Department of Computer Science and Engineering

University of Dhaka

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

---

CSE 3212: Numerical Analysis Lab

Batch: 28 / 3rd Year 2nd Semester 2024

Submitted to:
Dr. Muhammad Ibrahim
Palash Roy

Submitted by:
Farzana Tasnim (14)

Contents

1 Introduction

1.1 Bisection Method

The **Bisection Method** is one of the simplest techniques to find a zero of a nonlinear function. It is also called the *interval halving method*. To use the Bisection method, one needs an initial interval that is known to contain a zero of the function. The method systematically reduces the interval by dividing it into two equal parts, performing a simple test, and based on the result, discarding half of the interval. The procedure is repeated until the desired interval size or accuracy is obtained.

If $f(x)$ is continuous on $[a, b]$ and $f(a) \cdot f(b) < 0$, then there exists at least one root $c \in (a, b)$ such that $f(c) = 0$.

1.2 False Position (Regula Falsi) Method

The **False Position Method** is an improvement of the Bisection Method. Like Bisection, it requires an initial interval $[a, b]$ such that $f(a) \cdot f(b) < 0$. Instead of halving the interval, this method uses a straight line between the points $(a, f(a))$ and $(b, f(b))$ and takes the x-intercept of that line as the new approximation:

$$c = \frac{af(b) - bf(a)}{f(b) - f(a)}$$

The process continues by updating either a or b, depending on the sign of $f(c)$.

1.3 Newton–Raphson Method

The **Newton–Raphson Method** uses tangents to approximate the root. Starting with an initial guess x_0, the next approximation is given by:

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

This method converges quadratically near the root, provided that $f'(x) \neq 0$ and the initial guess is sufficiently close to the actual root.

1.4 Secant Method

The **Secant Method** is similar to the Newton–Raphson method but does not require explicit derivative calculation. It approximates the derivative

using two nearby points:

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

This method is especially useful when $f'(x)$ is difficult or expensive to compute.

1.5 Comparison Table

Method	**Type**	**Derivative**	**Convergence**	**Speed**
Bisection	Bracketing	No	Linear	Slow
False Position	Bracketing	No	Linear	Moderate
Newton–Raphson	Open	Yes	Quadratic	Very Fast
Secant	Open	No	Superlinear	Moderate

Table 1: Comparison of Root Finding Methods

2 Objectives

- To understand the principles of the Bisection, False Position, Newton–Raphson, and Secant methods.
- To implement the iterative algorithms in a computer program.
- To approximate the root of a nonlinear function to a desired accuracy.
- To compare convergence behavior and limitations of these methods.

3 Algorithms

3.1 Bisection Method

Algorithm 1 Bisection Method

Require: Function $f(x)$, interval $[a, b]$, tolerance ε

if $f(a) \cdot f(b) \geq 0$ **then**
  Print "No root guaranteed in [a, b]" and stop.
end if
Set $k = 0$
repeat
  $k \leftarrow k + 1$
  $x_r = \dfrac{a + b}{2}$
  Compute $f(x_r)$
  $e_a = \left| \dfrac{x_r - x_{r,old}}{x_r} \right| \times 100$
  if $f(a) \cdot f(x_r) < 0$ **then**
    $b \leftarrow x_r$
  else
    $a \leftarrow x_r$
  end if
  $x_{r,old} \leftarrow x_r$
until $e_a < \varepsilon$ or $f(x_r) = 0$
return x_r

3.2 False Position (Regula Falsi) Method

Algorithm 2 False Position Method

Require: Function $f(x)$, interval $[a, b]$, tolerance ε

Set $k = 0$
repeat
  $k \leftarrow k + 1$
  Compute $f(a)$ and $f(b)$
  $x_r = \dfrac{af(b) - bf(a)}{f(b) - f(a)}$
  Compute $f(x_r)$
  $e_a = \left| \dfrac{x_r - x_{r,old}}{x_r} \right| \times 100$
  if $f(a) \cdot f(x_r) < 0$ **then**
    $b \leftarrow x_r$
  else
    $a \leftarrow x_r$
  end if
  $x_{r,old} \leftarrow x_r$
until $e_a < \varepsilon$ or $f(x_r) = 0$
return x_r

3.3 Newton–Raphson Method

Algorithm 3 Newton–Raphson Method

Require: Function $f(x)$, derivative $f'(x)$, initial guess x_0, tolerance ε

Set $k = 0$
repeat
  $k \leftarrow k + 1$
  $x_{k+1} = x_k - \dfrac{f(x_k)}{f'(x_k)}$
  $e_a = \left| \dfrac{x_{k+1} - x_k}{x_{k+1}} \right| \times 100$
  $x_k \leftarrow x_{k+1}$
until $e_a < \varepsilon$ or $f(x_{k+1}) = 0$
return x_{k+1}

3.4 Secant Method

Algorithm 4 Secant Method

Require: Function $f(x)$, initial guesses x_0, x_1, tolerance ε

Set $k = 0$
repeat
  $k \leftarrow k + 1$
  Compute $f(x_0)$ and $f(x_1)$
  $x_2 = x_1 - f(x_1)\dfrac{x_1 - x_0}{f(x_1) - f(x_0)}$
  $e_a = \left|\dfrac{x_2 - x_1}{x_2}\right| \times 100$
  Update $x_0 \leftarrow x_1$, $x_1 \leftarrow x_2$
until $e_a < \varepsilon$ or $f(x_2) = 0$
return x_2

4 Implementation

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

Figure 1: Define Function

```python
def isBisectionApplicable(a, b):
    return f(a) * f(b) < 0

# ---------------- Bisection ----------------
def Bisection(a, b, eps_relative):
    if not isBisectionApplicable(a, b):
        print("f(a) * f(b) >= 0. No root guaranteed in [a, b].")
        return None
    print("\n=== Bisection Method ===")
    print(f"{'Iter':<6} {'xl':>10} {'xu':>10} {'xr':>10} {'f(xr)':>12} {'ea(%)':>12}")
    print("-"*65)

    iteration_data = []
    xr_old = a
    k = 0

    while True:
        k += 1
        xr = (a + b) / 2
        fxr = f(xr)
        ea = abs((xr - xr_old)/xr) * 100
        iteration_data.append([k, ea])

        print(f"{k:<6d} {a:>10.5f} {b:>10.5f} {xr:>10.5f} {fxr:>12.5f} {ea:>12.5f}")
        if f(xr) == 0 or ea < eps_relative:
            break
        if f(a) * fxr < 0:
            b = xr
        else:
            a = xr
        xr_old = xr
    return [row[0] for row in iteration_data], [row[1] for row in iteration_data]
```

Figure 2: Bisection implementation

```python
# ---------------- False Position ----------------
def FalsePosition(a, b, eps_relative):
    print("\n=== False Position Method ===")
    print(f"{'Iter':>6} {'xl':>10} {'xu':>14} {'xr':>10} {'f(xr)':>10} {'ea(%)':>12}")
    print("-"*70)

    iteration_data = []
    xr_old = b
    k = 0
    while True:
        k += 1
        fa, fb = f(a), f(b)
        xr = (a*fb - b*fa) / (fb - fa)
        fxr = f(xr)
        ea = abs((xr - xr_old)/xr) * 100 if k > 1 else None
        if ea is not None:
            iteration_data.append([k, ea])

        ea_str = f"{ea:>12.5f}" if ea is not None else f"{'N/A':>12}"
        print(f"{k:<6d} {a:>12.5f} {b:>14.5f} {xr:>10.5f} {fxr:>12.5f} {ea_str}")

        if ea is not None and ea < eps_relative:
            break
        if fa * fxr < 0:
            b = xr
        else:
            a = xr
        xr_old = xr
    return [row[0] for row in iteration_data], [row[1] for row in iteration_data]
```

Figure 3: False position implementation

```python
def df(h):
    return 3*h**2 - 10 - (5/2)*np.exp(-h/2)
```

Figure 4: Derivative Function

```python
# ---------------- Newton-Raphson ----------------
def newton_raphson(x0, tol=0.001, max_iter=100):
    print("\n=== Newton-Raphson Method ===")
    print(f"{'Iter':>6} {'x':>12} {'f(x)':>14} {'f(x)':>14} {'ea(%)':>12}")
    print("-"*70)

    ea_list, iter_list = [], []
    x = x0
    for i in range(1, max_iter+1):
        fx, dfx = f(x), df(x)
        if dfx == 0:
            break
        x_new = x - fx/dfx
        ea = abs((x_new - x)/x_new) * 100
        ea_list.append(ea)
        iter_list.append(i)

        print(f"{i:>6d} {x_new:>12.5f} {fx:>14.5f} {dfx:>14.5f} {ea:>12.5f}")
        if ea < tol:
            break
        x = x_new
    return iter_list, ea_list
```

Figure 5: Newton-Raphson implementation

```python
# ---------------- Secant ----------------
def secant_method(x0, x1, tol=0.001, max_iter=100):
    print("\n=== Secant Method ===")
    print(f"{'Iter':>6} {'x0':>12} {'x1':>12} {'x2':>12} {'f(x2)':>14} {'ea(%)':>12}")
    print("-"*80)

    ea_list, iter_list = [], []
    for i in range(1, max_iter+1):
        f0, f1 = f(x0), f(x1)
        denom = f1 - f0
        if denom == 0:
            break
        x2 = x1 - f1*(x1 - x0)/denom
        fx2 = f(x2)
        ea = abs((x2 - x1)/x2) * 100
        ea_list.append(ea)
        iter_list.append(i)

        print(f"{i:>6d} {x0:>12.5f} {x1:>12.5f} {x2:>12.5f} {fx2:>14.5f} {ea:>12.5f}")
        if ea < tol:
            break
        x0, x1 = x1, x2
    return iter_list, ea_list
```

Figure 6: Secant implementation

5 Output

```
=== Bisection Method ===
Iter           xl         xu         xr        f(xr)        ea(%)
------------------------------------------------------------------
1         0.10000    0.40000    0.25000     -0.07189     60.00000
2         0.10000    0.25000    0.17500      0.83645     42.85714
3         0.17500    0.25000    0.21250      0.38059     17.64706
4         0.21250    0.25000    0.23125      0.15391      8.10811
5         0.23125    0.25000    0.24063      0.04090      3.89610
6         0.24063    0.25000    0.24531     -0.01552      1.91083
7         0.24063    0.24531    0.24297      0.01268      0.96463
8         0.24297    0.24531    0.24414     -0.00142      0.48000
9         0.24297    0.24414    0.24355      0.00563      0.24058
10        0.24355    0.24414    0.24385      0.00210      0.12014
11        0.24385    0.24414    0.24399      0.00034      0.06004
12        0.24399    0.24414    0.24407     -0.00054      0.03001
13        0.24399    0.24407    0.24403     -0.00010      0.01501
14        0.24399    0.24403    0.24401      0.00012      0.00750
15        0.24401    0.24403    0.24402      0.00001      0.00375
16        0.24402    0.24403    0.24403     -0.00005      0.00188
17        0.24402    0.24403    0.24402     -0.00002      0.00094
```

Figure 7: Bisection Output

Bisection is guaranteed to converge if the initial interval brackets the root. It converges linearly, which is why the error gradually decreases over 17 iterations to near zero.

```
=== False Position Method ===
  Iter         xl           xu         xr       f(xr)        ea(%)
------------------------------------------------------------------
1          0.10000        0.40000    0.24645     -0.02920         N/A
2          0.10000        0.24645    0.24406     -0.00040     0.98095
3          0.10000        0.24406    0.24402     -0.00001     0.01340
4          0.10000        0.24402    0.24402     -0.00000     0.00018
```

Figure 8: False position Output

False Position often converges faster than Bisection for functions where one side of the interval is nearly flat. However, convergence may stall if one endpoint does not move significantly. In our output, convergence is achieved in just 4 iterations. Because root value is close to lower interval 0.1.

```
=== Newton-Raphson Method ===
  Iter           x           f(x)          f′(x)        ea(%)
--------------------------------------------------------------------------
     1    -1.04195      -11.26317       -4.43092    243.96086
     2     0.39216       15.70664      -10.95219    365.69568
     3     0.24108       -1.75155      -11.59350     62.66830
     4     0.24402        0.03543      -12.04175      1.20579
     5     0.24402        0.00001      -12.03421      0.00038
```

Figure 9: Newton Raphson Output

Newton-Raphson converges quadratically near the root, making it much faster than Bisection or False Position if the derivative is known and non-zero. The first two iterations show large errors because initial guesses were far from the root, but convergence accelerates once the estimate is closer.

```
=== Secant Method ===
  Iter          x0          x1          x2          f(x2)        ea(%)
-------------------------------------------------------------------------------
     1     1.50000     2.00000    -4.77520       -8.69786    141.88310
     2     2.00000    -4.77520   -21.79342   259857.48459     78.08882
     3    -4.77520   -21.79342    -4.77577       -8.71563    356.33363
     4   -21.79342    -4.77577    -4.77634       -8.73344      0.01195
     5    -4.77577    -4.77634    -4.49644       -0.59009      6.22492
     6    -4.77634    -4.49644    -4.47615       -0.04605      0.45312
     7    -4.49644    -4.47615    -4.47444       -0.00029      0.03837
     8    -4.47615    -4.47444    -4.47443       -0.00000      0.00024
```

Figure 10: Secant Output

Secant method often converges faster than Bisection and does not require derivatives like Newton-Raphson. However, it may be less stable and can diverge if initial guesses are poor. Our output shows a mix of large fluctuations and rapid convergence once near the root.

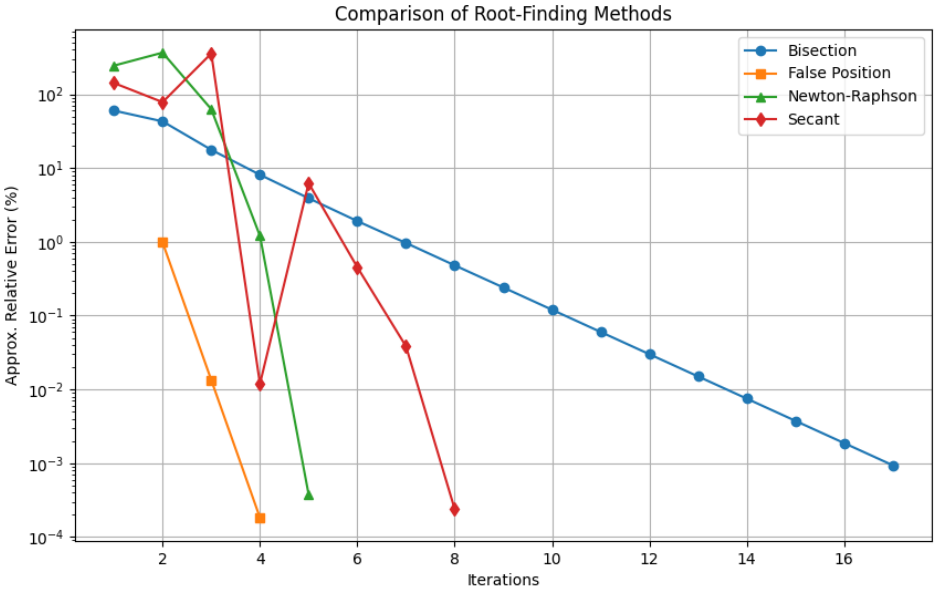

Figure 11: Output in Logarithm scale

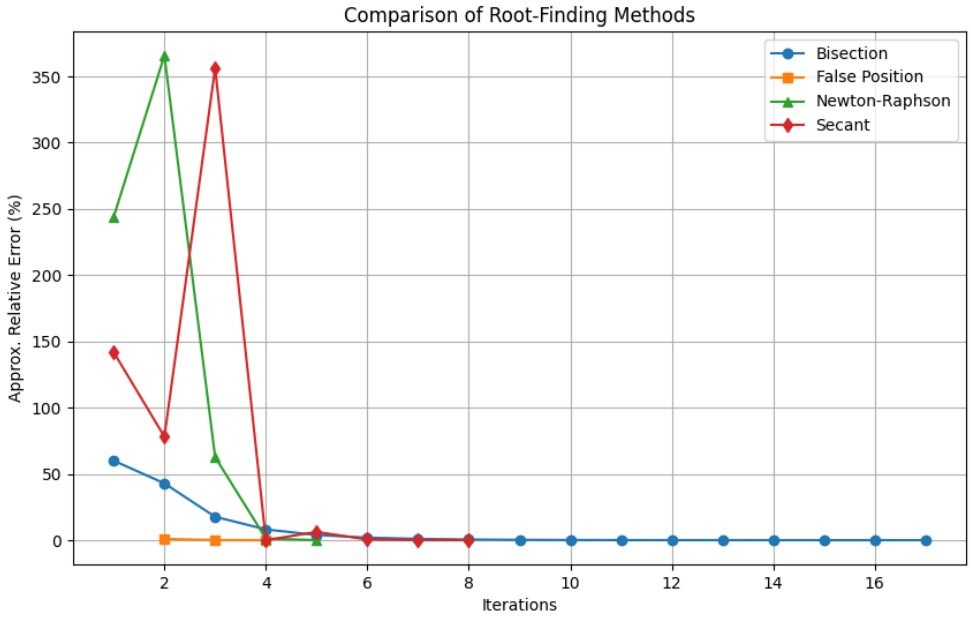

Figure 12: Output in normal scale

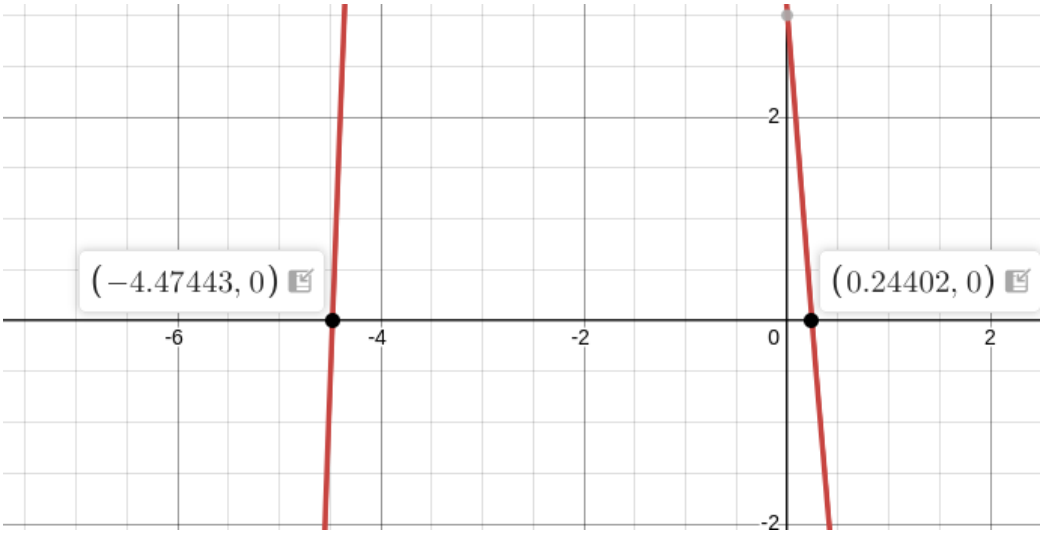

Figure 13: Actual Output Visualization in graph

6 Summary

In this experiment, four numerical methods for finding roots of nonlinear equations were implemented and analyzed: **Bisection, False Position, Newton–Raphson,** and **Secant methods**.

The **Bisection method** is a bracketing method that repeatedly halves the interval where a root is known to exist. It is simple and guarantees convergence, but the convergence rate is relatively slow.

The **False Position method** improves upon Bisection by using linear interpolation to approximate the root. It retains the bracketing property and often converges faster, although in some cases one endpoint may remain fixed, slowing convergence.

The **Newton–Raphson method** is an open method that uses the derivative of the function to approximate the root through tangent lines. It converges quadratically near the root and is very fast when the initial guess is close to the actual root, but it requires the derivative and may diverge if the guess is poor.

The **Secant method** is similar to Newton–Raphson but does not require an explicit derivative, instead approximating it using two previous points. It converges faster than bracketing methods and is practical when derivatives are difficult to compute.

The Python implementation demonstrated the iterative process, approximate relative errors, and convergence behavior. A comparative plot of error

versus iterations showed that Newton–Raphson converges fastest, followed by the Secant method, while Bisection and False Position converge more slowly but reliably.

Overall, this lab illustrates the trade-offs between **speed, reliability, and derivative requirement** in root-finding methods and emphasizes the importance of selecting the appropriate method based on the problem characteristics.