LAB 3

1.Gauss Jordan Method

Working Principle:

The Gauss-Jordan method is an algorithm for solving a system of linear equations. It transforms the augmented matrix into a reduced row echelon form (RREF) using row operations. The solution is obtained directly from the final matrix.

Pseudo code Input: Augmented matrix [A|b] representing the system of linear equations Ax = b Output: Solution vector x

1. Let matrix A be of size n x n and vector b of size n.
2. Combine A and b into an augmented matrix [A|b]. AugmentedMatrix = [A | b]
3. For each row i from 1 to n:

   a. Make the leading coefficient (the pivot) of the current row 1:

   - Find the pivot element A[i, i] (diagonal element of the current row).

   - If the pivot is not 1, divide the entire row i by the pivot (A[i, i]). - This will make A[i, i] = 1.

   - RowOperation: Row[i]= Row[i] / A[i, i]

   b. For each other row j (j ≠ i), eliminate the variable corresponding to the current column (A[i, i]):

   - Update row j to make the entry at position j, i (A[j, i]) zero.

   - Subtract a multiple of row i from row j, so that A[j, i] = 0.

   - RowOperation: Row[j] = Row[j] - A[j, i] * Row[i]
4. After all rows have been processed, the augmented matrix will be in reduced row echelon form (RREF):
   - The left n x n part of the augmented matrix [A|b] will be an identity matrix I.
   - The right column will contain the solution vector x.
5. Return the solution vector x (the last column of the augmented matrix).
   x =LastColumn(augmentedMatrix)

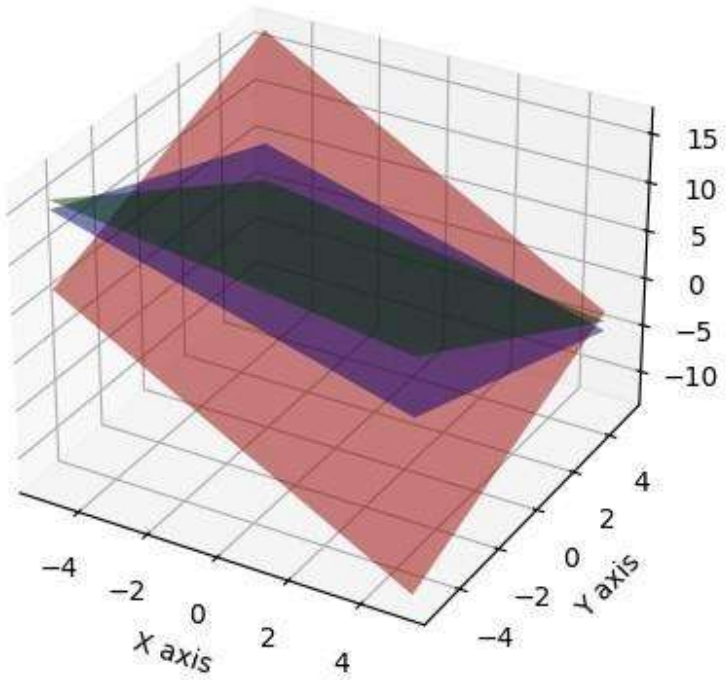
```
import numpy as np
```

```
def gauss_jordan(A, b):
    n = len(b)
    AugmentedMatrix = np.hstack([A, b.reshape(-1, 1)])  # Create
augmented matrix [A|b]

    # Perform Gauss-Jordan elimination
    for i in range(n):
        AugmentedMatrix[i] = AugmentedMatrix[i] / AugmentedMatrix[i,
i]  # Scale pivot to 1
        for j in range(n):
            if i != j:
                AugmentedMatrix[j] = AugmentedMatrix[j] -
AugmentedMatrix[j, i] * AugmentedMatrix[i]

    # Return the solution vector
    return AugmentedMatrix[:, -1]

# Example Usage
A = np.array([[2, -1, 1], [1, 3, 2], [1, 1, 1]], dtype=float)
b = np.array([2, 12, 5], dtype=float)
solution = gauss_jordan(A, b)
print("Solution:", solution)
```

```
Solution: [-1.  1.  5.]
```

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# Create the figure and 3D axis
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

# Define the plane equations based on the system
X = np.linspace(-5, 5, 100)
Y = np.linspace(-5, 5, 100)
X, Y = np.meshgrid(X, Y)
Z1 = (2 - 2*X + Y) / 1
Z2 = (12 - X - 3*Y) / 2
Z3 = 5 - X - Y

# Plot the planes
ax.plot_surface(X, Y, Z1, alpha=0.5, rstride=100, cstride=100,
color='r')
ax.plot_surface(X, Y, Z2, alpha=0.5, rstride=100, cstride=100,
color='g')
ax.plot_surface(X, Y, Z3, alpha=0.5, rstride=100, cstride=100,
color='b')

ax.set_xlabel('X axis')
ax.set_ylabel('Y axis')
```

```
ax.set_zlabel('Z axis')
plt.show()
```




```
import numpy as np
import sys


def read_augmented_matrix(n):
    print("Enter Augmented Matrix Coefficients:")
    matrix = np.zeros((n, n + 1))
    for i in range(n):
        for j in range(n + 1):
            matrix[i][j] = float(input(f"a[{i}][{j}] = "))
    print(matrix)
    return matrix


def gauss_jordan_elimination(a):
    n = len(a)
    for i in range(n):
        # Check for divide-by-zero
        if a[i][i] == 0.0:
            sys.exit("Divide by zero detected in pivot element!")

        for j in range(n):
            if i != j:
                ratio = a[j][i] / a[i][i]
                for k in range(n + 1):
                    a[j][k] -= ratio * a[i][k]
```

```
    # Extracting the solution
    x = np.zeros(n)
    for i in range(n):
        x[i] = a[i][n] / a[i][i]

    return x


def main():
    print("GAUSS-JORDAN ELIMINATION METHOD")
    print()

    default = input("Use default 3x4 matrix? (y/n): ").strip().lower() == 'y'

    if default:
        # Default 3x4 augmented matrix
        n = 3
        a = np.array([
            [2, 1, -1, 8],
            [-3, -1, 2, -11],
            [-2, 1, 2, -3]
        ], dtype=float)
        print("Using Default Matrix:")
        print(a)
    else:
        n = int(input("Enter number of unknowns: "))
        a = read_augmented_matrix(n)

    # Solve using Gauss-Jordan Elimination
    solution = gauss_jordan_elimination(a)

    # Display the solution
    print("Required solution is:")
    for i, value in enumerate(solution):
        print(f"X{i} = {value:.2f}", end="\t")


if __name__ == "__main__":
    main()
```

GAUSS-JORDAN ELIMINATION METHOD

Use default 3x4 matrix? (y/n): y
Using Default Matrix:
[[2. 1. -1. 8.]
 [-3. -1. 2. -11.]
 [-2. 1. 2. -3.]]
Required solution is:
    X0 = 2.00   X1 = 3.00   X2 = -1.00

Test Case: For the system of equations: 2x - y + z = 2 x + 3y + 2z = 12 x + y + z = 5

2.Gauss elimination method with partial pivoting

Working Principle: The Gauss Elimination Method transforms a system of linear equations into an upper triangular matrix by using row operations. Partial Pivoting is used to reduce numerical errors by swapping rows to ensure that the largest absolute value of each column is placed at the pivot position. This minimizes the potential for rounding errors during elimination.

Steps for Gauss Elimination with Partial Pivoting:

Forward Elimination: Using row operations to eliminate variables in each column and create an upper triangular matrix. For each column, identify the largest absolute value in that column (pivot) and swap rows. Eliminate all entries below the pivot using row operations. Back Substitution: After obtaining the upper triangular matrix, solve for the unknowns by substituting values from the last row upwards.

Pseudocode:

Input: Augmented matrix [A|b] of size n x n+1 (for system Ax = b) Output: Solution vector x of size n

1. For i = 1 to n-1:

a. Find the pivot row with the largest absolute value in column i.

b. Swap the current row with the pivot row.

c. For j = i+1 to n: - Eliminate the i-th variable from row j: Row[j] = Row[j] - (A[j, i] / A[i, i]) * Row[i]

2. Back Substitution:

a. For i = n-1 to 0: - x[i] = (b[i] - sum(A[i, j] * x[j] for j = i+1 to n)) / A[i, i]

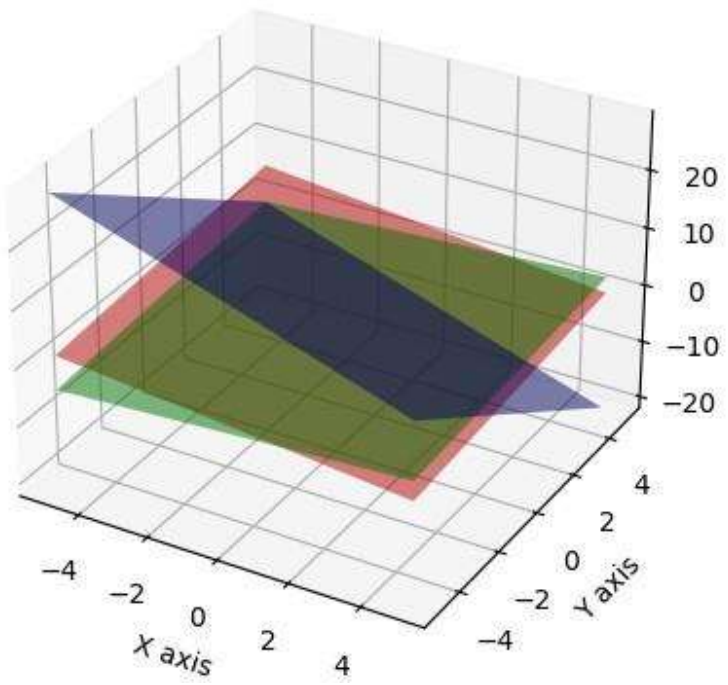
3. Return the solution vector x.

```
import numpy as np

def gauss_elimination_partial_pivoting(A, b):
    n = len(b)
    AugmentedMatrix = np.hstack([A, b.reshape(-1, 1)])  # Create
augmented matrix [A|b]

    # Forward Elimination
    for i in range(n):
        # Pivoting: Find the maximum element in the current column
        max_row = np.argmax(np.abs(AugmentedMatrix[i:n, i])) + i
        AugmentedMatrix[[i, max_row]] = AugmentedMatrix[[max_row, i]]
# Swap rows

        # Eliminate entries below the pivot
        for j in range(i+1, n):
            factor = AugmentedMatrix[j, i] / AugmentedMatrix[i, i]
            AugmentedMatrix[j, i:] -= factor * AugmentedMatrix[i, i:]

    # Back Substitution
    x = np.zeros(n)
    for i in range(n-1, -1, -1):
        x[i] = (AugmentedMatrix[i, -1] - np.dot(AugmentedMatrix[i,
i+1:n], x[i+1:n])) / AugmentedMatrix[i, i]

    return x

# Example Usage
A = np.array([[3, -2,  5], [1,  1, -3], [2,  3,  1]], dtype=float)
b = np.array([3, 3, 4], dtype=float)

solution = gauss_elimination_partial_pivoting(A, b)
print("Solution:", solution)
```

```
Solution: [ 1.73469388  0.28571429 -0.32653061]
```

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# Create the figure and 3D axis
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
```

```
# Define the plane equations based on the system
X = np.linspace(-5, 5, 100)
Y = np.linspace(-5, 5, 100)
X, Y = np.meshgrid(X, Y)
Z1 = (3 - 3*X + 2*Y) / 5
Z2 = (3 - X - Y) / -3
Z3 = (4 - 2*X - 3*Y) / 1

# Plot the planes
ax.plot_surface(X, Y, Z1, alpha=0.5, rstride=100, cstride=100,
color='r')
ax.plot_surface(X, Y, Z2, alpha=0.5, rstride=100, cstride=100,
color='g')
ax.plot_surface(X, Y, Z3, alpha=0.5, rstride=100, cstride=100,
color='b')

ax.set_xlabel('X axis')
ax.set_ylabel('Y axis')
ax.set_zlabel('Z axis')
plt.show()
```



Test Case: For the system of equations: 3x -

2y + 5z = 3 x + y - 3z = 3 2x + 3y + z = 4

3.Gauss-Seidal method

Working Principle: The Gauss-Seidel method is an iterative technique that modifies an initial guess for the solution vector x=(x 1 ,x 2 ,...,x n) to approach the true solution of the system of equations A $x = b$. The method updates each variable in turn using the most recent values available for the other variables.

Steps:

1. Start with an initial guess: $\boldsymbol{x^0} = (x_1^0, x_2^0, \dots, x_n^0)$
2. For each variable x_i, calculate the new value:

$$x_i^{\text{new}} = \frac{1}{a_{ii}}\left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{\text{new}} - \sum_{j=i+1}^{n} a_{ij}x_j^{\text{old}}\right)$$

where a_{ij} are the coefficients of the matrix $\boldsymbol{A}$, and b_i are the elements of the vector $\boldsymbol{b}$.

3. Repeat the process until the solution converges, i.e., the difference between the new and old solutions is sufficiently small (within a specified tolerance).

Pseudocode:

Input: Coefficient matrix A (n x n), vector b (n x 1), initial guess x0 (n x 1), tolerance tol, max iterations max_iter Output: Solution vector x (n x 1)

1. Set k = 0 (iteration counter)
2. Set x = x0 (initial guess)
3. While k < max_iter:
   a. For i = 1 to n: - Calculate the new value of x[i]: x[i] = (b[i] - sum(A[i,j] * x[j] for j = 1 to i-1) - sum(A[i,j] * x[j] for j = i+1 to n)) / A[i,i]
   b. Check if the solution has converged: - If the maximum difference between the new and old x is less than tol, break
   c. Increment k by 1
4. Return the solution vector x

```
import numpy as np

def gauss_seidel(A, b, x0, tol=1e-10, max_iter=1000):
    n = len(b)
    x = np.copy(x0)  # Initial guess
    for k in range(max_iter):
        x_old = np.copy(x)

        # Iterate over each variable
        for i in range(n):
            sum1 = np.dot(A[i, :i], x[:i])    # Sum for previous
variables
            sum2 = np.dot(A[i, i+1:], x_old[i+1:])  # Sum for future
```

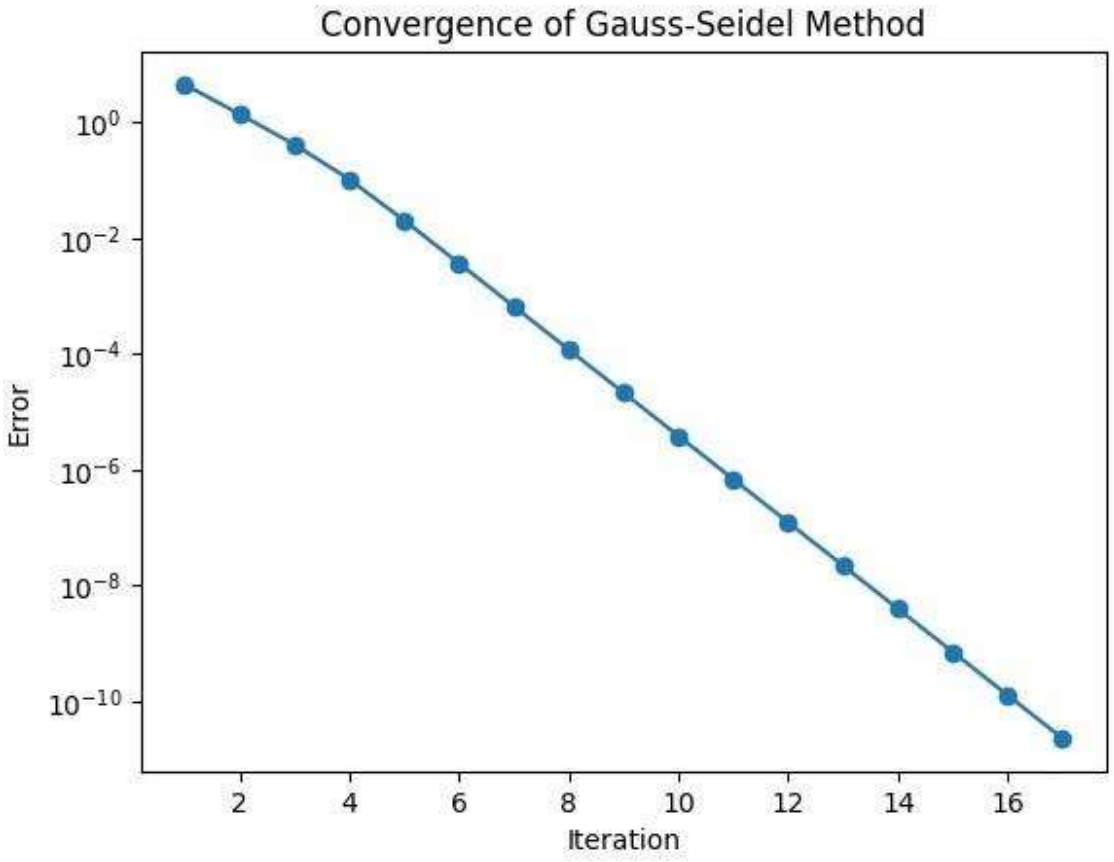
```
variables
            x[i] = (b[i] - sum1 - sum2) / A[i, i]  # Update variable

        # Check for convergence (if the change is small)
        if np.linalg.norm(x - x_old, ord=np.inf) < tol:
            print(f"Converged after {k+1} iterations")
            return x
    print("Max iterations reached")
    return x

# Example Usage
A = np.array([[4, -1, 0, 0],
              [-1, 4, -1, 0],
              [0, -1, 4, -1],
              [0, 0, -1, 3]], dtype=float)

b = np.array([15, 10, 10, 10], dtype=float)

x0 = np.zeros_like(b)

solution = gauss_seidel(A, b, x0)
print("Solution:", solution)
```

```
Converged after 17 iterations
Solution: [5. 5. 5. 5.]
```

```
import matplotlib.pyplot as plt

def gauss_seidel_with_error_plot(A, b, x0, tol=1e-10, max_iter=1000):
    n = len(b)
    x = np.copy(x0)
    error = []

    for k in range(max_iter):
        x_old = np.copy(x)

        # Iterate over each variable
        for i in range(n):
            sum1 = np.dot(A[i, :i], x[:i])    # Sum for previous
variables
            sum2 = np.dot(A[i, i+1:], x_old[i+1:])  # Sum for future
variables
            x[i] = (b[i] - sum1 - sum2) / A[i, i]  # Update variable

        # Compute error (max difference between old and new x)
        err = np.linalg.norm(x - x_old, ord=np.inf)
        error.append(err)

        # Check for convergence
        if err < tol:
            print(f"Converged after {k+1} iterations")
```

```
            break

    # Plot the error for each iteration
    plt.plot(range(1, len(error)+1), error, marker='o')
    plt.yscale('log')  # Log scale to show convergence better
    plt.xlabel('Iteration')
    plt.ylabel('Error')
    plt.title('Convergence of Gauss-Seidel Method')
    plt.show()

    return x

# Example Usage with Error Plot
solution = gauss_seidel_with_error_plot(A, b, x0)
print("Solution:", solution)
```

```
Converged after 17 iterations
```




```
Solution: [5. 5. 5. 5.]
```

Test Case 1: Consider the system of equations:

4x - y = 15 -x + 4y - z = 10 -y + 4z - w = 10 -z + 3w = 10

4.Power Method

Working Principle:

The Power Method is an iterative technique used to find the largest eigenvalue and its corresponding eigenvector of a matrix. The method works as follows:

1. Start with an initial guess for the eigenvector x0.
2. Multiply the matrix A by the current eigenvector approximation x_k to get a new vector y_k

$$y_k = A \cdot x_k$$

3. Normalize the resulting vector y_k to avoid numerical overflow:

$$x_{k+1} = \frac{y_k}{|y_k|}$$

4. Calculate the Rayleigh quotient to estimate the eigenvalue:

$$\lambda_k = \frac{x_k^T \cdot A \cdot x_k}{x_k^T \cdot x_k}$$

5. Repeat the process until the difference between consecutive eigenvalue estimates is smaller than a given tolerance.

Pseudocode: Input: Matrix A, Initial vector x0, Tolerance tol, Maximum iterations max_iter
Output: Eigenvalue λ, Eigenvector x

1. Initialize x0 with a random guess
2. Normalize x0
3. Set λ_old to 0
4. For k = 1 to max_iter:

   a. Multiply A by x_k to get y_k

   b. Normalize y_k to get x_(k+1)

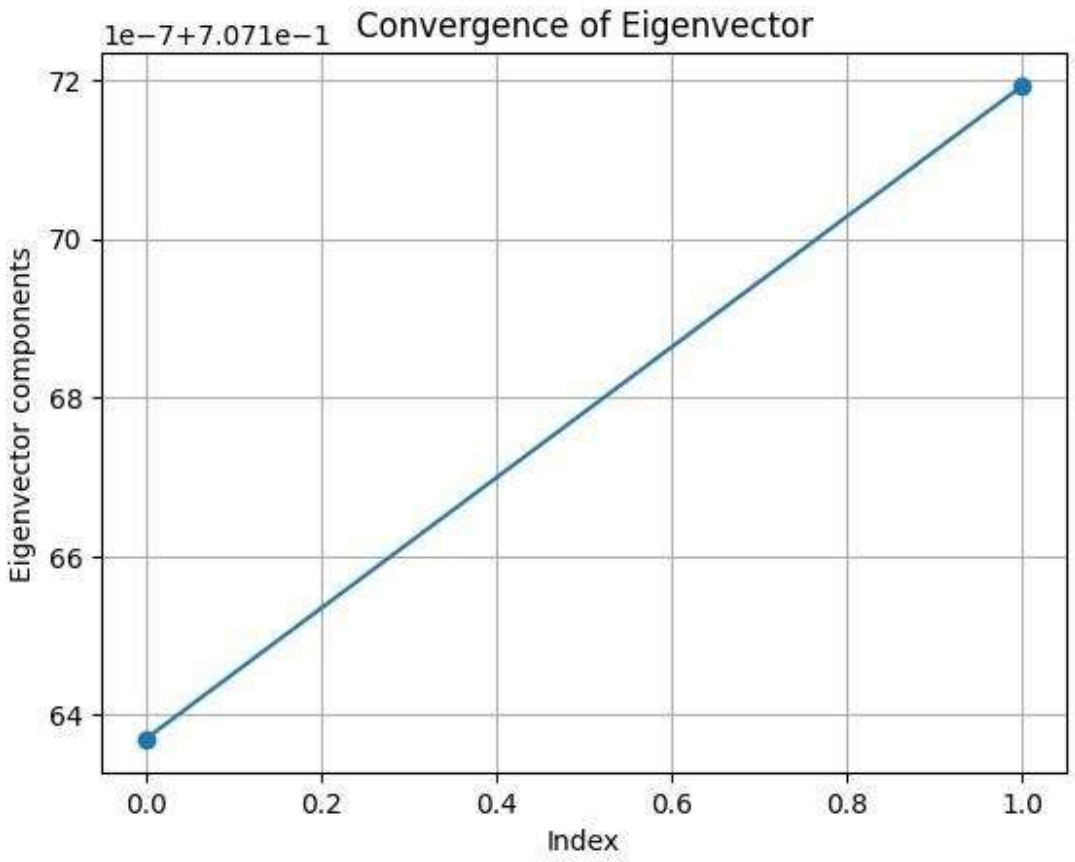
   c. Compute the Rayleigh quotient to estimate λ_k: λ_k = (x_k^T * A * x_k) / (x_k^T * x_k)

   d. If abs(λ_k - λ_old) < tol, break

   e. Set λ_old = λ_k
5. Return λ_k, x_k

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

def power_method(A, x0, tol=1e-6, max_iter=1000):
    # Normalize the initial guess
    x = x0 / np.linalg.norm(x0)
    lambda_old = 0

    # Iterate until convergence
    for i in range(max_iter):
        # Multiply A with the current eigenvector approximation
        y = np.dot(A, x)

        # Normalize the resulting vector
        x = y / np.linalg.norm(y)

        # Compute the Rayleigh quotient for eigenvalue
        lambda_new = np.dot(x.T, np.dot(A, x))

        # Check for convergence
        if abs(lambda_new - lambda_old) < tol:
            break

        lambda_old = lambda_new

    return lambda_new, x

# Example Test Case
A = np.array([[4, 1], [2, 3]])
x0 = np.random.rand(2)  # Random initial guess

# Call power method
eigenvalue, eigenvector = power_method(A, x0)

print("Largest Eigenvalue:", eigenvalue)
print("Corresponding Eigenvector:", eigenvector)

# Visualize the result
plt.plot(np.arange(len(eigenvector)), eigenvector, marker='o')
plt.title('Convergence of Eigenvector')
plt.xlabel('Index')
plt.ylabel('Eigenvector components')
plt.grid(True)
plt.show()
```

```
Largest Eigenvalue: 4.999999417469951
Corresponding Eigenvector: [0.70710637 0.70710719]
```




```python
import numpy as np

def read_matrix(n):
    matrix = np.zeros((n, n))
    print("Enter Matrix Coefficients:")
    for i in range(n):
        for j in range(n):
            matrix[i][j] = float(input(f"a[{i}][{j}] = "))
    return matrix


def read_vector(n):
    vector = np.zeros(n)
    print("Enter Initial Guess Vector:")
    for i in range(n):
        vector[i] = float(input(f"x[{i}] = "))
    return vector


def power_method(a, x, tolerable_error, max_iterations):
    lambda_old = 1.0
    step = 1
    while True:
        # Multiply matrix and vector
        x = np.matmul(a, x)
```

```
        # Compute new eigenvalue and normalize the vector
        lambda_new = max(abs(x))
        x = x / lambda_new

        # Display results
        print()
        print(f"STEP {step}")
        print("-" * 10)
        print(f"Eigenvalue = {lambda_new:.4f}")
        print("Eigenvector:")
        print("[", end=" ")
        for val in x:
            print(f"{val:.4f}", end="\t")
        print("]")

        # Check convergence
        error = abs(lambda_new - lambda_old)
        print(f"Error = {error:.6f}")
        if error < tolerable_error:
            print()
            print("Converged!")
            break

        if step >= max_iterations:
            print()
            print("Not convergent in given maximum iterations!")
            break

        lambda_old = lambda_new
        step += 1


def main():
    print("POWER METHOD IMPLEMENTATION")
    print()

    default = input("Use default matrix and vector? (y/n): ").strip().lower()
== "y"

    if default:
        # Default matrix and vector
        a = np.array([[2, 1, 1],
                      [1, 3, 2],
                      [1, 0, 0]], dtype=float)
        x = np.array([1, 1, 1], dtype=float)
        tolerable_error = 0.0001
        max_iterations = 100
        print("Using Default Values:")
```

```
        print("Matrix:")
        print(a)
        print("Initial Guess Vector:")
        print(x)
        print()
        print(f"Tolerable Error: {tolerable_error}")
        print(f"Maximum Iterations: {max_iterations}")
    else:
        # User input
        n = int(input("Enter order of matrix: "))
        a = read_matrix(n)
        x = read_vector(n)
        tolerable_error = float(input("Enter tolerable error: "))
        max_iterations = int(input("Enter maximum number of steps: "))

    power_method(a, x, tolerable_error, max_iterations)


if __name__ == "__main__":
    main()
```

POWER METHOD IMPLEMENTATION

Use default matrix and vector? (y/n): y
Using Default Values:
Matrix:
[[2. 1. 1.]
 [1. 3. 2.]
 [1. 0. 0.]]
Initial Guess Vector:
[1. 1. 1.]

Tolerable Error: 0.0001
Maximum Iterations: 100

STEP 1

Eigenvalue = 6.0000
Eigenvector:
[0.6667 1.0000 0.1667]
Error = 5.000000

STEP 2

Eigenvalue = 4.0000
Eigenvector:
[0.6250 1.0000 0.1667]
Error = 2.000000

STEP 3

Eigenvalue = 3.9583
Eigenvector:
[0.6105 1.0000 0.1579]
Error = 0.041667

STEP 4

Eigenvalue = 3.9263
Eigenvector:
[0.6059 1.0000 0.1555]
Error = 0.032018

STEP 5

Eigenvalue = 3.9169
Eigenvector:
[0.6044 1.0000 0.1547]
Error = 0.009426

STEP 6

Eigenvalue = 3.9138
Eigenvector:
[0.6039 1.0000 0.1544]
Error = 0.003132

STEP 7

Eigenvalue = 3.9127
Eigenvector:
[0.6037 1.0000 0.1543]
Error = 0.001026

STEP 8

Eigenvalue = 3.9124
Eigenvector:
[0.6037 1.0000 0.1543]
Error = 0.000337

STEP 9

Eigenvalue = 3.9123
Eigenvector:
[0.6036 1.0000 0.1543]
Error = 0.000111

STEP 10

Eigenvalue = 3.9122
Eigenvector:
[0.6036 1.0000 0.1543]
Error = 0.000036

Test Cases:

Input: $A=\begin{matrix}4 & 1\\ 2 & 3\end{matrix}$,

Initial guess =[0.5,0.5]

Expected Output: Eigenvalue≈5,

Eigenvector ≈ [0.707, 0.707]