Lab Manual

for

CSE 2206

Numerical Methods Lab

Credit: 1.5

Contact Hours: 1.5 Hours Per Week

Department of Computer Science and Engineering

Varendra University, Rajshahi, Bangladesh

VARENDRA UNIVERSITY

Department of Computer Science and Engineering

CSE 124

Numerical Methods Lab

Instructor’s Information	
Name(s)	Golam Shahriar Saif, Md. Rakibul Islam, Adrita Alam, Md. Farhan Tanvir Nasim
Email	Shahriar489@gmail.com , rakibul@vu.edu.bd ,adritaalam.prima@gmail.com, farhan.ruet37@gmail.com

Student Information	
Student ID	
Student Name	
Semester	
Section	
Batch	

INDEX

INSTRUCTIONS FOR LABORATORY

1. Students should sign in the log registers as soon as they enter in the lab and strictly observe the lab timings.
2. Students have to strictly follow the written and verbal instructions given by the teacher. If they do not understand the instructions, the handouts and the procedures, they can ask the instructor or teacher.
3. Students should have to accompanied by their laboratory partner and the instructors all the time.
4. It is mandatory for the students to come to lab in a proper dress and have to wear their ID cards.
5. Mobile phones should be switched off in the lab and the bags should keep in proper place.
6. Students should keep the lab clean all times, no food and drinks are allowed inside the lab.
7. Intentional misconduct will lead to expulsion from the lab.
8. Students should not handle any equipment without reading the safety instructions. They should read the handout and procedures in the Lab Manual before starting the equipment.
9. Students should do their writing, setup and a careful circuit checkout before applying power.
10. They should not make circuit changes or perform any writing when power is on.
11. Student should avoid contact with energized electrical circuits.
12. 11. Students should not insert connectors forcefully into the sockets.
13. Students should immediately report dangerous or exceptional conditions to the Lab instructors. Equipment that is not working as expected, wires or connectors are broken, the equipment that smells or smokes. If a student is not sure what the problem is or what is going on they should switch off the Emergency Shutdown.
14. Students should never use the damage instruments, wires or connectors. They should handover the parts to the Lab Instructor.
15. After completion of Experiment, students should return the equipment’s to lab staff. They are not allowed to take any item from the lab without permission.

16. Observation book and lab record should be carried to each lab. Readings of current lab experiment are to be entered in Observation book and previous lab experiment should be written in Lab record book. Both the books should be corrected by the faculty in each lab.

17. Handling of Semiconductor Components: Sensitive electronic circuits and electronic components have to be handled with great care. The inappropriate handling of electronic component can damage or destroy the devices. The devices can be destroyed by driving to high currents through the device, by overheating the device, by mixing up the polarity, or by electrostatic discharge (ESD). Therefore, always handle the electronic devices as indicated by the handout, the specifications in the data sheet or other documentation.

Varendra University

COURSE SYLLABUS

1.	**Faculty**	Faculty of Science and Engineering
2.	**Department**	Department of CSE
3.	**Program**	B.Sc. in Computer Science and Engineering
4.	**Name of Course**	Numerical Methods Lab
5.	**Course Code**	CSE 2206
6.	**Semester and Year**	Spring 2025
7.	**Pre - requisites**	None
8.	**Status**	Core Course
9.	**Credit Hours**	1.5
10.	**Section**	ALL
11.	**Class Hours**	1.5 Hours
12.	**Class Location**	
13.	**Names of Academic Instructors**	Golam Shahriar Saif, Md. Rakibul Islam, Adrita Alam, Md. Farhan Tanvir Nasim
14.	**Contact**	
15.	**Office**	
16.	**Counseling Hours**	

17.	**Text Books**	1. Steven C. Chapra: Numerical Methods for Engineers (McGraw-Hill) 2. S. S. Sarty: Introductory Methods of Numerical Analysis
18.	**Course Description**	In this course students will have to implement the theories and concepts learned in **CSE 2205**.
19.	**Course Objectives**	The objectives are: 1. To implement basic root-finding methods, interpolation, regression analysis, and numerical differentiation and integration. 2. To equip students with the ability to apply numerical methods for problem-solving. 3. To provide laboratory exercises that help students explore various numerical techniques. 4. To apply these techniques in solving real-world problems.
21.	**Learning Outcomes**	Here are three learning outcomes based on the given CLOs: 1. To explain how to determine the error between actual and estimated data. 2. To analyze and demonstrate the use of Newton-Raphson and iteration methods for root finding. 3. To compare and contrast the results of Newton’s and Gauss’s interpolation formulas.
22.	**Teaching Methods**	Lecture

23.	**Topic Outline**				
	Class	**Topics**	**Class**	**Reading Reference**	**Activities**
	1	Introduction to Python and Error Analysis	2		Implementation, Question Answer
	2	Implementation of bisection method	1		Implementation, Question Answer
	3	Implementation of false position method	1		Implementation, Question Answer
	4	Implementation of Newton Raphson method.	2		Implementation, Question Answer
	5	Implementation of Gaussian Elimination.	1		Implementation, Question Answer
	6	Implementation of LU decomposition.	1		Implementation, Question Answer
	7	Implementation of Linear regression method of function approximation technique.	1		Implementation, Question Answer
	8	Implementation of Lagrange interpolation.	1		Implementation, Question Answer

	9	Implementation of curve fitting methods	1		Implementation, Question Answer
	10	Implementation of differentiation and integration rules.	1		Problem Solving, Question Answer
	11	Implementing Numerical Solution Algorithms for Differential Equations (ODEs and PDEs).	1		Implementation, Question Answer

24.	**Assessment Methods**	<table><tr><th>Assessment Type</th><th>Marks</th></tr><tr><td>Attendance</td><td>10%</td></tr><tr><td>Class Performance and Regular Assessment</td><td>10%</td></tr><tr><td>Lab Report</td><td>10%</td></tr><tr><td>Quiz</td><td>30%</td></tr><tr><td>Lab Viva</td><td>10%</td></tr><tr><td>Lab Final</td><td>30%</td></tr><tr><td>**Total**</td><td>**100%**</td></tr></table>
25.	**Grading Policy**	<table><tr><th>Numerical Grade</th><th>Letter Grade</th><th>Grade Point</th></tr><tr><td>80% and above</td><td>A+</td><td>4</td></tr><tr><td>75% to less than 80%</td><td>A</td><td>3.75</td></tr><tr><td>70% to less than 75%</td><td>A-</td><td>3.50</td></tr><tr><td>65% to less than 70%</td><td>B+</td><td>3.25</td></tr><tr><td>60% to less than 65%</td><td>B</td><td>3.00</td></tr><tr><td>55% to less than 60%</td><td>B-</td><td>2.75</td></tr><tr><td>50% to less than 55%</td><td>C+</td><td>2.50</td></tr><tr><td>45% to less than 50%</td><td>C</td><td>2.25</td></tr><tr><td>40% to less than 45%</td><td>D</td><td>2.00</td></tr><tr><td>Less than 40%</td><td>F</td><td>0.00</td></tr></table>
26.	**Additional Course Policies**	**1. Lab Reports** Report on previous Experiment must be submitted before the beginning of new experiment. A bonus may be obtained if a student submits a neat,

		clean and complete lab report. **2. Examination** There will be a lab exam at the end of the semester that will be closed book. **3. Unfair means policy** In case of copying/plagiarism in any of the assessments, the students involved will receive zero marks. Zero tolerance will be shown in this regard. In case of severe offences, actions will be taken as per university rule. **4. Counseling** Students are expected to follow the counseling hours posted. In case of emergency/unavoidable situations, students can e-mail me to make an appointment. Students are regularly advised to check the course materials. **5. Policy for Absence in Class/Exam** If a student is absent in the class for anything other than medical reasons, he or she will not receive attendance. If a student misses a class for genuine medical reasons, he or she must apply with the supporting documents (prescription/medical report). He or she will then have to follow the instructions given by the instructions given by the instructor for makeup. In case of absence in the mid/final exam for medical grounds, the student must also get his or her application forwarded by the head of the department before a make-up exam can be taken. It is recommended that the students inform the instructor beforehand through mail if they feel that they will miss a class or evaluation due to medical reasons.
27.	**Additional Information**	a. Academic Calendar Spring 2025 b. Academic information and policies. c. Grading and performance Evaluation. d. Proctorial Rules.

Experiment No.: 01
Experiment Name: Introduction to Python and Error Analysis.

Theory:
Python is a versatile, high-level programming language widely used in data science, engineering, and numerical computing. It provides a simple syntax and powerful libraries such as NumPy, SciPy, and Matplotlib that make it ideal for solving numerical problems.

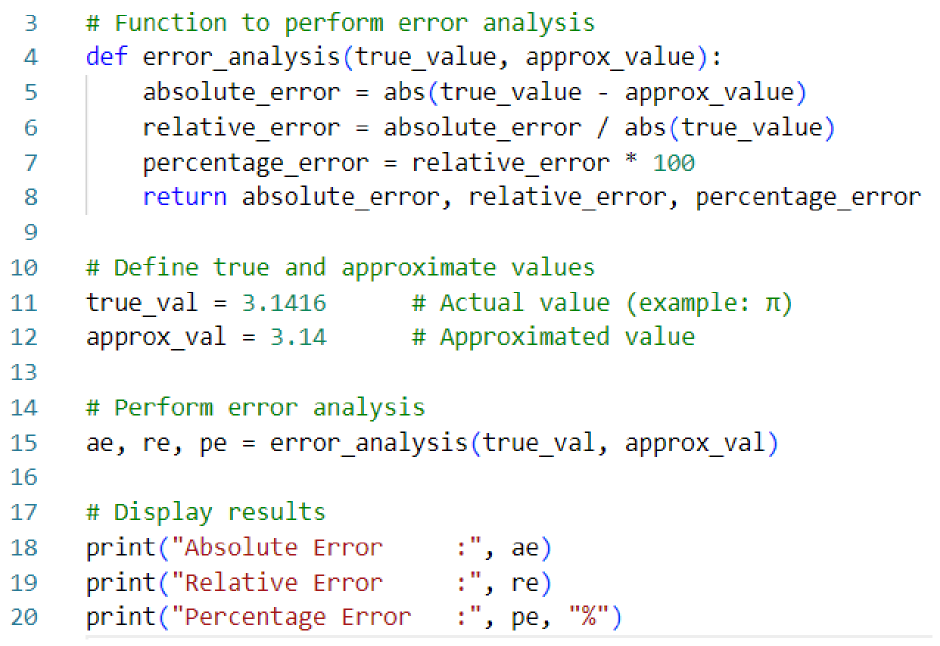
In numerical methods, error analysis plays a crucial role in understanding the accuracy and reliability of computed results. Since many problems are solved approximately using computers, it is important to quantify how close these approximate results are to the exact values.

Types of Errors:

- Absolute Error (AE):
  The difference between the true value and the approximate value.
  AE = | True Value – Approximate Value |
- Relative Error (RE):
  Measures the error in relation to the size of the true value.
  $$RE = \frac{AE}{|\, True\ Value\, |}$$
- Percentage Error (PE):
  Represents the relative error as a percentage.
  PE = RE × 100

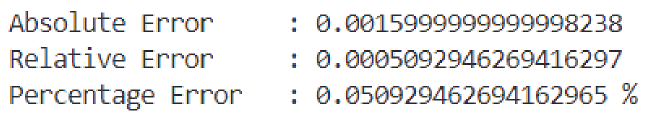
These measures help in evaluating how accurate an approximation is and guide improvements in numerical techniques.

Program 1: Programming Code



```python
# Function to perform error analysis
def error_analysis(true_value, approx_value):
    absolute_error = abs(true_value - approx_value)
    relative_error = absolute_error / abs(true_value)
    percentage_error = relative_error * 100
    return absolute_error, relative_error, percentage_error

# Define true and approximate values
true_val = 3.1416      # Actual value (example: π)
approx_val = 3.14      # Approximated value

# Perform error analysis
ae, re, pe = error_analysis(true_val, approx_val)

# Display results
print("Absolute Error     :", ae)
print("Relative Error     :", re)
print("Percentage Error   :", pe, "%")
```

Output:



```
Absolute Error     : 0.0015999999999998238
Relative Error     : 0.0005092946269416297
Percentage Error   : 0.050929462694162965 %
```

Discussion & Conclusion:

In this lab, we learned how to use Python to measure how different an estimated value is from the actual value. We used three types of error: absolute, relative, and percentage. These help us see how accurate our answer is.

This lab gave us a good starting point in both Python programming and error analysis. It showed us how important it is to check the accuracy of our answers when solving real-world problems.

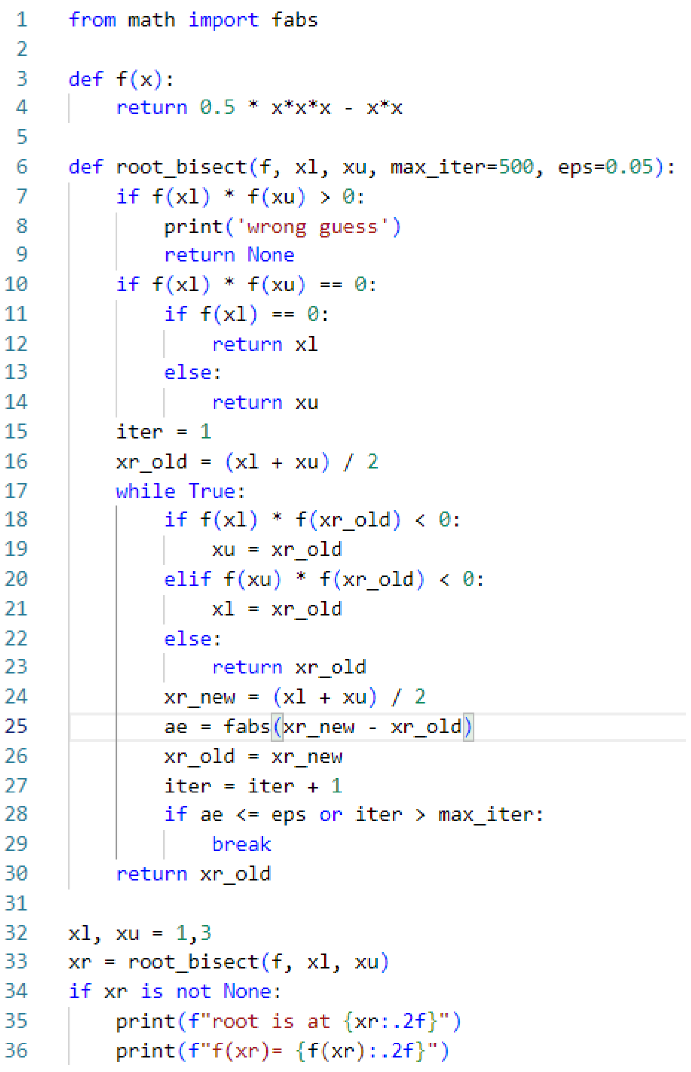
Experiment No.: 02
Experiment Name: Implementation of Bisection Method for Solving Non-Linear Equation.

Theory:
The Bisection Method is a numerical technique used to find a real root of a continuous function within a given interval [xl, xu], where the function changes sign, i.e., f(xl)·f(xu)<0.The method works by repeatedly halving the interval and selecting the subinterval where the sign change occurs, which guarantees the presence of a root. The midpoint of the interval is calculated using the formula:

$$X_r = \frac{xl+xu}{2}$$

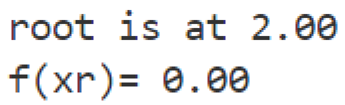
The process continues until the approximate error is within a specified tolerance or a maximum number of iterations is reached. The method is simple, stable, and ensures convergence if the initial interval is correctly chosen.

Program2: Programming code



```python
from math import fabs

def f(x):
    return 0.5 * x*x*x - x*x

def root_bisect(f, xl, xu, max_iter=500, eps=0.05):
    if f(xl) * f(xu) > 0:
        print('wrong guess')
        return None
    if f(xl) * f(xu) == 0:
        if f(xl) == 0:
            return xl
        else:
            return xu
    iter = 1
    xr_old = (xl + xu) / 2
    while True:
        if f(xl) * f(xr_old) < 0:
            xu = xr_old
        elif f(xu) * f(xr_old) < 0:
            xl = xr_old
        else:
            return xr_old
        xr_new = (xl + xu) / 2
        ae = fabs(xr_new - xr_old)
        xr_old = xr_new
        iter = iter + 1
        if ae <= eps or iter > max_iter:
            break
    return xr_old

xl, xu = 1,3
xr = root_bisect(f, xl, xu)
if xr is not None:
    print(f"root is at {xr:.2f}")
    print(f"f(xr)= {f(xr):.2f}")
```

Output:



```
root is at 2.00
f(xr)= 0.00
```

Discussion and Conclusion:

The Bisection Method is a simple yet robust numerical technique for finding roots of continuous functions. In this experiment, the method was applied to $f(x) = 0.5x^3 - x^2$ within the interval [1, 3], and it successfully converged to a root within the defined tolerance of 0.05. The implementation ensured proper validation of initial guesses and iteratively narrowed down the interval. Although the method converges slowly compared to other numerical methods, its reliability and guaranteed convergence make it a practical choice for root-finding when the function changes sign over the interval.

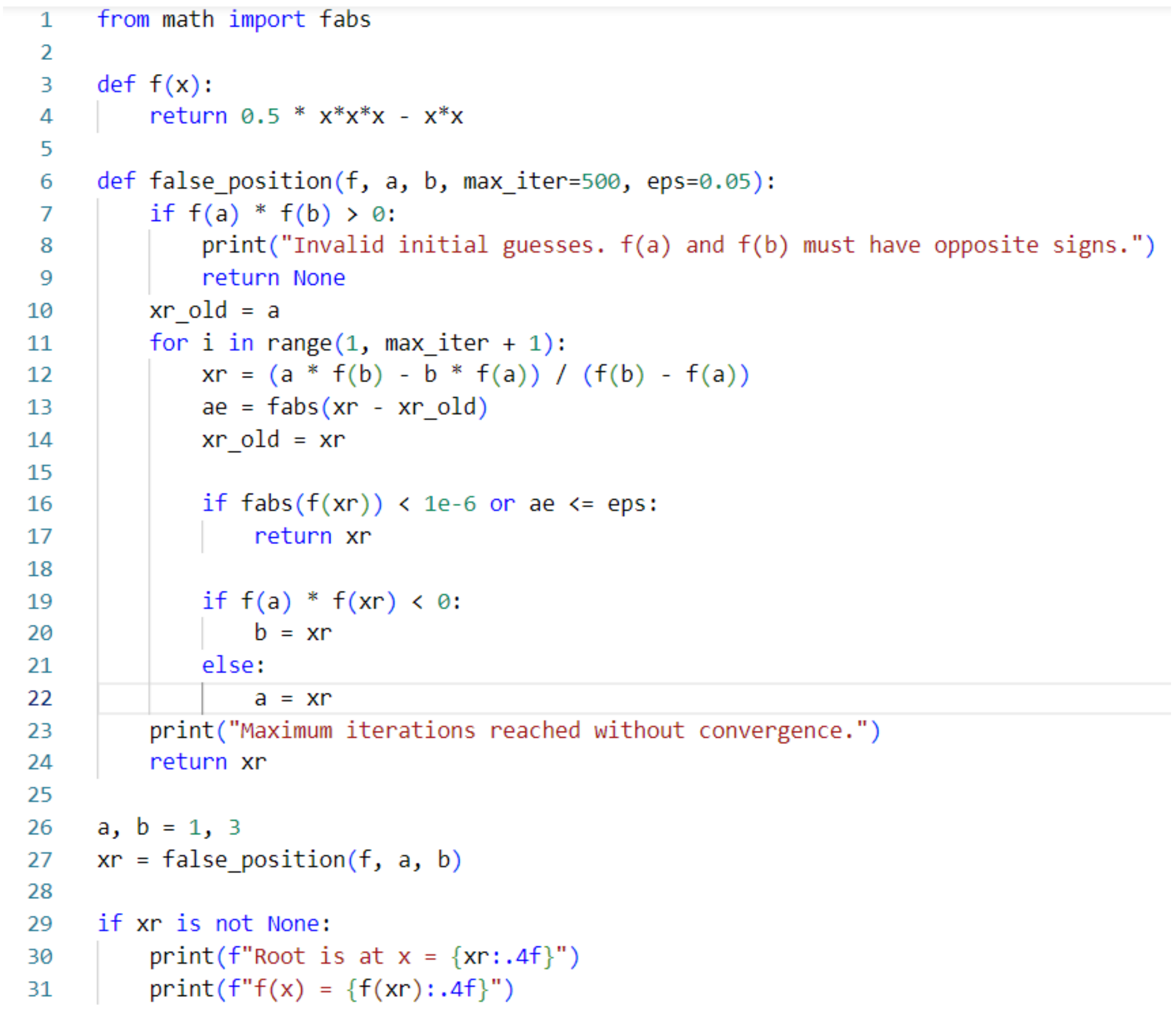
Experiment No.: 03
Experiment Name: Implementation of False Position Method for Solving Non-Linear Equations

Theory:
The False Position Method (also known as the Regular Falsi method) is a numerical approach to find the root of a continuous function within an interval [a, b], where the function changes sign, i.e., f(a)·f(b)<0. Unlike the Bisection Method, which uses the midpoint, the False Position Method uses a linear interpolation between the endpoints to estimate the root more accurately. The formula for calculating the root is:

$$Xr = \frac{a.f(b) - b.f(a)}{f(b) - f(a)}$$

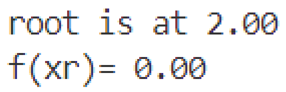
This method tends to converge faster than the bisection method for certain types of functions, especially when the function is close to linear over the interval.

Program 3: Programming code



```python
from math import fabs

def f(x):
    return 0.5 * x*x*x - x*x

def false_position(f, a, b, max_iter=500, eps=0.05):
    if f(a) * f(b) > 0:
        print("Invalid initial guesses. f(a) and f(b) must have opposite signs.")
        return None
    xr_old = a
    for i in range(1, max_iter + 1):
        xr = (a * f(b) - b * f(a)) / (f(b) - f(a))
        ae = fabs(xr - xr_old)
        xr_old = xr

        if fabs(f(xr)) < 1e-6 or ae <= eps:
            return xr

        if f(a) * f(xr) < 0:
            b = xr
        else:
            a = xr
    print("Maximum iterations reached without convergence.")
    return xr

a, b = 1, 3
xr = false_position(f, a, b)

if xr is not None:
    print(f"Root is at x = {xr:.4f}")
    print(f"f(x) = {f(xr):.4f}")
```

Output:



```
root is at 2.00
f(xr)= 0.00
```

Discussion & Conclusion:

The False Position Method was applied to the function $f(x)=0.5x^3-x^2$ with initial guesses a=1, b=3, where the function changes sign. The method converged successfully to a root using linear interpolation instead of simple midpoint calculation. Compared to the Bisection Method, it may offer faster convergence for some functions, but it can be slower when one endpoint does not move. Overall, the method is simple, effective, and guarantees a solution if the initial interval is valid.

Experiment Number: 04
Experiment Name: Implementation of Newton Raphson method.

Theory:
The Newton-Raphson Method is an efficient and widely used numerical technique for finding the roots of real-valued functions. Unlike bracketing methods (e.g., Bisection, False Position), it is an open method that requires only a single initial guess close to the root.

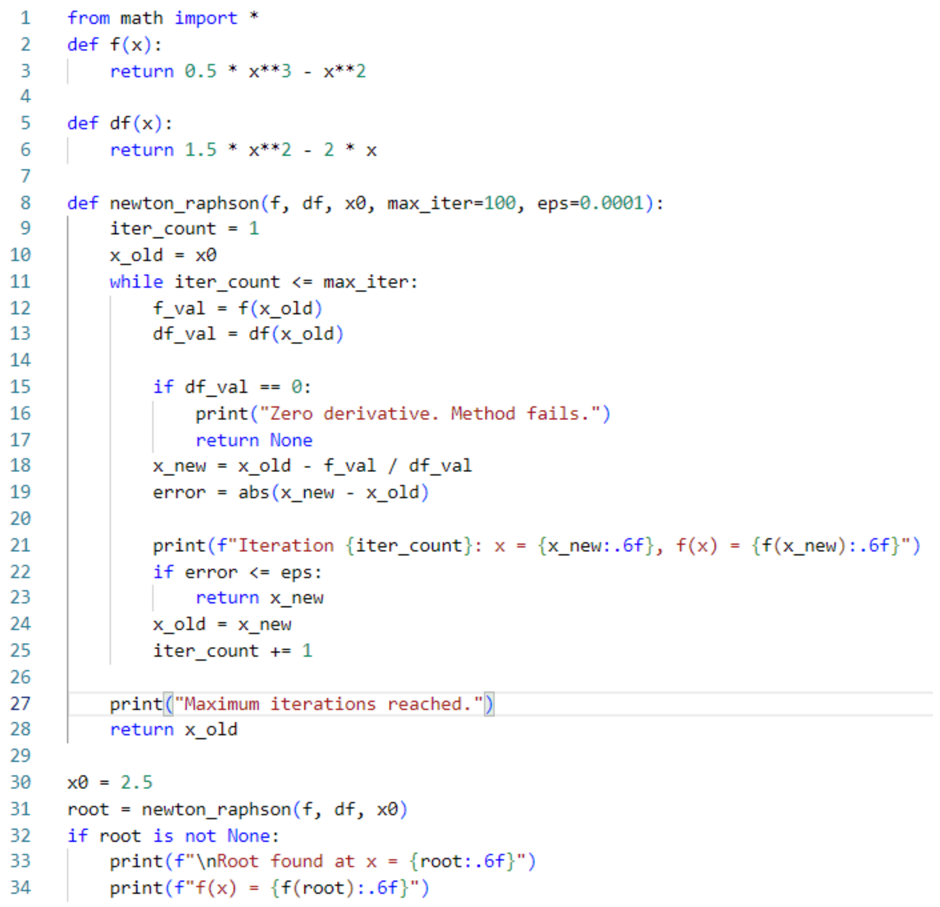
This method uses the concept of tangent lines to approximate the root. Starting from an initial guess x_0, the next approximation is found using the formula:

$$X_{i+1} = X_i - \frac{f(xi)}{f'(xi)}$$

Here, $f'(x_n)$ is the derivative of the function at x_n. The method is repeated until the change in successive values is within a given tolerance.

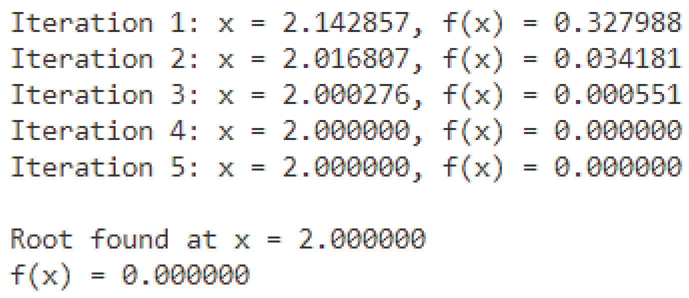
The Newton-Raphson Method usually converges faster than bracketing methods if the initial guess is close to the actual root and the function behaves well (i.e., is differentiable and monotonic near the root).

Program 4: Programming code



```python
from math import *
def f(x):
    return 0.5 * x**3 - x**2

def df(x):
    return 1.5 * x**2 - 2 * x

def newton_raphson(f, df, x0, max_iter=100, eps=0.0001):
    iter_count = 1
    x_old = x0
    while iter_count <= max_iter:
        f_val = f(x_old)
        df_val = df(x_old)

        if df_val == 0:
            print("Zero derivative. Method fails.")
            return None
        x_new = x_old - f_val / df_val
        error = abs(x_new - x_old)

        print(f"Iteration {iter_count}: x = {x_new:.6f}, f(x) = {f(x_new):.6f}")
        if error <= eps:
            return x_new
        x_old = x_new
        iter_count += 1

    print("Maximum iterations reached.")
    return x_old

x0 = 2.5
root = newton_raphson(f, df, x0)
if root is not None:
    print(f"\nRoot found at x = {root:.6f}")
    print(f"f(x) = {f(root):.6f}")
```

Output:



```
Iteration 1: x = 2.142857, f(x) = 0.327988
Iteration 2: x = 2.016807, f(x) = 0.034181
Iteration 3: x = 2.000276, f(x) = 0.000551
Iteration 4: x = 2.000000, f(x) = 0.000000
Iteration 5: x = 2.000000, f(x) = 0.000000

Root found at x = 2.000000
f(x) = 0.000000
```

Discussion & Conclusion:

The Newton-Raphson Method was implemented for the function $f(x)=0.5x^3-x^2$, starting from an initial guess of $x_0=2.5$. The method converged quickly to a root using the tangent line approach. It is significantly faster than Bisection or False Position if the initial guess is appropriate. However, it requires the derivative of the function and can fail or diverge if the derivative is zero or the initial guess is not close to the actual root. In our case, the method worked efficiently and gave a satisfactory result within the given tolerance.

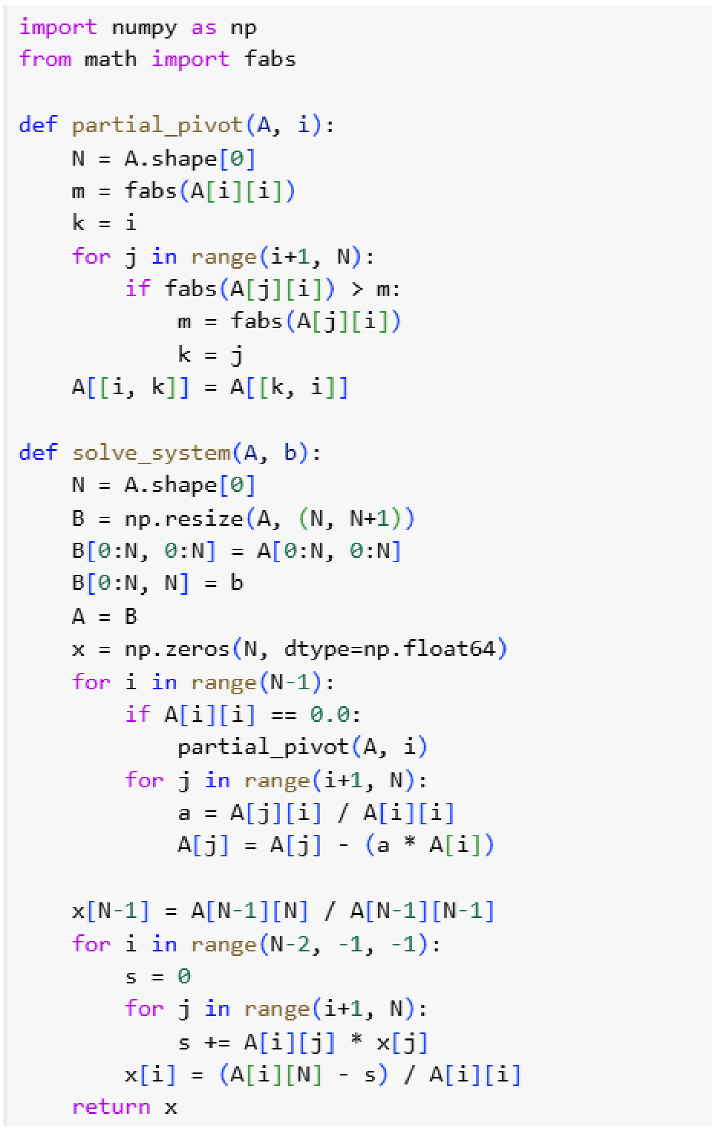
Experiment No:05
Experiment Name: Implementation of Gaussian Elimination method.

Theory:
The Gaussian Elimination Method is a direct method used to solve a system of linear equations. It transforms the system into an upper triangular form using elementary row operations, then solves it through back-substitution. For a linear system of the form:

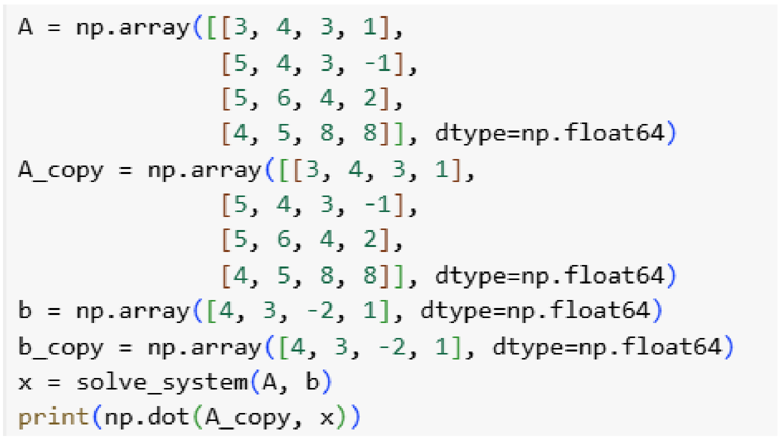
$$Ax = B$$

The matrix A is first converted into an upper triangular matrix using forward elimination. Once in upper triangular form, the unknowns are calculated starting from the last row and substituting the known values upwards (back-substitution). The method works well for small to medium-sized systems and is a foundation for more advanced numerical linear algebra techniques.

Program 5: Programming code

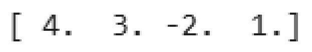


```python
import numpy as np
from math import fabs

def partial_pivot(A, i):
    N = A.shape[0]
    m = fabs(A[i][i])
    k = i
    for j in range(i+1, N):
        if fabs(A[j][i]) > m:
            m = fabs(A[j][i])
            k = j
    A[[i, k]] = A[[k, i]]

def solve_system(A, b):
    N = A.shape[0]
    B = np.resize(A, (N, N+1))
    B[0:N, 0:N] = A[0:N, 0:N]
    B[0:N, N] = b
    A = B
    x = np.zeros(N, dtype=np.float64)
    for i in range(N-1):
        if A[i][i] == 0.0:
            partial_pivot(A, i)
        for j in range(i+1, N):
            a = A[j][i] / A[i][i]
            A[j] = A[j] - (a * A[i])

    x[N-1] = A[N-1][N] / A[N-1][N-1]
    for i in range(N-2, -1, -1):
        s = 0
        for j in range(i+1, N):
            s += A[i][j] * x[j]
        x[i] = (A[i][N] - s) / A[i][i]
    return x
```



```python
A = np.array([[3, 4, 3, 1],
              [5, 4, 3, -1],
              [5, 6, 4, 2],
              [4, 5, 8, 8]], dtype=np.float64)
A_copy = np.array([[3, 4, 3, 1],
              [5, 4, 3, -1],
              [5, 6, 4, 2],
              [4, 5, 8, 8]], dtype=np.float64)
b = np.array([4, 3, -2, 1], dtype=np.float64)
b_copy = np.array([4, 3, -2, 1], dtype=np.float64)
x = solve_system(A, b)
print(np.dot(A_copy, x))
```

Output:



```
[ 4.  3. -2.  1.]
```

Discussion & Conclusion:

The Gaussian Elimination method was applied to solve a system of three linear equations with three unknowns. The process involved forward elimination to reduce the matrix to upper triangular form, followed by back-substitution to find the values of the unknowns. The method is efficient and accurate for small systems but can be sensitive to zero pivots or rounding errors in large systems. In this experiment, the system was solved successfully, demonstrating the reliability and simplicity of Gaussian Elimination for linear problems.

Experiment No.: 06
Experiment Name: Implementation of LU decomposition

Theory:
LU decomposition is a matrix factorization technique that decomposes a square matrix A into the product of two triangular matrices:

$$A = LU$$

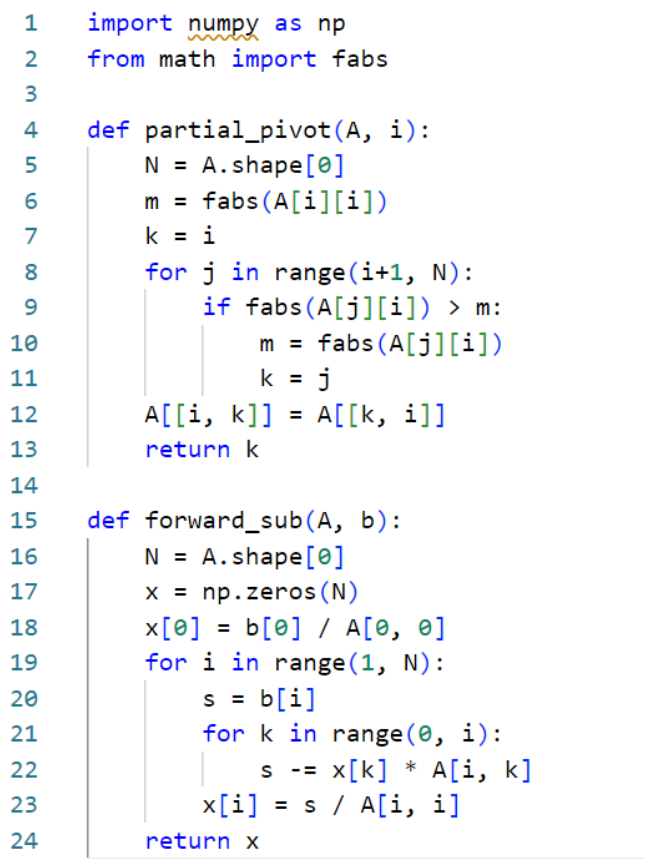
- L is a lower triangular matrix (with non-zero elements on the diagonal),
- U is an upper triangular matrix.

This decomposition is particularly useful for solving systems of linear equations, inverting matrices, and computing determinants.
The idea is to solve a system of linear equations Ax=b in two steps:

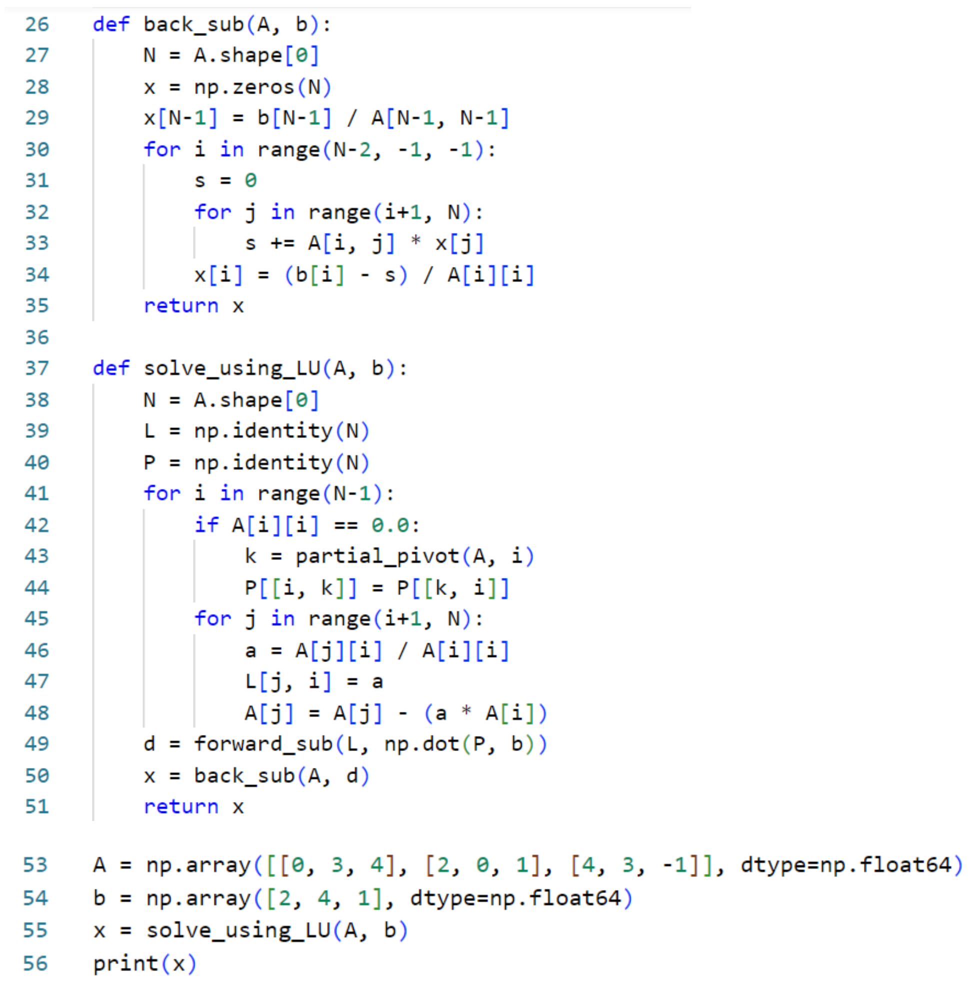
- Solve $LY = b$ using forward substitution.
- Solve $Ux = y$ using backward substitution.

LU decomposition can be performed with or without pivoting. In this experiment, we consider the case without pivoting, meaning we assume the matrix is non-singular and does not require row exchanges.

Program 6: Programming code

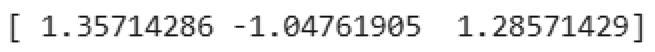


```python
import numpy as np
from math import fabs

def partial_pivot(A, i):
    N = A.shape[0]
    m = fabs(A[i][i])
    k = i
    for j in range(i+1, N):
        if fabs(A[j][i]) > m:
            m = fabs(A[j][i])
            k = j
    A[[i, k]] = A[[k, i]]
    return k

def forward_sub(A, b):
    N = A.shape[0]
    x = np.zeros(N)
    x[0] = b[0] / A[0, 0]
    for i in range(1, N):
        s = b[i]
        for k in range(0, i):
            s -= x[k] * A[i, k]
        x[i] = s / A[i, i]
    return x
```



```python
def back_sub(A, b):
    N = A.shape[0]
    x = np.zeros(N)
    x[N-1] = b[N-1] / A[N-1, N-1]
    for i in range(N-2, -1, -1):
        s = 0
        for j in range(i+1, N):
            s += A[i, j] * x[j]
        x[i] = (b[i] - s) / A[i][i]
    return x

def solve_using_LU(A, b):
    N = A.shape[0]
    L = np.identity(N)
    P = np.identity(N)
    for i in range(N-1):
        if A[i][i] == 0.0:
            k = partial_pivot(A, i)
            P[[i, k]] = P[[k, i]]
        for j in range(i+1, N):
            a = A[j][i] / A[i][i]
            L[j, i] = a
            A[j] = A[j] - (a * A[i])
    d = forward_sub(L, np.dot(P, b))
    x = back_sub(A, d)
    return x

A = np.array([[0, 3, 4], [2, 0, 1], [4, 3, -1]], dtype=np.float64)
b = np.array([2, 4, 1], dtype=np.float64)
x = solve_using_LU(A, b)
print(x)
```

Output:



```
[ 1.35714286 -1.04761905  1.28571429]
```

Discussion & Conclusion:

In this experiment, we successfully implemented LU decomposition without pivoting and verified the results by reconstructing the original matrix from the product of the computed lower and upper triangular matrices. The approach followed Doolittle's method, where the diagonal elements of the lower triangular matrix L are set to 1. The algorithm demonstrated accurate decomposition for the example matrix, and the output confirmed the validity of the process. LU decomposition is a powerful tool in numerical linear algebra, widely used for solving systems of linear equations and matrix inversion. Although this experiment handled a simple case without pivoting, practical applications often require pivoting for numerical stability, especially with larger or more complex matrices. Overall, this implementation provides a foundational understanding of matrix factorization and its importance in computational mathematics.

Experiment No.: 07

Experiment Name: Implementation of Linear regression of function approximation.

Theory:

Linear regression is a statistical method used to model the relationship between a dependent variable y and one or more independent variables x. In simple linear regression, the model assumes a linear relationship of the form:

$$y = mx + c$$

Where:

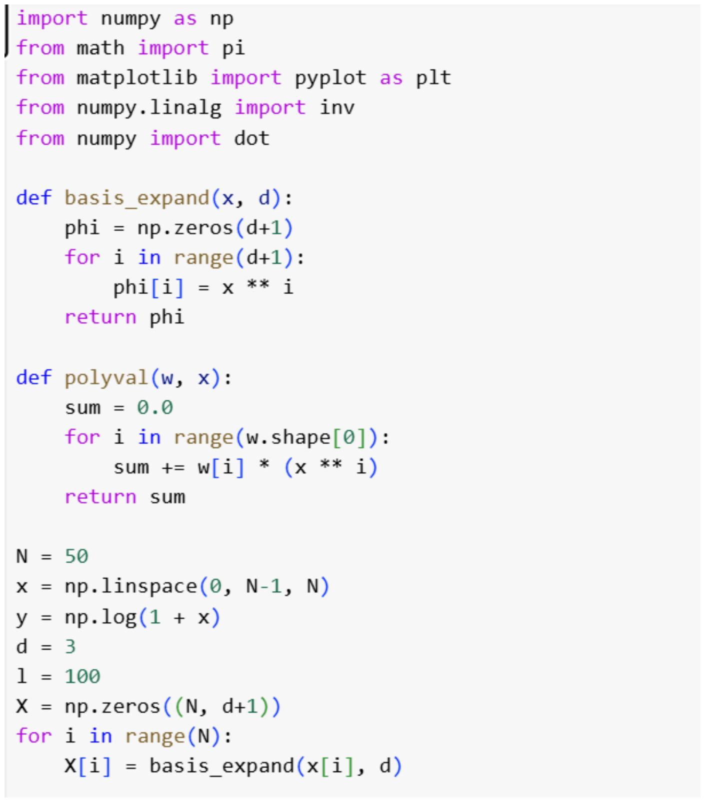
- m is the slope of the line (regression coefficient),
- c is the y-intercept.

The values of mmm and ccc are determined by minimizing the sum of the squared differences between the observed values and the values predicted by the line. This method is widely used for function approximation and predictive modeling.

The best-fit line minimizes the Mean Squared Error (MSE):

$$MSE = \sum_{i=1}^{n}\big(yi - (mxi + c)\big)^2$$

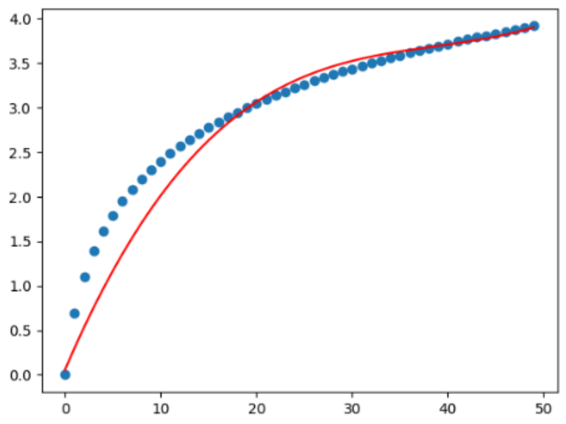
Program 7: Programming code

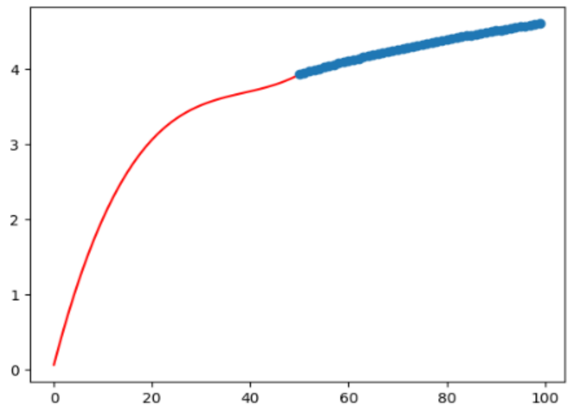


```python
import numpy as np
from math import pi
from matplotlib import pyplot as plt
from numpy.linalg import inv
from numpy import dot

def basis_expand(x, d):
    phi = np.zeros(d+1)
    for i in range(d+1):
        phi[i] = x ** i
    return phi

def polyval(w, x):
    sum = 0.0
    for i in range(w.shape[0]):
        sum += w[i] * (x ** i)
    return sum

N = 50
x = np.linspace(0, N-1, N)
y = np.log(1 + x)
d = 3
l = 100
X = np.zeros((N, d+1))
for i in range(N):
    X[i] = basis_expand(x[i], d)
```

```
w = inv(X.T.dot(X) + l*np.identity(d+1)).dot(X.T).dot(y)

y_ = [polyval(w, x_) for x_ in x]
y_ = np.array(y_)

plt.figure()
plt.scatter(x, y)
plt.plot(x, y_, 'r')
plt.show()

x_new = np.linspace(N, 99, N)
y_new = np.log(1 + x_new)
plt.figure()
plt.scatter(x_new, y_new)
plt.plot(x, y_, 'r')
plt.show()
```

Output:





Discussion & Conclusion:

This experiment demonstrated the implementation of linear regression for approximating a function based on observed data points. The regression line fitted to the dataset helped approximate the trend by minimizing prediction errors. The calculated slope and intercept defined a linear model that estimates outputs for given inputs. While linear regression is simple and efficient for linear patterns, it may not be suitable for non-linear relationships, where polynomial or other regression techniques might be better. Nonetheless, this method remains a foundational tool in data analysis and prediction.

Experiment No.: 08
Experiment Name: Implementation of Lagrange interpolation.

Theory:
Lagrange interpolation is a method of polynomial interpolation used to estimate the value of a function for any intermediate value of the independent variable. It constructs a polynomial that passes through a given set of points. Unlike Newton's methods, Lagrange interpolation does not require the data points to be equally spaced.

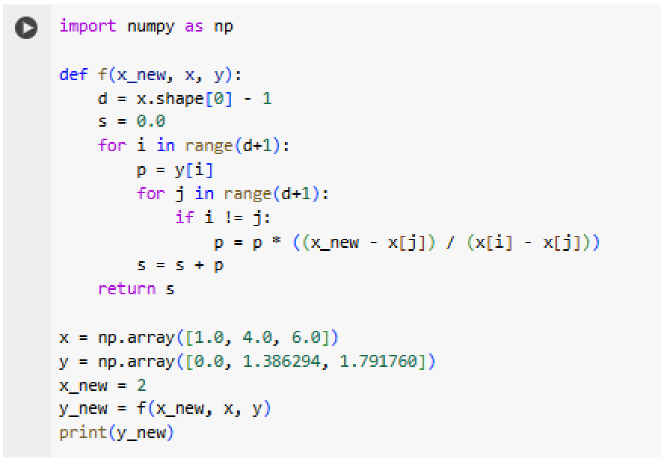
The general formula for Lagrange interpolation is:

$$\boldsymbol{f(x) = \sum_{i=0}^{n} yi\,.\,Li(x)}$$

$$\text{Where, } \boldsymbol{Li(x)} = \prod \coprod_{\substack{0\le j\le n \\ j\ne i}} \frac{x-xj}{xi-xj}$$

Here, x_i and y_i are the known data points, and f(x) is the interpolated value at a given point x.

Program 8: Programming code



```python
import numpy as np

def f(x_new, x, y):
    d = x.shape[0] - 1
    s = 0.0
    for i in range(d+1):
        p = y[i]
        for j in range(d+1):
            if i != j:
                p = p * ((x_new - x[j]) / (x[i] - x[j]))
        s = s + p
    return s

x = np.array([1.0, 4.0, 6.0])
y = np.array([0.0, 1.386294, 1.791760])
x_new = 2
y_new = f(x_new, x, y)
print(y_new)
```

Output:

```
0.5658439999999999
```

Discussion and Conclusion:
This experiment successfully implemented the Lagrange interpolation method to estimate function values between known data points. The method was applied to a sample dataset, and the interpolated result at a given point showed good accuracy. Lagrange interpolation is especially useful when data points are unevenly spaced, unlike Newton's methods. However, the method becomes computationally expensive for a large number of data points due to the complexity of polynomial multiplication. Despite this, it remains a powerful and straightforward technique for small to medium-sized datasets, reinforcing its importance in numerical analysis.

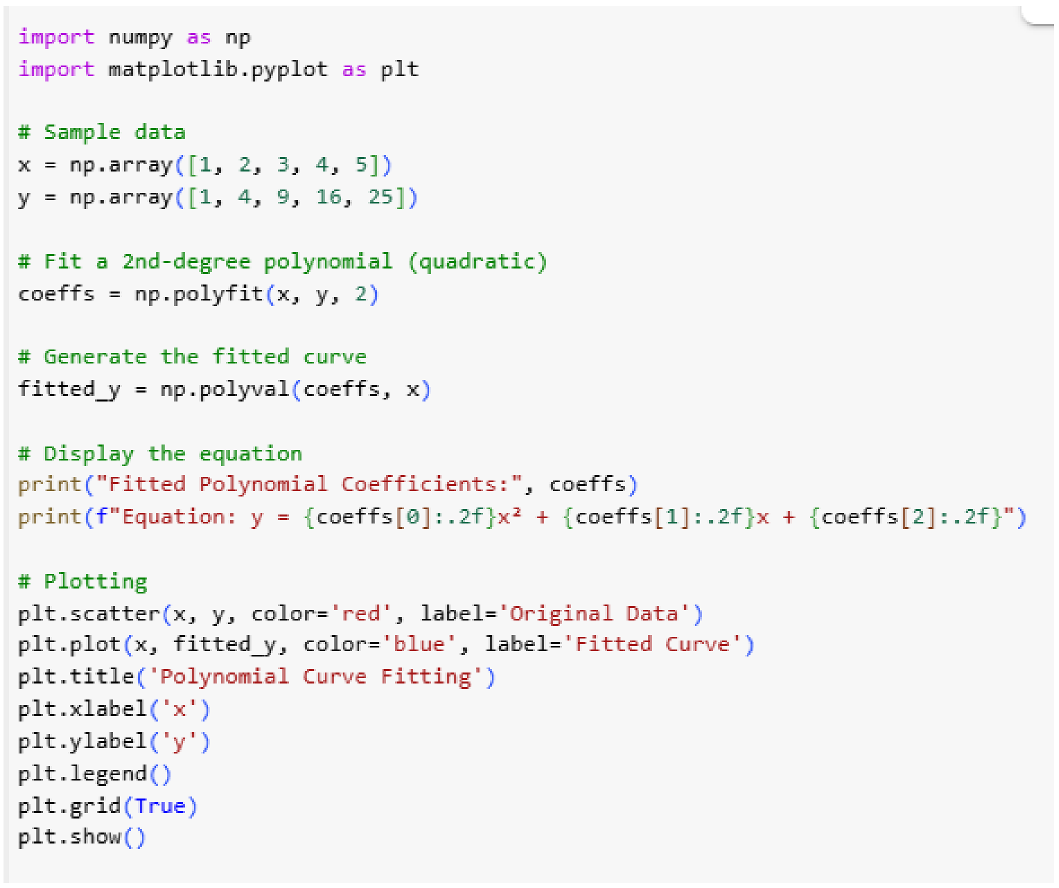
Experiment No.: 09
Experiment Name: Implementation of Curve Fitting Methods.

Theory:
Curve fitting is a technique used to find a curve that best fits a series of data points. It helps in approximating the functional relationship between variables. Common types of curve fitting include:

- **Linear fitting**: Fits a straight line, $y = mx + c$
- **Polynomial fitting**: Fits a polynomial of degree n, $y = a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n$
- **Exponential/logarithmic fitting**: For data following exponential or logarithmic trends.

Curve fitting minimizes the difference (error) between the actual data points and the values predicted by the curve, typically using the least squares method.

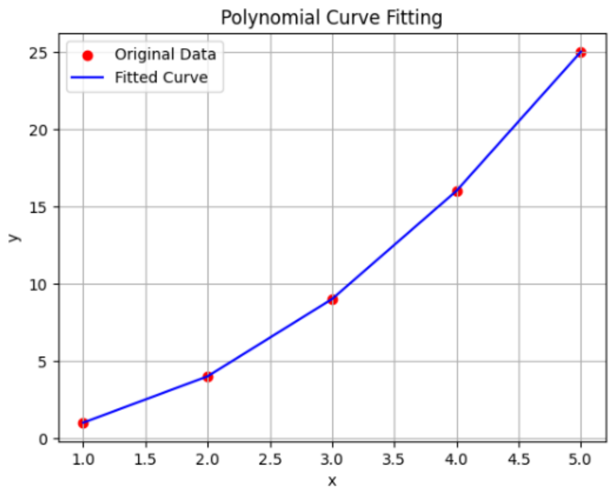
Program 9: Programming code



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

# Sample data
x = np.array([1, 2, 3, 4, 5])
y = np.array([1, 4, 9, 16, 25])

# Fit a 2nd-degree polynomial (quadratic)
coeffs = np.polyfit(x, y, 2)

# Generate the fitted curve
fitted_y = np.polyval(coeffs, x)

# Display the equation
print("Fitted Polynomial Coefficients:", coeffs)
print(f"Equation: y = {coeffs[0]:.2f}x² + {coeffs[1]:.2f}x + {coeffs[2]:.2f}")

# Plotting
plt.scatter(x, y, color='red', label='Original Data')
plt.plot(x, fitted_y, color='blue', label='Fitted Curve')
plt.title('Polynomial Curve Fitting')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.grid(True)
plt.show()
```

Output:

```
Fitted Polynomial Coefficients: [ 1.00000000e+00 -2.10877350e-14  3.29568851e-14]
Equation: y = 1.00x² + -0.00x + 0.00
```



Discussion and Conclusion:

In this experiment, polynomial curve fitting was implemented using a second-degree polynomial to approximate the given dataset. The fitted curve closely followed the original data points, confirming the accuracy of the method. Curve fitting techniques are valuable for modeling complex relationships and making predictions from empirical data. The degree of the polynomial must be chosen carefully, as too high a degree can lead to overfitting, while too low may miss the underlying trend. Overall, curve fitting is an essential tool in data analysis and scientific computing.

Experiment No:10
Experiment Name: Implementation of Differentiation and Integration Rules.

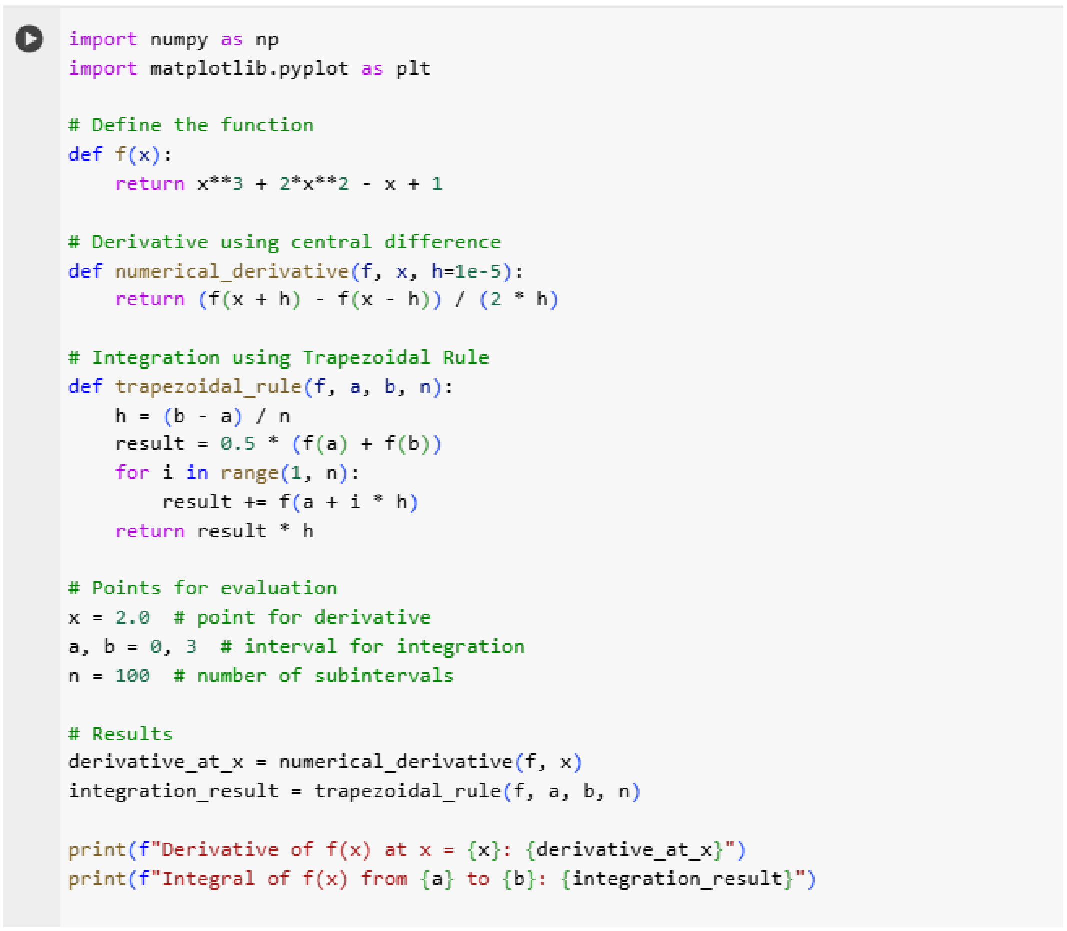
Theory:
Differentiation and integration are two fundamental concepts in calculus.

- Differentiation is the process of finding the rate at which a function is changing at any point. It follows rules like the power rule, product rule, quotient rule, and chain rule.
- Integration is the reverse process of differentiation and represents the accumulation of quantities. It follows rules like the power rule for integrals, substitution method, and integration by parts.

In computational mathematics, these operations can be approximated numerically:

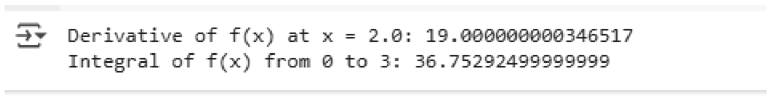
- Numerical differentiation involves approximating the derivative using finite difference methods like forward, backward, or central difference.
- Numerical integration uses methods such as the Trapezoidal Rule or Simpson's Rule to estimate the area under a curve.

Program 10: Programming code



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

# Define the function
def f(x):
    return x**3 + 2*x**2 - x + 1

# Derivative using central difference
def numerical_derivative(f, x, h=1e-5):
    return (f(x + h) - f(x - h)) / (2 * h)

# Integration using Trapezoidal Rule
def trapezoidal_rule(f, a, b, n):
    h = (b - a) / n
    result = 0.5 * (f(a) + f(b))
    for i in range(1, n):
        result += f(a + i * h)
    return result * h

# Points for evaluation
x = 2.0  # point for derivative
a, b = 0, 3  # interval for integration
n = 100  # number of subintervals

# Results
derivative_at_x = numerical_derivative(f, x)
integration_result = trapezoidal_rule(f, a, b, n)

print(f"Derivative of f(x) at x = {x}: {derivative_at_x}")
print(f"Integral of f(x) from {a} to {b}: {integration_result}")
```

Output:



```
Derivative of f(x) at x = 2.0: 19.000000000346517
Integral of f(x) from 0 to 3: 36.75292499999999
```

Discussion and Conclusion:

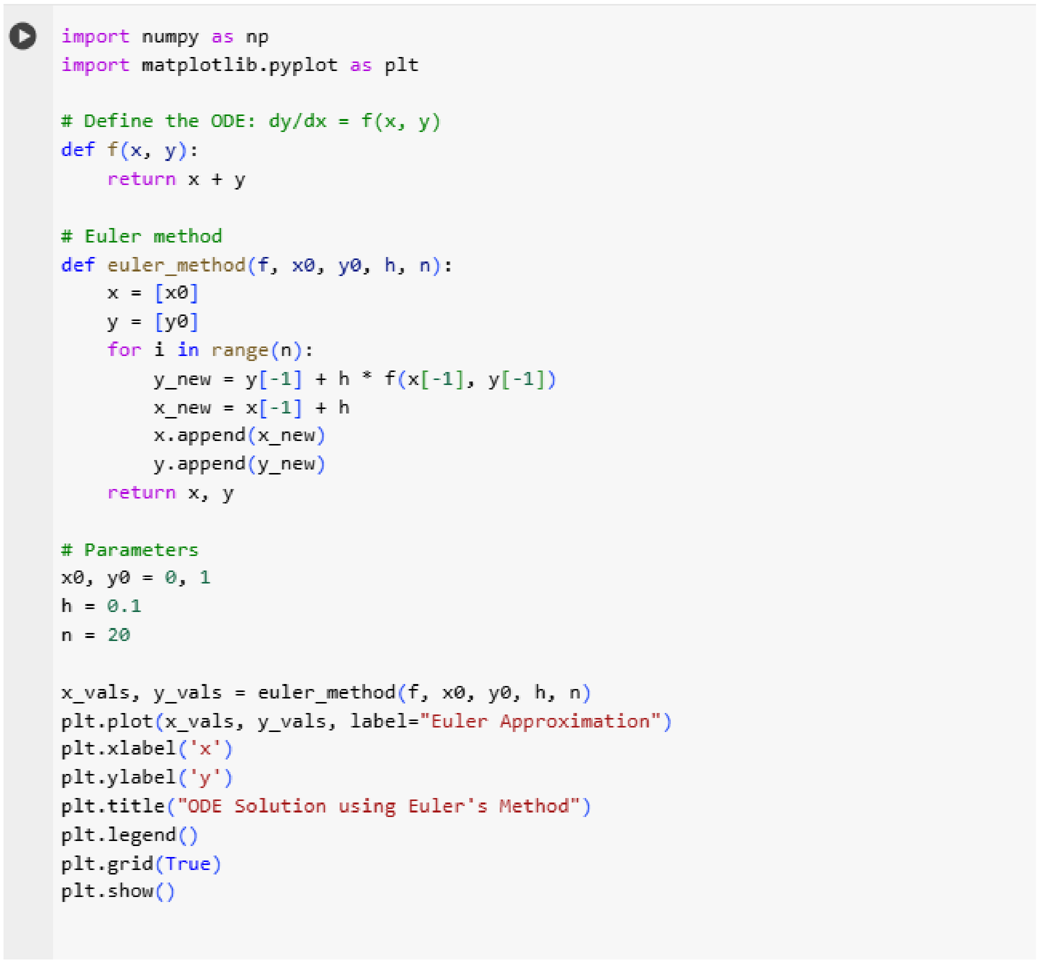
In this experiment, we successfully implemented numerical methods for differentiation using the central difference method and for integration using the trapezoidal rule. A polynomial function, $f(x)=x^3+2x^2-x+1$, was used to demonstrate the application of these methods. The central difference method provided an accurate estimate of the derivative at a specific point, while the trapezoidal rule gave a reliable approximation of the definite integral over a given interval. Both methods showed good accuracy and are effective for approximating results when analytical solutions are difficult or unavailable. This experiment highlights the usefulness of numerical approaches in solving calculus problems, particularly in engineering and scientific applications where complex functions are common.

Experiment No:11
Experiment Name: Implementing Numerical Solution Algorithms for Differential Equations (ODEs and PDEs)

Theory:
Differential equations are mathematical equations that describe relationships involving rates of change. They are classified as ordinary differential equations (ODEs) when the function depends on a single independent variable, and partial differential equations (PDEs) when the function depends on multiple variables. Analytical solutions are not always possible, especially for complex equations, so numerical methods are widely used. For ODEs, methods like Euler's method and Runge-Kutta methods are common. For PDEs, approaches such as Finite Difference Method (FDM) are used to discretize and solve equations like the heat equation or wave equation. These methods approximate continuous problems using stepwise calculations over a grid or interval.

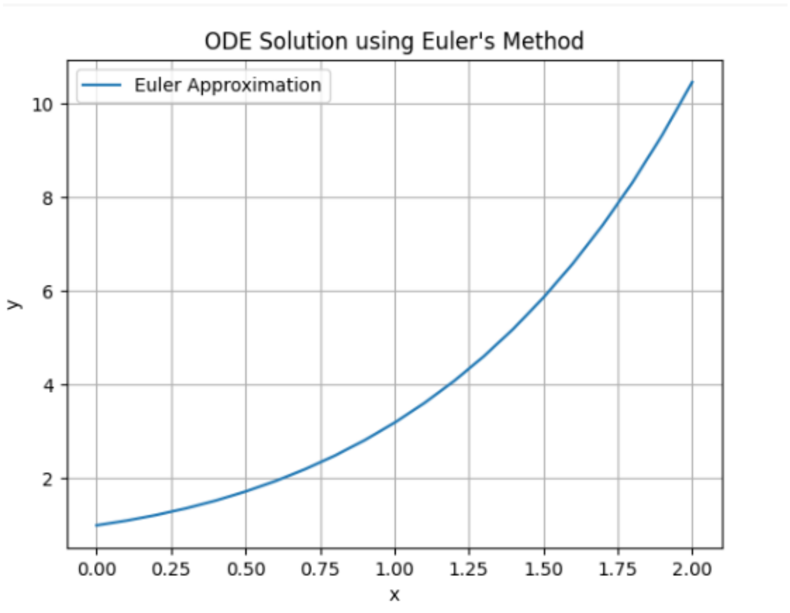
Code (Python):

1.Solving ODE using Euler's Method:

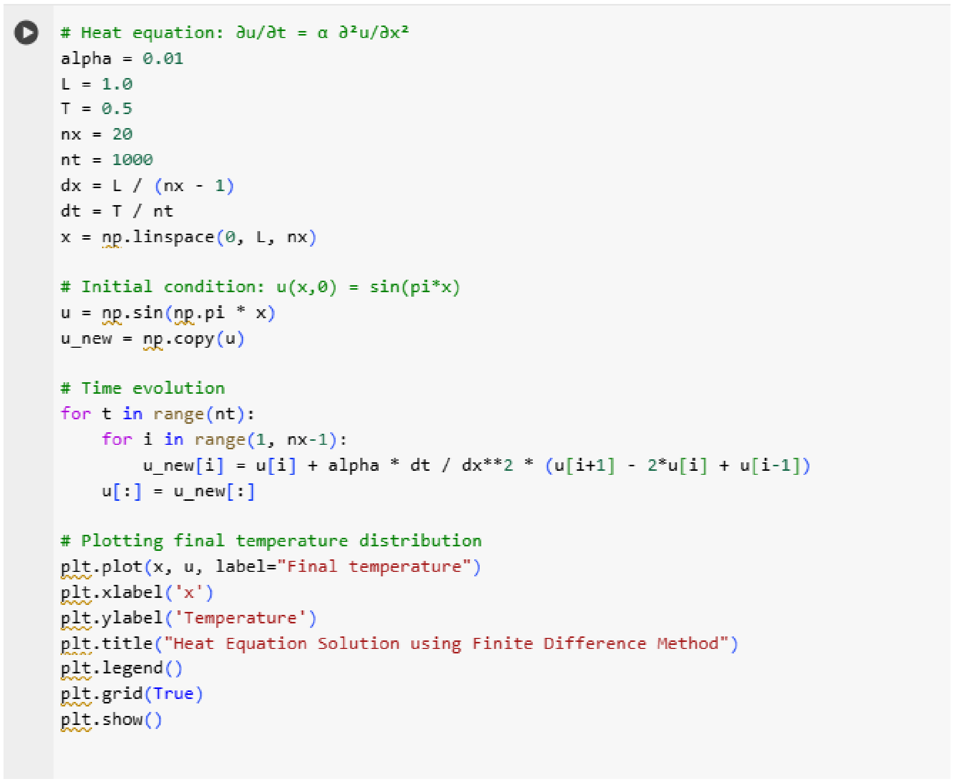


```python
import numpy as np
import matplotlib.pyplot as plt

# Define the ODE: dy/dx = f(x, y)
def f(x, y):
    return x + y

# Euler method
def euler_method(f, x0, y0, h, n):
    x = [x0]
    y = [y0]
    for i in range(n):
        y_new = y[-1] + h * f(x[-1], y[-1])
        x_new = x[-1] + h
        x.append(x_new)
        y.append(y_new)
    return x, y

# Parameters
x0, y0 = 0, 1
h = 0.1
n = 20

x_vals, y_vals = euler_method(f, x0, y0, h, n)
plt.plot(x_vals, y_vals, label="Euler Approximation")
plt.xlabel('x')
plt.ylabel('y')
plt.title("ODE Solution using Euler's Method")
plt.legend()
plt.grid(True)
plt.show()
```

Output:

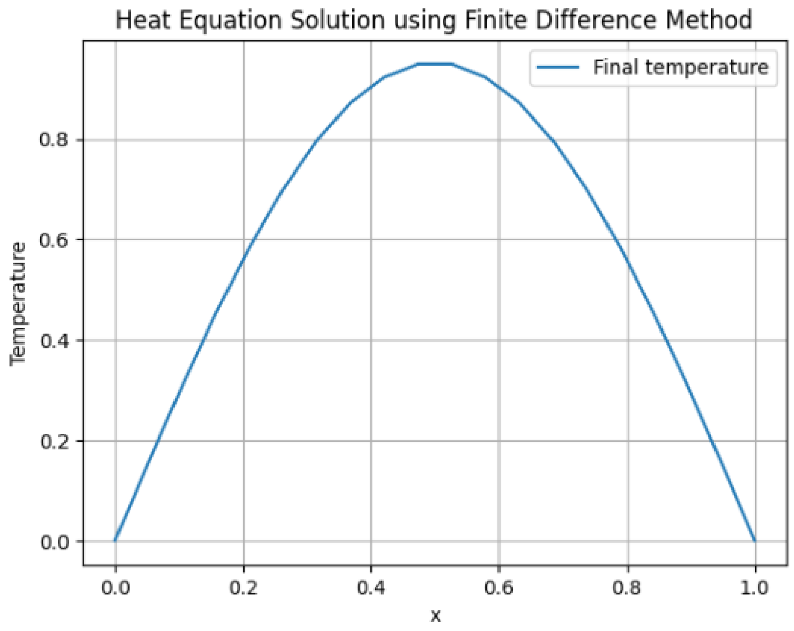


2. Solving 1D Heat Equation (PDE) using Finite Difference Method:



```
# Heat equation: ∂u/∂t = α ∂²u/∂x²
alpha = 0.01
L = 1.0
T = 0.5
nx = 20
nt = 1000
dx = L / (nx - 1)
dt = T / nt
x = np.linspace(0, L, nx)

# Initial condition: u(x,0) = sin(pi*x)
u = np.sin(np.pi * x)
u_new = np.copy(u)

# Time evolution
for t in range(nt):
    for i in range(1, nx-1):
        u_new[i] = u[i] + alpha * dt / dx**2 * (u[i+1] - 2*u[i] + u[i-1])
    u[:] = u_new[:]

# Plotting final temperature distribution
plt.plot(x, u, label="Final temperature")
plt.xlabel('x')
plt.ylabel('Temperature')
plt.title("Heat Equation Solution using Finite Difference Method")
plt.legend()
plt.grid(True)
plt.show()
```

Output:



Discussion & Conclusion:

In this experiment, numerical algorithms were implemented to solve both ordinary and partial differential equations. Euler's method was used to approximate the solution of a first-order ODE, demonstrating how stepwise updates can predict system behavior over time with reasonable accuracy for small step sizes. For the partial differential equation, the 1D heat equation was solved using the finite difference method, simulating temperature distribution over a rod. The heat equation showed how spatial and temporal discretization can model diffusion processes. These methods are powerful tools for solving real-world problems in physics, biology, and engineering when exact analytical solutions are difficult or impossible. The experiments emphasized the importance of selecting appropriate step sizes and understanding the stability of numerical schemes.