Math 4610 Lecture Notes

Root Finding Problems for Real Values Function of One Variable [*]

Joe Koebbe

September 10, 2019

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

Root Finding Problem: Definition of the Problem

Many problems can be recast in the form of finding places where a function is zero. In a standard first semester calculus course find extreme values of a function of one variable amounts to determining locations where the derivative of the function is zero. That is, a necessary condition for the existence of a local minimum or local maximum at a point x^8 is that the derivative is zero at x^* or

$$f'(x_0) = 0.$$

More often than not, we will need to deal with roots that are not exact in machine precision. For example, finding the roots of

$$sin(x) = 0$$

is easy from an analytic point of view, the zeros are $x_n = n\ \pi$ where n can be any integer. If n is not equal to zero, the root is an irrational number and cannot be represented exactly. So, we will need to settle for an approximation.

The general root finding problem can be written as follows: For a given real-valued function, f, of a single real variable find a real number, x^*, such that

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

There are all kinds of issues that arise in solving these types of problems. For example, the function may have multiple roots. In searching for a specific root, 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 in a variety of root finding problems.

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 equation

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

as

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

The resulting equation is called a fixed point equation and the equation suggests an iteration of the form

$$x_1 = g(x_0), x_2 = f(x_1), \ldots$$

where x_0 must be supplied to start the iteration. This iteration formula will produce a sequence of real numbers. The hope is that the sequence will converge to a solution of the fixed point equation and also a solution of the root finding problem.

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.

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

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

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

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

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 in the previous section. If the parameter is decreased, what happens to the convergence? This is covered in the homework tasks.

Root Finding Problems: Continuous Functions and the Bisection Method

Functional iteration is limited in applicability in the real world. So, we move on to the next algorithm for the root finding problem. In this section, we will assume that the function, f, is continuous on a closed and bounded interval $[a, b]$. 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 $f(a) \leq 0$ **and** $0 \leq f(b)$ (or vice versa), we know there is also a root of the function on the interval we have determined. There is a simple condition that can be test to verify an interval will work. That is,

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

This is enough to determine that the interval straddles the horizontal axis.

Root Finding Problems: Bisection and Convergence

If the original interval $[a, b]$ is bisected 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 the interval there are three possibilities

- $f(c) = 0$,
- $f(a) \cdot f(c) < 0$ 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, two have the root, $x^* = c$ and we are done searching. In the second case, we redefine the search interval to $[a, c]$ and in the third case, the search interval is redefined to be $[c, b]$. Once we have redefined the interval, we repeat the bisection on the new interval. The bisection will reduce the size of the search interval each time through. We just need to translate this into code.

Root Finding Problems: Bisection and Convergence

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

```
double bisectionMethod(typedef'd f, double a, double b, double tol, int maxiter)
{
  // set up some parameters
  double c;
  double error;
  int iter;

  // check the endpoints
  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

  //this part of the code will perform the iterations
  error = 10.0 * tol;
  iter = 0;
  while(error > tol && iter < maxiter) {
    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;

}
```

It should be noted that once an interval has been determined for 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 used to bound the error in any approximation of a root. That is,

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

A sequence of intervals is created by the Bisection method that we can write as $[a_i, b_i]$, for $i = 0, 1, \ldots$ where each new interval shares an endpoint from the previous interval and the other end point is the midpoint of the previous interval. Note that $[a_0, b_0] = [a, b]$ 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 amount of error in the approximation is computable at each iteration.
Suppose that we specify an error tolerance that is acceptable, say 10^{-d} where d is the number of digits of accuracy. Then define the number of iterations to reduce the error to the desired tolerance

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

Using a bit of algebra

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

or using the negattion of the inequality

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

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

Root Finding Problems: An Alternative Bisection Method Code

The alternative code is the following.

```
double bisectionMethod(typedef’d f, double a, double b, double tol, int maxiter)
{
  // set up some parameters
  double c;
  double error;

  // check the endpoints
  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

  //this part of the code will perform the iterations
  maxiter = - 2.0 * log2( tol / ( b - a ) );
  for(int i=0; i<maxiter; i++) {
    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 satisfy the desired tolerance.