UNIVERSITY OF BOHOL

Tagbilaran City, Bohol, Philippines

COLLEGE OF ENGINEERING, TECHNOLOGIES,

ARCHITECTURE AND FINE ARTS

COMPUTER ENGINEERING NUMERICAL METHODS

CPEP 221

NUMERICAL PLOTTING USING PYTHON

Submitted by:

ORIG, JEFF AXEL P.

Introduction

This project is a Python-based application designed to support the study and application of numerical methods through interactive equation plotting and root-finding. It provides users with a visual and analytical tool to explore how different numerical algorithms behave when solving nonlinear equations.

The application allows users to input any valid mathematical function in the format f(x) = ..., visualize the function on a graph, and apply a selection of numerical methods to approximate the roots. The application generates a table showing the detailed iteration process for each method, helping users understand how convergence is achieved over successive steps.

The following root-finding algorithms are implemented in this project:

- Incremental Search
- Bisection Method
- False Position Method (Regula Falsi)
- Newton-Raphson Method
- Secant Method

Each method provides both visual and tabular output, making the comparison between techniques straightforward and educational. Users can observe the graphical behavior of the function, track the intermediate values for each iteration, and assess the efficiency and accuracy of each method based on convergence speed and root approximation.

This application is developed entirely in Python, contained in a single script file. It leverages standard libraries such as Tkinter for the graphical user interface, Matplotlib for plotting, and SymPy and NumPy for symbolic and numerical computations. The result is a compact yet powerful tool for learning, teaching, and experimenting with root-finding algorithms.

Scope and Limitation

Scope

This project focuses on providing an interactive and educational environment for visualizing mathematical functions and solving nonlinear equations using five classical numerical methods. The key features covered within the scope of this application include:

- **Equation Input*:*** Accepts single-variable equations in the form f(x) = ..., supporting a wide range of mathematical functions (e.g., polynomials, exponentials, trigonometric functions).
- **Graph Plotting:** Plots the user-defined function on a 2D graph to help users visually interpret the function's behavior and approximate location of its roots.
- *Root-Finding Methods:* Implements the following numerical techniques:
  - Incremental Search
  - Bisection Method
  - False Position Method
  - Newton-Raphson Method
  - Secant Method
- **Iteration Table Output:** Displays a detailed step-by-step table showing the progress of each method toward finding a root, including relevant values such as iteration number, approximations, errors, and function values.
- **Graphical Feedback:** Highlights iterations or root intervals visually on the plotted graph for better understanding.
- **User-Friendly GUI:** Built with Tkinter, the interface allows for simple input, method selection, and visualization without the need for command-line interaction.

Limitations

While the project is functional and educational, it is subject to the following limitations:

- **Single Equation Handling:** Only supports one equation at a time; does not handle systems of equations or multivariable functions.
- **Initial Guess Sensitivity:** Methods like Newton-Raphson and Secant require appropriate initial values for convergence; poor input may result in divergence or invalid results.
- **No Symbolic Derivative Check:** For Newton-Raphson, the derivative must be computable; if the derivative is undefined at any step, the method may fail.
- **No Step-by-Step Animation:** The plotting is static per method run and does not animate the iterative process.
- **Basic Error Handling:** While the application handles many typical errors, complex input parsing or malformed expressions might cause unexpected behavior or crashes.
- **Performance:** Designed for educational use and small-scale functions; it may not perform well with extremely complex or computationally heavy equations.

PROBLEM REQUIREMENTS

Purpose

The purpose of creating this program is to explore and deepen knowledge in Python programming by applying it to a meaningful and educational project in numerical analysis. This project serves as a hands-on opportunity to integrate core Python concepts—such as functions, data structures, loops, and object-oriented programming—with external libraries like Tkinter, SymPy, NumPy, Pandas, and Matplotlib.

By developing a graphical user interface that implements multiple root-finding methods, the project bridges both programming and mathematical problem-solving. It allows the developer to gain practical experience in designing user-friendly applications, handling mathematical expressions, performing numerical computations, and generating dynamic visual outputs.

Ultimately, this project is a personal and academic exploration aimed at enhancing proficiency in Python, understanding how algorithms work step-by-step, and creating a functional tool that reflects both learning progress and programming capability.

Overall Description

This project is a standalone Python application designed to help users visualize and solve mathematical equations using five classical numerical root-finding methods: Incremental Search, Bisection, False Position (Regula Falsi), Newton-Raphson, and Secant Method. It features an intuitive graphical user interface (GUI) built with Tkinter, allowing users to input custom functions, select a method, define parameters, and view both the function plot and a detailed iteration table.

The application leverages Matplotlib for dynamic graph rendering, NumPy and SymPy for numerical and symbolic computations, and Pandas for organizing and displaying iteration data in tabular form. It also incorporates cross-platform UI enhancements for a consistent user experience on different operating systems.

All features are integrated into a single Python script, making the tool lightweight, portable, and ideal for students, educators, or self-learners exploring numerical methods and Python programming. This project demonstrates the practical application of Python in mathematical problem-solving and interactive software development.

Analysis

Input requirements

Common Inputs for All Methods:

- Function: f(x) in the form f(x) = expression (e.g., f(x) = x^3 - 0.165*x^2 + 3.993*10^-4)
- Interval:
  - x_min: Starting point of the search range
  - x_max: Ending point of the search range
- Tolerance: Acceptable error for stopping (e.g., 0.0001)
- Maximum Iterations: Upper limit for number of iterations (e.g., 100)

Method-Specific Inputs:

- Graphical Method:
  - Requires only the function and interval
  - No tolerance or iteration inputs needed
- Incremental Method:
  - Requires a step size to increment x values during sign change detection
- Bisection Method:
  - No extra input beyond common ones
  - Automatically detects intervals with sign changes
- Regula Falsi Method:
  - Same inputs as Bisection Method
  - Uses linear interpolation between endpoints
- Newton-Raphson Method:
  - Requires an initial guess close to the expected root
- Secant Method:
  - Requires both an initial guess and a second guess

Output requirements

- Roots Found:
  - List of root values xxx where f(x)≈0f(x) \approx 0f(x)≈0
  - Each root includes f(x)f(x)f(x) evaluated at that point
- Graph Output:
  - A plot of f(x)f(x)f(x) over the defined interval
  - Roots are marked on the graph
- Results Panel:
  - Text summary of the method used
  - Number of roots found
  - Root values and function evaluations
- Iterations Table (for iterative methods only):
  - Step-by-step breakdown of each iteration
  - Shows values such as:
    - Current approximations
    - Function evaluations
    - Approximate errors

- Remarks (e.g., "Converged", "Max iterations reached")

Necessary Formula and their Description

1. **Graphical Method**
   Formula:
   None explicitly, but relies on detecting sign changes and linear interpolation:
   $$x_r = x_i - \frac{f(x_i)(x_{i+1} - x_i)}{f(x_{i+1}) - f(x_i)}$$
   Description:
   Detects roots by plotting the function and finding where it crosses the x-axis. Uses interpolation to estimate root locations visually.

2. **Incremental Search**
   - Stepwise Evaluation:
   $$x_{i+1} = x_i + \Delta x$$
   - Sign Check:
   $$f(x_i) \cdot f(x_{i+1}) < 0$$
   Description:
   Scans through the interval using a fixed step size to find where a sign change occurs (indicating a root). Refines the estimate by narrowing the interval.

3. **Bisection Method**
   - Formula:
   $$c = \frac{a+b}{2}$$
   $$\epsilon_a = \left|\frac{c_{new} - c_{old}}{c_{new}}\right| \times 100\%$$
   Description:
   Repeatedly halves the interval and selects the subinterval where the sign change occurs. Guaranteed convergence if initial interval brackets a root.

4. **Regula Falsi (False Position)**
   - Formula
   $$c = a\frac{f(a)(b-a)}{f(b)-f(a)}$$
   $$\epsilon_a = \left|\frac{c_{new} - c_{old}}{c_{new}}\right| \times 100\%$$
   Description:
   Uses a secant line between $(a, f(a))$ and $(b, f(b))$ to estimate the root. More efficient than bisection in many cases.

5. **Newton-Raphson**
   - Formula:
   $$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$
   $$\varepsilon_a = \left|\frac{x_{n+1} - x_n}{x_{n+1}}\right| \times 100\%$$

Description:

Rapidly converges using the function and its derivative. Requires good initial guess and derivative evaluation.

6. **Secant Method**
   - Formula:

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

$$\varepsilon_a = \left|\frac{x_{n+1} - x_n}{x_{n+1}}\right| \times 100\%$$

Description:

Similar to Newton-Raphson, but estimates the derivative numerically using two recent points. Does not require an explicit derivative.

Design

Files and Their Descriptions

The project is contained in a single Python file named Project.py, which serves as the main script for the entire application. It includes the complete implementation of the Tkinter-based graphical user interface (GUI), user input handling for mathematical functions and method parameters, dark mode support, function plotting with zoom and pan capabilities using matplotlib, and dynamic rendering of results in a scrollable table. The file also integrates all root-finding method logic—including Graphical, Incremental Search, Bisection, Regula Falsi, Newton-Raphson, and Secant—along with mathematical parsing using sympy, numerical computations with numpy, and real-time graph updates based on user interaction.

User Interface Design

The user interface (UI) of the application is designed using Python's Tkinter library, enhanced with ttk themed widgets for a modern and consistent appearance. The main window is divided into intuitive sections: a function input field (f(x) =) at the top for users to enter mathematical expressions, followed by a method selection dropdown to choose among Graphical, Incremental Search, Bisection, Regula Falsi, Newton-Raphson, and Secant methods. Below that, dynamic input fields appear based on the selected method (e.g., interval bounds, initial guesses, tolerance, and max iterations). A plot area on the right side of the window uses matplotlib to graph the function and visually display roots, with interactive zooming and panning. The bottom section contains control buttons (e.g., "Plot Function", "Find Root", "Clear"), and a scrollable results table that displays iteration data and the computed root. A toggle for dark mode is also provided to switch themes. The layout ensures clarity, accessibility, and responsiveness for a smooth user experience.

Features of the Project

The project offers a comprehensive set of features for mathematical root-finding and function visualization. It supports multiple numerical methods including Graphical Method, Incremental Search, Bisection, Regula Falsi (False Position), Newton-Raphson, and Secant, each with dynamic input fields tailored to their specific requirements. Users can enter custom functions using standard Python math syntax and visualize them interactively through a matplotlib-based graph with support for zooming and panning. The application displays roots clearly marked on the graph and provides a scrollable results table showing step-by-step iteration data. A precision control option allows users to define the decimal accuracy of results. The UI includes dark mode support for comfortable viewing, along with reset and clear functionalities to manage inputs and outputs efficiently. Overall, the project combines educational value with usability, making it ideal for learning and exploring root-finding techniques.

Implementation

URL of the Saved Source Code

https://github.com/046289/NumericalMethodPython

Software Package Files and their Descriptive Information

- **numpy (import numpy as np)**
  - Used for fast and efficient numerical computations, especially when handling arrays and mathematical operations in root-finding methods.
- **matplotlib.pyplot (import matplotlib.pyplot as plt)**
  - Provides tools to plot functions and results. It enables graph rendering and customization in the application.
- **sympy (import sympy as sp)**
  - Enables symbolic mathematics such as parsing and differentiating user-defined functions (e.g., for Newton-Raphson method).
- **pandas (import pandas as pd)**
  - Used to organize and format tabular result data efficiently before displaying it in the UI.
- **re**
  - Supports regular expressions used in validating and parsing user input equations.
- **matplotlib.backends.backend_tkagg.FigureCanvasTkAgg**
  - Embeds matplotlib plots inside the Tkinter GUI, enabling interactive plotting within the app.
- **matplotlib.figure.Figure**
  - Allows creation of custom plot figures used for graphing user-input functions.
- **tkinter and tkinter.ttk, messagebox, scrolledtext**
  - Build the main GUI interface, themed widgets, dialogs (error/info boxes), and scrollable text areas.
- **matplotlib.patheffects**
  - Adds text effects such as outlining root labels for better visibility on plots.
- **matplotlib.cm.get_cmap**
  - Used for setting color maps (e.g., root highlight colors) for consistent styling in visual plots.
- **os**
  - Provides access to operating system functionalities such as path handling and checking platform specifics.
- **platform**
  - Used to identify the operating system, potentially for adjusting GUI behavior or file paths accordingly.

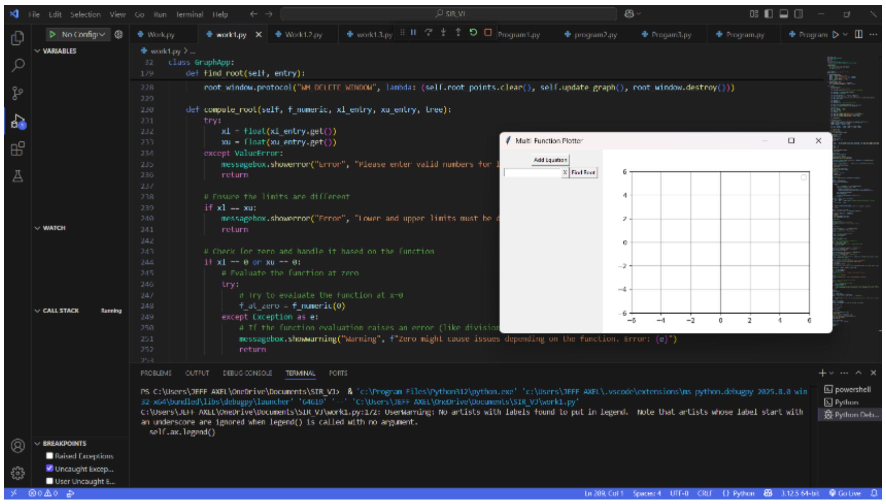
Function Declarations and their Descriptive Purposes

- **parse_function_input(input_str)**
  - Parses the user-input equation string in the format f(x) = ... and returns a symbolic function f(x) and a callable Python function. Ensures valid syntax before processing.
- **incremental_search(fx, x0, delta, max_iter)**

   - Implements the Incremental Search method for locating a root bracket. Iteratively checks intervals to detect a sign change.
- **bisection_method(fx, a, b, tol, max_iter)**
   - Executes the Bisection method by repeatedly halving the interval [a, b] and narrowing down where the root lies until tolerance is met.
- **regula_falsi_method(fx, a, b, tol, max_iter)**
   - Applies the False Position (Regula Falsi) method, using a secant-like interpolation between points to approximate the root.
- **newton_raphson_method(fx, dfx, x0, tol, max_iter)**
   - Performs the Newton-Raphson method, utilizing function values and their derivatives to rapidly converge to a root.
- **secant_method(fx, x0, x1, tol, max_iter)**
   - Implements the Secant method, approximating derivatives using previous two points to update guesses iteratively.
- **plot_function_on_canvas(f, x_range, roots=None)**
   - Draws the user-defined function on the matplotlib canvas embedded in the GUI. Optionally plots roots as red points and labels them.
- **display_table(data)**
   - Converts method output (iterations, values, etc.) into a table using pandas and displays it in a scrollable text box in the GUI.
- **solve_equation()**
   - Main handler function triggered by the "Solve" button. Parses input, selects the method, gathers parameters, performs the method, plots the result, and shows the table.
- **on_pan(event) and on_scroll(event)**
   - Enable interactivity in the graph. Panning adjusts the view, and scrolling zooms in or out using mouse controls.
- **toggle_dark_mode()**
   - Switches between light and dark mode themes for the GUI and plot styling.
- **clear_all()**
   - Clears user inputs, plot area, and result tables, resetting the GUI for a new session.

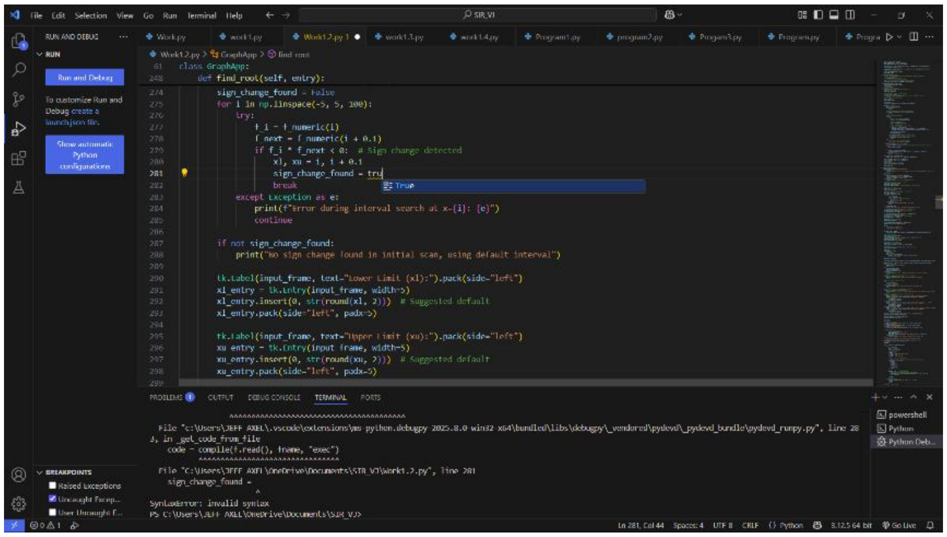
Testing and debugging

Initial output.



```
class GraphApp:
    def find_root(self, entry):
        root_window.protocol("WM_DELETE_WINDOW", lambda: (self.root_points.clear(), self.update_graph(), root_window.destroy()))

    def compute_root(self, f_numeric, xl_entry, xu_entry, tree):
        try:
            xl = float(xl_entry.get())
            xu = float(xu_entry.get())
        except ValueError:
            messagebox.showerror("Error", "Please enter valid numbers for l
            return

        # Ensure the limits are different
        if xl == xu:
            messagebox.showerror("Error", "Lower and upper limits must be d
            return

        # Check for zero and handle it based on the function
        if xl == 0 or xu == 0:
            # Evaluate the function at zero
            try:
                # Try to evaluate the function at x=0
                f_at_zero = f_numeric(0)
            except Exception as e:
                # If the function evaluation raises an error (like division
                messagebox.showwarning("Warning", f"Zero might cause issues depending on the function. Error: {e}")
                return
```

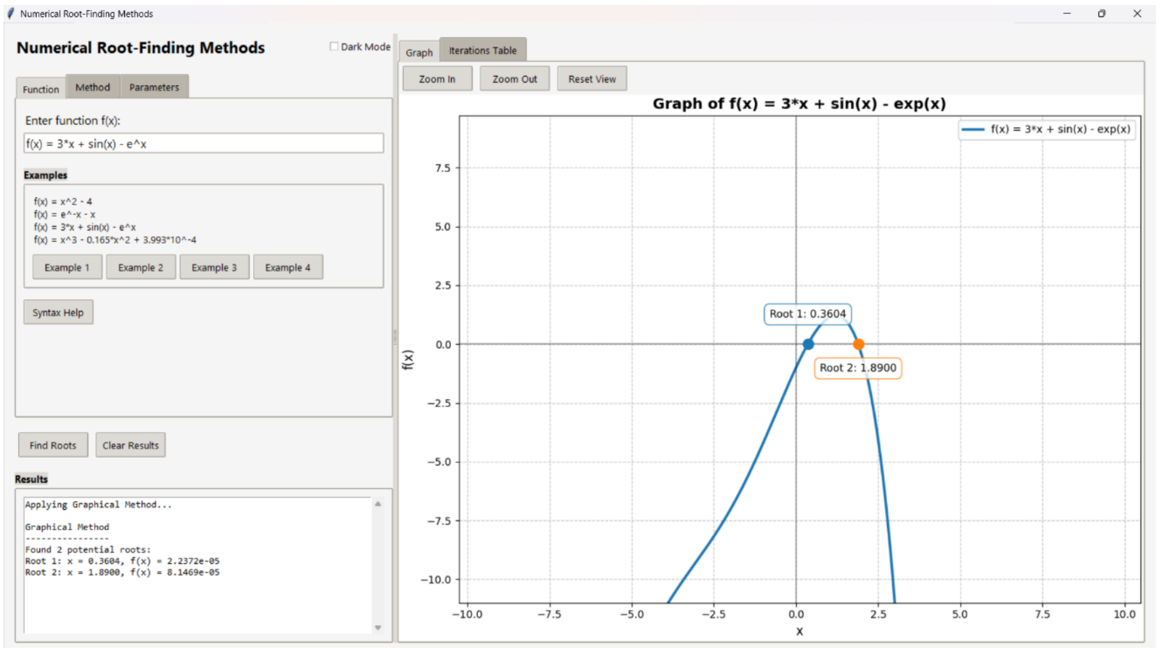
```
PS C:\Users\JEFF AXEL\OneDrive\Documents\SIR_VJ>  & 'c:\Program Files\Python312\python.exe' 'c:\Users\JEFF AXEL\.vscode\extensions\ms-python.debugpy-2025.8.0-win32-x64\bundled\libs\debugpy\launcher' '64619' '--' 'C:\Users\JEFF AXEL\OneDrive\Documents\SIR_VJ\work1.py'
C:\Users\JEFF AXEL\OneDrive\Documents\SIR_VJ\work1.py:172: UserWarning: No artists with labels found to put in legend.  Note that artists whose label start with
an underscore are ignored when legend() is called with no argument.
  self.ax.legend()
```



```python
class GraphApp:
    def find_root(self, entry):
        sign_change_found = False
        for i in np.linspace(-5, 5, 100):
            try:
                f_i = f_numeric(i)
                f_next = f_numeric(i + 0.1)
                if f_i * f_next < 0:  # Sign change detected
                    xl, xu = i, i + 0.1
                    sign_change_found = tru
                    break
            except Exception as e:
                print(f"Error during interval search at x={i}: {e}")
                continue

        if not sign_change_found:
            print("No sign change found in initial scan, using default interval")

        tk.Label(input_frame, text="Lower Limit (xl):").pack(side="left")
        xl_entry = tk.Entry(input_frame, width=5)
        xl_entry.insert(0, str(round(xl, 2)))  # Suggested default
        xl_entry.pack(side="left", padx=5)

        tk.Label(input_frame, text="Upper Limit (xu):").pack(side="left")
        xu_entry = tk.Entry(input_frame, width=5)
        xu_entry.insert(0, str(round(xu, 2)))  # Suggested default
        xu_entry.pack(side="left", padx=5)
```

```
                ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "c:\Users\JEFF AXEL\.vscode\extensions\ms-python.debugpy-2025.8.0-win32-x64\bundled\libs\debugpy\_vendored\pydevd\_pydevd_bundle\pydevd_runpy.py", line 28
3, in _get_code_from_file
    code = compile(f.read(), fname, "exec")
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\JEFF AXEL\OneDrive\Documents\SIR_VJ\Work1.2.py", line 281
    sign_change_found =
                       ^
SyntaxError: invalid syntax
PS C:\Users\JEFF AXEL\OneDrive\Documents\SIR_VJ>
```

Final output.



```
Applying Graphical Method...

Graphical Method
----------------
Found 2 potential roots:
Root 1: x = 0.3604, f(x) = 2.2372e-05
Root 2: x = 1.8900, f(x) = 8.1469e-05
```

UNIVERSITY OF BOHOL

CITY OF TAGBILARAN

CURRICULUM VITAE

I. PERSONAL INFORMATIONS

Name	**:**	Jeff Axel P. Orig
Address	**:**	Pamaong Stanai Rd., brg. Cogon, Tagbilaran City
Birthdate	**:**	June 26, 2005
Nationality	**:**	Filipino
Religion	**:**	Roman Catholic
Civil Status	**:**	Single
Parents	**:**	Eleazar Wilson Orig Juliet Orig

II. EDUCATIONAL BACKGROUND

Tertiary	**:**	University of Bohol, College of Engineering, Technology, Architecture and Fine Arts Dr. Cecilio Putong Street, Cogon Tagbilaran City, Bohol 2023-Present
Secondary	**:**	Cor Jesu Institute of Mabini Inc. Mabini, Davao de Oro S.Y. 2017-2023
Elementary	**:**	Maabini Central Elementary School Mabini, Davao de Oro S.Y. 2011-2017

Future Development

Recommendations

- *Add Additional Numerical Methods:* Integrate other root-finding algorithms such as Brent's Method or Muller's Method for more robustness and efficiency.
- *Error Handling Enhancements:* Improve input validation and exception handling for malformed equations, division by zero, or invalid method ranges.
- *Support for Complex Roots:* Extend functionality to detect and plot complex roots with separate visualization or textual explanation.
- *Interactive Graph Features:* Enable clickable root points to show coordinates, dynamic zoom/pan reset buttons, and real-time updates on parameter changes.
- *Method Comparison Mode:* Allow side-by-side comparisons of different methods on the same function to evaluate speed and convergence.
- *Export and Save Options:* Add functionality to export results and graphs to PDF or CSV formats for academic or professional use.
- *Multi-Equation Handling:* Support solving systems of equations or plotting multiple functions in a single graph window.
- *Cross-Platform Installers:* Package the project with tools like PyInstaller or cx_Freeze for easy distribution as a standalone application.
- *Mobile and Web Versions:* Port the application using Kivy for mobile or frameworks like Flask/Django with Plotly for interactive web-based tools.
- *Unit Testing and CI/CD Integration:* Add unit tests for core methods and automate testing workflows to ensure long-term code reliability and maintainability.

Project Cost

Expenses	Costs
Printing	110
Bookbinding	150
Total	260

Glossary

Root-Finding:

A numerical technique used to find values of x where a function f(x)=0f(x) = 0f(x)=0.

Incremental Search:

A simple method to detect an interval where a sign change occurs, indicating a possible root.

Bisection Method:

A bracketing method that repeatedly halves an interval where a sign change is detected to narrow down the root.

False Position (Regula Falsi):

A bracketing method like Bisection but uses linear interpolation between endpoints to estimate the root.

Newton-Raphson Method:

An open root-finding method using tangents (derivatives) to quickly approximate roots.

Secant Method:

An open method similar to Newton-Raphson, but it does not require the derivative and instead uses a secant line between two points.

Function Parser:

Converts a user-defined function in string format into a symbolic or numeric expression that can be evaluated.

Tkinter:

Python's standard GUI library used to build the graphical user interface of the application.

SymPy:

A Python library for symbolic mathematics, used for parsing and differentiating functions.

Matplotlib:

A plotting library in Python used to visualize mathematical functions and root approximations.

NumPy:

A Python library used for numerical computations, such as evaluating functions over arrays.

Pandas:

A data analysis library in Python used for organizing and displaying iteration data in tabular format.

ScrolledText:

A Tkinter widget that provides a scrollable text area for displaying method iteration logs or messages.

FigureCanvasTkAgg:

A bridge between Matplotlib and Tkinter that enables embedding plots into the GUI.

Bibliography

Kaw, A. (n.d.). *Chapter 03.03: Bisection Method of Solving a Nonlinear Equation*. Holistic Numerical Methods. Retrieved from https://nm.mathforcollege.com/nmsims/03NLE/03NLE_Bisection_Method/bisection-method_en.html