UNIVERSITY OF BOHOL

COLLEGE OF ENGINEERING, TECHNOLOGY, ARCHITECTURE, AND FINE ARTS

# Numerical Methods
# Final Requirements

Submitted by :
**Diacor, Jay Lord C.**
**BSCPE-2**

Submitted to :
**Engr. Victor John Anunciado**

## TABLE OF CONTENTS

## ***INTRODUCTION***

The purpose of this project is to implement and explore various numerical methods for finding roots of nonlinear equations, a fundamental problem in many scientific and engineering applications. For a computer engineering student like myself, this project has been invaluable in bridging theoretical mathematics with practical programming skills.

By developing and coding algorithms such as the graphical method, incremental search, bisection, Newton-Raphson, Secant, and Regula Falsi, I gained hands-on experience in algorithm design and numerical computation core competencies in computer engineering. This project enhanced my understanding of how numerical techniques can be efficiently translated into software, emphasizing the importance of accuracy, convergence, and computational efficiency.

Additionally, this work improved my proficiency in programming concepts such as iteration control, error handling, and function evaluation, which are essential for developing reliable and optimized software systems. It also provided insight into how low-level numerical details impact high-level application performance, a critical aspect when designing embedded systems, simulations, or real-time computing applications.

Overall, this project strengthened my analytical thinking and programming skills, preparing me to tackle complex computational problems encountered in both academic research and industry roles within computer engineering.

## ***OVERVIEW OF NUMERICAL METHODS***

This project implements several widely used numerical methods for root finding, each with its unique approach and advantages:

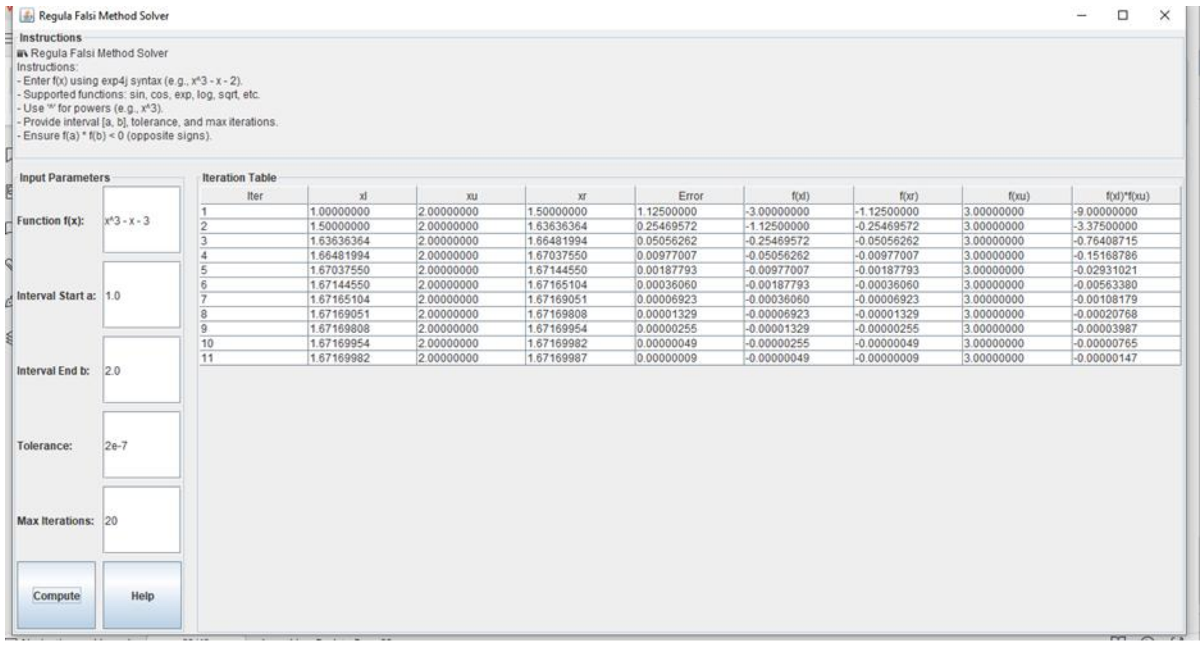
- Graphical Method: Estimates roots by plotting the function and visually identifying where it crosses the x-axis. Useful for initial guesses but not highly accurate.
- Incremental Search Method: Searches for a sign change in the function by incrementally stepping through values. Simple but can be slow and may miss roots if the step size is too large.

- Bisection Method: A robust bracketing method that repeatedly halves an interval where the function changes sign, guaranteeing convergence under certain conditions.
- Newton-Raphson Method: Uses the function and its derivative to rapidly converge to a root, offering quadratic convergence but requiring a good initial guess and differentiability.
- Secant Method: An approximation of Newton-Raphson that does not require derivatives, using two initial guesses to iteratively approximate the root.
- Regula Falsi (False Position) Method: Combines the bracketing approach of bisection with linear interpolation for faster convergence while maintaining bracketing.

[ Each method is implemented and tested to compare their efficiency, accuracy, and applicability to different types of nonlinear equations. ]

This project not only deepened my understanding of numerical root-finding techniques but also provided practical experience in implementing and analyzing algorithms from a computer engineering perspective. The subsequent sections will detail the programming requirements and provide comprehensive explanations of each numerical method included in this project, highlighting their principles, implementation, and performance.

***PROJECT REQUIREMENTS***



Instructions

Regula Falsi Method Solver
Instructions:
- Enter f(x) using exp4j syntax (e.g., x^3 - x - 2).
- Supported functions: sin, cos, exp, log, sqrt, etc.
- Use '^' for powers (e.g., x^3).
- Provide interval [a, b], tolerance, and max iterations.
- Ensure f(a) * f(b) < 0 (opposite signs).

Input Parameters

| | |
|---|---|
| Function f(x): | x^3 - x - 3 |
| Interval Start a: | 1.0 |
| Interval End b: | 2.0 |
| Tolerance: | 2e-7 |
| Max Iterations: | 20 |

Iteration Table

| Iter | xl | xu | xr | Error | f(xl) | f(xr) | f(xu) | f(xl)*f(xu) |
|---|---|---|---|---|---|---|---|---|
| 1 | 1.00000000 | 2.00000000 | 1.50000000 | 1.12500000 | -3.00000000 | -1.12500000 | 3.00000000 | -9.00000000 |
| 2 | 1.50000000 | 2.00000000 | 1.63636364 | 0.25469572 | -1.12500000 | -0.25469572 | 3.00000000 | -3.37500000 |
| 3 | 1.63636364 | 2.00000000 | 1.66481994 | 0.05056262 | -0.25469572 | -0.05056262 | 3.00000000 | -0.76408715 |
| 4 | 1.66481994 | 2.00000000 | 1.67037550 | 0.00977007 | -0.05056262 | -0.00977007 | 3.00000000 | -0.15168786 |
| 5 | 1.67037550 | 2.00000000 | 1.67144550 | 0.00187793 | -0.00977007 | -0.00187793 | 3.00000000 | -0.02931021 |
| 6 | 1.67144550 | 2.00000000 | 1.67165104 | 0.00036060 | -0.00187793 | -0.00036060 | 3.00000000 | -0.00563380 |
| 7 | 1.67165104 | 2.00000000 | 1.67169051 | 0.00006923 | -0.00036060 | -0.00006923 | 3.00000000 | -0.00108179 |
| 8 | 1.67169051 | 2.00000000 | 1.67169808 | 0.00001329 | -0.00006923 | -0.00001329 | 3.00000000 | -0.00020768 |
| 9 | 1.67169808 | 2.00000000 | 1.67169954 | 0.00000255 | -0.00001329 | -0.00000255 | 3.00000000 | -0.00003987 |
| 10 | 1.67169954 | 2.00000000 | 1.67169982 | 0.00000049 | -0.00000255 | -0.00000049 | 3.00000000 | -0.00000765 |
| 11 | 1.67169982 | 2.00000000 | 1.67169987 | 0.00000009 | -0.00000049 | -0.00000009 | 3.00000000 | -0.00000147 |

a.) can have equations be inputted dynamically.

b.) can how graph & table per method.

c.) can show multiple roots.

d.) display the roots & label each root in the graph.

e.) put legends and instructions on how to use or how you should input equation/data.

f.) tolerance error is in the hundredth thousands places

The project must fulfill the following key requirements to ensure flexibility, usability, and accuracy

a) ***Dynamic Equation Input***

The program allows users to input mathematical equations dynamically at runtime. This flexibility means the software can handle a wide range of functions without the need for recompilation or code changes. To implement this, the project uses ***Exp4j***, a Java library that parses and evaluates mathematical expressions efficiently. Exp4j supports common operators and functions, enabling the user to input complex equations in a natural form. Special attention was given to input validation and error handling to prevent parsing errors and ensure the program responds gracefully to invalid inputs.

b) ***Graph and Table Visualization per Method***

Visual feedback is essential for understanding the behavior of functions and the progress of numerical methods. Each method produces a graph plotting the function over the relevant interval along with visual markers for roots found. Tables accompany these graphs, detailing iteration steps, approximations, and error metrics to help users trace the algorithm's convergence. The project utilizes JFreeChart and JCommon libraries for creating rich, interactive charts in Java. These libraries provide customizable plotting capabilities and allow the addition of legends, labels, and markers essential for clarity. The graphical user interface (GUI) uses ***Absolute Layout Jar*** to place components precisely, ensuring a clean and intuitive presentation.

c) ***Handling Multiple Roots***

Functions can have multiple roots within a given domain, and the program is required to detect and compute all of them. This capability is critical for comprehensive analysis. The program implements scanning strategies combined with adjustable increments in the search interval to identify sign changes that indicate roots. Once a sign change is detected, the root-finding methods are applied locally to find the root with high accuracy. This approach avoids missing closely spaced roots and ensures thorough exploration of the function's domain.

d) ***Root Display and Labeling on Graphs***

All roots found are displayed both numerically in tables and visually labeled on the corresponding graphs. Root points are marked with distinct symbols, and labels are positioned clearly to avoid clutter. This dual representation enhances the user's ability to interpret the results quickly and verify the correctness of computations. The use of legends and tooltips in the charts further improves usability by providing contextual information about each root.

e) ***Legends and User Instructions***

To facilitate ease of use, all graphs include legends explaining the plotted elements, such as the function curve, root markers, and error bounds. Additionally, the program provides detailed instructions on how to input equations and data, including the syntax supported by ***Exp4j***. Clear guidance reduces user errors and helps beginners understand how to interact with the tool effectively.

f) ***High-Precision Tolerance***

The stopping criteria for iterative methods are based on a tolerance set to the hundred-thousandths place (0.00001). This high level of precision ensures that the roots found are accurate and suitable for engineering applications where precision is crucial. The tolerance level balances the trade-off between computational cost and solution accuracy, preventing excessive iterations while maintaining reliable results.

## ***Jar Files Used In the Programming Project***

1.***Exp4j***

Exp4j is a lightweight Java library for parsing and evaluating mathematical expressions. It supports arithmetic operations, functions (like sin, cos, log), and variables, allowing users to input mathematical formulas dynamically

Exp4j is the backbone of the program's dynamic equation input feature. When a user enters a function as a string (e.g., "x^3 - 4*x + 1"), Exp4j parses this string into an executable expression. During the root-finding algorithms, the program evaluates the function values by substituting different values of x into the parsed expression. This approach enables the same codebase to handle any user-defined function without hardcoding specific equations.

Usage Details:

The input string is validated and then compiled into an Expression object via Exp4j.

During iterations, the program calls setVariable("x", value) and evaluate() to get f(x).

Errors in expression syntax are caught to prompt users for corrections.

2. ***JFreeChart***

Purpose:
JFreeChart is a comprehensive charting library for Java that supports a variety of chart types such as line charts, scatter plots, bar charts, and more. It allows extensive customization for professional-quality visualizations.
JFreeChart is used to plot the mathematical function over specified intervals, highlight root points, and visualize convergence behaviors of the numerical methods. Each method's results are displayed graphically, enabling users to visually interpret root locations and algorithm progress.

Usage Details:

Function values over intervals are plotted as XY line charts.
Root points are added as markers or annotations on the chart to label them visually.
Legends and axis labels are included to clarify chart elements.
The charts are embedded inside the program's GUI panels for interactive viewing.

3. ***JCommon***

Purpose:
JCommon is a general-purpose Java library that provides utilities and helper classes required by JFreeChart. It handles tasks such as drawing shapes, managing fonts, and other low-level graphical operations.
While JFreeChart handles high-level charting, JCommon provides the underlying support, ensuring smooth rendering and layout of graphical components. It is automatically leveraged by JFreeChart for creating and managing chart elements.

Usage Details:

The project imports JCommon along with JFreeChart to ensure all charting dependencies are resolved.
Customizations like styling markers, managing colors, and formatting labels rely on JCommon functionalities.

4. ***Absolute Layout Jar***

Absolute Layout is a Java layout manager that allows placing GUI components at fixed positions and sizes, rather than relying on automatic layout arrangements.

This layout manager is used to design the graphical user interface with precise control over the placement of input fields, buttons, charts, and tables. By using Absolute Layout, the interface remains organized and visually consistent, ensuring that components like graphs and tables align properly without unwanted resizing or repositioning.

All GUI elements such as text input boxes, buttons, chart panels, and tables are positioned explicitly using pixel coordinates.This approach provides a clean and user-friendly interface that does not depend on platform-specific layout behaviors.

It also facilitates easy addition of legends, instructions, and other labels in fixed locations.

## *NETBEANS – JAVA*

### *Programming Language: Java*

The entire project was developed using Java, a widely-used, object-oriented programming language known for its portability, robustness, and extensive ecosystem. Java's platform independence ensures that the program can run on any operating system with a compatible Java Virtual Machine (JVM), making the tool accessible to a broad user base.

Benefits of Using Java:

Strong Library Support: Java offers rich libraries such as Exp4j for expression parsing and JFreeChart for graphical visualization, which were essential to implementing the project's requirements efficiently.

Object-Oriented Design: Java's OOP principles helped organize the code into modular, reusable components, improving maintainability and scalability.

Automatic Memory Management: The built-in garbage collector reduces memory management complexity, allowing focus on algorithm development.

Cross-Platform Compatibility: Java programs run seamlessly on different platforms, ensuring flexibility in deployment.

Community and Documentation: A large community and abundant documentation ease troubleshooting and provide resources for implementing complex features.

---

### *Integrated Development Environment: NetBeans*

The project was developed using NetBeans IDE, an open-source integrated development environment that provides comprehensive tools for Java development.

Advantages of NetBeans for This Project:

User-Friendly Interface: NetBeans offers an intuitive GUI with drag-and-drop support for designing interfaces, which streamlined the development of the project's graphical components using Absolute Layout.

Built-in Debugger: The powerful debugger helped identify and fix logic errors and runtime exceptions quickly, ensuring code reliability.

Project Management: NetBeans simplifies managing multiple source files and libraries, such as the external JAR files (Exp4j, JFreeChart, JCommon), through its project explorer and build tools.

Integrated Version Control: Support for Git and other version control systems facilitated source code management and collaboration.

Rich Plugin Ecosystem: The availability of plugins enhanced productivity by adding features like code refactoring, formatting, and performance profiling.

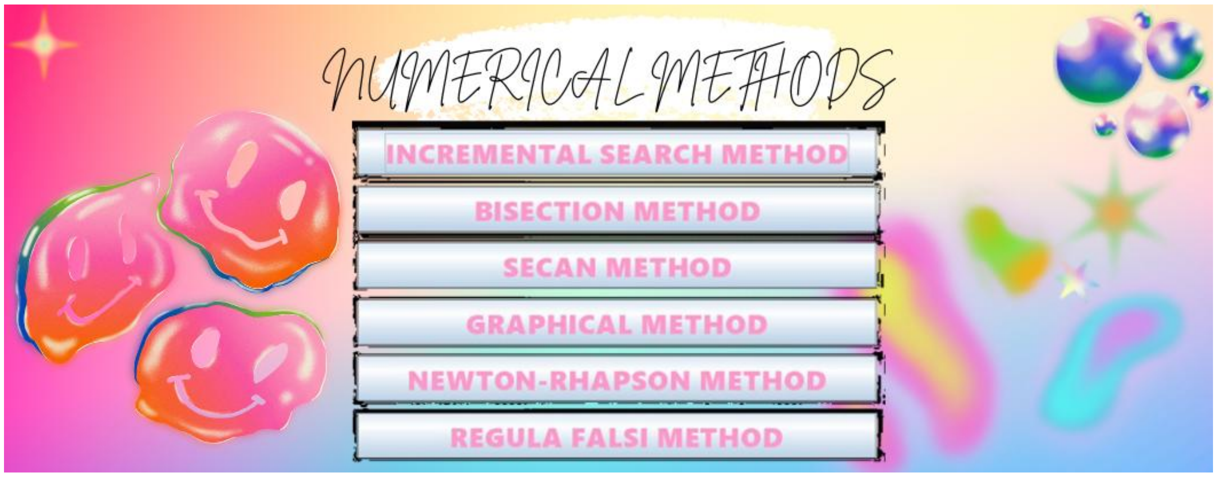
Cross-Platform Support: Like Java, NetBeans runs on multiple operating systems, making the development environment accessible regardless of the user’s OS.

---

Using Java in conjunction with NetBeans provided a powerful, efficient, and flexible development environment that significantly contributed to the successful implementation of this numerical methods project.

# BISECTION METHOD

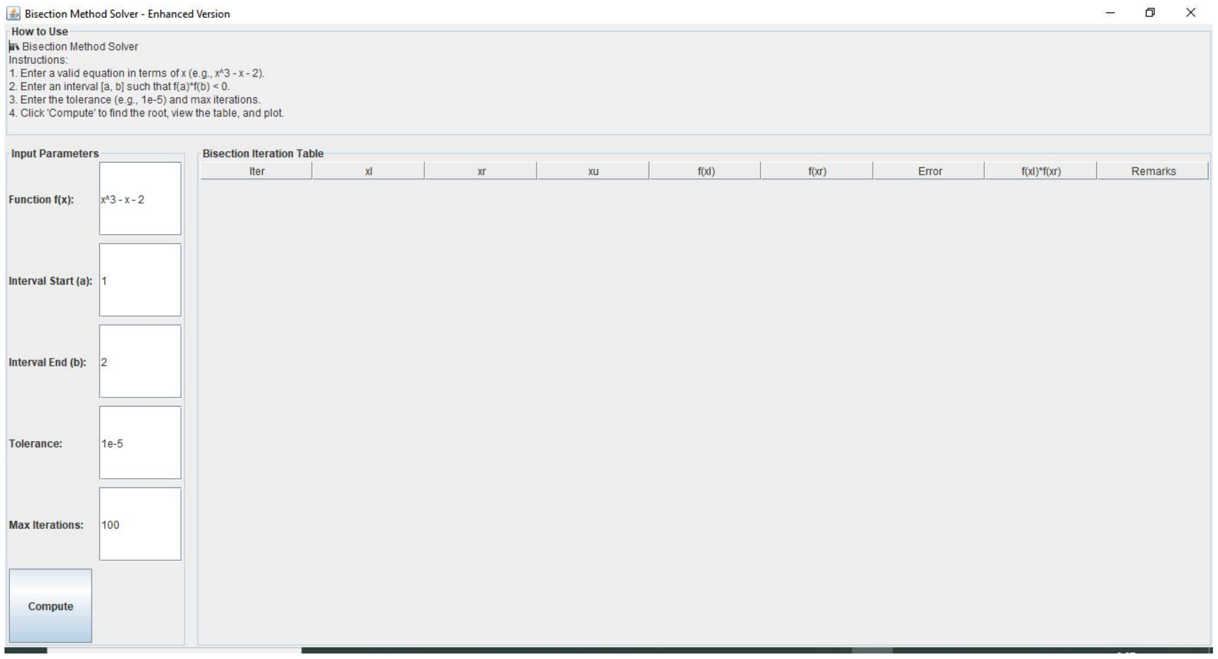
***BISECTION METHOD IN JAVA***



***Bisection Method - Java***

The Bisection Method is a bracketing numerical technique used to find roots of continuous functions. It iteratively halves the interval [xl,xu][x_l, x_u][xl ,xu ] until the function value at the midpoint approaches zero within a specified error threshold (Δ or epsilon). This implementation was done in Java with a graphical user interface and visual output.

| Library | Purpose |
|---|---|
| Jfreechart | Charting and graph visualization of function f(x). |
| Jcommon | Dependency required by JFreeChart |
| Exp4j | Math expression parsing and evaluation (e.g., user input x^2 - 4) |
| Absolute Layout Jar | Layout manager used for precise component positioning |



Bisection Method Solver - Enhanced Version

How to Use

Bisection Method Solver
Instructions:
1. Enter a valid equation in terms of x (e.g., x^3 - x - 2).
2. Enter an interval [a, b] such that f(a)*f(b) < 0.
3. Enter the tolerance (e.g., 1e-5) and max iterations.
4. Click 'Compute' to find the root, view the table, and plot.

Input Parameters

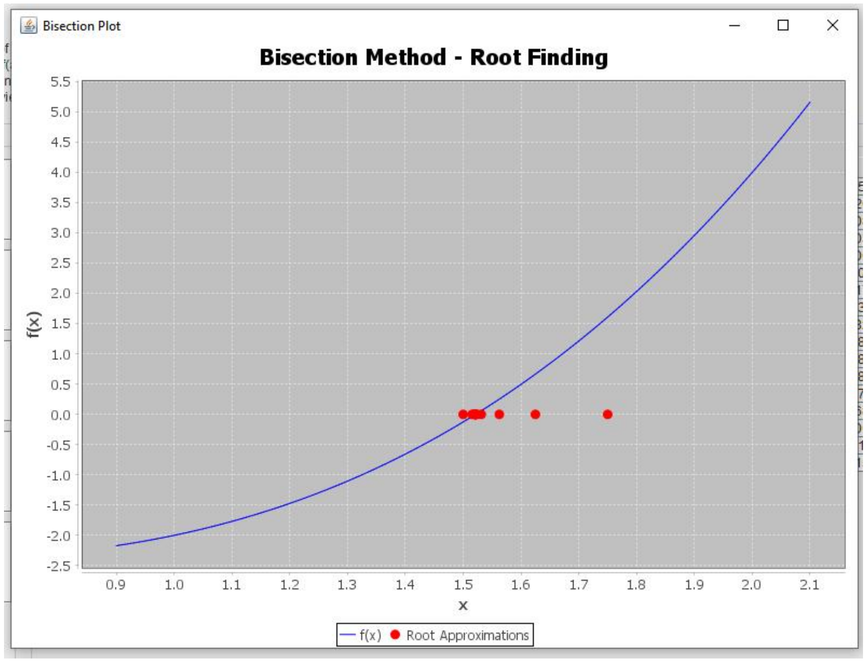
| | |
|---|---|
| Function f(x): | x^3 - x - 2 |
| Interval Start (a): | 1 |
| Interval End (b): | 2 |
| Tolerance: | 1e-5 |
| Max Iterations: | 100 |

Compute

Bisection Iteration Table

| Iter | xl | xr | xu | f(xl) | f(xr) | Error | f(xl)*f(xr) | Remarks |
|---|---|---|---|---|---|---|---|---|

```
String equation = equationf.getText(); // EQUATION TEXT FIELD
double lower = Double.parseDouble(lowerf.getText()); //LOWER LIMIT TEXT FIELD
double upper = Double.parseDouble(upperf.getText()); // UPPER LIMIT TEXT FIELD
double delta = Double.parseDouble(deltaf.getText()); // DELTA X TEXT FIELD
BisectionSolver solver = new BisectionSolver(equation, delta, 100);
List<BisectionSolver.IterationData> iterations = solver.solve(lower, upper);
```



1. ***BisectionSolver.java***

This class encapsulates the core logic of the Bisection Method algorithm. It takes a user-defined equation, tolerance level, and maximum iterations as input parameters. Using an internal EquationEvaluator, it iteratively narrows down the interval [xl,xu][x_l, x_u][xl ,xu ] where the root lies by checking the signs of the function at the interval endpoints and midpoint. It stores each iteration's data—including bounds, midpoint, function value, and approximate error—in a list for later display. The class also handles cases where the initial interval does not bracket a root by throwing an appropriate exception. This separation of solver logic makes the code modular and reusable.

---

2. ***BISECTIONMETHOD.java***

This class represents the graphical user interface (GUI) for the Bisection Method. It provides interactive input fields for the equation, interval bounds, tolerance, and maximum iterations. The interface includes a numpad and function buttons to assist users in entering mathematical expressions accurately. Input validation ensures only valid data is processed. Upon submission, it invokes the BisectionSolver to perform calculations and retrieves iteration data. The GUI then launches the results visualization window (BIGRAPH) and displays iteration details and graphs. This class bridges user input and solver execution, emphasizing user-friendliness and error handling.

---

3. ***BIGRAPH.java***

This class handles the visualization of the Bisection Method results. Using the JFreeChart library, it plots the function over the specified interval and annotates the graph with markers for the lower and upper bounds as well as the approximated root. It also displays a table summarizing each iteration's data—interval bounds, midpoint, function values, errors, and remarks—providing users with comprehensive insight into the method's progress. The graphical and tabular presentation together facilitate better understanding and verification of the root-finding process.

## *Input GUI: BISECTIONMETHOD.java*

- Provides a graphical user interface where users can:
- Enter lower and upper bounds (xl,xu)(x_l, x_u)(xl ,xu )
- Input a function f(x)f(x)f(x)
- Set a stopping criterion (delta/ε)
- Use a numpad for convenience

On ENTER, the system: *Parses inputs, Invokes BisectionSolver Passes results to BIGRAPH for visualization*

Backend Computation: ***BisectionSolver.java***

Algorithm Steps:

- Ensure f(x1).f(xu) < 0
- Calculate midpoint: xr = x1+xu / 2

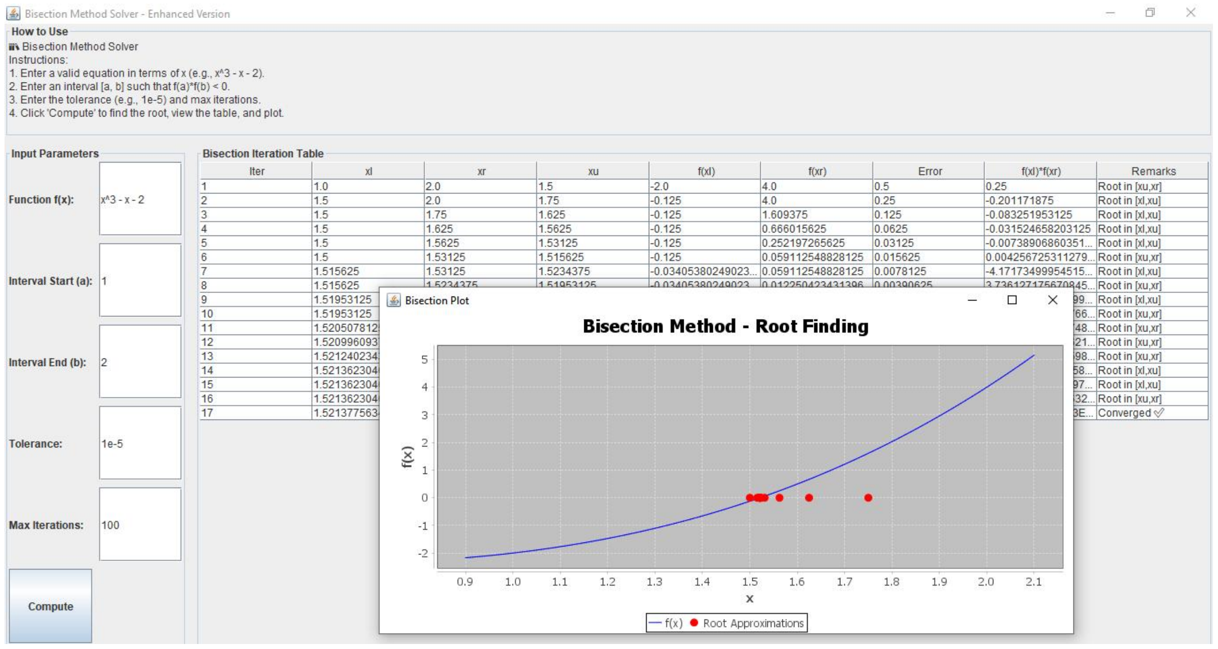
- Evaluate f(xr), update bounds based on sign
- Repeat until |ϵ|<δ or max iterations reached

The method logs each iteration in a custom IterationData class for later rendering.

Output GUI: ***BIGRAPH.java***

Displays:

- A table of iteration data: xl,xu,xr,f(xr),ϵx_l, x_u, x_r, f(x_r), \epsilonxl ,xu ,xr ,f(xr ),ϵ
- A plot of f(x)f(x)f(x) using JFreeChart
- Annotations for root, bounds, and x-axis



How to Use

Bisection Method Solver

Instructions:

1. Enter a valid equation in terms of x (e.g., x^3 - x - 2).
2. Enter an interval [a, b] such that f(a)*f(b) < 0.
3. Enter the tolerance (e.g., 1e-5) and max iterations.
4. Click 'Compute' to find the root, view the table, and plot.

Input Parameters

| | |
|---|---|
| Function f(x): | x^3 - x - 2 |
| Interval Start (a): | 1 |
| Interval End (b): | 2 |
| Tolerance: | 1e-5 |
| Max Iterations: | 100 |

Bisection Iteration Table

| Iter | xl | xr | xu | f(xl) | f(xr) | Error | f(xl)*f(xr) | Remarks |
|---|---|---|---|---|---|---|---|---|
| 1 | 1.0 | 2.0 | 1.5 | -2.0 | 4.0 | 0.5 | 0.25 | Root in [xu,xr] |
| 2 | 1.5 | 2.0 | 1.75 | -0.125 | 4.0 | 0.25 | -0.201171875 | Root in [xl,xu] |
| 3 | 1.5 | 1.75 | 1.625 | -0.125 | 1.609375 | 0.125 | -0.083251953125 | Root in [xl,xu] |
| 4 | 1.5 | 1.625 | 1.5625 | -0.125 | 0.666015625 | 0.0625 | -0.031524658203125 | Root in [xl,xu] |
| 5 | 1.5 | 1.5625 | 1.53125 | -0.125 | 0.252197265625 | 0.03125 | -0.00738906860351... | Root in [xl,xu] |
| 6 | 1.5 | 1.53125 | 1.515625 | -0.125 | 0.059112548828125 | 0.015625 | 0.004256725311279... | Root in [xu,xr] |
| 7 | 1.515625 | 1.53125 | 1.5234375 | -0.03405380249023... | 0.059112548828125 | 0.0078125 | -4.17173499954515... | Root in [xl,xu] |
| 8 | 1.515625 | 1.5234375 | 1.51953125 | -0.03405380249023 | 0.012250423431396 | 0.00390625 | 3.736127175670845... | Root in [xu,xr] |
| 9 | 1.51953125 | | | | | | 99... | Root in [xl,xu] |
| 10 | 1.51953125 | | | | | | 66... | Root in [xu,xr] |
| 11 | 1.52050781 | | | | | | 48... | Root in [xu,xr] |
| 12 | 1.52099609 | | | | | | 21... | Root in [xu,xr] |
| 13 | 1.52124023 | | | | | | 98... | Root in [xu,xr] |
| 14 | 1.52136230 | | | | | | 58... | Root in [xl,xu] |
| 15 | 1.52136230 | | | | | | 97... | Root in [xl,xu] |
| 16 | 1.52136230 | | | | | | 32... | Root in [xu,xr] |
| 17 | 1.52137756 | | | | | | 3E... | Converged ✓ |

Project Problems:

## *Finding multiple roots*

1. Multiple Roots Handling in bisection.py:
We will add logic to continue searching for additional roots in subintervals where sign changes occur. After finding one root, we'll re-adjust the interval and continue looking for the next root.
2. Plotting Multiple Roots in graph_bisection.py:
Modify the plotting function to handle multiple roots and display them on the graph.

## *To display the graph and table along the root/roots (if more than one root)*

Extract the root (last Xr from the last step).

Set the rootlabel text to show the root value formatted nicely.

Add a clear red dot or marker on the graph exactly at (root, 0).

Make sure the renderer for the root series is set properly and visible.

***Problem Root Analysis***

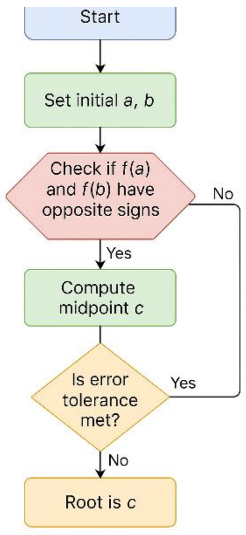
Renderer issue: setting a custom renderer only for series index 1 (the root), but your chart's default renderer might still draw the line for series 0 (function) and no shapes for series 1 unless explicitly enabled.

Shape visibility for root series: The RootCircleRenderer constructor disables lines (super(false, true)) which is good, but you also need to ensure shapes are visible for the root series.

Series indexing: You have three series: 0 = funcSeries, 1 = rootSeries, 2 = limitSeries.

Dataset for rootSeries: The rootSeries has exactly one point: (lastStep.Xr, 0.0). If lastStep.Xr is outside the domain or not visible in plot range, it won't appear.

1. *Make sure the root series has one point at (rootX, f(rootX)).*
2. *Assign the custom RootCircleRenderer to series 1.*
3. *Make sure RootCircleRenderer enables shapes visible for series 0 (its internal series indexing is always zero because it only renders one series).*
4. *Make sure the dotRadius and circleRadius are reasonable (your values are fine).*
5. *Verify that the root's X is within the plot's domain range.*



Input:
f(x) ← the function
xl, xu ← initial lower and upper bounds

```
   ε        ← error tolerance
   max_iter ← maximum allowed iterations

Output:
   Approximate root xr or error message

Begin
   fxl ← f(xl)
   fxu ← f(xu)

   If fxl * fxu > 0 then
      Output "No sign change detected. Root is not guaranteed."
      Exit
   End If

   iter ← 0
   xr_old ← xl

   Repeat
      xr ← (xl + xu) / 2
      fxr ← f(xr)
      iter ← iter + 1

      If iter > 1 then
         error ← abs((xr - xr_old) / xr)
      Else
         error ← ∞
      End If

      Save iteration data: (iter, xl, xu, xr, fxr, error)

      If fxr == 0 OR error < ε then
         Output "Root found:", xr
         Exit
      End If

      If fxl * fxr < 0 then
         xu ← xr
         fxu ← fxr
      Else
         xl ← xr
         fxl ← fxr
      End If

      xr_old ← xr

   Until iter ≥ max_iter
```

```
    Output "Max iterations reached. Approximate root:", xr
End
```

### *User Input Interface and Text Fields*

The Bisection Method implementation features a thoughtfully designed input interface to facilitate accurate and user-friendly data entry. Key input components include:

#### *Lower Limit ($X_l$) and Upper Limit ($X_u$) Fields*

These two text fields allow the user to specify the initial interval within which the root is sought. The program ensures that the lower limit is less than the upper limit and that the function values at these points have opposite signs, which is necessary for the bisection method to work correctly.

#### *Equation Field*

This text field accepts the mathematical function f(x)f(x)f(x) as a string input. To assist users in entering valid expressions, the interface supports common mathematical operators and functions. The input is parsed and evaluated dynamically using the Exp4j library, allowing for flexible and complex function definitions.

#### *Delta (Δ) Field*

This field specifies the tolerance or the desired accuracy for the root approximation. The method iterates until the approximate relative error falls below this threshold, ensuring precise results.

#### *Max Iterations Field*

This input limits the maximum number of iterations the algorithm performs. It serves as a safeguard to prevent infinite loops in cases where the method does not converge within the expected tolerance.

---

### *Numpad and Function Buttons*

To enhance usability and reduce input errors, the GUI includes a custom numpad with digits (0-9), decimal point, and arithmetic operators (+, -, *, /, ^). Additionally, function buttons for trigonometric operations (sin, cos, tan) are provided to help users input common functions quickly and correctly.

Each button appends its respective character or function to the currently active text field, improving input speed and reducing the likelihood of syntax errors. The interface also supports cursor positioning and backspace functionality, enabling users to edit their inputs conveniently.

---

### *Input Validation*

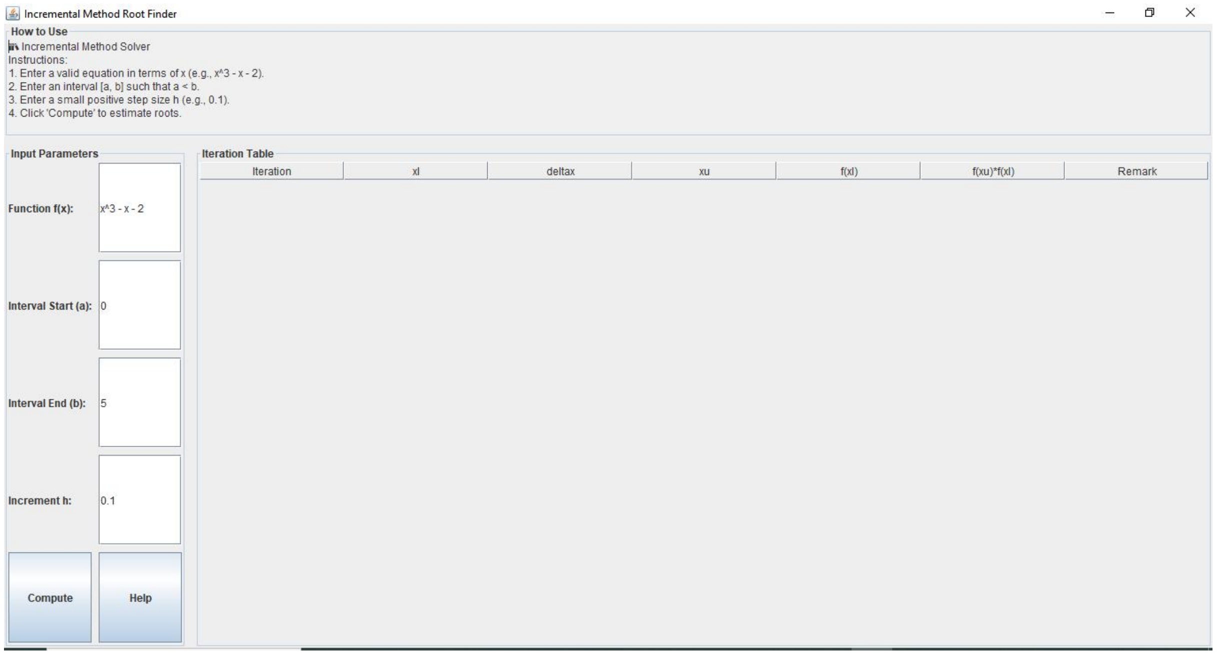
Before running the algorithm, the program performs thorough validation on all input fields to ensure:

All required fields are filled.
Numerical fields contain valid numbers (decimals or integers as appropriate).
The lower limit is less than the upper limit.
The maximum iterations value is positive.
The entered function is syntactically correct and evaluable at the input limits.
If any validation fails, informative error messages prompt the user to correct their inputs, preventing runtime errors and ensuring robust execution.

# INCREMENTAL SEARCH

## *INCREMENTAL SEARCH METHOD – JAVA*

The **Incremental Search Method** is a bracketing technique used to detect root intervals by evaluating a function f(x)f(x)f(x) at regular steps. When a sign change is detected between successive evaluations, a root is suspected in that subinterval. This implementation further uses the **Bisection Method** to refine the root once it's bracketed.



Incremental Method Root Finder

How to Use

Incremental Method Solver
Instructions:
1. Enter a valid equation in terms of x (e.g., x^3 - x - 2).
2. Enter an interval [a, b] such that a < b.
3. Enter a small positive step size h (e.g., 0.1).
4. Click 'Compute' to estimate roots.

Input Parameters

Function f(x): x^3 - x - 2
Interval Start (a): 0
Interval End (b): 5
Increment h: 0.1

Iteration Table

| Iteration | xl | deltax | xu | f(xl) | f(xu)*f(xl) | Remark |
|---|---|---|---|---|---|---|

### *INCREMENTAL.java – Input GUI*

- Allows the user to input:
- Start and end of the interval [xl,xu][x_l, x_u][xl ,xu ]
- Step size (Δx)
- Equation f(x)f(x)f(x)

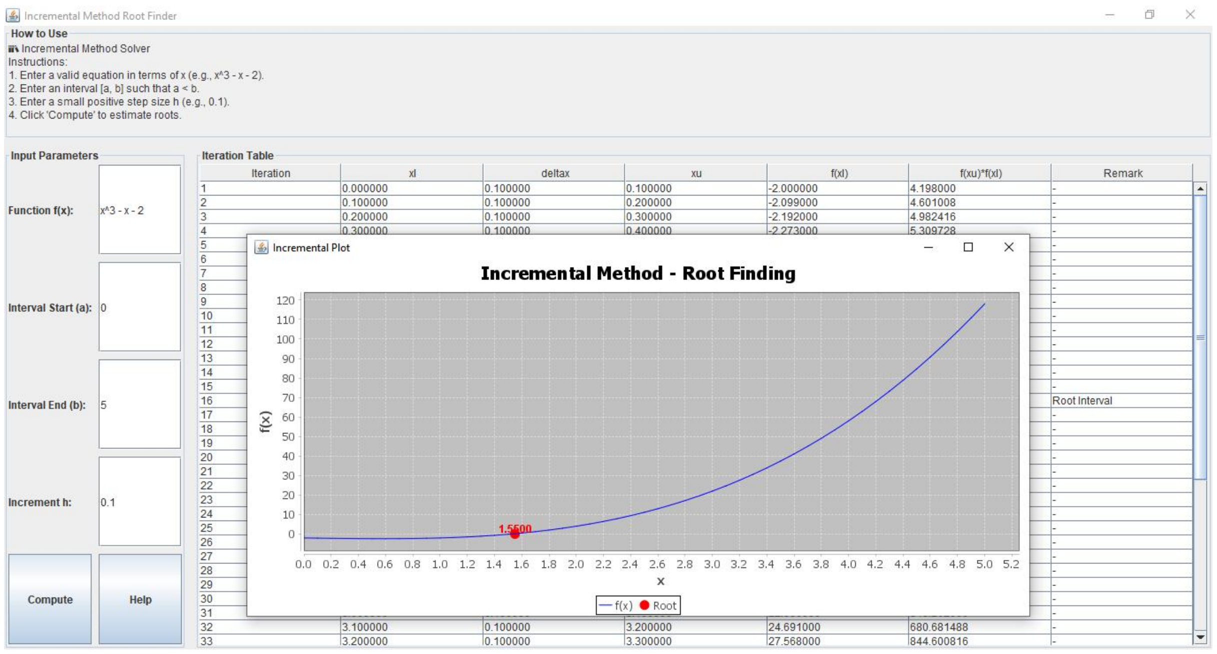
On pressing **ENTER**, input is sent to IGRAPHT.java, which performs the computation and visualization

- String function = equationf.getText(); // EQUATION TEXT FIELD
- Double start = Double.parseDouble(lowerf.getText()); // LOWER BOUND TEXT FIELD
- double end = Double.parseDouble(upperf.getText()); // UPPER BOUND TEXT FIELD
- double deltaX = Double.parseDouble(deltaf.getText()); // DELTA X TEXT FIELD
- double tolerance = 0.0001;
- IGRAPHT igraphtWindow = new IGRAPHT(function, start, end, deltaX, tolerance);

### *Output GUI: IGRAPHT.java*

Displays:

- A table showing: Xl, Δx, f(Xl), f(Xu) * f(Xl), and Remarks
- A graph of f(x)f(x)f(x) across the interval, with:
- Red dots for roots
- Blue rectangles for bounds



Incremental Method Root Finder

**How to Use**

Incremental Method Solver
Instructions:
1. Enter a valid equation in terms of x (e.g., x^3 - x - 2).
2. Enter an interval [a, b] such that a < b.
3. Enter a small positive step size h (e.g., 0.1).
4. Click 'Compute' to estimate roots.

**Input Parameters**

| | |
|---|---|
| **Function f(x):** | x^3 - x - 2 |
| **Interval Start (a):** | 0 |
| **Interval End (b):** | 5 |
| **Increment h:** | 0.1 |

**Iteration Table**

| Iteration | xl | deltax | xu | f(xl) | f(xu)*f(xl) | Remark |
|---|---|---|---|---|---|---|
| 1 | 0.000000 | 0.100000 | 0.100000 | -2.000000 | 4.198000 | - |
| 2 | 0.100000 | 0.100000 | 0.200000 | -2.099000 | 4.601008 | - |
| 3 | 0.200000 | 0.100000 | 0.300000 | -2.192000 | 4.982416 | - |
| 4 | 0.300000 | 0.100000 | 0.400000 | -2.273000 | 5.309728 | - |
| 5 | | | | | | - |
| 6 | | | | | | - |
| 7 | | | | | | - |
| 8 | | | | | | - |
| 9 | | | | | | - |
| 10 | | | | | | - |
| 11 | | | | | | - |
| 12 | | | | | | - |
| 13 | | | | | | - |
| 14 | | | | | | - |
| 15 | | | | | | - |
| 16 | | | | | | Root Interval |
| 17 | | | | | | - |
| 18 | | | | | | - |
| 19 | | | | | | - |
| 20 | | | | | | - |
| 21 | | | | | | - |
| 22 | | | | | | - |
| 23 | | | | | | - |
| 24 | | | | | | - |
| 25 | | | | | | - |
| 26 | | | | | | - |
| 27 | | | | | | - |
| 28 | | | | | | - |
| 29 | | | | | | - |
| 30 | | | | | | - |
| 31 | | | | | | - |
| 32 | 3.100000 | 0.100000 | 3.200000 | 24.691000 | 680.681488 | - |
| 33 | 3.200000 | 0.100000 | 3.300000 | 27.568000 | 844.600816 | - |

### *IncrementalSolver.java*

- Performs all core computations for the Incremental Search Method and root refinement using Bisection.
- Key Responsibilities:
- Parses and evaluates mathematical functions using exp4j
- Iteratively scans the interval [start,end][start, end][start,end] with step size Δx
- Detects root brackets via sign change: f(xl).f(xu) < 0
- Refines bracketed roots using an internal bisection method
- Prevents duplicate roots using a tolerance-based uniqueness check

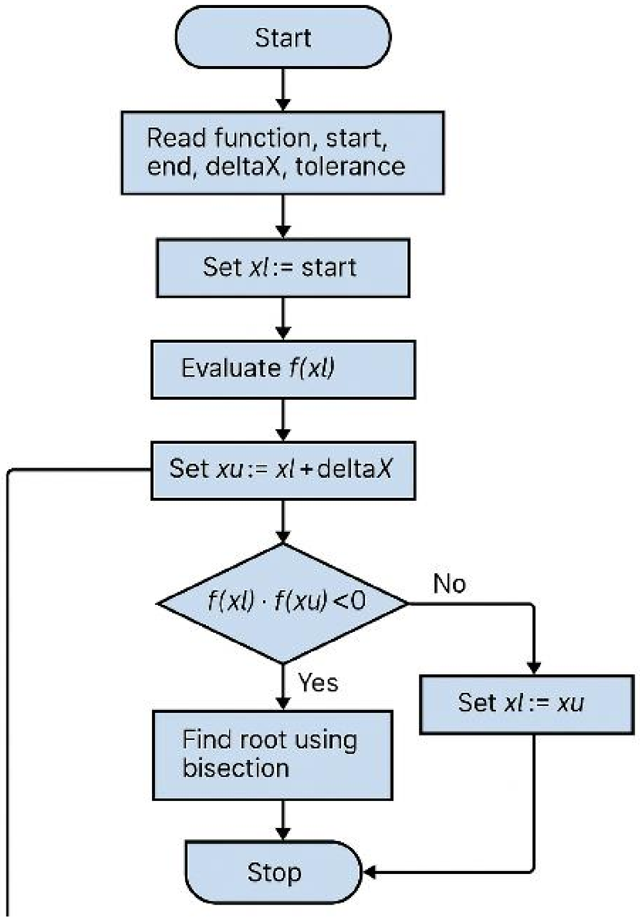
### *Incremental.java*

- Input fields for start, end, equation, and Δx
- Interactive numpad for user-friendly input
- Captures and validates user inputs
- On ENTER, parses inputs and launches IGRAPHT for visualization

### *Igrapht.java*

- Displays a table showing:

- xl,xu,f(xl),f(xu),f(xl)·f(xu)x_l, x_u, f(x_l), f(x_u), f(x_l) \cdot f(x_u)xl ,xu ,f(xl ),f(xu ),f(xl )·f(xu ), and remarks
- Calls IncrementalSolver to:
- Build iteration table
- Find and refine roots
- Plots:
- The function f(x)f(x)f(x)
- Red dots for detected roots
- Blue squares for limits
- Graphing Tool:
- Uses JFreeChart + JCommon

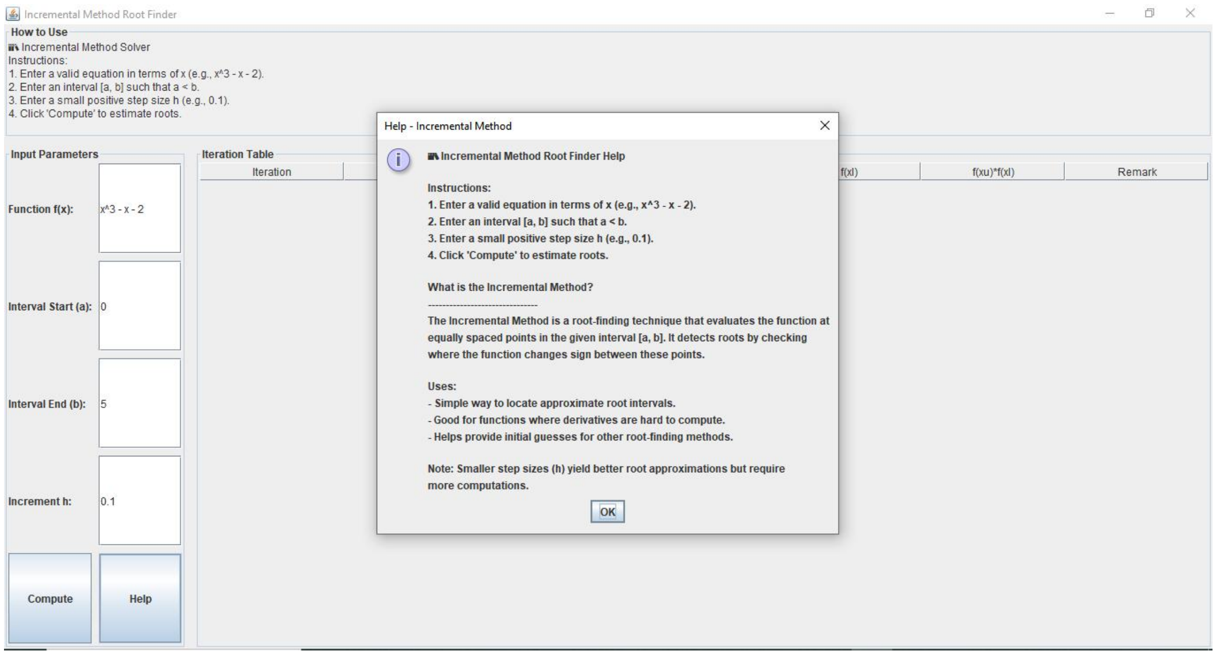


```
Input:
  - Function f(x)
  - Lower bound: start
  - Upper bound: end
  - Increment step size: deltaX
  - Tolerance for root refinement: tol

Initialize:
  xl = start
  xu = xl + deltaX
  roots = empty list

WHILE xu <= end DO
  Evaluate f(xl) → fxl
  Evaluate f(xu) → fxu

  IF fxl * fxu < 0 THEN
    // Sign change detected → root lies between xl and xu
    root = RefineRootBisection(f, xl, xu, tol)
    IF root is not already in roots (within tolerance) THEN
      Add root to roots
    ENDIF
  ENDIF

  xl = xu
  xu = xl + deltaX
ENDWHILE

Output: List of roots found within [start, end]
```

-----------------------------------

```
FUNCTION RefineRootBisection(f, xl, xu, tol)
  REPEAT
    xr = (xl + xu) / 2
    IF f(xl) * f(xr) < 0 THEN
      xu = xr
    ELSE
      xl = xr
    ENDIF
  UNTIL |xu - xl| < tol

  RETURN (xl + xu) / 2
END FUNCTION
```

- The method scans the interval from start to end in steps of deltaX.
- At each step, it checks for a sign change between consecutive points, indicating a root.
- When a sign change is detected, it uses the bisection method to refine the root location within the bracket to the desired tolerance.
- Roots are stored in a list, avoiding duplicates within the specified tolerance.
- The process repeats until the entire interval has been searched.



Incremental Method Solver
Instructions:
1. Enter a valid equation in terms of x (e.g., x^3 - x - 2).
2. Enter an interval [a, b] such that a < b.
3. Enter a small positive step size h (e.g., 0.1).
4. Click 'Compute' to estimate roots.

Function f(x): x^3 - x - 2
Interval Start (a): 0
Interval End (b): 5
Increment h: 0.1

Help - Incremental Method

**Incremental Method Root Finder Help**

**Instructions:**
**1. Enter a valid equation in terms of x (e.g., x^3 - x - 2).**
**2. Enter an interval [a, b] such that a < b.**
**3. Enter a small positive step size h (e.g., 0.1).**
**4. Click 'Compute' to estimate roots.**

**What is the Incremental Method?**
**------------------------------**
**The Incremental Method is a root-finding technique that evaluates the function at equally spaced points in the given interval [a, b]. It detects roots by checking where the function changes sign between these points.**

**Uses:**
**- Simple way to locate approximate root intervals.**
**- Good for functions where derivatives are hard to compute.**
**- Helps provide initial guesses for other root-finding methods.**

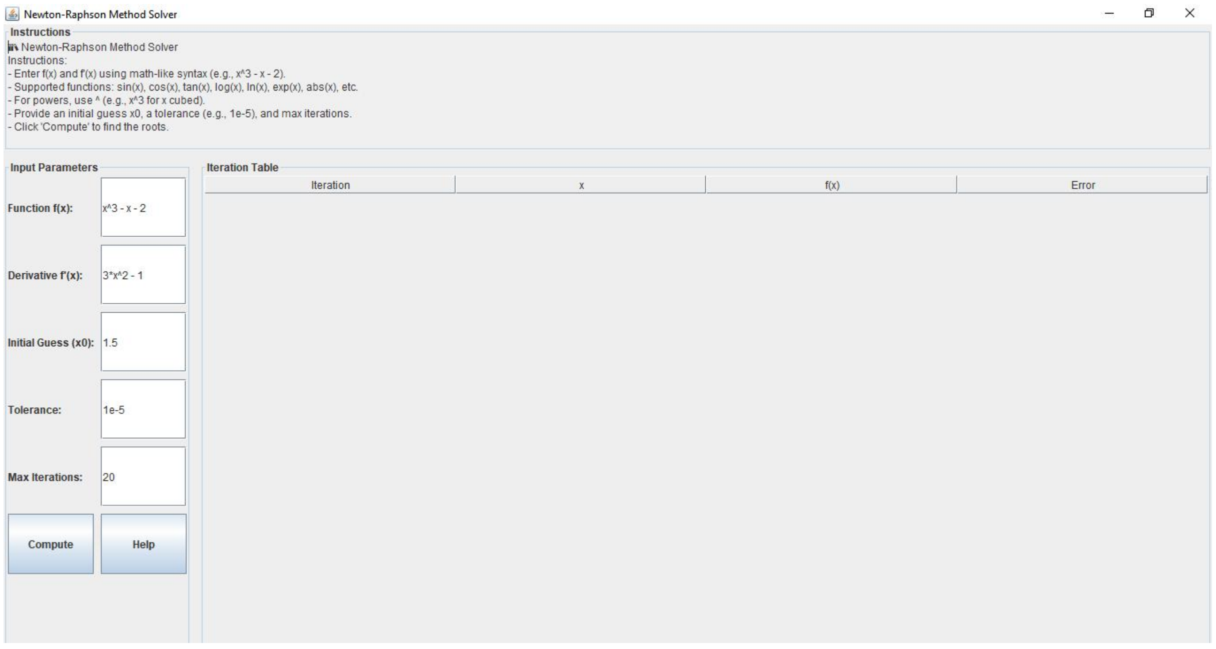
**Note: Smaller step sizes (h) yield better root approximations but require more computations.**

# NEWTON-RHAPSON METHOD

## *NEWTON-RHAPSON METHOD IN JAVA*

The Newton-Raphson Method is an iterative numerical technique used to find roots of a function f(x)=0f(x) = 0f(x)=0. Starting from an initial guess x0x_0x0 , the method uses the function and its derivative f′(x)f'(x)f′(x) to generate successively better approximations of the root. Each iteration updates the guess by its formula: xn+1 =xn – f(xn)/f’(xn)

until the change between iterations is less than a specified tolerance or a maximum number of iterations is reached. The method typically converges quickly when the initial guess is close to the actual root, but it requires the derivative to be non-zero and the function to be differentiable.



Newton-Raphson Method Solver

Instructions

Newton-Raphson Method Solver
Instructions:
- Enter f(x) and f'(x) using math-like syntax (e.g., x^3 - x - 2).
- Supported functions: sin(x), cos(x), tan(x), log(x), ln(x), exp(x), abs(x), etc.
- For powers, use ^ (e.g., x^3 for x cubed).
- Provide an initial guess x0, a tolerance (e.g., 1e-5), and max iterations.
- Click 'Compute' to find the roots.

Input Parameters

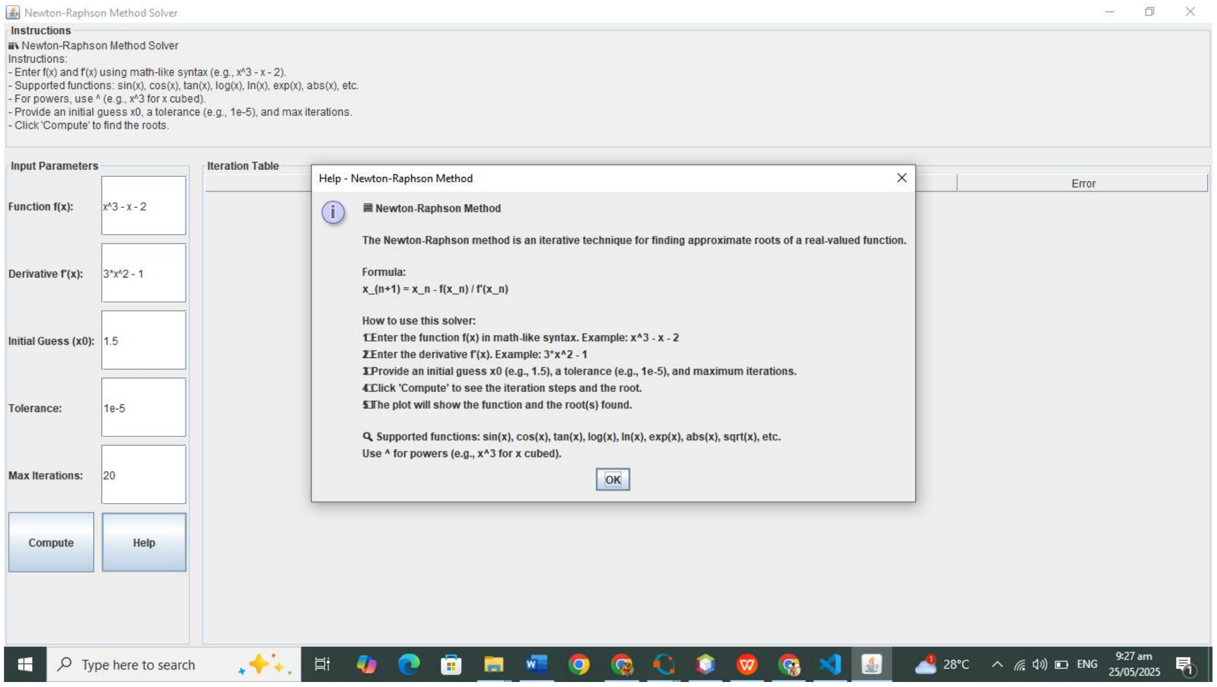
Function f(x): x^3 - x - 2
Derivative f'(x): 3*x^2 - 1
Initial Guess (x0): 1.5
Tolerance: 1e-5
Max Iterations: 20

Iteration Table

| Iteration | x | f(x) | Error |
|---|---|---|---|

### *Input Fields in the Newton-Raphson Program*

- ***Equation (f(x)) Field***: Input for the mathematical function whose root is to be found. Supports standard operators and functions, with input assistance via a custom numpad and function buttons (sin, cos, tan).
- ***Derivative (f′(x)) Field***: Input for the derivative of the function f′(x)f'(x)f′(x), necessary for iteration calculations.
- Initial Guess ($X_0$) Field: User-provided starting point for the iteration process. A good initial guess is critical for convergence.
- ***Step Size (Tolerance) Field***: Specifies the precision threshold at which the iterative process will stop.
- ***Numpad and Function Buttons***: Buttons allow easy insertion of digits, operators, and common functions into the active input field. The interface supports cursor positioning and backspace for editing.
- ***Clear and Delete Buttons***: For correcting input errors by clearing entire fields or deleting the last character.



Newton-Raphson Method Solver

Instructions

Newton-Raphson Method Solver
Instructions:
- Enter f(x) and f'(x) using math-like syntax (e.g., x^3 - x - 2).
- Supported functions: sin(x), cos(x), tan(x), log(x), ln(x), exp(x), abs(x), etc.
- For powers, use ^ (e.g., x^3 for x cubed).
- Provide an initial guess x0, a tolerance (e.g., 1e-5), and max iterations.
- Click 'Compute' to find the roots.

Input Parameters

Function f(x): x^3 - x - 2

Derivative f'(x): 3*x^2 - 1

Initial Guess (x0): 1.5

Tolerance: 1e-5

Max Iterations: 20

Help - Newton-Raphson Method

Newton-Raphson Method

The Newton-Raphson method is an iterative technique for finding approximate roots of a real-valued function.

Formula:
x_(n+1) = x_n - f(x_n) / f'(x_n)

How to use this solver:
1 Enter the function f(x) in math-like syntax. Example: x^3 - x - 2
2 Enter the derivative f'(x). Example: 3*x^2 - 1
3 Provide an initial guess x0 (e.g., 1.5), a tolerance (e.g., 1e-5), and maximum iterations.
4 Click 'Compute' to see the iteration steps and the root.
5 The plot will show the function and the root(s) found.

Supported functions: sin(x), cos(x), tan(x), log(x), ln(x), exp(x), abs(x), sqrt(x), etc.
Use ^ for powers (e.g., x^3 for x cubed).

### *1. NEWTONRHAPSON.java*

This class provides the GUI interface for the Newton-Raphson method. It contains text fields for the function, its derivative, initial guess, and tolerance, supported by a custom numpad and function buttons for easier input. The class handles user input validation, invokes the solver class on submission, and launches the results visualization window. It emphasizes usability and robust input management.

### *2. NewtonSolver.java*

Encapsulates the core Newton-Raphson algorithm. It uses the Exp4j library to parse and evaluate both the function and its derivative at given points. It iteratively computes root approximations until the stopping criterion is met. It returns a list of iteration details including current guess, function values, derivatives, and approximate errors, facilitating detailed analysis of convergence.

### *3. NRGRAPH.java*

This class handles displaying the results of the Newton-Raphson iterations. It shows a table listing iteration number, approximation, error, function value, and derivative at each step. It also plots the function curve with highlighted iteration points and the final root approximation, using JFreeChart. This visualization aids users in understanding the iteration behavior and verifying convergence.

START

|

V

Input f(x), f'(x), initial guess x0, tolerance tol, max iterations maxIter

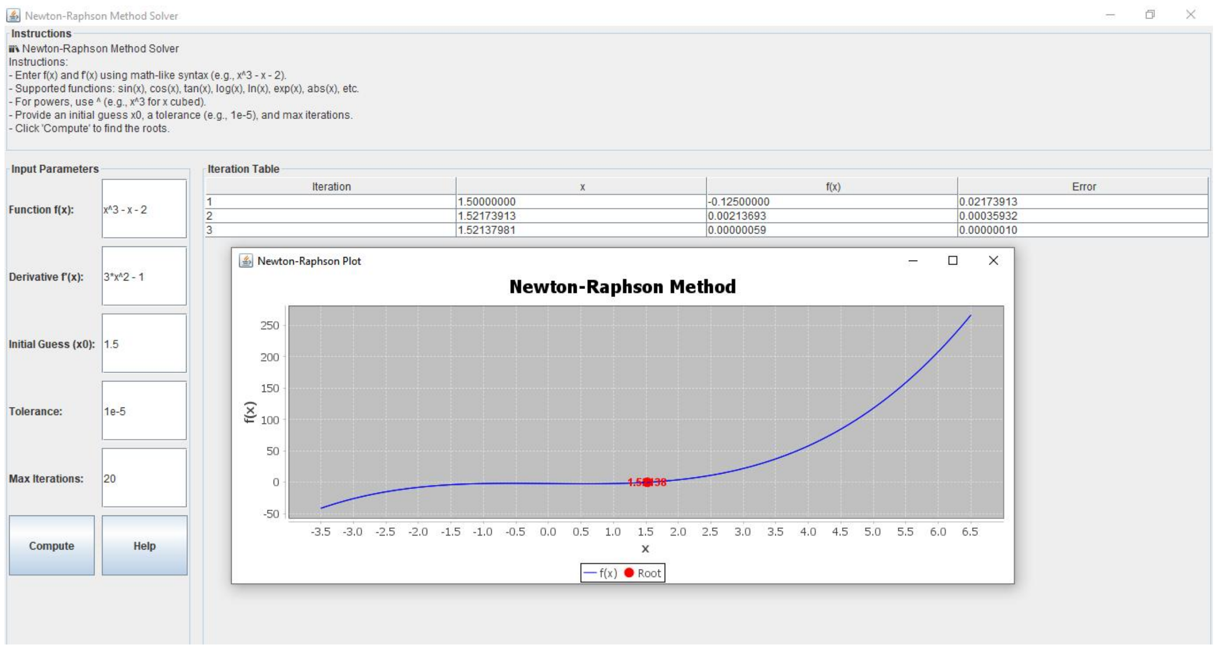
|

```
V

Initialize: i = 0, xi = x0, error = large number

|

V

Calculate f(xi), f'(xi)

|

V

Is f'(xi) = 0?

|-- Yes --> STOP with error "Derivative zero"

|

|-- No -->

|

Calculate xi+1 = xi - f(xi) / f'(xi)

|

V

Calculate error = |xi+1 - xi|

|

V

Increment i by 1

|

V

Is error < tol OR i >= maxIter?

|-- Yes --> Output xi+1 and iterations, STOP

|

|-- No -->

|

Assign xi = xi+1

|

V

Repeat calculation
```

The Newton-Raphson method provides an efficient and powerful technique for finding roots of nonlinear equations, leveraging both the function and its derivative to achieve rapid convergence. Through this project, implementing the Newton-Raphson method enhanced my understanding of iterative algorithms and the importance of good initial guesses and derivative calculations in numerical analysis.

The interactive program developed offers users a practical tool to experiment with this method, providing detailed iteration data and graphical visualization that aids in comprehending the convergence process. While the method is highly effective under suitable conditions, the project also highlights potential pitfalls such as derivative zeros and divergence, emphasizing the need for careful input and error handling.

Overall, this implementation not only solidifies the theoretical concepts but also equips users with a hands-on experience, bridging mathematical theory and real-world computational applications in computer engineering.



Newton-Raphson Method Solver
Instructions:
- Enter f(x) and f'(x) using math-like syntax (e.g., x^3 - x - 2).
- Supported functions: sin(x), cos(x), tan(x), log(x), ln(x), exp(x), abs(x), etc.
- For powers, use ^ (e.g., x^3 for x cubed).
- Provide an initial guess x0, a tolerance (e.g., 1e-5), and max iterations.
- Click 'Compute' to find the roots.

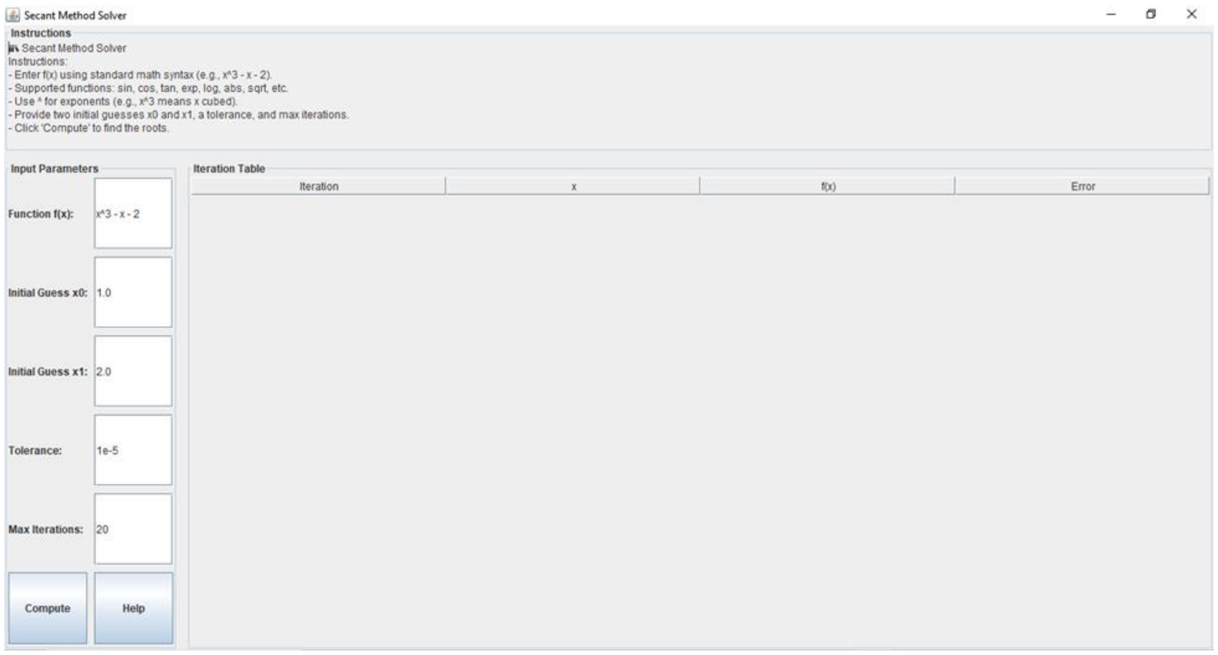
| | |
|---|---|
| Function f(x): | x^3 - x - 2 |
| Derivative f'(x): | 3*x^2 - 1 |
| Initial Guess (x0): | 1.5 |
| Tolerance: | 1e-5 |
| Max Iterations: | 20 |

| Iteration | x | f(x) | Error |
|---|---|---|---|
| 1 | 1.50000000 | -0.12500000 | 0.02173913 |
| 2 | 1.52173913 | 0.00213693 | 0.00035932 |
| 3 | 1.52137981 | 0.00000059 | 0.00000010 |

# SECANT METHOD

## *SECANT METHOD IN JAVA*

The Secant Method is an iterative root-finding technique that approximates the derivative using two initial guesses rather than requiring an explicit derivative function. It generates successive approximations to a root of f(x)=0f(x) = 0f(x)=0 by drawing secant lines through the points (xi-1,f(xi-1)) and (xi, f(xi)) and finding their intersection with the x-axis. The formula for the next approximation is: xi+1 =xi −f(xi ) * xi-xi-1/f(xi)-f(xi-1) This method can converge faster than the Bisection method and does not require the calculation of derivatives, but it needs two initial approximations and may fail if the function values at the guesses are very close.



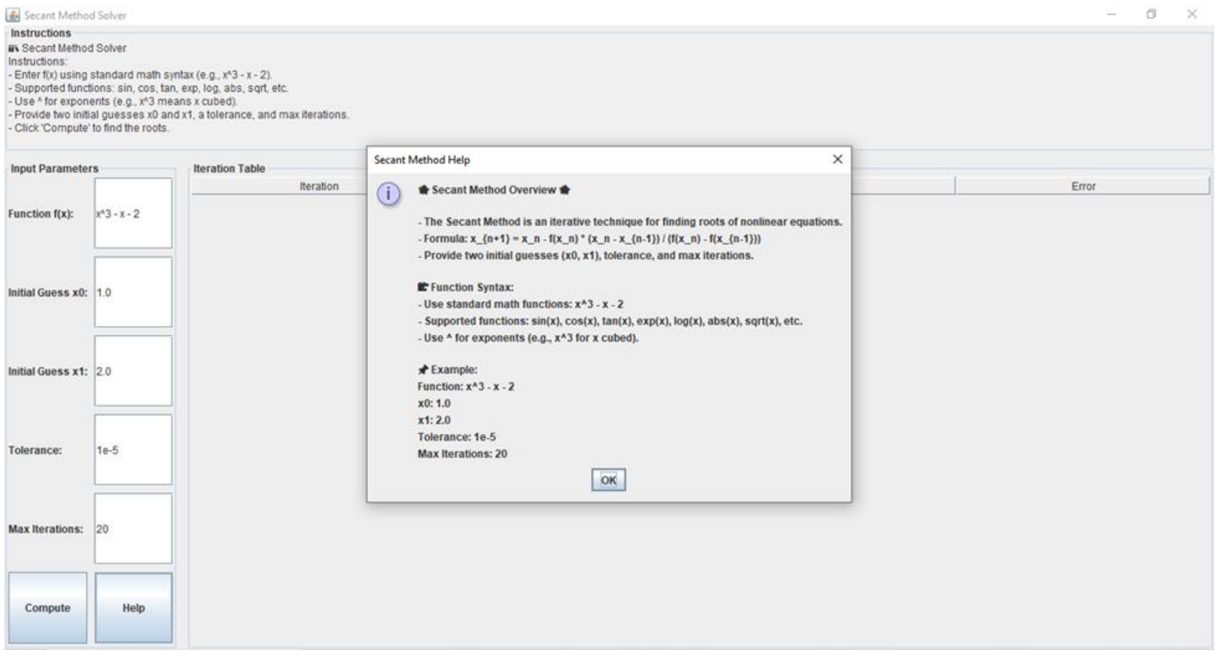
Secant Method Solver
Instructions:
- Enter f(x) using standard math syntax (e.g., x^3 - x - 2).
- Supported functions: sin, cos, tan, exp, log, abs, sqrt, etc.
- Use ^ for exponents (e.g., x^3 means x cubed).
- Provide two initial guesses x0 and x1, a tolerance, and max iterations.
- Click 'Compute' to find the roots.

Function f(x): x^3 - x - 2
Initial Guess x0: 1.0
Initial Guess x1: 2.0
Tolerance: 1e-5
Max Iterations: 20

| Iteration | x | f(x) | Error |
|---|---|---|---|

### *Input Fields in SECAN GUI*

- ***Equation Field:*** Text input for the function f(x)f(x)f(x) whose root is to be found. Supports standard mathematical functions and operators, aided by function buttons like sin, cos, tan, and a numpad for easier entry.
- ***Initial Guess $X_0$ Field:*** The first initial approximation of the root.
- ***Initial Guess $X_1$ Field:*** The second initial approximation.
- ***Step Size (Tolerance) Field:*** Defines the stopping criterion accuracy.
- ***Clear and Delete Buttons:*** Allow users to correct input.
- ***Need Help Button:*** Provides guidance on how to properly use the method and enter data.


Secant Method Solver

Instructions
Secant Method Solver
Instructions:
- Enter f(x) using standard math syntax (e.g., x^3 - x - 2).
- Supported functions: sin, cos, tan, exp, log, abs, sqrt, etc.
- Use ^ for exponents (e.g., x^3 means x cubed).
- Provide two initial guesses x0 and x1, a tolerance, and max iterations.
- Click 'Compute' to find the roots.

Input Parameters

| | |
|---|---|
| Function f(x): | x^3 - x - 2 |
| Initial Guess x0: | 1.0 |
| Initial Guess x1: | 2.0 |
| Tolerance: | 1e-5 |
| Max Iterations: | 20 |

Secant Method Help

**Secant Method Overview**

- The Secant Method is an iterative technique for finding roots of nonlinear equations.
- Formula: x_{n+1} = x_n - f(x_n) * (x_n - x_{n-1}) / (f(x_n) - f(x_{n-1}))
- Provide two initial guesses (x0, x1), tolerance, and max iterations.

**Function Syntax:**
- Use standard math functions: x^3 - x - 2
- Supported functions: sin(x), cos(x), tan(x), exp(x), log(x), abs(x), sqrt(x), etc.
- Use ^ for exponents (e.g., x^3 for x cubed).

**Example:**
Function: x^3 - x - 2
x0: 1.0
x1: 2.0
Tolerance: 1e-5
Max Iterations: 20

```
START
|
V
Input: function f(x), initial guesses x0, x1, tolerance tol, max iterations maxIter
|
V
Initialize i = 1
Calculate f(x0) and f(x1)
|
V
WHILE i <= maxIter DO
|
V
IF |f(x1) - f(x0)| < very small number THEN
STOP (Prevent division by zero)
ENDIF
|
V
```

```
Compute x2 = x1 - f(x1) * (x1 - x0) / (f(x1) - f(x0))
Calculate error = |x2 - x1|
|
V
Store iteration data: i, x0, x1, x2, error, f(x0), f(x1), f(x2)
|
V
IF error < tol THEN
STOP and output root approximation
ENDIF
|
V
Update: x0 = x1, f(x0) = f(x1)
Update: x1 = x2, f(x1) = f(x2)
Increment i by 1
|
V
ENDWHILE
|
V
Output last approximation and iterations
END
```

***SECAN.java (Input GUI)***

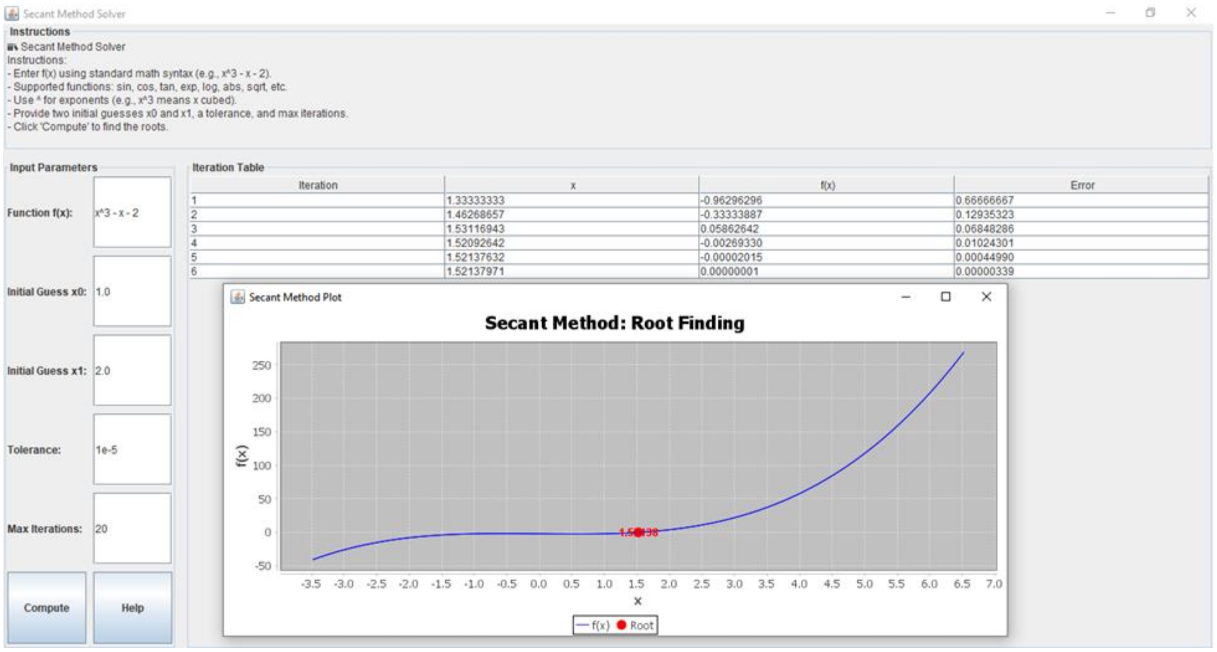
This class provides the user interface for entering the function, two initial guesses, and the tolerance for the Secant Method. It includes function buttons and a numpad to assist user input. Input validation and error handling ensure proper entries before computation. Upon submission, it calls the solver and launches the output display GUI.

***SecantSolver.java***

Implements the Secant Method algorithm. It parses the function expression using Exp4j with support for exponentiation and trigonometric functions. The solver performs iterative root approximations until the specified tolerance or maximum iterations are reached, storing each iteration's data (guesses, function values, error) for review.

***SGRAPH.java (Output GUI)***

Displays the results of the Secant Method iterations both as a detailed table and a graph. The table shows all iteration data, while the graph plots the function curve, secant iteration points, and final root approximation, using JFreeChart. Visual annotations enhance clarity and understanding of the convergence process.



Instructions
Secant Method Solver
Instructions:
- Enter f(x) using standard math syntax (e.g., x^3 - x - 2).
- Supported functions: sin, cos, tan, exp, log, abs, sqrt, etc.
- Use ^ for exponents (e.g., x^3 means x cubed).
- Provide two initial guesses x0 and x1, a tolerance, and max iterations.
- Click 'Compute' to find the roots.

Input Parameters

| | |
|---|---|
| Function f(x): | x^3 - x - 2 |
| Initial Guess x0: | 1.0 |
| Initial Guess x1: | 2.0 |
| Tolerance: | 1e-5 |
| Max Iterations: | 20 |

Iteration Table

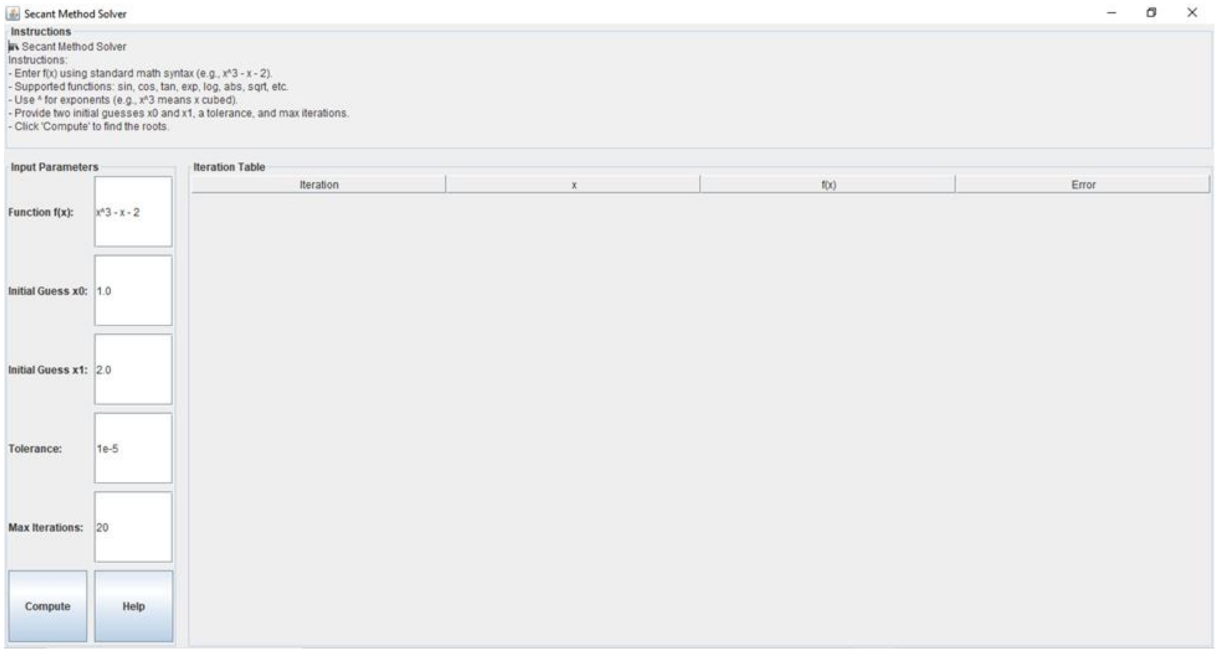
| Iteration | x | f(x) | Error |
|---|---|---|---|
| 1 | 1.33333333 | -0.96296296 | 0.66666667 |
| 2 | 1.46268657 | -0.33333887 | 0.12935323 |
| 3 | 1.53116943 | 0.05862642 | 0.06848286 |
| 4 | 1.52092642 | -0.00269330 | 0.01024301 |
| 5 | 1.52137632 | -0.00002015 | 0.00044990 |
| 6 | 1.52137971 | 0.00000001 | 0.00000339 |

# REGULA FALSUI METHOD

## *REGULA FALSI IN JAVA*

The Regula Falsi Method (also known as the False Position Method) is a numerical technique used to find the root of a continuous function. It is a hybrid of the bisection method and secant method, using a combination of interval halving and linear interpolation.

In each iteration, the method refines the interval by calculating a new approximation to the root based on the values of the function at the two endpoints of the interval. This method assumes that the function behaves linearly between the two endpoints and computes the intersection of the line with the x-axis as the new root approximation.



Secant Method Solver

**Instructions**

Secant Method Solver
Instructions:
- Enter f(x) using standard math syntax (e.g., x^3 - x - 2).
- Supported functions: sin, cos, tan, exp, log, abs, sqrt, etc.
- Use ^ for exponents (e.g., x^3 means x cubed).
- Provide two initial guesses x0 and x1, a tolerance, and max iterations.
- Click 'Compute' to find the roots.

**Input Parameters**

| | |
|---|---|
| Function f(x): | x^3 - x - 2 |
| Initial Guess x0: | 1.0 |
| Initial Guess x1: | 2.0 |
| Tolerance: | 1e-5 |
| Max Iterations: | 20 |

**Iteration Table**

| Iteration | x | f(x) | Error |
|---|---|---|---|

### *Input Fields in REGULAFALSI.java GUI*

***Equation Field***: A text field where the user can enter the function f(x)f(x)f(x) to be solved for. This supports mathematical functions and operators, with built-in assistance from function buttons and a numpad.

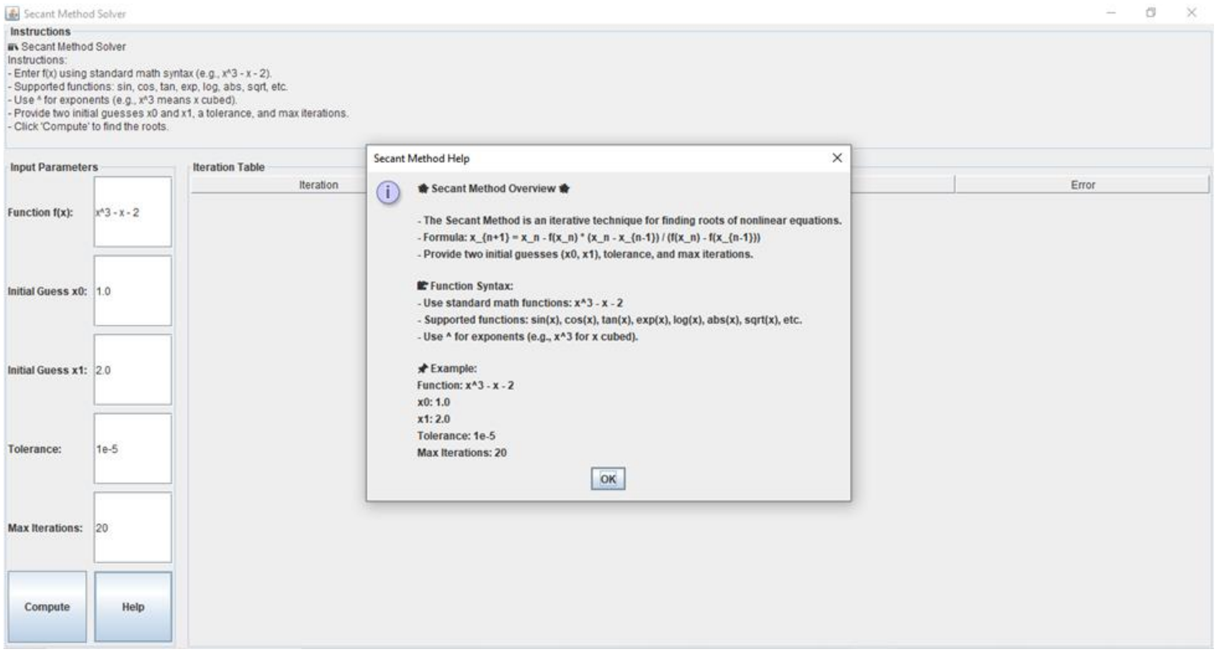
***Lower Bound ($X_l$) Field***: A text field for entering the lower bound of the interval where the root is expected. The function at this bound, f(Xl)f($X_l$)f(Xl ), should have a different sign from the function at the upper bound.

***Upper Bound ($X_u$) Field***: A text field for entering the upper bound of the interval where the root is expected. The function at this bound, f(Xu)f($X_u$)f(Xu ), should have a different sign from the function at the lower bound.

***Max Iterations Field***: A field for specifying the maximum number of iterations the algorithm should perform. This is a safeguard to prevent the algorithm from running indefinitely.

***Clear and Delete Buttons***: These buttons allow the user to clear inputs for correcting mistakes. The "Clear" button removes the last entered character, and the "Delete" button clears all fields.

***Need Help Button***: A button that provides an explanation of how to use the Regula Falsi method, including how to enter the function and bounds, as well as details on the method's behavior.



Secant Method Solver

**Instructions**

Secant Method Solver
Instructions:
- Enter f(x) using standard math syntax (e.g., x^3 - x - 2).
- Supported functions: sin, cos, tan, exp, log, abs, sqrt, etc.
- Use ^ for exponents (e.g., x^3 means x cubed).
- Provide two initial guesses x0 and x1, a tolerance, and max iterations.
- Click 'Compute' to find the roots.

**Input Parameters**

| | |
|---|---|
| Function f(x): | x^3 - x - 2 |
| Initial Guess x0: | 1.0 |
| Initial Guess x1: | 2.0 |
| Tolerance: | 1e-5 |
| Max Iterations: | 20 |

**Iteration Table**

| Iteration | Error |
|---|---|

Secant Method Help

**Secant Method Overview**

- The Secant Method is an iterative technique for finding roots of nonlinear equations.
- Formula: x_{n+1} = x_n - f(x_n) * (x_n - x_{n-1}) / (f(x_n) - f(x_{n-1}))
- Provide two initial guesses (x0, x1), tolerance, and max iterations.

Function Syntax:
- Use standard math functions: x^3 - x - 2
- Supported functions: sin(x), cos(x), tan(x), exp(x), log(x), abs(x), sqrt(x), etc.
- Use ^ for exponents (e.g., x^3 for x cubed).

Example:
Function: x^3 - x - 2
x0: 1.0
x1: 2.0
Tolerance: 1e-5
Max Iterations: 20

START

|

V

Input: f(x), lower bound xL, upper bound xU, tolerance tol, max iterations maxIter

|

V

Initialize: i = 1, xL, xU, and tolerance

|

V

WHILE i <= maxIter DO

|

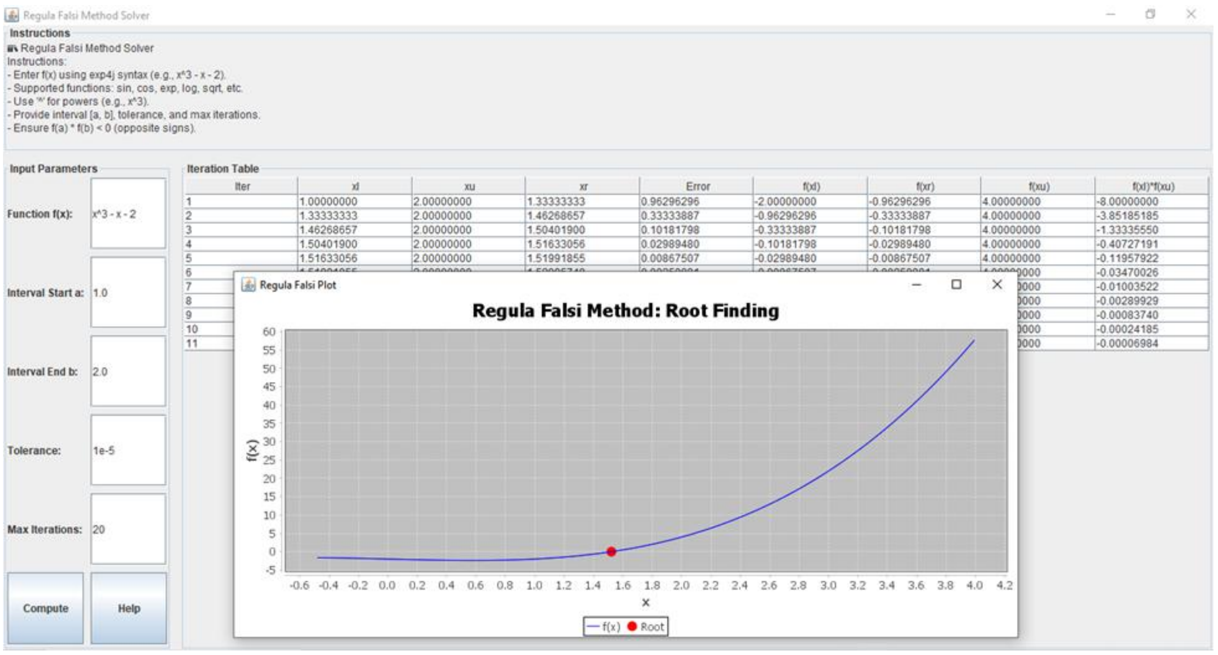
V

Calculate f(xL), f(xU)

|

V

```
Is f(xL) * f(xU) < 0?
|-- Yes --> Continue to next step
|-- No  --> STOP: f(xL) and f(xU) must have opposite signs
|
V
Calculate root approximation:
xR = xU - f(xU) * (xL - xU) / (f(xL) - f(xU))
|
V
Calculate error:
error = |xR - xU|
|
V
Store iteration data: i, xL, xU, xR, error, f(xL), f(xU), f(xR)
|
V
Is error < tol?
|-- Yes --> Output root approximation and iteration data
|-- No  --> Update: xL = xU, xU = xR, i = i + 1
|
V
END WHILE
|
V
Output: Root approximation xR and iteration data
END
```



Instructions

Regula Falsi Method Solver
Instructions:
- Enter f(x) using exp4j syntax (e.g., x^3 - x - 2).
- Supported functions: sin, cos, exp, log, sqrt, etc.
- Use '^' for powers (e.g., x^3).
- Provide interval [a, b], tolerance, and max iterations.
- Ensure f(a) * f(b) < 0 (opposite signs).

Input Parameters

| | |
|---|---|
| Function f(x): | x^3 - x - 2 |
| Interval Start a: | 1.0 |
| Interval End b: | 2.0 |
| Tolerance: | 1e-5 |
| Max Iterations: | 20 |

Iteration Table

| Iter | xl | xu | xr | Error | f(xl) | f(xr) | f(xu) | f(xl)*f(xu) |
|---|---|---|---|---|---|---|---|---|
| 1 | 1.00000000 | 2.00000000 | 1.33333333 | 0.96296296 | -2.00000000 | -0.96296296 | 4.00000000 | -8.00000000 |
| 2 | 1.33333333 | 2.00000000 | 1.46268657 | 0.33333887 | -0.96296296 | -0.33333887 | 4.00000000 | -3.85185185 |
| 3 | 1.46268657 | 2.00000000 | 1.50401900 | 0.10181798 | -0.33333887 | -0.10181798 | 4.00000000 | -1.33335550 |
| 4 | 1.50401900 | 2.00000000 | 1.51633056 | 0.02989480 | -0.10181798 | -0.02989480 | 4.00000000 | -0.40727191 |
| 5 | 1.51633056 | 2.00000000 | 1.51991855 | 0.00867507 | -0.02989480 | -0.00867507 | 4.00000000 | -0.11957922 |
| 6 | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | -0.03470026 |
| 7 | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | -0.01003522 |
| 8 | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | -0.00289929 |
| 9 | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | -0.00083740 |
| 10 | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | -0.00024185 |
| 11 | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | -0.00006984 |

***REGULAFALSI.java (Input GUI)***

This class provides the graphical user interface (GUI) for the Regula Falsi Method. It contains text fields for the user to input the function f(x)f(x)f(x), lower and upper bounds of the interval xLx_LxL and xUx_UxU , and the maximum number of iterations. The GUI includes function buttons (e.g., sin, cos, tan) and a numpad to assist users in entering valid mathematical expressions.

The input is validated to ensure that the bounds have opposite signs for f(x)f(x)f(x), and the method proceeds to call the solver and generate the results upon user input.

***RegulaSolver.java***

The core of the Regula Falsi Method implementation, this class handles the root-finding logic. It parses the input function using Exp4j and computes the root approximation iteratively. The method calculates new root approximations using the formula for Regula Falsi and stores each iteration's data. The stopping condition is based on error tolerance or the maximum number of iterations.

The class also handles validation to ensure that the function changes signs between the lower and upper bounds before proceeding.

***RegulaGraph.java (Output GUI)***

This class handles the output display of the results. It shows a JTable with iteration data, including lower and upper bounds, approximations, function values, and errors. It also generates a plot of the function f(x)f(x)f(x) with markers for the initial bounds and approximated roots using JFreeChart.

Each iteration point is plotted as a red marker, and the final root approximation is marked in orange, making it easy to visualize the method's progress. The graph is updated dynamically as the iterations proceed.

# GRAPHICAL METHOD

***GRAPHICAL METHOD IN JAVA***

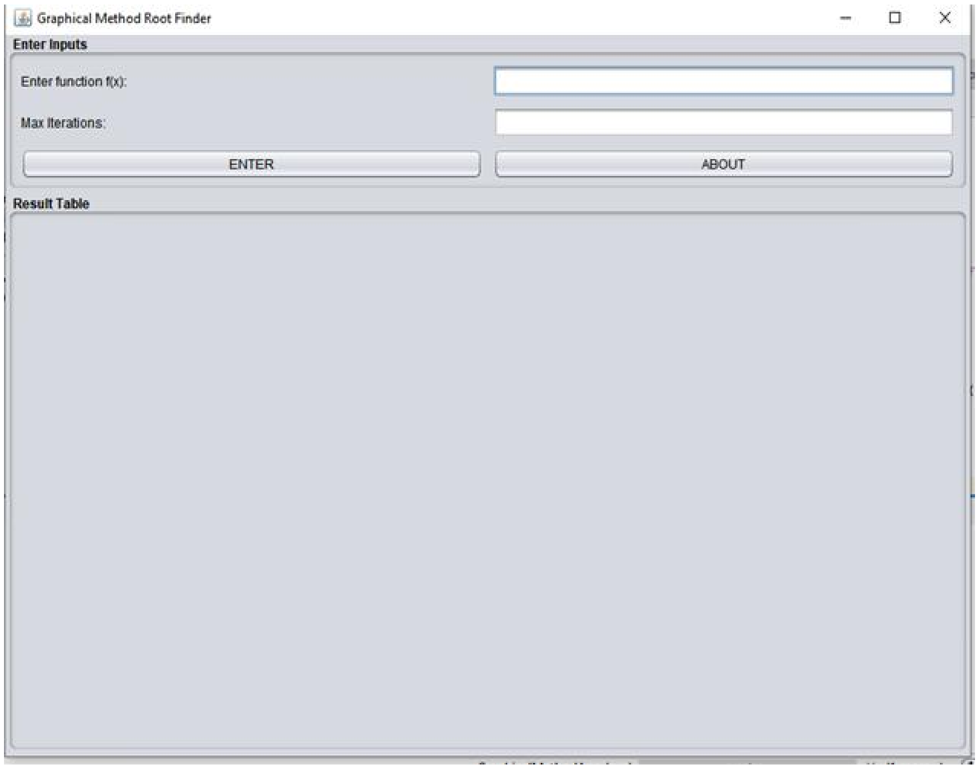
The Graphical Method is a root-finding technique that graphically approximates the roots of a function f(x)f(x)f(x) by plotting it and visually identifying where the curve intersects the x-axis. This method is primarily used to find initial approximations of roots that can later be refined using more precise methods, such as the Bisection Method or Newton-Raphson Method.

In this method:

The function is plotted within a specified range.

Points where the curve crosses the x-axis (roots) are identified visually.

The method provides an intuitive way to understand the behavior of a function and estimate where its roots may lie.



***Input Fields in GRAPHICAL GUI***

***Equation (f(x)) Field:***

This text field allows the user to input the mathematical expression of the function f(x)f(x)f(x) whose roots are to be approximated. The input supports various functions and operators, aided by a numpad and function buttons for sin, cos, tan, and more.

***Max Iterations Field:***

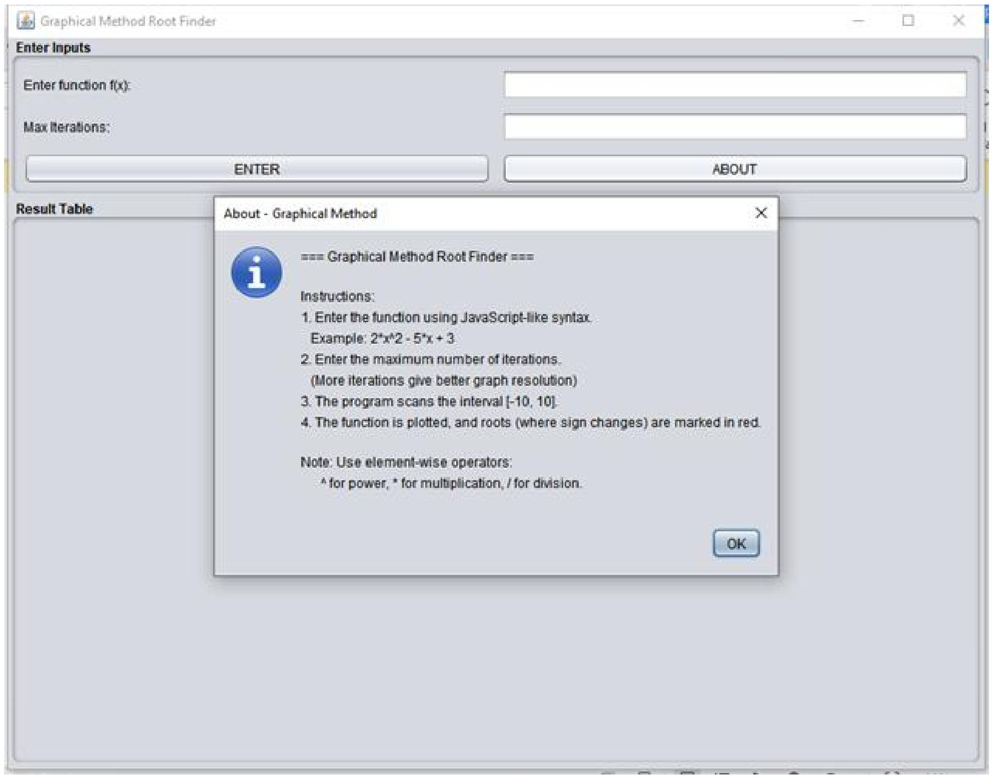
The number of iterations is used to define how finely the function is sampled during the plotting process. This field helps control the precision of the graphical search for roots.

***Clear and Delete Buttons***:

These buttons allow users to delete the last character or clear the entire field, providing an easy way to correct mistakes.

***Function Buttons:***

Buttons for common mathematical functions like sin, cos, and tan assist the user in inputting functions quickly and accurately.



```
=== Graphical Method Root Finder ===

Instructions:
1. Enter the function using JavaScript-like syntax.
   Example: 2*x^2 - 5*x + 3
2. Enter the maximum number of iterations.
   (More iterations give better graph resolution)
3. The program scans the interval [-10, 10].
4. The function is plotted, and roots (where sign changes) are marked in red.

Note: Use element-wise operators:
     ^ for power, * for multiplication, / for division.
```

START

|

V

Input: f(x), Max Iterations

|

V

Initialize: xStart = -10, xEnd = 10 (arbitrary range for initial plot), step = (xEnd - xStart) / Max Iterations

|

V

Calculate y = f(x) for each x from xStart to xEnd in steps of size 'step'

|

V

Store each (x, f(x)) pair in a list for table and graph plotting

|

V

Plot f(x) with x values on the horizontal axis and f(x) values on the vertical axis

|

V

Identify intersection points with the x-axis (roots)

|

V

Display roots and corresponding iterations in a table

|

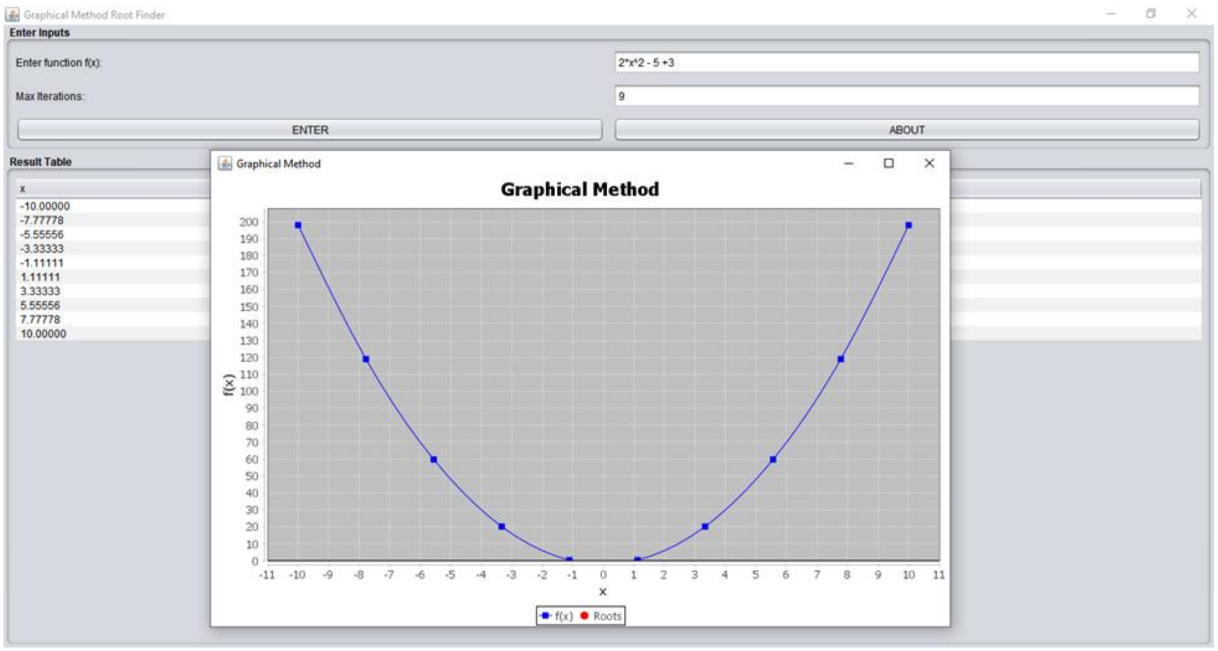
V

Output: List of roots found in the given range

|

V

END



Enter function f(x): 2*x^2 - 5 +3

Max Iterations: 9

| x |
|---|
| -10.00000 |
| -7.77778 |
| -5.55556 |
| -3.33333 |
| -1.11111 |
| 1.11111 |
| 3.33333 |
| 5.55556 |
| 7.77778 |
| 10.00000 |

***GRAPHICAL.java (Input GUI)***

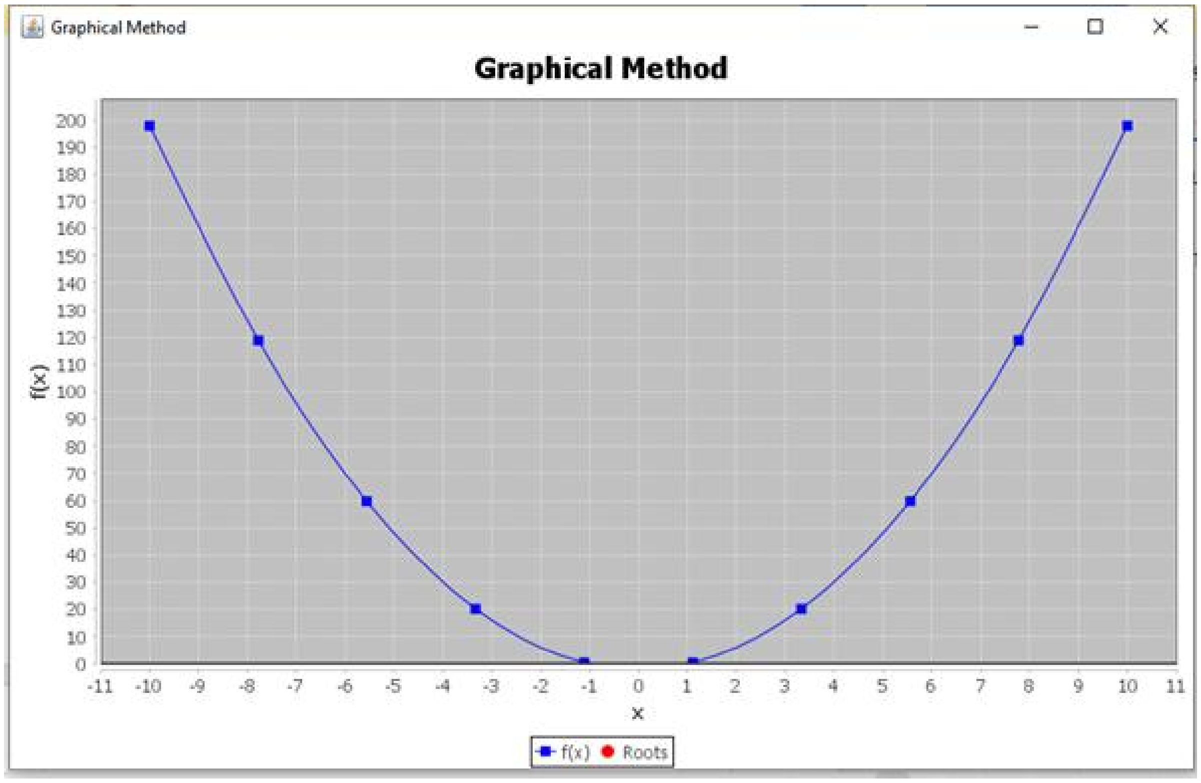
This class provides the user interface for the Graphical Method. It features input fields for entering the function f(x)f(x)f(x) and the maximum number of iterations for plotting. The class includes function buttons for entering trigonometric functions and a numpad for easy data entry. Once the user submits the input, the GUI launches the GraphicalSolver to plot the function and find potential roots.

***GraphicalSolver.java***

This class contains the core logic for solving the graphical method. It takes the function input, evaluates the function at regular intervals over a specified range, and detects when the function crosses the x-axis. The method stores the evaluated points for both table and graphical display. It is responsible for calculating and plotting the function's graph and identifying root approximations.

***GGRAPH.java (Output GUI)***

The output class GGRAPH displays the results of the graphical method. It shows a table with iteration data, including the x-values and function values. It also generates a graph using JFreeChart, which plots the function and marks the approximate roots. This provides a visual representation of the function's behavior and the estimated root locations.



## ***Project Scope***

This project focuses on implementing several fundamental numerical methods for root finding, including the Graphical Method, Incremental Search, Bisection, Newton-Raphson, Secant, and Regula Falsi methods. The primary scope includes:

Developing user-friendly graphical interfaces that allow dynamic input of mathematical functions.

Providing visual and tabular outputs to facilitate understanding of each method's iterative process.

Supporting multiple roots detection where applicable.

Incorporating input validation and error handling for robust performance.

Utilizing established Java libraries (Exp4j, JFreeChart) to support expression parsing and graphical plotting.

Ensuring tolerance controls with high precision (up to five decimal places).

Enabling educational insights through detailed iteration data and graphical visualizations.

The project is designed as a versatile learning tool for computer engineering students and practitioners to explore and apply numerical root-finding methods efficiently.

---

## ***Future Improvements***

While the current implementation provides a solid foundation, several enhancements could further increase the project's functionality, usability, and educational value:

Extended Function Support:
Enhance the parser to support more complex mathematical functions, piecewise definitions, or parametric functions.

Adaptive Interval Selection:
Implement automatic interval refinement or adaptive step sizes to improve root detection accuracy and reduce computation time.

Multi-root Detection and Separation:
Extend support to detect and isolate multiple roots more effectively, with options to zoom into intervals containing roots.

Performance Optimization:
Optimize algorithms for faster convergence and reduced computational overhead, especially for large iteration counts.

Mobile and Web Versions:
Develop versions for mobile platforms or as web applications to increase accessibility.

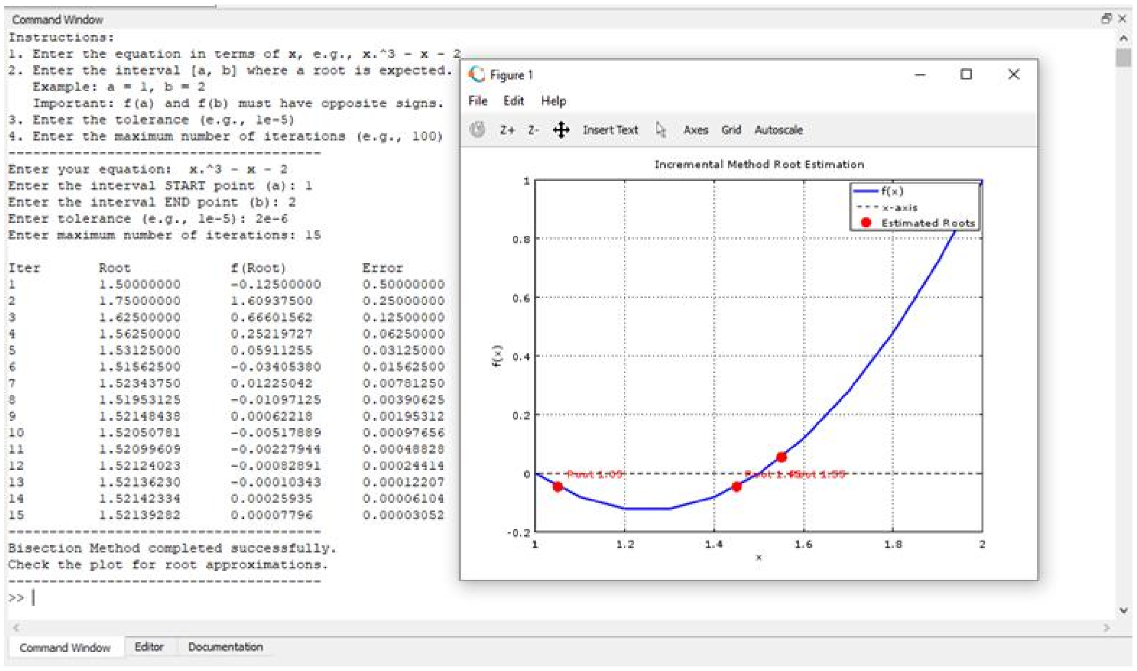
Interactive Graph Features:
Add zooming, panning, and point inspection capabilities to the graphs for enhanced user interaction.

Error and Exception Handling Enhancements:
Provide more detailed feedback and suggestions for invalid inputs or divergence cases.

Integration with Symbolic Computation:
Integrate with symbolic math engines to assist in derivative calculation or simplify expressions automatically.

User Tutorials and Documentation:
Embed guided tutorials within the application to help beginners understand each method step-by-step.

**MATLAB – MATHWORKS – NUMERICAL METHODS:**



```
Instructions:
1. Enter the equation in terms of x, e.g., x.^3 - x - 2
2. Enter the interval [a, b] where a root is expected.
   Example: a = 1, b = 2
   Important: f(a) and f(b) must have opposite signs.
3. Enter the tolerance (e.g., 1e-5)
4. Enter the maximum number of iterations (e.g., 100)
----------------------------------------
Enter your equation:  x.^3 - x - 2
Enter the interval START point (a): 1
Enter the interval END point (b): 2
Enter tolerance (e.g., 1e-5): 2e-6
Enter maximum number of iterations: 15

Iter      Root          f(Root)        Error
1         1.50000000    -0.12500000    0.50000000
2         1.75000000    1.60937500     0.25000000
3         1.62500000    0.66601562     0.12500000
4         1.56250000    0.25219727     0.06250000
5         1.53125000    0.05911255     0.03125000
6         1.51562500    -0.03405380    0.01562500
7         1.52343750    0.01225042     0.00781250
8         1.51953125    -0.01097125    0.00390625
9         1.52148438    0.00062218     0.00195312
10        1.52050781    -0.00517889    0.00097656
11        1.52099609    -0.00227944    0.00048828
12        1.52124023    -0.00082891    0.00024414
13        1.52136230    -0.00010343    0.00012207
14        1.52142334    0.00025935     0.00006104
15        1.52139282    0.00007796     0.00003052
----------------------------------------
Bisection Method completed successfully.
Check the plot for root approximations.
----------------------------------------
>> |
```

***MATLAB (short for Matrix Laboratory)*** is a high-level programming language and environment developed by MathWorks. It is designed specifically for:

- ✓ Numerical computing
- ✓ Data visualization
- ✓ Algorithm development
- ✓ Matrix and vector operations
- ✓ GUI-based applications
- ✓ Symbolic computation (via the Symbolic Math Toolbox)

MATLAB is especially popular in engineering, applied mathematics, physics, and data science due to its powerful built-in functions and ease of working with mathematical models and simulations.

1. Graphical User Interfaces (GUI)

Using uifigure, uipanel, uieditfield, uidropdown, and uiaxes, we can build interactive, windowed apps without needing external UI frameworks.

2. Symbolic Computation

The Symbolic Math Toolbox allows us to:

Parse user input like "x^3 - x - 2"

Convert it into symbolic expressions

Automatically generate anonymous functions using matlabFunction

3. Numerical Methods

MATLAB excels at root-finding because it supports:

High-precision floating-point arithmetic

Fast array manipulation

Real-time plotting and visualization
This allows us to implement methods like Bisection, Secant, and Newton-Raphson with clear and efficient code.

4. Dynamic Visualization

The uiaxes component enables:

Real-time plotting of user-defined functions

Color-coded and interactive visualizations

Overlaying roots, secants, tangents, and iterations directly on the graph

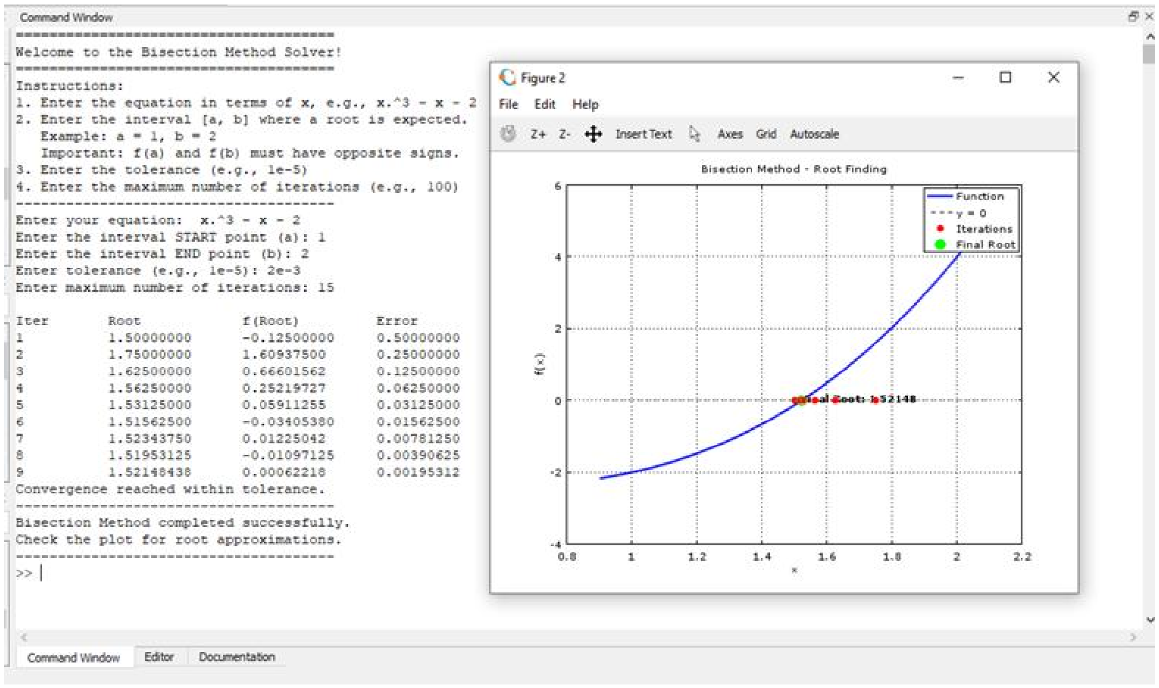
5. User Interaction

We use:

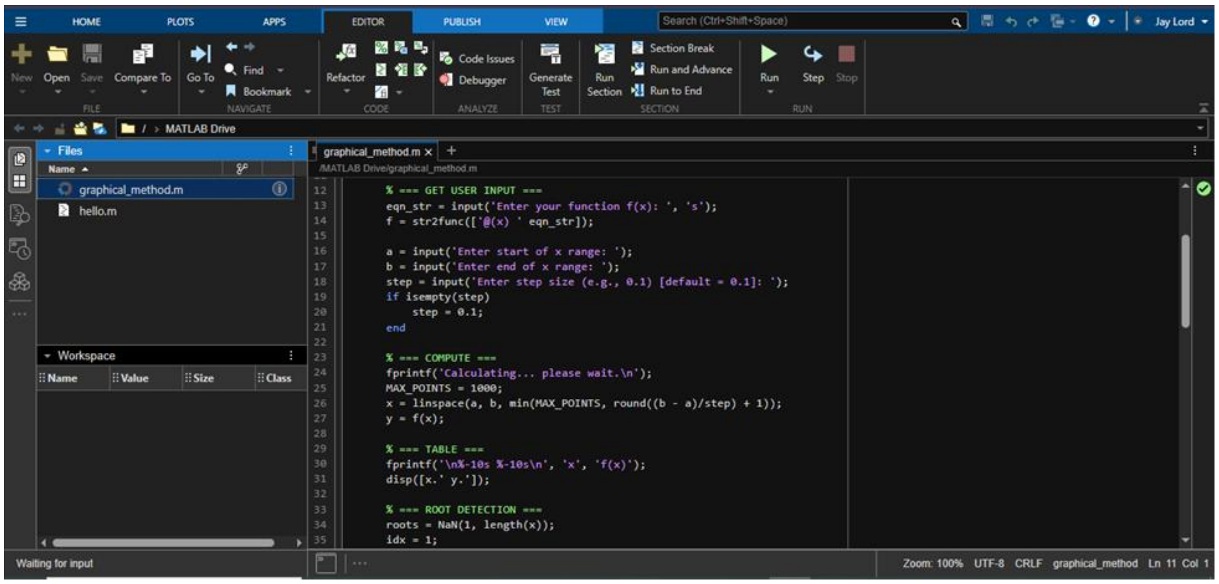
Callback functions like ButtonPushedFcn and ValueChangedFcn

Dynamic alerts with uialert

Animated button feedback using WindowButtonMotionFcn and timer



```
========================================
Welcome to the Bisection Method Solver!
========================================
Instructions:
1. Enter the equation in terms of x, e.g., x.^3 - x - 2
2. Enter the interval [a, b] where a root is expected.
   Example: a = 1, b = 2
   Important: f(a) and f(b) must have opposite signs.
3. Enter the tolerance (e.g., 1e-5)
4. Enter the maximum number of iterations (e.g., 100)
----------------------------------------
Enter your equation:  x.^3 - x - 2
Enter the interval START point (a): 1
Enter the interval END point (b): 2
Enter tolerance (e.g., 1e-5): 2e-3
Enter maximum number of iterations: 15

Iter       Root          f(Root)        Error
1          1.50000000    -0.12500000    0.50000000
2          1.75000000    1.60937500     0.25000000
3          1.62500000    0.66601562     0.12500000
4          1.56250000    0.25219727     0.06250000
5          1.53125000    0.05911255     0.03125000
6          1.51562500    -0.03405380    0.01562500
7          1.52343750    0.01225042     0.00781250
8          1.51953125    -0.01097125    0.00390625
9          1.52148438    0.00062218     0.00195312
Convergence reached within tolerance.
----------------------------------------
Bisection Method completed successfully.
Check the plot for root approximations.
----------------------------------------
>> |
```



```
    % === GET USER INPUT ===
    eqn_str = input('Enter your function f(x): ', 's');
    f = str2func(['@(x) ' eqn_str]);

    a = input('Enter start of x range: ');
    b = input('Enter end of x range: ');
    step = input('Enter step size (e.g., 0.1) [default = 0.1]: ');
    if isempty(step)
        step = 0.1;
    end

    % === COMPUTE ===
    fprintf('Calculating... please wait.\n');
    MAX_POINTS = 1000;
    x = linspace(a, b, min(MAX_POINTS, round((b - a)/step) + 1));
    y = f(x);

    % === TABLE ===
    fprintf('\n%-10s %-10s\n', 'x', 'f(x)');
    disp([x.' y.']);

    % === ROOT DETECTION ===
    roots = NaN(1, length(x));
    idx = 1;
```

The core of this MATLAB numerical methods Root Finder is built around a **Graphical User Interface (GUI)** created using MATLAB’s **App Designer functions** such as uifigure, uipanel, uidropdown, uieditfield, and uiaxes. When the app launches, it opens a main window (uifigure) titled *Root Finder*, divided into a sidebar and a main display area. The sidebar includes a dropdown for selecting the numerical method, as well as buttons for showing instructions and resetting the interface. The user enters a mathematical function (like x^3 - x - 2) into a text box at the top of the main area. Upon clicking the **Run** button, the app processes the function and applies the selected root-finding method.

The input function is first cleaned and parsed by the preprocess_equation function. This handles syntax conversion—for example, transforming ^ to .^ for element-wise operations, inserting * where multiplication is implied, and supporting LaTeX-style syntax like \sqrt{} or \frac{}{}. Once the equation is preprocessed, MATLAB's str2sym and matlabFunction are used to convert the text into a **symbolic expression**, then into a **numeric function*_