is an open-ended question, where you may want to consider things such as the Markov chain's convergence speed to long-term behaviour, human behaviour, and the effects of a real-world park layout.

Markov chain models offer a lot of advantages for analysing theme park visitor flow. The simplicity of building a transition matrix clearly represents the probabilities between rides and accurately predicts the long term visitor distributions. This helps to identify rides that are more popular than others and rides that are underused. It also helps with showing how quickly the system stabilizes. However, the model has significant limitations due to the simple assumptions. The Markov chain model assumes visitors next choice is dependent only on their current ride. This ignores factors such as past experiences, wait times or group preferences. As well as this the transition probabilities are treated as constant but does not look into the real world with events such as weather, ride closure, time of day as all of these events can cause fluctuations. The model also overlooks the park layout such as distances between rides which can heavily influence visitor decisions. Despite these limitations, a Markov chain remains a valuable tool for initial planning and theoretical insights.

Question 2: Flop multiplication

(a) Measure reasonably accurate time estimates for how long it takes to calculate A 2 for a broad range of matrix sizes up to and including 5000. Plot the times against the matrix size. (Important considerations: How many matrix sizes do you need? Do you get the same timings each time you run? How much are other things running on your computer affecting the results? Do you need to repeat something multiple times and take the minimum? Are you timing just what you care about or including extra stuff?)



(b) What is the theoretical flop count of multiplying two n×n matrices? Find a scaling factor (effectively time per flop) that makes the theoretical curve go through the time for your largest matrix and add the theoretical prediction curve to your plot. Hint: is this scaling factor the same every time?

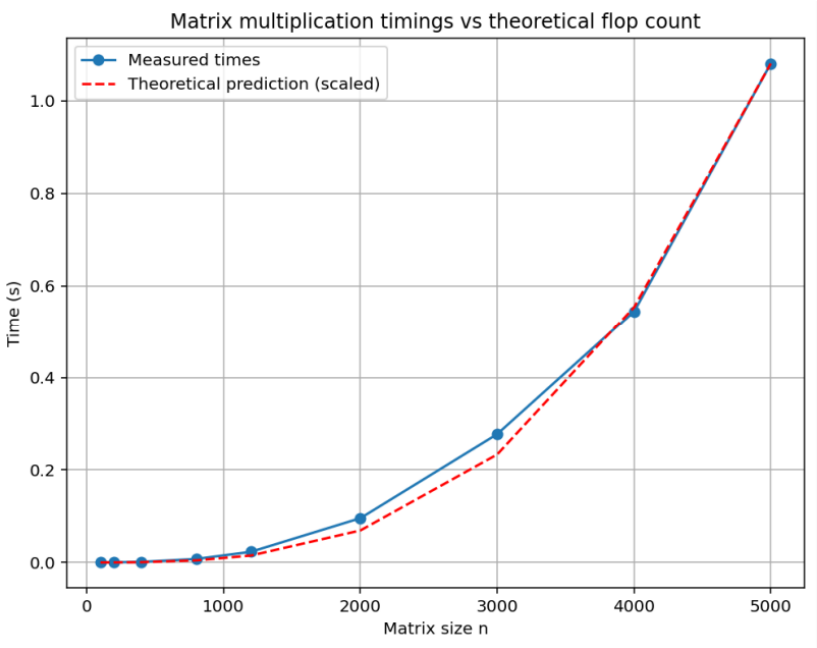
The Theoretical floating-point operation count (flop count) for multiplying two n x n matrices is approximately $2n^3$, accounting for both multiplications and additions. To then relate this to actual computation time, a scaling factor representing the average time per flop is determined by dividing the measured time for the largest matrix size by its theoretical flop count. This scaling factor allows us to plot a theoretical prediction curve. Hower this scaling factor is not constant across all matrix sizes due to different hardware limitations and memory bandwidth. As a result, while the theoretical curve provides a useful benchmark the actual runtimes may vary.

$$Scalling\ Factor\ = \frac{Measured\ Time\ for\ Largest\ Matrix}{Theoretical\ Number\ of\ FLOPs\ for\ Largest\ Matrix}$$

Time to multiply a 5000 x 5000 matrix: 1.107007 seconds

$$FLOPs\ =\ 2\ \times\ 5000^3\ =\ 250{,}000{,}000{,}000\ FLOPs$$

$$Scalling\ Factor\ =\ \frac{1.107}{250{,}000{,}000{,}000} = 4.428 \times 10^{-12}$$



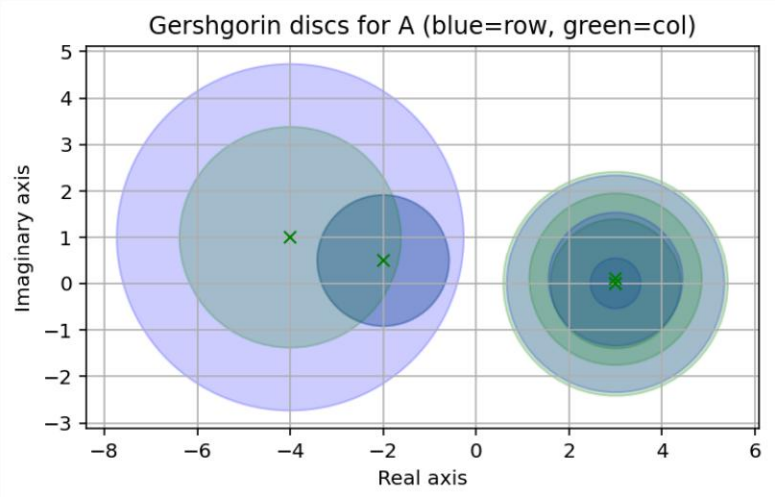
(c) How well do theory and practice agree in this case? If they disagree, what might be the reasons?

The measured timings do agree with the theoretical $O(n^3)$ flop count for large n, showing that matrix multiplication cost grows cubically with size as predicted. The theoretical curve captures the scaling trend well, but practical factors make the actual timing different from the ideal model at both ends of the size range.

Question 3: Complexities of Power

(a) Use Gershgorin's Theorem to find approximate locations for the eigenvalues of A.

Gershgorin's Theorem provides a way to estimate where the eigen values of a square matrix lie in the complex plane.

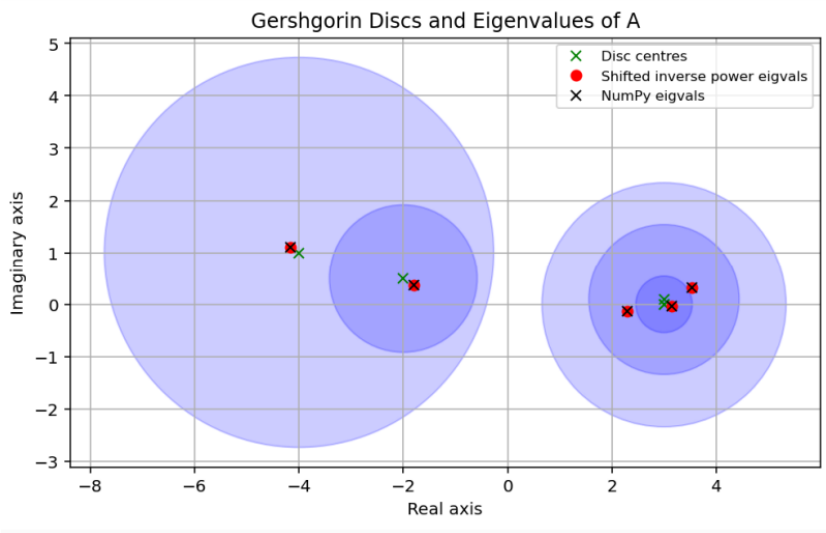


(b) Use the shifted inverse power method to compute all the eigenvalues of A.

The following Python function implements the shifted Inverse Power Method, which is a used to find an eigenvalue of a matrix A closest to a given shift q. This method is particularly useful when we want to locate eigenvalues near a specific complex number.

```
def shifted_inverse_power(A, shift, tolerance=1e-8, max_iterations=500):
    n = A.shape[0]
    identity = np.eye(n, dtype=complex)
    shifted_matrix = A - shift * identity
    vector = np.random.rand(n) + 1j * np.random.rand(n)
    vector = vector / np.linalg.norm(vector)
    prev_rayleigh = np.vdot(vector, A @ vector) / np.vdot(vector, vector)

    for iteration in range(max_iterations):
        y = np.linalg.solve(shifted_matrix, vector)
        vector_new = y / np.linalg.norm(y)
        rayleigh = np.vdot(vector_new, A @ vector_new) / np.vdot(vector_new, vector_new)
        if abs(rayleigh - prev_rayleigh) / abs(rayleigh) < tolerance:
            return rayleigh, vector_new
        vector = vector_new
        prev_rayleigh = rayleigh
    return rayleigh, vector_new
```

This method is applied with multiple shifts chosen near the Gershgorin disc centres and around them to approximate all eigenvalues of the matrix A.



Results

Eigenvalues found (shifted inverse power method):

Eigenvalue 1: -1.801494 + 0.359671j
Eigenvalue 2: 3.150457 + -0.039655j
Eigenvalue 3: 3.530571 + 0.312462j
Eigenvalue 4: 2.293779 + -0.139556j
Eigenvalue 5: -4.173313 + 1.107078j

Eigenvalues from NumPy:

Eigenvalue 1: -1.801494 + 0.359671j
Eigenvalue 2: 3.150457 + -0.039655j
Eigenvalue 3: 3.530571 + 0.312462j
Eigenvalue 4: 2.293779 + -0.139556j
Eigenvalue 5: -4.173313 + 1.107078j

(c) What weaknesses of the shifted inverse power method did you need to consider finding all the eigenvalues? Briefly describe how you worked around these weaknesses.

Repeated eigenvalues: Inverse iteration may converge to the same eigenvalue multiple times. Workaround would be after collecting eigenvalues found from all shifts, the code filters out the duplicates by comparing each new eigen value with already found ones with in a small tolerance (1e-6). Only the unique values get stored in (found_eigenvalues)

Need good shifts: if the shift q is too far from any eigenvalue, the method converges slowly or might not converge at all. The workaround would be to generate multiple shifts around each Gershgorin disc centre, scaled by the radius and offset by various values (0, +-0.5, +-0.5j). This helps cover a neighbourhood where eigenvalues are likely to be, increasing the chance that at least one shift is close to each eigenvalue.

Near-singular system solves: If the shift q is exactly an eigen value the matrix $(A - qI)$ becomes singular, and (np.linalg.solve) will fail. Workaround would be the (shifted_inverse_power)function catches this error (return None for eigenvalue), and calling loop skips these cases with (if val is None: continue)

```
offsets = [0, 0.5, -0.5, 0.5j, -0.5j]
shifts = []
for i in range(len(centres)):
    for offset in offsets:
        shifts.append(centres[i] + radii[i]*offset)

found_eigenvalues = []


for shift in shifts:
    val, _ = shifted_inverse_power(A, shift)
    if val is None:
        continue

    already_found = False
    for v in found_eigenvalues:
        if abs(val - v) < 1e-6:
            already_found = True
            break
    if not already_found:
        found_eigenvalues.append(val)
```

Here is the snippet of code that corresponds to how we worked around these weaknesses as described above.

Appendix

1.a: Code for steady state percentages (Q1a)

```python
import numpy as np
import matplotlib.pyplot as plt


adjacency = {
    "A": ["B", "F"],
    "B": ["A", "F", "C"],
    "C": ["E", "D", "B"],
    "D": ["C", "E"],
    "E": ["F", "C", "D"],
    "F": ["A", "B", "E", "H"],
    "G": ["H", "J"],
    "H": ["F", "I", "J", "G"],
    "I": ["H", "J"],
    "J": ["H", "I", "G"],
}

rides = list(adjacency.keys())
n = len(rides)

P = np.zeros((n, n))

for i, ride in enumerate(rides):
    neighbors = adjacency[ride]
    N_adj = len(neighbors)
    p_stay = 1 / (1 + 3 * N_adj)
    p_move = 3 / (1 + 3 * N_adj)
    P[i, i] = p_stay
    for neigh in neighbors:
        j = rides.index(neigh)
        P[i, j] = p_move



eigvals, eigvecs = np.linalg.eig(P.T)
steady = np.real(eigvecs[:, np.isclose(eigvals, 1)])
steady = steady[:, 0]
steady = steady / steady.sum()

for ride, prob in zip(rides, steady):
    print(f"{ride}: {prob*100:.2f}%")
```

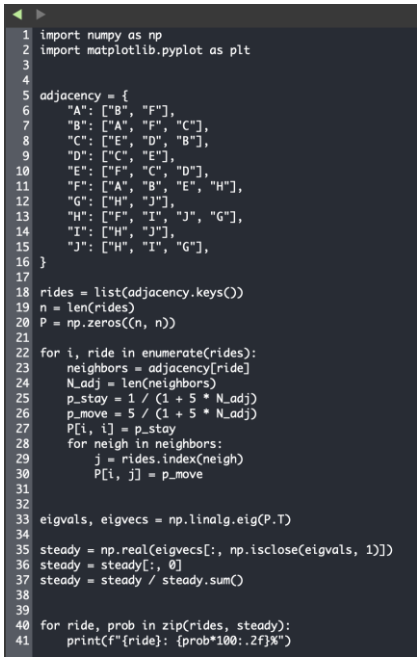
1.b: Code for graphs (Q1b)

```python
import numpy as np
import matplotlib.pyplot as plt


# --- Step 1: Define adjacency graph ---
# Example: A connected to B and C, B connected to A and D, etc.
# You need to replace this with the real ride connections.
adjacency = {
    "A": ["B", "F"],
    "B": ["A", "F", "C"],
    "C": ["E", "D", "B"],
    "D": ["C", "E"],
    "E": ["F", "C", "D"],
    "F": ["A", "B", "E", "H"],
    "G": ["H", "J"],
    "H": ["F", "I", "J", "G"],
    "I": ["H", "J"],
    "J": ["H", "I", "G"],
}

rides = list(adjacency.keys())
n = len(rides)

# --- Step 2: Build transition matrix ---
P = np.zeros((n, n))

for i, ride in enumerate(rides):
    neighbors = adjacency[ride]
    N_adj = len(neighbors)
    p_stay = 1 / (1 + 3 * N_adj)
    p_move = 3 / (1 + 3 * N_adj)

    # Stay probability
    P[i, i] = p_stay

    # Move to each neighbor
    for neigh in neighbors:
        j = rides.index(neigh)
        P[i, j] = p_move


# --- Step 3: Solve for steady-state ---
# Method 1: Eigenvector method
eigvals, eigvecs = np.linalg.eig(P.T)
# Find eigenvector corresponding to eigenvalue 1
steady = np.real(eigvecs[:, np.isclose(eigvals, 1)])
steady = steady[:, 0]
steady = steady / steady.sum()  # normalize



# Stationary distribution (from part a)
pi = steady

# Initial distribution: all at ride A
x = np.zeros(10)
x[0] = 1.0

# Track results
distributions = [x]
errors = []
norm_pi = np.max(pi)

# Iterate
for k in range(100):  # run up to 100 steps just to be safe
    rel_err = np.max(np.abs(x - pi)) / norm_pi
    errors.append(rel_err)
    if rel_err < 0.01:
        print(f"Relative error < 1% at step {k}")
        break
    x = x @ P
    distributions.append(x)

# Convert to array
distributions = np.array(distributions)

labels = [chr(ord('A') + i) for i in range(10)]

# --- Plot ---
fig, (ax1, ax2) = plt.subplots(2,1, figsize=(8,6), sharex=True)


# Distribution stackplot
ax1.stackplot(range(len(distributions)), distributions.T, labels=[f"Ride {l}" for l in labels])
ax1.set_ylabel("Probability")
ax1.set_title("Distribution of visitors over rides")
ax1.legend(loc="upper right", fontsize=8)


# Relative error plot
ax2.plot(errors)
ax2.set_yscale("log")
ax2.set_ylabel("Relative Error (∞-norm)")
ax2.set_xlabel("Steps")
ax2.axhline(0.01, color="red", linestyle="--", label="1% threshold")
ax2.legend()
ax2.grid(True)

plt.tight_layout()
plt.show()
```

1.c: Code for steady state percentages (Q1c.i)

```python
import numpy as np
import matplotlib.pyplot as plt


adjacency = {
    "A": ["B", "F"],
    "B": ["A", "F", "C"],
    "C": ["E", "D", "B"],
    "D": ["C", "E"],
    "E": ["F", "C", "D"],
    "F": ["A", "B", "E", "H"],
    "G": ["H", "J"],
    "H": ["F", "I", "J", "G"],
    "I": ["H", "J"],
    "J": ["H", "I", "G"],
}

rides = list(adjacency.keys())
n = len(rides)


P = np.zeros((n, n))

for i, ride in enumerate(rides):
    neighbors = adjacency[ride]
    N_adj = len(neighbors)
    p_stay = 0
    p_move = 1 / (N_adj)
    P[i, i] = p_stay
    for neigh in neighbors:
        j = rides.index(neigh)
        P[i, j] = p_move


eigvals, eigvecs = np.linalg.eig(P.T)

steady = np.real(eigvecs[:, np.isclose(eigvals, 1)])
steady = steady[:, 0]
steady = steady / steady.sum()  # normalize


for ride, prob in zip(rides, steady):
    print(f"{ride}: {prob*100:.2f}%")
```

1.d: Code for steady state percentages (Q1c.ii)



```python
import numpy as np
import matplotlib.pyplot as plt


adjacency = {
    "A": ["B", "F"],
    "B": ["A", "F", "C"],
    "C": ["E", "D", "B"],
    "D": ["C", "E"],
    "E": ["F", "C", "D"],
    "F": ["A", "B", "E", "H"],
    "G": ["H", "J"],
    "H": ["F", "I", "J", "G"],
    "I": ["H", "J"],
    "J": ["H", "I", "G"],
}

rides = list(adjacency.keys())
n = len(rides)
P = np.zeros((n, n))

for i, ride in enumerate(rides):
    neighbors = adjacency[ride]
    N_adj = len(neighbors)
    p_stay = 1 / (1 + 5 * N_adj)
    p_move = 5 / (1 + 5 * N_adj)
    P[i, i] = p_stay
    for neigh in neighbors:
        j = rides.index(neigh)
        P[i, j] = p_move


eigvals, eigvecs = np.linalg.eig(P.T)

steady = np.real(eigvecs[:, np.isclose(eigvals, 1)])
steady = steady[:, 0]
steady = steady / steady.sum()


for ride, prob in zip(rides, steady):
    print(f"{ride}: {prob*100:.2f}%")
```

1.e: Code for plot (Q1d option 1)

```python
import numpy as np
import matplotlib.pyplot as plt


# --- Step 1: Define adjacency graph ---
# Example: A connected to B and C, B connected to A and D, etc.
# You need to replace this with the real ride connections.
adjacency = {
    "A": ["B", "F"],
    "B": ["A", "F", "C", "E"],
    "C": ["E", "D", "B"],
    "D": ["C", "E", "I"],
    "E": ["F", "C", "D", "B"],
    "F": ["A", "B", "E", "H"],
    "G": ["H", "J"],
    "H": ["F", "I", "J", "G"],
    "I": ["H", "J", "D"],
    "J": ["H", "I", "G"],
}

rides = list(adjacency.keys())
n = len(rides)

# --- Step 2: Build transition matrix ---
P = np.zeros((n, n))

for i, ride in enumerate(rides):
    neighbors = adjacency[ride]
    N_adj = len(neighbors)
    p_stay = 1 / (1 + 3 * N_adj)
    p_move = 3 / (1 + 3 * N_adj)

    # Stay probability
    P[i, i] = p_stay

    # Move to each neighbor
    for neigh in neighbors:
        j = rides.index(neigh)
        P[i, j] = p_move


# --- Step 3: Solve for steady-state ---
# Method 1: Eigenvector method
eigvals, eigvecs = np.linalg.eig(P.T)
# Find eigenvector corresponding to eigenvalue 1
steady = np.real(eigvecs[:, np.isclose(eigvals, 1)])
steady = steady[:, 0]
steady = steady / steady.sum()  # normalize
```

```python

# Stationary distribution (from part a)
pi = steady

# Initial distribution: all at ride A
x = np.zeros(10)
x[0] = 1.0

# Track results
distributions = [x]
errors = []
norm_pi = np.max(pi)

# Iterate
for k in range(100):  # run up to 100 steps just to be safe
    rel_err = np.max(np.abs(x - pi)) / norm_pi
    errors.append(rel_err)
    if rel_err < 0.01:
        print(f"Relative error < 1% at step {k}")
        break
    x = x @ P
    distributions.append(x)

# Convert to array
distributions = np.array(distributions)

labels = [chr(ord('A') + i) for i in range(10)]

# --- Plot ---
fig, (ax1, ax2) = plt.subplots(2,1, figsize=(8,6), sharex=True)


# Distribution stackplot
ax1.stackplot(range(len(distributions)), distributions.T, labels=[f"Ride {l}" for l in labels])
ax1.set_ylabel("Probability")
ax1.set_title("Distribution of visitors over rides")
ax1.legend(loc="upper right", fontsize=8)


# Relative error plot
ax2.plot(errors)
ax2.set_yscale("log")
ax2.set_ylabel("Relative Error (∞-norm)")
ax2.set_xlabel("Steps")
ax2.axhline(0.01, color="red", linestyle="--", label="1% threshold")
ax2.legend()
ax2.grid(True)

plt.tight_layout()
plt.show()
```

1.f: Code to find the long term – distribution (Q1d option 1)

```python
import numpy as np
import matplotlib.pyplot as plt

adjacency = {
    "A": ["B", "F"],
    "B": ["A", "F", "C", "D"],
    "C": ["E", "D", "B"],
    "D": ["C", "E","B"],
    "E": ["F", "C", "D", "I"],
    "F": ["A", "B", "E", "H"],
    "G": ["H", "J"],
    "H": ["F", "I", "J", "G"],
    "I": ["H", "J","E"],
    "J": ["H", "I", "G"],
}

rides = list(adjacency.keys())
n = len(rides)

# --- Step 2: Build transition matrix ---
P = np.zeros((n, n))

for i, ride in enumerate(rides):
    neighbors = adjacency[ride]
    N_adj = len(neighbors)
    p_stay = 1 / (1 + 3 * N_adj)
    p_move = 3 / (1 + 3 * N_adj)

    # Stay probability
    P[i, i] = p_stay

    # Move to each neighbor
    for neigh in neighbors:
        j = rides.index(neigh)
        P[i, j] = p_move


# --- Step 3: Solve for steady-state ---
# Method 1: Eigenvector method
eigvals, eigvecs = np.linalg.eig(P.T)
# Find eigenvector corresponding to eigenvalue 1
steady = np.real(eigvecs[:, np.isclose(eigvals, 1)])
steady = steady[:, 0]
steady = steady / steady.sum() # normalize

for ride, prob in zip(rides, steady):
    print(f"{ride}: {prob*100:.2f}%")

```

1.g: Code for plot (Q1d option 2)

```python
import numpy as np
import matplotlib.pyplot as plt

adjacency = {
    "A": ["B", "F"],
    "B": ["A", "F", "C", "D"],
    "C": ["E", "D", "B"],
    "D": ["C", "E","B"],
    "E": ["F", "C", "D", "I"],
    "F": ["A", "B", "E", "H"],
    "G": ["H", "J"],
    "H": ["F", "I", "J", "G"],
    "I": ["H", "J","E"],
    "J": ["H", "I", "G"],
}

rides = list(adjacency.keys())
n = len(rides)

P = np.zeros((n, n))

for i, ride in enumerate(rides):
    neighbors = adjacency[ride]
    N_adj = len(neighbors)
    p_stay = 1 / (1 + 3 * N_adj)
    p_move = 3 / (1 + 3 * N_adj)
    P[i, i] = p_stay
    for neigh in neighbors:
        j = rides.index(neigh)
        P[i, j] = p_move


eigvals, eigvecs = np.linalg.eig(P.T)
steady = np.real(eigvecs[:, np.isclose(eigvals, 1)])
steady = steady[:, 0]
steady = steady / steady.sum() # normalize

pi = steady

x = np.zeros(10)
x[0] = 1.0

distributions = [x]
errors = []
norm_pi = np.max(pi)

```

```python
for k in range(100):
    rel_err = np.max(np.abs(x - pi)) / norm_pi
    errors.append(rel_err)
    if rel_err < 0.01:
        print(f"Relative error < 1% at step {k}")
        break
    x = x @ P
    distributions.append(x)

distributions = np.array(distributions)

labels = [chr(ord('A') + i) for i in range(10)]


fig, (ax1, ax2) = plt.subplots(2,1, figsize=(8,6), sharex=True)


ax1.stackplot(range(len(distributions)), distributions.T, labels=[f"Ride {l}" for l in labels])
ax1.set_ylabel("Probability")
ax1.set_title("Distribution of visitors over rides")
ax1.legend(loc="upper right", fontsize=8)

ax2.plot(errors)
ax2.set_yscale("log")
ax2.set_ylabel("Relative Error (=-norm)")
ax2.set_xlabel("Steps")
ax2.axhline(0.01, color="red", linestyle="--", label="1% threshold")
ax2.legend()
ax2.grid(True)

plt.tight_layout()
plt.show()
```

1.f: Code to find the long term – distribution (Q1d option 1)

```python
import numpy as np
import matplotlib.pyplot as plt

adjacency = {
    "A": ["B", "F"],
    "B": ["A", "F", "C", "D"],
    "C": ["E", "D", "B"],
    "D": ["C", "E","B"],
    "E": ["F", "C", "D", "I"],
    "F": ["A", "B", "E", "H"],
    "G": ["H", "J"],
    "H": ["F", "I", "J", "G"],
    "I": ["H", "J","E"],
    "J": ["H", "I", "G"],
}
rides = list(adjacency.keys())
n = len(rides)

P = np.zeros((n, n))

for i, ride in enumerate(rides):
    neighbors = adjacency[ride]
    N_adj = len(neighbors)

    total_weight = 1 + 3 * N_adj

    p_stay = 1 / total_weight
    p_move = 3 / total_weight

    P[i, i] = p_stay
    for neigh in neighbors:
        j = rides.index(neigh)
        P[i, j] += p_move

print("Row sums:", P.sum(axis=1))

eigvals, eigvecs = np.linalg.eig(P.T)
idx = np.argmin(np.abs(eigvals - 1))
steady = np.real(eigvecs[:, idx])
steady = steady / steady.sum()

for ride, prob in zip(rides, steady):
    print(f"{ride}: {prob*100:.2f}%")
```

2.a: Code to plot time of against matrix size, measured flop count (Q2a)

```
"""
Created on Fri Sep 19 17:44:34 2025

@author: theor
"""

import numpy as np
import matplotlib.pyplot as plt
import time

sizes = [100, 200, 400, 800, 1200, 2000, 3000, 4000, 5000]
times = []

for n in sizes:
    A = np.random.rand(n, n)
    t_best = float("inf")
    for _ in range(3):
        t0 = time.time()
        A2 = A @ A
        t1 = time.time()
        t_best = min(t_best, t1 - t0)
    times.append(t_best)

plt.plot(sizes, times, "o-", label="Measured times")
plt.xlabel("Matrix size n")
plt.ylabel("Time (s)")
plt.legend()
plt.grid()
plt.show()
```

2.b: Code to plot time against matrix size, theoretical flop count (2b)

```
Created on Fri Sep 19 17:48:05 2025

@author: theor
"""

import numpy as np
import matplotlib.pyplot as plt
import time

sizes = [100, 200, 400, 800, 1200, 2000, 3000, 4000, 5000]
times = []

for n in sizes:
    A = np.random.rand(n, n)
    t_best = float("inf")
    for _ in range(3):
        t0 = time.time()
        A2 = A @ A
        t1 = time.time()
        t_best = min(t_best, t1 - t0)
    times.append(t_best)

#Theoretical model
flops = [2 * (n**3) for n in sizes]

# Scale
scale = times[-1] / flops[-1]
theory_times = [scale * f for f in flops]

plt.figure(figsize=(8,6))
plt.plot(sizes, times, "o-", label="Measured times")
plt.plot(sizes, theory_times, "r--", label="Theoretical prediction (scaled)")
plt.xlabel("Matrix size n")
plt.ylabel("Time (s)")
plt.title("Matrix multiplication timings vs theoretical flop count")
plt.legend()
plt.grid(True)
plt.show()
```

3a) Code to produce Gershgorin's discs to find approximate locations for the eigenvalues of A

```python
import numpy as np
import matplotlib.pyplot as plt

A = np.array([
    [-2. + 0.5j,  0.1 + 0.j,  -0.5 + 1.j,   0. + -0.1j,  0.1 + 0.j],
    [ 0.2 + 0.j,  3. + 0.j,    0. + 0.j,    0.1 + 0.1j,  0. + 0.2j],
    [ 0. + 0.1j, -0.1 + 0.j,   3. + 0.j,    2. + 0.j,    0.1 + -0.1j],
    [ 1. + 0.2j,  0. + 1.j,    0.3 + 0.j,  -4. + 1.j,    1. + 1.j],
    [ 0.1 + 0.j,  0.2 + 0.j,   1. + 0.j,    0.1 + 0.1j,  3. + 0.1j]
], dtype=complex)

centres = np.diag(A)
row_radii = np.sum(np.abs(A), axis=1) - np.abs(centres)

col_radii = np.sum(np.abs(A), axis=0) - np.abs(centres)

fig, ax = plt.subplots()
ax.grid()

for i in range(A.shape[0]):
    # Row Gershgorin discs (blue)
    ax.add_patch(plt.Circle((centres[i].real, centres[i].imag), row_radii[i], color='blue', alpha=0.2))
    # Column Gershgorin discs (green)
    ax.add_patch(plt.Circle((centres[i].real, centres[i].imag), col_radii[i], color='green', alpha=0.2))
    # Plot centers
    ax.plot(centres[i].real, centres[i].imag, 'gx')

ax.set_aspect('equal')
plt.xlabel("Real axis")
plt.ylabel("Imaginary axis")
plt.title("Gershgorin discs for A (blue=row, green=col)")
plt.show()
```

3b) Code to generate plots of eigenvalues found using shifted inverse method and NumPy on top of the Gershgorin's discs

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

A = np.array([
    [-2+0.5j, 0.1, -0.5+1j, -0.1j, 0.1],
    [0.2, 3, 0, 0.1+0.1j, 0.2j],
    [0.1j, -0.1, 3, 2, 0.1-0.1j],
    [1+0.2j, 1j, 0.3, -4+1j, 1+1j],
    [0.1, 0.2, 1, 0.1+0.1j, 3+0.1j]
], dtype=complex)

centres = np.diag(A)
radii = np.sum(np.abs(A), axis=1) - np.abs(centres)

def shifted_inverse_power(A, shift, tolerance=1e-8, max_iterations=500):
    n = A.shape[0]
    identity = np.eye(n, dtype=complex)
    shifted_matrix = A - shift * identity
    vector = np.random.rand(n) + 1j * np.random.rand(n)
    vector = vector / np.linalg.norm(vector)
    prev_rayleigh = np.vdot(vector, A @ vector) / np.vdot(vector, vector)

    for iteration in range(max_iterations):
        y = np.linalg.solve(shifted_matrix, vector)
        vector_new = y / np.linalg.norm(y)
        rayleigh = np.vdot(vector_new, A @ vector_new) / np.vdot(vector_new, vector_new)
        if abs(rayleigh - prev_rayleigh) / abs(rayleigh) < tolerance:
            return rayleigh, vector_new
        vector = vector_new
        prev_rayleigh = rayleigh
    return rayleigh, vector_new


offsets = [0, 0.5, -0.5, 0.5j, -0.5j]
shifts = []
for i in range(len(centres)):
    for offset in offsets:
        shifts.append(centres[i] + radii[i]*offset)

found_eigenvalues = []


for shift in shifts:
    val, _ = shifted_inverse_power(A, shift)
    if val is None:
        continue

    already_found = False
    for v in found_eigenvalues:
        if abs(val - v) < 1e-6:
            already_found = True
            break
    if not already_found:
        found_eigenvalues.append(val)
```

```python
# ploting results
fig, ax = plt.subplots(figsize=(8, 8))
ax.grid(True)

for i in range(len(centres)):
    circle = plt.Circle((centres[i].real, centres[i].imag), radii[i], color='blue', alpha=0.2)
    ax.add_patch(circle)

for i in range(len(centres)):
    ax.plot(centres[i].real, centres[i].imag, 'gx')

for val in found_eigenvalues:
    ax.plot(val.real, val.imag, 'ro')

numpy_eigvals = np.linalg.eigvals(A)
for val in numpy_eigvals:
    ax.plot(val.real, val.imag, 'kx')

ax.set_aspect('equal')
ax.set_xlabel("Real axis")
ax.set_ylabel("Imaginary axis")
ax.set_title("Gershgorin Discs and Eigenvalues of A")

ax.plot([], [], 'gx', label='Disc centres')
ax.plot([], [], 'ro', label='Shifted inverse power eigvals')
ax.plot([], [], 'kx', label='NumPy eigvals')
ax.legend(loc='upper right', fontsize='small')

plt.show()

print("Eigenvalues found by shifted inverse power method:")
for i, val in enumerate(found_eigenvalues, 1):
    print(f"Eigenvalue {i}: {val.real:.6f} + {val.imag:.6f}j")

print("\nEigenvalues from NumPy:")
for i, val in enumerate(numpy_eigvals, 1):
    print(f"Eigenvalue {i}: {val.real:.6f} + {val.imag:.6f}j")
```