PHYS 361: Lab 6, Numerical Methods of Root Finding

Introduction

The Bisection Method

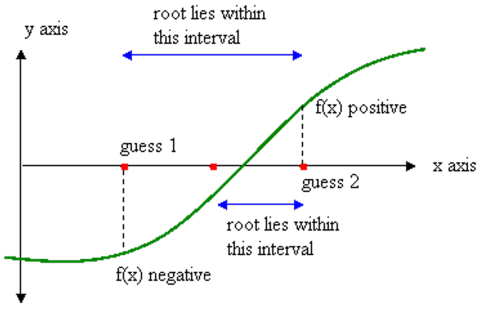

Figure 1: Bisection Method

The bisection method allows you to find the root of a known function, $f(x)$, based on the idea that *most* functions change sign or switch from being negative or positive on either side of the root. You can iterate or narrow in on the actual root with the following steps:

1. Evaluate the function at two initial values, x_l and x_u, that bound the root.

2. Find the mid-point of these two values, x_r. This midpoint is your current best guess for the root.
$$x_r = \frac{x_l + x_u}{2}. \tag{1}$$

3. Evaluate the function at the mid-point, x_r, and figure out whether x_l and x_r bound the root or x_r and x_u bound the root. ($f(x_l)f(x_r) < 0$ or $f(x_r)f(x_u) < 0$).

4. Repeat this process until the change in your solution or approximate error is very small,
$$|\epsilon_a| = \left| \frac{x_r^{new} - x_r^{old}}{x_r^{new}} \right|. \tag{2}$$
When ϵ_a is small (say less than 10^{-6}), x_r is very close to the actual root. You can determine how close by determining how small you want ϵ_a to be before you stop the code. You can check your solution by putting x_r back into your original function, $f(x)$, and seeing if it is equal to zero.

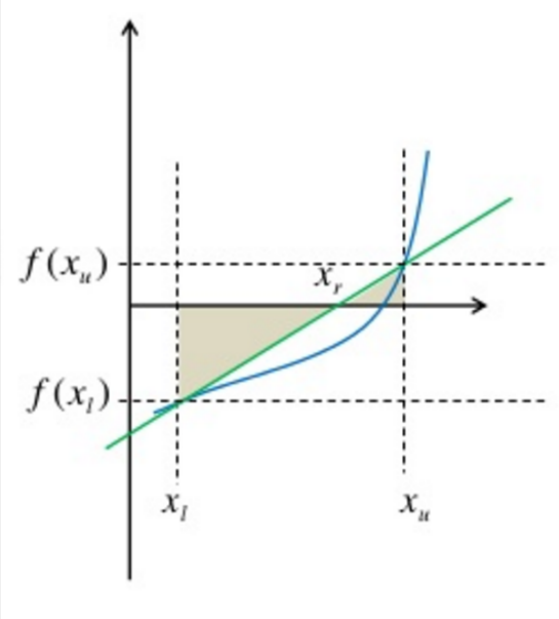

Figure 2: False Position Method

The False Position or Regular Falsi Method

The False Position Method is similar to the Bisection Method but uses a little geometry to choose a better value of the root rather than just picking a point halfway between the initial guesses. For this reason, it *usually* finds the root faster.

Note the green line that connects $f(x_l)$ and $f(x_u)$. The actual root (where the blue line is zero) is closer to where that line crosses the x-axis than the midpoint between x_l and x_u, for example.

Two triangles are formed with the x-axis when you draw a line that connects $f(x_l)$ and $f(x_u)$. The brown triangles are "similar triangles". That means the ratio or their heights is equal to the ratio of their widths or,

$$\frac{-f(x_l)}{f(x_u)} = \frac{x_u - x_r}{x_r - x_l} \tag{3}$$

You can rearrange the above equation to get,

$$x_r = x_u - \frac{f(x_u)(x_l - x_u)}{f(x_l) - f(x_u)} \tag{4}$$

You can iterate or narrow in on the actual root with the following steps:

1. Evaluate the function at two initial values, x_l and x_u, that bound the root.
2. Find the new x_r using the equation above.

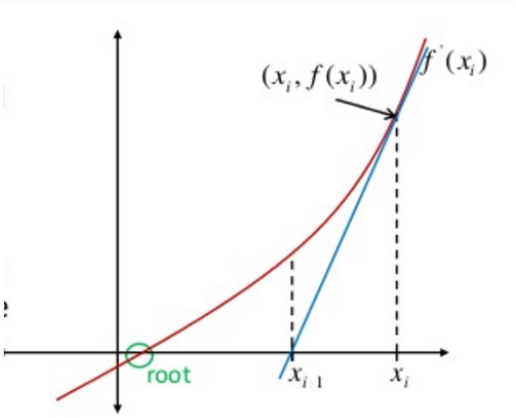

Figure 3: Newton Raphson Method

3. Evaluate the function at the mid-point, x_r, and figure out whether x_l and x_r bound the root or x_r and x_u bound the root. ($f(x_l)f(x_r) < 0$ or $f(x_r)f(x_u) < 0$).

4. Repeat this process until the change in your solution or approximate error is very small,

$$|\epsilon_a| = \left| \frac{x_r^{new} - x_r^{old}}{x_r^{new}} \right|. \tag{5}$$

Newton Raphson Method

The previous two methods are called "closed" methods because they are based on two values converging on the root from either side. "Open" methods require only one initial guess to find the root. (This is why many built-in algorithms like fzero only require one initial value.)

This method uses the first few terms in the Taylor Expansion to derive the following new guess for the root:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \tag{6}$$

Here x_{i+1} is the new guess for the root. It is calculated from your old guess, x_i, and the function and its derivative evaluated at the old guess, $f(x_i)$ and $f'(x_i)$. If you look at the image above. You are essentially using the derivative (or slope) of the line at x_i to draw a line that is tangent to the curve. The point where that line crosses the x-axis will *usually* converge towards the actual root of the function.

Again, we can continue to replace our "old" guess for the root with the new one, until the difference between the subsequent guesses or the approximate error is very small,

$$|\epsilon_a| = \left| \frac{x_{i+1} - x_i}{x_i} \right|. \tag{7}$$

The Problems!

You should upload a single script file for each of the question prompts below. **Please include your name and the problem number in the name of the script file.**

1. Determining the natural logarithm of a number p ($ln(p)$) is the same as finding a solution to the equation $f(x) = e^x - p = 0$. Write a MATLAB program that determines the natural logarithm of any number by solving this equation using the bisection method. The program should include the following features:

   - The upper and lower limit bounds, x_l and x_u, should be $x_l = e^0$ and $x_u = p$ if $p > e^1$ OR $x_l = -1/p$ and $x_u = e^0$ if $p < e^1$.
   - The iterations should stop when the approximate error is smaller than 1×10^{-4}.
   - The number of iterations should be limited to 100. If a solution is not obtained in 100 iterations, the function should stop and display an error message.
   - If zero or a negative number is entered for p, the program should stop and display an error message.

   You can download the bisection method code from the modules this week to help you get started. Check the program by evaluating the natural logarithm of (a) 510, (b) 1.35, (c) 1, and (d) -7.

2. Download the bisection method code used to solve the falling mass problem in the lectures this week. Save it with a new filename and modify it so that it solves a nonlinear equation $f(x) = 0$ using the false position (Regula Falsi) method. In the code:

   - Check whether the bounds are on the opposite sides of the solution. If not, the program should display an error message and ask the user for new bounds.
   - Loops should be limited to 100 iterations. If a solution is not obtained in that many iterations, the program should display an appropriate error message.

   Use the code to solve the same problem presented in class with the falling mass.

3. Steffensen's method is another way to find the numerical solution to a nonlinear equation (root finding). It is similar to the Newton-Raphson method, but it does not require the derivative of the function. The solution starts by choosing a point, x_i, near the solution, as the first estimate of the solution. The next estimate, x_{i+1} is calculated by:

$$x_{i+1} = x_i - \frac{f(x_i)^2}{f(x_i + f(x_i)) - f(x_i)} \tag{8}$$

   Write a `MATLAB` user-defined function that uses Steffensen's method (start by modifying the Newton-Raphson code we wrote in class). The input arguments should be

the function and the initial guess and the output argument should be the solution. The iterations should stop when the relative error is less than 10^{-6}. The number of iterations should be limited to 100 (to avoid an infinite loop). If a solution is not obtained in 100 iterations, the program should stop and display an error message. Test the function by using it to solve the solution to $x = 2e^{-x}$. You should check the answer either using the function or by using `MATLAB`'s built-in function `fzero`.