Math 4610 Lecture Notes

Root Finding Problems for Real Values Function of One Variable *

Joe Koebbe

December 16, 2019

*These notes are part of an Open Resource Educational project sponsored by Utah State University

Root Finding Problem: Definition of a Root Finding Problem

There are many mathematical problems are cast in terms of finding a point in some interval, (a, b), where a function, f, is zero. Finding such locations amounts to the solution of a root-finding problem. For example, in a standard first semester calculus course, the process of finding extreme values of a real-valued function, g, is presented. The problem in one variable can be recast or transformed with some work into the problem of determining locations where the derivative, g', is zero. This is true since a necessary condition for the existence of a local minimum or local maximum value of a differentiable function at a point x^* is that the derivative be zero. In this case, the problem of determining the location of a minimum or maximum value of a function is rewritten as finding the zeros of the derivative of the function. That is, find all points, x^*, such that

$$g'(x^*) = 0.$$

The result is a root finding problem for the derivative of a function.
The following is a general definition of the root finding problem for a real-valued function of a single real variable.

Definition 1 The General Root Finding Problem: *Given a real-valued function, f, of a single real variable find a point or points, x^*, in the domain of the function such that*

$$f(x^*) = 0$$

The value, x^*, is called a root or zero of the function f.

Solution of the general root finding problem seems like it should be easy. However, there are many sources of error and difficulties that are hidden within the definition of the function.
There are all kinds of issues that arise in solving root finding problems. For example, the function may have multiple roots that are close together. This is an issue if, for example, the multiplicity of the root you are looking for is in question. It might be the case that roots located close together may appear as multiple roots due to roundoff error or machine precision issues. In this case, it could be difficult to detect the difference in the locations of the roots. In searching for a specific root, say the largest or smallest, we may find other roots that are not of interest. To deal with all of the issues in this problem, we will develop a number of algorithms that can be used to overcome the problems that arise.
More often than not, we will need to locate roots that cannot be represented exactly due to finite precision in number representation. For example, finding the roots of

$$sin(x) = 0$$

is easy from an analytic point of view. This is a problem covered in all trigonometry courses in high school and college. The zeros are $x_n = n\ \pi$ where n is an arbitrary integer. If n is not equal to zero, the root is an irrational number and cannot be represented exactly in finite precision. So, we must be prepared to settle for an approximation of the roots of a function. It should be noted that an algebraic solution will be available only in cases where $f(x)$ has a simple definition, say a linear or quadratic polynomial. Also, we might be able to guarantee a solution exists, but there may be no analytic means of finding a root or multiple roots for the given function.
As a simple example of proving the existence of roots, consider the function

$$p(x) = 1 + 2\ x + 3\ x^2 + 5\ x^3 + \pi\ x^4 + e^1\ x^5$$

This is a polynomial of degree five. For any polynomial of odd degree, we know from our algebra background there is at least one real root. Since $p(x)$ is a polynomial of degree five, there must be at least one real root. However, based on the coefficients, it will likely be the case that there is no analytic method for computing a root for this problem.
One very complicated root finding problem involves one of the oldest unsolved problems in all of mathematics. The problem is the Riemann conjecture or Riemann hypothesis regarding the distribution of prime numbers in amongst all real numbers. The Riemann-Zeta function is

$$\zeta(s) = \sum_{n=1}^{\infty} \frac{1}{n^s}$$

where s represents an arbitrary complex number. This function is at the heart of the Riemann-conjecture and the distribution of prime numbers. This innocent looking formula is still not completely understood and the Riemann conjecture has elluded all efforts at a solution for more than 100 years. It should be noted that the distribution of primes is central in the development of data encryption strategies in cyber-security applications.

Root Finding Problems: Using Fixed Point Iteration

As a first attempt at determining the location of a root for a function, we might consider a modification of the root finding problem as follows. Given a function, f, we can rewrite the root finding equation

$$f(x) = 0$$

as

$$x = x - f(x) = g(x)$$

The resulting equation is called a fixed point equation or fixed point problem

$$x = g(x)$$

We will use the fixed point problem to define an algorithm for locating roots of a function. So, suppose we have an initial guess at the solution of the fixed point equation, x_0, that may or may not satisfy the equation. We can substitute the value into the fixed point function to obtain

$$x_1 = g(x_0)$$

If $x_0 = x^*$ then the output will be the same as the input, $x_1 = x^*$. If not, the value can be used as another approximation of x^*. We can repeat this process ad infinitum. A general formula for the iteration starts by providing an initial guess, x_0, and then compute

$$x_{k+1} = g(x_k)$$

for $k = 0, 1, 2, \ldots$ This iteration will produce an infinite sequence

$$\{x_k\}_{k=0}^{\infty} = \{x_0, x_1, x_2, \cdots\}$$

of approximations to the solution of the fixed point problem. Since the fixed point problem is equivalent to the root finding problem, we can treat the sequence as approximations of the root finding problem.

Even though we can generate any number of approximations of the solution of the fixed point problem in this way, there is no guarantee that any of these approximations are close to the solution we desire. If a tolerance is specified apriori there is no guarantee that the sequence will be close to anything. In mathmatical terms, what we want is

$$\lim_{k\to\infty} x_k = x^*$$

That is, we would really like the sequence to converge to a root. We will return to this topic after writing a bit of code and presenting an example.

Root Finding Problems: Coding Fixed Point Iteration

One can easily write a routine or computer code that implements fixed point iteration. The following code provides a template of how a reusable routine might be written:

```
//
// Author: Joe Koebbe
//
// Routine Name:         fproot
// Programming Language: Java
// Last Modified:        09/10/19
//
// Description/Purpose: The routine will generate a sequence of numbers
// using fixed point iteration.
//
// Input:
//
// FunctionObject f - the function defined in the root finding problem
// double x0 - the initial guess at the location of a fixed point
// double tol - the error tolerance allowed in the approximation of the
//              root finding problem
// int maxit - the maximum number of iterations allowed in the fixed point
//             iteration.
//
// Output:
//
// double x1 - the last number in the finite sequence that is an
//             approximation in the root finding problem
//
public double fproot(FunctionObject f, double x0, double tol, int maxit) {
  //
  // initialize the error in the routine so that the iteration loop will be
  // executed at least one time
  // -------------------------
  //
  double error = 10.0 * tol;
  //
  // initialize a counter for the number of iterations
  // ------------------------------------------------
  //
  int iter = 0;
  //
  // loop over the fixed point iterations as long as the error is larger
  // than the tolerance and the number of iterations is less than the
  // maximum number allowed
  // ---------------------
  //
  while(error > tol && iter < maxit) {
    //
    // update the number of iterations performed
    // -----------------------------------------
    //
    iter++;
```

```
        //
        // compute the next approximation
        // ------------------------------
        //
        double x1 = x0 - f(x0);
        //
        // compute the error using the difference between the iterates in the
        // loop
        // ----
        //
        error = Math.abs(x1 - x0);
        //
        // reset the input value to be the new approximation
        // --------------------------------------------------
        //
        x0 = x1;
        //
      }
      //
      // return the last value computed
      // -------------------------------
      //
      return x1;
      //
    }
```

There are a couple of features in the code that need to be explained.

- To make this work in the Java programming language, the method would need to be embedded in a class. That is, the code is not a standalone code.
- The first argument is an Java Object that needs to be created. The object is used to provide the function evaluation for any real input.
- The second argument is the initial guess at the solution of the problem.
- Since we know we are going to end up with at best an approximation of a root, the third argument in the function is an error tolerance that is acceptable to the calling routine.
- The final argument passed in limits the number of iterations allowed in the method. Note that if you are not careful, an infinite loop might be created due to the approximations used everywhere.

If we apply the code to any problem, we are assuming that the solution will pop out the end. There is no guarantee that this is the case. It is important to establish conditions that will guarantee the code will produce an approximate solution of the fixed point problem and thus provide a root for the original function, f.

Root Finding Problems: Analysis of Functional Iteration Using Taylor Series Expansion

The general iteration formula, given x_0, is the following.

$$x_{k+1} = g(x_k)$$

for $k = 0, 1, 2, \ldots$. We also know that for the fixed point problem, the solution satisfies the equation

$$x^* = g(x^*)$$

Subtracting the two equations gives

$$x_{k+1} - x^* = g(x_k) - g(x^*)$$

The Taylor expansion of $g(x_k)$ about the solution x^* is given by

$$g(x_k) = g(x*) + g'(x^*)(x_k - x^*) + \frac{1}{2}g''(x^*)(x_k - x^*)^2 + \ldots$$

Substituting the expansion into the equation above and truncating the series gives

$$x_{k+1} - x^* \approx g(x*) + g'(x^*)(x_k - x^*) - g(x^*) = g'(x^*)(x_k - x^*)$$

Taking absolute values the last equation can be written as

$$|x_{k+1} - x^*| \leq |g'(x^*)| \cdot |x_k - x^*|$$

One can read the previous expression as the difference (or error) in x_{k+1} is less than the magnitude of the derivative of the fixed point iteration function, g, times the difference (or error) in the previous approximation, x_k. Using

$$e_k = |x_k - x^*|$$

allows use to relate the error at successive steps as

$$e_{k+1} \leq |g'(x^*)| \cdot |e_{k+1}|$$

To get convergence to the fixed point (or root) we would like the error to be reduced at each step. This requires the condition

$$|g'(x^*)| < 1$$

For the general fixed point problem, this condition is required for convergence to the fixed point, x^*, or solution of the root finding problem. Note that this is a significant drawback of fixed point iteration as a means of solving root finding problems.

Root Finding Problems: An Example Using Functional Iteration

Suppose that we are interested in computing the roots of

$$f(x) = e^x - \pi$$

Analytically we can compute the solution by solving for x in the equation

$$e^x - \pi = 0$$

The value is $x = ln(\pi) \approx 1.144729886$. This is a very simple problem. However, it is always a good idea to test general methods on simple problems while developing algorithms and coding these up for use on real problems. Let's apply functional iteration to this root finding problem. First, we will need to create an associated function that defines a fixed point problem. One possibility is to choose

$$g_1(x) = x - f(x) = x - (e^x - \pi) = x - e^x + \pi$$

Let's check the condition for convergence by computing the derivative of g near at the solution above.

$$g_1'(x) = 1 - e^x = 1 - \pi \approx -2.14159245 \to |g_1'(x)| \approx 2.14159245$$

The value is bigger than 1 which means the sequence of iterates is not going to converge. So, the choice of $g(x)$ will not work.
As another option, consider a modification of the function. If

$$f(x) = e^x - \pi = 0$$

then

$$f(x) = \frac{1}{5}(e^x - \pi) = 0$$

which allows us to write

$$g_2(x) = x - \frac{1}{5}(e^x - \pi)$$

with derivative

$$g_2'(x) = 1 - \frac{1}{5}e^x$$

and near the solution

$$|g_2'(x)| = |1 - \frac{1}{5}\pi| < 1.0$$

So, we can expect better results in this case.

Root Finding Problems: Example Results Tabulated

For the two examples, the output for the two choices of the iteration function $g_1(x)$ or $g_2(x)$.

Table 1: Results for Functional Iteration for Two Different Iteration Functions

Iteration No.	$g_2(x) = x - (e^x - \pi)$	error	$g_1(x) = x - (e^x - \pi)$	error
01	1.08466220	8.46621990E-02	1.42331100	0.423310995
02	1.12129271	3.66305113E-02	0.41406250	1.00924850
03	1.13584745	1.45547390E-02	2.04270363	1.62864113
04	1.14140379	5.55634499E-03	-2.52713394	4.56983757
05	1.14349020	2.08640099E-03	0.53457117	3.06170511
06	1.14426863	7.78436661E-04	1.96944773	1.43487656
07	1.14455843	2.89797783E-04	-2.05567694	4.02512455
08	1.14466619	1.07765198E-04	0.95790958	3.01358652
09	1.14470625	4.00543213E-05	1.49325967	0.535350084
10	1.14472115	1.49011612E-05	0.18326997	1.30998969
11	1.14472663	5.48362732E-06	2.12372398	1.94045401

So, two completely different results are obtained. One converges with a slight modification to the first. The first function produces a sequence that does not converge and the second produces the correct result up to machine precision. That is, $x^* = 1.14472663$ with absolute error $5.48362732E - 06$. This is one of the reasons why functional iteration is not used as much. The problem is that there are infinitely many choices for the fixed point equation. Some will provide convergence and others will not come close.

Root Finding Problems: Convergence of Functional Iteration

If we end up using functional iteration, it will also pay to know how fast the sequence converges. Fewer iterations means faster results with few computations. The convergence of the sequence is determined by the same calculations as in the convergence justification above.

$$|x_{k+1} - x^*| \leq |g'(x^*)| \cdot |x_k - x^*|$$

For functional iteration the convergence rate is defined by

$$\text{rate of convergence} = |g'(x^*)|$$

The smaller the magnitude of the derivative, $|g'(x^*)|$, the faster the convergence will be.
As an example, consider changing the parameter $\frac{1}{5}$used to modify the iteration function,

$$g_2(x) = x - (e^x - \pi) \rightarrow \; g_2(x) = x - \frac{1}{5}(e^x - \pi)$$

to keep the original root the same in the previous section. If the parameter is decreased, the rate of convergence should be faster. This is covered in one of the homework tasks.

Root Finding Problems: Continuous Functions and the Bisection Method

On the positive side of things, the fixed point approach in the previous section requires very little of the function in the root finding problem. The only requirement is that f is a function at every input value. It is usually very easy to implement fixed point iteration for this type of problem. It may be difficult if not impossible to come up with a fixed point problem that will provide convergence to any fixed point. Due to slow convergence and issues finding a fixed point equation that works, functional iteration is limited in applicability in the real world.
So, we need to develop alternative algorithms for the root finding problem. In this section, we will assume that the function, f, is continuous on a closed and bounded interval $[a, b]$ where we expect to find a root. The main mathematical tool used in this case is the Intermediate Value Theorem for continuous functions.

Theorem: Suppose the function, f, is continuous on the closed and boundaed interval $[a, b]$. If M is any value between $f(a)$ and $f(b)$ then there exists a value $c \in (a, b)$ such that $f(c) = M$,

---

Now, if $f(a) \geq 0 \geq f(b)$ (or vice-versa) then there is at least one value, c in the interval (a, b) such that $f(c) = 0$. If we determine end-points of an interval such that $f(a) < 0$ **and** $0 < f(b)$ (or vice versa), we know there is also a root of the function somewhere in the interval we have selected. There is a simple condition that can be test to verify an interval contains an interval. That is,

$$f(a) \cdot f(b) < 0$$

This is enough to determine that the function crosses the horizontal axis at at least one point in the interval.

Root Finding Problems: Bisection and Convergence

Once we have determined an interval $[a, b]$ such that $f(a)\ f(b) < 0$ we can start work to determine the location of a root in the initial interval. We proceed by bisecting the original interval $[a, b]$ into two equal subintervals

$$[a, b] = [a, c] \cup [c, b]$$

where

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

Since there is at least one root on $[a, b]$ there are three possibilities that can occur in the bisection. These are:

- $f(c) = 0$,
- $f(a) \cdot f(c) < 0$ which implies there is a root in $[a, c]$, or
- $f(c) \cdot f(b) < 0$ which implies there is a root in $[c, b]$.

If the first condition is true, we have the root, $x^* = c$ and we are done searching. In the second case, we can redefine the search interval to $[a, c]$ and in the third case, the search interval will be redefined to be $[c, b]$. Once we have redefined the search interval, we repeat the bisection on this new search interval. The bisection will reduce the size of the search interval by a factor of two. We just need to translate this idea into a computer code in some language.

Root Finding Problems: A (First) Simple Bisection Code in C

The following routine, written in something like C implements the Bisection Method.

```
double bisectionMethod(typedef'd f, double a, double b, double tol,
                       int maxiter)
{
  //
  // set up some parameters and local variables to do the work
  // ---------------------------------------------------------
  //
  double c;
  double error;
  int iter;
  //
  // check the endpoints - if either is 0, we already have a root
  // ------------------------------------------------------------
  //
  if(f(a)==0) return a;
  if(f(b)==0) return b;
  //
  // check for a root in the interval
  // --------------------------------
  //
  if(f(a)*f(b) >= 0.0) throw an error or print a message
  //
  // set the error and iteration counter
  // -----------------------------------
  //
  error = 10.0 * tol;
  iter = 0;
  //
  // use a while loop to go until the tolerance is met or the maximum
  // number of iterations has been exceeded
  // --------------------------------------
  //
  while(error > tol && iter < maxiter) {
    //
    // update the iteration counter and compute the midpoint of the current
    // interval
    // --------
    //
    iter++;
    c = 0.5 * ( a + b );
    //
    // compute the sign change value
    // -----------------------------
    //
    double val = f(a) * f(c);
    //
    // reassign the end point based on the location of the root
    // --------------------------------------------------------
```

```
      //
      if(val<0.0) {
        b = c;
      } else {
        a = c;
      }
      //
      // compute the error in the approximation - this assumes a<b
      // --------------------------------------------------------
      //
      error = b - a
      //
    }
    //
    // return the midpoint as it is more accurate
    // ------------------------------------------
    //
    return c;
    //
  }
```

The first argument in the C method needs to be changed to a pointer to a function as an input to the method. This is left up to the reader to do. It should be noted that once an interval has been determined on which the function value changes sign, the Bisection Method will continue until a root is found, at least up to machine precision. We can take advantage of this property to redesign the algorithm to take a specific number of iterations instead of checking the error.

Root Finding Problems: The Bisection Method and Error Reduction

The fact the the interval size is being reduced in each iteration of bisection can be used as follows. The length of the original interval can be computed and used to bound the error in any approximation of a root. That is,

$$|x - x^*| \leq |b - a|$$

A sequence of intervals is created by the Bisection method that contains a root. We can use subscripts to define the intervals as the bisection proceeds. If we use $[a_i, b_i]$, for $i = 0, 1, \ldots$ where each new interval is selected after the previous interval is bisected. Note that if we are assuming $[a_0, b_0] = [a, b]$ in this argument. So, we can write the following set of inequalities

$$|x - x^*| < b_k - a_k < \frac{1}{2}(b_{k-1} - a_{k-1}) < \cdots < \frac{1}{2^k}(b_0 - a_0) = 2^{-k}(b - a)$$

This means that once the interval $[a, b]$ has been determined, the reduction in the error between iterations is computable.

Suppose that we specify an error tolerance that is acceptable, say 10^{-d} where d is the number of digits of accuracy. Then we can define the number of iterations to reduce the error to the desired tolerance as follows.

$$2^{-k}(b - a) < 10^{-d}$$

Using a bit of algebra

$$2^{-k} < \frac{10^{-d}}{(b - a)} \rightarrow -k < log_2\left(\frac{10^{-d}}{(b - a)}\right)$$

or flipping the inequality using a negative multiplier

$$-log_2\left(\frac{10^{-d}}{(b - a)}\right) < k$$

This gives us the total number of iterations needed to reduce the error to the desired tolerance. So, we can rewrite the code to take advantage of this calculation.

Root Finding Problems: An Alternative Bisection Method Code

The alternative C code to implement the alternate Bisection method where the number of iterations is computed ahead of time is the following.

```
double bisectionMethod(typedef'd f, double a, double b, double tol) {
  //
  // set up some parameters and local variables to do the work
  // ---------------------------------------------------------
  //
  double c;
  double error;
  //
  // check the endpoints - if either is 0, we already have a root
  // ------------------------------------------------------------
  //
  if(f(a)==0) return a;
  if(f(b)==0) return b;
  //
  // check for a root in the interval
  // --------------------------------
  //
  if(f(a)*f(b) >= 0.0) throw an error or print a message
  //
  // compute the number iterations needed to meet the tolerance given
  // ----------------------------------------------------------------
  //
  maxiter = - 2.0 * log2( tol / ( b - a ) );
  //
  // compute the iterations
  // ----------------------
  for(int i=0; i<maxiter; i++) {
    //
    // compute the midpoint of the current interval
    // --------------------------------------------
    //
    c = 0.5 * ( a + b );
    //
    // compute the sign change value
    // -----------------------------
    //
    double val = f(a) * f(c);
    //
    // reassign the end point based on the location of the root
    // --------------------------------------------------------
    //
    if(val<0.0) {
      b = c;
    } else {
      a = c;
    }
    //
```

```
    }
    //
    // return the midpoint as it is more accurate
    // ------------------------------------------
    //
    return c;
    //
  }
```

Note that the output value will be an approximation of a root in the original interval, $[a, b]$ that satisfies the desired tolerance.

Root Finding Problems: Bisection Method Examples

It is always a good idea to test the code you write. Using the example from our tests of functional iteration we can detemine whether or not the Bisection Method works and how this compares with the fixed point approach. So, we will consider the example in the section on functional iteration. That way, we can compare the results using Bisection to our previous work.
So, we will work with the easy example,

$$f(x) = e^x - \pi = 0$$

and apply the Bisection method on the interval $[-2.2, 6.8]$. Note that we do not need to come up with an alternate definition of the problem as in the case of functional iteration. The results shown include functional iteration and the Bisection method and are computed towards a tolerance of 10^{-7}.

Table 2: Results for Functional Iteration Compared to Bisection

Iteration No.	Bisection	Bisection error	Functional Iteration	Functional Iteration error
01	2.30000019	1.15527022	1.08466220	8.46621990E-02
02	5.00000715E-02	1.09472990	1.12129271	3.66305113E-02
03	1.17500019	3.02702188E-02	1.1358474	1.45547390E-02
04	0.612500131	0.532229841	1.14140379	5.55634499E-03
05	0.893750191	0.250979781	1.14349020	2.08640099E-03
06	1.03437519	0.110354781	1.14426863	7.78436661E-04
07	1.10468769	4.00422812E-02	1.14455843	2.89797783E-04
08	1.13984394	4.88603115E-03	1.14466619	1.07765198E-04
09	1.15742207	1.26920938E-02	1.14470625	4.00543213E-05
10	1.14863300	3.90303135E-03	1.14472115	1.49011612E-05
11	1.14423847	4.91499901E-04	1.14472663	5.48362732E-06
12	1.14643574	1.70576572E-03	No Data	No Data
13	1.14533710	6.07132912E-04	No Data	No Data
14	1.14478779	5.78165054E-05	No Data	No Data
15	1.14451313	2.16841698E-04	No Data	No Data
16	1.14465046	7.95125961E-05	No Data	No Data
17	1.14471912	1.08480453E-05	No Data	No Data
18	1.14475346	2.34842300E-05	No Data	No Data
19	1.14473629	6.31809235E-06	No Data	No Data
20	1.14472771	2.26497650E-06	No Data	No Data
21	1.14473200	2.02655792E-06	No Data	No Data
22	1.14472985	1.19209290E-07	No Data	No Data
23	1.14473093	9.53674316E-07	No Data	No Data
24	1.14473033	3.57627869E-07	No Data	No Data

The results actually show that the functional iteration approach actually converges faster and is more efficient. However, as mentioned earlier, the functional iteration approach requires the definition of an alternative problem. Bisection works as long as the function in question is continuous on a closed and bounded interval. The guarantee is that once a root is bracketed the method will trdge along until an approximate value for the root is determined up to a a given tolerance.

Root Finding Problems: Differentiable Functions

The next method that we can cover is Newton's method. This method is based on using some simple calculus manipulations to determine an iterative method for approximating roots of a nonlinear function. So, consider a function $f(x)$ that is twice differentiable in some open interval containing a root of the function. There are a couple of ways to develop Newton's method. For this set of notes, suppose that, x_0, is provided as an approximation of the root. We can expand the function using the unknown root, x^*, and the approximation, x_0. That is, using x_0 as the center of the expansion,

$$f(x^*) = f(x_0) + f'(x_0)\ (x^* - x_0) + \frac{1}{2}\ f'(x_0)\ (x^* - x_0)^2 + \cdots$$

The expansion can be truncated using Taylor's theorem with remainder to write

$$f(x^*) = f(x_0) + f'(x_0)\ (x^* - x_0) + \frac{1}{2}\ f'(\xi)\ (x^* - x_0)^2$$

where ξ is between x^* and x_0. This expansion works as long as the function, $f(x)$, is twice continuously differentiable. Mathematically, the differentiability condition implies that near x_0 (and x^*) the remainder term can be bounded as follows.

$$\frac{1}{2}\ f'(\xi)\ (x^* - x_0)^2 \le C\,(x^* - x_0)^2$$

If x^* and x_0 are sufficiently close, we can neglect this term and write the approximation

$$f(x^*) \approx f(x_0) + f'(x_0)\ (x^* - x_0)$$

Recall that $f(x^*) = 0$. So, we can write

$$0 \approx f(x_0) + f'(x_0)\ (x - x_0)$$

for any x near x^*. Using this, we can define another approximation using

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

This formula suggests an iteration. Given the output, x_1, given the input x_0, we can continue this process and compute another approximation, x_2, using

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

This leads to the definition of Newton's method.

Root Finding Problems: Definition of Newton's Method

Given the work in the previous section of these notes, we can define Newton's method as follows. Given an initial guess, x_0, the sequence of points, x_k, given by

$$x_k = x_{k-1} - \frac{f(x_{k-1})}{f'(x_{k-1})}$$

for $k = 1, 2, \ldots$ defines Newton's method for finding the roots of a function of a single variable.
There are certain restrictions that must be met when using Newton's method.

- The function must be twice continusously differentiable,
- the derivative of the function cannot be zero at the root, x^*, and
- the initial point must be chosen sufficiently close to the exact value of the root.

Root Finding Problems: Newton's Method Example and Comparison

Table 3: Results for Functional Iteration Compared to Bisection

Iteration No.	Newton Method	Newton Method Error	Bisection Method	Bisection Method
01	0.0000000000000000	1.1447299136769349	2.30000019	1.15527022
02	2.1415927410125732	0.99686282733563836	5.00000715E-02	1.09472990
03	1.5106280957127742	0.36589818203583935	1.17500019	3.02702188E-02
04	1.2042015115607474	5.9471597883812510E-002	0.612500131	0.532229841
05	1.1464638070151236	1.7338933381887411E-003	0.893750191	0.250979781
06	1.1447314160015734	1.5023246384693323E-006	1.03437519	0.110354781
07	1.1447299136780633	1.1284306822290091E-012	1.10468769	4.00422812E-02
08	No Data	No Data	1.13984394	4.88603115E-03
09	No Data	No Data	1.15742207	1.26920938E-02
10	No Data	No Data	1.14863300	3.90303135E-03
11	No Data	No Data	1.14423847	4.91499901E-04
12	No Data	No Data	1.14643574	1.70576572E-03
13	No Data	No Data	1.14533710	6.07132912E-04
14	No Data	No Data	1.14478779	5.78165054E-05
15	No Data	No Data	1.14451313	2.16841698E-04
16	No Data	No Data	1.14465046	7.95125961E-05
17	No Data	No Data	1.14471912	1.08480453E-05
18	No Data	No Data	1.14475346	2.34842300E-05
19	No Data	No Data	1.14473629	6.31809235E-06
20	No Data	No Data	1.14472771	2.26497650E-06
21	No Data	No Data	1.14473200	2.02655792E-06
22	No Data	No Data	1.14472985	1.19209290E-07
23	No Data	No Data	1.14473093	9.53674316E-07
24	No Data	No Data	1.14473033	3.57627869E-07

These results...