# HW3

February 11, 2024

## 1 Inverse Power Method

### 1.1 Introduction

The Inverse Power Method is an algorithm used in numerical linear algebra to find the eigenvalues and corresponding eigenvectors of a matrix. It is particularly useful for finding the eigenvalue of a matrix that is closest to a given number. This method is a variant of the Power Method, which is designed to find the eigenvalue of largest absolute value and its associated eigenvector. The Inverse Power Method, however, focuses on the smallest eigenvalue or those near a specific target value. It is particularly beneficial for dimensionality reduction, principal component analysis, and solving systems that require understanding of the underlying data structure through its eigenvectors and eigenvalues.

### 1.2 Basic Idea

The core idea behind the Inverse Power Method is to apply the Power Method to the inverse of the matrix $A$ (i.e. $A^{-1}$) rather than $A$ itself. Since the eigenvalues of $A^{-1}$ are the reciprocals of the eigenvalues of $A$, applying the Power Method to $A^{-1}$ naturally focuses on the smallest eigenvalue of $A$.

### 1.3 Steps of the Inverse Power Method

1. **Shift the Matrix (if necessary):** If a specific eigenvalue close to a known value $\mu$ is sought, apply a shift to the matrix. This involves computing $B = A - \mu I$, where $I$ is the identity matrix. This shift moves the eigenvalues of $A$ so that the one closest to $\mu$ becomes the dominant eigenvalue of $B^{-1}$.

2. **Choose an Initial Vector:** Select an initial guess vector $x_0$ that is not orthogonal to the desired eigenvector. This vector will be iteratively refined to approximate the eigenvector associated with the eigenvalue of interest.

3. **Iterate:** At each iteration, solve $Bx_{k+1} = x_k$ for $x_{k+1}$, where $B = A - \mu I$ (or $B = A$ if no shift is applied). This typically involves solving a system of linear equations, which can be computationally intensive but is manageable with modern algorithms and computing power.

4. **Normalization:** After each iteration, normalize the vector $x_{k+1}$ to prevent numerical overflow or underflow.

5. **Convergence Check:** Determine if the sequence of vectors $x_k$ is converging. This can be done by checking if the ratio $\frac{|x_{k+1}|}{|x_k|}$ stabilizes, or if changes between successive vectors $x_k$ and $x_{k+1}$ are below a certain threshold.

6. **Compute the Eigenvalue:** Once convergence is achieved, the eigenvalue $\lambda$ closest to $\mu$ can be approximated using the Rayleigh quotient, $\lambda \approx \frac{x^T A x}{x^T x}$, where $x$ is the converged eigenvector.

# 2 Deflation Method

## 2.1 Introduction

The Deflation Method is a technique used in numerical linear algebra to reduce the dimensionality of a problem, making it easier to find additional eigenvalues and eigenvectors of a matrix after the first few have already been found. It is particularly relevant when combined with methods like the Power Method or the Inverse Power Method, allowing these algorithms to iteratively find multiple eigenvalues and eigenvectors of a matrix. This approach is crucial in many data science applications where understanding the structure of data through its principal components or latent features is necessary.

## 2.2 Deflation Process

The Deflation Method typically involves modifying the original matrix $A$ after an eigenvalue $\lambda$ and its corresponding eigenvector $v$ have been identified. The goal is to "deflate" the matrix, effectively reducing its dimension in a way that the already found eigenvalue does not influence the search for the next ones. There are several ways to perform deflation, with one common approach being to subtract from $A$ a rank-one matrix constructed from the outer product of the eigenvector $v$:

$A' = A - \lambda v v^T$

where $v$ is normalized such that $v^T v = 1$. This operation reduces the influence of the found eigenvalue on $A'$, making it possible to apply the Inverse Power Method again to find the next eigenvalue and its corresponding eigenvector.

# 3 Relationship Between Deflation and the Inverse Power Method

The relationship between the Deflation Method and the Inverse Power Method becomes apparent when seeking to find more than the smallest eigenvalue (or those close to a target $\mu$) of a matrix $A$. After applying the Inverse Power Method to find the smallest eigenvalue (or the one closest to $\mu$), the deflation process adjusts the matrix $A$ so that the influence of this found eigenvalue and its corresponding eigenvector is "removed," enabling the subsequent application of the Inverse Power Method to find the next eigenvalue and eigenvector.

# 4 Implementation

## 4.1 Inverse Power Method

[97]:
```
import numpy as np

def inverse_power_method(A, mu=0, tol=1e-10, max_iterations=1000):
    n = A.shape[0]
    I = np.eye(n)  # Create an identity matrix of the same size as A
    # Initial guess for the eigenvector
    x = np.random.rand(n)
```

```
    x = x / np.linalg.norm(x)  # Normalize the initial vector

    for _ in range(max_iterations):
        try:
            # Solve (A - mu*I)x_{k+1} = x_k for x_{k+1}. Incorporate mu here
            y = np.linalg.solve(A - mu*I, x)
        except np.linalg.LinAlgError:
            print("A - mu*I is a Singular matrix")
            return None, None

        # Normalize y to get the next approximation of the eigenvector
        x_next = y / np.linalg.norm(y)

        # Check for convergence (if the direction of x_next and x are not␣
↪changing significantly)
        if np.linalg.norm(x_next - x) < tol:
            break

        x = x_next

    # Use the Rayleigh quotient to estimate the eigenvalue
    eigenvalue = np.dot(x.T, A.dot(x)) / np.dot(x.T, x)

    return eigenvalue, x
```

- This implementation starts with a random initial guess for the eigenvector and normalizes it.
- At each iteration, it solves the equation $Ax = \lambda x$ by solving $A^{-1}y = x$ for $y$ (using `np.linalg.solve`), where $y$ becomes the next approximation of the eigenvector associated with the smallest eigenvalue.
- The vector $y$ is normalized at each step to ensure numerical stability.
- The algorithm checks if the change in direction of the eigenvector approximation between iterations is below a tolerance threshold, indicating convergence.
- Once the eigenvector has been approximated, the corresponding eigenvalue is estimated using the Rayleigh quotient.

### 4.2 Deflation Method

[98]:
```
def deflate_symmetric_matrix(A, lambda1, v1):
    """
    Parameters:
    - A: The original symmetric matrix.
    - lambda1: The eigenvalue to be deflated.
    - v1: The corresponding normalized eigenvector of lambda1.

    Returns:
```

```
    - A_deflated: The deflated matrix.
    """
    n = A.shape[0]
    A_deflated = np.copy(A)
    for i in range(n):
        for j in range(i, n):  # Only need to iterate through half due to␣
↪symmetry
            A_deflated[i, j] -= lambda1 * v1[i] * v1[j]
            if i != j:
                A_deflated[j, i] = A_deflated[i, j]  # Ensure the matrix␣
↪remains symmetric

    return A_deflated
```

This function works by subtracting from each element of the matrix `A` the product of `lambda1`, the corresponding entry from the outer product of `v1` with itself, ensuring that the matrix remains symmetric. This adjustment effectively reduces the influence of the specified eigenvalue and its eigenvector, setting the stage for finding additional eigenpairs.

## 5 Test

For testing our methods we use the following script to create a random matrix and then symmetrize it to ensure it's symmetric. This approach guarantees the matrix has real eigenvalues, which is suitable for our demonstration.

[99]:
```
def create_symmetric_matrix(size):
    """
    Parameters:
    - size: The size of the matrix (number of rows & columns).

    Returns:
    - A symmetric matrix with real eigenvalues.
    """
    # Generate a random matrix
    random_matrix = np.random.rand(size, size)
    # Symmetrize the matrix to ensure it's symmetric
    symmetric_matrix = (random_matrix + random_matrix.T) / 2

    return symmetric_matrix
```

[105]:
```
import sys
import warnings
warnings.filterwarnings('ignore')

# Step 1: Generate a symmetric matrix
size = 4  # Size of the matrix
A = create_symmetric_matrix(size)
```

```
mu = 0
# Initialize lists to store eigenvalues and eigenvectors
eigenvalues = []
eigenvectors = []

# Number of eigenpairs to find
num_eigenpairs_to_find = 2

for _ in range(num_eigenpairs_to_find):
    # Step 2: Apply the inverse power method
    lambda1, v1 = inverse_power_method(A, mu)
    if (lambda1 is None) or (v1 is None):
        continue
    # Store the found eigenvalue and eigenvector
    eigenvalues.append(lambda1)
    eigenvectors.append(v1)

    # Step 3: Deflate the matrix
    A = deflate_symmetric_matrix(A, lambda1, v1)

# Display the found eigenvalues and eigenvectors
for i in range(len(eigenvalues)):
    print(f"Eigen pair number {i+1}")
    print(f"Eigenvalue:{eigenvalues[i]}")
    print(f"Eigenvector:{eigenvectors[i]}")
    print("-----------")
```

```
Eigen pair number 1
Eigenvalue:0.16802175264781405
Eigenvector:[ 0.28974981 -0.74836594 -0.0736326   0.59209097]
-----------
Eigen pair number 2
Eigenvalue:-1.0941040222874267e-16
Eigenvector:[ 0.28974981 -0.74836594 -0.0736326   0.59209097]
-----------
```

In this example we consider $\mu = 0$ (without any shift), so the eigenvalue and eigenvector found represent the least dominant (in terms of the magnitude of eigenvalues) characteristic of the system described by $A$

After successfully applying the Inverse Power Method to find an eigenvalue and its eigenvector, the Deflation Method can be used to modify the matrix so that the next smallest eigenvalue can be determined.