Math 4610 Lecture Notes

Errors and Computing the Rate of Convergence [*]

Joe Koebbe

October 24, 2019

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

Errors and Rate of Convergence: An Introduction

The basic mission in this section of the course notes is to determine convergence rates for various algorithms we use to compute approximate solutions in many mathematical problems. For example, we can consider applying the Bisection method for finding roots of real-valued functions of a real variable. The algorithm for computing approximate locations of roots amounts to an initial application of the Intermediate Value Theorem to a continuous function and then proceed by repeatedly halving the interval and determining which of the two subintervals contains a root for the function. In addition to the Bisection method, other methods, like Newton's method, can be applied to the same problem.

Given that there are multiple methods that can be applied to obtain an approximation for a root, it makes sense to determine which of these methods is "best" for a given function. One measurement of the effectiveness of a given method is whether or not the algorithm will converge to any type of solution. Another measure involves the rate at which a given method converges. For example, in the root finding method, if the function in question is continuous and an interval, $[a, b]$, exists such that $f(a) \cdot f(b) < 0$, the Bisection method is guaranteed to converge. For Newton's method, the function needs to be twice continuously differentiable, the derivative cannot be zero at or near the root, and finally the initial guess at a solution must be sufficiently close to the root. The last condition is something that causes some issues with both Newton's method and the Secant method. In fact, if the initial guess is not close enough, there is no guarantee that a root can be found. From this point of view, the Bisection method seems to be better. However, the Bisection method reduces the error in successive iteration slower than either Newton's method of the Secant method.

In this rest of this section of notes, we will investigate how to compute the rate of convergence for various methods to solve the same problem. Readers should realize that the same issue of convergence applies to just about any computational method for solving just about any type of mathematical problem. From root finding to solving linear systems of equations to image processing algorithms to numerical solution of differential equations, multiple methods have been developed to solve the same problem. We need ways to determine which method will be best for a given problem and conditions.

It should also be noted that there are a lot of other considerations that cause practical problem. For example, limitations on processors, CPU speed, and storage on a computer. We will consider these issues at another point in the course.

Errors and Rate of Convergence: Definitions of Convergence and Rate of Convergence

To start a discussion about rate of convergence, we need some definitions. We have already defined the absolute and relative error. The formulas for these are

$$\text{absolute error} = |x - x^*|$$

and

$$\text{relative error} = \frac{|x - x^*|}{|x^*|}$$

for approximations and solution values that are real numbers. Suppose now that an algorithm (say the Bisection method) has created a sequence of approximations

$$\{x_0, x_1, x_2, \ldots, x_k, \ldots\}$$

Note that it will be assumed that the approximations, x_k, are computed using the incerasing index. The error in each of these approximations is given by

$$e_k = x_k - x^*$$

or

$$e_k = \frac{x_k - x^*}{x^*}$$

depending on which error makes sense in the computation.
The result will be a sequence of errors of the form

$$\{e_0, e_1, e_2, \ldots, e_k, \ldots\}$$

The questions we need to answer as computational mathematicians are:

- **Convergence to the Solution:** For convergence can we determine if the error in our computation, e_k, goes to zero as the iteration index, k, tends to infinity. Thus if the iterations are converging to the exact solution,

$$\lim_{k\to\infty} |e_k| = 0$$

- **Reduction in Error per Iteration:** How much does the sequence of errors reduce for each iteration? We can state this in the following way.

$$|e_{k+1}| < |e_k|$$

  If this inequality cannot be established, then the approximations may not converge to the exact solution.
- **Rate of Convergence:** If the sequence of errors converges to zero, how fast is the convergence? We will measure this using the following inequality.

$$|e_{k+1}| < C \cdot |e_k|^r$$

  The positive constant C cannot depend on the index, k, and the power, r, is called the convergence rate. The convergence rate , r, must be greater than zero to guarantee convergence.

There are several situations that we will need to consider. Firet, if we decide to test the algorithm, it makes sense to test the algorithm for a problem where the solution is known. For example, suppose we consider the very simple equation

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

We can eaily compute a solution using simple algebraic techniques. The single root for this function is

$$x = \ln(\pi)$$

The solution is an irrational number. However, any hand held calculator can be used to compute an accurate approximation of the solution. This solution can slso easily be intrduced into a root finding code.
Note that we can choose a simpler problem with integer roots, say

$$f(x) = x^2 - 5\ x + 6 = 0$$

with one of two roots $x^* = 3$ to search for. As soon as the computer starts to work, there will be a loss of precision since roundoff error will creep in to the iterations. So, using an example like

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

is as good as any. This example will be used to generate results to make sure we generate a sequence of approximations that converge to the known solution given above. The second situation is when we are faced with approximating a root when the exact solution is not known. A third problem involves finding multiple unknown roots for a given function. The multiple roots problem will be discussed in the last section in these notes. The rest of notes in this section will treat these problems.

Errors and Rate of Convergence: Case 1. An Exact Solution is Known

For this case, we will use the example given previously where

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

with $x = \ln(\pi)$. Suppose that the closed interval, $[-2.2, 6.8]$, is chosen to test the Bisection code (written in Fortran that you might be required to translate).

```
      program main
c
c variables for the test
c ---------------------
c
      real a, b, c, pi, tol
      real ek, ekp1
      real a11, a12, a22, b1, b2
      real aval, rval
      real detval
      integer n
c
c initialize the interval and tolerance desired
c ---------------------------------------------
c
      a = -2.2
      b = 6.8
      tol = 0.0000001
c
c initialize storage for the linear regression to determine the
c computational rate of convergence - note that the normal equations
c matrix is symmetric.
c -------------------
c
      a11 = 0.0
      a12 = 0.0
      a22 = 0.0
c
      b1 = 0.0
      b2 = 0.0
c
c compute the number of bisections needed for the given tolerance
c ---------------------------------------------------------------
c
      n = - alog10( tol / ( b - a ) ) / alog10(2.0) - 1
      a11 = dble(n)
c
c print out a header for the output
c --------------------------------
c
      print *," Iter:   ek      ekp1     log(ek)      log(ekp1)"
      print *," ----------------------------------------------"
c
c compute an upper bound on the initial error
```

```
c -------------------------------------------
c
      ek = b - a
c
c do the iterations
c -----------------
c
      do 1 i=1, n
c
c compute the midpoint of the interval
c ------------------------------------
c
         c = 0.5 * ( a + b )
c
c test for the root
c -----------------
c
         if(f(a)*f(c) .lt. 0.0) then
            b = c
         else
            a = c
         endif
c
c compute an upper bound on the error for the root location
c ---------------------------------------------------------
c
         ekp1 = b - a
c
c print out the next line in the table of errors
c ----------------------------------------------
c
         print *, i, ek, ekp1, alog10(ek), alog10(ekp1)
c
c update the error variables in the normal equations
c --------------------------------------------------
c
         a12 = a12 + alog10(ek)
         a22 = a22 + alog10(ek) * alog10(ek)
         b1 = b1 + alog10(ekp1)
         b2 = b2 + alog10(ek) * alog10(ekp1)
c
c update the errors for the next step
c -----------------------------------
c
         ek = ekp1
c
    1 continue
c
c compute the solution using the inverse of the 2x2 matrix for the linear
c reduction of the matrix
c -----------------------
c
      detval = a11 * a22 - a12 * a12
      aval = ( a22 * b1 - a12 * b2 ) / detval
      rval = ( a11 * b2 - a12 * b1 ) / detval
c
c output the results - note that the constant needs to be exponentiated
c ---------------------------------------------------------------------
c
      print *, "shift constant:   ", exp(aval)
      print *, "rate of convergence:   ", rval
c
c this code will be used to test convergence rates for the bisection method
c -------------------------------------------------------------------------
c
      stop
      end
c
c a simple function used to test the bisection method
c ---------------------------------------------------
c
      real function f(x)
      real pi
      pi = 3.141592653589793
      f = exp(x) - pi
      return
      end
```

Since the function is continuous on the given interval and

$$f(-2.2) \times f(6.8) < 0$$

the Bisection method will produce an approximate solution that is accurate up to machine precision. The relationship we need for analyzing convergence involve the current error and the previous error. The code above produces five columns of output. The first column is the iteration index, the second is the previous error and the third is the current error. The output looks like the following. The last two columns will be used below after performing a transformation on the error data.
The code produces the following output

```
Iter:     ek               ekp1           log(ek)          log(ekp1)
--------------------------------------------------------------------
   1   9.00000000       4.50000000      0.954242527      0.653212488
   2   4.50000000       2.25000000      0.653212488      0.352182508
   3   2.25000000       1.12500012      0.352182508       5.11525720E-02
   4   1.12500012      0.562500060       5.11525720E-02 -0.249877438
   5  0.562500060      0.281250000     -0.249877438     -0.550907493
   6  0.281250000      0.140625000     -0.550907493     -0.851937473
   7  0.140625000       7.03125000E-02 -0.851937473      -1.15296745
   8   7.03125000E-02   3.51562500E-02  -1.15296745      -1.45399749
   9   3.51562500E-02   1.75781250E-02  -1.45399749      -1.75502753
  10   1.75781250E-02   8.78906250E-03  -1.75502753      -2.05605745
  11   8.78906250E-03   4.39453125E-03  -2.05605745      -2.35708737
  12   4.39453125E-03   2.19726562E-03  -2.35708737      -2.65811753
  13   2.19726562E-03   1.09863281E-03  -2.65811753      -2.95914745
  14   1.09863281E-03   5.49316406E-04  -2.95914745      -3.26017737
  15   5.49316406E-04   2.74658203E-04  -3.26017737      -3.56120753
```

```
    16   2.74658203E-04   1.37329102E-04  -3.56120753      -3.86223745
    17   1.37329102E-04   6.86645508E-05  -3.86223745      -4.16326761
    18   6.86645508E-05   3.43322754E-05  -4.16326761      -4.46429729
    19   3.43322754E-05   1.71661377E-05  -4.46429729      -4.76532745
    20   1.71661377E-05   8.58306885E-06  -4.76532745      -5.06635761
    21   8.58306885E-06   4.29153442E-06  -5.06635761      -5.36738729
    22   4.29153442E-06   2.14576721E-06  -5.36738729      -5.66841745
    23   2.14576721E-06   1.07288361E-06  -5.66841745      -5.96944761
    24   1.07288361E-06   4.76837158E-07  -5.96944761      -6.32163000
    25   4.76837158E-07   2.38418579E-07  -6.32163000      -6.62265968

shift constant:     0.741362631
rate of convergence:      1.00143242
```

It is easy to see that the error is getting smaller and as shown in the size of the error appears to be headed to zero. With roundoff error, we already know that the computational error, e_k, is actually bounded below. Another question to be answered is how fast is the convergence?

Errors and Rate of Convergence: Analysis of the Output Data

As shown in the output from the code in the previous section show a rate of convergence. The next step is to determine a way to analyze the results. The relationship between successive steps is given by

$$|e_{k+1}| \leq C|e_k|^r$$

It should be noted that in the data from the example, there are 25 conditions on the rate relationship. That is,

$$\begin{array}{lcl} |e_1| \leq C\ |e_0|^r & \rightarrow & 9.00000000 \leq C(4.50000000)^r \\ |e_2| \leq C\ |e_1|^r & \rightarrow & 4.50000000 \leq C(2.25000000)^r \\ |e_3| \leq C\ |e_1|^r & \rightarrow & 2.25000000 \leq C(1.25000000)^r \\ & \vdots & \\ |e_{25}| \leq C\ |e_{24}|^r & \rightarrow & 4.76837158E-07 \leq C(2.38418579E-07)^r \end{array}$$

The goal is to compute values for C and r that define the relationship. So, the input and output for the relationship are the values for e_k and e_{k+1} are known and we want to obtain C and r to complete the definition of the error relationship. For the example above, this means we have 25 conditions for the two unknowns, C and r. This is a classic over-determined system. Due to roundoff error and other machine precision issues, the best we can do is fit the errors data to the parameters C and r.

Errors and Rate of Convergence: Apply A Log-log Transform

The first step in obtaining a fit is to transform the relationship using properties of logarithms. So,

$$|e_{k+1}| = C|e_k|^r$$

becomes

$$log(|e_{k+1}|) = log(C|e_k|^r) = log(C) + log(|e_k|^r) = log(C) + r\ log(|e_k|) = a + r\ log(|e_k|)$$

The logarithmic transformation turns the equality into a linear polynomial in the two unkbown parameters r and $a = log(C)$. This brings us back to the output from the code for the Bisection method. The last two columns are the logarithmic values we need in the transformed error relationship. So, we can write

$$\begin{array}{lcl} log(|e_1|) = a + r\ log(|e_0|) & \rightarrow & (0.954242527) = a + r\ (0.653212488) \\ log(|e_2|) = a + rlog(|e_1|) & \rightarrow & (0.653212488) = a + r\ (0.352182508) \\ |e_3| \leq C\ |e_1|^r & \rightarrow & (0.352182508) = a + r\ (5.11525720E-02) \\ & \vdots & \\ |e_{25}| \leq C\ |e_{24}|^r & \rightarrow & (-6.32163000) = a + r\ (-6.62265968) \end{array}$$

We can write the conditions from the last equations in a matrix form as follows:

$$\begin{bmatrix} 1.0 & 0.653212488 \\ 1.0 & 0.352182508 \\ 1.0 & 5.11525720E-02 \\ \vdots & \vdots \\ 1.0 & -6.62265968 \end{bmatrix} \begin{bmatrix} a \\ r \end{bmatrix} = \begin{bmatrix} 0.954242527 \\ 0.653212488 \\ 0.352182508 \\ \vdots \\ -6.32163000 \end{bmatrix} \quad (1)$$

This system is over-determined. We are trying to determine two parameters, a and r using 25 constraints. It is highly unlikely when roundoff and other errors effect these equations to expect to obtain a system with a unique solution. We can, however, come up with the next best thing. That is, we can project the problem into a space where a unique solution exists. This process goes by the official name of linear regression.
A shorthand symbolic form of our system of equations can be written as the matrix equation

$$A\ x = b$$

The normal equations for this system of equations is obtained by multiplying both sides of the equation by the transpose of the matrix. That is,

$$A^T\ A\mathbf{x} = A^T\ b$$

Note that for our problem the result of this projection into the column space of the matrix, A, is a 2×2 linear system of equations. Note that the coefficient matrix is symmetric and is positive definite is the two columns of A are independent of each other.
The matrix equation can be written in the form

$$A^T\ A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$$

where $a_{21} = a_{12}$. A closed formula for the inverse of the matrix can be written down as follows.

$$((A^T\ A) = \frac{1}{(a_{11}a_{22} - a_{12}a_{21})} \begin{bmatrix} a_{22} & -a_{21} \\ -a_{12} & a_{11} \end{bmatrix}$$

The details of these calculations are embedded in the code segment given above. The result of the computations in the code and as shown in the output from the code are the following.

```
shift constant:           0.74136263
rate of convergence:      1.00143242
```

The shift constant is the constant, C, in the error reduction from one step to the next. It should be noted that the computational convergence rate for the bisection method is about one - indicating the convergence is linear.

Errors and Rate of Convergence: Summary of Steps to Compute Computational Convergence Rates

The steps needed to compute the computational convergence for the case when a solution is known are the following.

1. Write a code and test the code to make sure that a solution is reached.
2. Embed a computation in the working code to determine the absolute error in the approximation using the and store this error in an array.
3. Compute the log-log transform of the error at successive steps.
4. Perform a linear regression of the error data to a linear polynomial based on the transformed relationship between errors at successive iterations of the algorithm.

Note that these instructions can be applied to just about any numerical algorithm. Any competent computational scienttist should have a self-contained code for computing convergence rates. In the code given above all the steps are embedded in the computer code. If there is a need for this type of code in any other algorithm, the least squares approach would need to be recoded in terms of the new application or coding environment.
As an example, consider the following method developed for the Java coding language.

```
public double convergenceRate(double [] errArray)
{
  //
  // test to see if there is anything in the error array
  // --------------------------------------------------
  //
  if(errArray == null) return -1.0;
  //
  // initialize the symmetric 2x2 matrix
  // ----------------------------------
  //
  double a11 = ((double) n);
  double a12 = 0.0;
  double a22 = 0.0;
  double b1 = 0.0;
  double b2 = 0.0;
  //
  // get the number of array values
  // ------------------------------
  //
  int n = errArray.length;
  //
  // loop over pairs of errors to do the computation
  // ----------------------------------------------
  //
  for(int i=0; i<(n-1); i++) {
    //
    // get the two successive errors
    // ----------------------------
    //
    double ek = errArray[i];
    double ekp1 = errArray[i+1];
```

```
        //
        // update the matrix entries
        // ------------------------
        //
        a12 = a12 + alog10(ek)
        a22 = a22 + alog10(ek) * alog10(ek)
        //
        // update the right side entries
        // ----------------------------
        //
        b1 = b1 + alog10(ekp1)
        b2 = b2 + alog10(ek) * alog10(ekp1)
        //
      }
      //
      // compute the action of the inverse of the 2x2 on the right hand side
      // -------------------------------------------------------------------
      //
      double detval = a11 * a22 - a12 * a12;
      double aval = ( a22 * b1 - a12 * b2 ) / detval;
      double rval = ( a11 * b2 - a12 * b1 ) / detval;
      //
      // return the convergence rate
      // --------------------------
      //
      return rval;
      //
    }
```

In a Java program or jar file, it might be a good idea to include this in a package with its own Application Programming Interface (API). The code can be translated or modified as needed. In object oriented languages (like Java) a programming might overload the definition of the method to possibly return the shift constant, $log(C)$. As a last note on the code above, the method in Java was harvested and translated out of the Bisection method code. All you need to do is have a couple of terminal windows and a cut and paste feature on your desktop. Now on to a more realistic treatment of computational convergence rate.

Errors and Rate of Convergence: Case 2. An Exact Solution is Not Known

In most problems, we will not know the exact solution and may have absolutely no idea where the solution resides in our mathematical system. One way to get an estimate of the convergence rate of an algorithm implemented into computer code involves the following steps.

1. Given that a code is working properly, run the code until an approximate solution is obtained or at least several approximations have been compouted.
2. If n approximations are collected, use the last approximation as the exact value.
3. Then compute the rate using the first $n-1$ approximations.

Let's consider a modification of the self-contained method written in Java in the last section. Instead of passing in the errors at each step, the new code will pass in the approximations obtained in an array.

```
public double convergenceRate(double [] approxArray, double exact)
{
  //
  // test to see if there is anything in the error array
  // ---------------------------------------------------
  //
  if(errArray == null) return -1.0;
  //
  // initialize the symmetric 2x2 matrix
  // -----------------------------------
  //
  double a11 = ((double) n);
  double a12 = 0.0;
  double a22 = 0.0;
  double b1 = 0.0;
  double b2 = 0.0;
  //
  // get the number of array values and compute the errors between the
  // approximations and the last array element
  // -----------------------------------------
  //
  int n = approxArray.length;
  double [] errArray = new double[n-1];
  for(int i=0; i<(n-2); i++) {
     errArray[i] = Math.abs(exact - approxArray[i]);
  }
  //
  // overload the method to use the code in the previous case where the
  // exact value was known
  // ---------------------
  //
  return convergenceRate(errArray);
  //
}
```

The second incarnation of the method actually uses the first through overloading the definition of the method. The second version uses a value specified in the second argument that will serve as the solution. The code computes an error for each of the approximations based on the inputs and then uses the previous version to return the computational convergence rate.
So, why does this work? The justification comes from a simple application of the triangle inequality. The erro between an approximation, x, and the exact value, x^*, can be written

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

using a cleverly chosen form of the number zero. Next, we can apply the triangle inequality to obtain

$$e_k \leq |x_k - x_n| + |x_n - x^*|$$

If the algorithm appears to be converging, then we would expect that

$$\lim_{n \to \infty} |x_n - x^*| = 0$$

regardless of how fast this is happening. So, we can use

$$e_k \leq |x_k - x_n|$$

as an approximation of the successive approximations compared to the exact value. In matheamtical terms we say that the second term is negligible. The second version of the convergenceRate() method can be used for either of the cases.

Errors and Rate of Convergence: An Example When the Exact Value is Unknown

We can use the Bisection method example earlier in this section. Suppose that we do not have an exact value for the root in the example problem. We can consider the first 15 iterations in the example as approximations and the 16^{th} as a good approximation as the exact value. A modified version of the original Bisection code in our example can be created that (1) computes a total of 16 approximations, (2) sets $x^* = x_{16}$, (3) computes the approximate error between the first 15 approximations abd the very last approximation.

```
      program main
c
c variables for the test
c ----------------------
c
      real a, b, c(1000), pi, tol
      real ek(1000)
      real a11, a12, a22, b1, b2
      real aval, rval
      real detval
      integer n
c
c initialize the interval and tolerance desired
c ---------------------------------------------
c
      a = -2.2
      b = 6.8
      tol = 0.0000001
c
c compute the number of bisections needed for the given tolerance
c ---------------------------------------------------------------
c
      n = 16
c
c do the bisection iterations needed
c ----------------------------------
c
      do 1 i=1, n
         c(i) = 0.5 * ( a + b )
         if(f(a)*f(c(i)) .lt. 0.0) then
            b = c(i)
         else
            a = c(i)
         endif
    1 continue
c
c print out a header for the output
c ---------------------------------
c
      print *," Iter:     ek    ekp1    log(ek)     log(ekp1)"
      print *," --------------------------------------------"
c
c initialize storage for the linear regression to determine the computational
c rate of convergence - note that the normal equations matrix is symmetric.
```

```
c -------------------------------------------------------------------------
c
      a11 = dble(n-1)
      a12 = 0.0
      a22 = 0.0
      b1 = 0.0
      b2 = 0.0
c
c loop over the approximations computed computing the approximate errors
c ----------------------------------------------------------------------
c
      do 2 i=1,n-1
         ek(i) = abs( c(16) - c(i) )
    2 continue
      do 3 i=1,n-2
c
c print out the next line in the table of errors
c ----------------------------------------------
c
         print *, i, ek(i), ek(i+1), alog10(ek(i)), alog10(ek(i+1))
c
c update the error variables in the normal equations
c --------------------------------------------------
c
         a12 = a12 + alog10(ek(i))
         a22 = a22 + alog10(ek(i)) * alog10(ek(i))
         b1 = b1 + alog10(ek(i+1))
         b2 = b2 + alog10(ek(i)) * alog10(ek(i+1))
c
    3 continue
c
c compute the solution using the inverse of the 2x2 matrix for the linear
c reduction of the matrix
c ----------------------
c
      detval = a11 * a22 - a12 * a12
      aval = ( a22 * b1 - a12 * b2 ) / detval
      rval = ( a11 * b2 - a12 * b1 ) / detval
c
c output the results - note that the constant needs to be exponentiated
c ---------------------------------------------------------------------
c
      print *, "shift constant:   ", exp(aval)
      print *, "rate of convergence:   ", rval
c
c this code will be used to test convergence rates for the bisection method
c -------------------------------------------------------------------------
c
      stop
      end
c
c a simple function used to test the bisection method
c ---------------------------------------------------
c
```

```
      real function f(x)
      real pi
      pi = 3.141592653589793
      f = exp(x) - pi
      return
      end
```

```
 Iter:     ek              ekp1             log(ek)       log(ekp1)
 -------------------------------------------------------------------
   1   1.15534973       1.09465039       6.27134740E-02   3.92754339E-02
   2   1.09465039       3.03497314E-02   3.92754339E-02  -1.51784515
   3   3.03497314E-02  0.532150328      -1.51784515     -0.273965687
   4  0.532150328      0.250900269     -0.273965687     -0.600498915
   5  0.250900269      0.110275269     -0.600498915     -0.957521856
   6  0.110275269       3.99627686E-02 -0.957521856      -1.39834440
   7   3.99627686E-02   4.80651855E-03  -1.39834440      -2.31816936
   8   4.80651855E-03   1.27716064E-02  -2.31816936      -1.89375448
   9   1.27716064E-02   3.98254395E-03  -1.89375448      -2.39983940
  10   3.98254395E-03   4.11987305E-04  -2.39983940      -3.38511610
  11   4.11987305E-04   1.78527832E-03  -3.38511610      -2.74829412
  12   1.78527832E-03   6.86645508E-04  -2.74829412      -3.16326737
  13   6.86645508E-04   1.37329102E-04  -3.16326737      -3.86223745
  14   1.37329102E-04   1.37329102E-04  -3.86223745      -3.86223745

shift constant:     0.639538586
rate of convergence:     0.886137068
```

There are a few things to notice in the output. First, the computational convergence rate given in the output is less than 1. This is due to the fact that the approximations have not converged and we will always end up with an approximation. It would be a good idea to take more iterations before computing the convergence rate.
The modified code in this example uses the midpoint of the interval containing a root to define the error. The original code used the interval length as an upper bound on the error in the approximation. This will give a different approximation of the errors in the approximations.

Errors and Rate of Convergence: Error Reduction per Iteration Perturbation

There are times when it is important to know about how much the error has been reduced from one iterate to the next. A typical example in root finding problems involves using two different methods, say the Bisection method to start the root finding and then switch to Newton's method once the interval is small enough. These are called hybrid methods and are handled in another part of these notes. If we can estimate the reduction in error at each step, we can estimate the number of steps needed to meet a given tolerance in the error. The Bisection method is very easily analyzed in this setting.

Suppose we want to reduce the error by one order of magnitude using Bisection. This means that we would need to take some number of steps using the Bisection method to achieve the reduction. We can assume that assumptions for successful application of the Bisection method are satisfied. The inequality we need is

$$|e_{k+p}| = \left(\frac{1}{2}\right)^p |e_k| \leq 10^{-1}$$

The last expression states that we need to find p that makes this true - that is, how many iterations are required to make sure the error is reduced by one decimal digit. Recall that computers work in binary arithmetic. Rewriting and using logarithms we find

$$log_{10}\left(\frac{1}{2}\right)^p) \leq log_{10}(10^{-1}) = -1$$

Solving for p gives

$$p = \frac{-1}{log_{10}(\frac{1}{2})} = \frac{1}{log_{10}(2)} \approx 4$$

This states that in order to reduce the error in the sequence of approximations provided by the Bisection method, we need to take just under 4 iterations. Since the iteration counter is an integer, we will need to round up to 4 iterations.

Consider again the data generated by the Bisection method in the first example and look at the iterations from $k = 8$ through $k = 12$ in the second column of output.

```
  8   7.03125000E-02   3.51562500E-02  -1.15296745      -1.45399749
  9   3.51562500E-02   1.75781250E-02  -1.45399749      -1.75502753
 10   1.75781250E-02   8.78906250E-03  -1.75502753      -2.05605745
 11   8.78906250E-03   4.39453125E-03  -2.05605745      -2.35708737
 12   4.39453125E-03   2.19726562E-03  -2.35708737      -2.65811753
```

Notice that for $k = 11$ the error is not quite one order of magnitude less than the error for $k = 8$. However, the difference in errors between $k = 8$ and $k = 12$ is more than one order of magnitude in reduction of the error.

The methods used in the reduction of error can typically be applied aposteriori meaning after the fact. For example, for Newton's method, there are ways to analytically compute the convergence rate and thus the error reduction in each step as long as the iterates are sufficiently close to the root in question. However, until the iterates are close enough there is no guarantee as to how much the error has been reduced in one step. Recall that for Newton's method

$$|e_{k+1}| \leq C|e_k|^2$$

where C depends on the second derivative of the function at a root that has been located. The dependence can be determined using a Taylor series expansion. If the second derivative is large, the reduction per iteration may be affected by the constant, C. As the iterates get closer to a root, Newton's method will become more predicatble in terms of the error reduction. Note that in most cases, Newton's method is very fast.