**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 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.

• Percentage Error (PE):

Represents the relative error as a percentage.

PE = RE × 100

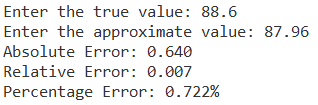
These measures help in evaluating how accurate an approximation is and guide improvements in

numerical techniques.

**Program 1:** Programming Code

1. true\_value **=** float(input("Enter the true value: "))
2. approx\_value **=** float(input("Enter the approximate value: "))
4. abs\_error **=** abs(true\_value**-** approx\_value)
5. rel\_error **=** abs\_error **/** true\_value
6. perc\_error **=** rel\_error **\*** 100
8. print(f"Absolute Error: {abs\_error:.3f}")
9. print(f"Relative Error: {rel\_error:.3f}")
10. print(f"Percentage Error: {perc\_error:.3f}%")

**Output:**

****

**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:**

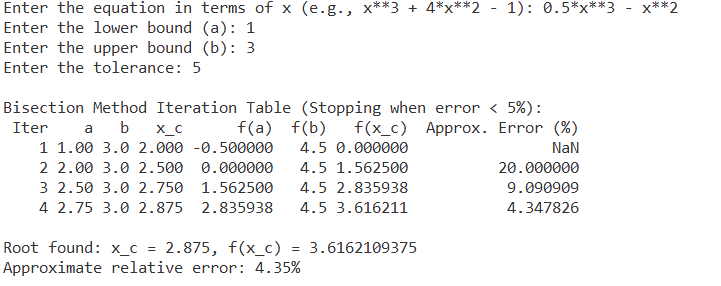
The Bisection Method is a numerical technique used to find a real root of a continuous function within a given interval [ⅹl, xu], where the function changes sign, i.e., f(xl)⋅f(xu)<0. The method works by repeatedly halving the interval and selecting the subinterval where the sign change occurs, which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

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

1. **import** pandas as pd
3. **def** f(x, equation):
4. **return** eval(equation)
6. equation **=** input("Enter the equation in terms of x (e.g., x\*\*3 + 4\*x\*\*2 - 1): ")
7. a **=** float(input("Enter the lower bound (a): "))
8. b **=** float(input("Enter the upper bound (b): "))
9. tol **=** float(input("Enter the tolerance: "))
11. fa **=** f(a, equation)
12. fb **=** f(b, equation)
14. **if** fa **\*** fb > 0:
15. print("Bisection method fails. f(a) and f(b) should have opposite signs.")
16. **else**:
17. data **=** []
18. prev\_c **=** None
19. iteration **=** 1
21. **while** True:
22. c **=** (a **+** b) **/** 2
23. fc **=** f(c, equation)
25. **if** prev\_c **is** **not** None:
26. approx\_error **=** abs((c **-** prev\_c) **/** c) **\*** 100
27. **else**:
28. approx\_error **=** None
30. data.append([iteration, a, b, c, fa, fb, fc, approx\_error])
32. **if** approx\_error **is** **not** None **and** approx\_error < tol:
33. **break**
35. prev\_c **=** c
37. **if** fa **\*** fc < 0:
38. b **=** c
39. fb **=** fc
40. **else**:
41. a **=** c
42. fa **=** fc
44. iteration **+=** 1
46. df **=** pd.DataFrame(data, columns**=**["Iter", "a", "b", "x\_c", "f(a)", "f(b)", "f(x\_c)", "Approx. Error (%)"])
47. print("\nBisection Method Iteration Table (Stopping when error < 5%):")
48. print(df.to\_string(index**=**False))
50. print(f"\nRoot found: x\_c = {c}, f(x\_c) = {fc}")
51. print(f"Approximate relative error: {approx\_error:.2f}%")

**Output:**

****

**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)=0.5x3 - x2 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:**

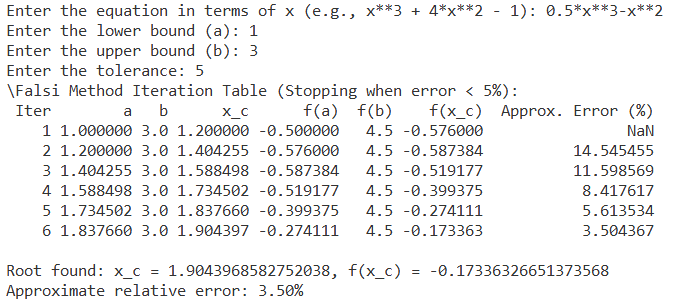
The False Position Method (also known as the Regular Falsi method) is a numerical approach to find the root of a continuous function within an interval [a, b], where the function changes sign, i.e., f(a)⋅f(b)<0. Unlike the Bisection Method, which uses the midpoint, the False Position Method uses a linear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

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.

1. **import** pandas as pd
3. **def** f(x, equation):
4. **return** eval(equation)
6. equation **=** input("Enter the equation in terms of x (e.g., x\*\*3 + 4\*x\*\*2 - 1): ")
7. a **=** float(input("Enter the lower bound (a): "))
8. b **=** float(input("Enter the upper bound (b): "))
9. tol **=** float(input("Enter the tolerance: "))
11. fa **=** f(a, equation)
12. fb **=** f(b, equation)
14. **if** fa **\*** fb > 0:
15. print("Bisection method fails. f(a) and f(b) should have opposite signs.")
16. **else**:
17. data **=** []
18. prev\_c **=** None
19. iteration **=** 1
21. **while** True:
22. c **=** (a **\*** fb **-** b **\*** fa) **/** (fb **-** fa)
23. fc **=** f(c, equation)
25. **if** prev\_c **is** **not** None:
26. approx\_error **=** abs((c **-** prev\_c) **/** c) **\*** 100
27. **else**:
28. approx\_error **=** None
30. data.append([iteration, a, b, c, fa, fb, fc, approx\_error])
32. **if** approx\_error **is** **not** None **and** approx\_error < tol:
33. **break**
35. prev\_c **=** c
37. **if** fa **\*** fc < 0:
38. b **=** c
39. fb **=** fc
40. **else**:
41. a **=** c
42. fa **=** fc
44. iteration **+=** 1
46. df **=** pd.DataFrame(data, columns**=**["Iter", "a", "b", "x\_c", "f(a)", "f(b)", "f(x\_c)", "Approx. Error (%)"])
47. print("\Falsi Method Iteration Table (Stopping when error < 5%):")
48. print(df.to\_string(index**=**False))
50. print(f"\nRoot found: x\_c = {c}, f(x\_c) = {fc}")
51. print(f"Approximate relative error: {approx\_error:.2f}%")

**Output:**

****

**Discussion & Conclusion:**

The False Position Method was applied to the function f(x)=0.5x3−x2 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.