NUMERICAL ANALYSIS

NAGANANDANA NAGENDRA

2021A7PS0215U

15/04/2024

NEWTON'S METHOD

Newton's method, also known as Newton-Raphson method, is an iterative root-finding algorithm used for finding successively better approximations to the roots (or zeroes) of a real-valued function.

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

where,

Xn+1 : next approximation

Xn : current approximation

f(Xn) : value of the function with current approximation

f '(Xn) : derivative of the function with current approximation

PROCEDURE:

Program code:

1. Define the function f(x) and its derivative f'(x).
2. Input the function, its derivative, initial guess for x, and the number of iterations.
3. Initialize an empty list to store the roots.
4. Iterate through the specified number of iterations using a for loop.
5. Inside the loop, calculate the next approximation of the root using Newton's method formula.
6. Append the new approximation to the list of roots.
7. Update the current guess for x with the new approximation.
8. Print the final root found after the specified number of iterations.
9. Plot the function and tangent lines generated during each iteration.

10. Display the plot.

Function plot:

1. Generate a vector of x values to plot the function f(x) over a specified interval.

2. Evaluate the function f(x) for each x value to obtain corresponding y values.

3. Plot the function curve using the x and y values.

4. For each root approximation found during iterations, calculate and plot the tangent line using the derivative at that point.

5. Mark the final root on the plot with a distinctive marker.

6. Add labels to the x-axis and y-axis to provide context.

7. Set the location of the x-axis and y-axis to the origin to improve visualization.

PROBLEMS:

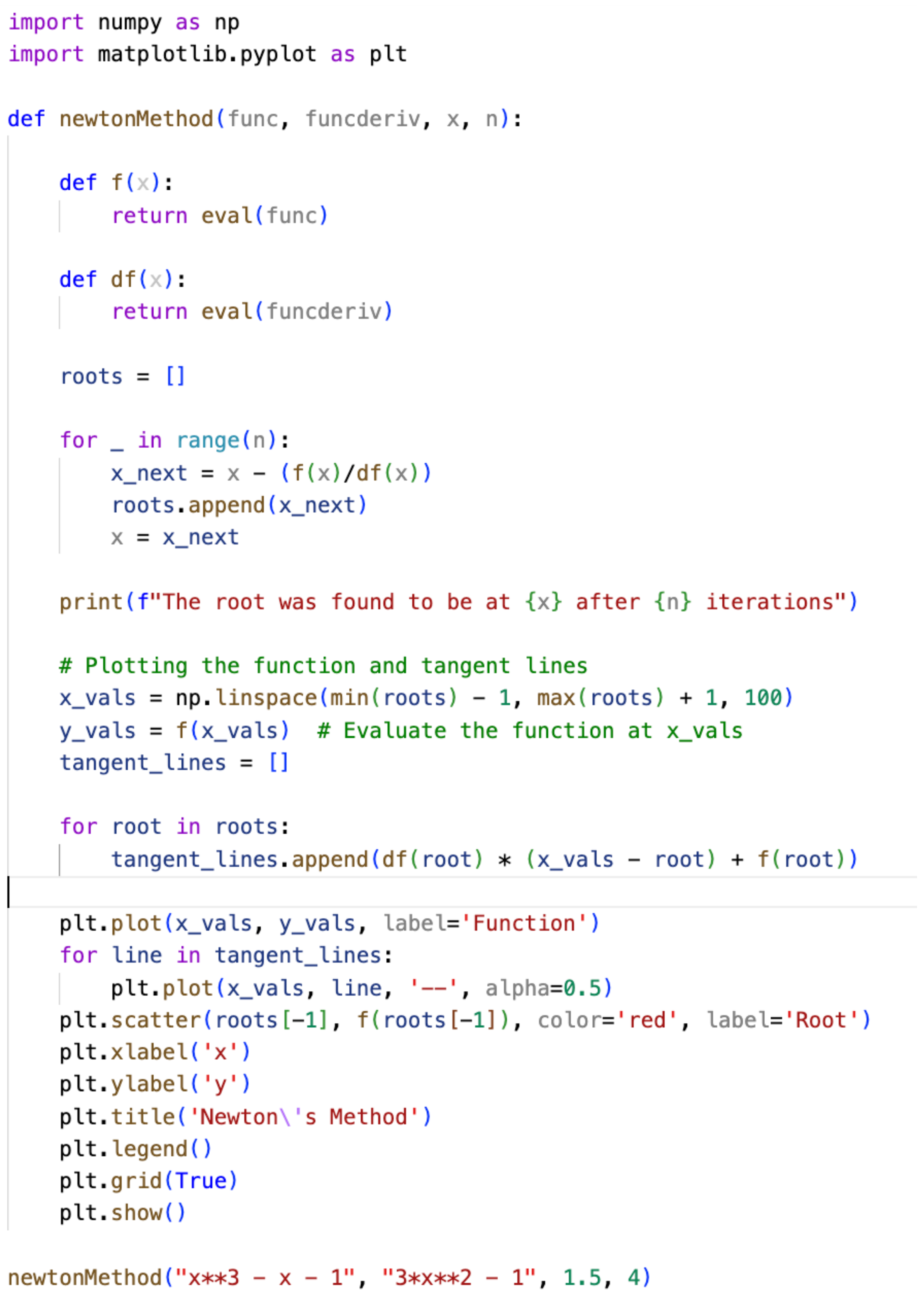
Q1

Suppose you are designing a roller coaster track, and you need to determine the height of a loop at a certain distance along the track to ensure that the roller coaster has enough velocity to complete the loop without falling off. The height of the loop at a distance x along the track can be modeled by the function:

f(x) = x**3 - x - 1

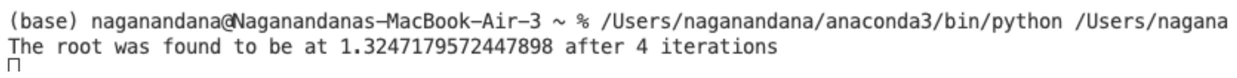
You want to find the distance x at which the height of the loop reaches a critical value, say 20 meters. Use Newton's method to find the distance x at which the height of the loop equals 20 meters, given that the initial guess for x is 1.5 meters. Additionally, plot the roller coaster track's height function and the tangent lines generated during each iteration of Newton's method to visualize the convergence to the solution.

CODE:



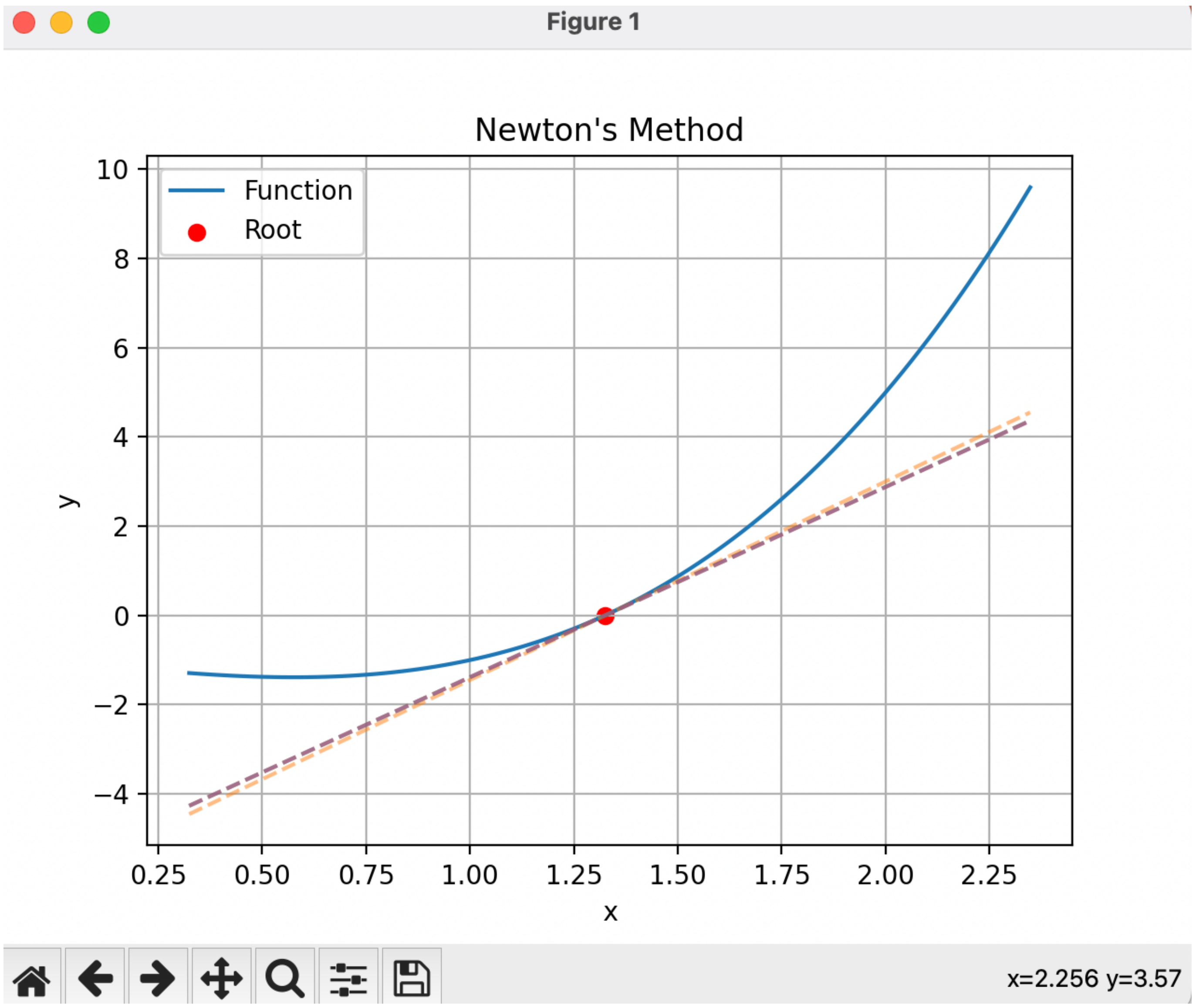
```python
import numpy as np
import matplotlib.pyplot as plt

def newtonMethod(func, funcderiv, x, n):

    def f(x):
        return eval(func)

    def df(x):
        return eval(funcderiv)

    roots = []

    for _ in range(n):
        x_next = x - (f(x)/df(x))
        roots.append(x_next)
        x = x_next

    print(f"The root was found to be at {x} after {n} iterations")

    # Plotting the function and tangent lines
    x_vals = np.linspace(min(roots) - 1, max(roots) + 1, 100)
    y_vals = f(x_vals)  # Evaluate the function at x_vals
    tangent_lines = []

    for root in roots:
        tangent_lines.append(df(root) * (x_vals - root) + f(root))

    plt.plot(x_vals, y_vals, label='Function')
    for line in tangent_lines:
        plt.plot(x_vals, line, '--', alpha=0.5)
    plt.scatter(roots[-1], f(roots[-1]), color='red', label='Root')
    plt.xlabel('x')
    plt.ylabel('y')
    plt.title('Newton\'s Method')
    plt.legend()
    plt.grid(True)
    plt.show()

newtonMethod("x**3 - x - 1", "3*x**2 - 1", 1.5, 4)
```

OUTPUT:



```
(base) naganandana@Naganandanas-MacBook-Air-3 ~ % /Users/naganandana/anaconda3/bin/python /Users/nagana
The root was found to be at 1.3247179572447898 after 4 iterations
```

SOLUTION GRAPH:



Q2

Suppose you are managing a manufacturing plant, and you need to optimize the production process to minimize costs. The total cost C(x) of producing x units of a product can be modeled by the equation:

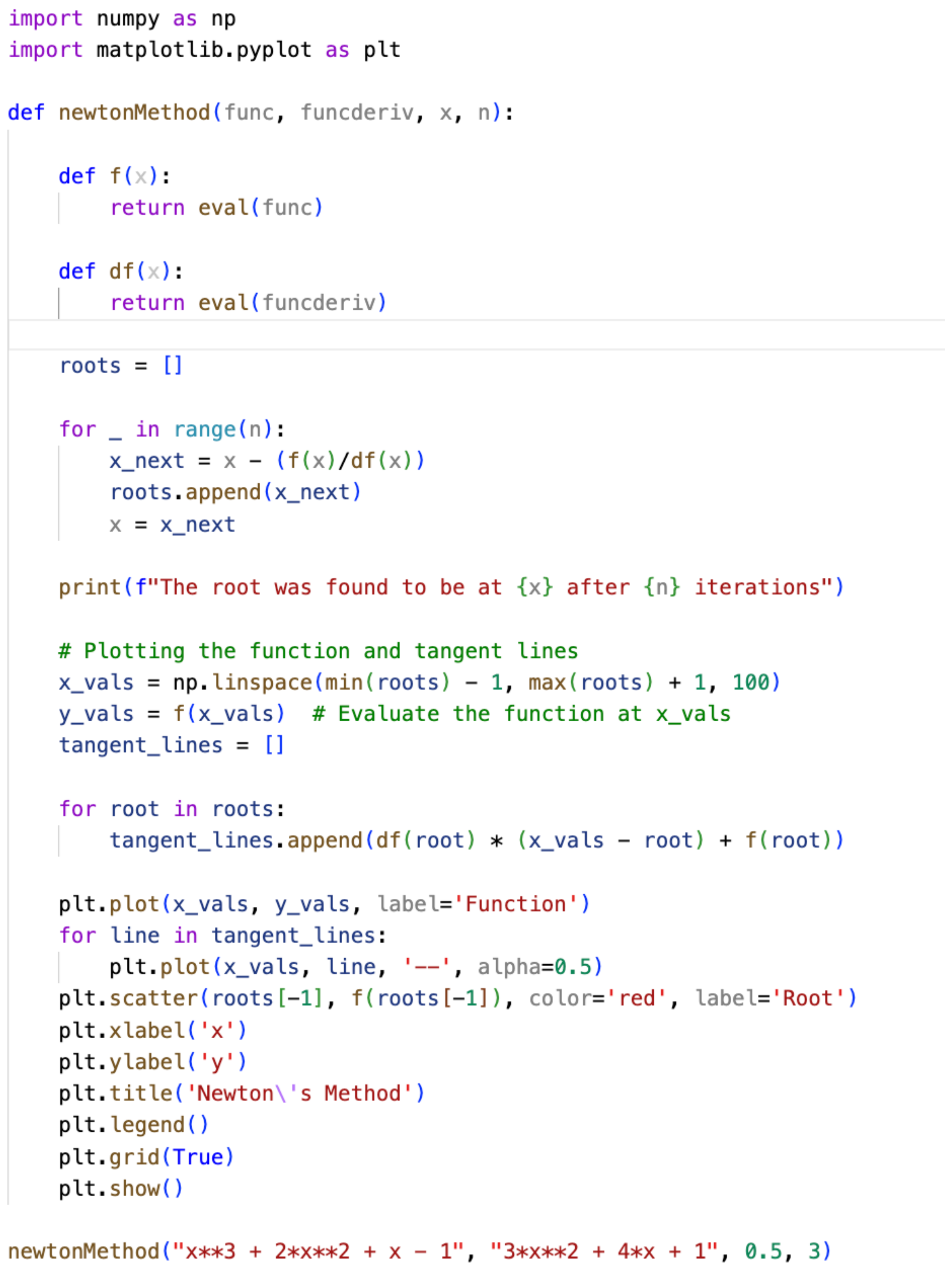
C(x) = x^3 + 2 * x^2 + x - 1

where:

x is the number of units of the product produced.

You want to find the production quantity x that minimizes the total cost C(x). Use Newton's method to find the production quantity x that minimizes the total cost, given that the initial guess for x is x0 = 0.5 units. Additionally, plot the total cost function C(x) and the tangent lines generated during each iteration of Newton's method to visualize the convergence to the solution.

CODE:

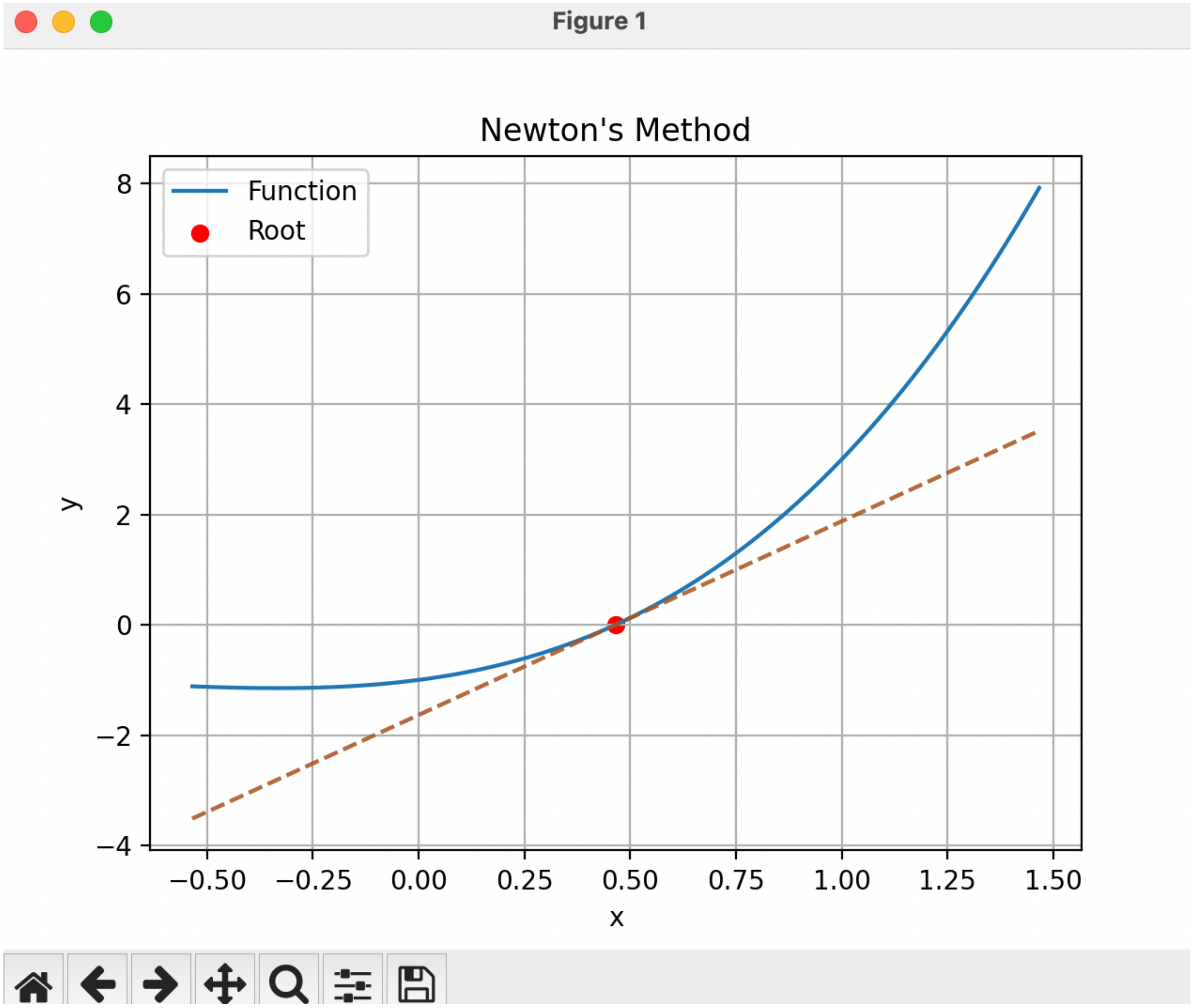


```python
import numpy as np
import matplotlib.pyplot as plt

def newtonMethod(func, funcderiv, x, n):

    def f(x):
        return eval(func)

    def df(x):
        return eval(funcderiv)

    roots = []

    for _ in range(n):
        x_next = x - (f(x)/df(x))
        roots.append(x_next)
        x = x_next

    print(f"The root was found to be at {x} after {n} iterations")

    # Plotting the function and tangent lines
    x_vals = np.linspace(min(roots) - 1, max(roots) + 1, 100)
    y_vals = f(x_vals)  # Evaluate the function at x_vals
    tangent_lines = []

    for root in roots:
        tangent_lines.append(df(root) * (x_vals - root) + f(root))

    plt.plot(x_vals, y_vals, label='Function')
    for line in tangent_lines:
        plt.plot(x_vals, line, '--', alpha=0.5)
    plt.scatter(roots[-1], f(roots[-1]), color='red', label='Root')
    plt.xlabel('x')
    plt.ylabel('y')
    plt.title('Newton\'s Method')
    plt.legend()
    plt.grid(True)
    plt.show()

newtonMethod("x**3 + 2*x**2 + x - 1", "3*x**2 + 4*x + 1", 0.5, 3)
```

OUTPUT:

```
(base) naganandana@Naganandanas-MacBook-Air-3 ~ % /Users/naganandana
The root was found to be at 0.4655712318780663 after 3 iterations
```

GRAPH:



GAUSS ELIMINATION METHOD WITH PARTIAL PIVOTING

Gaussian elimination with partial pivoting selects the pivot element as the largest absolute value in each column to minimize numerical errors during elimination. It reduces the risk of division by small numbers, ensuring more accurate solutions for systems of linear equations.

This method swaps rows strategically and performs elimination steps to transform the matrix into upper triangular form, facilitating easier back substitution for finding variable values.

PROCEDURE:

Program code:

1. Validate input matrices to ensure correctness
2. Initialize variables and augmented matrix
3. Perform partial pivoting to select pivot elements
4. Apply Gaussian elimination to transform matrix into upper triangular form
5. Solve for X using back substitution
6. Display solution vector X with corresponding variable labels

Procedure for Gaussian Elimination with Partial Pivoting:

1. Check if matrix A is square and matrix B has correct size
2. Create augmented matrix by concatenating A and B along columns
3. For each iteration i from 0 to n-1:
    - Perform partial pivoting to swap rows if necessary
    - Check for divide by zero error
    - Eliminate variables below the pivot element

4. Solve for X using back substitution:

   - Calculate Xm directly

   - For each row k from n-2 down to 0, calculate Xk

5. Print solution vector X with appropriate labels for each variable Xi

PROBLEMS:

Q1

Use Gauss Elimination method with partial pivoting to solve the given system of equations:

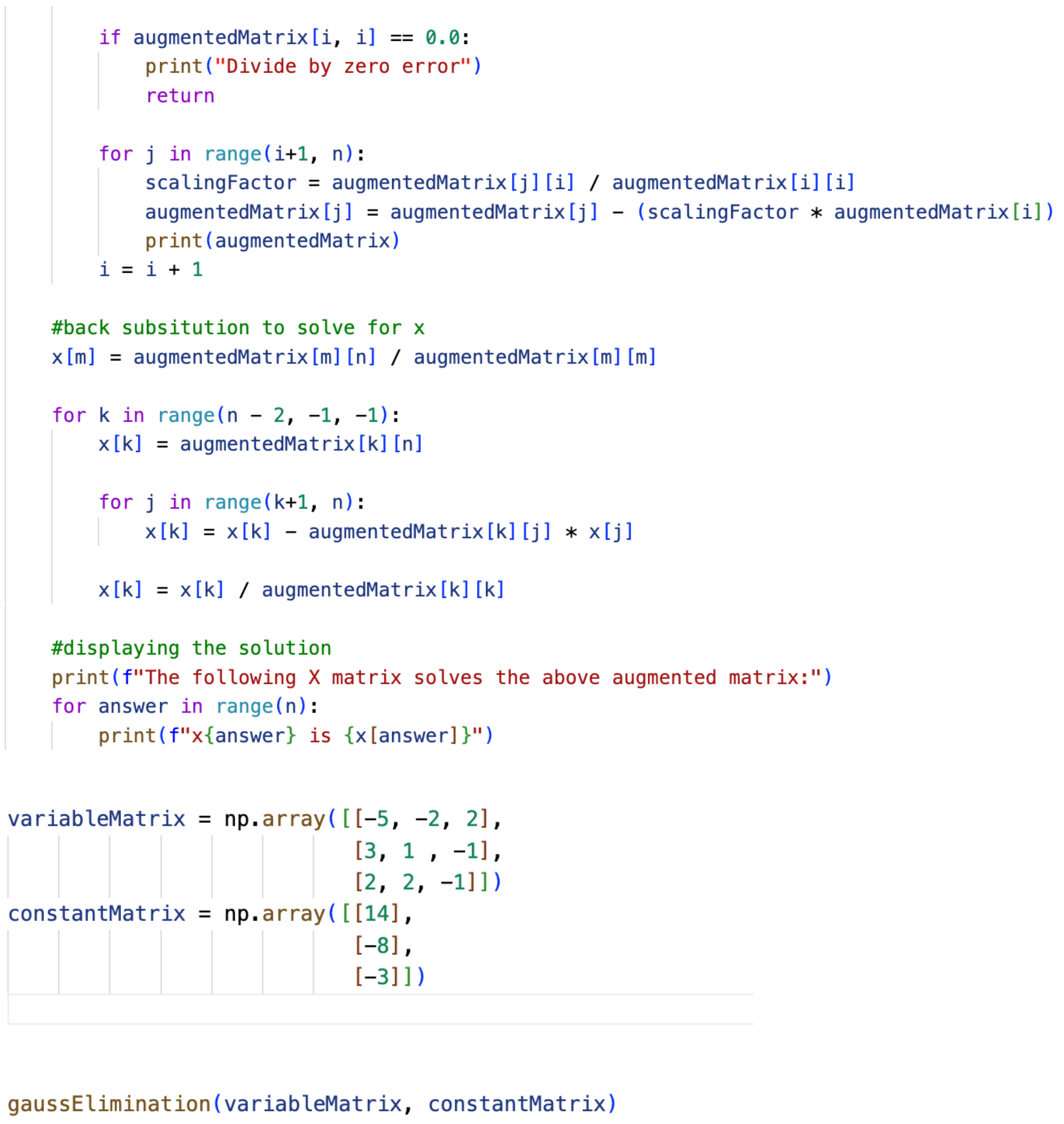
$-5x - 2y + 2z = 14$

$3x + y - z = -8$

$2x + 2y - z = -3$

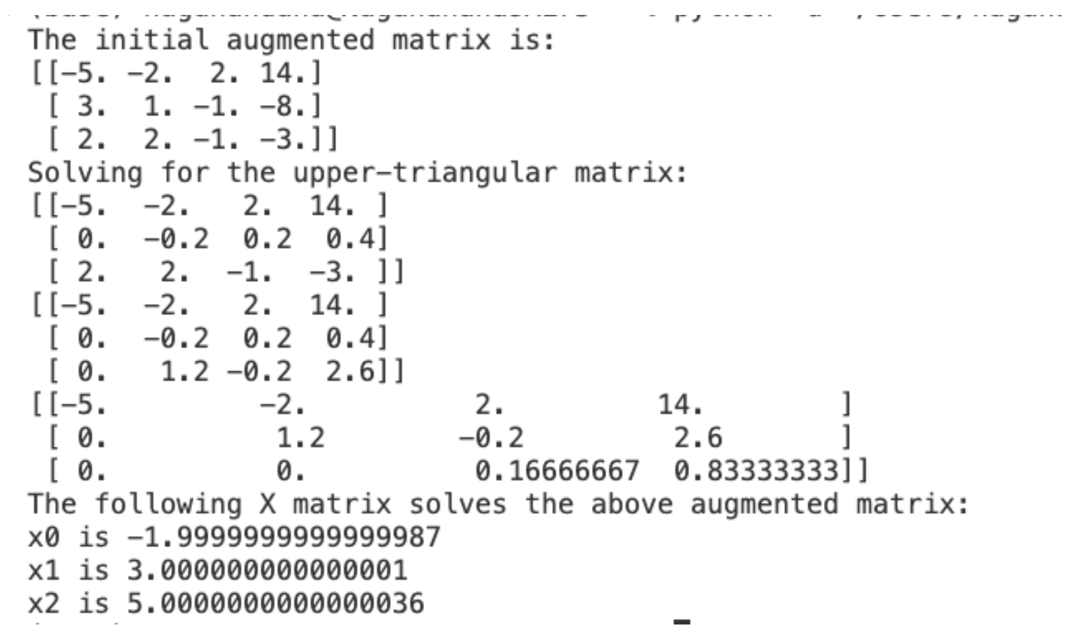
CODE:

```python
import numpy as np

def gaussElimination(aMatrix, bMatrix):

    #prevent future problems by adding contingencies (ie, check correctness of input)

    if aMatrix.shape[0] != aMatrix.shape[1]:
        print("Error: Matrix A is not square")
        return

    if bMatrix.shape[1] > 1 or bMatrix.shape[0] != aMatrix.shape[0]:
        print("Error: Matrix B is not correct size")
        return

    #initializing necessary variables
    n = len(bMatrix)
    m = n - 1
    i = 0
    x = np.zeros(n)
    newline = "\n"

    #create augmented matrix through Numpy's concatenate feature, and force data type to float
    augmentedMatrix = np.concatenate((aMatrix, bMatrix), axis=1, dtype=float)
    print(f"The initial augmented matrix is: {newline}{augmentedMatrix}")
    print("Solving for the upper-triangular matrix:")

    #applying gauss elimination with partial pivoting

    while i < n:

        #partial pivoting
        for p in range(i+1, n):
            if abs(augmentedMatrix[i, i]) < abs(augmentedMatrix[p, i]):
                augmentedMatrix[[p, i]] = augmentedMatrix[[i, p]]
```



```python
        if augmentedMatrix[i, i] == 0.0:
            print("Divide by zero error")
            return

        for j in range(i+1, n):
            scalingFactor = augmentedMatrix[j][i] / augmentedMatrix[i][i]
            augmentedMatrix[j] = augmentedMatrix[j] - (scalingFactor * augmentedMatrix[i])
            print(augmentedMatrix)
        i = i + 1

    #back subsitution to solve for x
    x[m] = augmentedMatrix[m][n] / augmentedMatrix[m][m]

    for k in range(n - 2, -1, -1):
        x[k] = augmentedMatrix[k][n]

        for j in range(k+1, n):
            x[k] = x[k] - augmentedMatrix[k][j] * x[j]

        x[k] = x[k] / augmentedMatrix[k][k]

    #displaying the solution
    print(f"The following X matrix solves the above augmented matrix:")
    for answer in range(n):
        print(f"x{answer} is {x[answer]}")


variableMatrix = np.array([[-5, -2, 2],
                            [3, 1 , -1],
                            [2, 2, -1]])
constantMatrix = np.array([[14],
                            [-8],
                            [-3]])


gaussElimination(variableMatrix, constantMatrix)
```

OUTPUT:



```
The initial augmented matrix is:
[[-5. -2.  2. 14.]
 [ 3.  1. -1. -8.]
 [ 2.  2. -1. -3.]]
Solving for the upper-triangular matrix:
[[-5.  -2.   2.  14. ]
 [ 0.  -0.2  0.2  0.4]
 [ 2.   2.  -1.  -3. ]]
[[-5.  -2.   2.  14. ]
 [ 0.  -0.2  0.2  0.4]
 [ 0.   1.2 -0.2  2.6]]
[[-5.         -2.          2.         14.        ]
 [ 0.          1.2        -0.2         2.6       ]
 [ 0.          0.          0.16666667  0.83333333]]
The following X matrix solves the above augmented matrix:
x0 is -1.9999999999999987
x1 is 3.000000000000001
x2 is 5.0000000000000036
```

Q2

The previous system of equations was easy to solve by hand. However, when given a large matrix, it becomes cumbersome to solve it without a program.

Solve the following matrixes:

```
A = [[1, 1, 1, 0, 0, 0, 0, 0, 0],
                          [0, 0, 0, 9, 3, 1, 0, 0, 0],
                          [9, 3, 1, 0, 0, 0, 0, 0, 0],
                          [0, 0, 0, 25, 5, 1, 0, 0, 0],
                          [0, 0, 0, 0, 0, 0, 25, 5, 1],
                          [0, 0, 0, 0, 0, 0, 64, 8, 1],
                          [6, 1, 0, -6, -1, 0, 0, 0, 0],
                          [0, 0, 0, 10, 1, 0, -10, -1, 0],
                          [2, 0, 0, 0, 0, 0, 0, 0, 0]]
```

```
B = [[2],
     [3],
     [3],
     [9],
     [9],
     [10],
     [0],
     [0],
     [0]]
```

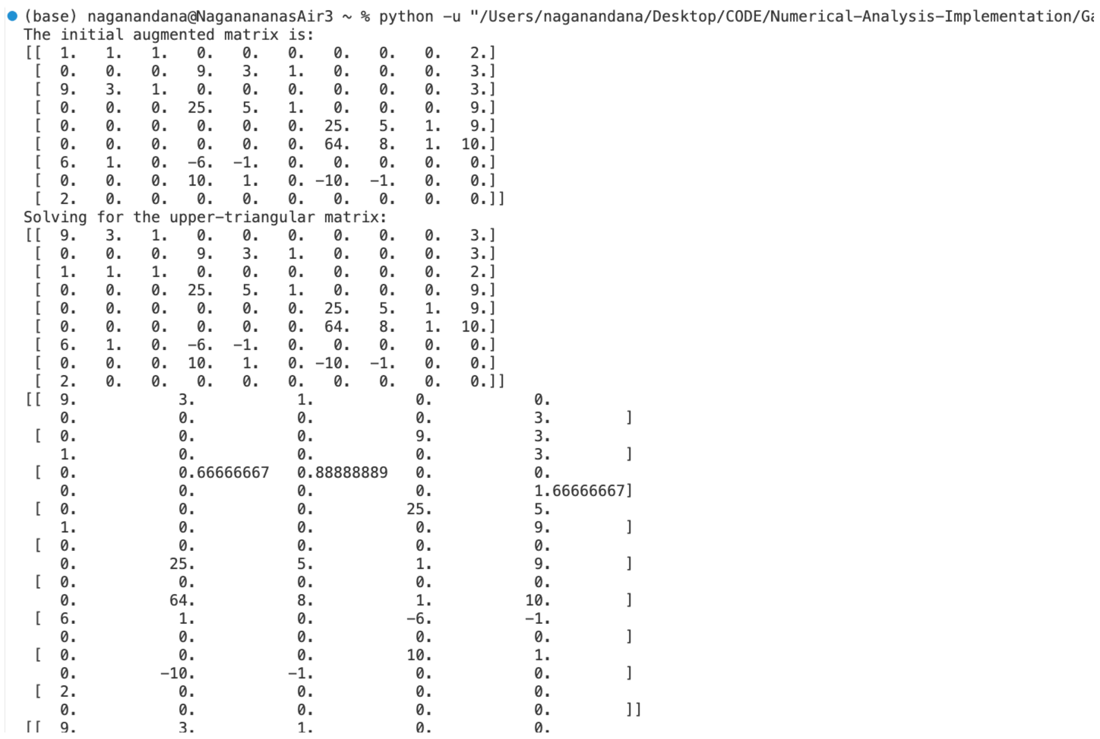
CODE:

```python
import numpy as np

def gaussElimination(aMatrix, bMatrix):

    #prevent future problems by adding contingencies (ie, check correctness of input)

    if aMatrix.shape[0] != aMatrix.shape[1]:
        print("Error: Matrix A is not square")
        return

    if bMatrix.shape[1] > 1 or bMatrix.shape[0] != aMatrix.shape[0]:
        print("Error: Matrix B is not correct size")
        return

    #initializing necessary variables
    n = len(bMatrix)
    m = n - 1
    i = 0
    x = np.zeros(n)
    newline = "\n"

    #create augmented matrix through Numpy's concatenate feature, and force data type to float
    augmentedMatrix = np.concatenate((aMatrix, bMatrix), axis=1, dtype=float)
    print(f"The initial augmented matrix is: {newline}{augmentedMatrix}")
    print("Solving for the upper-triangular matrix:")

    #applying gauss elimination with partial pivoting

    while i < n:

        #partial pivoting
        for p in range(i+1, n):
            if abs(augmentedMatrix[i, i]) < abs(augmentedMatrix[p, i]):
                augmentedMatrix[[p, i]] = augmentedMatrix[[i, p]]
        if augmentedMatrix[i, i] == 0.0:
            print("Divide by zero error")
            return

        for j in range(i+1, n):
            scalingFactor = augmentedMatrix[j][i] / augmentedMatrix[i][i]
            augmentedMatrix[j] = augmentedMatrix[j] - (scalingFactor * augmentedMatrix[i])
            print(augmentedMatrix)
        i = i + 1

    #back subsitution to solve for x
    x[m] = augmentedMatrix[m][n] / augmentedMatrix[m][m]

    for k in range(n - 2, -1, -1):
        x[k] = augmentedMatrix[k][n]

        for j in range(k+1, n):
            x[k] = x[k] - augmentedMatrix[k][j] * x[j]

        x[k] = x[k] / augmentedMatrix[k][k]

    #displaying the solution
    print(f"The following X matrix solves the above augmented matrix:")
    for answer in range(n):
        print(f"x{answer} is {x[answer]}")
```

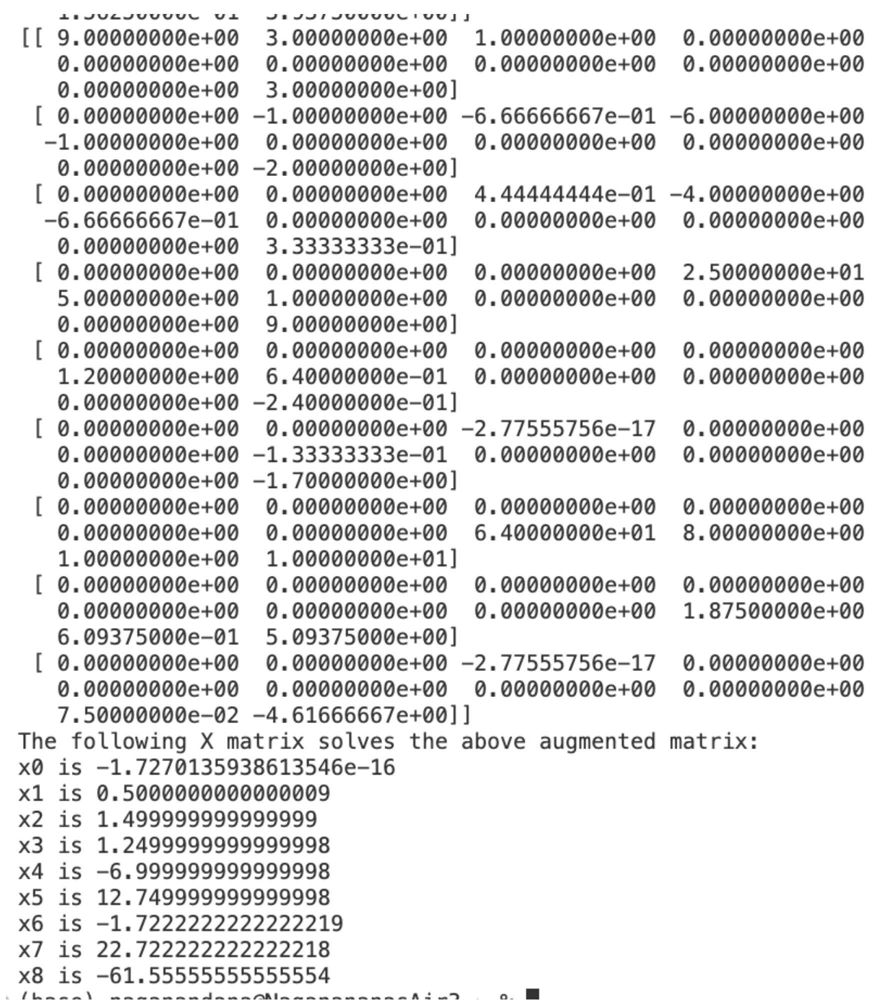
```
variableMatrix = np.array([[1, 1, 1, 0, 0, 0, 0, 0, 0],
                           [0, 0, 0, 9, 3, 1, 0, 0, 0],
                           [9, 3, 1, 0, 0, 0, 0, 0, 0],
                           [0, 0, 0, 25, 5, 1, 0, 0, 0],
                           [0, 0, 0, 0, 0, 0, 25, 5, 1],
                           [0, 0, 0, 0, 0, 0, 64, 8, 1],
                           [6, 1, 0, -6, -1, 0, 0, 0, 0],
                           [0, 0, 0, 10, 1, 0, -10, -1, 0],
                           [2, 0, 0, 0, 0, 0, 0, 0, 0]])

constantMatrix = np.array([[2],
                           [3],
                           [3],
                           [9],
                           [9],
                           [10],
                           [0],
                           [0],
                           [0]])
```

```
gaussElimination(variableMatrix, constantMatrix)
```

OUTPUT:



```
● (base) naganandana@NaganananasAir3 ~ % python -u "/Users/naganandana/Desktop/CODE/Numerical-Analysis-Implementation/Ga
 The initial augmented matrix is:
 [[  1.   1.   1.   0.   0.   0.   0.   0.   0.   2.]
  [  0.   0.   0.   9.   3.   1.   0.   0.   0.   3.]
  [  9.   3.   1.   0.   0.   0.   0.   0.   0.   3.]
  [  0.   0.   0.  25.   5.   1.   0.   0.   0.   9.]
  [  0.   0.   0.   0.   0.   0.  25.   5.   1.   9.]
  [  0.   0.   0.   0.   0.   0.  64.   8.   1.  10.]
  [  6.   1.   0.  -6.  -1.   0.   0.   0.   0.   0.]
  [  0.   0.   0.  10.   1.   0. -10.  -1.   0.   0.]
  [  2.   0.   0.   0.   0.   0.   0.   0.   0.   0.]]
 Solving for the upper-triangular matrix:
 [[  9.   3.   1.   0.   0.   0.   0.   0.   0.   3.]
  [  0.   0.   0.   9.   3.   1.   0.   0.   0.   3.]
  [  1.   1.   1.   0.   0.   0.   0.   0.   0.   2.]
  [  0.   0.   0.  25.   5.   1.   0.   0.   0.   9.]
  [  0.   0.   0.   0.   0.   0.  25.   5.   1.   9.]
  [  0.   0.   0.   0.   0.   0.  64.   8.   1.  10.]
  [  6.   1.   0.  -6.  -1.   0.   0.   0.   0.   0.]
  [  0.   0.   0.  10.   1.   0. -10.  -1.   0.   0.]
  [  2.   0.   0.   0.   0.   0.   0.   0.   0.   0.]]
 [[  9.          3.          1.          0.          0.
     0.          0.          0.          0.          3.        ]
  [  0.          0.          0.          9.          3.
     1.          0.          0.          0.          3.        ]
  [  0.          0.66666667  0.88888889  0.          0.
     0.          0.          0.          0.          1.66666667]
  [  0.          0.          0.         25.          5.
     1.          0.          0.          0.          9.        ]
  [  0.          0.          0.          0.          0.
     0.         25.          5.          1.          9.        ]
  [  0.          0.          0.          0.          0.
     0.         64.          8.          1.         10.        ]
  [  6.          1.          0.         -6.         -1.
     0.          0.          0.          0.          0.        ]
  [  0.          0.          0.         10.          1.
     0.        -10.         -1.          0.          0.        ]
  [  2.          0.          0.          0.          0.
     0.          0.          0.          0.          0.        ]]
 [[  9.          3.          1.          0.          0.
```



```
    1.56250000e-01  3.59375000e+00]]
 [[ 9.00000000e+00  3.00000000e+00  1.00000000e+00  0.00000000e+00
    0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
    0.00000000e+00  3.00000000e+00]
  [ 0.00000000e+00 -1.00000000e+00 -6.66666667e-01 -6.00000000e+00
   -1.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
    0.00000000e+00 -2.00000000e+00]
  [ 0.00000000e+00  0.00000000e+00  4.44444444e-01 -4.00000000e+00
   -6.66666667e-01  0.00000000e+00  0.00000000e+00  0.00000000e+00
    0.00000000e+00  3.33333333e-01]
  [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  2.50000000e+01
    5.00000000e+00  1.00000000e+00  0.00000000e+00  0.00000000e+00
    0.00000000e+00  9.00000000e+00]
  [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
    1.20000000e+00  6.40000000e-01  0.00000000e+00  0.00000000e+00
    0.00000000e+00 -2.40000000e-01]
  [ 0.00000000e+00  0.00000000e+00 -2.77555756e-17  0.00000000e+00
    0.00000000e+00 -1.33333333e-01  0.00000000e+00  0.00000000e+00
    0.00000000e+00 -1.70000000e+00]
  [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
    0.00000000e+00  0.00000000e+00  6.40000000e+01  8.00000000e+00
    1.00000000e+00  1.00000000e+01]
  [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
    0.00000000e+00  0.00000000e+00  0.00000000e+00  1.87500000e+00
    6.09375000e-01  5.09375000e+00]
  [ 0.00000000e+00  0.00000000e+00 -2.77555756e-17  0.00000000e+00
    0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00
    7.50000000e-02 -4.61666667e+00]]
 The following X matrix solves the above augmented matrix:
 x0 is -1.7270135938613546e-16
 x1 is 0.5000000000000009
 x2 is 1.499999999999999
 x3 is 1.2499999999999998
 x4 is -6.999999999999998
 x5 is 12.749999999999998
 x6 is -1.7222222222222219
 x7 is 22.722222222222218
 x8 is -61.55555555555554
 (base) naganandana@NaganananasAir3 ~ % █
```

VIDEO LINK – SCAN QR CODE: