

# Lecture 5

## The Bisection Method and Locating Roots

### Bisecting and the `if ... else ... end` statement

Recall the bisection method. Suppose that  $c = f(a) < 0$  and  $d = f(b) > 0$ . If  $f$  is continuous, then obviously it must be zero at some  $x^*$  between  $a$  and  $b$ . The bisection method then consists of looking half way between  $a$  and  $b$  for the zero of  $f$ , i.e. let  $x = (a + b)/2$  and evaluate  $y = f(x)$ . Unless this is zero, then from the signs of  $c$ ,  $d$  and  $y$  we can decide which new interval to subdivide. In particular, if  $c$  and  $y$  have the same sign, then  $[x, b]$  should be the new interval, but if  $c$  and  $y$  have different signs, then  $[a, x]$  should be the new interval. (See Figure 5.1.)

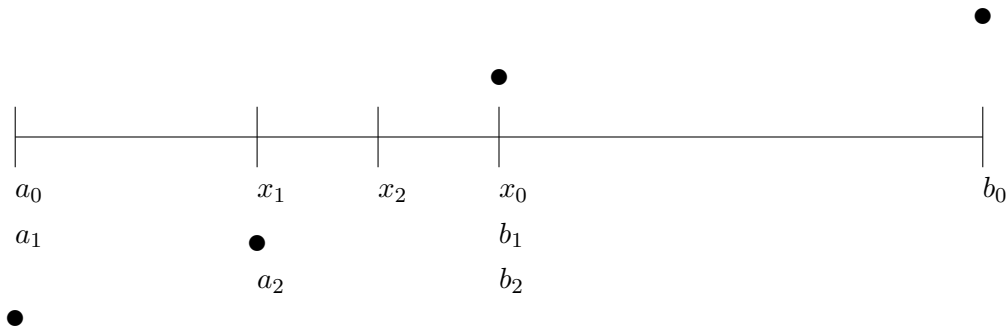


Figure 5.1: The bisection method.

Deciding to do different things in different situations in a program is called *flow control*. The most common way to do this is the `if ... else ... end` statement which is an extension of the `if ... end` statement we have used already.

### Bounding the Error

One good thing about the bisection method, that we don't have with Newton's method, is that we always know that the actual solution  $x^*$  is inside the current interval  $[a, b]$ , since  $f(a)$  and  $f(b)$  have different signs. This allows us to be sure about what the maximum error can be. Precisely, the error is always less than half of the length of the current interval  $[a, b]$ , i.e.

$$\{\text{Absolute Error}\} = |x - x^*| < (b - a)/2,$$

where  $x$  is the center point between the current  $a$  and  $b$ .

The following function program (available to download as `mybisection.m`) does  $n$  iterations of the bisection method and returns not only the final value, but also the maximum possible error:

```
function [x e] = mybisection(f,a,b,n)
% function [x e] = mybisection(f,a,b,n)
% Does n iterations of the bisection method for a function f
% Inputs: f -- a function
%         a,b -- left and right edges of the interval
%         n -- the number of bisections to do.
% Outputs: x -- the estimated solution of f(x) = 0
%         e -- an upper bound on the error

% evaluate at the ends and make sure there is a sign change
c = f(a); d = f(b);
if c*d > 0.0
    error('Function has same sign at both endpoints.')
end
disp('          x          y')
for i = 1:n
    % find the middle and evaluate there
    x = (a + b)/2;
    y = f(x);
    disp([    x    y])
    if y == 0.0    % solved the equation exactly
        a = x;
        b = x;
        break    % jumps out of the for loop
    end
    % decide which half to keep, so that the signs at the ends differ
    if c*y < 0
        b=x;
    else
        a=x;
    end
end
% set the best estimate for x and the error bound
x = (a + b)/2;
e = (b-a)/2;
end
```

Another important aspect of bisection is that it always works. We saw that Newton's method can fail to converge to  $x^*$  if  $x_0$  is not close enough to  $x^*$ . In contrast, the current interval  $[a, b]$  will always be decreased by a factor of 2 at each step and so it will always eventually shrink down as small as you wish.

## Locating the roots (if any)

The bisection method and Newton's method are both used to obtain closer and closer approximations of a solution, but both require starting places. The bisection method requires two points  $a$  and  $b$  that have a root

between them, and Newton's method requires one point  $x_0$  which is reasonably close to a root. How do you come up with these starting points? It depends. If you are solving an equation once, then the best thing to do first is to just graph it. From an accurate graph you can see approximately where the graph crosses zero.

There are other situations where you are not just solving an equation once, but have to solve the same equation many times, but with different coefficients. This happens often when you are developing software for a specific application. In this situation the first thing you want to take advantage of is the natural domain of the problem, i.e. on what interval is a solution physically reasonable. If that is known, then it is easy to get close to the root by simply checking the sign of the function at a fixed number of points inside the interval. Whenever the sign changes from one point to the next, there is a root between those points. The following program will look for the roots of a function  $f$  on a specified interval  $[a_0, b_0]$ .

```
function [a,b] = myrootfind(f,a0,b0)
% function [a,b] = myrootfind(f,a0,b0)
% Looks for subintervals where the function changes sign
% Inputs: f -- a function
%         a0 -- the left edge of the domain
%         b0 -- the right edge of the domain
% Outputs: a -- an array, giving the left edges of subintervals
%           on which f changes sign
%         b -- an array, giving the right edges of the subintervals
n = 1001; % number of test points to use
a = []; % start empty array
b = [];
% split the interval into n-1 intervals and evaluate at the break points
x = linspace(a0,b0,n);
y = f(x);
% loop through the intervals
for i = 1:(n-1)
    if y(i)*y(i+1) <= 0 % The sign changed, record it
        a = [a x(i)];
        b = [b x(i+1)];
    end
end
if size(a,1) == 0
    warning('no roots were found')
end
end
```

To see this program in action try the following:

```
f = @(x) sin(x)-2*x^4+0.5
x = -1:.01:1;
y = f(x);
plot(x,y) % see that there are two roots
[a,b] = myrootfind(f,a0,b0) % observe that it finds two roots
```

The final situation is writing a program that will look for roots with no given information. This is a difficult problem and one that is not often encountered in actual applications.

Once a root has been located on an interval  $[a, b]$ , these  $a$  and  $b$  can serve as the beginning points for the bisection and secant methods (see the next section). For Newton's method one would want to choose  $x_0$  between  $a$  and  $b$ . One obvious choice would be to let  $x_0$  be the bisector of  $a$  and  $b$ , i.e.  $x_0 = (a + b)/2$ . An even better choice would be to use the secant method to choose  $x_0$ .

## Exercises

- 5.1 Modify `mybisect` to solve until the absolute error is bounded by a given tolerance. Use a `while` loop to do this. Run your program on the function  $f(x) = 2x^3 + 3x - 1$  with starting interval  $[0, 1]$  and a tolerance of  $10^{-8}$ . How many steps does the program use to achieve this tolerance? (You can count the steps by adding 1 to a counting variable `i` in the loop of the program.) How big is the final residual  $f(x)$ ? Turn in your program and a brief summary of the results.
- 5.2 Perform 3 iterations of the bisection method on the function  $f(x) = x^3 - 4$ , with starting interval  $[1, 3]$ . By hand, but use a calculator.) Calculate the errors and percentage errors of  $x_0$ ,  $x_1$ ,  $x_2$ , and  $x_3$ . Compare the errors with those in exercise 3.2.