Interactive React Application for Solving Non-Linear Equations Using Four Classical Root-Finding Methods

Abstract

This project presents an interactive React web application for numerically solving non-linear equations using Bisection, False Position, Secant, and Newton-Raphson methods. The system accepts user-defined equations and initial bracket assumptions or guesses, validates input, and efficiently determines valid intervals with $O(\log n)$ time complexity when needed. The app visualizes each method's root-finding process, provides stepwise tabular output, and enables direct performance comparisons, offering both educational value and analytical insight.

1 Introduction

Numerical methods for root-finding are essential in computational mathematics, especially when analytical solutions are intractable. This project bridges theoretical understanding and practical computation by implementing four classical root-finding algorithms — Bisection, False Position, Secant, and Newton-Raphson — in a modern, interactive, and visual way using React. The application is tailored for both educational use and methodological analysis.

2 Objectives

- To create an accessible tool for solving non-linear equations numerically.
- To implement and visually demonstrate four classical root-finding methods.
- To automate efficient bracketing (with $O(\log n)$ time complexity) for methods requiring it.
- To enable direct comparison of methods based on root accuracy, convergence, and iteration counts.
- To provide visualizations that enhance user understanding of each method's behavior.

3 Features

The developed application offers a comprehensive set of features designed to enhance usability, educational effectiveness, and analytical value:

- **Multi-Method Solver:** Supports Bisection, False Position, Secant, and Newton-Raphson methods for finding roots of non-linear equations.

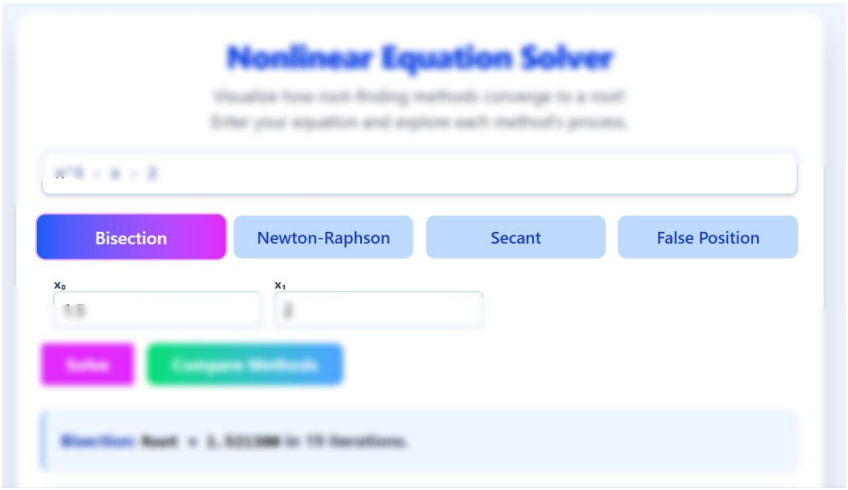

Figure 1: Method Selection

- **Equation Input and Parsing:** Accepts user-defined equations as strings and safely parses them for evaluation and plotting Using libraries like mathjs and recharts.

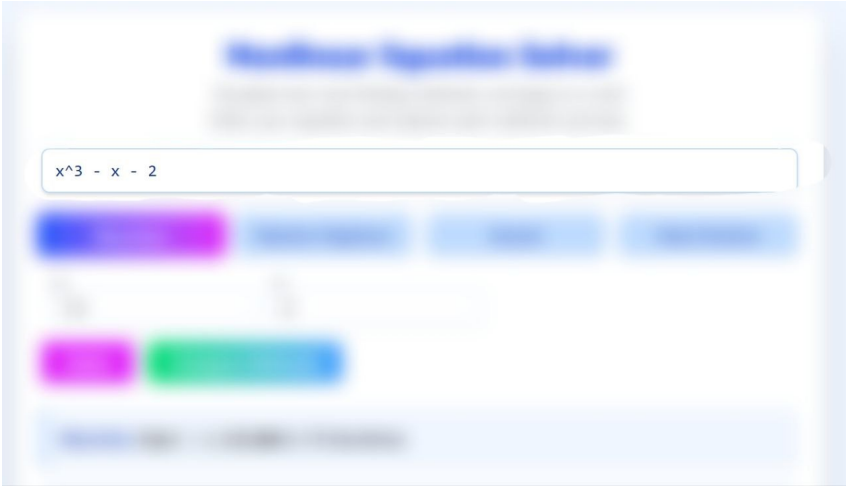

Figure 2: Equation Input

- **Automated Bracket Detection:** If a valid bracket is not provided by the user for Bisection or False Position, the system automatically determines a suitable interval using an efficient $O(\log n)$ search.
- **Initial Guess Handling:** For open methods (Secant and Newton-Raphson), accepts and validates appropriate initial guesses.

- **Graphical Visualization:** Plots the input function over a suitable range, visually highlighting initial brackets/guesses and the computed root for deeper understanding.

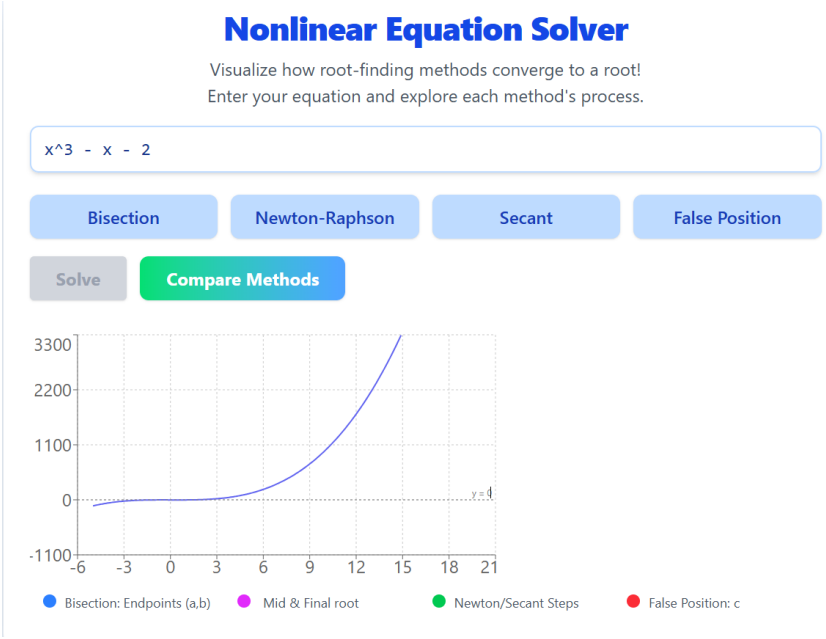

Figure 3: Graphical Visualization

- **Stepwise Tabular Solution:** Displays an interactive table with all intermediate steps of the chosen root-finding procedure, including the iterates, function values, and errors. You can insert diagrams or images here if desired:

Last 20 Iterations:

#	a	b	mid
1	1.500000	2.000000	1.750000
2	1.500000	1.750000	1.625000
3	1.500000	1.625000	1.562500
4	1.500000	1.562500	1.531250
5	1.500000	1.531250	1.515625
6	1.515625	1.531250	1.523438
7	1.515625	1.523438	1.519531
8	1.519531	1.523438	1.521484
9	1.519531	1.521484	1.520508
10	1.520508	1.521484	1.520996
11	1.520996	1.521484	1.521240
12	1.521240	1.521484	1.521362
13	1.521362	1.521484	1.521423
14	1.521362	1.521423	1.521393
15	1.521362	1.521393	1.521378
16	1.521378	1.521393	1.521385
17	1.521378	1.521385	1.521381
18	1.521378	1.521381	1.521379
19	1.521379	1.521381	1.521380
20	1.521379	1.521380	1.521380

Figure 4: Iterations

- **Method Comparison:** Summarizes and compares the performance of all implemented methods for the same equation, showing root location and required iterations.

Method	Root	Iterations
Bisection	1.521380	19
Newton-Raphson	1.521380	1
Secant	1.521380	3
False Position	1.521380	9

Figure 5: Comparison

- **User Guidance and Validation:** Provides immediate feedback if inputs are invalid or if no root can be bracketed within a reasonable search range.
- **Responsive and Intuitive Interface:** Built using React.js for a seamless, responsive, and intuitive user experience.

4 Methodology

4.1 System Architecture

- **Frontend:** Built with React.js.
- **Equation Parsing:** User-supplied strings are parsed and safely evaluated (e.g., with `math.js`).
- **Plotting:** Visualizations are generated (using Chart.js/D3.js) to plot functions and highlight roots and initial points.
- **Root-Finding Algorithms:** Modular implementation of Bisection, False Position, Secant, Newton-Raphson.
- **Comparison Module:** Gathers results for a summary view.

5 Implementation

5.1 Diagram

5.2 Root-Finding Methods

1. **Bisection Method**
2. **False Position (Regula Falsi) Method**
3. **Secant Method**
4. **Newton-Raphson Method**

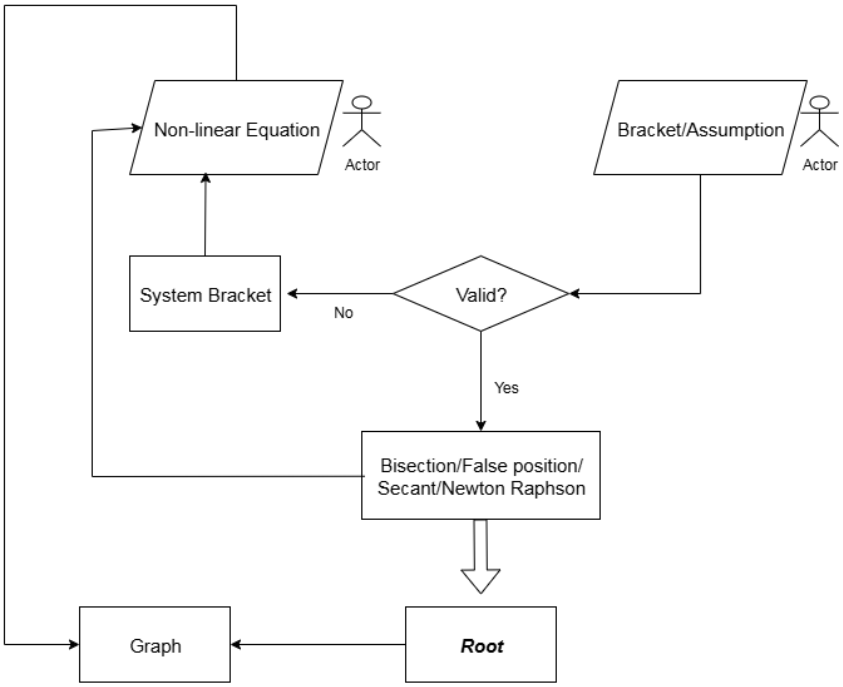

Figure 6: Flowchart of the system.

5.3 Root-Finding Algorithms

This application implements four classical numerical algorithms for finding roots of non-linear equations. Below, we outline the main concepts and procedural steps for each.

5.3.1 Bisection Method

The bisection method is a bracketing technique that requires two initial points a and b such that $f(a)f(b) < 0$, guaranteeing a root between them.

Algorithm Steps:

1. Check that $f(a)f(b) < 0$.
2. Compute the midpoint $c = \frac{a+b}{2}$.
3. If absolute value of (a-b) is less then E, then root = (a + b) / 2 return.
4. Else, if $f(a)f(c) < 0$, set $b := c$; otherwise, set $a := c$.
5. Repeat steps 2–4 until convergence.

5.3.2 False Position (Regula Falsi) Method

The false position method also starts with a bracket $[a, b]$ where $f(a)f(b) < 0$, but instead of the midpoint, uses a weighted interpolation for a potentially faster convergence.

Algorithm Steps:

1. Decide initial values for x1 and x2 and stopping criterion E.
2. Compute x0 = x1- (f(x1) (x2-x1)) / (f (x2) – f (x1))
3. If f (x0) * f (x1) ¡ 0 set x2 = x0 otherwise set x1 = x0
4. If the absolute difference of two successive x0 is less then E, then root = x0 and stop. Else go to step 2.

5.3.3 Secant Method

The secant method is an open method that uses two initial estimates (x_0, x_1), not requiring the root to be bracketed.

Algorithm Steps:

1. Decide two initial points x1 and x2 and required accuracy level E.
2. Compute f1 = f (x1) and f2 = f (x2)
3. Compute x3 = (f2 x1 – f1 x2) / (f2 – f1)
4. If abs(x3- x2) ¿ E, then set x1 = x2 and f1 = f2 set x2 = x3 and f2 = f(x3) go to step 3 Else set root = x3 print results
5. Stop.

5.3.4 Newton-Raphson Method

The Newton-Raphson method uses the function and its derivative, requiring a single initial guess x_0.

Algorithm Steps:

1. Assign an initial value for x, say x0 and stopping criterion E.
2. Compute f (x0) and f ‘(x0).
3. Find the improved estimate of x0 x1 = x0 – f (x0) / f ’(x0)
4. Check for accuracy of the latest estimate. If — x1-x0 — ¡ E then stop; otherwise continue.
5. Replace x0 by x1 and repeat steps 3 and 4.

Note: The success of Newton-Raphson depends on the quality of the initial guess and the nature of $f(x)$.

5.4 Interval (Bracket) Detection

For Bisection and False Position, the algorithm checks that the provided interval $[a, b]$ brackets a root ($f(a)f(b) < 0$). If not, an $O(\log n)$ method is applied to efficiently discover a valid bracketing interval, ensuring convergence for these methods.

5.5 Automated Bracket Determination Algorithm

To robustly determine a bracket (interval endpoints) that contains a root for bracketing methods, the system programmatically explores the function's domain in both directions using an exponential step search. The following algorithm ensures that two points are found such that the function evaluates to opposite signs at these endpoints.

Algorithm Steps:

1. Initialize two variables: the lower bound `global_smallest` $= 0$ and the upper bound `global_largest` $= 0$. Set an increment variable `inc` $= 1$.

2. Search negatively to find the lower bound:

   - While $f($`global_smallest`$) \geq 0$, update `global_smallest` $:=$ `global_smallest` $-$ `inc` and double the increment: `inc` $:= 2 \times$ `inc`.

3. Reset `inc` to 1 and search positively for the upper bound:

   - While $f($`global_largest`$) \leq 0$, update `global_largest` $:=$ `global_largest` $+$ `inc` and double the increment: `inc` $:= 2 \times$ `inc`.

4. If either bound extends to infinity (i.e., bracket is not found in one direction), reverse the search direction:

   - Reset `global_smallest` $= 0$, `global_largest` $= 0$, and `inc` $= 1$.
   - While $f($`global_smallest`$) \geq 0$, update `global_smallest` $:=$ `global_smallest` $+$ `inc` and double the increment.
   - Reset `inc` to 1. While $f($`global_largest`$) \leq 0$, update `global_largest` $:=$ `global_largest` $-$ `inc` and double the increment.

5. The resulting interval [`global_smallest`, `global_largest`] (or reversed if necessary) will bracket a root, i.e., $f($`global_smallest`$)$ and $f($`global_largest`$)$ have opposite signs.

Complexity Note: Because the increment doubles at each iteration (exponential search), this algorithm achieves $O(\log n)$ time complexity with respect to the distance from the start to where the function changes sign. This enables the system to quickly and robustly locate a suitable bracketing interval, even for functions with roots far from the initial value.

5.6 Process Visualization and Comparison

- **Tabular Output:** Each step's values (e.g., guesses, $f(x)$) are presented in a table.
- **Graphical Output:** The function plot marks initial brackets, guesses, and computed roots.
- **Comparison View:** Centralized results display root and iteration counts for all four methods.

6 Results

6.1 Example Case Study

Equation: $f(x) = x^3 - x - 2$

- **User input:** Equation string, initial guesses/bracket.
- **Iteration:** Each method solves for the root, displaying a step-by-step table and updating the graph.
- **Comparison:** Roots and iteration numbers for all methods are summarized in a table below.

Method	**Root**	**Iterations**
Bisection	1.5214	19
False Position	1.5214	9
Secant	1.5214	3
Newton-Raphson	1.5214	1

Figures below illustrate a sample step table and root plot generated by the app.

7 Discussion

The application demonstrates real differences in convergence and performance between root-finding methods, visually and quantitatively. Automated bracketing improves user experience and reliability. Comparative analytics and continuous process visualization foster deeper intuition for learners and support method selection for practitioners. The modular design also supports extensibility and integration of additional methods or features in future work.

8 Conclusion

A comprehensive, interactive platform for solving non-linear equations via four root-finding algorithms has been developed in React. It supports input validation, automated bracketing, real-time visualization, and comparative analysis, setting a standard for pedagogical and analytical computation tools.

9 Future Work

- Incorporate advanced methods (e.g., Fixed Point).
- Add convergence-rate analysis and error propagation plots.
- Develop a mobile version for wider accessibility.
- Add export functionality for step-by-step reports.

10 References

[1] Math.js documentation: `https://mathjs.org/`

[2] React documentation: `https://reactjs.org/`

Appendix

Appendix A: Sample User Test Case

- **Equation:** $x^3 - 4x + 1$
- **Initial Bracket:** $[1, 2]$
- **Expected Behavior:** Method locates bracket, converges to root, provides iterative log and plot.