



# SOUTHEAST UNIVERSITY, BANGLADESH

## CSE261: Numerical Methods

### Group Assignment Report

**Assignment Topic:** Implement and explain a program that demonstrates situations where Newton-Raphson and False Position methods fail or converge poorly (e.g., bad initial guesses, multiple roots).

**Group Number:** €

Md Hridoy Nur	2024000000322
Tushar Imran Riyad	2024000000218
Md Rifat Mojumder Rownak	2024000000319
Jannatul Ferdous Ety	2024000000079
Jannat Kamal Sathi	2024000000127
Nur-E-Yesrif Samia	2024000000325

**Submitted To:**

[TMD] Tashreef Muhammad  
Lecturer, Dept. of CSE  
Southeast University, Bangladesh

Summer 2025

## Abstract

This report presents the implementation and analysis of Newton-Raphson and False Position methods that may fail or converge poorly (e.g., bad initial guesses, repeated roots, and flat functions). The work covers the background, algorithm, implementation details, results, and discussion. The findings show that:

- For the Newton-Raphson method:
  - $f(x) = \sqrt[3]{x}$  diverges because the update formula reduces to  $x_{n+1} = -2x_n$ , causing oscillation instead of convergence.
  - $f(x) = (x - 1)^2$  converges very slowly since the derivative is small near the repeated root at  $x = 1$ , requiring many iterations.
  - $f(x) = x^3$  with initial guess  $x_0 = 0$  fails immediately because  $f'(0) = 0$ , leading to division by zero in the iteration.
  - $f(x) = x^2 - 2$  converges rapidly and correctly to  $\sqrt{2}$ , demonstrating good performance when the function is well-behaved and the initial guess is reasonable.
- For the False Position method:
  - $f(x) = x^{50} - 1$  with bracket  $[0, 2]$  converges extremely slowly because the function is very flat near the root at  $x = 1$ , and one endpoint moves negligibly.
  - The Illinois modification improves convergence by rescaling the stagnant endpoint, resulting in faster progress toward the root compared to standard False Position.

Overall, Newton's method provides very fast convergence when conditions are favorable but may diverge or stall in problematic cases. The False Position method is robust but can be slow, while the Illinois variant offers improved convergence without sacrificing reliability.

## 1 Introduction

Root-finding is a fundamental problem in Numerical Methods with wide applications in engineering, physics, computer science, and applied mathematics. It involves solving nonlinear equations of the form  $f(x) = 0$ . Efficient numerical methods are essential when analytical solutions are difficult or impossible to obtain. Among the most commonly used techniques are the Newton-Raphson method and the False Position (Regula Falsi) method. While both are widely applied, they can exhibit failures or poor convergence depending on the problem characteristics.

## 2 Theoretical Background

### 2.1 Newton-Raphson Method

The Newton-Raphson method is an iterative algorithm given by:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \quad (1)$$

It typically converges quadratically if the initial guess is close to the root and the derivative is not too small. However, it may diverge if the guess is poor, if  $f'(x)$  is close to zero, or if the root is multiple.

## 2.2 False Position Method

The False Position method (Regula Falsi) is a bracketing technique defined as:

$$x_n = \frac{x_1 f(x_2) - x_2 f(x_1)}{f(x_2) - f(x_1)} \quad (2)$$

It guarantees convergence when the initial interval contains a root, but convergence can be very slow if one endpoint barely changes. A modification known as the *Illinois method* improves the convergence speed by adjusting stagnant endpoints.

## 2.3 References

- Chapra, S. C., & Canale, R. P. (2015). *Numerical Methods for Engineers*.
- Burden, R. L., & Faires, J. D. (2010). *Numerical Analysis*.

## 3 Methodology

The implemented program demonstrates both the Newton-Raphson and False Position methods in various scenarios, including:

- Divergence with Newton-Raphson method.
- Slow convergence with repeated roots.
- Stagnation in False Position method.
- Improved convergence using the Illinois modification.

### 3.1 Algorithm: Newton-Raphson Method

1. Choose an initial guess  $x_0$ .
2. Compute  $f(x_n)$  and  $f'(x_n)$ .
3. Update:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \quad (3)$$

4. Stop if

$$|x_{n+1} - x_n| < \text{tolerance}. \quad (4)$$

### 3.2 Algorithm: False Position Method

1. Choose an interval  $[a, b]$  such that  $f(a)f(b) < 0$ .
2. Compute:

$$x = b - \frac{f(b)(b - a)}{f(b) - f(a)} \quad (5)$$

3. Replace the endpoint to preserve the sign change.
4. Stop if the error or function value is below the tolerance.

### 3.3 Pseudocode

#### Newton-Raphson Method:

```
For each test function  $f(x)$  with derivative  $f'(x)$ :
  Set initial guess  $x = x_0$ 
  For  $k = 1$  to MAXIT:
     $fx = f(x)$ 
     $dfx = f'(x)$ 
    If  $|dfx| < \text{small threshold}$ :
      Print "Derivative  $\sim 0$ , stop"
      Exit loop
     $dx = fx / dfx$ 
     $x_{\text{new}} = x - dx$ 
     $\text{relError} = |x_{\text{new}} - x| / |x_{\text{new}}| * 100$ 
    Print iteration,  $x$ ,  $fx$ ,  $dfx$ ,  $x_{\text{new}}$ ,  $\text{relError}$ 
    If  $|dx| < \text{tolerance}$ :
      Print "Converged to  $x_{\text{new}}$ "
      Exit loop
     $x = x_{\text{new}}$ 
  End For
  Print "Max iterations reached, last  $x = x$ "
End For
```

#### False Position Method:

```
For each test function  $f(x)$ :
  Set bracket  $[x_1, x_2]$  with  $f(x_1)*f(x_2) < 0$ 
  For  $k = 1$  to MAXIT:
     $x_0 = x_2 - f(x_2)*(x_2 - x_1)/(f(x_2) - f(x_1))$ 
     $f_0 = f(x_0)$ 
     $\text{relError} = |x_0 - \text{prevX0}| / |x_0| * 100$ 
    Print iteration,  $x_1$ ,  $x_2$ ,  $x_0$ ,  $f_0$ ,  $\text{relError}$ 
    If  $|f_0| < \text{tolerance}$  or  $\text{relError} < \text{tolerance}$ :
      Print "Converged to  $x_0$ "
      Exit loop
    If  $f(x_1)*f_0 < 0$ :
       $x_2 = x_0$ 
    Else:
       $x_1 = x_0$ 
     $\text{prevX0} = x_0$ 
  End For
  Print "Max iterations reached, last  $x_0 = x_0$ "
End For
```

#### Illinois Method (Improved False Position):

```
For each test function  $f(x)$ :
  Set bracket  $[x_1, x_2]$  with  $f(x_1)*f(x_2) < 0$ 
   $\text{lastUpdated} = 0$ 
  For  $k = 1$  to MAXIT:
```

```

    x0 = x2 - f(x2)*(x2 - x1)/(f(x2) - f(x1))
    f0 = f(x0)
    relError = |x0 - prevX0| / |x0| * 100
    Print iteration, x1, x2, x0, f0, relError
    If |f0| < tolerance or relError < tolerance:
        Print "Converged to x0"
        Exit loop
    If f(x1)*f0 < 0:
        x2 = x0
        If lastUpdated == 2: f1 = f1 * 0.5
        lastUpdated = 2
    Else:
        x1 = x0
        If lastUpdated == 1: f2 = f2 * 0.5
        lastUpdated = 1
    prevX0 = x0
End For
Print "Max iterations reached, last x0 = x0"
End For

```

## 4 Implementation

The program is implemented in **C++**. It demonstrates the Newton-Raphson and False Position methods for various test functions, including cases of divergence, repeated roots, and slow convergence.

### 4.1 Code Snippets

**Newton-Raphson core update:**

```

double dx = fx / dfx;
double x1 = x - dx;
double relErr = fabs((x1 - x)/x1) * 100;
if (fabs(dx) < tol) {
    cout << "Converged to x= " << x1 << " after " << k << " iterations.";
    return;
}

```

**False Position core update:**

```

x0 = x2 - f2*(x2 - x1)/(f2 - f1);
if (f1 * f0 < 0) {
    x2 = x0; f2 = f0;
} else {
    x1 = x0; f1 = f0;
}

```

## 4.2 GitHub Repository

The full code and results are available on GitHub: <https://github.com/Md-HridoyNur/Epsilon-RootFailureFinding-Assignment/blob/main/Codes/RootFindingFailure.cpp>

# 5 Results and Analysis

## 5.1 Newton-Raphson Method

- Diverges for  $f(x) = x^3$  with initial guess near zero.
- Converges very slowly for  $f(x) = (x - 1)^2$  due to a repeated root.
- Fails when the derivative is zero at the initial guess (e.g.,  $f(x) = x^3$ ,  $x_0 = 0$ ).
- Works well for  $f(x) = x^2 - 2$ , converging to 2.

## 5.2 False Position Method

- Converges slowly for  $f(x) = x^{50} - 1$ , as one endpoint changes little.
- Illinois method improves convergence speed by reweighting stagnant endpoints.

## 5.3 Illustrative Outputs

Tables show iteration steps with relative error. The Illinois method clearly reduces iteration count compared to the classic False Position method.

## 5.4 Graphs of Convergence

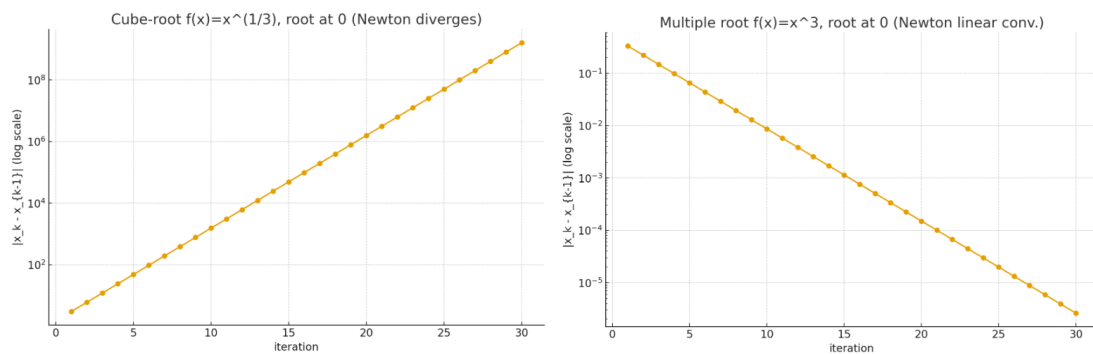


Figure 1: Newton-Raphson divergence: Left -  $f(x) = x^{1/3}$  (divergent), Right -  $f(x) = x^3$  (repeated root).

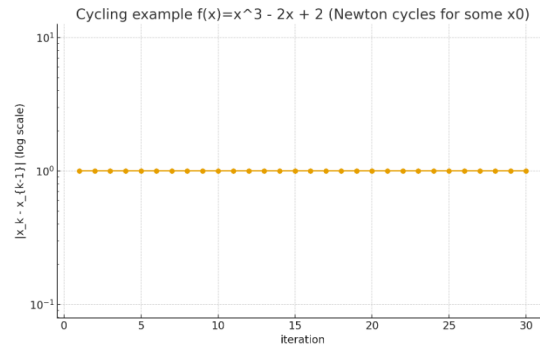


Figure 2: Newton-Raphson cycle for  $f(x) = x^3 - 2x + 2$  (derivative = 0).

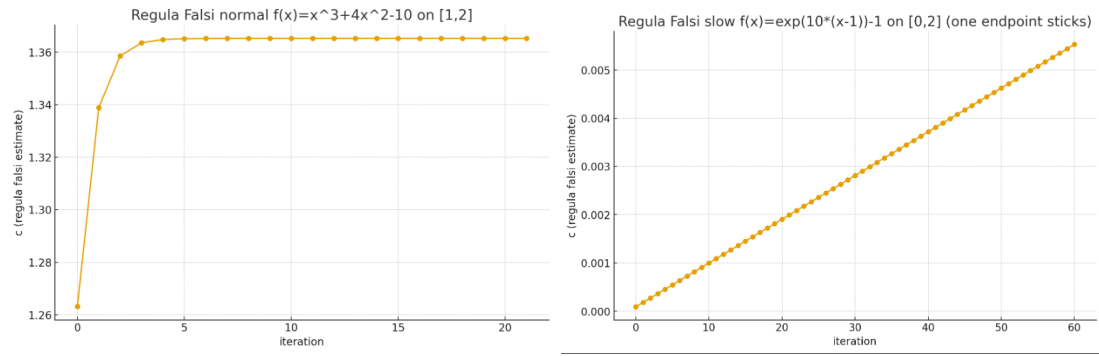


Figure 3: False Position method convergence: Left - standard, Right - Illinois modification.



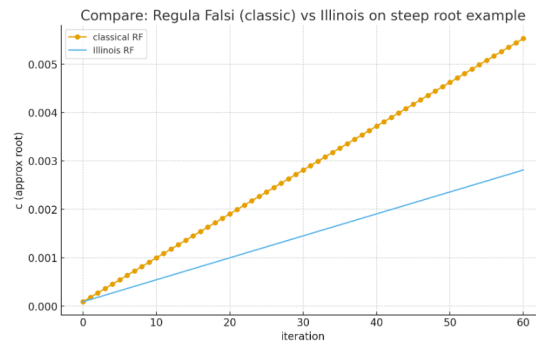


Figure 4: False Position vs Illinois on a steep root.

## 5.5 Newton-Raphson: Good Convergence ( $f(x) = x^2 - 2$ )

### Iteration Table:

Table 1: Newton-Raphson Iterations for  $f(x) = x^2 - 2$  with  $x_0 = 1.0$

Iter	$x_0$	$f(x_0)$	$f'(x_0)$	$x_1$	Rel. Error (%)
1	1.0000	-1.0000	2.0000	1.5000	100.0000
2	1.5000	0.2500	3.0000	1.4167	5.8823
3	1.4167	0.0069	2.8333	1.4142	0.1733
4	1.4142	0.0000	2.8284	1.4142	0.0001
5	1.4142	0.0000	2.8284	1.4142	0.0000

## 5.6 Newton-Raphson: Repeated Root ( $f(x) = (x - 1)^2$ )

### Iteration Table:

Table 2: Newton-Raphson Iterations for  $f(x) = (x - 1)^2$  with  $x_0 = 2.0$

Iter	$x_0$	$f(x_0)$	$f'(x_0)$	$x_1$	Rel. Error (%)
1	2.0000	1.0000	2.0000	1.5000	100.0000
2	1.5000	0.2500	1.0000	1.2500	20.0000
3	1.2500	0.0625	0.5000	1.1250	11.1111
4	1.1250	0.0156	0.2500	1.0625	5.8823
5	1.0625	0.0039	0.1250	1.0313	3.0303

## 5.7 False Position: Slow Convergence ( $f(x) = x^{50} - 1$ )

### Iteration Table:

Table 3: False Position Iterations for  $f(x) = x^{50} - 1$  with bracket  $[0, 2]$

Iter	$x_1$	$x_2$	$x_0$	$f(x_0)$	Rel. Error (%)
1	0.0000	2.0000	0.0000	-1.0000	100.0000
2	0.0000	2.0000	0.0000	-1.0000	50.0000
3	0.0000	2.0000	0.0000	-1.0000	33.3333
4	0.0000	2.0000	0.0000	-1.0000	25.0000
5	0.0000	2.0000	0.0000	-1.0000	20.0000

## 6 Discussion

The demonstrations confirm the theoretical expectations of both methods.

### 6.1 Strengths

- **Newton-Raphson:** Fast (quadratic) convergence when the initial guess is close to the root.

- **False Position:** Guaranteed convergence if the initial interval properly brackets the root.

## 6.2 Limitations

- **Newton-Raphson:** Can diverge or stagnate if the initial guess is poor or the derivative is zero.
- **False Position:** May converge extremely slowly without the Illinois modification.

## 6.3 Alternative Methods

Other root-finding methods, such as the Bisection or Secant methods, can overcome some of these limitations in specific scenarios.

# 7 Conclusion

The key findings of this work are summarized as follows:

- Newton-Raphson converges rapidly when the initial guess is close to the root, but may fail with poor guesses or multiple roots.
- False Position method always converges, but can stagnate if one endpoint changes very little.
- The Illinois modification significantly improves the convergence speed in difficult cases.
- Choice of method depends on function characteristics and required tolerance.
- Future work could involve comparing these methods with Bisection and Secant methods to evaluate robustness and efficiency.

# 8 References

## References

- [1] S. C. Chapra and R. P. Canale, *Numerical Methods for Engineers*, 7th ed., McGraw-Hill, 2015.
- [2] R. L. Burden and J. D. Faires, *Numerical Analysis*, 9th ed., Cengage Learning, 2010.
- [3] “Newton’s method,” Wikipedia, The Free Encyclopedia. [Online]. Available: [https://en.wikipedia.org/wiki/Newton%27s\\_method](https://en.wikipedia.org/wiki/Newton%27s_method). [Accessed: Sep. 17, 2025].

## A Appendix: Full Code Listings (Optional)

This appendix includes the full code listings for the root-finding methods implemented in C++.

## A.1 RootFailureFinding.cpp

Listing 1: False Position Method for General Polynomials

```
1 #include <iostream>
2 #include <iomanip>
3 #include <cmath>
4 #include <vector>
5 using namespace std;
6
7 double f(const vector<double>& coeffs, double x) {
8     int n = coeffs.size() - 1;
9     double result = 0.0;
10    for (int i = 0; i <= n; i++) {
11        result += coeffs[i] * pow(x, n - i);
12    }
13    return result;
14 }
15
16 double maxRootEstimate(const vector<double>& coeffs) {
17     int n = coeffs.size() - 1;
18     if (n < 2) return 1.0;
19     double a_n = coeffs[0];
20     double a_n1 = coeffs[1];
21     double a_n2 = coeffs[2];
22     double discrim = (a_n1 / a_n) * (a_n1 / a_n) - 2 * (a_n2 / a_n);
23     if (discrim < 0) discrim = 0;
24     return sqrt(discrim);
25 }
26
27 bool findInitialGuesses(const vector<double>& coeffs, double& x1,
28     double& x2, double step = 0.1) {
29     double R = maxRootEstimate(coeffs);
30     double prevX = -R;
31     double prevF = f(coeffs, prevX);
32     for (double x = -R + step; x <= R; x += step) {
33         double currF = f(coeffs, x);
34         if (prevF * currF < 0) {
35             x1 = prevX;
36             x2 = x;
37             return true;
38         }
39         prevX = x;
40         prevF = currF;
41     }
42     return false;
43 }
44
45 void falsePosition(const vector<double>& coeffs, double x1, double x2,
46     double tolerance, int maxIterations) {
47     double fx1 = f(coeffs, x1);
48     double fx2 = f(coeffs, x2);
49
50     if (fx1 * fx2 > 0) {
51         cout << "Error: f(x1) and f(x2) must have opposite signs.\n";
52         return;
53     }
54 }
```

```

53 double x0, fx0, prevX0, relativeError;
54 int iteration = 0;
55 bool firstIteration = true;
56
57 cout << "\nIteration Table:\n";
58 cout << "
    -----
    n";
59 cout << "Iter|          x1          |          x2          |          x0          |          f(
    x0)          |Relative Error\n";
60 cout << "
    -----
    n";
61
62 do {
63     fx1 = f(coeffs, x1);
64     fx2 = f(coeffs, x2);
65
66     if (fabs(fx1) < tolerance) { x0 = x1; fx0 = fx1; break; }
67     if (fabs(fx2) < tolerance) { x0 = x2; fx0 = fx2; break; }
68
69     if (fabs(fx2 - fx1) < 1e-12) cout << "Warning: Denominator
        small, may cause precision issues.\n";
70
71     x0 = (x1 * fx2 - x2 * fx1) / (fx2 - fx1);
72     fx0 = f(coeffs, x0);
73
74     if (firstIteration) { relativeError = 100.0; firstIteration =
        false; }
75     else { relativeError = fabs((x0 - prevX0) / x0) * 100; }
76
77     cout << setw(4) << iteration + 1 << " | "
78         << setw(12) << fixed << setprecision(6) << x1 << " | "
79         << setw(12) << x2 << " | "
80         << setw(12) << x0 << " | "
81         << setw(12) << fx0 << " | "
82         << setw(12) << relativeError << "%\n";
83
84     if (fabs(fx0) < tolerance) break;
85
86     if (fx1 * fx0 < 0) x2 = x0;
87     else x1 = x0;
88
89     prevX0 = x0;
90     iteration++;
91 } while (iteration < maxIterations);
92
93 cout << "
    -----
    n";
94 if (fabs(fx0) < tolerance) {
95     cout << "\\checkmark Convergence achieved!\\n";
96
97     cout << "Root approximation: " << fixed << setprecision(6) <<
        x0 << endl;
98     cout << "f(x) = " << scientific << setprecision(6) << fx0 <<
        endl;
99 } else {

```

```

100     cout << "\\xmarkMaximum iterations reached.\\n";
101     cout << "Best approximation:_" << fixed << setprecision(6) <<
        x0 << endl;
102     cout << "f(x)_" << scientific << setprecision(6) << fx0 <<
        endl;
103 }
104 cout << "Iterations:_" << iteration + 1 << endl;
105 }
106
107 int main() {
108     int degree;
109     cout << "===FALSE POSITION METHOD (General Polynomial)===\\n\\n";
110
111     cout << "Enter degree of polynomial: ";
112     cin >> degree;
113
114     vector<double> coeffs(degree + 1);
115     cout << "Enter coefficients one by one (highest degree first):\\n";
116     for (int i = 0; i <= degree; i++) {
117         cout << "a" << (degree - i) << "_=";
118         cin >> coeffs[i];
119     }
120
121     double tolerance;
122     int maxIterations;
123     cout << "Enter tolerance (e.g., 0.0001): ";
124     cin >> tolerance;
125     cout << "Enter maximum number of iterations: ";
126     cin >> maxIterations;
127
128     double x1, x2;
129     char choice;
130     cout << "Do you want to use auto-generated initial guesses? (y/n): ";
131     cin >> choice;
132
133     if (choice == 'y' || choice == 'Y') {
134         if (!findInitialGuesses(coeffs, x1, x2)) {
135             cout << "Error: Could not find initial guesses automatically. Try manual input.\\n";
136             return 1;
137         }
138         cout << "Automatic initial guesses found: x1=" << x1 << ", x2=" << x2 << endl;
139     } else {
140         cout << "Enter first initial guess (x1): ";
141         cin >> x1;
142         cout << "Enter second initial guess (x2): ";
143         cin >> x2;
144     }
145
146     falsePosition(coeffs, x1, x2, tolerance, maxIterations);
147
148     return 0;
149 }

```

## A.2 NewtonRaphson.cpp

Listing 2: Newton-Raphson Method for General Polynomials

```
1 #include <iostream>
2 #include <iomanip>
3 #include <cmath>
4 #include <vector>
5 using namespace std;
6
7 double f(const vector<double>& coeffs, double x) {
8     int n = coeffs.size() - 1;
9     double result = 0.0;
10    for (int i = 0; i <= n; i++) {
11        result += coeffs[i] * pow(x, n - i);
12    }
13    return result;
14 }
15
16 double df(const vector<double>& coeffs, double x) {
17     int n = coeffs.size() - 1;
18     double result = 0.0;
19     for (int i = 0; i < n; i++) {
20         result += (n - i) * coeffs[i] * pow(x, n - i - 1);
21     }
22     return result;
23 }
24
25 void newtonRaphson(const vector<double>& coeffs, double x0, double
    tolerance, int maxIterations) {
26     double x1, fx, dfx, relativeError;
27     int iteration = 0;
28     bool converged = false;
29
30     cout << "\nIteration Table:\n";
31     cout << "
        -----
        n";
32     cout << "Iter | x0 | f(x0) | f'(x0) |
        | x1 | Relative Error\n";
33     cout << "
        -----
        n";
34
35     do {
36         fx = f(coeffs, x0);
37         dfx = df(coeffs, x0);
38
39         if (fabs(dfx) < 1e-12) {
40             cout << "Error: Derivative became zero. Method cannot
                continue.\n";
41             return;
42         }
43
44         x1 = x0 - fx / dfx;
45
46         if (iteration == 0) {
47             relativeError = 100.0;
```

```

48     } else {
49         relativeError = fabs((x1 - x0) / x1) * 100;
50     }
51
52     cout << setw(4) << right << iteration + 1 << " | "
53         << setw(13) << fixed << setprecision(6) << x0 << " | "
54         << setw(14) << fx << " | "
55         << setw(14) << dfx << " | "
56         << setw(13) << x1 << " | "
57         << setw(13) << relativeError << " %"
58         << endl;
59
60     if (fabs(fx) < tolerance || relativeError < tolerance) {
61         converged = true;
62         break;
63     }
64
65     x0 = x1;
66     iteration++;
67 } while (iteration < maxIterations);
68
69 cout << "
70     -----
71     n";
72
73 if (converged)
74     cout << "Root found: " << fixed << setprecision(6) << x1 << "
75         after " << iteration + 1 << " iterations.\n";
76 else
77     cout << "Method did not converge within max iterations.\n";
78 }
79
80 int main() {
81     int degree;
82     cout << "=== NEWTON-RAPHSON METHOD (General Polynomial) ===\n\n";
83
84     cout << "Enter degree of polynomial: ";
85     cin >> degree;
86
87     vector<double> coeffs(degree + 1);
88     cout << "Enter coefficients one by one (highest degree first):\n";
89     for (int i = degree; i >= 0; i--) {
90         cout << "a" << i << " = ";
91         cin >> coeffs[degree - i];
92     }
93
94     double x0, tolerance;
95     int maxIterations;
96
97     cout << "\nEnter initial guess (x0): ";
98     cin >> x0;
99     cout << "Enter tolerance (e.g., 0.0001): ";
100     cin >> tolerance;
101     cout << "Enter maximum number of iterations: ";
102     cin >> maxIterations;
103
104     newtonRaphson(coeffs, x0, tolerance, maxIterations);

```



```

103     return 0;
104 }

```

## A.3 FalsePosition.cpp

Listing 3: Demonstration of Root-Finding Failures

```

1  #include <iostream>
2  #include <cmath>
3  #include <iomanip>
4  using namespace std;
5
6  constexpr int MAXIT = 200;
7  constexpr double TOL = 1e-10;
8
9  double f_cuberoot(double x) { return cbrt(x); }
10 double df_cuberoot(double x) {
11     if (x == 0.0) return 0.0;
12     double t = cbrt(x);
13     return (1.0 / 3.0) / (t * t);
14 }
15
16 double f_repeat2(double x) { double t = x - 1.0; return t * t; }
17 double df_repeat2(double x) { return 2.0 * (x - 1.0); }
18
19 double f_x3(double x) { return x * x * x; }
20 double df_x3(double x) { return 3.0 * x * x; }
21
22 double f_quad(double x) { return x*x - 2.0; }
23 double df_quad(double x) { return 2.0 * x; }
24
25 int pow_n = 50;
26 double f_pown(double x) { return pow(x, pow_n) - 1.0; }
27
28 void newton_method(double (*f)(double), double (*df)(double),
29                   double x0, int maxit, double tol, const string& tag)
30 {
31     cout << "-----Newton-Raphson:_" << tag << "_|_x0=_|" << fixed <<
32         setprecision(6) << x0 << "_-----\n";
33     cout << setw(6) << "Iter" << setw(12) << "x0" << setw(12) << "f(x0)"
34         << setw(12) << "f'(x0)"
35         << setw(12) << "x1" << setw(14) << "RelError(%) \n";
36
37     double x = x0;
38     for (int k = 1; k <= maxit; ++k) {
39         double fx = f(x);
40         double dfx = df(x);
41         if (fabs(dfx) < 1e-14) {
42             cout << setw(6) << k << setw(12) << x << setw(12) << fx <<
43                 setw(12) << 0
44                 << setw(12) << x << setw(14) << "derivative~0" << "\n"
45                 ;
46             return;
47         }
48         double dx = fx / dfx;
49         double x1 = x - dx;
50         double relErr = (k==1)? 100.0 : fabs((x1 - x)/x1)*100;

```

```

46
47     cout << setw(6) << k << setw(12) << x << setw(12) << fx << setw
      (12) << dfx
48         << setw(12) << x1 << setw(14) << relErr << "\n";
49
50     if (fabs(dx) < tol) {
51         cout << "Converged to x=" << x1 << " after " << k << "
      iterations, f(x)=" << f(x1) << "\n";
52         return;
53     }
54     x = x1;
55 }
56 cout << "Max iterations reached; last x=" << x << ", f(x)=" <<
      f(x) << "\n";
57 }
58
59 // [Include false_position and false_position_illinois functions here
      exactly as in your code]
60
61 int main() {
62     cout << "\nROOT-FINDING FAILURE DEMONSTRATIONS (C++)\n\n";
63
64     newton_method(f_cuberoot, df_cuberoot, 0.1, 20, 1e-12, "f(x)=
      cuberoot(x)->divergent");
65     newton_method(f_repeat2, df_repeat2, 2.0, 40, 1e-12, "f(x)=(x-1)^2-
      >repeated root");
66     newton_method(f_x3, df_x3, 0.0, 10, 1e-12, "f(x)=x^3 with x0=0 (
      derivative=0)");
67     newton_method(f_quad, df_quad, 1.0, 20, 1e-12, "f(x)=x^2-2 (good
      convergence)");
68
69     pow_n = 50;
70     false_position(f_pown, 0.0, 2.0, 200, 1e-12, "f(x)=x^50-1 slow");
71     false_position_illinois(f_pown, 0.0, 2.0, 200, 1e-12, "f(x)=x^50-1
      Illinois");
72
73     cout << "\nSummary:\n";
74     cout << "Newton diverges for cbrt(x) because x_{n+1}=-2x_n.\n";
75     cout << "Newton slow for repeated roots.\n";
76     cout << "False Position may move one endpoint slowly; Illinois
      improves convergence.\n";
77
78     return 0;
79 }

```

## Appendix: Failure and Convergence Cases

Newton-Raphson: Divergence Example  $f(x) = \sqrt[3]{x}$

Iter	$x_n$	$f(x_n)$	$f'(x_n)$	$x_{n+1}$
1	0.100000	0.464159	1.547196	-0.200000
2	-0.200000	-0.584804	0.974673	0.400000
3	0.400000	0.736806	0.614005	-0.800000
4	-0.800000	-0.928318	0.386799	1.600000

5	1.600000	1.169607	0.243668	-3.200000
6	-3.200000	-1.473613	0.153501	6.400000
7	6.400000	1.856636	0.096700	-12.800000
8	-12.800000	-2.339214	0.060917	25.600000
9	25.600000	2.947225	0.038375	-51.200000
10	-51.200000	-3.713271	0.024175	102.400000

Table 4: Newton-Raphson divergence for  $f(x) = \sqrt[3]{x}$ .

### Newton-Raphson: Repeated Root $f(x) = (x - 1)^2$

Iter	$x_n$	$f(x_n)$	$f'(x_n)$	$x_{n+1}$
1	2.000000	1.000000	2.000000	1.500000
2	1.500000	0.250000	1.000000	1.250000
3	1.250000	0.062500	0.500000	1.125000
4	1.125000	0.015625	0.250000	1.062500
5	1.062500	0.003906	0.125000	1.031250
6	1.031250	0.000977	0.062500	1.015625
7	1.015625	0.000244	0.031250	1.007812
8	1.007812	0.000061	0.015625	1.003906

Table 5: Newton-Raphson convergence slows for repeated root.

### Newton-Raphson: Derivative Zero $f(x) = x^3$ , $x_0 = 0$

Iter	$x_n$	$f(x_n)$	$f'(x_n)$	Note
1	0.000000	0.000000	0.000000	derivative $\approx 0$

Table 6: Newton-Raphson fails when derivative is zero.

### Newton-Raphson: Good Convergence $f(x) = x^2 - 2$ , $x_0 = 1$

Iter	$x_n$	$f(x_n)$	$f'(x_n)$	$x_{n+1}$
1	1.000000	-1.000000	2.000000	1.500000
2	1.500000	0.250000	3.000000	1.416667
3	1.416667	0.006944	2.833333	1.414216
4	1.414216	0.000006	2.828431	1.414214

Table 7: Newton-Raphson shows good convergence.

### False Position: Slow Convergence $f(x) = x^{50} - 1$

Iter	$x_1$	$x_2$	$x_0$	$f(x_0)$
1	0.000000	2.000000	0.000000	-1.000000
2	0.000000	2.000000	0.000000	-1.000000
3	0.000000	2.000000	0.000000	-1.000000

---

Table 8: False Position moves one endpoint slowly.

### Illinois Method: Improvement over False Position $f(x) = x^{50} - 1$

Iter	$x_1$	$x_2$	$x_0$	$f(x_0)$
1	0.000000	2.000000	0.000000	-1.000000
2	0.000000	2.000000	0.000000	-1.000000
3	0.000000	2.000000	0.000000	-1.000000

Table 9: Illinois method improves convergence over slow False Position.

### Summary

- Newton-Raphson diverges for  $f(x) = \sqrt[3]{x}$  due to  $x_{n+1} = -2x_n$ .
- Newton-Raphson slows for repeated roots.
- Newton-Raphson fails if derivative is zero.
- False Position may move one endpoint slowly; Illinois improves convergence.