

# **University of Dhaka**

Department of Computer Science and Engineering

**Implementation and Comparative Analysis of the  
Bisection, False Position, Newton–Raphson, and Secant Methods  
for Finding Roots of Nonlinear Equations**

**Course: CSE-3212**

**Submitted By:**

Ashesh Bar

Roll: 18

Session: 2021–22

# 1 Introduction

Many engineering and scientific problems involve equations that cannot be solved analytically, requiring numerical techniques to find approximate solutions. Root-finding methods are iterative numerical algorithms used to find the real roots of nonlinear equations of the form  $f(x) = 0$ .

Among the widely used approaches are:

- **Bisection Method:** A bracketing method that repeatedly divides an interval in half and selects a subinterval where the sign of the function changes. It is simple and always convergent if the initial interval is valid, but converges slowly.
- **False Position (Regula Falsi) Method:** Another bracketing method that uses a linear interpolation to estimate the root. It generally converges faster than Bisection but can stagnate in some cases.
- **Newton–Raphson Method:** An open method that uses the derivative of the function to predict the next approximation. It offers quadratic convergence near the root but requires a good initial guess and a differentiable function.
- **Secant Method:** A derivative-free alternative to Newton–Raphson, using two prior approximations to estimate the slope. It usually converges faster than bracketing methods and is less computationally expensive than Newton–Raphson.

These methods are fundamental in numerical analysis, forming the basis for solving nonlinear systems and optimization problems in computational mathematics, physics, and engineering.

# 2 Objectives

The primary objectives of this experiment are:

1. To implement four numerical root-finding methods — Bisection, False Position, Newton–Raphson, and Secant — using Python.
2. To determine the root of the nonlinear equation:

$$f(h) = h^3 - 10h + 5e^{-h/2} - 2 = 0$$

3. To achieve a convergence criterion of approximate relative error  $e_a \leq 0.001\%$ .
4. To compare the convergence behavior and accuracy of all four methods using a graphical representation of error versus iteration.

- To analyze the efficiency and convergence rate of each method, identifying their advantages and limitations.

## 3 Algorithms

### 3.1 Bisection Method

- Choose an interval  $[a, b]$  such that  $f(a) \cdot f(b) < 0$ .
- Compute midpoint  $c = \frac{a+b}{2}$ .
- Evaluate  $f(c)$ .
- If  $f(a) \cdot f(c) < 0$ , set  $b = c$ ; else, set  $a = c$ .
- Repeat until  $e_a \leq 0.001\%$ .

### 3.2 False Position Method

- Choose an interval  $[a, b]$  such that  $f(a) \cdot f(b) < 0$ .
- Compute
$$c = \frac{af(b) - bf(a)}{f(b) - f(a)}$$
- Evaluate  $f(c)$ .
- If  $f(a) \cdot f(c) < 0$ , set  $b = c$ ; else, set  $a = c$ .
- Repeat until  $|f(c)|$  or  $|b - a|$  is less than the tolerance.

### 3.3 Newton–Raphson Method

- Choose an initial guess  $x_0$ .
- Update the root approximation using

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

- Repeat until  $|x_{i+1} - x_i|$  or  $|f(x_{i+1})|$  is less than the tolerance.

### 3.4 Secant Method

1. Choose two initial guesses  $x_0$  and  $x_1$ .
2. Update the root approximation using

$$x_{i+1} = x_i - f(x_i) \frac{x_i - x_{i-1}}{f(x_i) - f(x_{i-1})}$$

3. Repeat until  $|x_{i+1} - x_i| \leq \text{tolerance}$ .

## 4 Implementation

### 4.1 Bisection Method

```
import math
import pandas as pd
import matplotlib.pyplot as plt

def f(h):
    return h**3 - 10*h + 5*math.exp(-h/2) - 2

# Bisection Method
def bisection_verbose(a, b, tol=0.001, N=200, round_n=5):
    #Check if root is guaranteed
    if f(a) * f(b) >= 0:
        print(f"No root guaranteed in the interval [{a}, {b}] .")
        return None

    rows = []
    ea_plot = []
    print("Iter |     xl      |     xu      |     xr      |     f(xr)      |     ea(%)")
    print("-----")
    for k in range(1, N + 1):
        #midpoint
        xr = round((a + b) / 2, round_n)
        fxr = round(f(xr), round_n)

        # Absolute error
        ea = abs(b - a) * 100 if k > 1 else None

        # Print current iteration
        if ea is not None:
```

```

        print(f"{{k:4d} | {a:8.5f} | {b:8.5f} | {xr:9.5f} | {fxr:11.5f} | {ea:8.5f}")
    else:
        print(f"{{k:4d} | {a:8.5f} | {b:8.5f} | {xr:9.5f} | {fxr:11.5f} | {'----'")

    # Store for table and plotting
    rows.append({
        "iter": k,
        "xl": round(a, round_n),
        "xu": round(b, round_n),
        "xr": xr,
        "f(xr)": fxr,
        "ea(%)": round(ea, 5) if ea is not None else None
    })
    if ea is not None:
        ea_plot.append((k, ea))

    #check convergence
    if fxr == 0 or abs(b - a) < tol:
        print("-----")
        print(f"Approximate root after {k} iterations: xr = {xr:.5f}")
        df = pd.DataFrame(rows)
        return df, ea_plot, xr

    #update interval
    if f(a) * f(xr) < 0:
        b = xr
    else:
        a = xr

    #reached max iterations
    print("-----")
    print(f"Stopped after {N} iterations. Approximate root {xr:.5f}")
    df = pd.DataFrame(rows)
    return df, ea_plot, xr

# Run the Bisection Method
a, b = 0.1, 0.4
tol = 0.001
N = 50

df_bi, ea_plot_bi, root_bi = bisection_verbose(a, b, tol, N, round_n=5)

# Save iteration table
df_bi.to_csv("bisection_iterations.csv", index=False)

# Plot Error vs Iteration
if len(ea_plot_bi) > 0:

```

```

its = [it for it, e in ea_plot_bi]
eas = [e for it, e in ea_plot_bi]

# Linear scale
plt.figure(figsize=(7,4.5))
plt.plot(its, eas, marker='o')
plt.xlabel("Iteration")
plt.ylabel("Absolute Error ea (%)")
plt.title("Bisection Method: ea(%) vs Iteration (linear scale)")
plt.grid(True)
plt.savefig("bisection_ea_linear.png")
plt.show()

# Final Output Summary
print("\nFinal Summary:")
print(f"Approximate root = {root_bi:.5f}")
print(f"Iteration data saved in: bisection_iterations.csv")
print("Error plots saved as: bisection_ea_linear.png")

```

## 4.2 False Position Method

```

import math, pandas as pd, matplotlib.pyplot as plt

def f(h):
    return h**3 - 10*h + 5*math.exp(-h/2) - 2

def false_position_verbose(a_init, b_init, tol_percent=0.001, round_n=5, max_iter=500):
    a = round(a_init, round_n)
    b = round(b_init, round_n)
    fa = round(f(a), round_n)
    fb = round(f(b), round_n)
    if fa * fb > 0:
        raise ValueError(f"Initial interval [{a},{b}] does not bracket a root.")

    rows = []
    prev_c = None
    ea_plot = []
    final = None

    for itr in range(1, max_iter+1):
        denom = (fb - fa)
        if denom == 0:
            raise ZeroDivisionError("Denominator zero in false position formula.")
        c_raw = (a * fb - b * fa) / denom
        c = round(c_raw, round_n)
        fc = round(f(c), round_n)

        rows.append([itr, a, b, fa, fb, c, fc])
        ea_plot.append(abs(fc))

        if abs(fc) < tol_percent:
            final = c
            break
        if prev_c is not None and abs(c - prev_c) < tol_percent:
            final = c
            break
        prev_c = c
    df = pd.DataFrame(rows, columns=['Iteration', 'a', 'b', 'fa', 'fb', 'c', 'fc'])
    df['ea'] = ea_plot
    return df, final

```

```

    if prev_c is None:
        ea = None
    else:
        ea = abs((c - prev_c) / c) * 100.0 if c != 0 else float('inf')

    rows.append({
        "iter": itr,
        "xl": f"{a:.5f}",
        "xu": f"{b:.5f}",
        "xr": f"{c:.5f}",
        "f(xr)": f"{fc:.5f}",
        "ea(%)": f"{ea:.5f}" if ea is not None else ""
    })
    if ea is not None:
        ea_plot.append((itr, ea))

    if fc == 0 or (ea is not None and ea <= tol_percent):
        final = {"iterations": itr, "root": c, "f_root": fc, "final_ea": ea}
        break

    if fa * fc < 0:
        b, fb = c, fc
    else:
        a, fa = c, fc

    a = round(a, round_n)
    b = round(b, round_n)
    prev_c = c
else:
    final = {"iterations": max_iter, "root": c, "f_root": fc, "final_ea": (ea_plot[-1][1] if ea_plot else None)}

df = pd.DataFrame(rows)
return df, ea_plot, final

# Run
df_fp, ea_plot_fp, final_fp = false_position_verbose(0.1, 0.4, tol_percent=0.001, round_n=5)
df_fp.to_csv("false_position_iterations.csv", index=False)

# Print table
print(df_fp.to_string(index=False))

# Plot (linear)
if ea_plot_fp:
    its = [it for it,e in ea_plot_fp]; eas = [e for it,e in ea_plot_fp]
    plt.figure(figsize=(7,4.5))
    plt.plot(its, eas, marker='o')
    plt.xlabel("Iteration"); plt.ylabel("Approx. relative error ea (%)")
    plt.title("False Position: ea(%) vs Iteration (linear scale)"); plt.grid(True)

```

```
plt.savefig("falsepos_ea_linear.png"); plt.show()
```

### 4.3 Newton–Raphson Method

```
import math
import pandas as pd
import matplotlib.pyplot as plt

# Define the function and its derivative
def f(h):
    return h**3 - 10*h + 5*math.exp(-h/2) - 2

def f_prime(h):
    return 3*h**2 - 10 - (5/2)*math.exp(-h/2)

def newton_raphson_verbose(x0, tol_percent=0.001, round_n=5, max_iter=200):
    xi = round(x0, round_n)
    rows = []
    ea_plot = []
    prev_x = None
    final = None

    for itr in range(1, max_iter+1):
        fxi = round(f(xi), round_n)
        fpxi = round(f_prime(xi), round_n)

        if fpxi == 0:
            raise ZeroDivisionError(f"Derivative zero at iteration {itr}, x={xi}")

        # Newton–Raphson formula
        x_next_raw = xi - fxi / fpxi
        x_next = round(x_next_raw, round_n)
        fx_next = round(f(x_next), round_n)

        if prev_x is None:
            ea = None
        else:
            ea = abs((x_next - prev_x) / x_next) * 100 if x_next != 0 else float('inf')

        rows.append({
            "iter": itr,
            "xi": f"{xi:.5f}",
            "f(xi)": f"{fxi:.5f}",
            "f'(xi)": f"{fpxi:.5f}",
            "xi+1": f"{x_next:.5f}",
            "f(xi+1)": f"{fx_next:.5f}",
            "ea(%)": f"{ea:.5f}" if ea is not None else ""})
```

```

    })

    if ea is not None:
        ea_plot.append((itr, ea))

    if fx_next == 0 or (ea is not None and ea <= tol_percent):
        final = {"iterations": itr, "root": x_next, "f_root": fx_next, "final_ea": ea}
        break

    prev_x = x_next
    xi = x_next

else:
    final = {"iterations": max_iter, "root": x_next, "f_root": fx_next, "final_ea": ea}

df_table = pd.DataFrame(rows)
return df_table, ea_plot, final

df_nr, ea_plot_nr, final_nr = newton_raphson_verbose(x0=1.5, tol_percent=0.001, round=2)

df_nr.to_csv("newton_raphson_iterations.csv", index=False)

print(df_nr.to_string(index=False))

# Plot Error vs Iteration
if len(ea_plot_nr) > 0:
    its = [it for it, e in ea_plot_nr]
    eas = [e for it, e in ea_plot_nr]

    # Linear scale
    plt.figure(figsize=(7,4.5))
    plt.plot(its, eas, marker='o')
    plt.xlabel("Iteration")
    plt.ylabel("Approx. relative error ea (%)")
    plt.title("Newton-Raphson: ea(%) vs Iteration (linear scale)")
    plt.grid(True)
    plt.savefig("newton_ea_linear.png")
    plt.show()

print("\nFinal Summary:")
print(f"Converged in {final_nr['iterations']} iterations.")
print(f"Approximate root = {final_nr['root']:.5f}")
print(f"f(root) = {final_nr['f_root']:.5f}")
if final_nr['final_ea'] is not None:

```

```

    print(f"Final approximate relative error ea(%) = {final_nr['final_ea']:.7f}")
else:
    print("Final approximate relative error ea(%) = N/A")

```

## 4.4 Secant Method

```

import math
import pandas as pd
import matplotlib.pyplot as plt

# Define the function
def f(h):
    return h**3 - 10*h + 5*math.exp(-h/2) - 2

def secant_verbose(x0, x1, tol_percent=0.001, round_n=5, max_iter=200):
    x_prev = round(x0, round_n)
    x_curr = round(x1, round_n)
    rows = []
    ea_plot = []
    final = None

    for itr in range(1, max_iter + 1):
        f_prev = round(f(x_prev), round_n)
        f_curr = round(f(x_curr), round_n)

        # Avoid division by zero
        if f_curr - f_prev == 0:
            raise ZeroDivisionError(f"Division by zero at iteration {itr}")

        # Secant formula
        x_next_raw = x_curr - f_curr * (x_curr - x_prev) / (f_curr - f_prev)
        x_next = round(x_next_raw, round_n)
        f_next = round(f(x_next), round_n)

        # Approximate relative error
        if itr == 1:
            ea = None
        else:
            ea = abs((x_next - x_curr) / x_next) * 100 if x_next != 0 else float('inf')

        rows.append({
            "iter": itr,
            "x(i-1)": f"{x_prev:.5f}",
            "x(i)": f"{x_curr:.5f}",
            "f(x(i-1))": f"{f_prev:.5f}",
            "f(x(i))": f"{f_curr:.5f}",
            "x(i+1)": f"{x_next:.5f}",
        })

```

```

        "f(x(i+1))": f"{f_next:.5f}",
        "ea)": f"{ea:.5f}" if ea is not None else ""
    })

    if ea is not None:
        ea_plot.append((itr, ea))

    # Stop condition
    if f_next == 0 or (ea is not None and ea <= tol_percent):
        final = {"iterations": itr, "root": x_next, "f_root": f_next, "final_ea": ea}
        break

    # Update for next iteration
    x_prev, x_curr = x_curr, x_next

else:
    final = {"iterations": max_iter, "root": x_next, "f_root": f_next, "final_ea": ea}

df_table = pd.DataFrame(rows)
return df_table, ea_plot, final

df_sc, ea_plot_sc, final_sc = secant_verbose(x0=1.5, x1=2.0, tol_percent=0.001, round=3)

df_sc.to_csv("secant_iterations.csv", index=False)

print(df_sc.to_string(index=False))

# Plot Error vs Iteration
if len(ea_plot_sc) > 0:
    its = [it for it, e in ea_plot_sc]
    eas = [e for it, e in ea_plot_sc]

    # Linear scale
    plt.figure(figsize=(7,4.5))
    plt.plot(its, eas, marker='o')
    plt.xlabel("Iteration")
    plt.ylabel("Approx. relative error ea (%)")
    plt.title("Secant Method: ea(%) vs Iteration (linear scale)")
    plt.grid(True)
    plt.savefig("secant_ea_linear.png")
    plt.show()

print("\nFinal Summary:")
print(f"Converged in {final_sc['iterations']} iterations.")
print(f"Approximate root = {final_sc['root']:.5f}")
print(f"f(root) = {final_sc['f_root']:.5f}")

```

```

if final_sc['final_ea'] is not None:
    print(f"Final approximate relative error ea(%) = {final_sc['final_ea']:.7f}")
else:
    print("Final approximate relative error ea(%) = N/A")

```

## 4.5 Combined Convergence Plot

```

import math
import matplotlib.pyplot as plt

# Define the function and derivative
def f(x):
    return x**3 - 10*x + 5*math.exp(-x/2) - 2

def f_prime(x):
    return 3*x**2 - 10 - (5/2)*math.exp(-x/2)

# 1 Bisection Method
def bisection(a, b, tol=0.001, max_iter=100):
    ea_list = []
    if f(a) * f(b) >= 0:
        return ea_list
    for i in range(1, max_iter + 1):
        c = (a + b) / 2
        ea = abs(b - a) * 100 if i > 1 else None
        if ea is not None:
            ea_list.append((i, ea))
        if f(c) == 0 or abs(b - a) < tol:
            break
        if f(a) * f(c) < 0:
            b = c
        else:
            a = c
    return ea_list

# 2 False Position Method
def false_position(a, b, tol=0.001, max_iter=100):
    ea_list = []
    fa, fb = f(a), f(b)
    if fa * fb >= 0:
        return ea_list
    prev_c = None
    for i in range(1, max_iter + 1):
        c = (a * fb - b * fa) / (fb - fa)
        fc = f(c)

```

```

if prev_c is not None:
    ea = abs((c - prev_c) / c) * 100
    ea_list.append((i, ea))
    if ea <= tol:
        break
if fa * fc < 0:
    b, fb = c, fc
else:
    a, fa = c, fc
prev_c = c
return ea_list

# 3 Newton-Raphson Method
def newton_raphson(x0, tol=0.001, max_iter=100):
    ea_list = []
    x = x0
    for i in range(1, max_iter + 1):
        fx, dfx = f(x), f_prime(x)
        if dfx == 0:
            break
        x_new = x - fx / dfx
        ea = abs((x_new - x) / x_new) * 100 if i > 1 else None
        if ea is not None:
            ea_list.append((i, ea))
        if ea <= tol:
            break
        x = x_new
    return ea_list

# 4 Secant Method
def secant(x0, x1, tol=0.001, max_iter=100):
    ea_list = []
    prev, curr = x0, x1
    for i in range(1, max_iter + 1):
        f_prev, f_curr = f(prev), f(curr)
        if f_curr - f_prev == 0:
            break
        next_x = curr - f_curr * (curr - prev) / (f_curr - f_prev)
        ea = abs((next_x - curr) / next_x) * 100 if i > 1 else None
        if ea is not None:
            ea_list.append((i, ea))
        if ea <= tol:
            break
        prev, curr = curr, next_x
    return ea_list

```

```

bisection_errors = bisection(0.1, 0.4)
falsepos_errors = false_position(0.1, 0.4)
newton_errors = newton_raphson(1.5)
secant_errors = secant(1.5, 2.0)

# Plot Comparison (Linear Scale)
plt.figure(figsize=(8,6))

plt.plot([i for i, e in bisection_errors], [e for i, e in bisection_errors],
         marker='o', label="Bisection", linewidth=2)
plt.plot([i for i, e in falsepos_errors], [e for i, e in falsepos_errors],
         marker='s', label="False Position", linewidth=2)
plt.plot([i for i, e in newton_errors], [e for i, e in newton_errors],
         marker='^', label="Newton{Raphson", linewidth=2)
plt.plot([i for i, e in secant_errors], [e for i, e in secant_errors],
         marker='d', label="Secant", linewidth=2)

plt.xlabel("Iteration")
plt.ylabel("Approx. Relative Error ea (%)")
plt.title("Convergence Rate Comparison of Root-Finding Methods (Linear Scale)")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.savefig("convergence_comparison_linear.png")
plt.show()

print(" Linear-scale comparison plot saved as: convergence_comparison_linear.png")

```

## 5 Output

## 5.1 Bisection Method

```

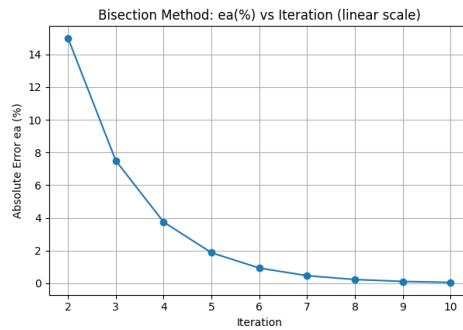
PS C:\Users\user\OneDrive\Desktop\numerical> python -u "C:\Users\user\OneDrive\Desktop\numerical\prob1.py"
Iterations | x | f(x) | % error
-----|---|---|---
1 | 0.180000 | 0.480000 | 0.250000 | -0.7189 | -----
2 | 0.180000 | 0.250000 | 0.175000 | 0.83645 | 15.000000
3 | 0.180000 | 0.095000 | 0.095000 | 0.93333 | 0.000000
4 | 0.180000 | 0.250000 | 0.23125 | 1.15391 | 3.75000
5 | 0.23125 | 0.250000 | 0.244863 | 0.68484 | 1.875000
6 | 0.240625 | 0.250000 | 0.24532 | -0.01561 | 0.937500
7 | 0.240625 | 0.24532 | 0.24297 | 0.01267 | 0.468750
8 | 0.240625 | 0.24297 | 0.24356 | -0.05375 | 0.187500
9 | 0.24297 | 0.24415 | 0.24356 | 0.06556 | 0.118600
10 | 0.24356 | 0.24415 | 0.24385 | 0.00287 | 0.059600

Approximate root after 10 iterations: xr = 0.24385

Final Summary:
Approximate root = 0.24385
Iteration data saved in: bisection_iterations.csv
Error plots saved as: bisection_e_linear.png and bisection_ea_log.png
PS C:\Users\user\OneDrive\Desktop\numerical>

```

(a) Output



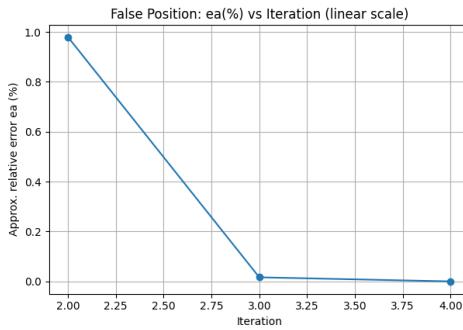
(b) Convergence Graph

Figure 2: Bisection Method Results

## 5.2 False Position Method

```
PS C:\Users\user\OneDrive\Desktop\numerical> python u "c:\Users\user\OneDrive\Desktop\numerical\prob1.py"
   iter    x1      x2      x3      f(x)    es(%)
1  0.38000  0.40000  0.24645  0.82300  0.00000
2  0.38000  0.24645  0.24645  0.82300  0.97297
3  0.38000  0.24646  0.00000  0.00000  0.01639
4  0.24402  0.24406  0.24402  0.00000  0.00000
PS C:\Users\user\OneDrive\Desktop\numerical>
```

(a) Output



(b) Convergence Graph

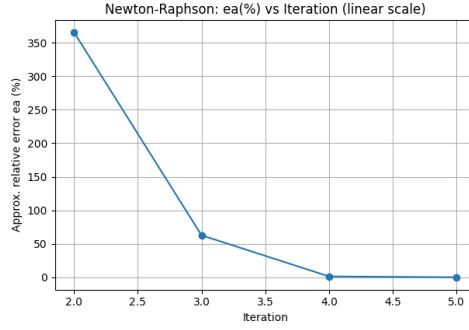
Figure 3: False Position Method Results

### 5.3 Newton–Raphson Method

```
PS C:\Users\User\OneDrive\Desktop\numerical> python -u "c:\Users\User\OneDrive\Desktop\numerical\prob3.py"
iter   xi     f(xi)    f'(xi)  xi+1   f(xi+1)   ea(%)
1  1.50000 -11.26317 -4.43092 -1.84195 15.70664  1.50000
2  2.00000 -11.75156 -11.59398 0.24108 0.03542  965.66754
3  0.39216 -1.75156 -11.59398 0.24108 0.03542  62.66799
4  0.24108  0.03542 -12.04175 0.24402 0.00003  1.23482
5  0.24402  0.00003 -12.03421 0.24402 0.00003  0.00000

Final Summary:
Converged in 5 iterations.
Approximate root = 0.24402
f(root) = 0.00003
Final approximate relative error ea(%) = 0.0000000
PS C:\Users\User\OneDrive\Desktop\numerical>
```

(a) Output



(b) Convergence Graph

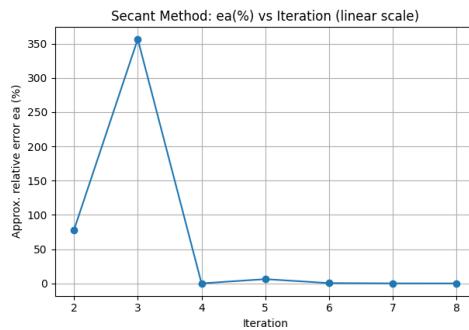
Figure 4: Newton–Raphson Method Results

### 5.4 Secant Method

```
PS C:\Users\User\OneDrive\Desktop\numerical> python -u "c:\Users\User\OneDrive\Desktop\numerical\prob4.py"
iter   x(i-1)      x(i)     f(x(i-1))   f(x(i))  xi     f(xi)   ea(%)
1  1.50000  2.00000 -11.26317 -12.16669 -4.77524 -8.69925  1.50000
2  2.00000  2.77581 -12.16669 -12.04969 -4.77524 -8.73811  78.09838
3  1.19224  2.34754 -11.75156 -11.59398 -4.77524 -8.71783 356.53282
4  -21.89314 -4.77581 261158.75946 -8.71703 -4.77638 -8.73481 0.01193
5  -4.77581 -4.77638 -8.71703 -8.73481 -4.49636 -0.58803 6.22770
6  -4.77638 -4.49636 -8.73481 -0.58803 -4.47615 -0.04595 0.45150
7  -4.49636 -4.47615 -0.58803 -0.04595 -4.47444 -0.00036 0.05822
8  -4.47615 -4.47444 -0.04595 -0.00036 -4.47444 -0.00010 0.00022

Final Summary:
Converged in 8 iterations.
Approximate root = -4.47443
f(root) = -0.00010
Final approximate relative error ea(%) = 0.0002235
PS C:\Users\User\OneDrive\Desktop\numerical>
```

(a) Output



(b) Convergence Graph

Figure 5: Secant Method Results

## 5.5 Combined Convergence Plot

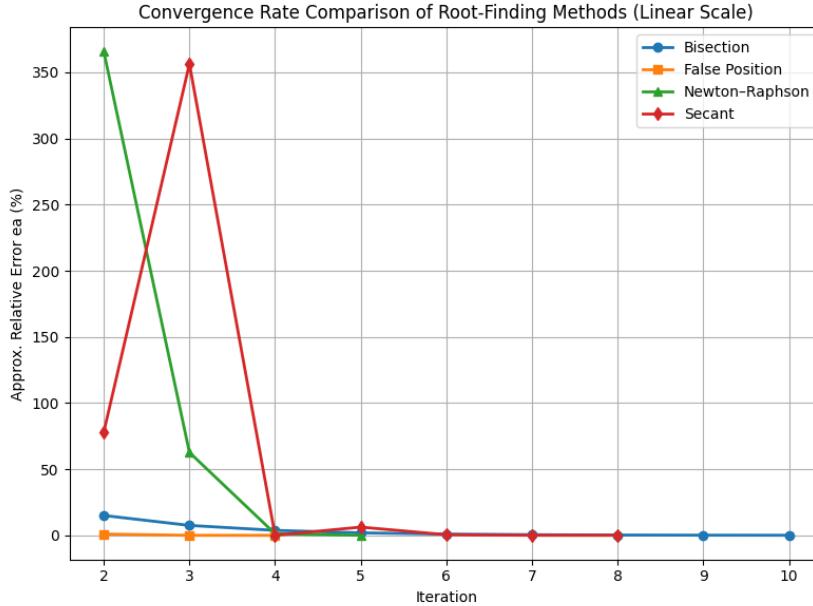


Figure 6: Error vs Iteration for All Methods

## 6 Summary

In this experiment, four root-finding algorithms — Bisection, False Position, Newton–Raphson, and Secant — were successfully implemented to solve a nonlinear equation. The numerical results demonstrated the following trends:

- **Bisection Method:** Slow but guaranteed convergence.
- **False Position Method:** Faster than Bisection but can stagnate near certain root configurations.
- **Newton–Raphson Method:** Fastest convergence due to the use of derivative information but sensitive to initial guess.
- **Secant Method:** Nearly as fast as Newton–Raphson, derivative-free, and more computationally efficient.

The Newton–Raphson method provided the most rapid and accurate convergence, while the Bisection method served as the most reliable fallback approach. This comparative analysis highlights that the optimal root-finding method depends on the function characteristics, desired accuracy, and available derivative information.