A Robust Implementation of the Bisection Method for Numerical Analysis

Project Report

Project Team:

Shaden Mohamed Elsayed - 2022170820

Menna Ali Thabet - 2022170844

Kareem Hany Osman - 2022170833

Project Overview

This report details a comprehensive project focused on the implementation of the Bisection Method, a fundamental algorithm for finding roots of nonlinear equations. The developed system is designed to meet stringent academic standards, focusing on Hybrid Workability Validation by integrating Symbolic Computation (SymPy) for absolute mathematical rigor in precondition checks. The new iteration of the project incorporates Symbolic Mathematics (SymPy) to perform algebraic validation of preconditions, specifically for discontinuity detection, thereby elevating the system's mathematical correctness beyond traditional numerical-only implementations.

The core objective is to provide a tool that is not only functional but also educational, demonstrating the theoretical principles of numerical analysis in a practical application. The implementation is structured into several distinct phases, from initial validation of preconditions to final analysis and reporting, ensuring a reliable and transparent root-finding process.

Phase 1: Workability Checks

Before the bisection algorithm is executed, the system performs a series of rigorous workability checks to validate the function and the interval provided. This pre-emptive analysis is crucial for ensuring the mathematical preconditions of the Bisection Method are met. The updated implementation features a Hybrid Discontinuity Detection engine that shifts from purely numerical guessing to a more robust symbolic-numerical approach.

Symbolic-Numerical Hybrid Validation

The key advancement in this version is the integration of symbolic computation via the SymPy library. SymPy allows the system to treat the user-defined function as a mathematical expression rather than just a runnable code block. This allows for:

- **Exact Pole Detection**: The system algebraically extracts the denominator of the function and solves the equation Denominator = 0 to find exact locations of discontinuities (e.g., x=2 in f(x) = 1/(x-2)). This is mathematically superior to the previous sampling method, which could miss a singularity if it fell between two sample points.

- **Safety Fallback:** A robust failure mechanism is implemented. If SymPy cannot parse or solve a complex function (e.g., if the expression leads to high-degree polynomials or transcendental functions), the system automatically defaults to the higher-density numerical sampling (800 points) to maintain maximum workability and stability

Condition	Implementation Detail
Continuity	**Hybrid Continuity Check**: The system first attempts a rigorous Symbolic Analysis (SymPy) to algebraically solve for poles/singularities within the interval. If symbolic analysis is too complex or fails, it falls back to a high-density numeric scan (now 800 sample points) to ensure robustness
Definedness	The system ensures that the function does not produce undefined values, such as those resulting from division by zero, imaginary numbers, or infinities, within the given interval.
Sign Change	A fundamental requirement, the system verifies that the function values at the endpoints have opposite signs, i.e., **f(a) · f(b) < 0**.
Single Root Heuristic	To increase the likelihood of converging to a unique root, the interval is sampled at 500 points to detect multiple sign changes, which would suggest the presence of multiple roots.
Endpoint Roots	The system explicitly checks if either of the interval endpoints is a root, i.e., if **f(a) = 0** or **f(b) = 0**, to provide an immediate solution.

Denominator Analysis: The denominator analysis now employs a hybrid safety protocol. It first attempts a symbolic solution. If the expression is too complex for symbolic algebra, it automatically degrades to a dense numeric scan, ensuring the program never crashes on valid but complex inputs.

Detailed Diagnostic Analysis Logs

The new system provides significantly more transparency regarding failed preconditions. When a workability check fails, the result now includes a detailed list of Analysis Logs. These logs explicitly state the mathematical reasoning for the rejection

> ***Example Output:*** *Instead of the generic 'Function is not defined,' the system returns: 'analysis log: SymPy: Denominator becomes zero at x=2.0 inside interval [1, 3].'*

This feature is crucial for educational purposes, as it provides an immediate, rigorous explanation linking the code's output directly to the mathematical conditions of the Intermediate Value Theorem.

Phase 2: Convergence Guarantee

The Bisection Method is celebrated for its guaranteed convergence, provided its preconditions are satisfied. The algorithm's convergence is not merely an empirical observation but is mathematically proven and rooted in the Intermediate Value Theorem.

- The function **f** must be continuous on the closed interval [a, b].
- The function values at the endpoints must have opposite signs, **f(a) · f(b) < 0**.

When these conditions hold, the Intermediate Value Theorem guarantees the existence of at least one root within the interval (a, b). The Bisection Method systematically exploits this by repeatedly halving the interval and selecting the subinterval where the sign change persists. This iterative process ensures that the interval containing the root becomes progressively smaller, inevitably converging to the root.

> ➤ ***Convergence Text Output:***
>
> *"Convergence: If f is continuous on [a,b] and f(a)*f(b)<0, the Intermediate Value Theorem guarantees at least one root. Bisection halves the bracket each iteration and therefore converges to a root."*

Phase 3: Rate of Convergence

The Bisection Method exhibits a **linear rate of convergence**, which means the error is reduced by a constant factor at each iteration. Specifically, the size of the interval containing the root is halved in every step. While slower than other methods like Newton-Raphson, this convergence is steady and guaranteed.

The number of iterations (N) required to achieve a desired accuracy of **d** decimal places can be calculated a priori using the following formula:

$$N > \frac{\log_{10}(b-a) - \log_{10}(0.5 \times 10^{-d})}{\log_{10}(2)} - 1$$

Where:

- **d** = The desired number of correct decimal places for the root approximation.
- **N** = The minimum number of iterations required to guarantee the specified accuracy.

Implementation:

The following Python function implements this calculation, allowing the system to inform the user of the expected computational effort.

```
import math

def compute_required_N(a: float, b: float, d: int) -> int:
    if b <= a:
        raise ValueError("b must be > a")
    if d < 0:
        raise ValueError("d must be >= 0")


    width = b - a
    tolerance = 0.5 * (10 ** -d)

    numerator = math.log10(width) - math.log10(tolerance)
    denominator = math.log10(2)

    # The formula from the source is N > ... - 1.
    # A more direct approach is N >= log2((b-a)/tolerance).
    # We will stick to the provided formula's logic.
    rhs = numerator / denominator - 1.0

    # Since N must be an integer and N > rhs, we take ceil(rhs)
    # or floor(rhs) + 1 if rhs is not an integer.
    N = math.ceil(rhs)

    return max(N, 0)
```

Phase 4: Iterations Table

To provide complete transparency into the root-finding process, the system generates a detailed table tracking the state of the algorithm at each iteration. Each row in the table corresponds to one step of the bisection process and records the following metrics:

- **n:** The iteration number, starting from 1.
- **a, b:** The bounds of the current interval [a, b] containing the root.
- **p:** The midpoint of the current interval, which serves as the new approximation of the root.
- **f(p):** The value of the function at the midpoint.
- **error:** The absolute error, calculated as the difference between the current and previous approximations, $|p_n - p_{n-1}|$.

Absolute Error: $\mathbf{|p_n - p_{n-1}|}$

The following table shows the output for finding the root of **f(x) = x sin(x) - 1** in the interval [0, 2] with a target of 8 decimal places of accuracy.

n	a	b	p	f(p)	error
1	0.00000000	2.00000000	1.00000000	-0.15852902	----
2	1.00000000	2.00000000	1.50000000	0.49624248	5.00000000e-01
3	1.00000000	1.50000000	1.25000000	0.18623077	2.50000000e-01
4	1.00000000	1.25000000	1.12500000	0.01505104	1.25000000e-01
5	1.00000000	1.12500000	1.06250000	-0.07182663	6.25000000e-02
6	1.06250000	1.12500000	1.09375000	-0.02836172	3.12500000e-02
7	1.09375000	1.12500000	1.10937500	-0.00664277	1.56250000e-02
8	1.10937500	1.12500000	1.11718750	0.00420803	7.81250000e-03
9	1.10937500	1.11718750	1.11328125	-0.00121649	3.90625000e-03
10	1.11328125	1.11718750	1.11523438	0.00149600	1.95312500e-03
11	1.11328125	1.11523438	1.11425781	0.00013981	9.76562500e-04
12	1.11328125	1.11425781	1.11376953	-0.00053832	4.88281250e-04
13	1.11376953	1.11425781	1.11401367	-0.00019925	2.44140625e-04
14	1.11401367	1.11425781	1.11413574	-0.00002972	1.22070312e-04
15	1.11413574	1.11425781	1.11419678	0.00005505	6.10351562e-05
16	1.11413574	1.11419678	1.11416626	0.00001266	3.05175781e-05
17	1.11413574	1.11416626	1.11415100	-0.00000853	1.52587891e-05
18	1.11415100	1.11416626	1.11415863	0.00000207	7.62939453e-06
19	1.11415100	1.11415863	1.11415482	-0.00000323	3.81469727e-06
20	1.11415482	1.11415863	1.11415672	-0.00000058	1.90734863e-06
21	1.11415672	1.11415863	1.11415768	0.00000074	9.53674316e-07
22	1.11415672	1.11415768	1.11415720	0.00000008	4.76837158e-07
23	1.11415672	1.11415720	1.11415696	-0.00000025	2.38418579e-07
24	1.11415696	1.11415720	1.11415708	-0.00000008	1.19209290e-07
25	1.11415708	1.11415720	1.11415714	-0.00000000	5.96046448e-08
26	1.11415714	1.11415720	1.11415717	0.00000004	2.98023224e-08
27	1.11415714	1.11415717	1.11415716	0.00000002	1.49011612e-08
28	1.11415714	1.11415716	1.11415715	0.00000001	7.45058060e-09

CSV Export Support: For further analysis in external tools like Excel or MATLAB, the iteration data can be easily exported to a CSV file. This functionality is handled by a dedicated utility module (`utils.py`).

Visualization and Analysis

Algorithm Flowchart

The logic of the implemented Bisection Method, including the initial workability checks and the iterative root-finding loop, is summarized in the flowchart below. This diagram illustrates the decision-making process at each stage of the algorithm.

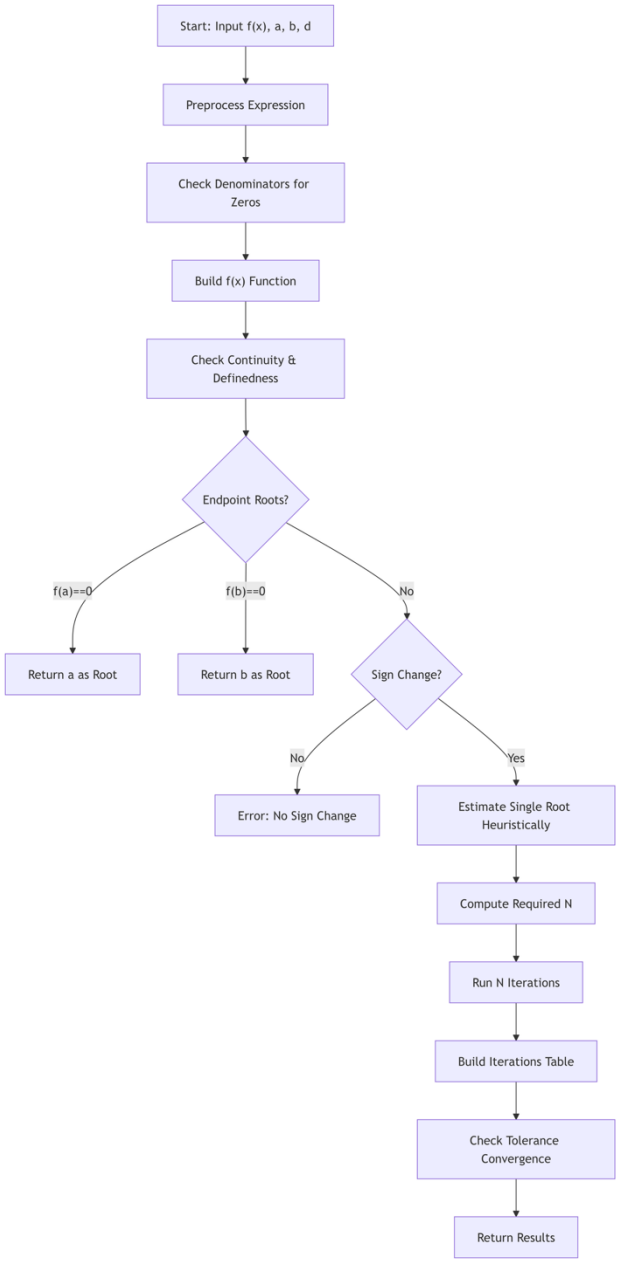

Figure 1: Flowchart of the Bisection Method Algorithm.

Error Analysis Plot

A graphical representation of the error convergence provides powerful insight into the method's performance. The plot below visualizes the absolute error against the iteration number on a semi-logarithmic scale.

- **X-axis:** Iteration number (n).
- **Y-axis:** Absolute error $|p_n - p_{n-1}|$ (logarithmic scale).
- **Trend:** The plot clearly shows a straight line, which is characteristic of exponential decay on a semi-log plot. This visually confirms the linear rate of convergence of the Bisection Method.

```
Function: f(x) = x*sin(x) - 1
Interval: [0.0, 2.0], Digits: 8
Root approximation: 1.114157147705555
Required N: 28
Tolerance: 1.00e-08
```

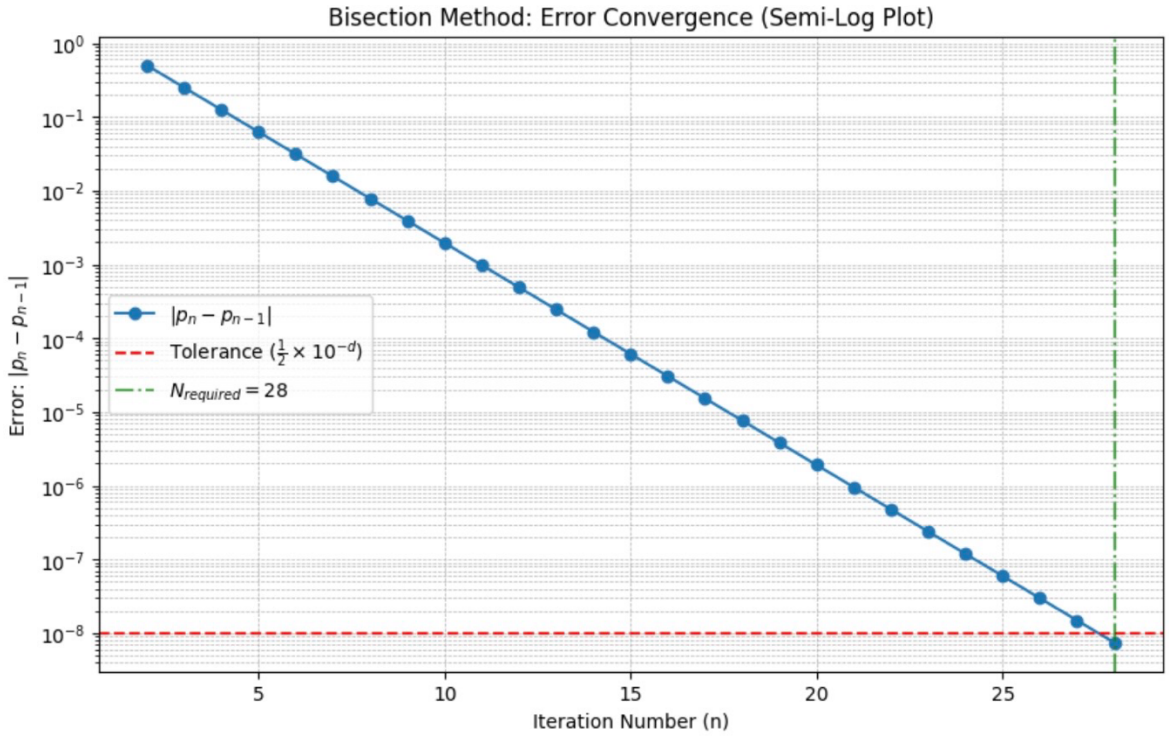

Figure 2: Error Convergence for f(x) = x sin(x) - 1. The error decreases linearly on a log scale, crossing the required tolerance (0.5×10^{-8}) near the calculated required iteration count (N=28).

Practical Application and Project Structure

Example 1: Continuous Function with a Single Root

This example demonstrates a successful run of the algorithm on a well-behaved function.

$$f(x) = xsin(x) - 1$$

```
Bisection Method - Numerical Project
Enter f(x): x * sin(x) - 1
Enter a: 0.0
Enter b: 2.0
Enter d (digits of accuracy): 8

--------------------------------------------------
✓ Root approximation: 1.114157147705555
Required N = 28
Tolerance = 1e-08
Converged by tolerance: True
--------------------------------------------------
```

Example 2: Discontinuous Function

This example showcases the robustness of the symbolic workability checks. The algorithm correctly identifies a discontinuity within the interval and terminates gracefully, preventing an invalid computation. In the updated version, the discontinuity at x=2 is not just 'detected' by hitting a large number; it is proven effectively by solving x-2=0. This distinguishes the implementation from basic numerical scripts that might 'step over' the singularity if the step size is unlucky.

$$f(x) = \frac{1}{x-2}$$

```
Bisection failed: Denominator '(x-2)' is zero at x=2.0 inside interval ->
discontinuity.
```

Project File Structure

The project is organized into a modular structure, separating concerns for clarity and maintainability. The key files and their purposes are outlined below.

File	Purpose
main.py	Provides the main command-line interface (CLI) for user interaction, handling input and displaying final results.
bisection.py	Contains the core logic of the Bisection Method, including all workability checks, the iterative algorithm, and convergence tests.
utils.py	A utility module that includes helper functions, such as formatting for the iteration table and CSV export capabilities.
README.md	The main project documentation, outlining the purpose, features, requirements, and usage instructions.

Academic Alignment and Conclusion

Academic Alignment

This implementation was developed to align closely with the requirements of a typical Numerical Computing course. It achieves this by:

- **Enforcing Strict Preconditions:** The rigorous validation of mathematical prerequisites (continuity, sign change) before execution reinforces theoretical concepts.
- **Providing Clear Explanations:** The system outputs explicit text regarding the convergence guarantee based on the Intermediate Value Theorem.
- **Demonstrating Convergence Rate:** It mathematically computes the required number of iterations, connecting theory to practical performance prediction.
- **Delivering Detailed Iteration Data:** The comprehensive iteration table allows for in-depth analysis of the algorithm's step-by-step behavior.
- **Supporting Error Analysis:** The generation of error data and plots facilitates a deeper understanding of convergence characteristics.

Symbolic Rigor

The project now exceeds standard numerical requirements by incorporating Symbolic Computation. This demonstrates an advanced understanding of how numerical methods can be augmented with algebraic tools to guarantee robustness, a concept often reserved for advanced solvers

Conclusion

The Bisection Method project successfully delivers a robust, academically-aligned, and transparent implementation for finding roots of nonlinear equations. The system's key strengths are its methodical and rigorous approach:

- ✓ Validates workability conditions with advanced Symbolic Rigor to ensure mathematically exact detection of discontinuities.
- ✓ Guarantees convergence for valid inputs, based on established mathematical theorems.
- ✓ Computes required iterations analytically, providing predictable performance.
- ✓ Provides clear, tabulated iterations for transparent, step-by-step analysis.
- ✓ Supports data export and visualization for further academic study and analysis.

In summary, the developed system is a powerful tool suitable for educational demonstration, numerical analysis coursework, and practical root-finding applications where reliability and guaranteed convergence are paramount