****

**“Numerical Methods Project”**

****

[**GitHub Repository Link**](https://github.com/sahermuhamed1/Hybrid-Methods-in-Numerical-Methods)

**Department: Scientific Computing (SC)**

**Supervisor: Eng. Abdullah El-Ghamry**

**Project Title: Hybrid Methods in**

**Numerical System**

**Team Members:**

* **Saher Muhamed Muhamed (Section 2)**
* **Ziad Ehab (Section )**
* **Hassan (Section )**
* **Gallall Ayman (Section )**

**Report Content:**

* **Hybrid Methods Definition**
* **Bi-Section Method**
* **Newton Method**
* **Secant Method**
* **Newton Raphson and Bi-section Method**
* **Bisection-Trisection method**
* **Steffensen Method**
* **Pengasus method**
* **Anderson Bjorck Method**
* **Ridder's Method**
* **Illinois Method**
* **GUI Implementation Using StreamLit**

# Hybrid Methods Definition:

hybrid methods refer to algorithms that combine two or more different numerical techniques to solve a problem more efficiently or robustly. The goal of hybrid methods is to leverage the strengths of each individual method while mitigating their weaknesses. These methods are often used for root-finding, optimization, and solving differential equations.

### Characteristics of Hybrid Methods:

1. **Efficiency**: Hybrid methods often converge faster than individual methods by utilizing a combination of rapid convergence techniques with methods that provide better global convergence properties.
2. **Robustness**: They increase the likelihood of finding a solution by combining methods that are less likely to fail under different conditions.
3. **Adaptability**: Hybrid methods can switch between different techniques based on the behavior of the function or problem at hand.

### Examples of Hybrid Methods in Numerical Methods:

1. **Newton-Bisection Method**: Combines the Newton-Raphson method and the Bisection method. The Newton-Raphson method provides fast convergence when the initial guess is close to the root, while the Bisection method ensures global convergence and reliability. If the Newton-Raphson step does not converge, the method falls back to the Bisection step.
2. **Bisection-Trisection Method**: Utilizes the Bisection method for its reliability and the Trisection method for faster convergence. The Bisection method is used initially to narrow down the interval containing the root, and once the interval is sufficiently small, the Trisection method takes over to quickly converge to the root.
3. **Steffensen Method**: an iterative numerical technique used for finding the roots of a real-valued function. It is an improvement over the simple fixed-point iteration method and is designed to accelerate convergence.

### Benefits of Hybrid Methods:

* **Increased Convergence Speed**: By combining methods that have different convergence properties, hybrid methods can achieve faster convergence compared to using a single method.
* **Improved Reliability**: The fallback mechanism ensures that if one method fails or becomes inefficient, another method can take over to continue the process.
* **Enhanced Flexibility**: They can adapt to various types of functions and problem conditions, making them versatile tools in numerical analysis.

# Bisection Method:

The Bisection Method is a numerical technique used to find a root of a continuous function. It is a bracketing method, meaning it requires two initial guesses that bound (or bracket) the root. The method iteratively narrows the interval that contains the root by halving the interval and selecting the subinterval in which the function changes sign. It is simple, robust, and guarantees convergence if the initial interval is chosen correctly.

### Equation:

1. Compute the midpoint **X=a+b / 2**
2. Evaluate the function at the midpoint **f(c).**
3. Determine which subinterval contains the root:

* if **f(a) . f(c)< 0**, then the root lies between **[a,c]**
* if **f(c) . f(b)< 0**, then the root lies between **[a,c]**

1. Replace the interval [a,b] with the subinterval that contains the root.
2. Repeat the process until find the error that is less than tolerance.

### Example:

Let's find a root of the function f (x) = x3 - x - 2 using the Bisection Method.

1. **Initial Interval:** Choose a = 1 and b = 2:

f(1)=1^3 - 2 - 2 = - 2

f(2)=2^3 - 2 - 2= 4

*Since f(1) . f(2) < 0, the root lies in the interval [1, 2].*

1. **First Iteration**:

C = a+b / 2

f(1.5)=1.5^3 - 1.5 - 2 = - 0.125

*Since f(1) . f(1.5) < 0, the root lies in the interval [1, 1.5].*

1. **Seoond Iteration**:

a = 1 , b = 1.5

c = 1+1.5 /2 = 1.25

f(1.25) =1.25^3 - 1.25 - 2 = - 1.296875

*Since f(1). f(1.25) < 0,the root lies in the interval [1, 1.25].*

1. **Third Iteration**:

a = 1,b = 1.25

c = 1+1.25 / 2 = 1.125

f(1.125) = 1.125^3 - 1.125 -2 = - 1.248046875

*Since f(1) . f(1.125) < 0, the root lies in the interval [1, 1.125].*

1. **Continue this process until the interval [a, b] is sufficiently small**: Error> tolerance

### Convergence Criteria:

Error (E) is smaller than the Tolerance given

# Newton Method

**Newton Method**

The Newton-Raphson Method, often simply called the Newton Method, is a powerful numerical technique used for finding the roots of a real-valued function. It is an iterative method that starts with an initial guess and refines it successively to converge towards the root. The Newton Method is widely used due to its fast convergence rate, especially when the initial guess is close to the root.

**Equation**

Given a continuous function f(x) and an initial guess x0​, the Newton-Raphson iteration formula is:



where:

* xn+1​ is the next approximation of the root.
* xn is the current approximation of the root.
* f(xn) is the value of the function at xn​.
* f′(xn) is the derivative of the function evaluated at xnx\_nxn​.

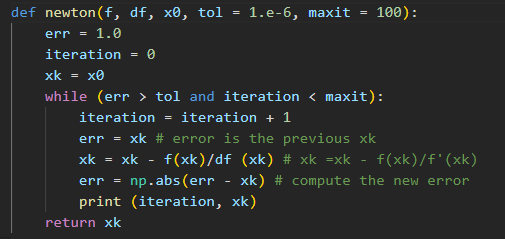
**Example of Usage**

Let's find a root of the function f(x)= x^2 - 4 using the Newton Method.

1. **Initial Guess**: Choose an initial guess, x0=2, which is close to the root.
2. **First Iteration**:
   * Evaluate **f(x0**) = f(2) = 2^2 - 4 =0.
   * Evaluate the derivative **f ′(x0)**= 2x .
   * Compute the next approximation using the Newton iteration formula:

**x1 = x0 – [f(x) / f `(x)]** = 2 – [0/2] = 2; The root is already found, and the iteration can terminate.

**Python Code:**

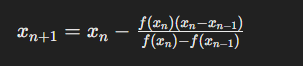
****

# Secant Method

The Secant Method is a numerical technique used for finding the roots of a real-valued function. It is an iterative method that does not require the derivative of the function, unlike the Newton Method. Instead, it approximates the derivative using finite differences. The Secant Method is similar to the Newton Method but replaces the derivative with a finite difference approximation. This makes it more flexible in cases where the derivative is difficult to compute.

**Equation**

Given two initial guesses x0​ and x1, the Secant Method iteration formula is:

****

where:

* xn+1​ is the next approximation of the root,
* xn​ are the current and previous approximations of the root, respectively,
* f(xn) and f(xn−1) are the values of the function at xn and xn−1​, respectively.

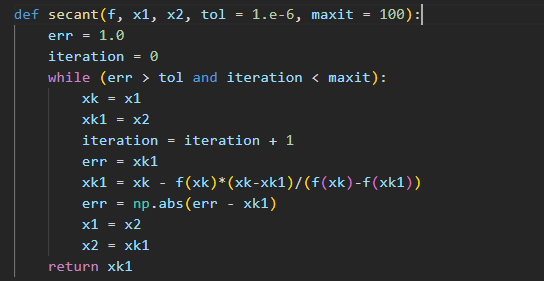
**Example of Usage**

Let's find a root of the function f(x)=x^3 - 3x + 1 using the Secant Method.

1. **Initial Guesses**: Choose two initial guesses, x0​=0 and x1 = 1.
2. **First Iteration**:
   * Evaluate **f(x0) = f(0) = 1** and **f(x1) = f(1)=−1.**
   * Compute the next approximation using the Secant Method formula:

**x2 = x1 – [f(x1)(x1−x0)] / [f(x1) – f(x0)]**

1. **Second Iteration**:
   * Evaluate **f(x1)=f(1) = -1** and **f(x2)=f(0.5)= 0.375.**
   * Compute the next approximation: **x3 = x2 – [f(x2)(x2−x1)] / [f(x2) – f(x1)]** Continue this process until convergence.

**Python Code:**

# Bi-section Tri-Section Method

The provided method is an implementation of the Bisection-Trisection Method for finding roots of a real-valued function within a given interval. This method combines aspects of both the Bisection Method and the Trisection Method to potentially improve convergence speed.

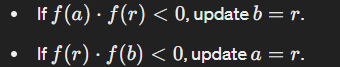
### Method Description

The Bisection-Trisection Method follows these steps:

1. **Initialization**: Define the function f(x), the initial interval [a,b], the tolerance tol, and the maximum number of iterations imax.
2. **Iterative Process**:
   * Start a loop that iterates from 1 to imax.
   * Perform a Bisection step:

Determine which subinterval contains the root:

* + if **f(a) . f(c)< 0**, then the root lies between **[a,c]**
  + if **f(c) . f(b)< 0**, then the root lies between **[a,c]**

1. **Perform a Trisection step :**

* Check if the absolute value of f(r) is less than the tolerance tol.
* If satisfied, return the iteration count i, the root r, and the updated intervals a and b.

1. • Repeat until the maximum number of iterations is reached.

### Summary:

In summary, the Bisection-Trisection Method provides a hybrid approach to root finding, leveraging the strengths of both the Bisection and Trisection methods to potentially achieve faster convergence.

### Python Code:

# Newton Bi-Section Method

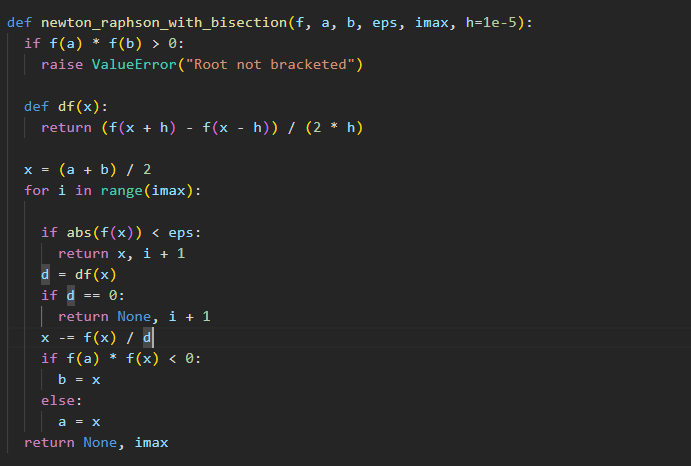
a hybrid approach combining the Newton-Raphson Method with the Bisection Method for finding the root of a real-valued function within a given interval. This method attempts to leverage the fast convergence of the Newton-Raphson Method while ensuring robustness by switching to the Bisection Method if needed.

### Method Description

The Newton Bi-Section Method follows these steps:

1. **Initialization**: Define the function **f(x)**, the initial interval **[a,b],** the desired level of accuracy **epsilon (ϵ)**, the maximum number of iterations **imax**, and an optional parameter h for the finite difference calculation.
2. **Check Initial Bracketing**: Ensure that the initial interval brackets a root (i.e., **f(a)⋅f(b)<0**). If not, raise an error.
3. **Define Derivative Function**: Approximate the derivative function **f′(x)** using finite differences.
4. **Iterative Process**:
   * Start a loop that iterates up to imax times.
   * Calculate the next approximation of the root using the Newton-Raphson Method: **xn+1=xn−f(xn)/f′(xn)** ​
   * If the derivative **f′(x**) becomes zero during the iteration, return None.
   * If the function value at the current approximation is within the desired level of accuracy **epsilon (ϵ)**, return the root and the iteration count.
   * If the function values at the **endpoints a and z** have opposite signs, update **b to x**; otherwise, update **a to x.**
   * Repeat until convergence or the maximum number of iterations is reached.

### Python Code:



# Steffensen Method

The Steffensen Method is an iterative numerical technique used for finding the root of a real-valued function. *It is similar to the Newton-Raphson Method but incorporates a self-accelerating mechanism to improve convergence speed*, especially in cases where the Newton-Raphson Method converges slowly or fails.

### Definition

The Steffensen Method iteratively improves an initial guess **x0**​ by approximating the derivative using finite differences and updating the guess based on a modified Newton-like iteration formula.

### Equation

Given a function **f(x)** and an initial guess **x0**​, the Steffensen Method iteration formula is:



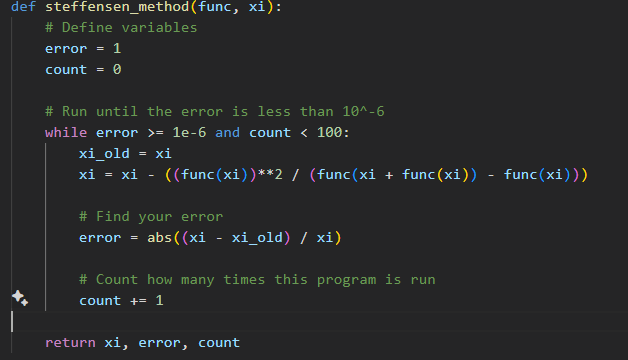
where:

* xn+1 is the next approximation of the root,
* xn is the current approximation of the root,
* f(xn)) is the value of the function at xn ​.

### How to Use the Equation

1. **Initialization**: Define the function f(x) and choose an initial guess x0​.
2. **Iteration**:
   * Start with the initial guess x0.
   * Compute f(xn) and f(xn+f(xn).
   * Update the guess using the Steffensen iteration formula.
   * Repeat until the desired level of accuracy is achieved or a maximum number of iterations is reached.

### Python Code:



# Pegasus Method

The Pegasus method is a numerical technique used for finding the root of a real-valued function. It is an iterative method that improves upon the Regula False (False Position) method by modifying the way it adjusts the interval endpoints to ensure faster convergence.

**Parameters:**

* `**f**`: The function for which we are finding the root.
* `**a, b**`: The interval \([a, b]\) where the root is sought.
* `**tol**`: The tolerance for convergence.
* `**max\_iter**`: The maximum number of iterations

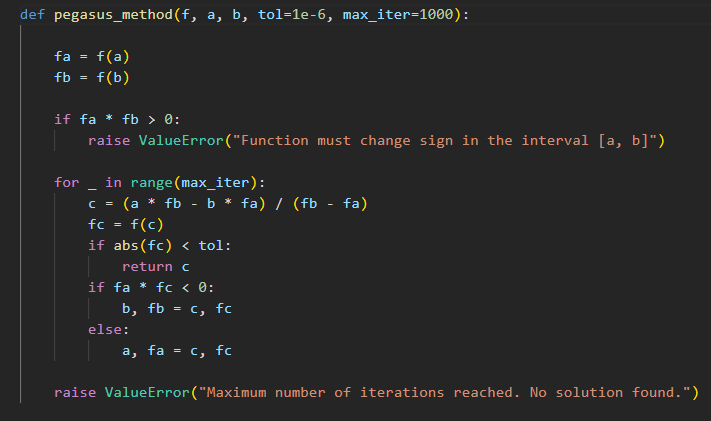
**Returns:**

- The approximated root of the function.

**Method:**

* + Compute the function values at the endpoints**, ( f(a) ) and ( f(b) ).**
  + Check if the function changes sign in the interval **([a, b]).** If not, raise an error.
  + Iterate up to `max\_iter` times:
    - Compute the new approximation for the root**, ( c ).**
    - Evaluate the function at the new point, **( f(c) ).**
    - Check if the function value at the new point is within the specified tolerance.
    - Adjust the interval endpoints based on the sign of the function at the new point.
  + If the method does not converge within the specified iterations, raise an error.

**Python Code:**

****

# Anderson-Björck Method

The Anderson-Björck method is an iterative numerical technique used for finding the root of a real-valued function. It is designed to improve the convergence of the Secant method by dynamically adjusting the interval endpoints to ensure faster and more reliable convergence.

**Parameters:**

- **`f`**: The function for which we are finding the root.

- **`x0, x1**`: Initial guesses for the root.

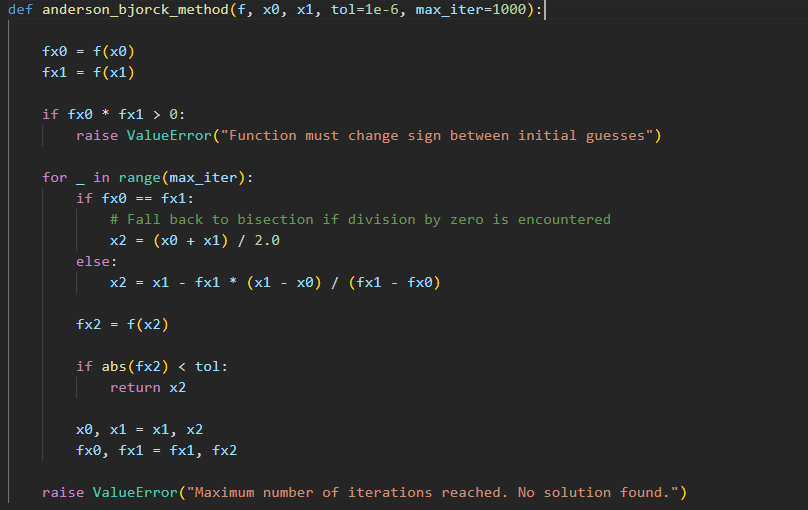
- **`tol**`: The tolerance for convergence.

- `**max\_iter**`: The maximum number of iterations.

**Method**:

* + Compute the function values at the initial guesses, ( f(x0)) and ( f(x1) ).
  + Check if the function changes sign between the initial guesses. If not, raise an error.
  + Iterate up to `max\_iter` times:
    - Compute the new approximation for the root, ( x2 ), using the Anderson-Björck formula.
    - Evaluate the function at the new point, ( f(x2) ).
    - Check if the function value at the new point is within the specified tolerance.
    - Update the guesses for the next iteration.
  + If the method does not converge within the specified iterations, raise an error.

**Python Code:**

****

# Ridder Method

Ridder’s method is a numerical technique used to find the roots of a real-valued function. It combines the principles of bisection and the secant method to achieve faster convergence. This method is particularly useful when the function changes sign over an interval **[a, b].**

**Method Description**

Given a continuous function ( f(x) ) and an interval ([a, b]) where ( f(a) ) and ( f(b) ) have opposite signs (i.e., ( f(a)f(b) < 0), Ridder’s method proceeds as follows:

1. **Initial Checks**: Ensure the function values at the endpoints have opposite signs.

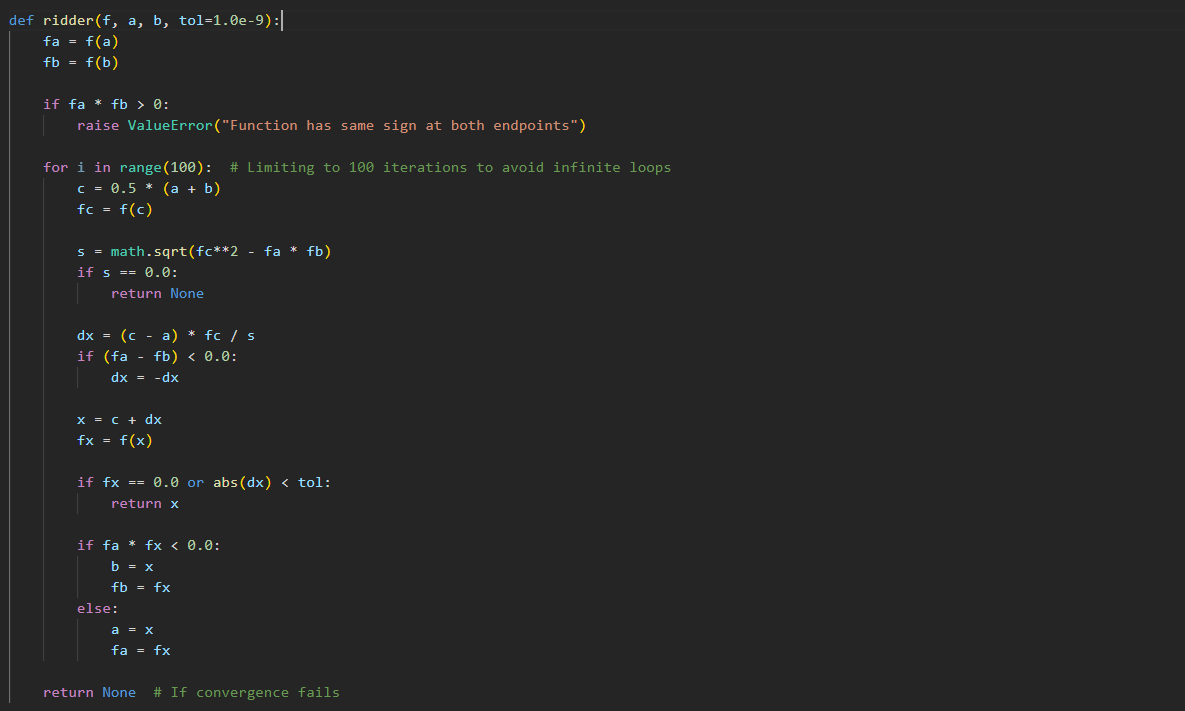
2. **Iterative Process**:

* Compute the midpoint ( c ) of the interval ([a, b]).
* Evaluate the function at the midpoint, ( f(c) ).
* Calculate a correction term using:
* [S = sqrt{f(c)^2 – f(a)f(b)}]
* Determine the new approximation ( x ) using: [X = c + (c – a) \frac{f(c)}{s}]
* Adjust the interval ([a, b]) based on the sign of ( f(x) ) to ensure ( f(a) ) and ( f(b) ) continue to have opposite signs.

3. **Convergence Check**: The method stops when the correction term is smaller than a specified tolerance or the function value at the new approximation is zero.

**Summary**

In summary, Ridder’s method is a valuable tool for root finding in numerical analysis, offering a balance between robustness and speed of convergence.

****

**Python Code:**

# Illinois Method

**Illinois** **Method**

The Illinois method is a numerical technique used to find roots of nonlinear equations of the form 𝑓(𝑥)=0. It is a hybrid method that combines the robustness of bracketing methods with enhancements to improve convergence speed. This report discusses the Illinois method, provides a detailed example, and includes Python code with output to demonstrate its practical application.

**Example of Usage**

Let's find a root of the function f(x)= x^2 - 4 using the Illinois Method.

1. **Initial Guess**: Choose an initial interval [𝑎, 𝑏] such that 𝑓(𝑎)⋅𝑓(𝑏) < 0.
2. **First Iteration**:
   * Compute the point of linear interpolation: 𝑐 = 𝑏 −𝑓(𝑏) \* (𝑏−𝑎) / 𝑓(𝑏) − 𝑓(𝑎)
   * Evaluate 𝑓(𝑐).
   * Update the interval:
     1. If 𝑓(𝑎)⋅𝑓(𝑐) < 0, set 𝑏=𝑐 and modify 𝑓(𝑎) by halving it for the next iteration.
     2. If 𝑓(𝑏)⋅𝑓(𝑐) < 0, set 𝑎=𝑐 and modify 𝑓(𝑏) by halving it for the next iteration.

**Advantages and Limitations**

* **Advantages**:
  + Enhanced Convergence.
  + Combines Robustness of Bracketing Methods with Efficiency of Open Methods.
  + Simplicity of Implementation.
  + Suitable for Continuous Functions.
* **Disadvantages**:
  + Potential for Division by Zero
  + Potential for Overshooting

In summary, The Illinois method enhances the Regula Falsi method by dynamically adjusting the function values at fixed endpoints, resulting in faster convergence. This hybrid approach maintains the robustness of bracketing methods while addressing their convergence issues. The provided Python implementation and example demonstrate its effectiveness in finding the roots of nonlinear equations efficiently.

**Example**

Consider finding the root of the function 𝑓(𝑥)=𝑥2−4. The exact roots are 𝑥=±2. We will use the Illinois method to find the positive root starting with the interval [1,3].

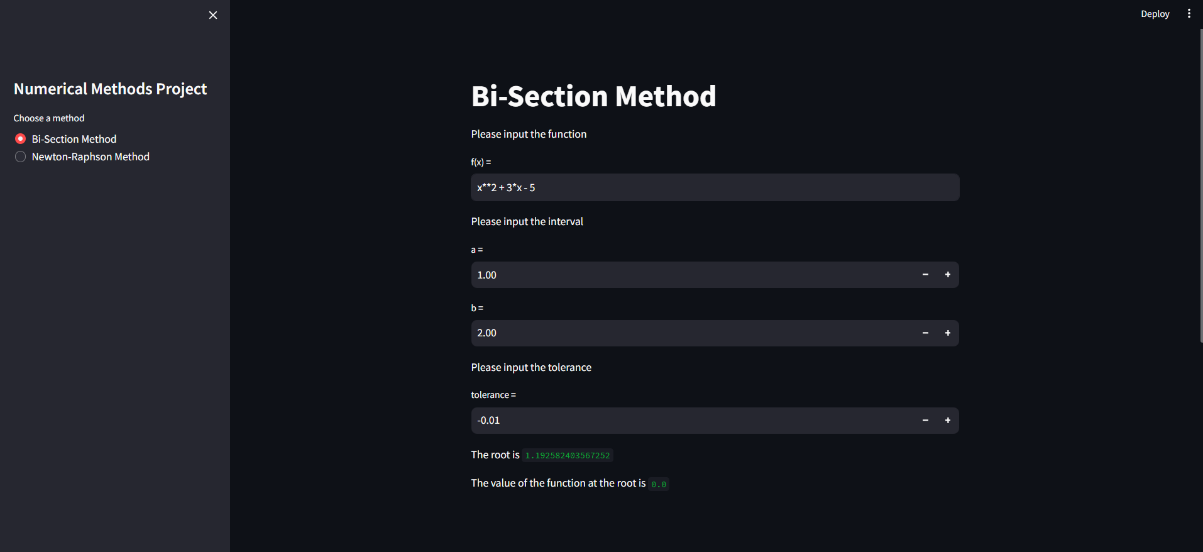
# GUI Implementation Using Streamlit

Streamlit is a Python library that simplifies the process of building interactive web applications for data science and machine learning projects. With Streamlit, you can create powerful web apps with minimal code, making it an ideal choice for showcasing and deploying your hybrid algorithms.

To implement a web app GUI using Streamlit for your hybrid algorithms, follow these steps:

1. **Install Streamlit**: Begin by installing Streamlit using pip, if you haven't already. You can install it by running pip install streamlit in your terminal or command prompt.
2. **Import Libraries**: In your Python script, import the necessary libraries including Streamlit and any other libraries required for your algorithms and visualization.
3. **Define Functions**: Define functions for each of your hybrid algorithms. These functions should take input parameters such as the function to find the root of, initial guesses, tolerance, maximum iterations, etc., and return the result (root), error, and/or iteration count.
4. **Create the Streamlit App**: Write the Streamlit app code. Start by using Streamlit's st.title, st.sidebar, and other layout elements to structure your app. Use Streamlit widgets such as text inputs, sliders, and buttons to allow users to input parameters for the algorithms.
5. **Invoke Algorithms**: Inside the Streamlit app, invoke your hybrid algorithm functions with the input parameters provided by the user. Display the results (root, error, iteration count) using Streamlit's text and visualization components.
6. **Run the App**: Once you've written your Streamlit app code, save the script and run it using the command streamlit run your\_script.py. This command will start a local server and open your web app in a browser window.
7. **Interact with the App**: Users can interact with the web app by adjusting input parameters, clicking buttons, and viewing the results displayed on the web page. Streamlit automatically updates the app in real-time as users interact with it.
8. **Deploy the App**: Once you're satisfied with your web app, you can deploy it to a web server or cloud platform for others to access. Streamlit offers various deployment options including Streamlit Sharing, Heroku, AWS, and more.

By following these steps, you can create a user-friendly web app GUI using Streamlit to showcase and utilize your hybrid algorithms for root-finding problems. Streamlit's simplicity and flexibility make it an excellent choice for quickly building and deploying data science web applications.

**Implemented By: Saher Muhamed**

Thank You

By Eng. Saher Muhamed