Q.1. Write a program which demonstrate Addition of two complex numbers

Program

```python
def add_complex_numbers(num1, num2):
    result = num1 + num2
    return result
# Taking input for complex numbers
real_part1 = float(input("Enter the real part of the first complex number: "))
imag_part1 = float(input("Enter the imaginary part of the first complex number: "))
complex_num1 = complex(real_part1, imag_part1)
real_part2 = float(input("Enter the real part of the second complex number: "))
imag_part2 = float(input("Enter the imaginary part of the second complex number: "))
complex_num2 = complex(real_part2, imag_part2)
# Adding complex numbers
result_complex = add_complex_numbers(complex_num1, complex_num2)
# Displaying the result
print(f"The sum of {complex_num1} and {complex_num2} is: {result_complex}")
```

OUTPUT

```
Enter the real part of the first complex number: 5
Enter the imaginary part of the first complex number: -4
Enter the real part of the second complex number: 2
Enter the imaginary part of the second complex number: 3
The sum of (5-4j) and (2+3j) is: (7-1j)
```

Q.2. Write a program for Displaying the conjugate of a complex number

Program

```
def conjugate_complex_number(num):
    conjugate_num = num.conjugate()
    return conjugate_num

# Taking input for a complex number
real_part = float(input("Enter the real part of the complex number: "))
imag_part = float(input("Enter the imaginary part of the complex number: "))
complex_num = complex(real_part, imag_part)

# Calculating and displaying the conjugate
conjugate_result = conjugate_complex_number(complex_num)
print(f"The conjugate of {complex_num} is: {conjugate_result}")
```

OUTPUT

```
Enter the real part of the complex number: 2
Enter the imaginary part of the complex number: 3
The conjugate of (2+3j) is: (2-3j)
```

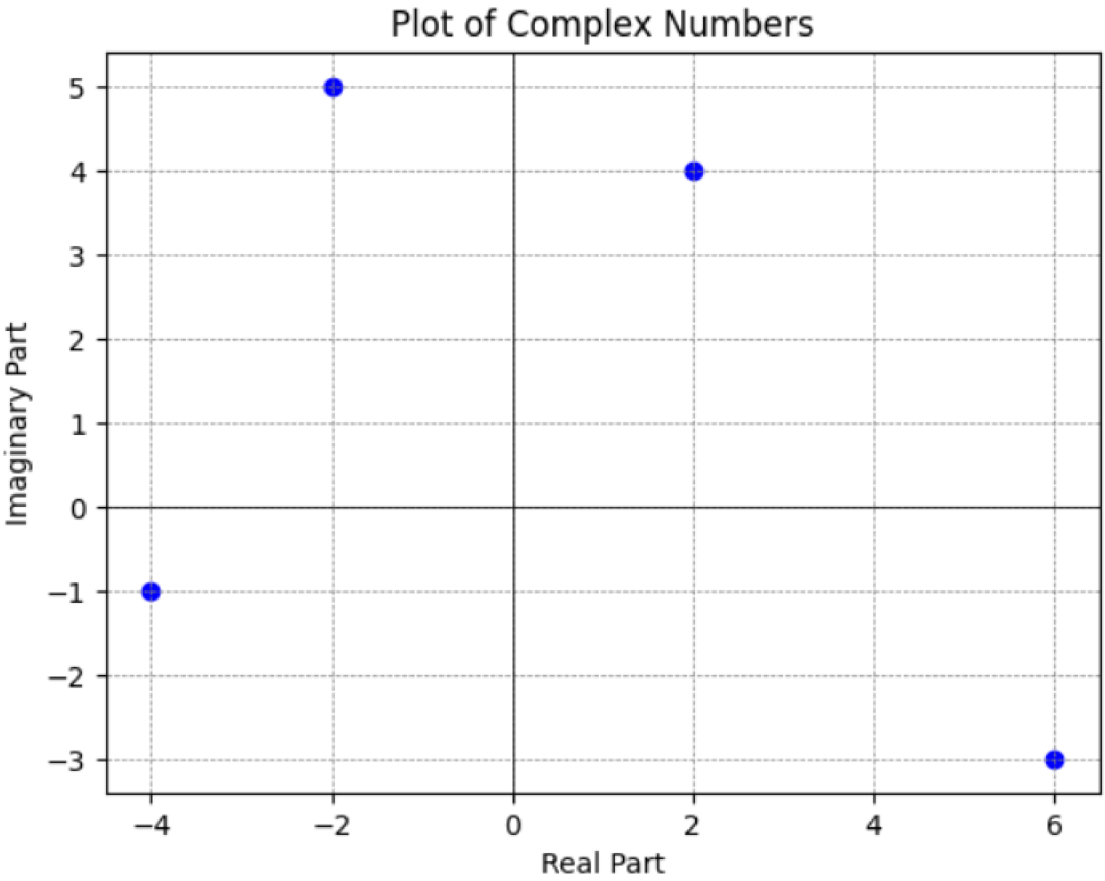
Q.3. Write a program which is Plotting a set of complex numbers

Program

```
import matplotlib.pyplot as plt
# Function to plot complex numbers
def plot_complex_numbers(complex_numbers):
    real_parts = [num.real for num in complex_numbers]
    imag_parts = [num.imag for num in complex_numbers]
    plt.scatter(real_parts, imag_parts, color='blue', marker='o')
    plt.axhline(0, color='black',linewidth=0.5)
    plt.axvline(0, color='black',linewidth=0.5)
    plt.grid(color = 'gray', linestyle = '--', linewidth = 0.5)
    plt.title('Plot of Complex Numbers')
    plt.xlabel('Real Part')
    plt.ylabel('Imaginary Part')
    plt.show()
# Taking input for a set of complex numbers
num_count = int(input("Enter the number of complex numbers to plot: "))
complex_numbers = []
for i in range(num_count):
    real_part = float(input(f"Enter the real part of complex number {i+1}: "))
    imag_part = float(input(f"Enter the imaginary part of complex number {i+1}: "))
    complex_numbers.append(complex(real_part, imag_part))
# Plotting the set of complex numbers
plot_complex_numbers(complex_numbers)
```

OUTPUT

```
Enter the number of complex numbers to plot: 4
Enter the real part of complex number 1: 2
Enter the imaginary part of complex number 1: 4
Enter the real part of complex number 2: -2
Enter the imaginary part of complex number 2: 5
Enter the real part of complex number 3: -4
Enter the imaginary part of complex number 3: -1
Enter the real part of complex number 4: 6
Enter the imaginary part of complex number 4: -3
```


Plot of Complex Numbers
Imaginary Part
Real Part
5
4
3
2
1
0
−1
−2
−3
−4
−2
0
2
4
6

Q.4. Write a program for Creating a new plot by rotating the given number by a degree 90, 180, 270 degrees and by scaling by a number a = 1/2, a = 1/3, a = 2 etc.

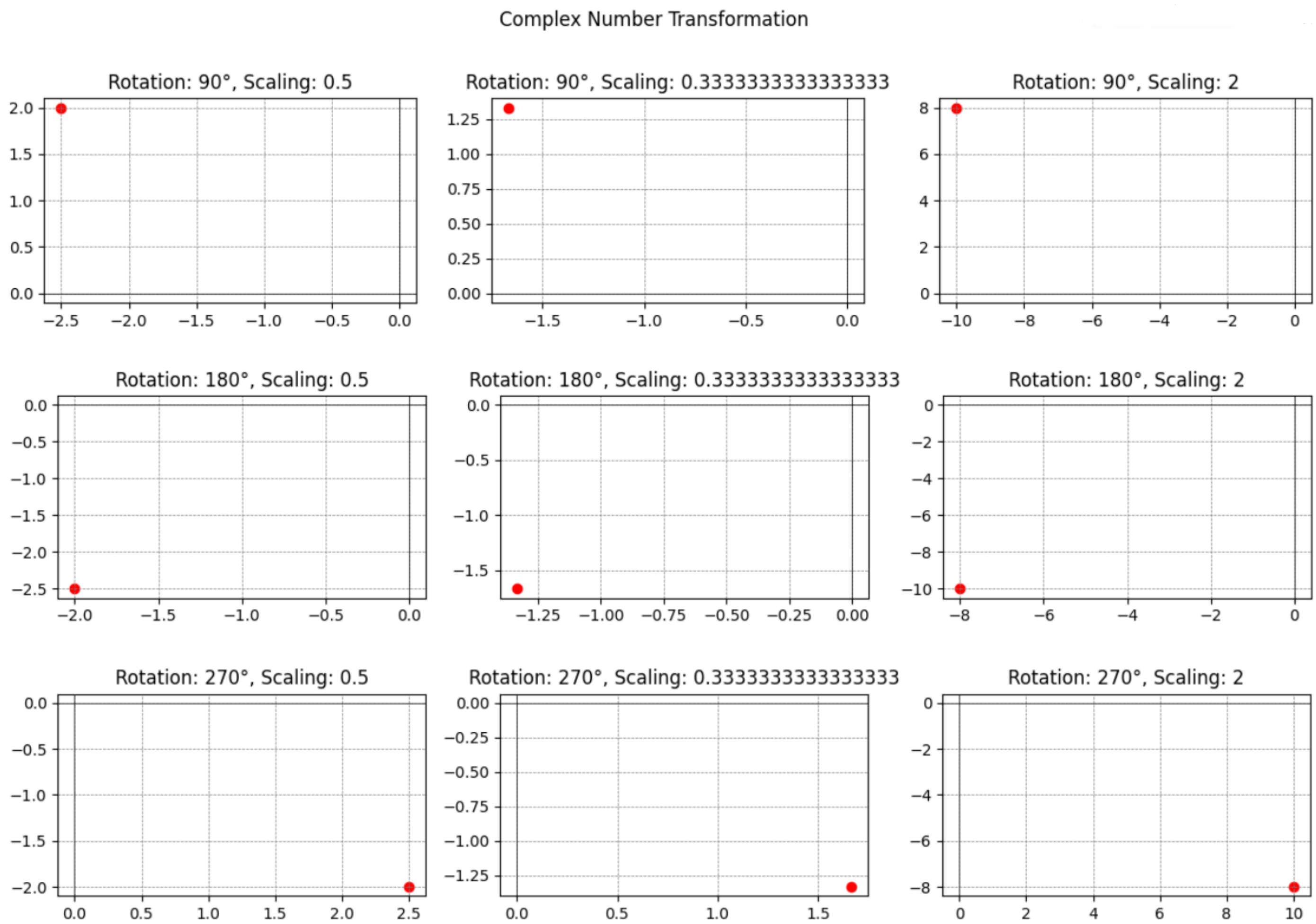
Program

```
import matplotlib.pyplot as plt

import numpy as np


def rotate_and_scale_complex_number(complex_num, rotation_angle,
scaling_factor):

    # Rotate the complex number

    rotated_num = np.exp(1j * np.radians(rotation_angle)) * complex_num


    # Scale the rotated complex number

    scaled_num = scaling_factor * rotated_num


    return scaled_num


# Taking input for a complex number

real_part = float(input("Enter the real part of the complex number: "))

imag_part = float(input("Enter the imaginary part of the complex number: "))

complex_num = complex(real_part, imag_part)


# Creating plots with different rotations and scaling

rotation_angles = [90, 180, 270]

scaling_factors = [1/2, 1/3, 2]


plt.figure(figsize=(12, 8))


for i, angle in enumerate(rotation_angles):

    for j, scale_factor in enumerate(scaling_factors):

        # Calculate the rotated and scaled complex number

        new_complex_num = rotate_and_scale_complex_number(complex_num, angle,
scale_factor)


        # Plot the complex number
```

```
        plt.subplot(len(rotation_angles), len(scaling_factors), i *
len(scaling_factors) + j + 1)

        plt.scatter(new_complex_num.real, new_complex_num.imag, color='red',
marker='o')

        plt.axhline(0, color='black', linewidth=0.5)

        plt.axvline(0, color='black', linewidth=0.5)

        plt.grid(color='gray', linestyle='--', linewidth=0.5)

        plt.title(f'Rotation: {angle}°, Scaling: {scale_factor}')


plt.suptitle('Complex Number Transformation')

plt.tight_layout(rect=[0, 0, 1, 0.97])

plt.show()
```

OUTPUT

```
Enter the real part of the complex number: 4
Enter the imaginary part of the complex number: 5
```



Q.5. Write a program to enter a vector u and v as a n-list.

Program

```
def enter_vector(vector_name):
    try:
        n = int(input(f"Enter the dimension of the {vector_name} vector (n): "))
        if n < 1:
            raise ValueError("Dimension must be a positive integer.")

        vector = []
        for i in range(n):
            element = float(input(f"Enter the {i+1}-th element of the {vector_name} vector: "))
            vector.append(element)

        return vector
    except ValueError as ve:
        print(f"Error: {ve}")
        return None

# Example usage:
vector_u = enter_vector("u")
vector_v = enter_vector("v")

if vector_u is not None and vector_v is not None:
    print("Entered vector u:", vector_u)
    print("Entered vector v:", vector_v)
```

Output

```
Enter the dimension of the u vector (n): 3
Enter the 1-th element of the u vector: 1
Enter the 2-th element of the u vector: 2
Enter the 3-th element of the u vector: 3
Enter the dimension of the v vector (n): 3
Enter the 1-th element of the v vector: 4
Enter the 2-th element of the v vector: 5
Enter the 3-th element of the v vector: 6
Entered vector u: [1.0, 2.0, 3.0]
Entered vector v: [4.0, 5.0, 6.0]
```

Q.6. Write a program to find the vector au + bv for different values of a and b

Program

```
def enter_vector(vector_name):
    try:
        n = int(input(f"Enter the dimension of the {vector_name} vector (n): "))
        if n < 1:
            raise ValueError("Dimension must be a positive integer.")

        vector = []
        for i in range(n):
            element = float(input(f"Enter the {i+1}-th element of the {vector_name} vector: "))
            vector.append(element)

        return vector
    except ValueError as ve:
        print(f"Error: {ve}")
        return None

def linear_combination(u, v, a, b):
    if len(u) != len(v):
        raise ValueError("Vectors must have the same dimension for linear combination.")

    result = [a * u[i] + b * v[i] for i in range(len(u))]
    return result

# Example usage:
vector_u = enter_vector("u")
vector_v = enter_vector("v")

if vector_u is not None and vector_v is not None:
    a = float(input("Enter the value of 'a': "))
```

```
b = float(input("Enter the value of 'b': "))
result_vector = linear_combination(vector_u, vector_v, a, b)
print(f"The result of {a}*u + {b}*v is:", result_vector)
```

Output

```
Enter the dimension of the u vector (n): 3
Enter the 1-th element of the u vector: 2
Enter the 2-th element of the u vector: 1
Enter the 3-th element of the u vector: 3
Enter the dimension of the v vector (n): 3
Enter the 1-th element of the v vector: 3
Enter the 2-th element of the v vector: 2
Enter the 3-th element of the v vector: 1
Enter the value of 'a': 2
Enter the value of 'b': 5
The result of 2.0*u + 5.0*v is: [19.0, 12.0, 11.0]
```

Q.7. Write a program to find the dot product of u and v

Program

```
def enter_vector(vector_name):
    try:
        n = int(input(f"Enter the dimension of the {vector_name} vector (n): "))
        if n < 1:
            raise ValueError("Dimension must be a positive integer.")

        vector = []
        for i in range(n):
            element = float(input(f"Enter the {i+1}-th element of the {vector_name} vector: "))
            vector.append(element)

        return vector
    except ValueError as ve:
        print(f"Error: {ve}")
        return None

def dot_product(u, v):
    if len(u) != len(v):
        raise ValueError("Vectors must have the same dimension for dot product.")

    result = sum(u[i] * v[i] for i in range(len(u)))
    return result

# Example usage:
vector_u = enter_vector("u")
vector_v = enter_vector("v")

if vector_u is not None and vector_v is not None:
    result_dot_product = dot_product(vector_u, vector_v)
    print(f"The dot product of u and v is:", result_dot_product)
```

Output

```
Enter the dimension of the u vector (n): 3
Enter the 1-th element of the u vector: 2
Enter the 2-th element of the u vector: -1
Enter the 3-th element of the u vector: 3
Enter the dimension of the v vector (n): 3
Enter the 1-th element of the v vector: 4
Enter the 2-th element of the v vector: 5
Enter the 3-th element of the v vector: -2
The dot product of u and v is: -3.0
```

Q.8. Write a program to perform Matrix Addition, Subtraction, Multiplication

Program

```
def enter_matrix(matrix_name):
    try:
        rows = int(input(f"Enter the number of rows for {matrix_name}: "))
        cols = int(input(f"Enter the number of columns for {matrix_name}: "))

        if rows < 1 or cols < 1:
            raise ValueError("Rows and columns must be positive integers.")

        matrix = []
        for i in range(rows):
            row = []
            for j in range(cols):
                element = float(input(f"Enter the element at position ({i+1}, {j+1}): "))
                row.append(element)
            matrix.append(row)

        return matrix
    except ValueError as ve:
        print(f"Error: {ve}")
        return None


def print_matrix(matrix):
    for row in matrix:
        print(row)


def matrix_addition(matrix_a, matrix_b):
    if len(matrix_a) != len(matrix_b) or len(matrix_a[0]) != len(matrix_b[0]):
        raise ValueError("Matrices must have the same dimensions for addition.")
```

```
    result = [[matrix_a[i][j] + matrix_b[i][j] for j in
range(len(matrix_a[0]))] for i in range(len(matrix_a))]
    return result


def matrix_subtraction(matrix_a, matrix_b):
    if len(matrix_a) != len(matrix_b) or len(matrix_a[0]) !=
len(matrix_b[0]):
        raise ValueError("Matrices must have the same dimensions for
subtraction.")

    result = [[matrix_a[i][j] - matrix_b[i][j] for j in
range(len(matrix_a[0]))] for i in range(len(matrix_a))]
    return result


def matrix_multiplication(matrix_a, matrix_b):
    if len(matrix_a[0]) != len(matrix_b):
        raise ValueError("Number of columns in the first matrix must be
equal to the number of rows in the second matrix for multiplication.")

    result = [[sum(matrix_a[i][k] * matrix_b[k][j] for k in
range(len(matrix_a[0]))) for j in range(len(matrix_b[0]))] for i in
range(len(matrix_a))]
    return result


# Example usage:
matrix_a = enter_matrix("Matrix A")
matrix_b = enter_matrix("Matrix B")


if matrix_a is not None and matrix_b is not None:
    print("\nMatrix A:")
    print_matrix(matrix_a)


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


    # Matrix Addition
```

```
result_addition = matrix_addition(matrix_a, matrix_b)
print("\nMatrix Addition (A + B):")
print_matrix(result_addition)

# Matrix Subtraction
result_subtraction = matrix_subtraction(matrix_a, matrix_b)
print("\nMatrix Subtraction (A - B):")
print_matrix(result_subtraction)

# Matrix Multiplication
result_multiplication = matrix_multiplication(matrix_a, matrix_b)
print("\nMatrix Multiplication (A * B):")
print_matrix(result_multiplication)
```

Output

```
Enter the number of rows for Matrix A: 2
Enter the number of columns for Matrix A: 2
Enter the element at position (1, 1): 2
Enter the element at position (1, 2): 3
Enter the element at position (2, 1): 1
Enter the element at position (2, 2): 4
Enter the number of rows for Matrix B: 2
Enter the number of columns for Matrix B: 2
Enter the element at position (1, 1): 1
Enter the element at position (1, 2): 2
Enter the element at position (2, 1): 3
Enter the element at position (2, 2): 0

Matrix A:
[2.0, 3.0]
[1.0, 4.0]

Matrix B:
[1.0, 2.0]
[3.0, 0.0]

Matrix Addition (A + B):
[3.0, 5.0]
[4.0, 4.0]

Matrix Subtraction (A - B):
[1.0, 1.0]
[-2.0, 4.0]

Matrix Multiplication (A * B):
[11.0, 4.0]
[13.0, 2.0]
```

Q.9. Write a program to Check if matrix is invertible. If yes then find Inverse.

Program

```
import numpy as np


def enter_matrix(matrix_name):
    try:
        rows = int(input(f"Enter the number of rows for {matrix_name}: "))
        cols = int(input(f"Enter the number of columns for {matrix_name}: "))

        if rows < 1 or cols < 1:
            raise ValueError("Rows and columns must be positive integers.")

        matrix = []
        for i in range(rows):
            row = []
            for j in range(cols):
                element = float(input(f"Enter the element at position ({i+1}, {j+1}): "))
                row.append(element)
            matrix.append(row)

        return matrix
    except ValueError as ve:
        print(f"Error: {ve}")
        return None


def print_matrix(matrix):
    for row in matrix:
        print(row)


def is_invertible(matrix):
    try:
        inverse = np.linalg.inv(matrix)
```

```
        return True
    except np.linalg.LinAlgError:
        return False

def find_inverse(matrix):
    return np.linalg.inv(matrix)

# Example usage:
matrix_a = enter_matrix("Matrix A")

if matrix_a is not None:
    print("\nMatrix A:")
    print_matrix(matrix_a)

    if is_invertible(matrix_a):
        inverse_matrix = find_inverse(matrix_a)
        print("\nThe matrix is invertible.")
        print("\nInverse of Matrix A:")
        print_matrix(inverse_matrix)
    else:
        print("\nThe matrix is not invertible.")
```

Output

```
Enter the number of rows for Matrix A: 2
Enter the number of columns for Matrix A: 2
Enter the element at position (1, 1): 1
Enter the element at position (1, 2): -1
Enter the element at position (2, 1): 2
Enter the element at position (2, 2): 2

Matrix A:
[1.0, -1.0]
[2.0, 2.0]

The matrix is invertible.

Inverse of Matrix A:
[0.5  0.25]
[-0.5   0.25]
```

Q.10. Write a program to convert a matrix into its row echelon form. (Order 2).

Program

```
def enter_matrix(matrix_name):
    try:
        matrix = []
        print(f"Enter the elements for {matrix_name} matrix:")
        for i in range(2):
            row = []
            for j in range(2):
                element = float(input(f"Enter the element at position
({i+1}, {j+1}): "))
                row.append(element)
            matrix.append(row)

        return matrix
    except ValueError as ve:
        print(f"Error: {ve}")
        return None


def print_matrix(matrix):
    for row in matrix:
        print(row)


def row_echelon_form(matrix):
    if matrix[0][0] == 0:
        matrix[0], matrix[1] = matrix[1], matrix[0]

    pivot = matrix[0][0]

    if pivot != 0:
        multiplier = -matrix[1][0] / pivot
        matrix[1] = [matrix[1][i] + multiplier * matrix[0][i] for i in
range(2)]
```

```
    return matrix


# Example usage:
matrix_a = enter_matrix("Matrix A")


if matrix_a is not None:
    print("\nMatrix A:")
    print_matrix(matrix_a)


    row_echelon_result = row_echelon_form(matrix_a)
    print("\nRow Echelon Form of Matrix A:")
    print_matrix(row_echelon_result)
```

Output

```
Enter the elements for Matrix A matrix:
Enter the element at position (1, 1): 2
Enter the element at position (1, 2): 1
Enter the element at position (2, 1): -1
Enter the element at position (2, 2): 3

Matrix A:
[2.0, 1.0]
[-1.0, 3.0]

Row Echelon Form of Matrix A:
[2.0, 1.0]
[0.0, 3.5]
```

Q.11. Write a program to find rank of a matrix

Program

```
def rank_of_matrix(matrix):
    rows = len(matrix)
    cols = len(matrix[0]) if matrix else 0
    rank = 0
    for i in range(min(rows, cols)):
        # Make the diagonal element in the current column equal to 1
        if matrix[i][i] != 0:
            rank += 1
            for j in range(rows):
                if j != i:
                    ratio = matrix[j][i] / matrix[i][i]
                    for k in range(cols):
                        matrix[j][k] -= ratio * matrix[i][k]

    return rank
# Example usage:
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
print("Matrix:")
for row in matrix:
    print(row)
print("Rank of the matrix:", rank_of_matrix(matrix))
```

Output

```
Matrix:
[0, 1, -1]
[2, 1, 1]
[4, 0, 1]
Rank of the matrix: 2
```

Q.12. Write a program to calculate eigenvalue and eigenvector (Order 2)

Program

```
import numpy as np
def calculate_eigen(matrix):
    # Check if the matrix is 2x2
    if matrix.shape != (2, 2):
        raise ValueError("Input matrix must be a 2x2 matrix")
    # Calculate the characteristic equation coefficients
    a = 1
    b = -np.trace(matrix)
    c = np.linalg.det(matrix)
    # Calculate eigenvalues using the quadratic formula
    discriminant = b**2 - 4*a*c
    if discriminant < 0:
        print("Eigenvalues are complex.")
    else:
        eigenvalue1 = (-b + np.sqrt(discriminant)) / (2*a)
        eigenvalue2 = (-b - np.sqrt(discriminant)) / (2*a)
        print("Eigenvalues:", eigenvalue1, eigenvalue2)
        # Calculate eigenvectors
        eigenvector1 = np.linalg.solve(matrix - eigenvalue1 *
np.identity(2), np.array([1, 0]))
        eigenvector2 = np.linalg.solve(matrix - eigenvalue2 *
np.identity(2), np.array([1, 0]))
        print("Eigenvector 1:", eigenvector1)
        print("Eigenvector 2:", eigenvector2)
# Example usage
matrix = np.array([[4, 2],
                   [1, 3]])
calculate_eigen(matrix)
```

Output

```
Eigenvalues: 4.999999999999999 2.000000000000001
Eigenvector 1: [7.50599938e+14 3.75299969e+14]
Eigenvector 2: [-3.75299969e+14  3.75299969e+14]
```