**Experiment No. 01**

**Experiment Name : Introduction to Python and Error Analysis.**

**Theory :**

In numerical analysis and scientific computation, errors occur due to approximation.

There are mainly three important types of errors:

1. **Absolute Error (AE):** Difference between the true value and the approximate value.

**AE=**∣**True Value−Approximate Value**∣

1. **Relative Error (RE):**It is the ratio of the absolute error to the true value.

**RE=AE/**∣**True Value**∣

1. **Percentage Error (PE):**It is the relative error expressed in percentage form.

**PE=RE\*100**

**Code:**

from math import fabs

def calculate\_error(T, A):

    ae = fabs(T-A)

    re = fabs(ae/T)

    pe = re\*100

    return ae, re, pe

true\_value = float(input("Enter true value: "))

approx\_value = float(input("Enter approx value: "))

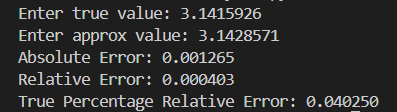
ae, re, pe = calculate\_error(true\_value, approx\_value)

print(f"Absolute Error: {ae:.6f}")

print(f"Relative Error: {re:.6f}")

print(f"True Percentage Relative Error: {pe:.6f}")

**Output:**



**Discussion & Conclusion:**

In this experiment, we calculated absolute, relative, and percentage errors. These errors show how much the approximate value differs from the true value. Percentage error is most useful because it gives the error in a simple percentage form, which is easy to understand and compare.

This experiment demonstrates the importance of error analysis in measuring accuracy. Absolute, relative and percentage errors help us effectively evaluate the difference between true and approximate values.

**Experiment No.** **02**

**Experiment Name : Implementation of Bisection Method for Solving Non-Linear Equation.**

**Theory:**

The Bisection Method is a numerical method used to find the root of a continuous function.

It works on the principle of the Intermediate Value Theorem, which states:

If a function f(x) is continuous in the interval [xl, xu] and f(xl)⋅f(xu)<0 , then there exists at least one root

between xl and xu

**Steps:**

1. Choose two initial guesses xl and xu such that f(xl)⋅f(xu)<0

2. Find midpoint: Xr=xl+xu/2

3. Check if f(xr)=0 or the error is less than tolerance. If yes, stop.

4. Otherwise, replace either xl or xu depending on the sign of f(xr).

5. Repeat until the desired accuracy is achieved.

**Code:**

**from** math **import** fabs

**def** **f**(x, expr):

**return** eval(expr)

**def** **bisection**(expr, xl, xu, eps=0.001, max\_itr=100):

**if** f(xl, expr) \* f(xu, expr) > 0:

print("Wrong guess! f(xl) and f(xu) must have opposite signs.")

**return** None

**for** i **in** range(max\_itr):

xr = (xl + xu) / 2

**if** f(xr, expr) == 0 **or** fabs(f(xu, expr) - f(xl, expr)) < eps:

**return** xr

**if** f(xl, expr) \* f(xr, expr) < 0:

xu = xr

**else**:

xl = xr

**return** xr

expr = input("Enter function f(x): ")

xl = float(input("Enter lower guess xl: "))

xu = float(input("Enter upper guess xu: "))

eps = float(input("Enter tolerance (eps): "))

max\_itr = int(input("Enter maximum iterations: "))

xr = bisection(expr, xl, xu, eps, max\_itr)

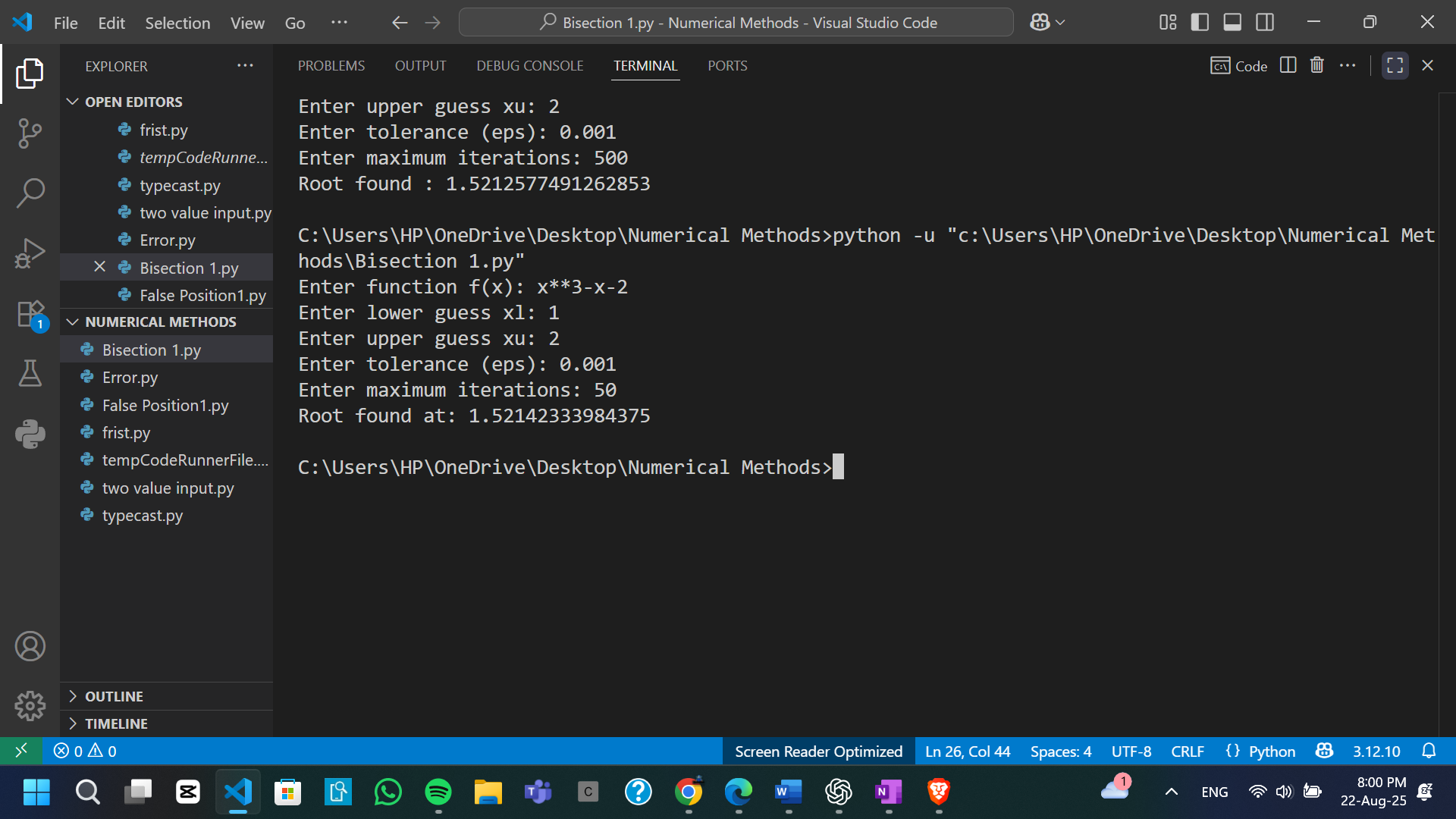
**if** xr **is** **not** None:

print("Root found at:", xr)

**else**:

print("No root found.")

**Output:**



**Discussion & Conclusion:**

The Bisection Method is one of the simplest and most reliable root-finding techniques. Its main requirement is that the function must take opposite signs at the chosen interval endpoints, which ensures the existence of a root within that interval. Although the method converges more slowly compared to advanced techniques such as the Newton-Raphson method, it guarantees convergence provided the initial guesses are valid.

From the experiment, it is evident that the Bisection Method is highly effective for approximating roots of non-linear equations. Its straightforward implementation, coupled with assured convergence, makes it a valuable tool in numerical computations, especially when reliability is prioritized over speed.

**Experiment No. 03**

**Experiment Name : Implementation of False Position Method for Solving Non-Linear Equations**

**Theory:**

The False Position Method (Regula Falsi) is a numerical method used to approximate the

root of a function.

It is similar to the Bisection Method, but instead of taking the midpoint, it finds the

intersection of the secant line with the x-axis

**Formula:**

Xr=xl⋅f(xu)−xu⋅f(xl)/f(xu)−f(xl)

Steps:

1. Choose initial guesses xl and xu such that f(xl)⋅f(xu)<0.

2. Compute root approximation using the formula above.

3. If f(xr)= or error < tolerance, stop.

4. Otherwise, replace xlor xu depending on the sign of f(xr)..

5. Repeat until desired accuracy is achieved.

The false position method often converges faster than the bisection method.

**Code:**

**from** math **import** fabs

**def** **f**(x, expr):

**return** eval(expr)

**def** **false\_position**(expr, xl, xu, eps=0.001, max\_itr=200):

**if** f(xl, expr) \* f(xu, expr) > 0:

print("Wrong guess! f(xl) and f(xu) must have opposite signs.")

**return** None

**for** i **in** range(max\_itr):

xr =(xl\*f(xu,expr)-xu\*f(xl, expr))/(f(xu, expr)-f(xl, expr))

**if** f(xr, expr) == 0 **or** fabs(f(xr, expr)) < eps:

**return** xr

**if** f(xl, expr) \* f(xr, expr) < 0:

xu = xr

**else**:

xl = xr

**return** xr

expr = input("Enter function f(x): ")

xl = float(input("Enter lower guess xl: "))

xu = float(input("Enter upper guess xu: "))

eps = float(input("Enter tolerance (eps): "))

max\_itr = int(input("Enter maximum iterations: "))

xr = false\_position(expr, xl, xu, eps, max\_itr)

**if** xr **is** **not** None:

print("Root found :", xr)

**else**:

print("No root found.")

**Output:**

A screen shot of a computer

AI-generated content may be incorrect.

**Discussion & Conclusion:**

The False Position Method enhances the Bisection Method by using a secant line to approximate the root, which generally leads to faster convergence. However, in certain cases, convergence can slow down if one endpoint remains unchanged over many iterations.

From the experiment, it is evident that the False Position Method is both reliable and efficient for approximating roots of non-linear equations. Compared to the Bisection Method, it often achieves results more quickly while still maintaining the guarantee of convergence, making it a practical choice for numerical problem-solving.