FINDING ROOTS OF EQUATIONS

Root finding is a skill that is particularly well suited for computer programming. Unless the roots of an equation are easy to find, iterative methods that can evaluate a function hundreds, thousands, or millions of times will be required. Another was to say "root finding" is to say "what value of *x* for a function will give an answer of zero". Also, root finding can be thought of as finding where two functions intersect.

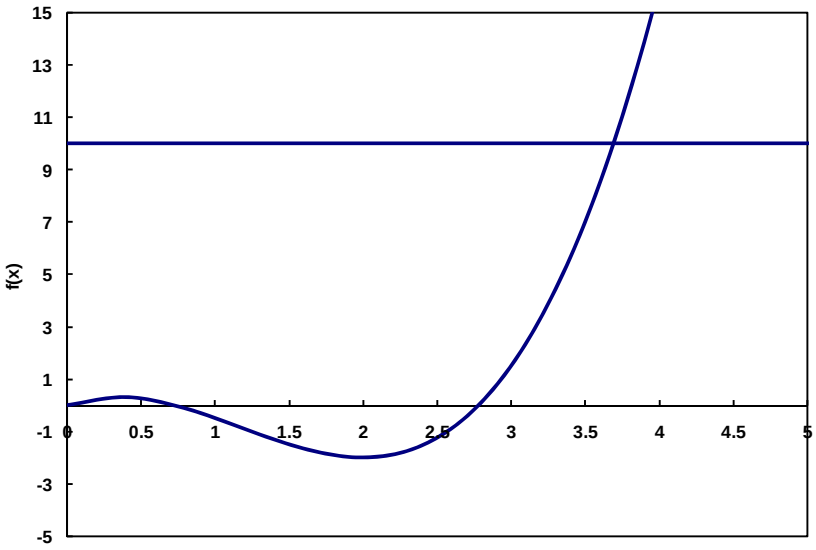
As an example throughout the root finding tutorial, the following equation will be used:

$$y = x^3 - 3.5x^2 + 2x = 10$$

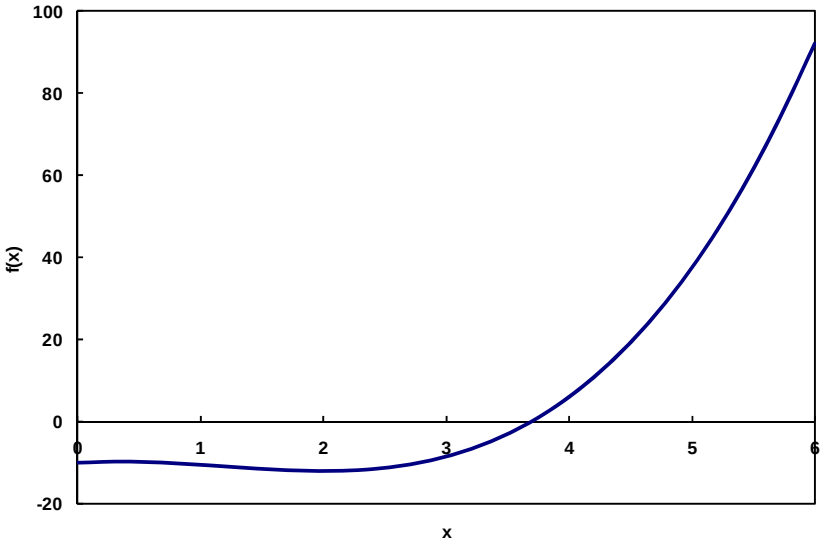
The question that will be answered is what value of *x* will yield a value of *y* equal to 10. The equation above can be rewritten and a value of *x* that makes *f(x)* equal to zero will be sought.

$$f(x) = x^3 - 3.5x^2 + 2x - 10 = 0$$

One way to find roots is to make plots. For the first equation, the intersection of $y = 10$ and $y = x^3 - 3.5x^2 + 2x$ can be found:



Alternatively, the root can be found by finding where *f(x)* crosses the zero line.



The problem with using plots is that MATLAB cannot look at the plot and point its finger at the answer like a human would. If the user is going to base another subsequent calculation on this root, a number value that MATLAB understands will be required to move on instead of a plot that MATLAB has difficulty in translating.

There are two classes of root finding numerical methods that will be covered in this tutorial. They are bracketing methods and open methods. In general, no one method is guaranteed to find all of the roots of an equation. Often times multiple attempts with multiple methods is needed to obtain the roots. Additionally, the exact root is rarely found. Instead, the MATLAB user will allow for some error in the calculation and the root will be found with that error in mind.

COMMON TROUBLE

There are several scenarios which can cause problems for root finding algorithms.

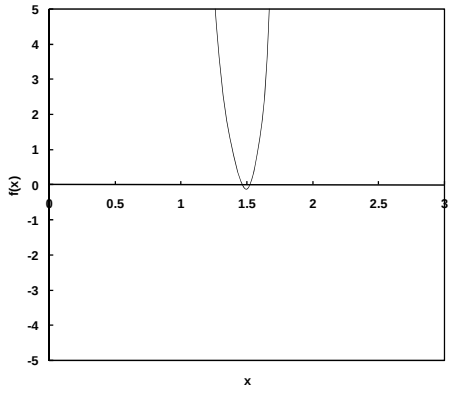
Multiple Roots

It can be difficult for most techniques to identify all roots of an equation. Most can be modified to search for multiple roots but it is impossible to assure that all roots have been found without doing infinite attempts. Luckily, there are usually physical limits in engineering problems, which lead to only one root having physical significance. In chemistry, a scenario can arise where the concentration of some species is the solution of a quadratic equation. Usually, there will be a positive and negative root. The negative root cannot be the correct answer, so the positive root is always chosen. In fact, it is possible to avoid even looking for the negative root. Sometimes complex roots are found for values that cannot be imaginary. In that case, the imaginary roots are left out.

Close Roots

It is possible to have two roots that are so close together that the numerical method cannot distinguish them. The error tolerance may be too large to find both roots. Luckily, the user has kept the tolerance sufficiently small that it may not matter which of the two roots is actually being found. Either one may be good enough or the fact that two exist may not matter.

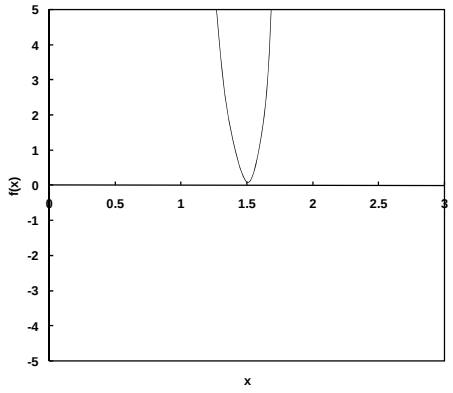
The plot below shows a line with two roots very close and on either side of 1.5.



Not Quite a Root

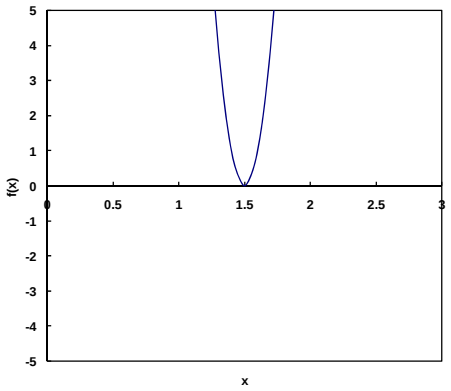
Some functions approach zero and get very close without actually reaching zero. If the difference between the function and zero is smaller than the tolerance, then the numerical method may incorrectly identify the minimum as a root.

The plot below shows a line that almost has a root at 1.5. It does not and a large error tolerance would incorrectly identify 1.5 as a root.



Double Roots

If an equation has a double root, many numerical methods have difficulty finding the root. In this case, the sign of the value does not change before and after the root.



If for some reason any of these problems are suspected, one can plot the function to assure that these phenomena do not exist or lower the error tolerance to a point that these issues are not problems, but the algorithm does not take forever to solve.

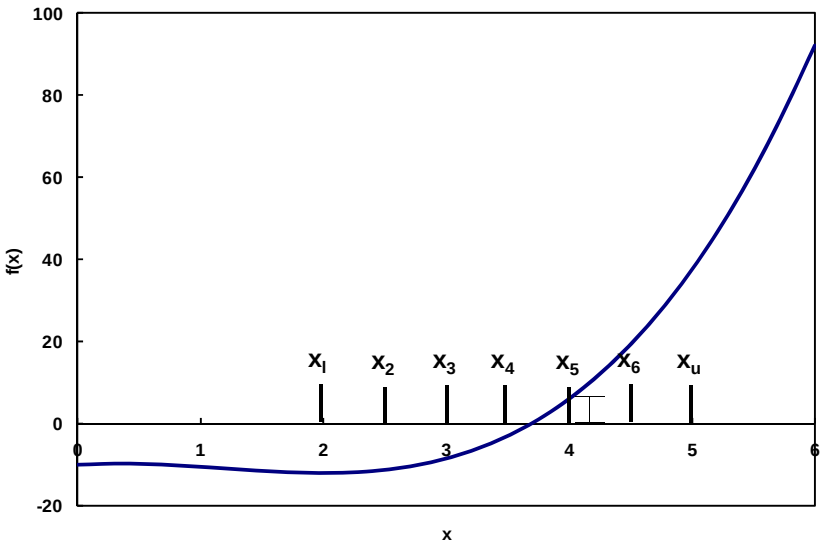
BRACKETING METHODS

The first class of numerical methods is known as bracketing methods. They work by choosing two values of x, in the above case, that bracket the root. Then different methods are used to zero in on the actual root. Usually a tolerance is specified, so that the exact root is not found. The value that is closer to the root than the tolerance is kept.

Incremental Search

An *incremental search* is carried out by evaluating the function at the lower bound, then the function is evaluated at a value one step higher than the lower bound and then another and then another, etc. At every calculation, the error between the function evaluation and zero is calculated. The lowest error and corresponding x value are kept until the upper bound has been reached. At this point, the x value with the lowest error can be kept and reported as the root. Alternatively, this root can be compared to the allowed error tolerance. If is too high, the incremental search can be carried out again with a smaller step size. This procedure can be repeated until a root with an error that is less than the tolerance is found.

The plot below shows how the *incremental search* would work. The lower bound is 2, the upper bound is 5, and the step size is 0.5. The function is evaluated at x_l, x_1, …,x_u. The smallest error shown for x_5 could be compared to a tolerance. If the error is too large, then a smaller step size could be specified and the procedure repeated. If the error is less than the tolerance, then x_5 is the root.



The computer code to carry this out is similar to the integration code considered in lab. In general, *incremental search* is the least efficient method, but is guaranteed to find a root. Modified properly it can identify multiple roots. It does have difficulty with not-quite-a-roots and if the two roots are closer than the step size, it will have difficulty finding close roots. Double roots are not a problem. Usually, the bracket can be chosen to find only physically possible roots.

Bisection Method

The bisection method starts by picking an upper and lower bound that bracket the root. An initial guess for the root (x_{mid1}) is taken as the midpoint between the upper bound and the lower bound. The function is evaluated at this point and the error is compared the tolerance. If the error is too large, then either the upper bound (x_u) or the lower bound (x_l) is reset to equal the midpoint. Then the midpoint of the new bracket (x_{mid2}) is found and the procedure is repeated until a root meets the error tolerance.

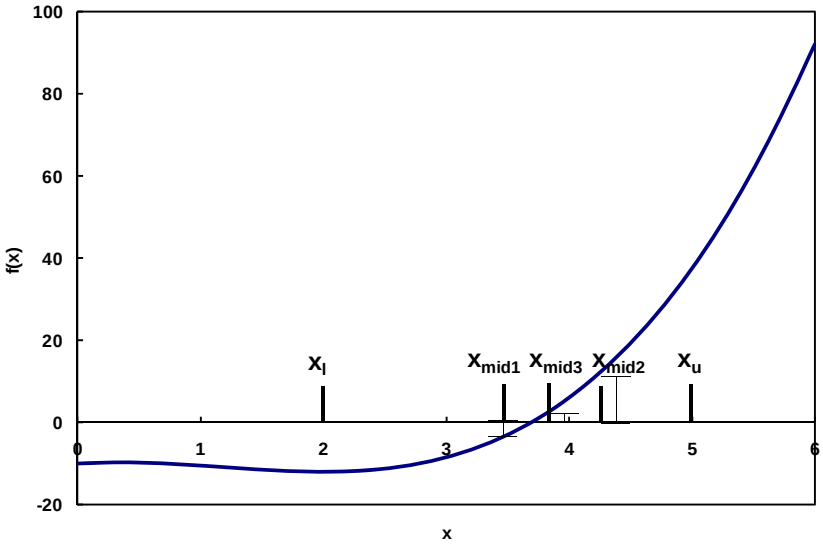
The not-so-obvious step is to determine which bound needs to be reset. The rule of thumb is that if the function evaluated at the lower bound times the function evaluated at the midpoint is less than zero, then the upper bound should be replaced by the midpoint. If the product is greater than zero, then the lower bound should be replaced.

If $f(x_l)f(x_{mid}) < 0$, then replace x_u with x_{mid}.

If $f(x_l)f(x_{mid}) > 0$, then replace x_l with x_{mid}.

If the product is less than zero then x_l and x_{mid} are on opposite sides of the zero line, so they must bracket the root. If the product is greater than zero, then they are on the same side of the zero line, so they do not bracket the root.

In the figure below, the midpoint of the upper and lower bound is x_{mid1}. If the error is not small enough, the lower bound is reset to x_{mid1}. The new midpoint is x_{mid2}. The error actually increases, so the upper bound is reset to x_{mid2}. The new midpoint, x_{mid3}, is then evaluated in $f(x)$. This continues until one of the midpoints meets the tolerance and that value is the root.



The major issue that this method has is that it cannot handle double roots at all. At a double root, the sign of the function does not change, so the *if* statements above do not work.

Pseudo code for Bisection

1) Input the upper bound, lower bound, and tolerance
2) Calculate the midpoint
3) Evaluate the function at the midpoint
4) Compare the value to the tolerance
    a. If less than the tolerance, report as the root
    b. If greater than the tolerance, reassign upper or lower bound
        i. If f(mid)*f(low) < 0, upper bound is equal to midpoint
        ii. If f(mid)*f(low)>0, lower bound is equal to midpoint
5) Repeat

MATLAB Code for Bisection

```
function [root]=bisection(funcname,xl,xu,tol)
fr=tol+1; % initializes the error for the while loop
while abs(fr)>tol % needs the absolute value of the error in case it is negative
        r=(xu-xl)/2+xl;
        [fr]=feval(funcname,r);
        [fl]=feval(funcname,xl);
```

```
        if fr*fl<0
                xu=r;
        else
                xl=r;
        end
end
root=r;
```

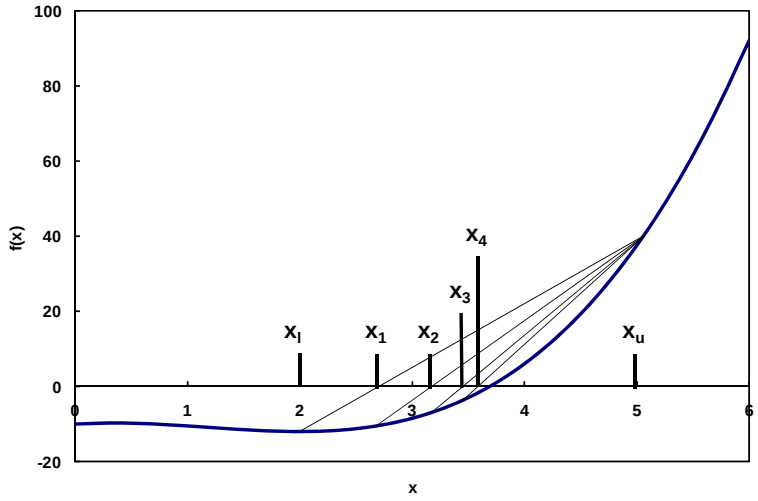
False Position Method

The false position is similar to bisection except in how it calculates the guess. Instead of using the midpoint, false position calculates the equation of a line connecting the two points on the function corresponding to the upper and lower bound. Wherever this line intersects the origin is the guess for the root. The error of this root is calculated and if it is not sufficiently close it will reset either the upper or lower bound in the exact same fashion as the bisection method and repeat the procedure. The equation for the guess of the root is give below:

$$x_{guess} = x_u - \frac{f(x_u)(x_l - x_u)}{f(x_l) - f(x_u)}$$

The false position method suffers from the same problems as the bisection method. Usually, the false position method finds the answer faster, but there are some cases where this is not true.

In the plot below, a line connects the function evaluated at the lower bound and the upper bound. Where this line crosses the y-axis is the first guess, x_1. If the error is not small enough, a new line is drawn. The second line connects the function evaluated at x_u and the new x_l, which was reset to x_1. In the plot, four guesses are shown. In this example, the lower bound was reset after each guess. That is not always the case; the upper bound can also be reset.



OPEN METHODS

There are three open methods that will be discussed in this tutorial: fixed point iteration, Newton-Raphson, and Secant method. The first one requires no understanding of calculus. It is the easiest to implement, but has a low success rate for finding roots. The Newton-Raphson method requires calculating the derivative of a function and using the derivative to identify the next guess. The secant method is similar to Newton-Raphson, but the slope is not calculated from the derivative.

The common theme to these methods is that they require only one initial guess. The guess does not need to be near the root. If multiple roots exist only one will be found by these methods unless multiple initial guesses are used. Newton-Raphson is the most accepted of all of the methods covered in this tutorial, but it is not always perfect.

The major difference for the three methods is how they find the next guess starting from the initial guess.

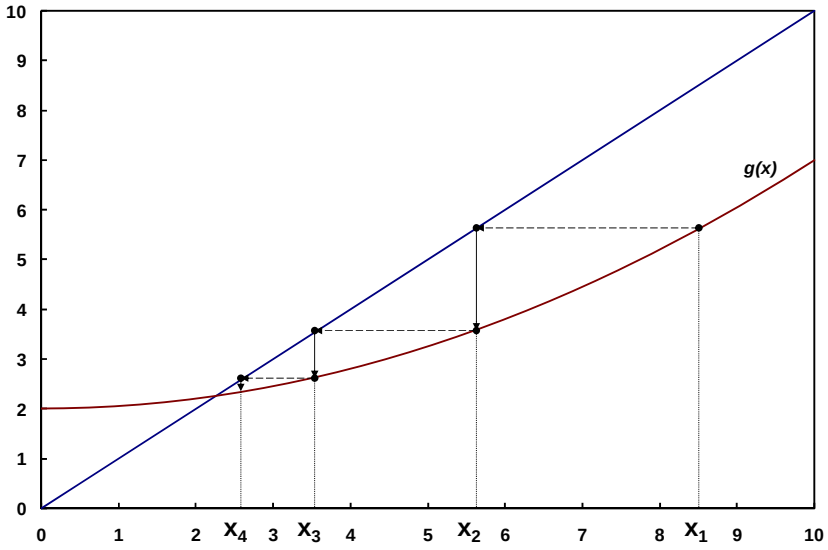
Fixed Point Iteration

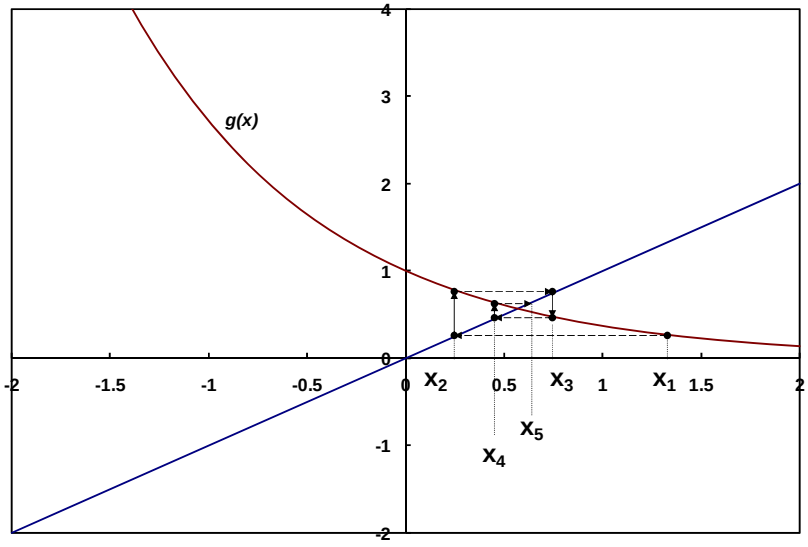
Normally, in root finding, one is trying to find where a function, *f(x)*, intersects the x-axis. In fixed point iteration, *x* is added to both sides of the expression *f(x) = 0* and the intersection of *x* and *f(x) + x* is sought. It is definitely not an intuitive method, but if you plug the initial guess into the right hand side of the equation below, the *x* calculated on the left hand side becomes the new guess.

$$x_{i+1} = f(x_i) + x_i$$

In this equation, x_i is the i^{th} guess and the next guess, x_{i+1}, comes from evaluating the right hand side of the equation. The right hand side is often called *g(x)* to make the notation simpler.

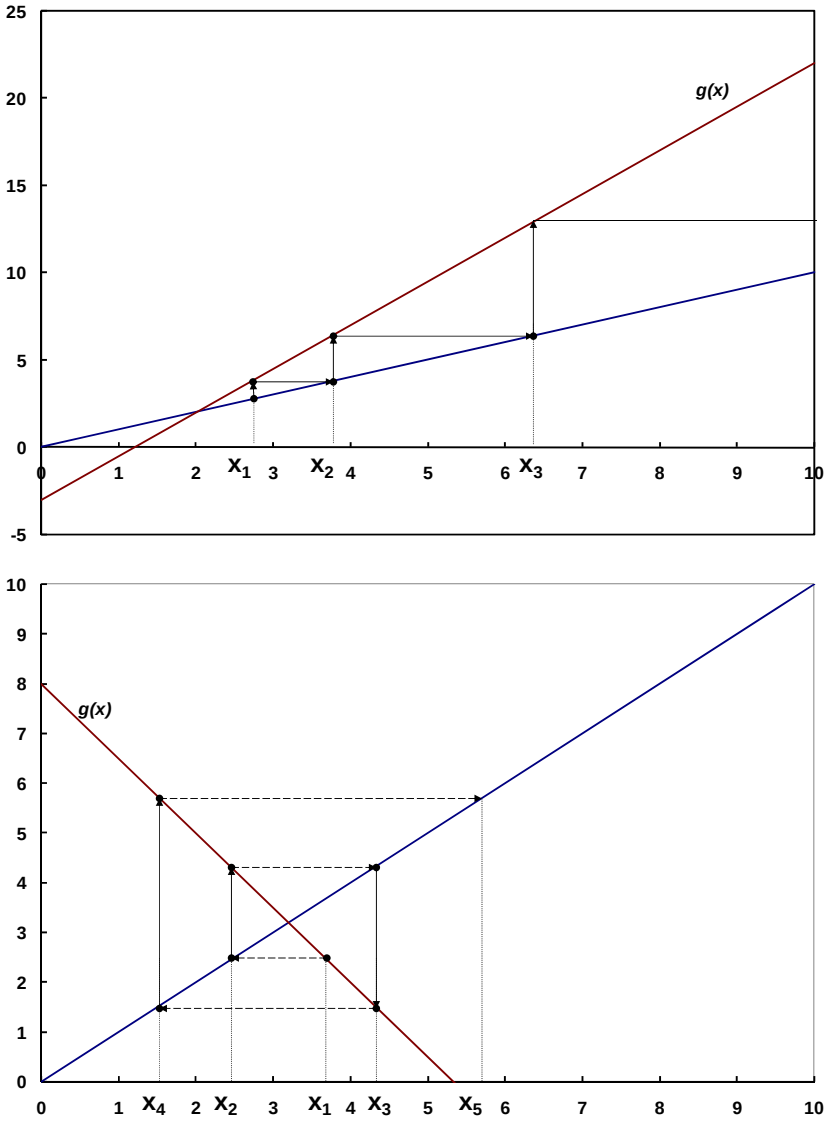
The plots below show in a graphical fashion how this method works. First, evaluate *g(x)* at x_1, the initial guess. These are represented by solid vertical arrows. Then find where this is equal to *x*. This step is represented by dashed horizontal arrows. Then use this new *x* in *g(x)* and keep repeating until the root is found. Each subsequent guess is labeled and indicated with a dotted line down to the x-axis. Two types of convergence exist The first one below is known as a direct convergence where each guess gets closer and closer to the root without passing it. In the second plot, spiral convergence occurs where each subsequent guess overshoots the real root, but the method still zeros in on the actual root.





The final question to answer is: how does one know when to stop? The error tolerance as described previously is perfectly okay to use. Another error that is measured is the difference in each subsequent guess. If the difference between x_i and x_{i+1} is small enough then there is a good chance the root has been found. [Remember that the previously described error is the difference between *g(x)* and *x* whereas the new definition is the difference between x_i and x_{i+1}.]

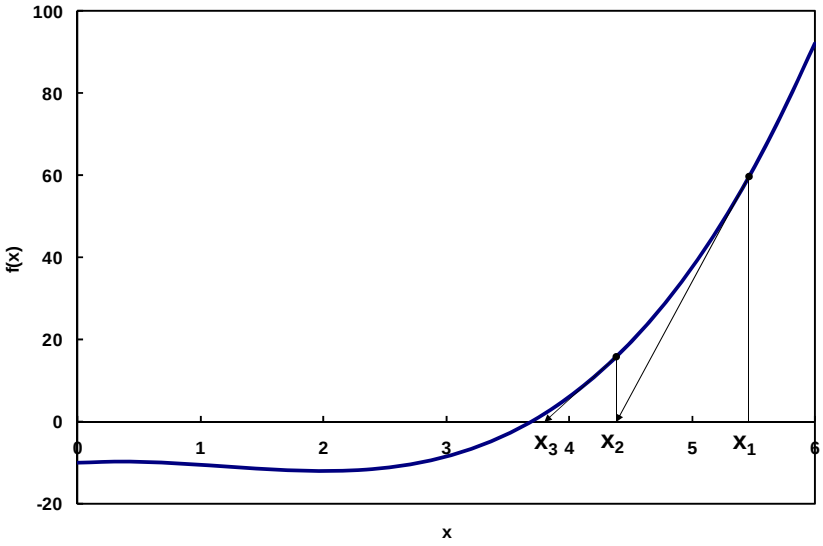
The major weakness of fixed point iteration is that it will not find the root for all equations. Two examples are given below where the solution diverges, which means that it gets further and further away from the root as it iterates. In the first case, there is a direct divergence and in the second case there is spiral divergence. It will do this until the user hits control+C.



The major benefit of fixed point iteration is that it does not require knowledge of the derivative. Unfortunately, for this method at least, this benefit is of little value if the method diverges away from the actual root.

Newton Raphson Method

The Newton Raphson (NR) Method uses the slope of the line, i.e. the derivative, at the guess to find the next guess. This is depicted graphically below:



The function and its derivative are evaluated at the initial guess. These values are plugged into an equation for the next guess and then the pattern is repeated. The equation for the next guess is give below:

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

The *f'(x)* indicates the first derivative of *f(x)*. This equation is found by equating the slope to the change in the y-axis divided by the change in the x-axis:

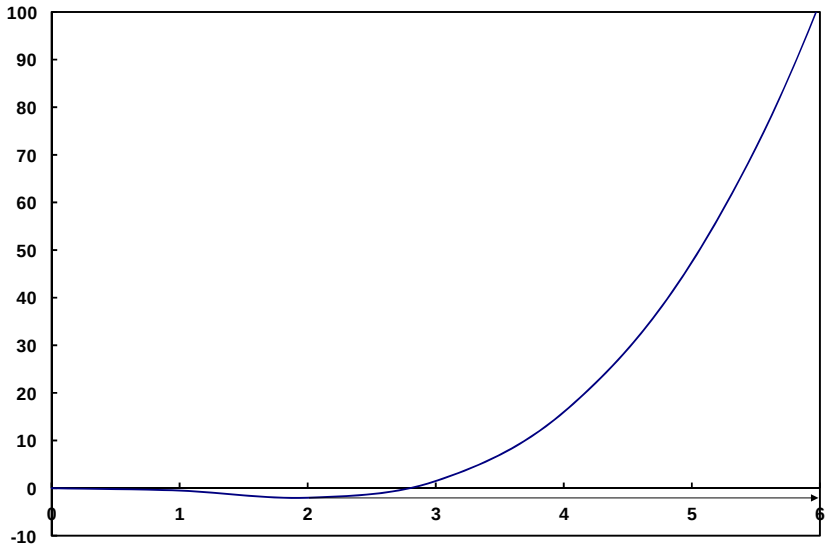
$$f'(x_i) = \frac{f(x_i) - f(x_{i+1})}{x_i - x_{i+1}} = \frac{f(x_i) - 0}{x_i - x_{i+1}} = \frac{f(x_i)}{x_i - x_{i+1}}$$

The function evaluated at x_{i+1} is equal to zero, so that term falls out. This equation can be rearranged to solve for x_{i+1}.

The NR Method does not have the convergence issues that fixed point iteration has. However, there are certain circumstances where the convergence is slow. Generally speaking these problems exist near local minima and maxima, i.e. where the slope of a line is near zero. If the slope is exactly zero, then the equation to find x_{i+1} is undefined and the problem cannot be solved.

In the NR method, the change in the guess is used as the error condition rather than the difference between the function and zero. The reason is that it is possible for the error as defined by the difference between the function and zero to shrink on some steps and then increase on others. This can cause problems if the growing or shrinking of the error is a stopping condition. However, whenever an open method is approaching a root, the difference between each subsequent guess always decreases with each new guess. Below is an example of a function that is slow to converge due to an initial guess near a local minimum. In fact, if the exact point of the local minimum is the guess, the method will

not work, because the derivative is zero and the equation for the next guess will be undefined. If the guess is close to the minimum, as shown below, the next guess will be far from the root even though the initial guess was quite close.



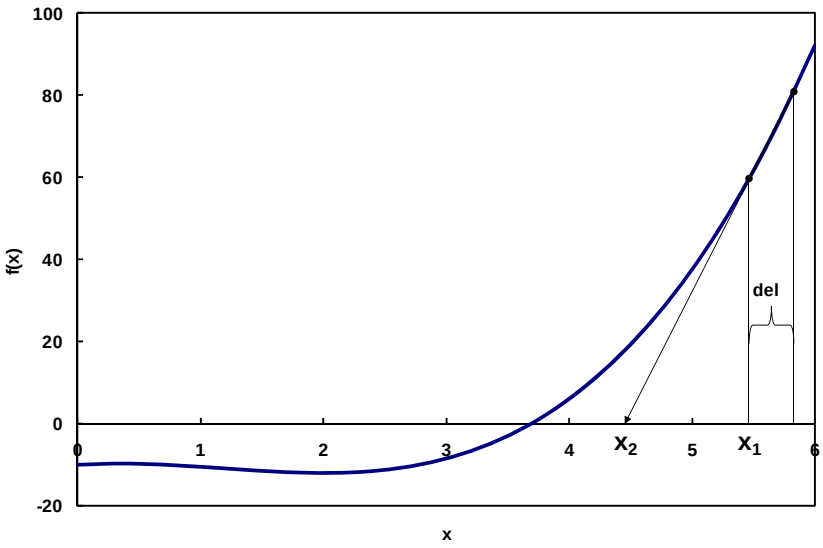
Finally, the major limitation of the NR method is the requirement that the derivative must be known. Even for students who have taken Calculus I, this can be a challenging problem. Think about the bungee jumping equation. The final method will help out with this problem.

Secant Method

The secant method is nearly the same as the NR method. The only difference is that the slope is not calculated from the derivative. It is calculated by specifying some incremental change in *x* that will be called *del*. The slope is then calculated by the following equation:

$$slope = \frac{f(x + del) - f(x)}{(x + del) - x} = \frac{f(x + del) - f(x)}{del}$$

This equation will replace the *f'(x)* used in the NR method. As *del* goes to zero, the secant method approaches the NR method. It is shown graphically below. As long as del is small enough, the secant method will look and act almost exactly like the NR method.



The secant method suffers from most of the same problems as the NR method. However, the analytical derivative does not need to be known and it does not usually lead to an undefined next guess due to a zero derivative.

FINAL NOTE ON PROBLEMS WITH ROOT FINDING

Several examples of problems that exist in root finding have been described throughout this tutorial, such as close roots, double roots, divergence, etc. A student who has mastered the material in this tutorial, should have a decent idea of how to address many of these issues. Due to a lack of time, only a few strategies will be discussed, but the student should be aware that the problems exist and have a good idea of how they would try to go about solving them.

DO IT YOURSELF

The student should be capable of writing code to use any of these methods to calculate roots. Due to the parallels between root finding and numerical integration, few examples of code have been given on purpose to force the student to try these on their own.

The student should also focus on using the *feval* function described in another tutorial, so that these algorithms can be written without being specific to certain functions. In fact, the function whose root is sought should be an input to the root finding m-file.