Roots of Equations and the Bisection Method

Leigh J. Little SUNY Brockport October 22, 2018

Contents

1 A More Universal Definition of the Problem 1

2 Bisection Method 2
2.1 Deciding Which Interval to Keep . . . 2
2.2 How to Stop? . . . 3
2.3 A Working Algorithm . . . 4
2.4 Improving the Algorithm . . . 5
2.4.1 Counting the Steps . . . 5
2.4.2 Cap on the Number of Steps . . . 5
2.4.3 Status Indicator . . . 5
2.4.4 Recording the Progress of the Algorithm . . . 6
2.4.5 A Better Test for Which Interval to Keep . . . 6
2.4.6 Evaluating $f(x)$ in its Own Function . . . 6
2.4.7 Writing the Whole Algorithm as a Function . . . 7
2.4.8 Doing Too Much Work . . . 7

1 A More Universal Definition of the Problem

From the last discussion, we have the equations

$$\begin{aligned} \cos x &= x \\ ye^y &= 7 \\ t^t &= \sqrt{2} \\ x^5 - 4x^4 + 3x^3 - 2x^2 + x - 1 &= 0 \end{aligned}$$

These equations have no analytical (*i.e.,* paper and pencil) solutions, yet solutions do exist. Before proceeding to methods for generating approximate solutions to equations such as these, we need to devise a slightly better statement of the problem.

Each of the equations above can be written in the form

$$f(x) = 0.$$

For example,

$$\begin{aligned} \cos x &= x &\rightarrow\quad & f(x) = \cos x - x = 0 \\ ye^y &= 7 &\rightarrow\quad & f(y) = ye^y - 7 = 0 \\ t^t &= \sqrt{2} &\rightarrow\quad & f(t) = t^t - \sqrt{2} = 0 \end{aligned}$$

Thus, we can express the problem we are trying to solve as:

Find one or more values of x satisfying $f(x) = 0$.

Values of x that satisfy this equation are called the *roots* of the equation.

Such problems are known as root-finding problems and the methods used to compute approximation solutions are root-finding methods. All such methods are part of a larger class of methods known as *iterative methods*.

An iterative method is essentially intelligent trial and error. Starting from some initial guess for the solution, carry out a series of steps that modifies this initial guess and (hopefully) produces a better guess. This series of steps is repeated with the new guess until the method produces the desired level of accuracy.

An unfortunate characteristic of iterative methods is that they are not guaranteed to work, meaning instead of calculating a series of values that get closer to the solution, the computed values can either diverge to $\pm\infty$ or oscillate between values.

2 Bisection Method

We have already seen how the bisection method works.

```
Given f(x) and some interval [a,b] containing a single (or root) x = c
    such that f(c) = 0

*  Compute the midpoint c of [a,b]
     c = (a+b)/2
     This creates 2 intervals:  [a,c] and [c,b].
     The root has to lie in one of these
   Decide which half to keep
     If keeping [a,c], set b = c, goto *
     If keeping [c,b], set a = c, goto *
```

This is a simple looping operation. It modifies the original interval $[a, b]$ into a new, smaller interval $[a, b]$. This new interval also contains the desired root. This needs to be translated into a working MATLAB program.

2.1 Deciding Which Interval to Keep

In order to decide which interval to keep and which to discard, information about the function needs to be incorporated. For example, consider Figure 1. From the graph, if can be seen that the function crosses the

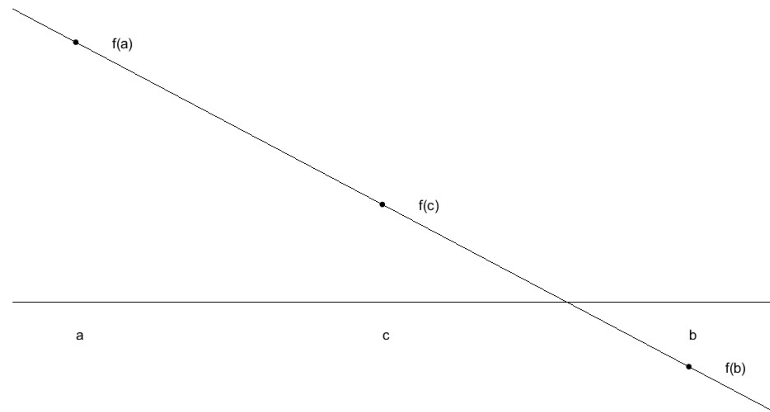

Figure 1: Function that crosses the x-axis in $[c, b]$ in a positive to negative sense.

x-axis in the interval $[c, b]$. Using the corresponding function values, it can be seen the interval in which two function values have the same sign should be discarded. However, the function can have the same sign in a positive sense like in Figure 1 or a negative sense as shown in Figure 2. This leads to the test

```
Define:  fa = f(a), fb = f(b), fc = f(c)
if((fa > 0 & fc > 0) | (fa < 0 & fc < 0))
    a = c;     % f doesnt change sign in first interval, so keep second
else
    b = c;     % f doesnt change sign in second interval, so keep first
```

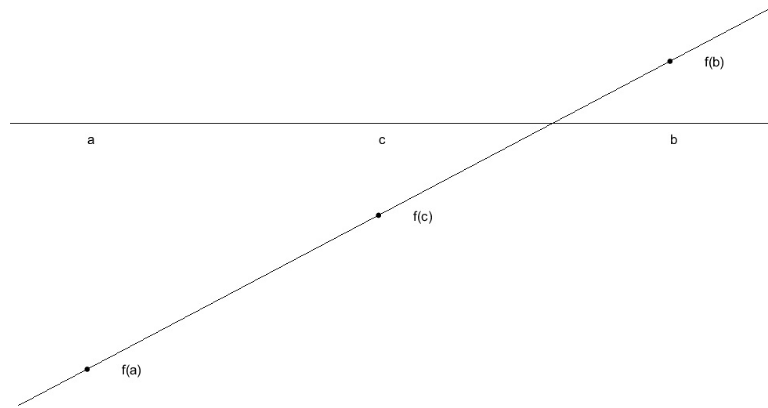

Figure 2: Function that crosses the x-axis in $[c, b]$ in a negative to positive sense.

```
end
```

Note that the parentheses in the `if` clause are important, otherwise the conjunction operators will not be interpreted in the correct way.

2.2 How to Stop?

In its current form, the algorithm is an infinite loop. A criteria for terminating the loop needs to be devised. There are two approaches that can be used to terminate the algorithm.

- Interval criteria - The bisection method produces a series of intervals that get smaller by a factor of 2 each pass through the loop. The solution to the equation is guaranteed to be in each interval. Thus, we can stop the process when a and b are sufficiently close. In particular, we can stop when

$$\text{Relative Distance} = \frac{|a - b|}{|a| + |b|} \leq \text{TOL}$$

  where TOL is some predefined tolerance level. The relative distance between a and b is essentially the same as the relative error between these values, however the formula is modified slightly to account for the possibility that a and b might have different signs or one of a and b might be zero.

- Function criteria - The ultimate goal of the bisesction method is to compute a value c satisfying $f(c) = 0$. Thus, we can stop the process when

$$|f(c)| \leq \text{TOL}.$$

It would seem that either of these criteria could be used to stop the process, however, consider Figure 3. The slope of $f(x)$ is nearly zero in the vicinity of the desired root. What this means is that a test based on the magnitude of the function can trigger the end of the process while a and b are still relatively far apart.

Similarly, consider Figure 4. The slope of $f(x)$ is nearly infinity in the vicinity of the desired root. What this means is that a test based on the relative distance between a and b can trigger, but he magniture of the function can still be relativly large.

In order to build an algorithm that avoids these problems, it is necessary to test both criteria. We will use the following test:

```
err = abs(a-b)/(abs(a)+abs(b));
if (err < tol1 & abs(fc) < tol2)
   clean up and stop
endif
```

The phrase `clean up` means do any final calculations or steps than need to be performed before exiting the main bisection loop.

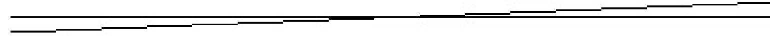

Figure 3: Function with f'(x) nearly zero in the vicinity of the root.

2.3 A Working Algorithm

Combining the parts above, it is not difficult to arrive at a working algorithm for implementing the bisection method.

```
Given:  A continuous f(x) on [a,b] with exactly one c in [a,b] such that f(c) = 0.
        Values for tol1, tol2
% This example solves t^t - 3 = 0

tol1 = 1.0e-8;
tol2 = 1.0e-8;
a = 0.4;
b = 2;
flag = 0;
while(flag == 0)
  fa = a^a - 3;
  fb = b^b - 3;
  c = (a+b)/2;
  fc = c^c - 3;
  if ((fa < 0 & fc < 0) | (fa > 0 & fc > 0))
     b = c;
  else
     a = c;
  end
  err = abs(a-b)/(abs(a)+abs(b));
  if(err <= tol1 & abs(fc) <= tol2)
     flag = 1;
  end
end
```

Saving this in a script called `testbis.m` and executing it gives

```
>> format long e
>> testbis
>> c
c =
     1.825455021858216e+00
>> fc
fc =
    -5.125603053102168e-09
```

The value of c is accurate to at least 7 digits because the value of `tol1` is $1.0e(-8) = 0.1e(-7)$.

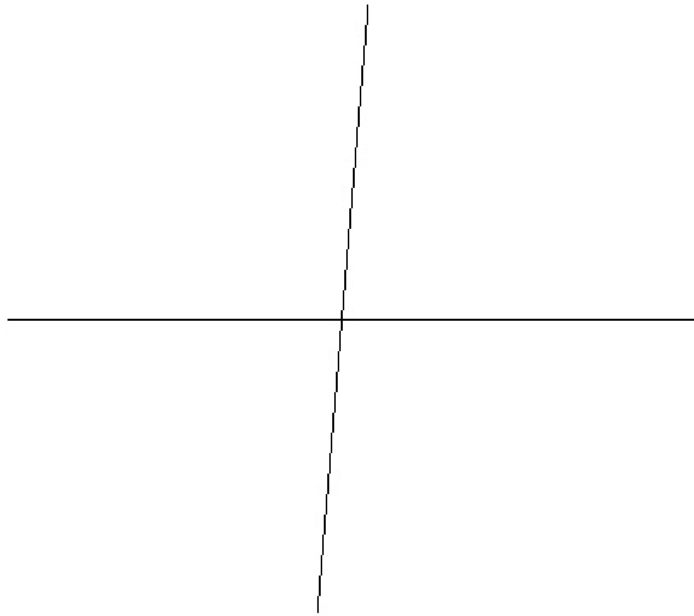

Figure 4: Function with f'(x) nearly infinity in the vicinity of the root.

2.4 Improving the Algorithm

The initial goal of this discussion was to obtain a working bisection algorithm. However, the algorithm given in the previous section has several problems. Some of these are mechanical or diagnostic in nature while other problems relate to ease of use, amount of effort and logic simplification.

2.4.1 Counting the Steps

One important piece of information to include in the algorithm is a count of the number of times the loop must execute before reaching converenge. This is typically stored in a variable called `its` (short for iterations).

2.4.2 Cap on the Number of Steps

The bisection method is an iterative method. All iterative methods should have a cap on the maximum number of steps allowed. This is to prevent the algorithm from going into an inifinite loop due to pathelogical rounding error cases. This is an input variable and is typically called `maxits`. is typically called

2.4.3 Status Indicator

Another imporant piece of information would be a variable that indicates if the algorithm converges or if it reached the maximum allowed number of steps without converging. This type of variable is a status indicator. For example, we could set

```
stat = 0;
```

if the algorithm terminates properly and

```
stat = 1;
```

if the algorithm doesn't converge within `maxits` iterations.

2.4.4 Recording the Progress of the Algorithm

It is often helpful to keep a record of how the algorithm is progressing. This can be done by storing the value of `err` and `abs(f(c))` in arrays. These arrays can be plotted when the algorithm terminates to see how much progress towards the solution is being made at each step.

2.4.5 A Better Test for Which Interval to Keep

All of the above improvements relate to diagnostic information. They provide additional information about how the algorithm performs. However, there are some changes to the algorithm that relate to computational efficiency or robustness.

The current test that is used to determine which interval to keep is overly complex. A simpler test to use would be

```
if(fa*fc > 0)
   a = c;
else
   b = c;
end
```

However, this can fail due to *overflow* if `fa` and `fc` are extremely large. Overflow occurs whenever a number gets too large to store on the computer. In this case, the numerical value of `fa*fc` would be replaced by `NaN`. The `if` test would have no meaning and would never be true. Similarly, if these valuese are extremely small, an underflow condition can occur. This happens when the number is too small to store, in which `fa*fc` would again be a `NaN`.

A better choice would be to use the *signum* or sign function. This function is defined as

$$\text{sign}(x) = \begin{cases} 1 & \text{if} \quad x > 0 \\ 0 & \text{if} \quad x = 0 \\ -1 & \text{if} \quad x < 0 \end{cases}$$

This will prevent `fa*fc` from overflowing or underflowing. The new test would look like

```
if(sign(fa)*sign(fc) > 0)
   a = c;
else
   b = c;
end
```

2.4.6 Evaluating $f(x)$ in its Own Function

In its current form the algorithm requires $f(x)$ to be evaluated three times each pass through the loop. Everytime the problem is changed, the function must be changed. This can result in reliability problems. Consider the third problem you had to do for homework. This is a long formula that presents many opportunities for sign errors, missing terms, *etc.*. You might type in this formula correctly for two of the three times you need it, but then make an error the third time.

A better way to evaluate the function would be to store $f(x)$ in its own function file instead of hardcoding the function into the main loop. For example, you could enter the statements below into a file named `myfun.m`

```
function y = myfun(t)
y = t^t - 3;
```

When you need to compute function values in the main algorithm, you can do

```
fa = myfun(a);
fb = myfun(b);
fc = myfun(c);
```

whenever you need to evaluate $f(x)$.

2.4.7 Writing the Whole Algorithm as a Function

Once the algorithm has been cleaned up, it should be itself written as a function. This notion is guided by good programming practice. Once you have an algorithm that works and does what you want, you should never touch it again unless you need to change what it does or how it does it.

Right now, changing the problem requires changing the values of `a` and `b` in addition to possibly changing `maxit` and the tolerances. It is very easy to accidentaly change other parts of the program when changing these values (I have done this type of thing many times myself).

Fortunately, it is not difficult to write the function. All we need to do is determine what the inputs and outputs of the algorithm are, then wrap up the code in a function syntax. For example, you can do something like

```
function [c,its,stat,histerr,histfc] = bisec(a,b,tol1,tol2,maxits)
    ..... algorithm goes here
    ..... except for statements that assign values to
    ..... a,b,tol1,tol2,maxits
```

Put this framework in a file names `bisec.m`. The routine can now be called like any other MATLAB function.

The values of `a`, `b`, `tol1`, `tol2` and `maxits` would be assigned in the main MATLAB workspace, then sent to the function as inputs. When the algorithm finishes, the values of `c`, `its`, `stat` and the history arrays are returned as outputs back to the main MATLAB workspace.

2.4.8 Doing Too Much Work

A drawback of the routine we have developed is that is does far more computational work than is necessary. The *complexity* of an algorithm is a measure of how many calculations are needed for the routine to do its job. If we examine the routine, it would appear that there is not much work being done. Each pass through the loop, we evaluate the function 3 times and compute the value of c.

Unfortunately 'evalute the function' can be a misleading statement. When you hear a phrase like this, you normally think of functions like the ones at the top of this discussion; *i.e.,* functions that are inexpensive to evaluate. However, it is not uncommon to encounter functions that can take 2-3 minutes or 2-3 days of computer time to evaluate.

This can be remedied by looking carefully at the algorithm. It turns out that we only need to evaluate the function once per pass through the loop provided we re-use function evaluations when feasible. In particular, we can do the following:

- Evaluate `fa` and `fb` outside the main loop.
- When we reset the endpoints of the interval, we also reset the function values using the current value of `fc`.

The resulting key steps would look like

```
function .......
......
fa = myfun(a);
fb = myfun(b);
while ......
    .....
    fc = myfun(c);
    .....
    .....
    if(sign(fa)*sign(fc) > 0)
       a = c;
      fa = fc;
    else
       b = c;
```

```
        fb = fc;
      end
      .....
  end
```

This version of the routine will produce the same output, but will only require `its+2` function evaluations instead of `3*its` evaluations.