INDIAN INSTITUTE OF TECHNOLOGY KANPUR

Department of Sustainable Energy Engineering

SEE-609A

Final Project Report

ON

Objective:- Develop a Newton-Raphson-based non-linear equation system solver. Use the Secant method for calculating Jacobian and freeze the Jacobian for several iterations (try variations of freezing for 10,20, 50,100 iterations and see the effect on speed and number of iterations required.). Also, you can test the number of equations and freezing time.

- Nishant Kumar (22129268)
- Ashish Thakur (231290603)
- Kunal Shivshankar Harinkhede (231290402)
- Yogender Singh Pal (231290021)

Overview

The development of a Newton-Raphson based nonlinear equation system solver, which can utilize the Secant method for calculating the Jacobian has been addressed in this project. The focus has been on evaluating the impact of freezing the Jacobian for a definite number of iterations. Solving nonlinear equation system is important in various real-life engineering scenarios, such as optimization problems, power system analysis, Heat transfer analysis, Thermodynamics analysis, etc.

The primary objective is to create a solver that can efficiently handle nonlinear systems of equations and explore the effect of freezing the Jacobian on convergence speed as well as the accuracy of the solution.

Solving non-linear equation system is essential for myriad reasons:

- Non-linear systems generally arise in real-world scenarios across various fields. So, the solution of these systems is essential for practical applications.
- Many optimization problems may involve evaluating the roots of non-linear equations. Also, designing efficient systems often requires solving the present non-linear equations to optimize parameters and achieve desired outcomes.
- In engineering, non-linear systems appear in areas such as power systems, control systems, structural mechanics, and fluid dynamics. Good solutions to these systems are crucial for designing trustworthy and effective systems.
- Development of efficient numerical methods for solving nonlinear systems upgrades computational efficiency. This is especially relevant for large-scale problems where rapid convergence can significantly reduce computational costs.
- In scientific research, solving non-linear equations is fundamental to modeling physical systems. It enables us to simulate and understand intricate systems, contributing to advancements in various scientific endeavors.
- Solution of the non-linear equations is foundational to the development of innovative technologies. It plays a crucial role in fields such as machine learning, image processing, and data analysis, where optimization and pattern recognition are key components.
- Understanding and solving non-linear systems is a fundamental aspect of mathematical education. Developing effective methods for solving such systems enhances educational resources and facilitates learning in applied mathematics and engineering. A fast and effective method of solving the system of non-linear equations actually given an edge for quick and efficient systems.

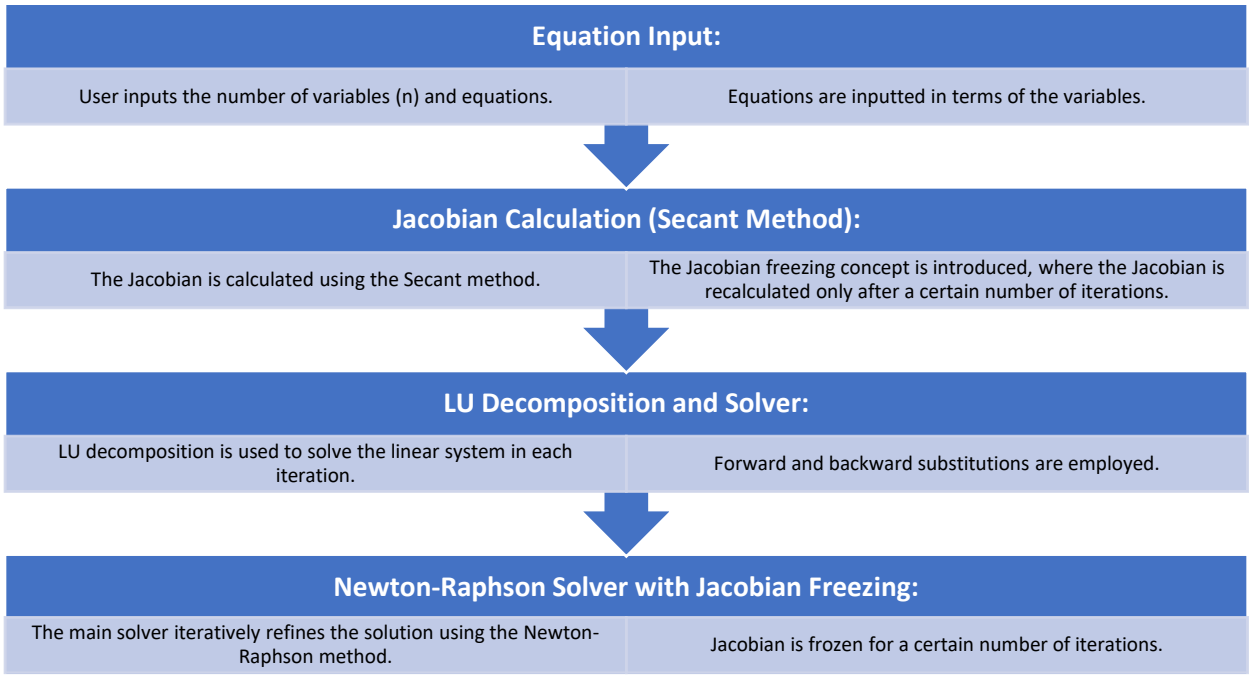
In summary, addressing the challenges associated with non-linear equation system is essential for advancing scientific understanding, optimizing processes, and solving real-world problems across diverse fields. The development of efficient numerical methods, as presented in the problem statement, contributes to the broader goals of enhancing computational techniques

and facilitating breakthroughs in various disciplines. In this current age of computers, the requirement of fast methods of solving the system of non-linear equations is eminent.

Methodology

The Newton-Raphson method aims to iteratively refine an initial guess of the solution to a system of equations. For the system of non-linear equations, the Jacobian matrix represents the system's sensitivity to changes in the variables. But, when the equation is subject to uncertainty, then Secant method is employed to approximate the Jacobian by finite differences. To enhance the speed of calculation frozen Jacobian approach involves calculating the Jacobian which periodically reduces computational overhead.

Implementation:



The program developed is a generic program for solving system of non-linear equations using Newton-Raphson method. This program takes input from the user for the number of variables and the number of equations. After taking the input, the program asks for initial guesses as per number of variables. After it, the program automatically calculates the Jacobian and freezes for a certain number of iterations. If the solution is not converged, the Jacobian is updated dynamically, and solution is obtained. When the solution is generated, then the number of iterations, freeze number and results are printed.

Results and Discussions

Generic program for Newton-Raphson base nonlinear equation system solver:

```
import numpy as np

def equations_and_variable_input():
    n_val = int(input("Enter the total number of variables (n): "))

    equation_mat = []
    variables = [f'x{i}' for i in range(1, n_val + 1)]

    for p in range(n_val):
        equation = input(f"Enter equation {p + 1} in terms of {',
'.join(variables)}: ")
        equation_mat.append(equation)

    return equation_mat, variables, n_val

def eval_eq(equation, variables, values):
    for var, value in zip(variables, values):
        equation = equation.replace(var, str(value))
    return equation

def jacobian_using_secant(equations, variables, var0, h=1e-06):
    n_val = len(variables)
    identity_matrix = np.eye(n_val)
    jacobian = np.zeros((n_val, n_val))

    for i in range(n_val):
        var_h = var0 + h * identity_matrix[i]

        equations_at_varh = [eval(eval_eq(eq, variables, var_h)) for
eq in equations]
        equations_at_var0 = [eval(eval_eq(eq, variables, var0)) for eq
in equations]

        jacobian[:, i] = (np.array(equations_at_varh) -
np.array(equations_at_var0)) / h

    return jacobian

equations, variables, n_val = equations_and_variable_input()

var0 = [float(x) for x in input("Enter initial guess (comma-
```

```
separated): ").split(",")]

jacobian_mat = jacobian_using_secant(equations, variables, var0)
print("Jacobian Matrix:\n", jacobian_mat)

def lu_decomposition(A):
    n = len(A)
    L = np.zeros((n, n))
    U = np.zeros((n, n))

    for i in range(n):
        L[i][i] = 1.0
        for j in range(i, n):
            U[i][j] = A[i][j] - sum(L[i][k] * U[k][j] for k in
range(i))
        for j in range(i + 1, n):
            L[j][i] = (A[j][i] - sum(L[j][k] * U[k][i] for k in
range(i))) / U[i][i]

    return L, U

def lu_solver(L, U, b):
    n = len(b)
    # Solve Ly = b
    y = np.zeros(n)
    for i in range(n):
        y[i] = b[i] - sum(L[i][j] * y[j] for j in range(i))

    # Solve Ux = y
    x = np.zeros(n)
    for i in range(n - 1, -1, -1):
        x[i] = (y[i] - sum(U[i][j] * x[j] for j in range(i + 1, n))) /
U[i][i]

    return x



def jacobian_freezing_solver(F, J, var0, tol=1e-6, max_iter=1000000,
freez=30):
    x = var0
    B = jacobian_mat
    for iteration in range(max_iter):
```

```
        if iteration < freez:
            F_x = np.array([eval(eval_eq(eq, variables, x)) for eq in
equations])
            delta_x = -lu_solver(*lu_decomposition(B), F_x)
            x_next = x + delta_x
            F_x_next = np.array([eval(eval_eq(eq, variables, x_next))
for eq in equations])
            y = F_x_next - F_x
        else:
            B = jacobian_using_secant(equations, variables, x)
            F_x = np.array([eval(eval_eq(eq, variables, x)) for eq in
equations])
            delta_x = -lu_solver(*lu_decomposition(B), F_x)
            x_next = x + delta_x


        x = x_next
        if np.linalg.norm(delta_x) < tol:
            return x, iteration + 1
    return x, max_iter



freezing_iterations = [ 10, 20, 50, 100]
tolerance_values = [1e-4, 1e-6, 1e-8]

for freezing_iter in freezing_iterations:
    for tol_value in tolerance_values:
        solution, iterations = jacobian_freezing_solver(equations,
jacobian_mat, var0, tol=tol_value, freez=freezing_iter)
        print(f"\nFreezing Iterations: {freezing_iter}, Tolerance:
{tol_value}")
        print("Solution:", solution)
        print("Iterations:", iterations)
```

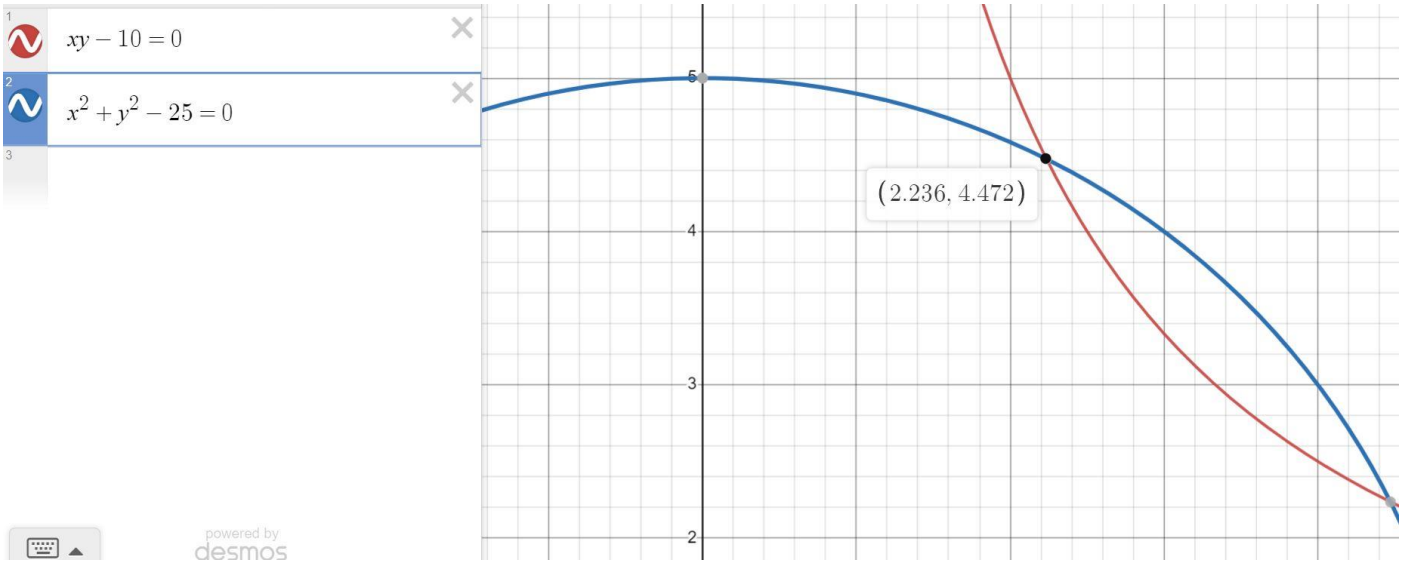
Test Cases:- Now, we'll take three test cases where we'll compare the solution using the method used in the program with analytical/exact solution.

We'll use following set of non-linear equations:

1. $$x_1 x_2 - 10 = 0$$
$$x_1^2 + x_2^2 - 25 = 0$$

Initial guess: (2,4)

Exact solution: (2.236, 4.472)

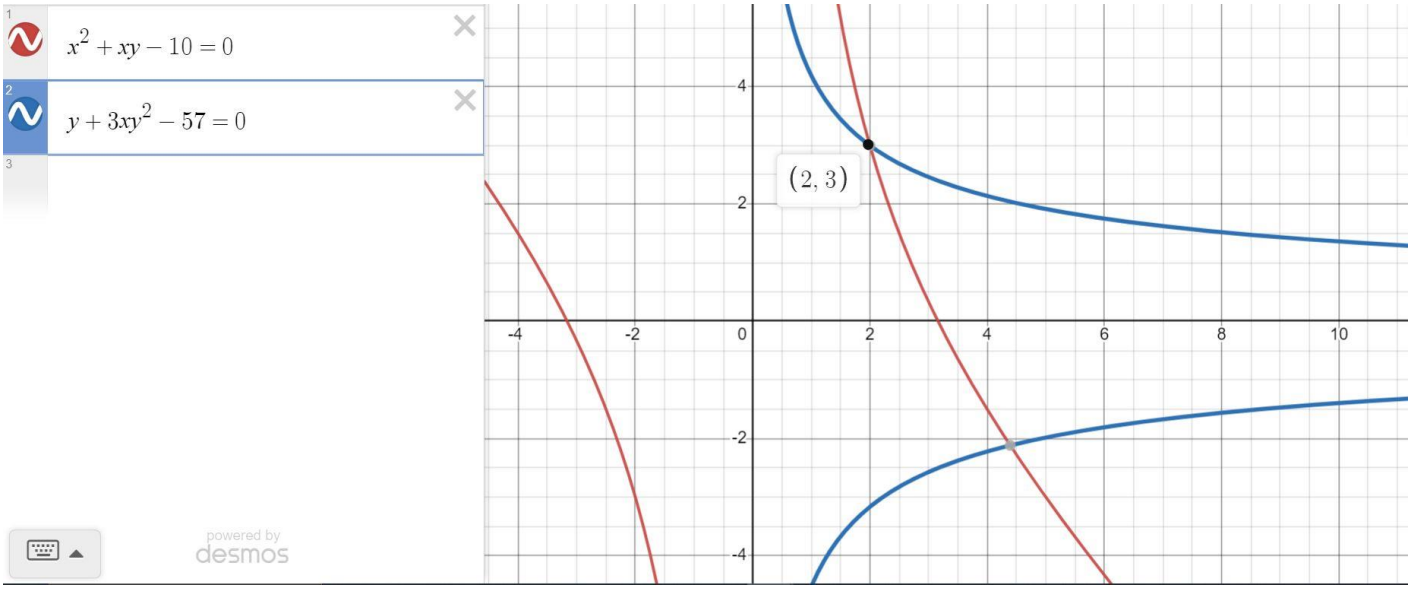


2. $$x_1^2 + x_1 x_2 - 10 = 0$$

$$x_2 + 3x_1 x_2^2 - 57 = 0$$

Initial guess: (1.5,3.5)

Exact solution: (2,3)

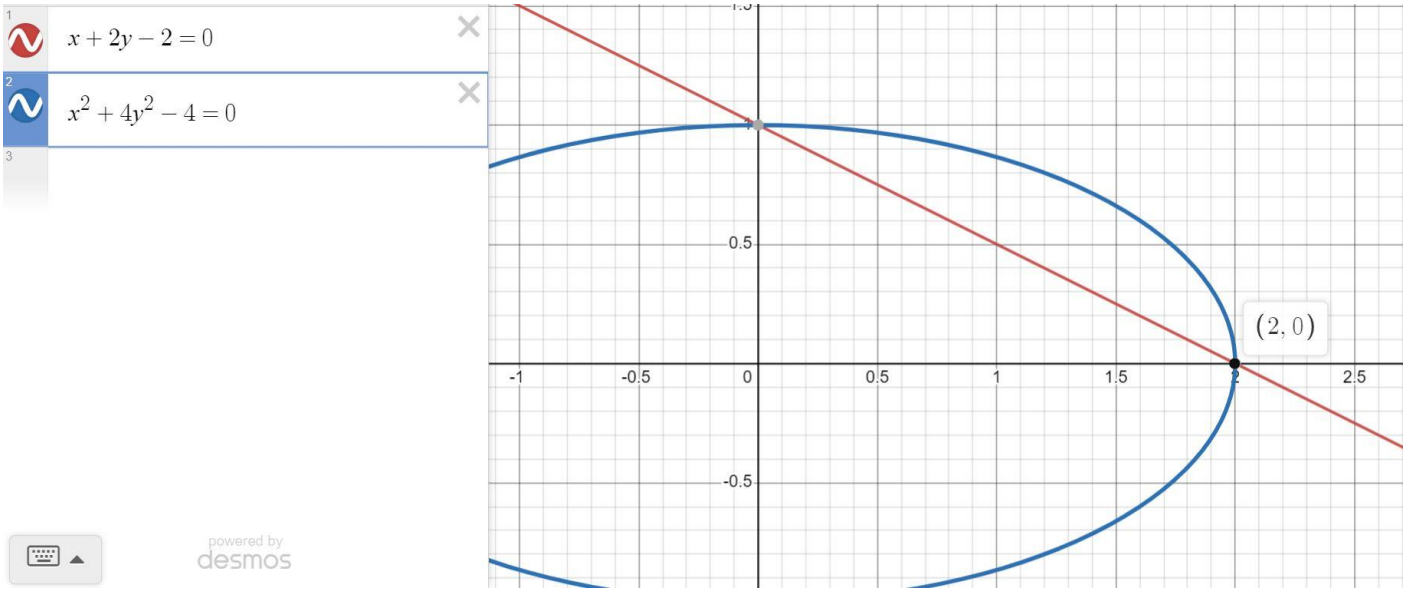


3. $$x_1 + 2x_2 - 2 = 0$$

$$x_1^2 + 4x_2^2 - 4 = 0$$

Initial guess: (1.5,0)

Exact solution: (2,0)



Now, analyzing the solutions given by the solver:

→ <u>For first set of equations:</u>

The given input is:

```
Enter the total number of variables (n): 2
Enter equation 1 in terms of x1, x2: x1*x2 - 10
Enter equation 2 in terms of x1, x2: x1**2 + x2**2 - 25
Enter initial guess (comma-separated): 2,4
```

The Jacobian Matrix comes out as:

```
Jacobian Matrix:
 [[4.       2.      ]
 [4.000001 8.000001]]
```

After freezing Jacobian at various iterations, the output comes out as:

Freezing Iterations: 10, Tolerance: 0.0001

Solution: [2.23607075 4.47214151]

Iterations: 5

Freezing Iterations: 10, Tolerance: 1e-06

Solution: [2.23606802 4.47213603]

Iterations: 7

Freezing Iterations: 10, Tolerance: 1e-08

Solution: [2.23606798 4.47213595]

Iterations: 10

Freezing Iterations: 20, Tolerance: 0.0001

Solution: [2.23607075 4.47214151]

Iterations: 5

Freezing Iterations: 20, Tolerance: 1e-06

Solution: [2.23606802 4.47213603]

Iterations: 7

Freezing Iterations: 20, Tolerance: 1e-08

Solution: [2.23606798 4.47213595]

Iterations: 10

Freezing Iterations: 50, Tolerance: 0.0001

Solution: [2.23607075 4.47214151]

Iterations: 5

Freezing Iterations: 50, Tolerance: 1e-06

Solution: [2.23606802 4.47213603]

Iterations: 7

Freezing Iterations: 50, Tolerance: 1e-08

Solution: [2.23606798 4.47213595]

Iterations: 10

Freezing Iterations: 100, Tolerance: 0.0001

Solution: [2.23607075 4.47214151]

Iterations: 5

Freezing Iterations: 100, Tolerance: 1e-06

Solution: [2.23606802 4.47213603]

Iterations: 7

Freezing Iterations: 100, Tolerance: 1e-08

Solution: [2.23606798 4.47213595]

Iterations: 10

→ For second set of equations:

The given input is:

```
Enter the total number of variables (n): 2
Enter equation 1 in terms of x1, x2: x1**2 + x1*x2 - 10
Enter equation 2 in terms of x1, x2: x2 + 3*x1*x2**2 - 57
Enter initial guess (comma-separated): 1.5,3.5
```

The Jacobian matrix comes out as:

```
Jacobian Matrix:
 [[ 6.500001    1.5       ]
 [36.74999999 32.5000045 ]]
```

After freezing Jacobian at various iterations, the output comes out as:

Freezing Iterations: 10, Tolerance: 0.0001

Solution: [2.00000522 2.99999375]

Iterations: 7

Freezing Iterations: 10, Tolerance: 1e-06

Solution: [2.00000005 3.00000001]

Iterations: 10

Freezing Iterations: 10, Tolerance: 1e-08

Solution: [2. 3.]

Iterations: 12

Freezing Iterations: 20, Tolerance: 0.0001

Solution: [2.00000522 2.99999375]

Iterations: 7

Freezing Iterations: 20, Tolerance: 1e-06

Solution: [2.00000005 3.00000001]

Iterations: 10

Freezing Iterations: 20, Tolerance: 1e-08

Solution: [2. 3.]

Iterations: 13

Freezing Iterations: 50, Tolerance: 0.0001

Solution: [2.00000522 2.99999375]

Iterations: 7

Freezing Iterations: 50, Tolerance: 1e-06

Solution: [2.00000005 3.00000001]

Iterations: 10

Freezing Iterations: 50, Tolerance: 1e-08

Solution: [2. 3.]

Iterations: 13

Freezing Iterations: 100, Tolerance: 0.0001

Solution: [2.00000522 2.99999375]

Iterations: 7

Freezing Iterations: 100, Tolerance: 1e-06

Solution: [2.00000005 3.00000001]

Iterations: 10

Freezing Iterations: 100, Tolerance: 1e-08

Solution: [2. 3.]

Iterations: 13

→ For third set of equations:

The given input is:

```
Enter the total number of variables (n): 2
Enter equation 1 in terms of x1, x2: x1 + 2*x2 - 2
Enter equation 2 in terms of x1, x2: x1**2 + 4*x2**2 - 4
Enter initial guess (comma-separated): 1.5,0
```

The Jacobian matrix comes out as:

```
Jacobian Matrix:
 [[1.00000000e+00 2.00000000e+00]
 [3.00000100e+00 3.99991151e-06]]
```

After freezing Jacobian at various iterations, the output comes out as:

Freezing Iterations: 10, Tolerance: 0.0001

Solution: [2.00001192e+00 -5.95866464e-06]

Iterations: 9

Freezing Iterations: 10, Tolerance: 1e-06

Solution: [2.00000000e+00 2.86362935e-17]

Iterations: 12

Freezing Iterations: 10, Tolerance: 1e-08

Solution: [2.00000000e+00 2.86362935e-17]

Iterations: 12

Freezing Iterations: 20, Tolerance: 0.0001

Solution: [2.00001192e+00 -5.95866464e-06]

Iterations: 9

Freezing Iterations: 20, Tolerance: 1e-06

Solution: [2.00000015e+00 -7.35635040e-08]

Iterations: 13

Freezing Iterations: 20, Tolerance: 1e-08

Solution: [2.00000000e+00 -9.08196206e-10]

Iterations: 17

Freezing Iterations: 50, Tolerance: 0.0001

Solution: [2.00001192e+00 -5.95866464e-06]

Iterations: 9

Freezing Iterations: 50, Tolerance: 1e-06

Solution: [2.00000015e+00 -7.35635040e-08]

Iterations: 13

Freezing Iterations: 50, Tolerance: 1e-08

Solution: [2.00000000e+00 -9.08196206e-10]

Iterations: 17

Freezing Iterations: 100, Tolerance: 0.0001

Solution: [2.00001192e+00 -5.95866464e-06]

Iterations: 9

Freezing Iterations: 100, Tolerance: 1e-06

Solution: [2.00000015e+00 -7.35635040e-08]

Iterations: 13

Freezing Iterations: 100, Tolerance: 1e-08

Solution: [2.00000000e+00 -9.08196206e-10]

Iterations: 17

- After comparing, approximate solutions (using method used in our program) are close to exact solutions.

Conclusion

The program developed calculates the solution of inputs given to the program. Jacobian has been calculated using the Secant method. The Jacobian matrix has been frozen for a certain number of iterations. In case of not getting the converged solution, the Jacobian matrix has been updated dynamically, automatically in the program so that the solution can be found. So, the objective of the task has largely been met using the developed program except for the limitation when the initial guess is chosen badly. This special case of choice of bad initial guess makes the program not provide a solution for the Jacobian Freezing condition.

However, it must be emphasized that the code can be further improved to take inputs for choices for the user to determine if the program should be run with Jacobian Freezing method or dynamically updated Jacobian method. This will add further flexibility to the user.

References

1. Lecture Slides of Dr. Lalit M. Pant for the course SEE609A: Mathematics and Computational Methods for Engineers(2023-24-I)
2. CHAPRA, Steven, and CANALE, Raymond. NUMERICAL METHODS for ENGINEERS, KUWAITical Guide. United Kingdom, McGraw-Hill Education, 2016.
3. www.desmos.com/calculator.
4. www.w3schools.com