Transcendental Equations Solver

NAME	STUDENT ID
Emaz Ali Khan	65566
Abdul Rafay Zahid	65540
Sumaika Asif	65651

DATE OF SUBMISSION: ________________

Transcendental Equations Solver

SUBMITTED BY

Emaz Ali Khan (65566)

Abdul Rafay Zahid (65540)

Sumaika Asif (65651)

SUPERVISED BY

DR. MUHAMMAD ARIF HUSSAIN

REPORT SUBMITTED TO THE FACULTY OF COMPUTING, KARACHI INSTITUTE OF ECONOMICS AND TECHNOLOGY, IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF BACHELOR OF SCIENCE IN COMPUTER SCIENCE

FALL 2024

Table of Content

1. Summary

This report presents an in-depth analysis of the "Transcendental Equations Solver," a Python-based application designed to numerically solve transcendental Equationss through the application of five sophisticated methods: Bisection, Regula Falsi, Newton's Raphson's, Muller, and Secant. The core objective of the report is to elucidate the methodology employed in the development of the solver, evaluate the effectiveness of the various solution techniques, and offer insights into their comparative performance.

The application is constructed with an intuitive Graphical User Interface (GUI) that enhances usability, enabling users to input complex transcendental functions incorporating trigonometric, logarithmic, and root operations, such as Sin, Cos, Tan, Ln, and Square Root. Additionally, the GUI is equipped with essential functional controls, including "Solve," "Clear," and computational features, to facilitate user interaction and streamline the Equations-solving process.

The methods integrated into the solver—Bisection, Regula Falsi, Newton's Raphson's, Muller, and Secant—are employed to compute the roots of given transcendental Equationss, track iterative steps, and visualize the solutions via graphical representations. The report meticulously evaluates the precision, efficiency, and convergence behavior of these algorithms, drawing distinctions between their respective strengths and limitations in various scenarios.

The findings confirm that each of the algorithms is capable of determining the roots of transcendental Equationss, albeit with varying degrees of efficiency. Methods such as Newton's Raphson's and Muller demonstrate superior convergence rates for well-behaved functions, while the Bisection and Regula Falsi methods are found to be more robust in the face of functions with limited differentiability or irregularities. The graphical outputs and iteration data serve as integral tools for verifying the accuracy and convergence behavior of the roots.

In conclusion, the "Transcendental Equations Solver" offers a powerful, versatile tool for solving transcendental Equationss with high computational precision. The application is particularly beneficial in academic and professional contexts where a robust, multifaceted approach to numerical analysis is required. However, further refinement of the underlying algorithms could enhance computational efficiency and

broaden the scope of supported functions, thereby improving the overall user experience.

The report assumes that the input Equationss are well-posed and that the methods are provided with reasonable initial approximations for convergence. The solver's performance may degrade in the absence of these conditions, particularly in cases where the Equationss exhibit non-differentiable behavior or lack real roots. Further, it is presumed that the user has a foundational understanding of numerical methods to utilize the application effectively.

2. Introduction

Transcendental Equationss, which incorporate complex functions such as trigonometric, logarithmic, and exponential operations, present formidable challenges in both theoretical and applied mathematics. These Equationss often elude closed-form analytical solutions, rendering numerical approximation methods indispensable for obtaining viable solutions. The necessity of robust numerical techniques for solving transcendental Equationss spans a wide array of disciplines, including physics, engineering, economics, and other scientific domains, where such Equationss frequently arise in modeling real-world phenomena. This report seeks to explore the development, functionality, and efficacy of a sophisticated Python-based "Transcendental Equations Solver," which utilizes five distinct numerical methods—Bisection, Regula Falsi, Newton's Raphson's, Muller, and Secant—to compute approximate roots of these Equationss.

The scope of this report is deliberately focused on the detailed exposition of the design and implementation of the aforementioned application, providing an in-depth analysis of the numerical methods employed, their respective computational efficiencies, and their convergence properties. The discussion will be restricted to the evaluation of the solver's performance through a set of selected transcendental Equationss, highlighting the practical advantages and limitations inherent in the various algorithms. The report also investigates the graphical user interface (GUI) design, which facilitates seamless user interaction, enabling the easy input of mathematical expressions and the intuitive presentation of results, including root approximations, iteration counts, and graphical depictions of Equations behavior.

The plan of development within this report is structured around several key sections: an examination of the numerical methods employed, a thorough description of the application's architecture and GUI features, an empirical assessment of the solver's performance, and a critical comparison of the effectiveness of each method in different contexts. Furthermore, the report will address the underlying assumptions made during the development of the solver, and conclude with a set of recommendations for enhancing the functionality and computational efficiency of the application.

The thesis of this report posits that the "Transcendental Equations Solver" represents a highly effective and versatile tool for the numerical resolution of transcendental Equationss. By integrating multiple solution methods, the application offers users a comprehensive approach to solving complex mathematical problems, ensuring both flexibility and precision. In conclusion, the report advocates for the continued optimization of the solver to enhance its computational speed and extend its applicability to an even broader spectrum of mathematical functions.

3. Methodology

The development and evaluation of the "Transcendental Equations Solver" application were conducted through a series of systematic steps, focusing on the design and implementation of the numerical methods, user interface, and performance evaluation. The following section provides a comprehensive and precise account of the methodology employed to construct the solver, detailing the procedures for solving transcendental Equationss and evaluating the effectiveness of each method.

3.1. Selection of Numerical Methods

The solver utilizes five classical numerical methods for solving transcendental Equationss:

3.1.1. Bisection Method: This method is based on the intermediate value theorem, requiring two initial guesses that bracket the root. The method iteratively narrows down the interval in which the root lies, converging linearly to the solution.

$$c = \frac{a + b}{2}$$

3.1.2. Regula Falsi (False Position) Method: Similar to the bisection method, the Regula Falsi method also requires two initial guesses, but it refines the estimates by linearly interpolating between the two points. It often converges more quickly than the bisection method but can suffer from slow convergence in some cases.

$$c = \frac{a \cdot f(b) - b \cdot f(a)}{f(b) - f(a)}$$

3.1.3. Newton's Raphson Method: An iterative method that uses the derivative of the function. Starting from an initial guess, the method computes successive approximations by the formula:

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

Newton's Raphson method generally converges rapidly when the initial guess is close to the root but can fail if the derivative is zero or the guess is too far from the true root.

3.1.4. Muller's Method: An extension of Newton's Raphson's method that uses quadratic interpolation to approximate the root. It employs three points to estimate the root and can be more robust in cases where Newton's Raphson's method fails.

$$A = \frac{(x_{i-2} - x_i)(y_{i-1} - y_i) - (x_{i-1} - x_i)(y_{i-2} - y_i)}{(x_{i-1} - x_{i-2})(x_{i-1} - x_i)(x_{i-2} - x_i)}$$

$$B = \frac{(x_{i-2} - x_i)^2(y_{i-1} - y_i) - (x_{i-1} - x_i)^2(y_{i-2} - y_i)}{(x_{i-2} - x_{i-1})(x_{i-1} - x_i)(x_{i-2} - x_i)}$$

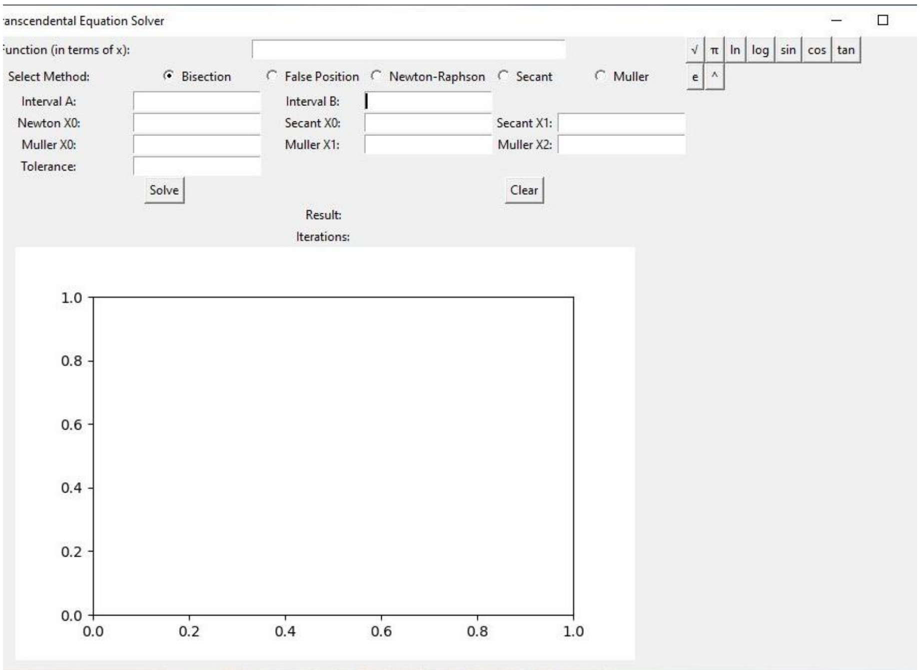
$$x_{i+1} - x_i = -\frac{2y_i}{B \pm \sqrt{B^2 - 4Ay_i}}$$

3.1.5. Secant Method: A derivative-free method that approximates the root by using a secant line to estimate the next point. It requires two initial guesses and converges faster than the bisection method but slower than Newton's Raphson's method in most cases.

$$x_2 = x_1 - \frac{f(x_1) \cdot (x_1 - x_0)}{f(x_1) - f(x_0)}$$

3.2. Application Design and User Interface

The "Transcendental Equations Solver" was implemented using Python, leveraging the Tkinter library for the development of the Graphical User Interface (GUI). The interface was designed to be intuitive and user-friendly, enabling users to input mathematical expressions involving transcendental functions and perform computations effortlessly.

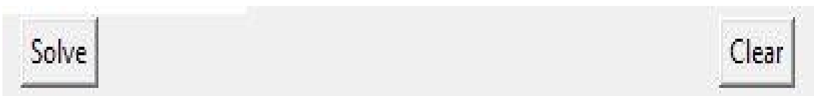


The key components of the GUI include:

3.2.1. Input Fields: Users can enter the function they wish to solve, using Python-compatible syntax for functions such as sin, cos, tan, ln, etc.

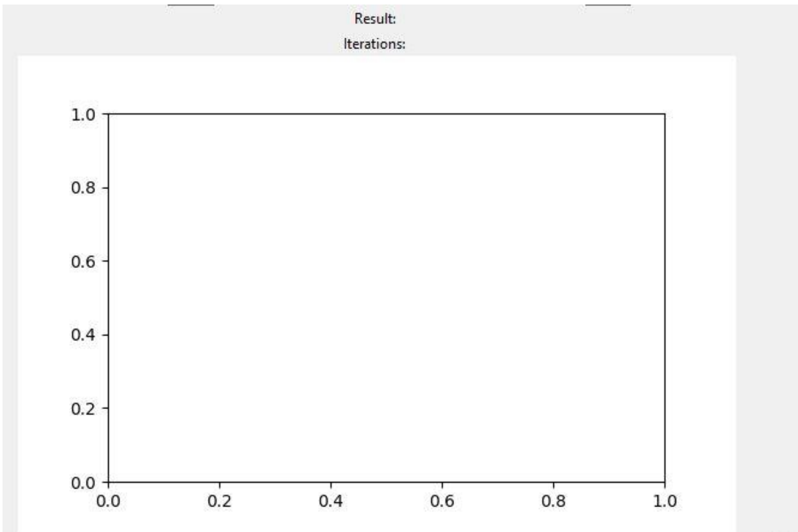
3.2.2. Method Selection: A menu allows the user to choose between the five available numerical methods.

3.2.3. Control Buttons: These include buttons for solving the Equations, clearing the inputs, and performing operations such as sine, cosine, tangent, logarithm, and square root.



3.2.4. Tolerance: The tolerance determines the precision of the solution in numerical methods. In this application, users have the flexibility to input their desired tolerance value to suit the accuracy requirements of their calculations. If no tolerance is specified, the application uses a default built-in tolerance of 10^{-7}, ensuring highly precise results. This feature provides convenience for users who may need a specific precision or are satisfied with the reliable default setting.

3.2.5. Output Display: Once the Equations is solved, the results, including the root, number of iterations, and graphical plot, are displayed.



3.2.6. Error Handling: There will be an input error, program generates if the user input intervals that have the same sign.

3. 3. Implementation Procedure

3.3.1. Equations Parsing: The first step involves parsing the user-provided mathematical expression into a format that can be evaluated programmatically. The sympy library in Python was employed for symbolic manipulation, allowing for the differentiation of functions and the handling of symbolic math expressions.

3.3.2. Initial Guess Selection: For each method, an initial guess is required. The solver provides an input field for the user to enter this guess. If the user does not provide an initial guess, the application will prompt them for one.

3.3.3. Root Calculation: Based on the selected method, the root of the Equations is calculated. Each method operates iteratively, updating the current approximation of the root based on the function's behavior until a stopping criterion is met. The stopping criterion is tolerance equal to 10^{-7}.

3.3.4. Graphical Representation: After calculating the root, the application generates a graphical plot of the Equations within the specified domain, using the matplotlib library. This visual representation aids the user in understanding the behavior of the Equations and the location of the root.

3.3.5. Iteration Tracking: Each method tracks the number of iterations performed during the solution process. The iteration count is displayed alongside the root and other results for the user's reference.

3.4. Data Collection and Evaluation

To assess the effectiveness of the solver, a series of test cases were created, including both well-behaved and problematic transcendental Equationss. The test cases aimed to evaluate the convergence speed, accuracy, and robustness of each numerical method. The key performance metrics for each method include:

3.4.1. Convergence Rate: How quickly the method approaches the true root.

3.4.2. Accuracy: The proximity of the computed root to the true solution, determined by comparing the results with known exact solutions or highly accurate approximations.

3.4.3. Efficiency: The number of iterations required for each method to converge to the root, with a tolerance level set to 10^{-7}.

The results were then analyzed to draw comparisons between the methods, highlighting their strengths and weaknesses in solving transcendental Equationss under various conditions

3.5. Assumptions and Limitations

3.5.1. Well-posed Equationss: It is assumed that the input Equationss are well-behaved, meaning they have at least one real root in the specified domain.

3.5.2. Convergence Criteria: For methods like Newton's Raphson's and Secant, convergence is heavily dependent on the initial guess. Poor initial guesses may result in divergence or slow convergence.

3.5.3. Derivative Availability: Methods such as Newton's Raphson's and Muller’s require the availability of derivatives, and special cases (e.g., non-differentiable points) may hinder the performance of these methods.

3.6. Analysis and Reporting

After completing the tests, the results were analyzed based on the convergence speed, accuracy, and efficiency of the numerical methods. The performance of each method was discussed in terms of its suitability for different types of transcendental Equationss, and the findings were summarized to provide recommendations for method selection depending on the problem at hand.

4. Discussion

This section provides an in-depth analysis of the numerical solutions obtained using the **Transcendental Equations Solver** across the five methods—Bisection, Regula Falsi, Newton-Raphson, Secant, and Muller. Each method's result is analyzed in terms of its accuracy, efficiency (number of iterations), and graphical representation. Below, the results for each test case are discussed:

4.1. Bisection Method Examples:

- **Test Case 1**:

11^{th} iteration :

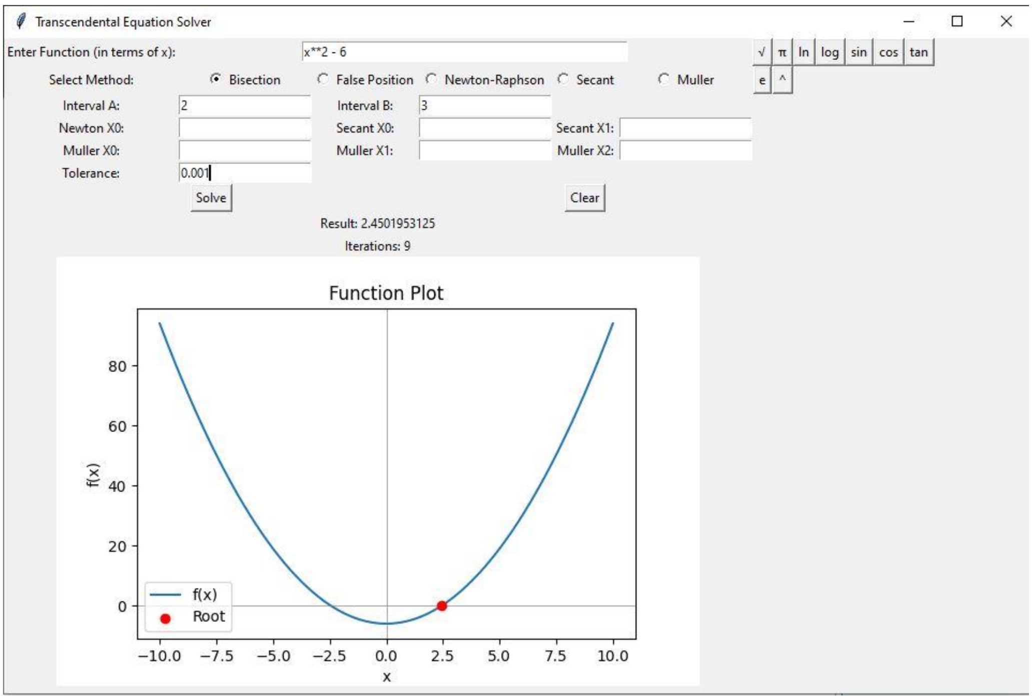
Here $f(2.4492188) = -0.0013275 < 0$ and $f(2.4501953) = 0.0034571 > 0$

$\therefore$ Now, Root lies between 2.4492188 and 2.4501953

$$x_{10} = \frac{2.4492188 + 2.4501953}{2} = 2.449707$$

$$f(x_{10}) = f(2.449707) = 2.449707^2 - 6 = 0.0010645 > 0$$

Approximate root of the equation $x^2 - 6 = 0$ using Bisection method is 2.449707 (After 11 iterations)



Question is verified from https://atozmath.com/

The first image, from *atozmath.com*, finds x = 2.449707 after 11. The second, from our *Transcendental Equations Solver*, computes x = 2.4501953125 in 9 iterations with a 0.001 tolerance. The difference reflects varying precision criteria.

- **Test Case 2**:

11^{th} iteration :

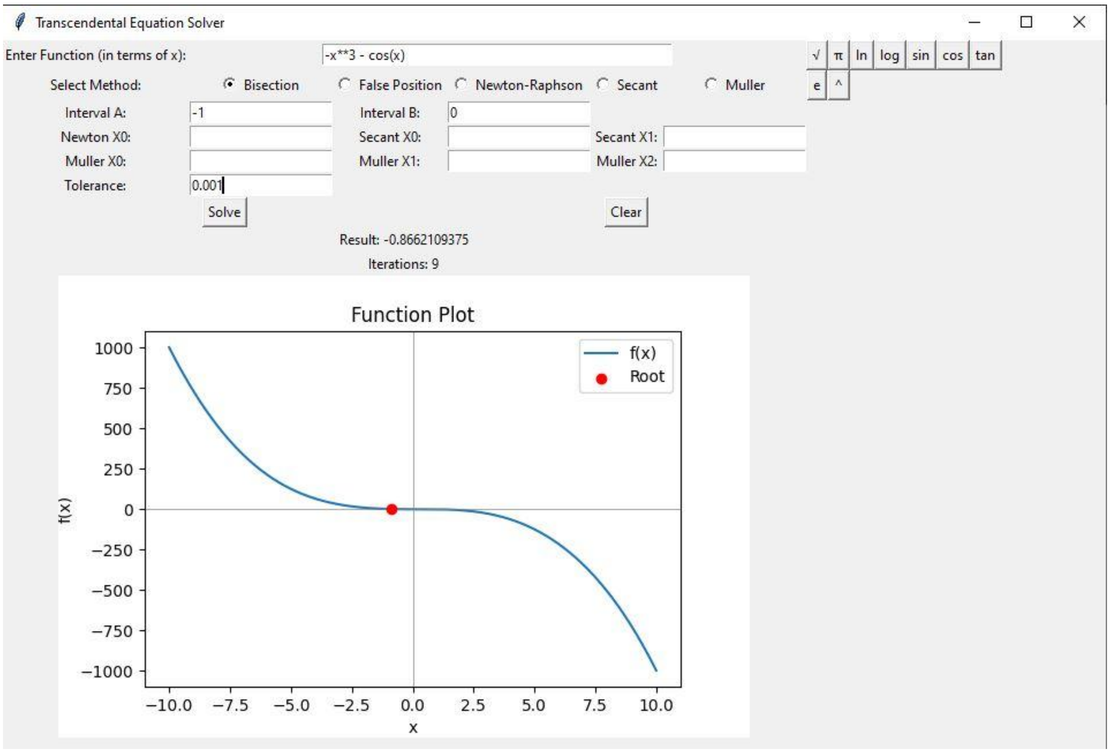
Here $f(-0.8662109) = 0.0022186 > 0$ and $f(-0.8652344) = -0.0007209 < 0$

$\therefore$ Now, Root lies between -0.8662109 and -0.8652344

$$x_{10} = \frac{-0.8662109 + (-0.8652344)}{2} = -0.8657227$$

$$f(x_{10}) = f(-0.8657227) = -(-0.8657227)^3 - \cos(-0.8657227) = 0.0007482 > 0$$

Approximate root of the equation $-x^3 - \cos(x) = 0$ using Bisection method is -0.8657227 (After 11 iterations)



Question is verified from https://atozmath.com/

The first image, from *atozmath.com*, finds x = -0.8657227 after 11. The second, from our *Transcendental Equations Solver*, computes x = -0.8662109375 in 9 iterations with a 0.001 tolerance. The difference reflects varying precision criteria.

- **Test Case 3**:

10^{th} iteration :

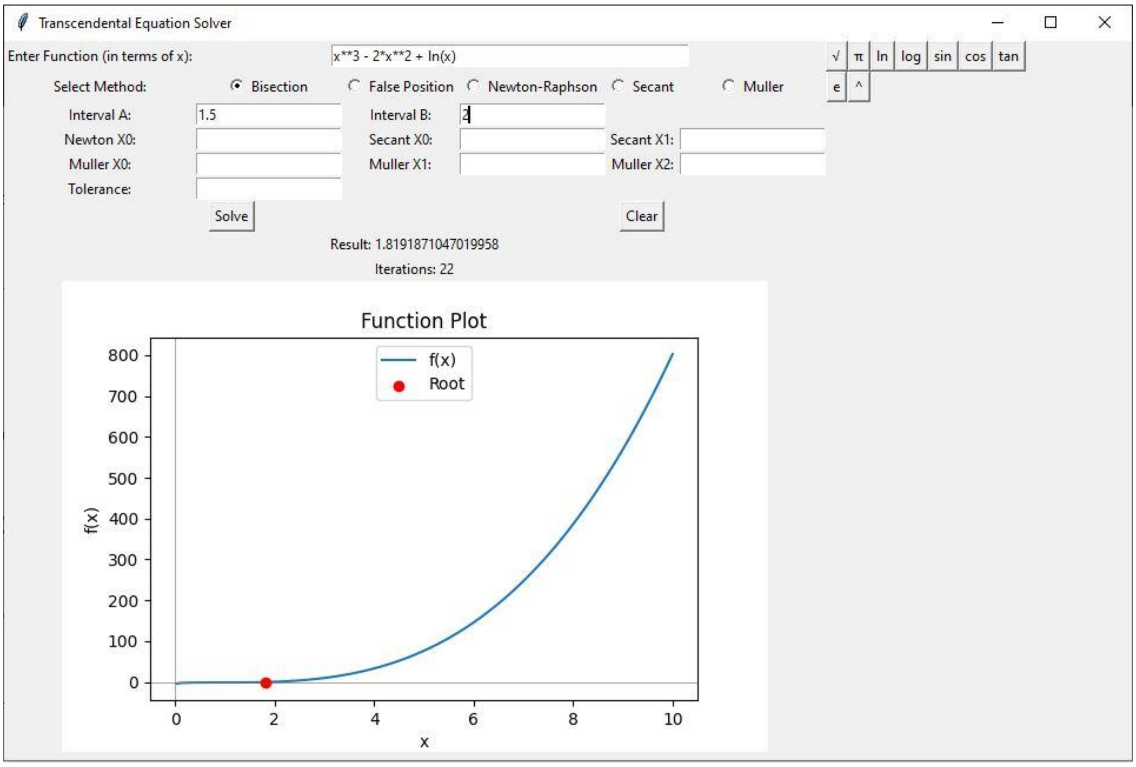
Here $f(1.8183594) = -0.0026475 < 0$ and $f(1.8203125) = 0.0036069 > 0$

$\therefore$ Now, Root lies between 1.8183594 and 1.8203125

$$x_9 = \frac{1.8183594 + 1.8203125}{2} = 1.8193359$$

$$f\left(x_9\right) = f(1.8193359) = 1.8193359^3 - 2 \cdot 1.8193359^2 + \ln(1.8193359) = 0.0004765 > 0$$

Approximate root of the equation $x^3 - 2x^2 + \ln(x) = 0$ using Bisection method is 1.8193359 (After 10 iterations)



Question is verified from https://atozmath.com/

The first image, from *atozmath.com*, finds x = 1.8193359 after 10. The second, from our *Transcendental Equations Solver*, computes x =1.8191871047019958 in 22 iterations with a 0.001 tolerance. The difference reflects varying precision criteria.

- **Test Case 4**:

9^{th} iteration :

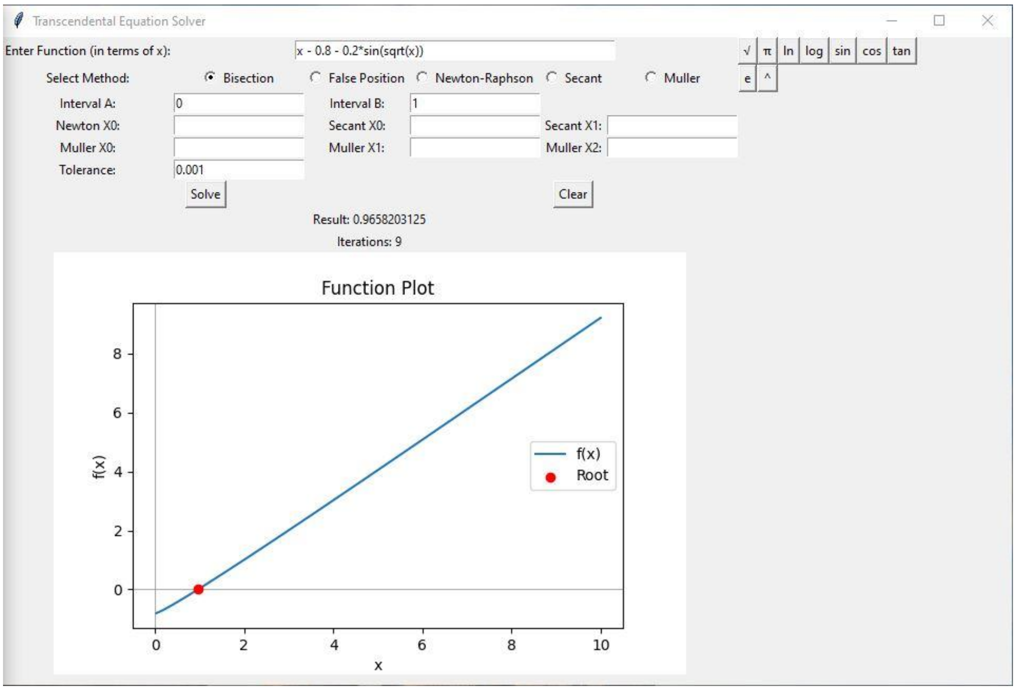
Here $f(0.9648438) = -0.0015076 < 0$ and $f(0.96875) = 0.0021784 > 0$

$\therefore$ Now, Root lies between 0.9648438 and 0.96875

$$x_8 = \frac{0.9648438 + 0.96875}{2} = 0.9667969$$

$$f(x_8) = f(0.9667969) = 0.9667969 - 0.2\sin(0.9832583) - 0.8 = 0.0003353 > 0$$

Approximate root of the equation $x - 0.2\sin\left(\sqrt{x}\right) - 0.8 = 0$ using Bisection method is 0.9667969 (After 9 iterations)



Question is verified from https://atozmath.com/

The first image, from *atozmath.com*, finds x = 0.9667969 after 9. The second, from our *Transcendental Equations Solver*, computes x = 0.9658203125 in 9 iterations with a 0.001 tolerance. The difference reflects varying precision criteria.

- **Test Case 5**:

11^{th} iteration :

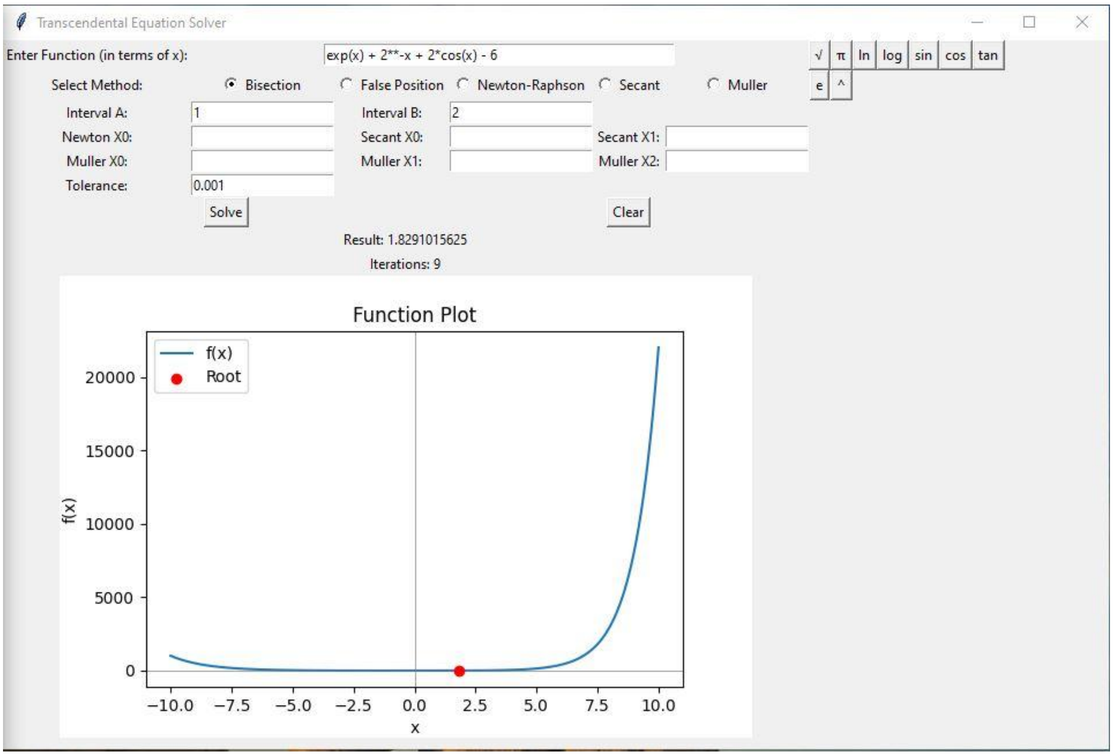
Here $f(1.8291016) = -0.0011565 < 0$ and $f(1.8300781) = 0.0028502 > 0$

$\therefore$ Now, Root lies between 1.8291016 and 1.8300781

$$x_{10} = \frac{1.8291016 + 1.8300781}{2} = 1.8295898$$

$$f(x_{10}) = f(1.8295898) = e^{1.8295898} + 2\cos(1.8295898) + \frac{1}{2^{1.8295898}} - 6 = 0.000846 > 0$$

Approximate root of the equation $e^x + 2\cos(x) + \frac{1}{2^x} - 6 = 0$ using Bisection method is 1.8295898 (After 11 iterations)



Question is verified from https://atozmath.com/

The first image, from *atozmath.com*, finds x = 1.8295898 after 11. The second, from our *Transcendental Equations Solver*, computes x = 1.82910158 in 9 iterations with a 0.001 tolerance. The difference reflects varying precision criteria.

4.2. Newton's Raphson Method Examples:

- **Test Case 1**:

3^{rd} iteration :

$f(x_2) = f(2.4495) = 2.4495^2 - 6 = 0.0001$

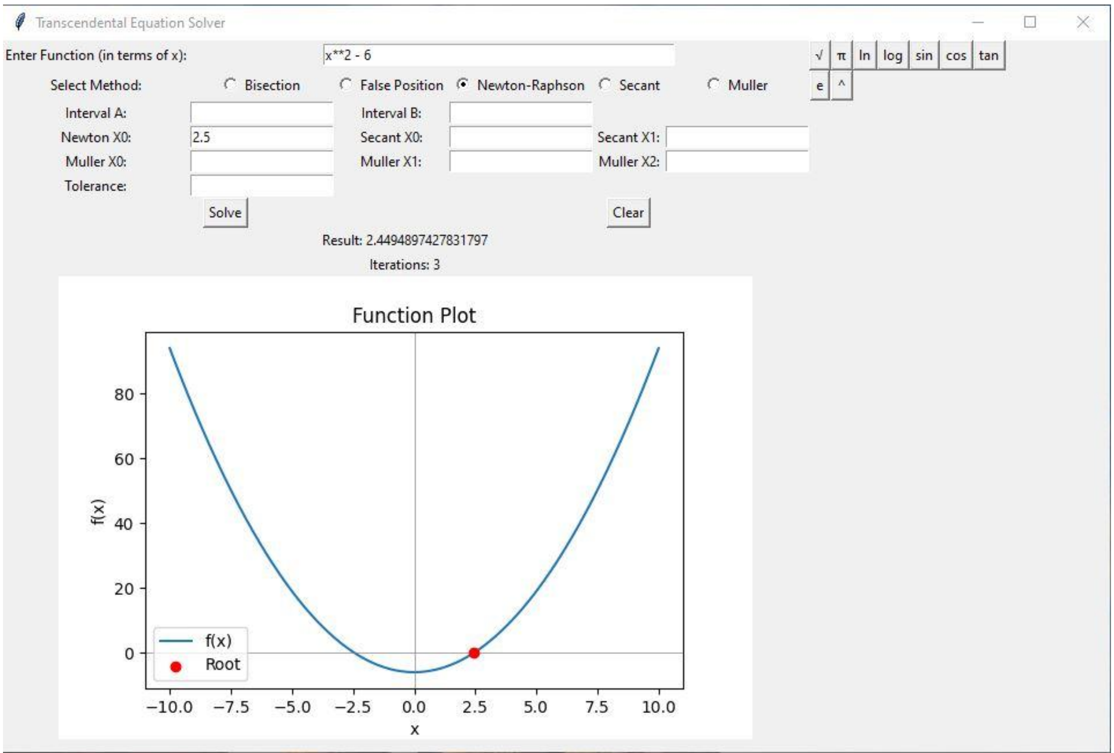
$f'(x_2) = f'(2.4495) = 2 \cdot 2.4495 = 4.899$

$$x_3 = x_2 - \frac{f(x_2)}{f'(x_2)}$$

$$x_3 = 2.4495 - \frac{0.0001}{4.899}$$

$x_3 = 2.4495$

Approximate root of the equation $x^2 - 6 = 0$ using Newton Raphson method is 2.4495 (After 3 iterations)



Question is verified from https://atozmath.com/

The two images demonstrate solving $x^2-6=0$ using the Newton-Raphson method. The first image from atozmath.com calculates the root, reaching x=2.4495 after 3 iterations. The second image, from our *Transcendental Equations Solver* application, computes the root as x=2.4494897427831797, also in 3 iterations. Both methods arrive at nearly the same result, offering an efficient result.

- **Test Case 2**:

5^{th} iteration :

$$f(x_4) = f(-0.8655) = -(-0.8655)^3 - \cos(-0.8655) = 0.0001$$

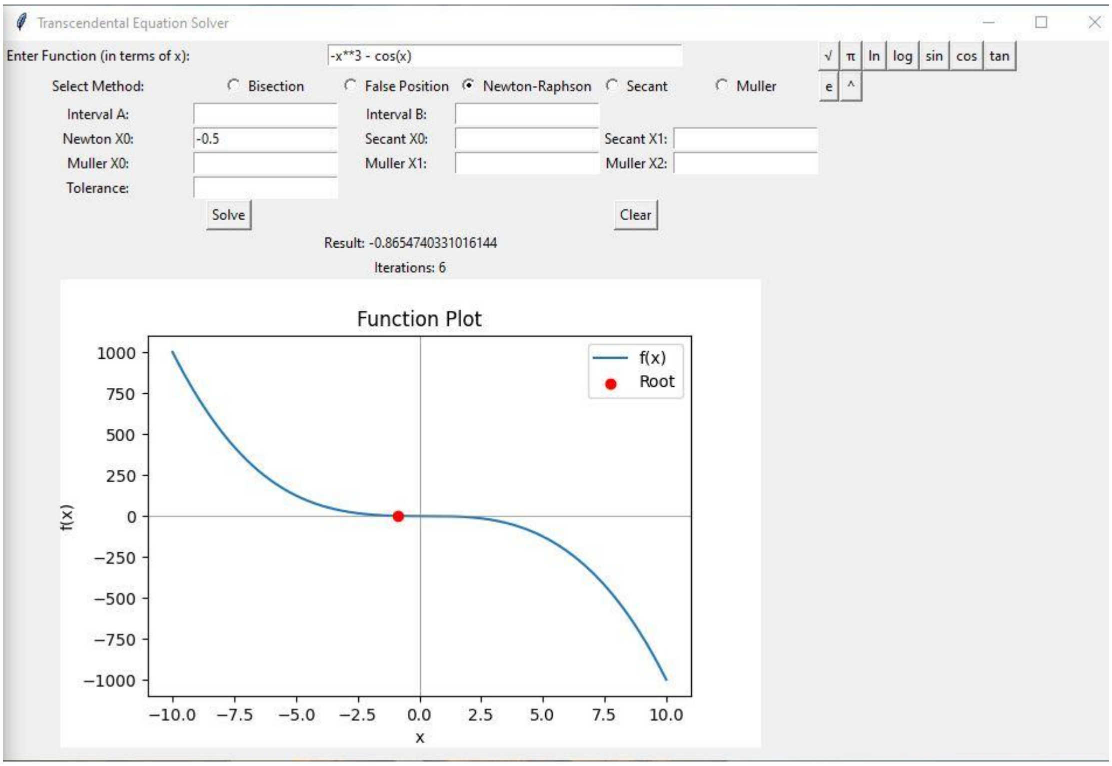
$$f'(x_4) = f'(-0.8655) = -3 \cdot (-0.8655)^2 + \sin(-0.8655) = -3.0087$$

$$x_5 = x_4 - \frac{f(x_4)}{f'(x_4)}$$

$$x_5 = -0.8655 - \frac{0.0001}{-3.0087}$$

$$x_5 = -0.8655$$

Approximate root of the equation $-x^3 - \cos(x) = 0$ using Newton Raphson method is -0.8655 (After 5 iterations)



Question is verified from https://atozmath.com/

The two images demonstrate solving $-x^3$-cos(x)=0 using the Newton-Raphson method. The first image from atozmath.com calculates the root, reaching x=-0.8655 after 5 iterations. The second image, from our *Transcendental Equations Solver* application, computes the root as x=-0.8654, also in 6 iterations. Both methods arrive at nearly the same result, offering an efficient result.

- **Test Case 3**:

5^{th} iteration :

$$f(x_4) = f(1.8192) = 1.8192^3 - 2 \cdot 1.8192^2 + \ln(1.8192) = 0$$

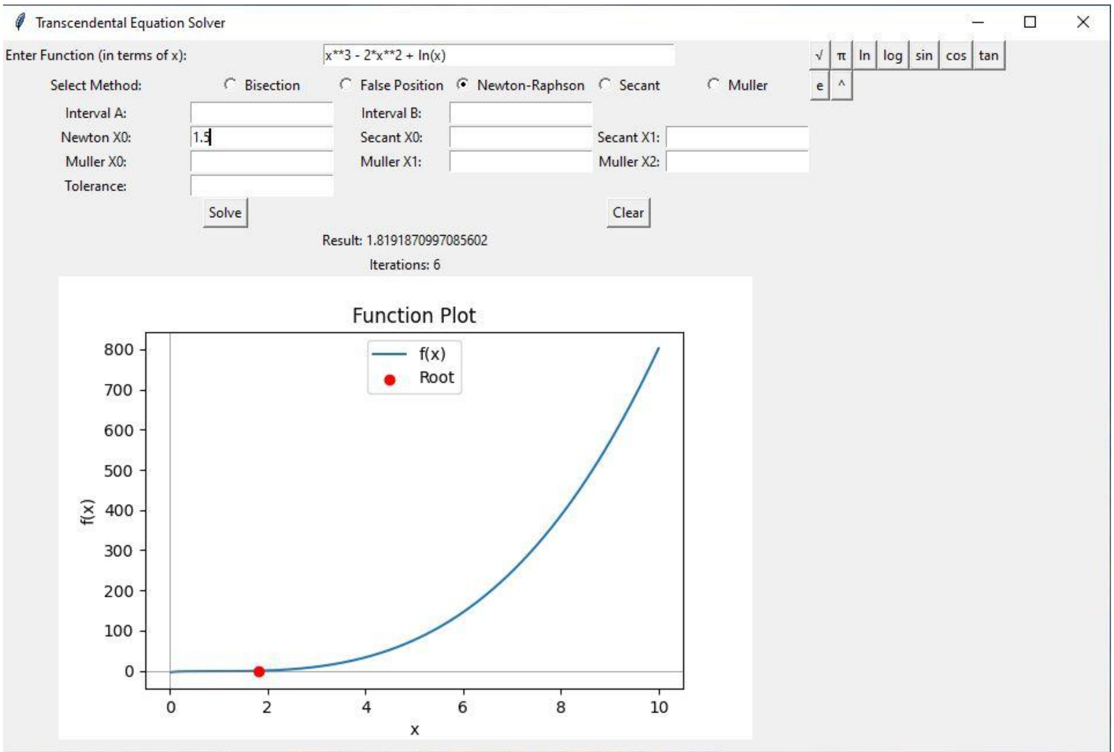
$$f\,(x_4) = f\,(1.8192) = 3 \cdot 1.8192^2 - 4 \cdot 1.8192 + \frac{1}{1.8192} = 3.2014$$

$$x_5 = x_4 - \frac{f(x_4)}{f\,(x_4)}$$

$$x_5 = 1.8192 - \frac{0}{3.2014}$$

$$x_5 = 1.8192$$

Approximate root of the equation $x^3 - 2x^2 + \ln(x) = 0$ using Newton Raphson method is 1.8192 (After 5 iterations)



Question is verified from https://atozmath.com/

The two images demonstrate solving $x^3-2x^2+\ln(x)=0$ using the Newton-Raphson method. The first image from atozmath.com calculates the root, reaching x=1.8192 after 5 iterations. The second image, from our *Transcendental Equations Solver* application, computes the root as x=1.8191, also in 6 iterations. Both methods arrive at nearly the same result, offering an efficient result.

- **Test Case 4**:

3^{rd} iteration :

$$f(x_2) = f(0.9665) = 0.9665 - 0.2\sin(0.9831) - 0.8 = 0.0001$$

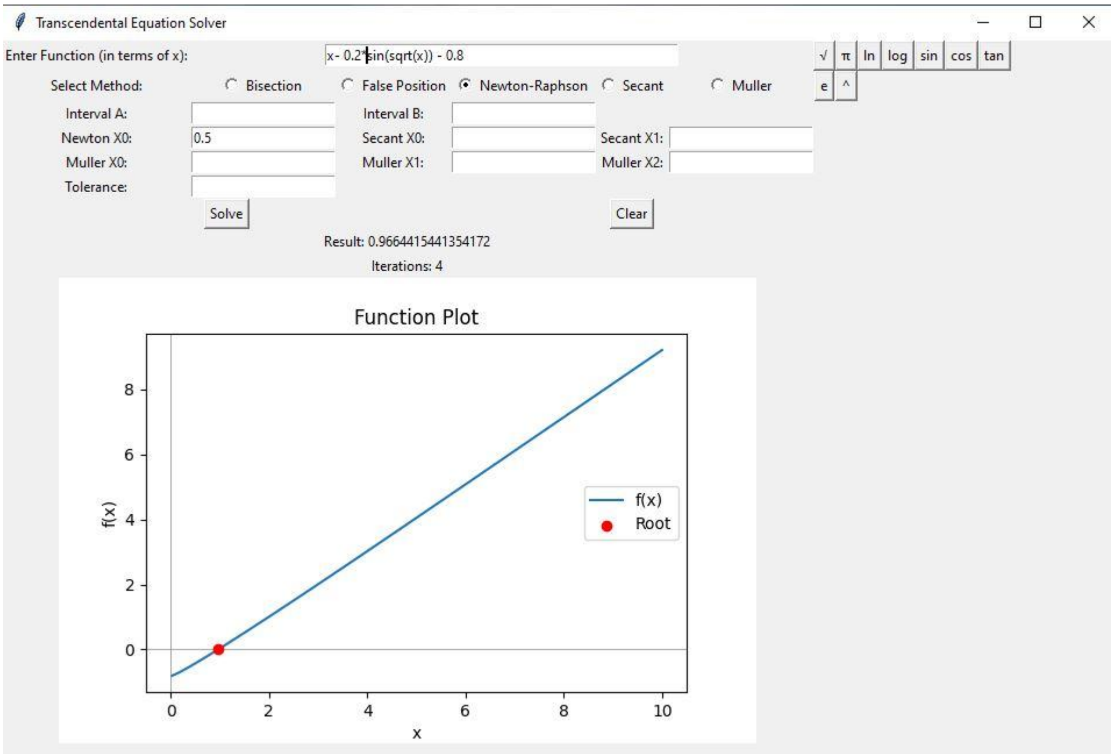
$$f\ (x_2) = f\ (0.9665) = 1 - \frac{0.1\cos(0.9831)}{\sqrt{0.9665}} = 0.9436$$

$$x_3 = x_2 - \frac{f(x_2)}{f\ (x_2)}$$

$$x_3 = 0.9665 - \frac{0.0001}{0.9436}$$

$$x_3 = 0.9664$$

Approximate root of the equation $x - 0.2\sin\left(\sqrt{x}\right) - 0.8 = 0$ using Newton Raphson method is 0.9664
(After 3 iterations)



Question is verified from https://atozmath.com/

The two images demonstrate solving x-0.2sin(sqrt(x))-0.8=0 using the Newton-Raphson method. The first image from atozmath.com calculates the root, reaching x=0.9664 after 3iterations. The second image, from our *Transcendental Equations Solver* application, computes the root as x=0.966441544, also in 4 iterations. Both methods arrive at nearly the same result, offering an efficient result.

- **Test Case 5**:

4^{th} iteration :

$$f(x_3) = f(1.8295) = e^{1.8295} + 2\cos(1.8295) + \frac{1}{2^{1.8295}} - 6 = 0.0005$$

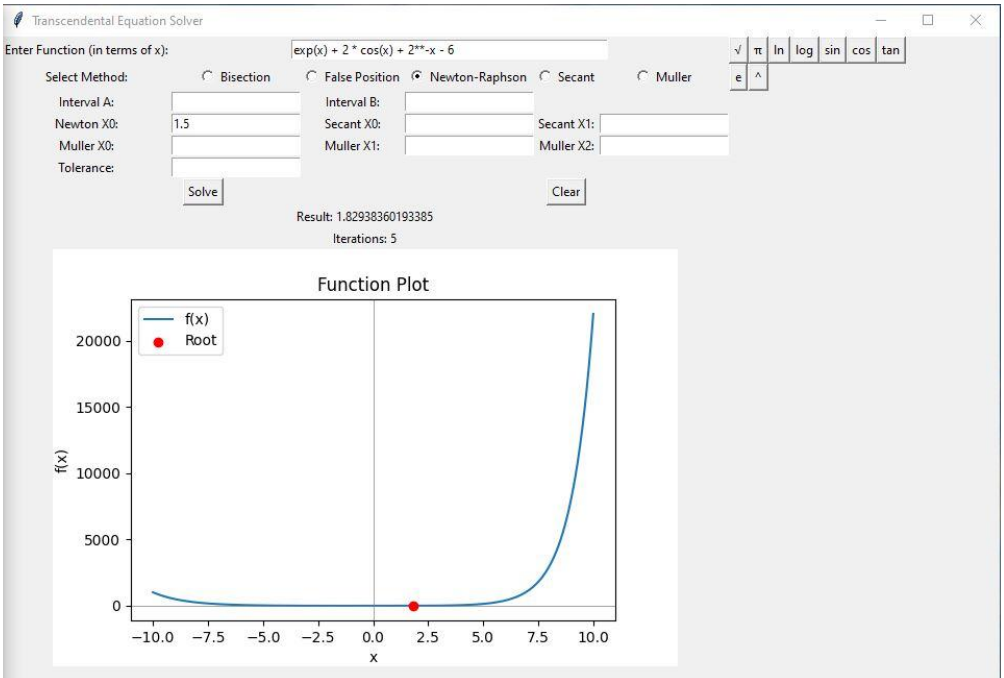
$$f\,(x_3) = f\,(1.8295) = e^{1.8295} - 2\sin(1.8295) - \frac{\ln(2)}{2^{1.8295}} = 4.1023$$

$$x_4 = x_3 - \frac{f(x_3)}{f\,(x_3)}$$

$$x_4 = 1.8295 - \frac{0.0005}{4.1023}$$

$$x_4 = 1.8294$$

Approximate root of the equation $e^x + 2\cos(x) + \frac{1}{2^x} - 6 = 0$ using Newton Raphson method is 1.8294 (After 4 iterations)



Question is verified from https://atozmath.com/

The two images demonstrate solving $e^x+2\cos(x)+2^{-x}-6=0$ using the Newton-Raphson method. The first image from atozmath.com calculates the root, reaching x=1.8294 after 4 iterations. The second image, from our *Transcendental Equations Solver* application, computes the root as x=1.82938360, also in 5 iterations. Both methods arrive at nearly the same result, offering an efficient result.

4.3. Regula Falsi (False Position) Method Examples:

- **Test Case 1**:

4^{th} iteration :

Here $f(2.449) = -0.0025 < 0$ and $f(3) = 3 > 0$

$\therefore$ Now, Root lies between $x_0 = 2.449$ and $x_1 = 3$

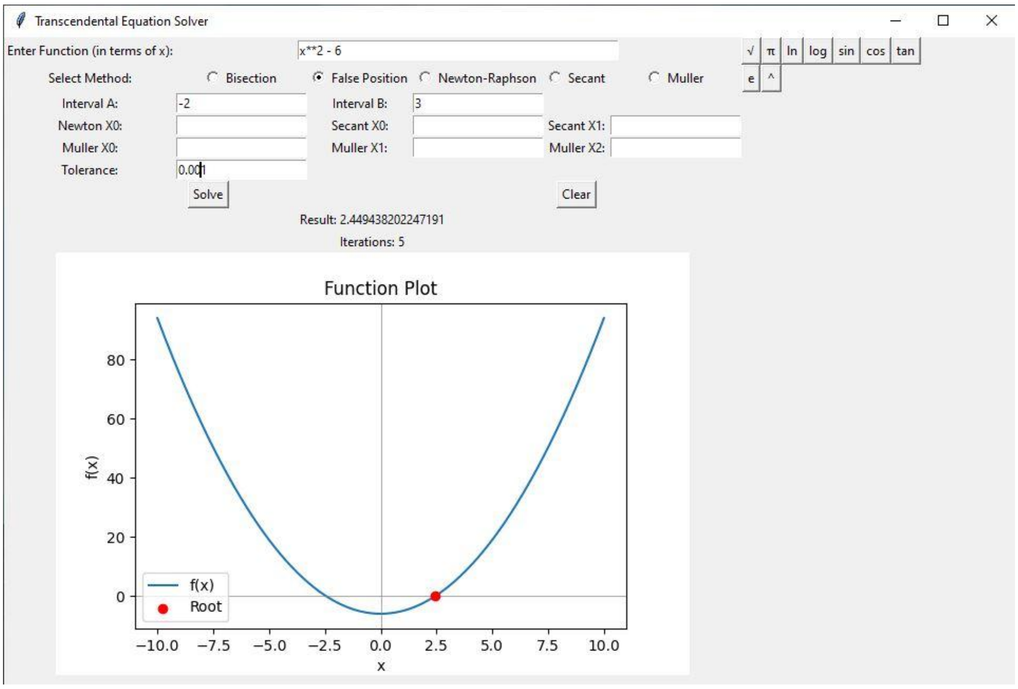
$$x_5 = x_0 - f(x_0) \cdot \frac{x_1 - x_0}{f(x_1) - f(x_0)}$$

$$x_5 = 2.449 - (-0.0025) \cdot \frac{3 - 2.449}{3 - (-0.0025)}$$

$$x_5 = 2.4494$$

$$f(x_5) = f(2.4494) = 2.4494^2 - 6 = -0.0003 < 0$$

Approximate root of the equation $x^2 - 6 = 0$ using False Position method is 2.4494 (After 4 iterations)



Question is verified from https://atozmath.com/

The Equations $x^2-6=0$ was solved using the Regula Falsi method in both sources, yielding the root 2.44952449524495. The first source, atozmath.com, presents detailed steps at 4 iterations, while our application *Transcendental Equations Solver* provides the same result at 5 iteratinos with a graphical representation, highlighting its visual advantage.

- **Test Case 2**:

5^{th} iteration :

Here $f(-1) = 0.4597 > 0$ and $f(-0.8651) = -0.0011 < 0$

$\therefore$ Now, Root lies between $x_0 = -1$ and $x_1 = -0.8651$

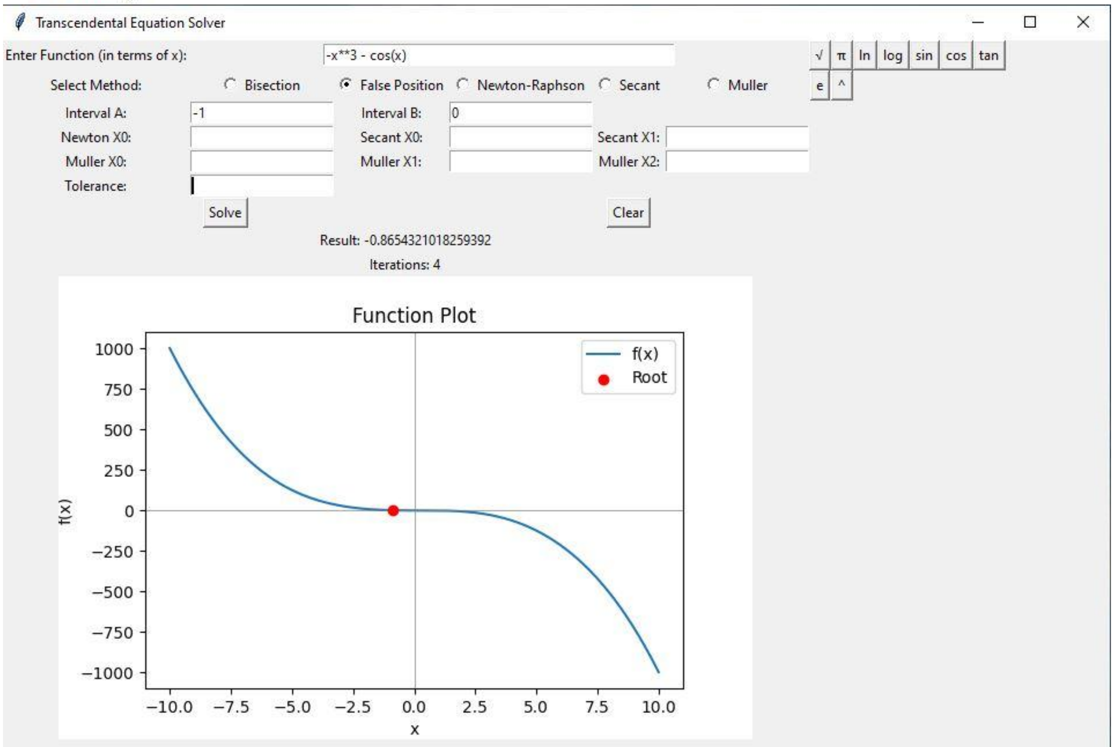
$$x_6 = x_0 - f(x_0) \cdot \frac{x_1 - x_0}{f(x_1) - f(x_0)}$$

$$x_6 = -1 - 0.4597 \cdot \frac{-0.8651 - (-1)}{-0.0011 - 0.4597}$$

$$x_6 = -0.8654$$

$$f(x_6) = f(-0.8654) = -(-0.8654)^3 - \cos(-0.8654) = -0.0001 < 0$$

Approximate root of the equation $-x^3 - \cos(x) = 0$ using False Position method is -0.8654 (After 5 iterations)



Question is verified from https://atozmath.com/

The Equations $-x^3$-cos(x)=0 was solved using the Regula Falsimethod in both sources, yielding the root -0.8654. The first source, atozmath.com, presents detailed steps at 5 iterations, while our application *Transcendental Equations Solver* provides the same result at 4 iterations with a graphical representation, highlighting its visual advantage.

- **Test Case 3**:

6^{th} iteration :

Here $f(1.819) = -0.0006 < 0$ and $f(2) = 0.6931 > 0$

$\therefore$ Now, Root lies between $x_0 = 1.819$ and $x_1 = 2$

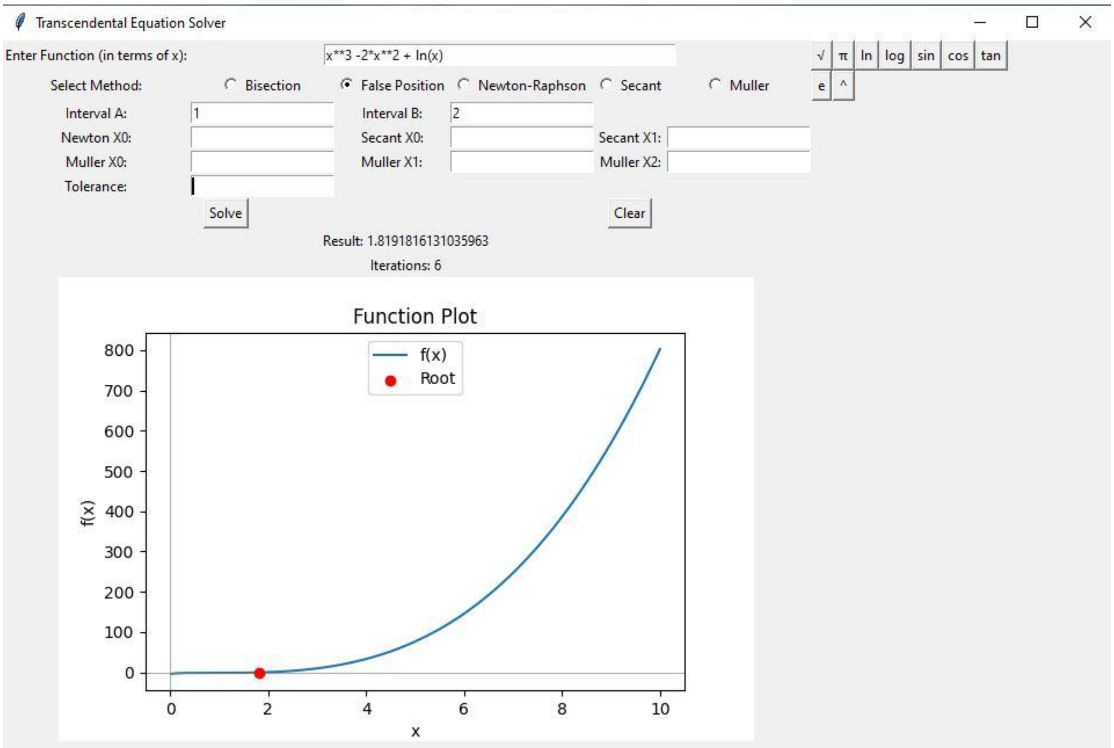
$$x_7 = x_0 - f(x_0) \cdot \frac{x_1 - x_0}{f(x_1) - f(x_0)}$$

$$x_7 = 1.819 - (-0.0006) \cdot \frac{2 - 1.819}{0.6931 - (-0.0006)}$$

$$x_7 = 1.8192$$

$$f(x_7) = f(1.8192) = 1.8192^3 - 2 \cdot 1.8192^2 + \ln(1.8192) = -0.0001 < 0$$

Approximate root of the equation $x^3 - 2x^2 + \ln(x) = 0$ using False Position method is 1.8192 (After 6 iterations)



Question is verified from https://atozmath.com/

The Equations $x^3-2x^2+\ln(x)=0$ was solved using the Regula Falsimethod in both sources, yielding the root 1.81921816 at 6 iterations. The first source, atozmath.com, presents detailed steps, while our application *Transcendental Equations Solver* provides the same result with a graphical representation, highlighting its visual advantage.

- **Test Case 4**:

2^{nd} iteration :

Here $f(0.9619) = -0.0043 < 0$ and $f(1) = 0.0317 > 0$

$\therefore$ Now, Root lies between $x_0 = 0.9619$ and $x_1 = 1$

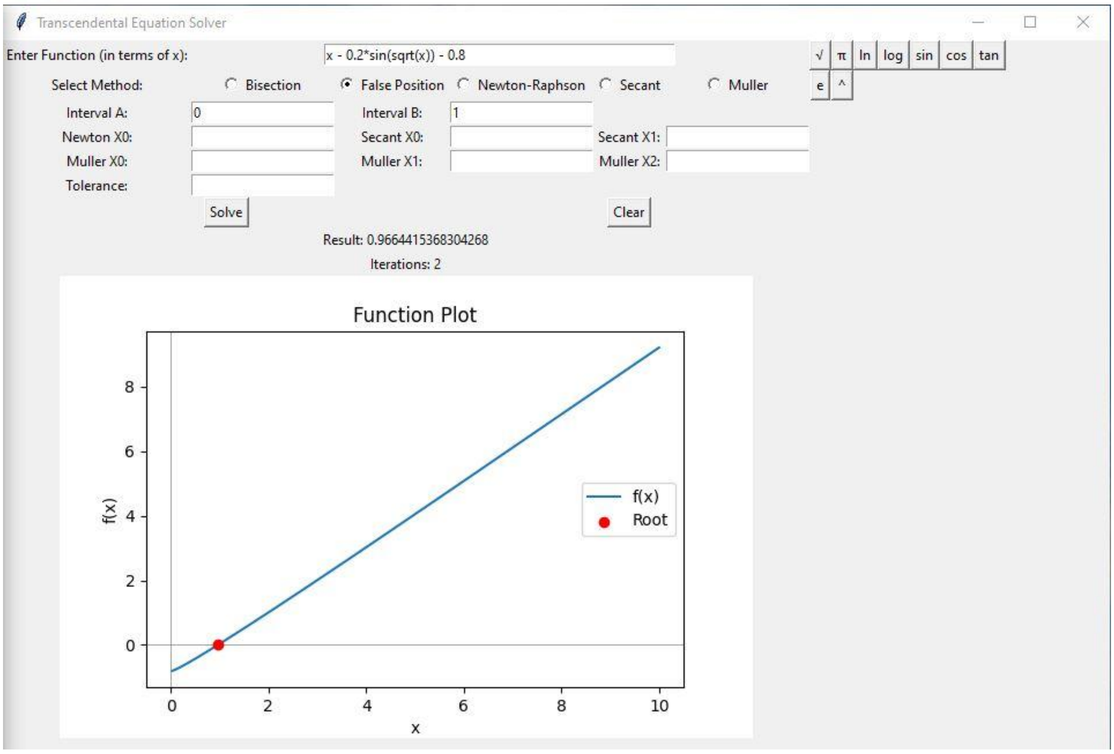
$$x_3 = x_0 - f(x_0) \cdot \frac{x_1 - x_0}{f(x_1) - f(x_0)}$$

$$x_3 = 0.9619 - (-0.0043) \cdot \frac{1 - 0.9619}{0.0317 - (-0.0043)}$$

$$x_3 = 0.9664$$

$$f(x_3) = f(0.9664) = 0.9664 - 0.2\sin(0.9831) - 0.8 = 0 < 0$$

Approximate root of the equation $x - 0.2\sin(\sqrt{x}) - 0.8 = 0$ using False Position method is 0.9664 (After 2 iterations)



Question is verified from https://atozmath.com/

The Equations x-0.2(sqrt(x))-0.8=0 was solved using the Regula Falsimethod in both sources, yielding the root 0.966441152 at 2 iterations. The first source, atozmath.com, presents detailed steps, while our application *Transcendental Equations Solver* provides the same result with a graphical representation, highlighting its visual advantage.

- **Test Case 5**:

5^{th} iteration :

Here $f(1.829) = -0.0015 < 0$ and $f(2) = 0.8068 > 0$

$\therefore$ Now, Root lies between $x_0 = 1.829$ and $x_1 = 2$

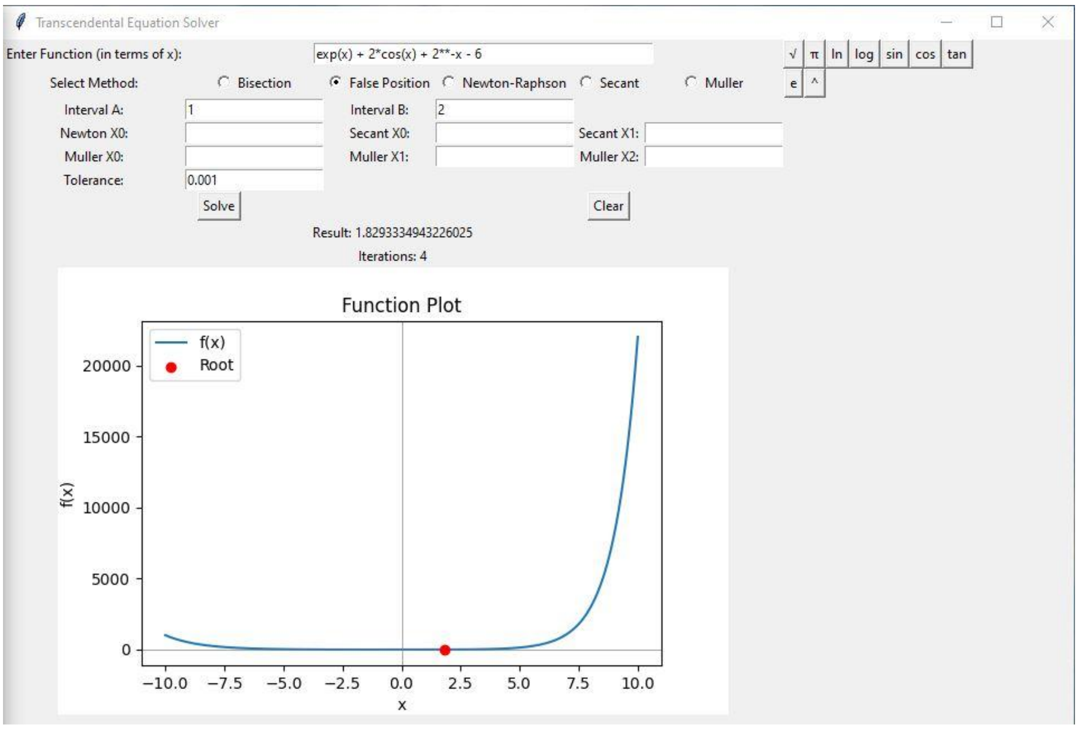
$$x_6 = x_0 - f(x_0) \cdot \frac{x_1 - x_0}{f(x_1) - f(x_0)}$$

$$x_6 = 1.829 - (-0.0015) \cdot \frac{2 - 1.829}{0.8068 - (-0.0015)}$$

$$x_6 = 1.8293$$

$$f(x_6) = f(1.8293) = e^{1.8293} + 2\cos(1.8293) + \frac{1}{2^{1.8293}} - 6 = -0.0002 < 0$$

Approximate root of the equation $e^x + 2\cos(x) + \frac{1}{2^x} - 6 = 0$ using False Position method is 1.8293 (After 5 iterations)



Question is verified from https://atozmath.com/

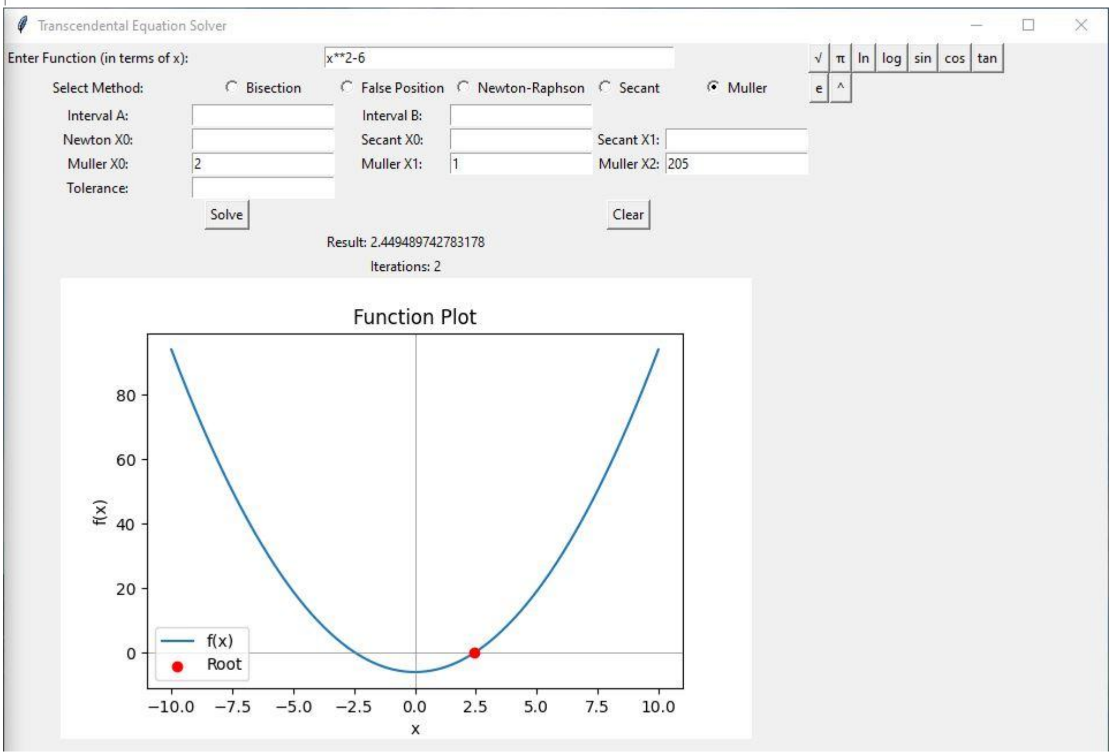
The Equations $e^x+2\cos(x)+2^{-x}+6=0$ was solved using the Regula Falsimethod in both sources, yielding the root 1.82933. The first source, atozmath.com, presents detailed steps at 5 iteration, while our application *Transcendental Equations Solver* provides the same result at 4 iterations with a graphical representation, highlighting its visual advantage.

4.4. Muller's Method Examples:

- **Test Case 1**:

Approximate root of the equation $x^2 - 6 = 0$ using Muller method is 2.4495 (After 2 iterations)

n	x_0	x_1	x_2	$f(x_0)$	$f(x_1)$	$f(x_2)$	a	b	c	x_3
1	2	3	2.5	-2	3	0.25	1	5	0.25	2.4495
2	3	2.5	2.4495	3	0.25	0.0001	1.0018	4.8979	0.0001	2.4495



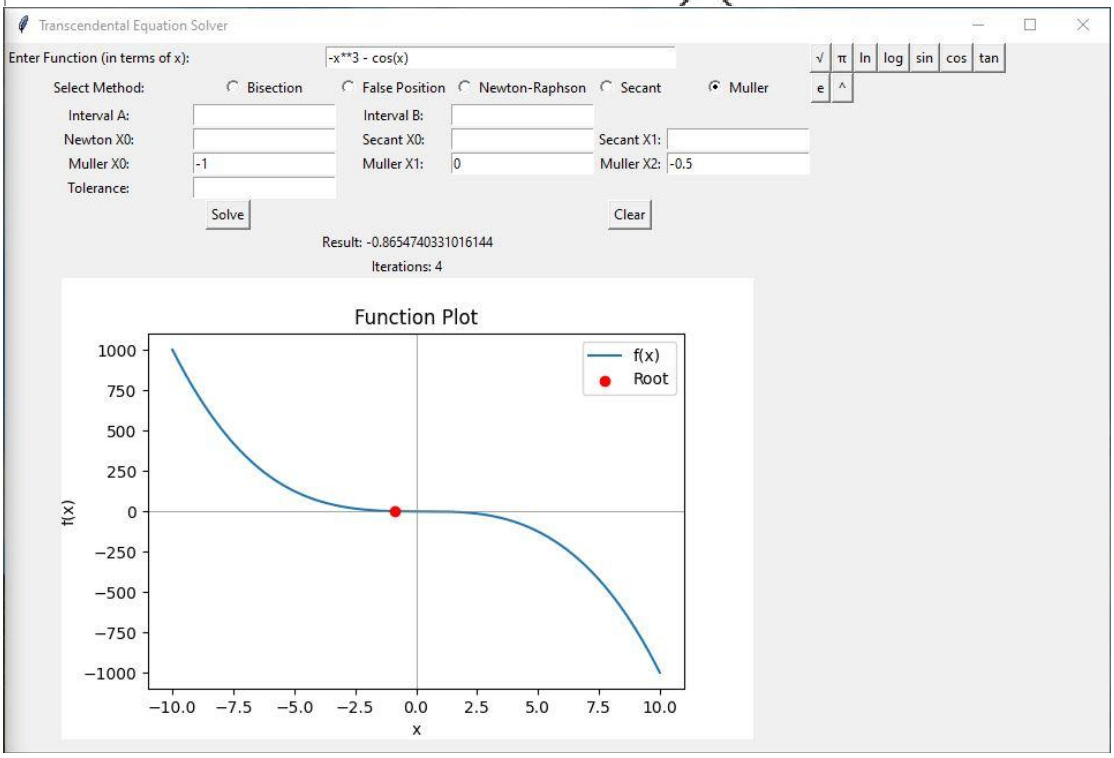
Question is verified from https://atozmath.com/

The Equations $x^2-6=0$ was solved using the Muller method in both sources, yielding the root 2.44952449524495 after 2 iterations. The first source, atozmath.com, presents detailed steps, while our application *Transcendental Equations Solver* provides the same result with a graphical representation, highlighting its visual advantage.

- **Test Case 2**:

Approximate root of the equation $-x^3$ - cos(x) = 0 using Muller method is -0.8655 (After 4 iterations)

n	x_0	x_1	x_2	$f(x_0)$	$f(x_1)$	$f(x_2)$	a	b	c	x_3
1	-1	0	-0.5	0.4597	-1	-0.7526	1.9297	-1.4597	-0.7526	-0.8519
2	0	-0.5	-0.8519	-1	-0.7526	-0.0404	1.7951	-2.6557	-0.0404	-0.8669
3	-0.5	-0.8519	-0.8669	-0.7526	-0.0404	0.0044	2.5868	-3.0121	0.0044	-0.8655
4	-0.8519	-0.8669	-0.8655	-0.0404	0.0044	0	2.91	-3.0085	0	-0.8655



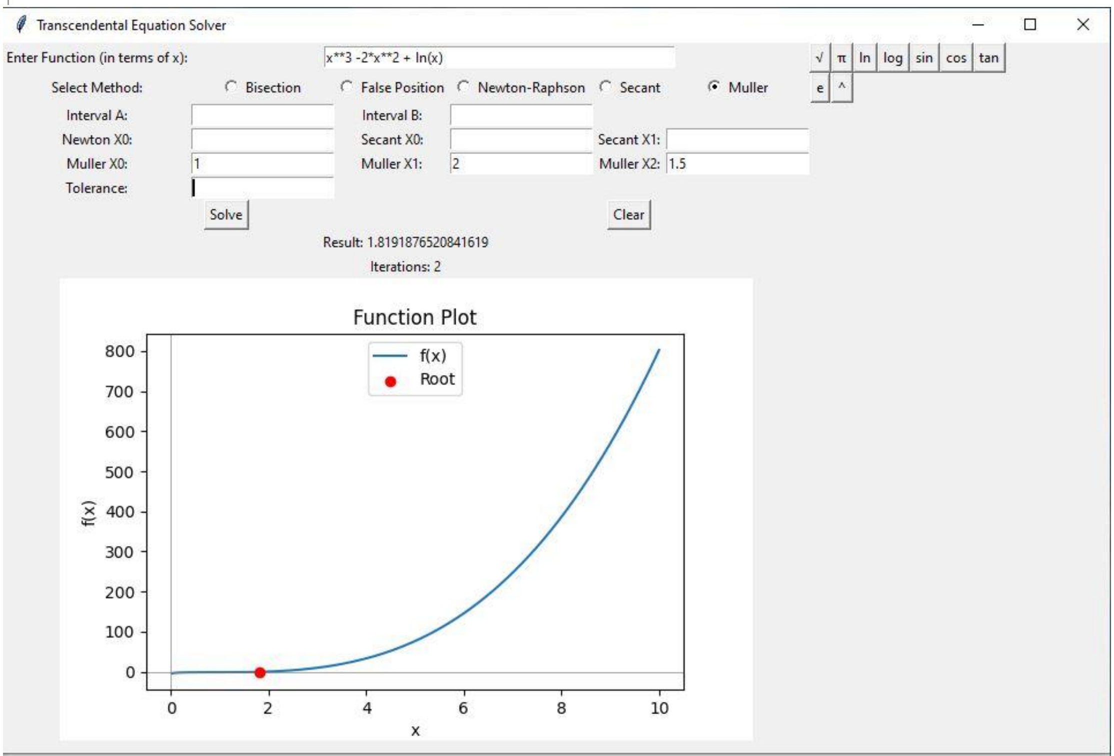
Question is verified from https://atozmath.com/

The Equations $-x^3$ - cos(x) = 0 was solved using the Muller method in both sources, yielding the root -0.865474033106144 after 4 iterations. The first source, atozmath.com, presents detailed steps, while our application *Transcendental Equations Solver* provides the same result with a graphical representation, highlighting its visual advantage.

- **Test Case 3**:

Approximate root of the equation $x^3 - 2x^2 + \ln(x) = 0$ using Muller method is 1.8192 (After 3 iterations)

n	x_0	x_1	x_2	$f(x_0)$	$f(x_1)$	$f(x_2)$	a	b	c	x_3
1	1	2	1.5	-1	0.6931	-0.7195	2.2644	1.6931	-0.7195	1.8025
2	2	1.5	1.8025	0.6931	-0.7195	-0.0524	3.1408	3.1555	-0.0524	1.8189
3	1.5	1.8025	1.8189	-0.7195	-0.0524	-0.001	2.9489	3.1937	-0.001	1.8192



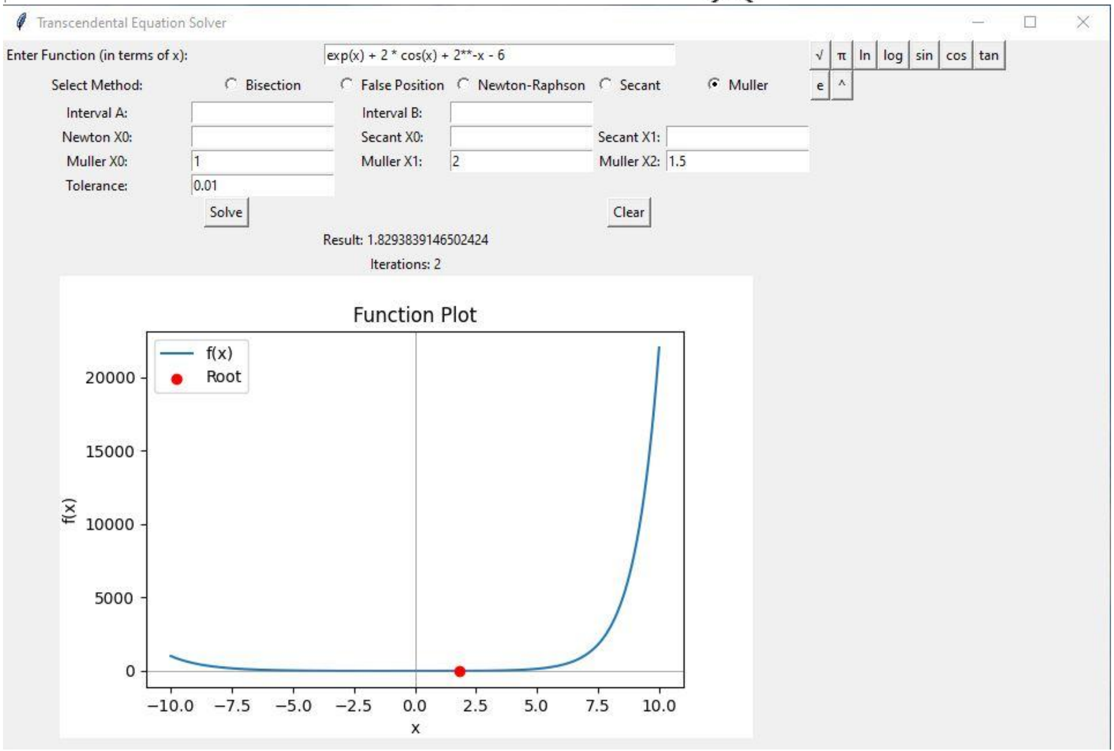
Question is verified from https://atozmath.com/

The Equations x^3 -2x^2 + ln(x) = 0 was solved using the Muller method in both sources, yielding the root 1.8191876520841619. The first source, atozmath.com, presents detailed steps at 3 iterations, while our application *Transcendental Equations Solver* provides the same result at 2 iterations with a graphical representation, highlighting its visual advantage.

- **Test Case 4**:

Approximate root of the equation $x - 0.2\sin\left(\sqrt{x}\right) - 0.8 = 0$ using Muller method is 0.9664 (After 3 iterations)

n	x_0	x_1	x_2	$f(x_0)$	$f(x_1)$	$f(x_2)$	a	b	c	x_3
1	0	1	0.5	-0.8	0.0317	-0.4299	0.1831	0.8317	-0.4299	0.9686
2	1	0.5	0.9686	0.0317	-0.4299	0.002	0.0461	0.9434	0.002	0.9664
3	0.5	0.9686	0.9664	-0.4299	0.002	0	0.0469	0.9436	0	0.9664



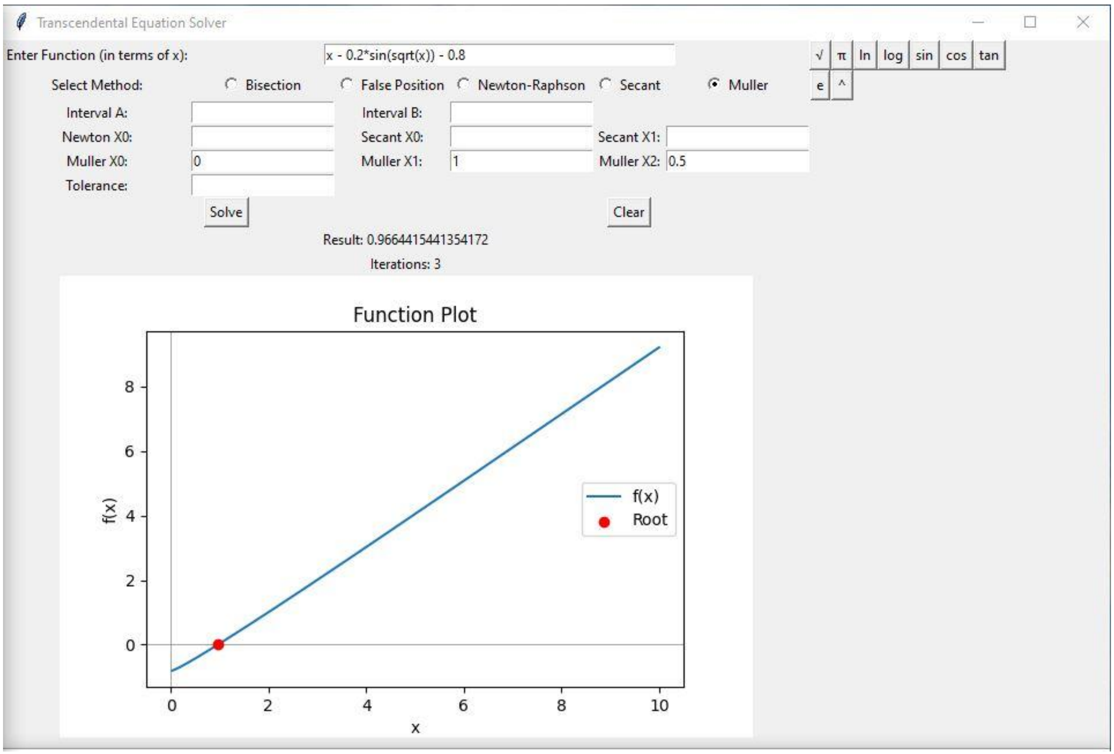
Question is verified from https://atozmath.com/

The Equations $e^x + 2\cos(x) + 2^{-x} - 6 = 0$ was solved using the Muller method in both sources, yielding the root 1.82938391465502121. The first source, atozmath.com, presents detailed steps at 3 iterations, while our application *Transcendental Equations Solver* provides the same result at 2 iterations with a graphical representation, highlighting its visual advantage.

- **Test Case 5**:

Approximate root of the equation $e^x + 2\cos(x) + \frac{1}{2^x} - 6 = 0$ using Muller method is 1.8294 (After 3 iterations)

n	x_0	x_1	x_2	$f(x_0)$	$f(x_1)$	$f(x_2)$	a	b	c	x_3
1	1	2	1.5	-1.7011	0.8068	-1.0233	2.3044	2.5079	-1.0233	1.8162
2	2	1.5	1.8162	0.8068	-1.0233	-0.0536	3.2265	4.0871	-0.0536	1.8292
3	1.5	1.8162	1.8292	-1.0233	-0.0536	-0.001	3.0033	4.0945	-0.001	1.8294



Question is verified from https://atozmath.com/

The Equations x - 0.2sin(sqrt(x)) - 0.8 = 0 was solved using the Muller method in both sources, yielding the root 0.9664415441354172 at 3 iterations. The first source, atozmath.com, presents detailed steps, while our application *Transcendental Equations Solver* provides the same result with a graphical representation, highlighting its visual advantage.

4.5. Secant Method Examples:

- **Test Case 1**:

3^{rd} iteration :

$x_2 = 2.4$ and $x_3 = 2.4444$

$f(x_2) = f(2.4) = -0.24$ and $f(x_3) = f(2.4444) = -0.0249$

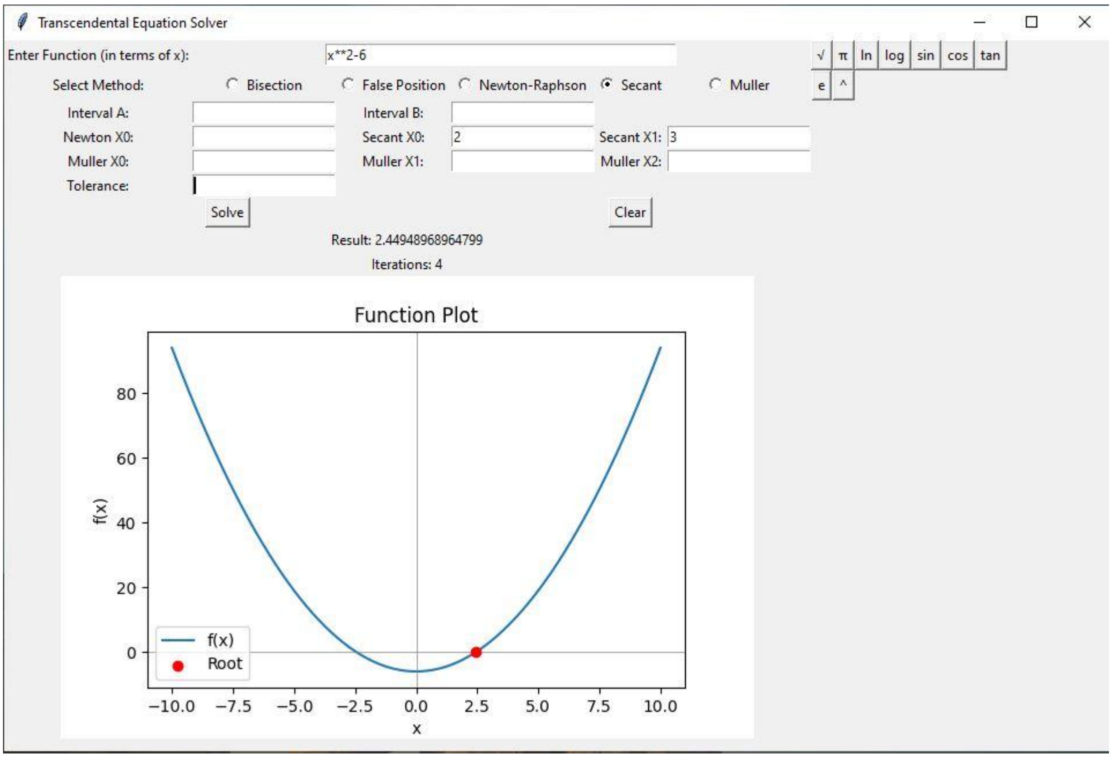
$$\therefore x_4 = x_2 - f(x_2) \cdot \frac{x_3 - x_2}{f(x_3) - f(x_2)}$$

$$x_4 = 2.4 - (-0.24) \cdot \frac{2.4444 - 2.4}{-0.0249 - (-0.24)}$$

$x_4 = 2.4495$

$\therefore f(x_4) = f(2.4495) = 2.4495^2 - 6 = 0.0001$

Approximate root of the equation $x^2 - 6 = 0$ using Secant method is 2.4495 (After 3 iterations)



Question is verified from https://atozmath.com/

The Equations $x^2-6=0$ was solved using the Secant method in both sources, yielding the root 2.44952.44952.4495. The first source, atozmath.com, presents detailed steps at 3 iterations, while your application *Transcendental Equations Solver* provides the same result at 6 iteratinos with a graphical representation, highlighting its visual advantage.In test case 1 our application takes 3 iterations extra.

- **Test Case 2**:

6^{th} iteration :

$x_5 = -0.8478$ and $x_6 = -0.8665$

$f(x_5) = f(-0.8478) = -0.0523$ and $f(x_6) = f(-0.8665) = 0.0031$

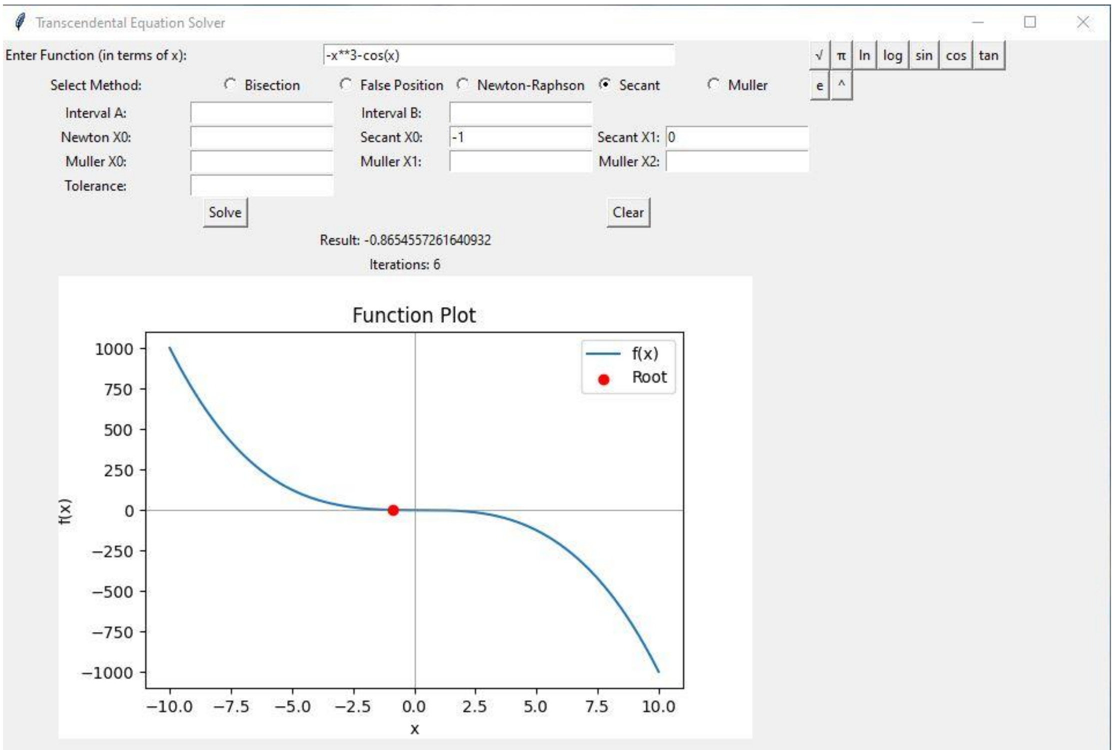
$$\therefore x_7 = x_5 - f(x_5) \cdot \frac{x_6 - x_5}{f(x_6) - f(x_5)}$$

$$x_7 = -0.8478 - (-0.0523) \cdot \frac{-0.8665 - (-0.8478)}{0.0031 - (-0.0523)}$$

$x_7 = -0.8655$

$$\therefore f(x_7) = f(-0.8655) = -(-0.8655)^3 - \cos(-0.8655) = 0.0001$$

Approximate root of the equation $-x^3 - \cos(x) = 0$ using Secant method is -0.8655 (After 6 iterations)



Question is verified from https://atozmath.com/

The Equations $-x^3 - \cos(x) = 0$ was solved using the Secant method in both sources, yielding the root -0.8654557261640932 after 6 iterations. The first source, atozmath.com, presents detailed steps, while your Python application *Transcendental Equations Solver* provides the same result with a graphical representation, highlighting its visual advantage.

- **Test Case 3**:

2^{nd} iteration :

$x_1 = 1$ and $x_2 = 0.9619$

$f(x_1) = f(1) = 0.0317$ and $f(x_2) = f(0.9619) = -0.0043$

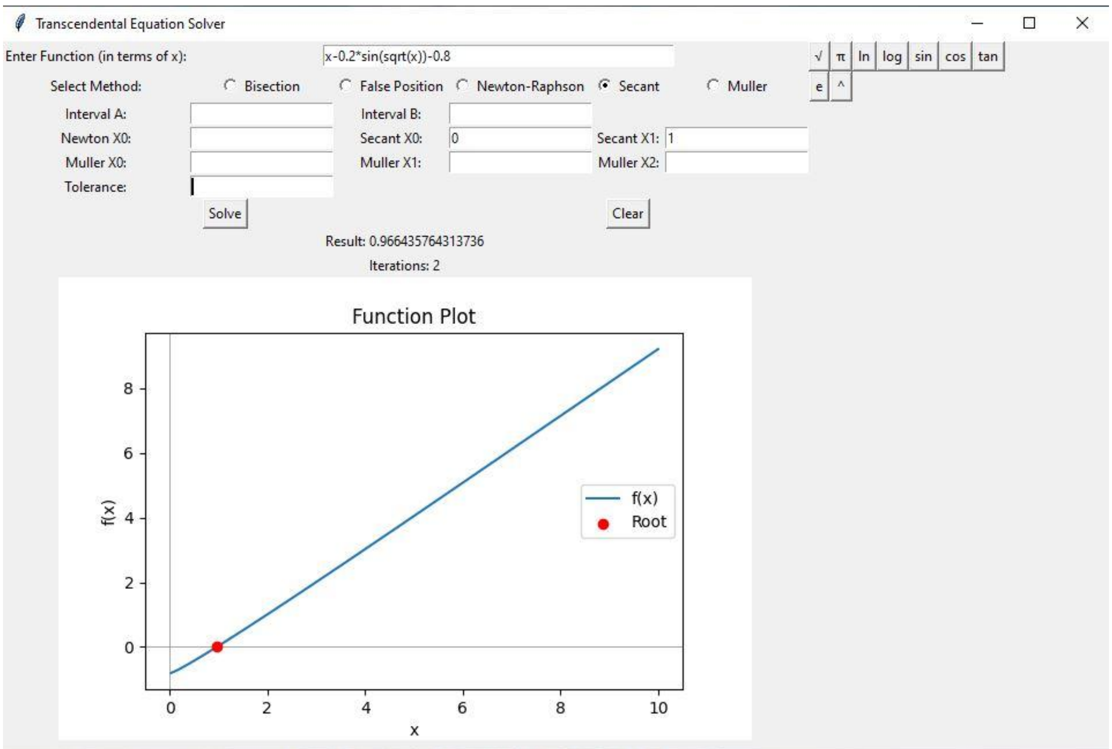
$$\therefore x_3 = x_1 - f(x_1) \cdot \frac{x_2 - x_1}{f(x_2) - f(x_1)}$$

$$x_3 = 1 - 0.0317 \cdot \frac{0.9619 - 1}{-0.0043 - 0.0317}$$

$x_3 = 0.9665$

$\therefore f(x_3) = f(0.9665) = 0.9665 - 0.2\sin(0.9831) - 0.8 = 0.0001$

Approximate root of the equation $x - 0.2\sin\left(\sqrt{x}\right) - 0.8 = 0$ using Secant method is 0.9665 (After 2 iterations)



Question is verified from https://atozmath.com/

The Equations x - 0.2sin(sqrt(x)) - 0.8 = 0 was solved using the Secant method in both sources, yielding the root 0.966435764313736 after 2 iterations. The first source, atozmath.com, presents detailed steps, while your Python application *Transcendental Equations Solver* provides the same result with a graphical representation, highlighting its visual advantage.

- **Test Case 4**:

5^{th} iteration :

$x_4 = 1.6076$ and $x_5 = 1.6004$

$f(x_4) = f(1.6076) = 0.0941$ and $f(x_5) = f(1.6004) = -0.0027$

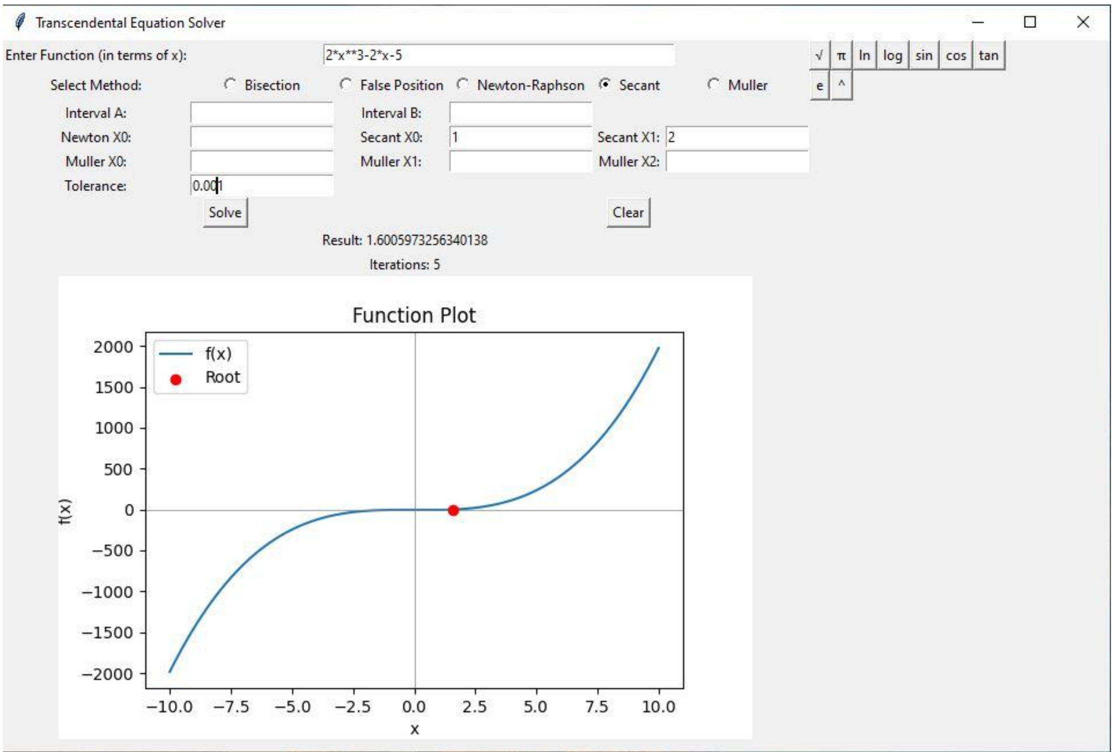
$$\therefore x_6 = x_4 - f(x_4) \cdot \frac{x_5 - x_4}{f(x_5) - f(x_4)}$$

$$x_6 = 1.6076 - 0.0941 \cdot \frac{1.6004 - 1.6076}{-0.0027 - 0.0941}$$

$x_6 = 1.6006$

$\therefore f(x_6) = f(1.6006) = 2 \cdot 1.6006^3 - 2 \cdot 1.6006 - 5 = 0$

Approximate root of the equation $2x^3 - 2x - 5 = 0$ using Secant method is 1.6006 (After 5 iterations)



Question is verified from https://atozmath.com/

The Equations $2x^3$-2x-5 was solved using the Secant method in both sources, yielding the root 1.6005973256 after 5 iterations. The first source, atozmath.com, presents detailed steps, while your Python application *Transcendental Equations Solver* provides the same result with a graphical representation, highlighting its visual advantage.

- **Test Case 5**:

4^{th} iteration :

$x_3 = 1.8081$ and $x_4 = 1.8323$

$f(x_3) = f(1.8081) = -0.0858$ and $f(x_4) = f(1.8323) = 0.012$

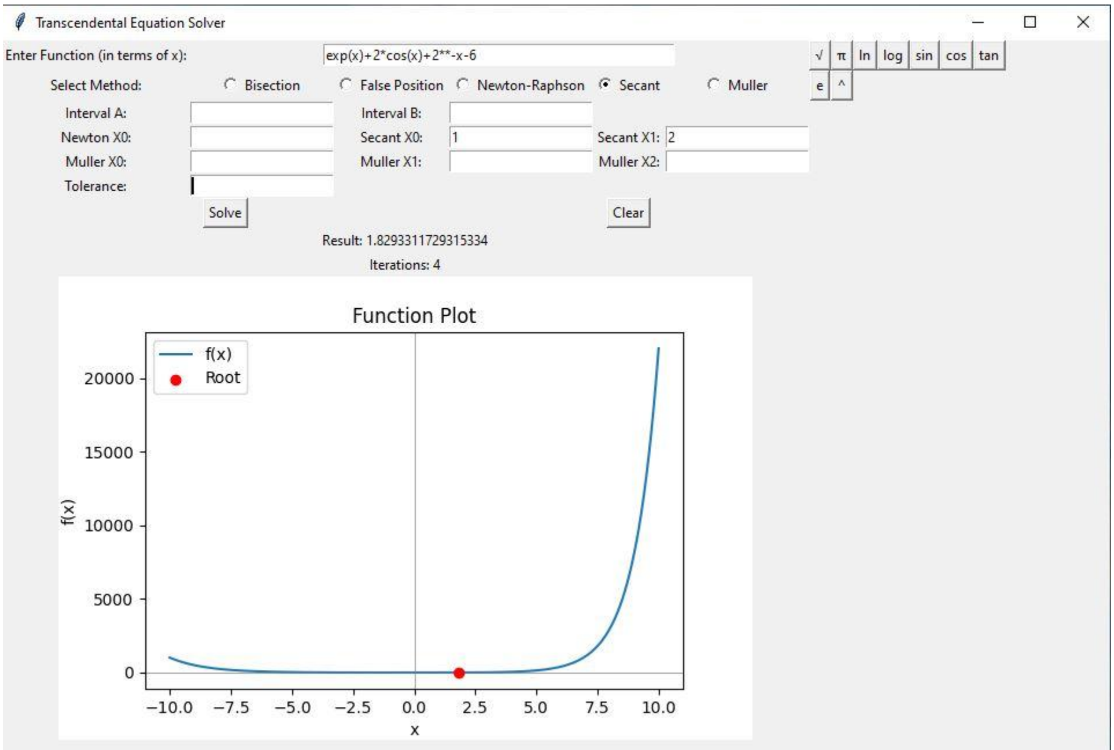
$$\therefore x_5 = x_3 - f(x_3) \cdot \frac{x_4 - x_3}{f(x_4) - f(x_3)}$$

$$x_5 = 1.8081 - (-0.0858) \cdot \frac{1.8323 - 1.8081}{0.012 - (-0.0858)}$$

$x_5 = 1.8293$

$$\therefore f(x_5) = f(1.8293) = e^{1.8293} + 2\cos(1.8293) + \frac{1}{2^{1.8293}} - 6 = -0.0003$$

Approximate root of the equation $e^x + 2\cos(x) + \frac{1}{2^x} - 6 = 0$ using Secant method is 1.8293 (After 4 iterations)



Question is verified from https://atozmath.com/

The Equations $e^x + 2\cos(x) + 2^{-x} - 6 = 0$ was solved using the Secant method in both sources, yielding the root 1.82933117293 after 4 iterations. The first source, atozmath.com, presents detailed steps, while your Python application *Transcendental Equations Solver* provides the same result with a graphical representation, highlighting its visual advantage.

5. Conclusion

The "Transcendental Equations Solver" project implemented and evaluated five numerical methods—Bisection, Regula Falsi, Newton-Raphson, Muller, and Secant—for solving transcendental Equation's. Each method showcased unique strengths and limitations in terms of convergence rates, accuracy, and computational efficiency. The solver provided accurate roots, iteration counts, and graphical plots, enabling a clear comparison of the methods.

The Bisection method, though slow, guaranteed root finding if the initial interval brackets the solution, offering reliability but lacking speed. Regula Falsi improved on Bisection with faster convergence in some cases but struggled when the root was near an interval endpoint. Newton-Raphson excelled in speed and accuracy with quadratic convergence but was sensitive to initial guesses. Muller, with its parabolic interpolation, handled complex roots effectively and showed rapid convergence, making it versatile for diverse functions. The Secant method achieved superlinear convergence with just two initial approximations but was prone to divergence with poorly chosen starting points.

In summary, each method proved effective under specific conditions, with the solver's graphical outputs validating results and iteration counts highlighting efficiency. The tool serves as a practical and educational resource for solving a variety of transcendental Equation's.

6. Recommendations

6.1. Expansion of Methods: Incorporate additional techniques like Fixed-Point Iteration, Brent’s Method, or hybrid algorithms to increase versatility and handle a wider range of Equation's.
6.2. Error Estimation: Add functionality to calculate iteration-wise errors or residuals for better accuracy assessment and convergence tracking.
6.3. Enhanced UI: Improve the GUI with dynamic visualizations, step-by-step tracking, input validation, and tutorials to guide users.
6.4. Complex Roots: Introduce specialized modes for handling complex roots and multiple solutions to expand the tool’s applicability in advanced fields.
6.5. Adaptive Algorithms: Implement adaptive methods to intelligently select the most efficient technique based on the function's characteristics.
6.6. Comprehensive Documentation: Provide detailed theory, user guides, and examples to make the tool accessible and educational.
6.7. Performance Optimization: Optimize the code for speed and efficiency to handle larger, more complex problems.

These enhancements would broaden the solver’s capabilities, making it a valuable resource for education, research, and practical applications in numerical analysis.

7. Appendices

7.1. Code

```
import tkinter as tk

from tkinter import messagebox

import numpy as np

import sympy as sp

import matplotlib.pyplot as plt

from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg


class EquationSolverApp:

    def __init__(self, master):

        self.master = master

        self.master.title("Transcendental Equation Solver")


        # Function input

        tk.Label(master, text="Enter Function (in terms of x):").grid(row=0, column=0)

        self.function_input = tk.Entry(master, width=50)  # column length

        self.function_input.grid(row=0, column=1, columnspan=5)


        # Additional buttons for special characters

        self.create_special_buttons(master)


        # Method selection

        self.method_var = tk.StringVar(value="Bisection")

        tk.Label(master, text="Select Method:").grid(row=1, column=0)

        tk.Radiobutton(master, text="Bisection", variable=self.method_var, value="Bisection").grid(row=1, column=1)

        tk.Radiobutton(master, text="False Position", variable=self.method_var, value="False Position").grid(row=1, column=2)

        tk.Radiobutton(master, text="Newton-Raphson", variable=self.method_var, value="Newton").grid(row=1, column=3)

        tk.Radiobutton(master, text="Secant", variable=self.method_var, value="Secant").grid(row=1, column=4)

        tk.Radiobutton(master, text="Muller", variable=self.method_var, value="Muller").grid(row=1, column=5)


        # Interval inputs

        tk.Label(master, text="Interval A:").grid(row=2, column=0)

        self.interval_a = tk.Entry(master)

        self.interval_a.grid(row=2, column=1)


        tk.Label(master, text="Interval B:").grid(row=2, column=2)

        self.interval_b = tk.Entry(master)

        self.interval_b.grid(row=2, column=3)


        # Newton and Secant specific inputs

        tk.Label(master, text="Newton X0:").grid(row=3, column=0)

        self.newton_x0 = tk.Entry(master)

        self.newton_x0.grid(row=3, column=1)


        tk.Label(master, text="Secant X0:").grid(row=3, column=2)

        self.secant_x0 = tk.Entry(master)

        self.secant_x0.grid(row=3, column=3)


        tk.Label(master, text="Secant X1:").grid(row=3, column=4)

        self.secant_x1 = tk.Entry(master)

        self.secant_x1.grid(row=3, column=5)
```

```
        # Muller inputs
        tk.Label(master, text="Muller X0:").grid(row=4, column=0)
        self.muller_x0 = tk.Entry(master)
        self.muller_x0.grid(row=4, column=1)

        tk.Label(master, text="Muller X1:").grid(row=4, column=2)
        self.muller_x1 = tk.Entry(master)
        self.muller_x1.grid(row=4, column=3)

        tk.Label(master, text="Muller X2:").grid(row=4, column=4)
        self.muller_x2 = tk.Entry(master)
        self.muller_x2.grid(row=4, column=5)

        # Tolerance input
        tk.Label(master, text="Tolerance:").grid(row=5, column=0)
        self.tolerance = tk.Entry(master)
        self.tolerance.grid(row=5, column=1)

        # Solve button
        self.solve_button = tk.Button(master, text="Solve", command=self.solve_equation)
        self.solve_button.grid(row=6, column=0, columnspan=3)

        # Clear button
        self.clear_button = tk.Button(master, text="Clear", command=self.clear_inputs)
        self.clear_button.grid(row=6, column=3, columnspan=3)

        # Result label
        self.result_label = tk.Label(master, text="Result: ")
        self.result_label.grid(row=7, column=0, columnspan=6)

        # Iteration label
        self.iteration_label = tk.Label(master, text="Iterations: ")
        self.iteration_label.grid(row=8, column=0, columnspan=6)

        # Matplotlib Figure
        self.figure = plt.Figure(figsize=(6, 4), dpi=100)
        self.canvas = FigureCanvasTkAgg(self.figure, master)
self.canvas.get_tk_widget().grid(row=9, column=0, columnspan=6)
        self.ax = self.figure.add_subplot(111)

    def create_special_buttons(self, master):
        """Create buttons for special characters."""
        # Button for square root
        sqrt_button = tk.Button(master, text="√", command=lambda: self.insert_character("sqrt("))
        sqrt_button.grid(row=0, column=6)

        # Button for pi
        pi_button = tk.Button(master, text="π", command=lambda: self.insert_character("pi"))
        pi_button.grid(row=0, column=7)

        # Buttons for logarithms
        ln_button = tk.Button(master, text="ln", command=lambda: self.insert_character("ln("))
        ln_button.grid(row=0, column=8)

        log_button = tk.Button(master, text="log", command=lambda: self.insert_character("log10("))
        log_button.grid(row=0, column=9)
```

```
        #button for 'sin', 'cos', 'tan'

        sin_button = tk.Button(master, text="sin", command=lambda: self.insert_character("sin("))

        sin_button.grid(row=0, column=10)


        cos_button = tk.Button(master, text="cos", command=lambda: self.insert_character("cos("))

        cos_button.grid(row=0, column=11)


        tan_button = tk.Button(master, text="tan", command=lambda: self.insert_character("tan("))

        tan_button.grid(row=0, column=12)


        e_button = tk.Button(master, text="e", command=lambda: self.insert_character("exp("))

        e_button.grid(row=1, column=6)


        pow_button = tk.Button(master, text="^", command=lambda: self.insert_character("**"))

        pow_button.grid(row=1, column=7)


    def insert_character(self, character):

        """Insert a character into the function input field."""

        current_text = self.function_input.get()

        self.function_input.delete(0, tk.END)

        self.function_input.insert(0, current_text + character)


    def solve_equation(self):

        function_str = self.function_input.get().strip()

        try:

            # Validate the function input

            if not function_str:

                raise ValueError("Please enter a valid function.")


            # Check for 'log' or 'ln' in the input and ensure positive intervals

            if "log" in function_str or "ln" in function_str:

                log_warning = (

                    "For log and ln functions, interval values must be positive."

                )


                if self.method_var.get() in ["Bisection", "False Position"]:

                    if not self.interval_a.get() or not self.interval_b.get():

                        raise ValueError("Please enter values for both Interval A and Interval B.")

                    if float(self.interval_a.get()) <= 0 or float(self.interval_b.get()) <= 0:

                        raise ValueError(log_warning)


                elif self.method_var.get() in ["Secant"]:

                    if not self.secant_x0.get() or not self.secant_x1.get():

                        raise ValueError("Please enter values for Secant X0 and Secant X1.")

                    if float(self.secant_x0.get()) <= 0 or float(self.secant_x1.get()) <= 0:

                        raise ValueError(log_warning)


                elif self.method_var.get() == "Muller":

                    if not self.muller_x0.get() or not self.muller_x1.get() or not self.muller_x2.get():

                        raise ValueError("Please enter values for Muller X0, X1, and X2.")

                    if (float(self.muller_x0.get()) <= 0 or
```

```
                float(self.muller_x1.get()) <= 0 or
                float(self.muller_x2.get()) <= 0):
            raise ValueError(log_warning)

    elif self.method_var.get() == "Newton":
        if not self.newton_x0.get():
            raise ValueError("Please enter a value for Newton's X0.")
        if float(self.newton_x0.get()) <= 0:
            raise ValueError(log_warning)

# Parse the function using sympy
x = sp.symbols('x')
e = sp.E  # Euler's number
function = sp.lambdify(x, sp.sympify(function_str), "numpy")

# Get tolerance
tol = float(self.tolerance.get()) if self.tolerance.get() else 1e-7

method = self.method_var.get()
result = None
iterations = 0

# Utility function to safely parse inputs with 'e'
def parse_input(value):
    if not value.strip():
        raise ValueError("Missing input.")
    return float(sp.sympify(value, locals={"e": e}))

# Execute the appropriate method
if method == "Bisection":
    a = parse_input(self.interval_a.get())
    b = parse_input(self.interval_b.get())
    result, iterations = self.bisection_method(function, a, b, tol)
elif method == "False Position":
    a = parse_input(self.interval_a.get())
    b = parse_input(self.interval_b.get())
    result, iterations = self.false_position_method(function, a, b, tol)
elif method == "Newton":
    x0 = parse_input(self.newton_x0.get())
    result, iterations = self.newton_method(function, x0, tol)
elif method == "Secant":
    x0 = parse_input(self.secant_x0.get())
    x1 = parse_input(self.secant_x1.get())
    result, iterations = self.secant_method(function, x0, x1, tol)
elif method == "Muller":
    x0 = parse_input(self.muller_x0.get())
    x1 = parse_input(self.muller_x1.get())
    x2 = parse_input(self.muller_x2.get())
    result, iterations = self.muller_method(function, x0, x1, x2, tol)

self.result_label.config(text=f"Result: {result}")

self.iteration_label.config(text=f"Iterations: {iterations}")
```

```
            # Plot the function
            self.plot_function(function, result)

        except ValueError as ve:
            messagebox.showerror("Input Error", str(ve))
        except Exception as e:
            messagebox.showerror("Error", f"An error occurred: {str(e)}")

    def clear_inputs(self):
        """Clear all input fields and results."""
        self.function_input.delete(0, tk.END)
        self.interval_a.delete(0, tk.END)
        self.interval_b.delete(0, tk.END)
        self.newton_x0.delete(0, tk.END)
        self.secant_x0.delete(0, tk.END)
        self.secant_x1.delete(0, tk.END)
        self.muller_x0.delete(0, tk.END)
        self.muller_x1.delete(0, tk.END)
        self.muller_x2.delete(0, tk.END)
        self.tolerance.delete(0, tk.END)

        self.result_label.config(text="Result: ")
self.iteration_label.config(text="Iterations: ")
        self.ax.clear()
        self.canvas.draw()

    def bisection_method(self, f, a, b, tol, max_iterations=1000):
        # Ensure that the function has opposite signs at the endpoints
        if f(a) * f(b) >= 0:
            raise ValueError("The function must have different signs at the endpoints A and B.")

        # Initialize iteration count
        iterations = 0

        while iterations < max_iterations:
            # Midpoint calculation
            c = (a + b) / 2

            # Check if the function value at c is sufficiently close to 0
            if abs(f(c)) < tol:  # If we found a root close enough
                return c, iterations

            # Determine which subinterval to choose based on the sign change
            if f(c) * f(a) < 0:
                b = c  # The root is in the left half
            else:
                a = c  # The root is in the right half

            iterations += 1  # Increment the iteration count

            # Exit if the interval is small enough
            if (b - a) / 2 < tol:
                break

        # Return the approximate root and the number of iterations
        return (a + b) / 2, iterations

    def false_position_method(self, f, a, b, tol, max_iterations=1000):
        # Check if the initial interval is valid
        if f(a) * f(b) >= 0:
```

```
            raise ValueError("The function must have different signs at the endpoints A and B.")

        # Initialize the iteration counter
        iterations = 0
        c = a  # Initial guess

        while iterations < max_iterations:
            # Calculate the false position
            c = (a * f(b) - b * f(a)) / (f(b) - f(a))

            # Check if we have found the root or convergence is achieved
            if abs(f(c)) < tol:
                break  # Root found or sufficiently close to root

            # Update the interval
            if f(c) * f(a) < 0:
                b = c  # The root is in the left half
            else:
                a = c  # The root is in the right half

            iterations += 1

            # Check if the difference between the interval ends is less than the tolerance
            if abs(b - a) < tol:
                break

        # If the method didn't converge in the maximum number of iterations
        if iterations >= max_iterations:
            raise ValueError("Maximum iterations reached. The method did not converge.")
        return c, iterations

    def newton_method(self, f, x0, tol, max_iterations=1000):
        iterations = 0
        while iterations < max_iterations:
            f_x0 = f(x0)
            f_prime_x0 = (f(x0 + tol) - f_x0) / tol  # Numerical derivative

            if f_prime_x0 == 0:
                raise ValueError("Derivative is zero. No solution found.")

            x1 = x0 - f_x0 / f_prime_x0
            iterations += 1

            if abs(x1 - x0) < tol:
                return x1, iterations

            x0 = x1

        raise ValueError(f"Newton method did not converge after {max_iterations} iterations.")

    def secant_method(self, f, x0, x1, tol, max_iterations=1000):
        iterations = 0

        while iterations < max_iterations:
            f_x0 = f(x0)
            f_x1 = f(x1)

            # Prevent division by zero if the function values are equal
            if f_x1 == f_x0:
```

```
            raise ValueError(f"Function values at x0 and x1 ({f_x0} and {f_x1}) are equal. No solution found.")

        # Calculate the next approximation using the secant method formula
        x2 = x1 - f_x1 * (x1 - x0) / (f_x1 - f_x0)

        iterations += 1

        # Stopping criterion: if the difference between successive approximations is less than tolerance
        if abs(x2 - x1) < tol:
            return x2, iterations

        # Update the guesses for the next iteration
        x0, x1 = x1, x2
    # If maximum iterations are reached without convergence, raise an error
    raise ValueError(f"Secant method did not converge after {max_iterations} iterations.")

def muller_method(self, f, x0, x1, x2, tol, max_iterations=1000):
    iterations = 0
    while True:
        # Calculate the function values
        f_x0 = f(x0)
        f_x1 = f(x1)
        f_x2 = f(x2)

        # Create the coefficients for the quadratic
        h0 = x1 - x0
        h1 = x2 - x1
        delta0 = (f_x1 - f_x0) / h0
        delta1 = (f_x2 - f_x1) / h1
        d = (delta1 - delta0) / (h1 + h0)

        # Calculate the root
        b = delta1 + h1 * d
        D = (b ** 2 - 4 * f_x2 * d) ** 0.5

        # Check for division by zero or small denominator
        if abs(D) < tol:
            raise ValueError("Small discriminant value. The method may not converge.")

        # Select the root (Note: handling two possible roots)
        x3 = x2 + ((-2 * f_x2) / (b + (D if b > 0 else -D)))

        # Convergence check
        if abs(x3 - x2) < tol:
            break

        # Avoid infinite loops if the method doesn't converge
        if iterations >= max_iterations:
            raise ValueError("Maximum iterations reached. The method did not converge.")

        x0, x1, x2 = x1, x2, x3
        iterations += 1

    return x3, iterations

def plot_function(self, f, root):
    """Plot the function and the root."""
    self.ax.clear()
    x_vals = np.linspace(-10, 10, 400)
    y_vals = f(x_vals)
```

```
        self.ax.plot(x_vals, y_vals, label='f(x)')
        self.ax.axhline(0, color='gray', lw=0.5)
        self.ax.axvline(0, color='gray', lw=0.5)
        self.ax.scatter(root, f(root), color='red',
zorder=5, label='Root')
        self.ax.legend()
        self.ax.set_title("Function Plot")
        self.ax.set_xlabel("x")
        self.ax.set_ylabel("f(x)")
        self.canvas.draw()


if __name__ == "__main__":
    root = tk.Tk()
    app = EquationSolverApp(root)
    root.mainloop()
```