5.2 Algorithms for Finding Zeros

In this section, we will discuss three introductory algorithms that can be used to find the zeros of a function. Although we will thoroughly discuss and explain each algorithm, we will leave the implementation to our readers.

We will begin the discussion by explaining how to pass a function to another function using function handles. We will also explore some minimal error checking and expand our knowledge of Matlab's **fprintf** command.

Passing Function Handles

We begin by defining a task.

▶ **Example 1.** *Write a function named* **signChange** *that will accept a function handle f and two integers a and b that define a search interval $[a, b]$. When the function is called, it will begin a search for two consecutive integers k and $k + 1$ within the interval $[a, b]$ until one of two things occurs:*

1. *The function changes sign on the interval $[k, k + 1]$; i.e., $f(k)$ and $f(k + 1)$ differ in sign (one plus, the other minus).*
2. *The function is zero at either k or $k + 1$.*

The function **signChange** *should print the current value of k and $f(k)$ until one of the two conditions is reached.*

First, define the function header with three inputs and no outputs.

```
function signChange(f,a,b)
```

Matlab's **nargin** command determines the number of arguments passed to the function by the user. For example, if the user calls the function with **signChange(f)**, then **nargin** equals 1. On the other hand, if the user calls the function with **signChange(f,-10)**, then **nargin** equals 2.

During development of the function, it would be nice if we didn't have to return to the command window to input a function handle and endpoints of the search interval. If we can somehow set these arguments in the function, then we can use the F5 function key to run the function from the editor. We could construct the following check for this purpose.

```
if nargin==0
    f=@(x) x^2-2*x-2;
    a=-10;
    b=10;
end
```

If we press F5 within the editor, the function is called with no arguments (no input). But our code snippet sees that the number of input arguments is zero, then sets a function handle to an anonymous function and establishes endpoints for the search interval. This code snippet will also be called if the user enters the following command at the Matlab prompt.

```
>> signChange
```

Let's push this code snippet a bit further. Suppose the user enters the following command at the Matlab prompt.

```
>> signChange(f)
```

There is only one input argument, so **nargin** equals 1. In this case, let's set the upper and lower bounds of the search interval for the user. We will choose $a = -10$ and $b = 10$ in this case.

```
if nargin==1
    a=-10;
    b=10;
end
```

Finally, suppose that the user enters the following command at the Matlab prompt.

```
>> signChange(f,-2)
```

There are two input arguments, so **nargin** equals 2. In this case, let's set the upper bound of the search interval for the user.

```
if nargin==2
    b=10;
end
```

A final version would use an **if..elseif..end** structure to include all three cases in one code snippet.

```
if nargin==0
    f=@(x) x^2-2*x-2;
    a=-10;
    b=10;
elseif nargin==1
    a=-10;
    b=10;
elseif nargin==2
    b=10;
end
```

We will next do some checking of user input. For example, we will insist that the endpoints of the search interval $[a, b]$ are integers and $a < b$.

```
if (floor(a)~=a) || (floor(b)~=b) || (a>=b)
    error('signChange(f,a,b): Inputs a and b must be distinct
    integers with a<b.')
end
```

There are several new constructs in this code snippet that warrant explanation.

i.) Matlab's **floor(a)** command finds the greatest integer that is less than or equal to the input number a. For example, **floor(2.3)=2**. If **floor(a)** does not equal a, then a is not an integer.

ii.) You will recall that the logical operator | means "or." The double bar is a "short circuit" form of the "or" operator. What this means is if the first part of the statement **expression || expression** evaluates as true, the "or" statment has to be true, so Matlab doesn't bother evaluating the expression to the right of the double bars. In this case, if a is not an integer (**floor(a)~=a**, then the remainder of the or statement is not evaluated because it has to be true regardless of whether the remaining expressions are true or false. However, if the first statement is false, then Matlab will evaluate the second

expression (checking if b is an integer), and again will cease evaluation if the expression is true. Similar comments are in order for the final expression. Using the "short circuit" operator instead of the usual "or" operator saves execution time.

iii.) Matlab's **error** command, in its simplest form, displays its input string and terminates the function. Thus, if either a or b are not integers, or if a is greater than or equal to b, Matlab will display the error message and terminate the function.

We next check to see if the first arguemnt is a proper function handle.

```
if ~isa(f,'function_handle')
    error('signChange(f,a,b): f must be a function handle.')
end
```

Matlab's **isa** is used to determine if an object is of a certain class[2]. In this case, if the argument f is not a function handle, then the **error** command prints the error message to the command window and terminates the function.

Next, we print some headers for the output produced by the function, specifically, k and $f(k)$.

```
fprintf('\n%15s%15s\n','k','f(k)')
```

Note that the format string **'\n%15s%15s\n'** has line returns at the front and end, surrounding two **%15s** conversion specifications. The **s** indicates a string replacement and **%15s** reserves 15 spaces, in which the replacement string will be right-justified.

Next, we set a starting value for k.

```
k=a;
```

Then we write the main loop. We use a **while** construct to halt execution of the loop if one of two events happnes: (1) we find a zero and $f(k) = 0$, or (2) k equals the terminating value of the search interval ($k = b$).

[2] for detailed help on Matlab's **isa** ("is a") command, type **doc isa** at the command prompt.

```
while (f(k)~=0) && k<b
    fprintf('%15d%15.6f\n', k, f(k));
    if f(k)*f(k+1)>0
        k=k+1;
    else
        k=k+1;
        break;
    end
end
fprintf('%15d%15.6f\n', k, f(k));
```

Some comments are definitely in order, for this is a bit of tricky code, to say the least.

i.) The command **fprintf('%15d%15.6f\n', k, f(k))** is used to print the current value of k and the function value at k, namely $f(k)$. Because k is an integer, we use the conversion specification **%15d** to create a field having wdith 15, in which the value of k is printed right-justified. The second conversion specification, **%15.6f** creates a field of width 15, which will hold the fixed point value of $f(k)$, right-justified, with 6 decimal places.

ii.) We then evaluate the product **f(k)*f(k+1)**. If this product is larger than zero, then two things are true: (1) neither endpoint of the interval $[k, k+1]$ is zero, and (2); the function has the same sign at each endpoint of the interval. Hence, we have not found a sign change or a zero, so we want to continue, which we do by incrementing k and returning to the head of the **while** loop. On the other hand, if the product is less than or equal to zero, we either experience a zero at $f(k+1)$ or a change of sign at $f(k+1)$. In either case, we increment k, then exit the loop with the **break** command.

iii.) Once out of the loop, we must print the current values of k and $f(k)$.

signChange.m

For convenience and clarity, we now produce the program in toto.

```
function signChange(f,a,b)

if nargin==0
    f=@(x) x^2-2*x-3;
    a=-10;
    b=10;
elseif nargin==1
    a=-10;
    b=10;
elseif nargin==2
    b=10;
end

if (floor(a)~=a) || (floor(b)~=b) || (a>=b)
    error('signChange(f,a,b): Inputs a and b must be distinct
    integers with a<b.')
end

if ~isa(f,'function_handle')
    error('signChange(f,a,b): f must be a function handle.')
end

fprintf('\n%15s%15s\n','k','f(k)')
k=a;
while (f(k)~=0) && k<b
    fprintf('%15d%15.6f\n', k, f(k));
    if f(k)*f(k+1)>0
        k=k+1;
    else
        k=k+1;
        break;
    end
end
fprintf('%15d%15.6f\n', k, f(k));

commandwindow
```

If we run the program from the editor by pressing the F5 key, the function receives no arguments, so **nargin** equals zero and defaults are set for both the function handle and the endpoints of the search interval (**f=@(x) x^2-2*x-3**, **a=-10**,

and **b=10**. This function has a zero at $x = -1$, which is evident in the output produced by the program.

```
     k          f(k)
   -10     117.000000
    -9      96.000000
    -8      77.000000
    -7      60.000000
    -6      45.000000
    -5      32.000000
    -4      21.000000
    -3      12.000000
    -2       5.000000
    -1       0.000000
```

If we return to the command window, we can create a new anonymous function then use the function **signChange** to search the interval $[-6, 6]$ for a zero or a change of sign.

```
>> f=@(x) x^2-2*x-2;
>> signChange(f,-6,6)

     k          f(k)
    -6      46.000000
    -5      33.000000
    -4      22.000000
    -3      13.000000
    -2       6.000000
    -1       1.000000
     0      -2.000000
```

Note that f experiences a change in sign between $x = -1$ and $x = 0$. Hence, f must have a zero in the interval $[-1, 0]$.

As a final example, we create a new anonymous function, but we feed only the left endpoint of the search interval. In this case, **nargin** equals 2, so our program sets a default for the right endpoint of the search interval (**b=10**). In this case, no zero is found, nor is a change in sign found, so the function terminates at the right-endpoint of the search interval.

```
>> f=@(x) x^2-2*x+2;
>> signChange(f,7)

             k           f(k)
             7      37.000000
             8      50.000000
             9      65.000000
            10      82.000000
```

Note that our **signChange** function can be improved significantly. For example, consider the function $f(x) = (x - 1.2)(x - 1.4)$. This function negative on $1.2 < x < 1.4$, but positive everywhere else. Because the **signChange** function only examines intervals of the form $[k, k + 1]$, where k is an integer, the routine will never discover the change in sign or the zeros of this function.

```
>> f=@(x) (x-1.2)*(x-1.4);
>> signChange(f,-1,3)

             k           f(k)
            -1       5.280000
             0       1.680000
             1       0.080000
             2       0.480000
```

Thus, one possible improvement might be to consider search intervals with non integer endpoints and allowing incrementsal changes different from 1.

Now that we've introduced the fundamentals of passing function handles to functions, we'll now explore algorithms that are used to find the zeros of functions.

The Bisection Method

If a function is continuous on a closed interval $[a, b]$, and if the function has different signs at each endpoint of $[a, b]$, then the function must attain a zero between a and b. Examples are shown in **Figures 5.1**(a) and **5.1**(b).

A natural thought comes to mind. Is there any way we can "squeeze" the interval $[a, b]$, making it narrower so that it "tightly wraps" itself around the zero crossing of the function? The answer is "yes," using a technique known in numerical analysis as the *bisection method*.

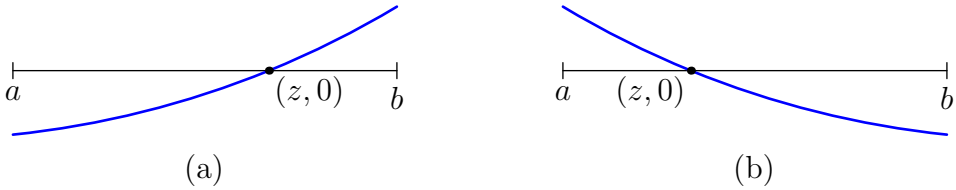


Figure 5.1. If the function has unlike signs at the endpoints of an interval, the function must have a zero in the interval.

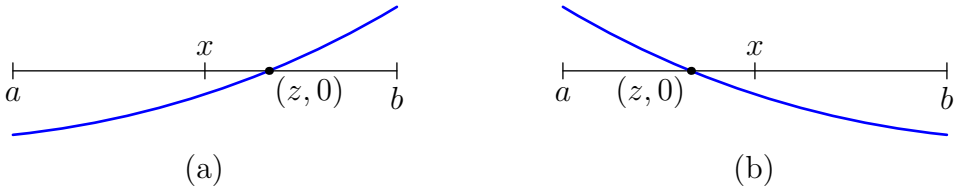


Figure 5.2. The first step of the bisection method is to find the midpoint of the interval $[a, b]$.

The idea is a simple one. First, calcuate the position of the midpoint of the interval with $x = (a + b)/2$ as shown in **Figures 5.2**(a) and **5.2**(b).

The next step is to determine if there is a change in sign at the endpoints of the interval $[a, x]$, where x is the midpoint of the interval $[a, b]$.[3]

a.) In **Figure 5.2**(a), there is no sign change at the endpoints of the interval $[a, x]$, as both $f(a) < 0$ and $f(x) < 0$. Hence, the zero crossing must take place in the interval $[x, b]$. In this case, we set $a = x$, as shown in **Figure 5.3**(a). This is the new search interval $[a, b]$.

b.) In **Figure 5.2**(b), there is a sign change at the endpoints of the interval $[a, x]$, as $f(a) > 0$ and $f(x) < 0$. Hence, the zero crossing must take place in the interval $[a, x]$. In this case, we set $b = x$, as shown in **Figure 5.3**(b). This is the new search interval $[a, b]$.

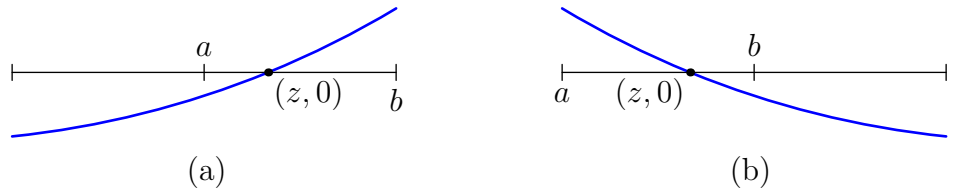


Figure 5.3. Determining the next search interval $[a, b]$.

Now we iterate. We use the new search interval $[a, b]$, find the midpoint, then determine which half interval exhibits a sign change on its endpoints, then set either $a = x$ or $b = x$, accordingly.

[3] Alternatively, you could try to determine if there is a sign change at the endpoints of the interval $[x, b]$.

We need a criterion for stopping the iteration. You'll note in **Figures 5.3**(a) and **5.3**(b) that the search interval $[a, b]$ is *halved* at each successive iteration. A stopping criterion that comes to mind is to halt the iteration when the length of the search interval falls below a certain tolerance, that is, when **abs(a-b)<tol**, where **tol** is an acceptably small number.

On the other hand, recall the work we did with relative error and how it related to the number of significant digits in a numerical solution. If we report b as the true solution and let a be its approximation, then the relative error would be

$$\text{Relative Error} = \frac{|b-a|}{|b|}.$$

We could terminate the iteration when the relative error falls below a certain tolerance. But what should we choose for the tolerance? One choice is to use Matlab's **eps**, which represents the distance between 1.0 and the next available floating point number. The value of **eps** is 2^{-52}.

```
>> eps
ans =
   2.2204e-16
>> 2^(-52)
ans =
   2.2204e-16
```

So, terminate the iteration when

$$\frac{|b-a|}{|b|} < \text{eps},$$

or equivalently, when **abs(b-a) < eps * abs(b)**.

Finally, we need some way of determining when a sign change occurs. Matlab's **sign** command provides the solution, returning -1 when the input is negative, 0 when the input is zero, and 1 when the input is positive.

```
>> x=[-5,0,7];
>> sign(x)
ans =
    -1     0     1
```

Thus, we can test for a sign change in f at the endpoints of the interval $[a, x]$ with **sign(f(a))==sign(f(x))**. If this expression returns true, then we know that the function agrees in sign at each endpoint of the interval $[a, x]$.

The following snippet codes the heart of the bisection method.

```
k=0;
while abs(a-b)>eps*abs(b)
    x=(a+b)/2;
    if sign(f(a))==sign(f(x))
        a=x;
    else
        b=x;
    end
    k=k+1;
end
```

The counter k is used to keep track of the number of iterations.

In the exercises, you will be ask to code a working function that will take a function handle and a search interval and return a zero of the function.

Newton's Method

Newton's method for finding the zeros of a function is quite simple to visualize. As seen in **Figures 5.4**(a) and (b).

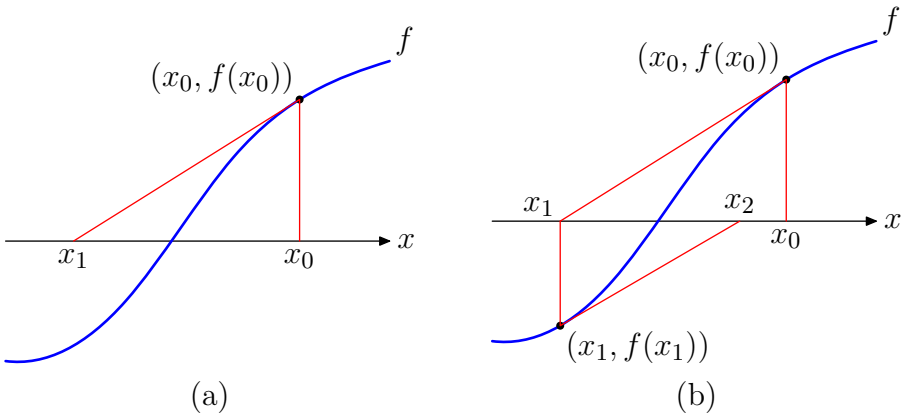


Figure 5.4. A geometrical interpretation of Newton's Method.

In **Figure 5.4**(a), we make a guess at the zero (labeled x_0 in **Figure 5.4**(a)). We then evaluate the function at this guess and draw a vertical line from the x-axis to the function at the point $(x_0, f(x_0))$. We then draw a tangent line to

the graph of f at the point $(x_0, f(x_0))$, and mark the point x_1 where this tangent line intersects the x-axis.

Now we iterate. A vertical line from x_2 to the graph of f, followed by a second tangent line to the graph of f at the point $(x_1, f(x_1))$. Mark the point x_2 where this second tangent line intersects the x-axis.

If conditions are just right, then as we iterate repeatedly, the sequence of points x_0, x_1, x_2, x_3, x_4, etc., will ultimately converge to a zero of the function, as shown in **Figure 5.5**.

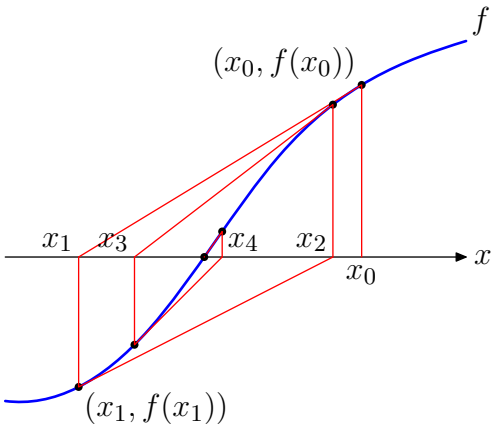


Figure 5.5. Under proper conditions, Newton's Method will converge to a zero of the function.

At the nth step, we need to find the equation of the line tangent to the graph of f at the point $(x_n, f(x_n))$. From calculus, we know that the slope of this tangent line is $f'(x_n)$. Thus, using the point-slope form of a line, the equation of the tangent line is

$$y - f(x_n) = f'(x_n)(x - x_n).$$

To find where this tangent line intersects the x-axis, we set y equal to 0 and solve for x.

$$\begin{aligned} -f(x_n) &= f'(x_n)(x - x_n) \\ -\frac{f(x_n)}{f'(x_n)} &= x - x_n \\ x &= x_n - \frac{f(x_n)}{f'(x_n)} \end{aligned}$$

Because the next term in the sequence is x_{n+1}, we have the following result.

Newton's Method. The following recursive definition is known as Newton's Method.

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{5.1}$$

Under proper conditions, the sequence of points x_0, x_1, x_2, ... generated by this recursive defintion will converge to a zero of the function f.

We say "under the proper conditions" because there are some circumstances under which Newton's Method will iterate indefinitely or leak off to infinity. We will explore some of these cases in the exercises.

We will now explore coding Newton's Method. This version of Newton's Method will require five input arguments, but will pass not output back to the caller.

```
function newton(f,fp,x0,rtol,iter)
```

The input arguments are described as follows.

f	=	A handle to the function definition of f
fp	=	A handle to the derivative definition of f'
x_0	=	An initial guess of the zero of f
$rtol$	=	The relative tolerance used in stopping criteria
$iter$	=	The maximum number of iterations allowed

We run a check to see if f and fp are true function handles. If not, we terminate execution with an error message.

```
if ~isa(f,'function_handle') || ~isa(fp,'function_handle')
    error('newton(f,fp,x0): f and fp must be function handles.')
end
```

We initialize a counter, then print a header for the printed output.

```
k=1;
fprintf('%20s\n','x')
```

Next comes the main loop.

```
while k<=iter
    x=x0-f(x0)/fp(x0);
    fprintf('%20.14f\n',x)
    if abs(x-x0)<rtol*abs(x)
        return
    end
    x0=x;
    k=k+1;
end
fprintf('Newton''s method failed after %d iterations.\n',k)
```

A number of comments are in order.

1.) Sometimes Newton's Method can work its way into an infinite loop, so we limit the number of iterations with **while k<=iter**. When k exceeds the maximum number of iterations (user input), the loop terminates with the failure message that follows the loop.
2.) We take the initial guess x_0 and use Newton's Recursion formula to determine the next value of x with **x=x0-f(x0)/fp(x0)**. Then we print the value of x to the command window with a nicely formatted **fprintf** command.
3.) If the relative error is below **rtol** (also user input), then the **return** command exits the function and returns control to the caller. Otherwise, we set $x_0 = x$, increment the counter, and iterate.

To test the function, we create an anonymous function definition for $f(x) = x^2 - 2x - 3$.

```
>> f=@(x) x.^2-2*x-2;
```

To use Newton's Method effectively, it is helpful to make a good guess near an actual zero. To that end, we use the anonymous function to plot the graph of f. The following commands were used to produce the image in **Figure 5.6**.

```
x=linspace(-2,4);
plot(x,f(x))
```

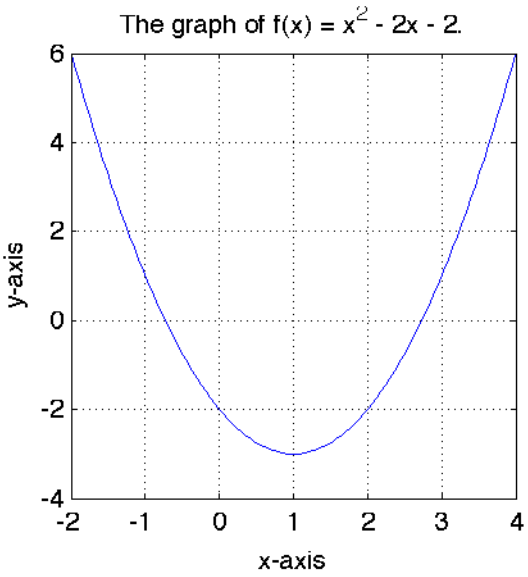


Figure 5.6. Estimating the zeros of $f(x) = x^2 - 2x - 2$ from its plot.

In **Figure 5.6**, it appears that the graph of f has an x-intercept near -0.6. We'll use this as our initial guess to Newton's Method. We'll set the relative tolerance to **eps** and the maximum number of iterations to 100. Using calculus, the derivative of $f(x) = x^2 - 2x - 2$ is $f'(x) = 2x - 2$, so we create an anonymous function for the derivative.

```
>> fp = @(x) 2*x-2;
```

Now we can call Newton's Method.

```
>> newton(f,fp,-0.6,eps,100)
                    x
   -0.73750000000000
   -0.73205935251799
   -0.73205080758996
   -0.73205080756888
   -0.73205080756888
```

Note the rapid convergence of Newton's Method. This rapid convergence is also evident in the image in **Figure 5.5**.

In this particular case, we can check the solution with the quadratic formula. Indeed, if $x^2 - 2x - 2 = 0$, then the quadratic formula provides two solutions,

$$x = \frac{2 \pm \sqrt{4+8}}{2} = 1 \pm \sqrt{3}.$$

We can use Matlab to approximate the first of these solutions.

```
>> format long
>> 1-sqrt(3)
ans =
  -0.73205080756888
```

Note the agreement with Newton's Method. To find the second solution, we would use the graph of f in **Figure 5.6** to approximate the second zero, then make a second call to the function **newton**. We will leave this computation to our readers.

Using an Options Structure

Our current version of Newton's Method displays the approximation at each iteration. However, there will be times when we don't want to see all of this information and simply have the routine pass the final approximation of the zero to the caller.

We will use an options structure, with three fields, to control how our routine behaves. We will use the following fields.

1. **options.display** can be set to the string **'none'** or the string **'iter'**. In the latter case ('iter'), our Newton routine will display the approximation of the zero at each iteration. In the first case ('none'), the display of intermediate results is suppressed and only the final zpproximation of the zero is returned to the caller.
2. **options.rtol** can be set by the user. In essence, it controls the number of significant digits in the solution.
3. **options.iter** can also be set by the user. It controls the maximum number of iterations allowed by Newton's method before reporting a failed search.

Thus, the header for our function adjusts to reflect this input.

```
function x=newton(f,fp,x0,options)
```

First, we check the number of input arguments passed to the routine by the caller. If there are three input arguments, then we know that the user did not pass an

options structure and set some defaults. If the number of input arguments is four, then we set defaults using the fields of the options structure.

```
if nargin==3
    display='none';
    rtol=eps;
    iter=100;
elseif nargin==4
    display=options.display;
    rtol=options.rtol;
    iter=options.iter;
else
    error('Use syntax: newton(f,fp,x0,options)')
end
```

Next, we run a check for a valid function handles for the function and its derivative.

```
if ~isa(f,'function_handle') || ~isa(fp,'function_handle')
    error('newton(f,fp,x0): f and fp must be function handles.')
end
```

The remainder of the code is similar to the first newton routine.

```
k=1;
if strcmp(display,'iter')
    fprintf('%20s\n','x')
end
while k<=iter
    x=x0-f(x0)/fp(x0);
    if strcmp(display,'iter')
        fprintf('%20.14f\n',x)
    end
    if abs(x-x0)<rtol*abs(x)
        return
    end
    x0=x;
    k=k+1;
end
fprintf('Newton''s method failed after %d iterations.\n',k)
```

There are two minor differences in the code displayed above.

1. The headers are printed only if **strcmp(display,'iter')** evaluates as true; that is, if the user passed an options structure with **options.display='iter'**.

```
if strcmp(display,'iter')
    fprintf('%20s\n','x')
end
```

2. Each approximation of the zero is printed only if **strcmp(display,'iter')** evaluates as true; that is, if the user passed an options structure with **options.display='iter'**.

```
if strcmp(display,'iter')
    fprintf('%20.14f\n',x)
end
```

Testing the Routine. Let's again use $f(x) = x^2 - 2x - 2$ and $f'(x) = 2x - 2$.

```
>> f=@(x) x^2-2*x-2;
>> fp=@(x) 2*x-2;
```

If we do not send an options structure, then the default behavior is to use a relative tolerance of **eps**, 100 maximum iterations, and to suppress display of approximations as they are calculated. The zero is returned and stored.

```
>> z=newton(f,fp,2)
z =
   2.73205080756888
```

On the other hand, we can create an options structure and make our own choices.

```
>> options.display='iter';
>> options.rtol=1e-4;
>> options.iter=200;
```

Now we can pass the options structure to the function. Because we assigned the string 'iter' to **options.display**, the routine displays each approximation as it is calculated.

```
>> newton(f,fp,2,options);
                  x
    3.00000000000000
    2.75000000000000
    2.73214285714286
    2.73205081001473
```

Note that we lose some precision with the larger **rtol**.

5.2 Exercises

1. Write a function **function x=bisection(f,a,b)** that codes the bisection method as described in the text. The inputs should be as follows: **f** is a function handle, **a** and **b** are endpoints of the search interval. In addition, $f(a)$ and $f(b)$ must have opposite sign. Your routine should do the following:

i.) Check to see if **f** is a true function handle.
ii.) Check that $a < b$ and $f(a)$ and $f(b)$ are opposite in sign.
iii.) Return b as the zero in x when **abs(b-a) < eps * abs(b)**.

Check your bisection function by finding the zeros of $f(x) = (x-1)(x+5)$, which are $x = 1$ and $x = 5$. Do this by creating an anonymous function **f=@(x) (x+5)*(x-1)**, then executing **bisection(f,-5.5,-4.5)** and **bisection(f,0.5,1.5)**.You should be able to do this in cell enabled mode by calling the external bisection function. Publish the result to HTML. Please include your function code as commented text in your published HTML file.

2. In the narrative, we showed how to use an options structure (as in **x=newton(f,fp,x0,options)**) to change the type of information displayed, the relative tolerance, and the maximum number of iterations. Adjust your bisection method from **Exercise 1** to accept an options structure (as in **x=bisection(f,a,b,options)** with one field, **options.display**. If **options.display** is set to 'iter', list the approximation to the zero at each iteration, otherwise, suppress this information and simply return the zero in the output variable x. Test your code by finding the positive zero of the function $f(x) = x^2 - 2$.

In **Exercises 2-6**, perform each of the following tasks.

i.) Write an anonymous function for the given function, then use your anonymous function to plot the graph on a domain the exhibits all of the zeros of the function. Turn on the grid, label the axes, and provide an appropriate title.
ii.) Use the graph to locate an interval on which the function changes sign. Call your bisection routine with **x=bisection(f,a,b)** and use the returned value to mark the zero in your plot with an asterisk. Use the text command to mark the point with its coordinates.

3. $f(x) = x^2 - 3x - 11$

4. $f(x) = x^2 + 5x - 12$

5. $f(x) = 14 - 6x - x^2$

6. $f(x) = 29 - 11x - x^2$

In **Exercises 7-10**, perform each of the following tasks.

i.) Create anonymous functions for the given function and its derivative, then use the anonymous function to plot the graph of the function on a domain that exhibits all of the zeros of the function. Turn on the grid, label the axes, and provide an appropriate title.

ii.) Use the graph to approximate a zero, send this estimate to the function **newton** with appropriate options structure, then use the returned value to mark the zero on your plot with an asterisk. Use the text command to mark the point with its coordinates.

7. $f(x) = 2x^2 - x - 72,\ f'(x) = 4x - 1$

8. $f(x) = 83 - x - 2x^2,\ f'(x) = -1 - 4x$

9. $f(x) = x^3 - 3x^2 - 19x - 2,\ f'(x) = 3x^2 - 6x - 19$

10. $f(x) = 15 + 23x - 2x^2 - x^3,\ f'(x) = 23 - 4x - 3x^2$

---

11. One difficulty with Newton's method is the fact that you must also calculate the derivative to implement the routine. This can be tedious at times. However, at each iteration, we can replace the derivative with the slope of the secant line. Suppose that we have calculated $x_1, \ldots, x_n$. With Newton's method, the next point in the sequence would be

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$

We can approximate the slope of the tangent line ($f'x_n$) with the slope of the secant line pictured in **Figure 5.7**(a). Then,

$$x_{n+1} = x_n - \frac{f(x_n)}{\dfrac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}}},$$

or equivalently,

$$x_{n+1} = x_n - \frac{f(x_n)(x_n - x_{n-1})}{f(x_n) - f(x_{n-1})}.$$

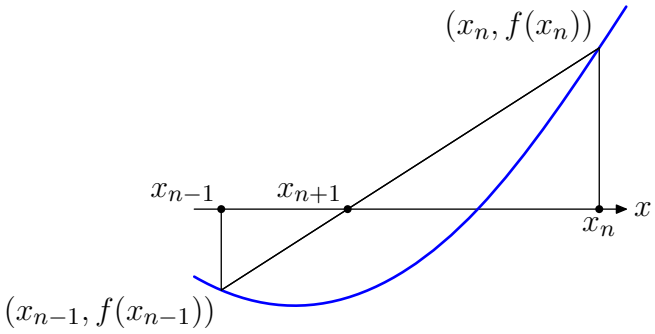


Figure 5.7. The secant method explained.

Write a function **z=secant(f,a,b,options)**, where f is an anonymous function and a and b are two initial guesses at the zero. Create an options structure similar to that used in Newton's method in the narrative so that the user has a choice of viewing the convergence at each iteration, or suppressing the view of the convergence and simply returning the estimate of the zero to the caller. Test your function throughly by finding the positive zero of $f(x) = x^2 - 2$.

12. Write a GUI that has the following components.

i.) An axes for plotting.
ii.) An edit box for entering an equation.
iii.) Edit boxes for the domain [xmin, xmax].
iv.) A push button that activates a callback that uses Matlab's **ginput** (type **doc ginput** to obtain help on using this command) command to allow the user to click near a zero. The callback should take the x-value of the result and pass it to Newton's Method. Use the zero returned by Newton's Method to plot and label the zero on the plot with its coordinates.

13. Write a GUI that has the following components.

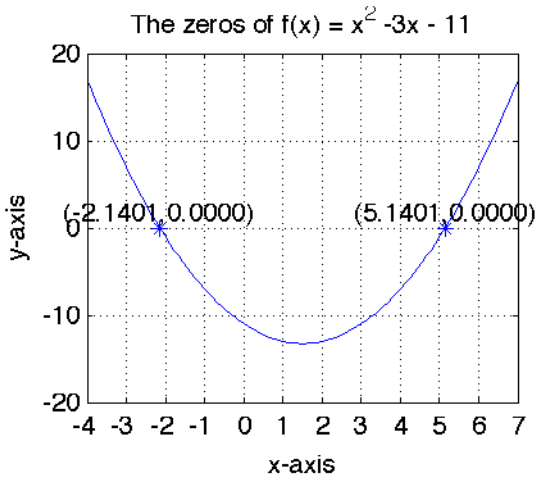
i.) An axes for plotting.
ii.) An edit box for entering an equation.
iii.) Edit boxes for the domain [xmin, xmax].
iv.) A push button that activates a callback that uses Matlab's **ginput** (type **doc ginput** to obtain help on using this command) command to allow the user to click twice in the axes, once on each side of a potential zero. The callback should take the x-values of the result and pass it to the Bisection Method. Use the zero returned by Bisection Method to plot and label the zero on the plot with its coordinates.

5.2 Answers

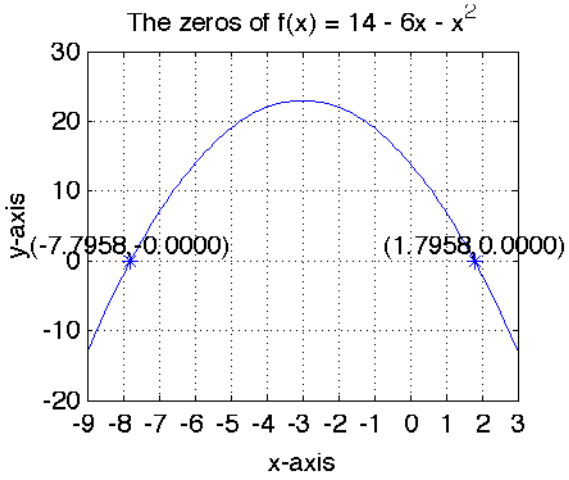
3. The following code also shows how to save the image to a file.

```
f=@(x) x.^2-3*x-11;
xmin=-4;
xmax=7;

x=linspace(xmin,xmax);
plot(x,f(x))
xlabel('x-axis')
ylabel('y-axis')
title('The zeros of f(x) = x^2 -3x - 11')
grid on

x=bisection(f,-3,-2);
line(x,f(x),...
    'LineStyle','None',...
    'Marker','*')
pointStr=sprintf('(%.4f,%.4f)',x,f(x));
text(x,f(x),pointStr,...
    'HorizontalAlignment','center',...
    'VerticalAlignment','bottom')

x=bisection(f,5,6);
line(x,f(x),...
    'LineStyle','None',...
    'Marker','*')
pointStr=sprintf('(%.4f,%.4f)',x,f(x));
text(x,f(x),pointStr,...
    'HorizontalAlignment','center',...
    'VerticalAlignment','bottom')

set(gca,'XLim',[xmin,xmax],'XTick',xmin:xmax)
set(gcf,'PaperPosition',[0,0,3,2.5])
print -dpng ZeroAlgorithmsExercise3.png
```

The code above was used to produce the following image.



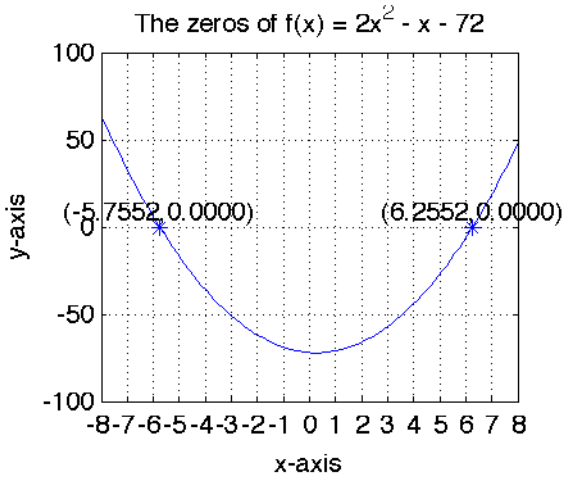
5.



7. This code also shows how to save the image to a file.

```
close all
clear all
clc

f=@(x) 2*x.^2-x-72;
fp=@(x) 4*x-1;
xmin=-8;
xmax=8;

options.display='None';
options.rtol=eps;
options.iter=100;

x=linspace(xmin,xmax);
plot(x,f(x))
xlabel('x-axis')
ylabel('y-axis')
title('The zeros of f(x) = 2x^2 - x - 72')
grid on

x=newton(f,fp,-7,options);
line(x,f(x),...
    'LineStyle','None',...
    'Marker','*')
pointStr=sprintf('(%.4f,%.4f)',x,f(x));
text(x,f(x),pointStr,...
    'HorizontalAlignment','center',...
    'VerticalAlignment','bottom')

x=newton(f,fp,6,options);
line(x,f(x),...
    'LineStyle','None',...
    'Marker','*')
pointStr=sprintf('(%.4f,%.4f)',x,f(x));
text(x,f(x),pointStr,...
    'HorizontalAlignment','center',...
    'VerticalAlignment','bottom')

set(gca,'XLim',[xmin,xmax],'XTick',xmin:xmax)
set(gcf,'PaperPosition',[0,0,3,2.5])
print -dpng ZeroAlgorithmsExercise7.png
```

The code results in the following image.



9.