

## Chapter 5

# Solving Algebraic Equations in Matlab

$$10\pi - 50\theta + 25\sin\theta = 0$$
$$\theta = ?$$

### 5.1 ME 218 LAB: Solving Equations of One-Variable

Complicated equations often cannot be solved algebraically. In such cases we must use numerical methods. This tutorial will overview several methods of numerical solution. As an example, we will solve the following equation:

$$f(x) = \frac{3}{500} (x^3 - 18.535x^2 + 25.697x + 28.099) = 0. \quad (5.1)$$

$$g(x) = 2xe^{\cos(3x)}e^{-x} + 70 \quad (5.2)$$

Specifically, we wish to find the value(s) of  $x$  where the function  $f$  is equal to zero. Create an m-file called onevariable.m. As you work through the tutorial below, add the necessary commands to the m-file.

### 5.1.1 Graphical solution of equations of one variable

The simplest way to find a solution to Eqn. (5.1) is to plot up the function and just look at it! In order to plot the function, we need two vectors: one vector of values along the horizontal axis and another vector of values containing the value of the function (the vertical axis).

Create the above functions `f.m` and `g.m` using the lessons learned in previous section. They will be used in the problem below.

**Problem 5.1** *In `onevariable.m`, create a vector  $x = [-10:0.1:20]$  and enter the commands to plot the function  $f$  at these values of  $x$ . Label the horizontal axis “ $x$ ” and the vertical axis “ $f$ .” Give the plot a title containing your name. Turn in your plot and `m`-file commands used to generate the the plot.*

**Problem 5.2** *In the Figure Window of the previous problem, use the magnifying glass tool to zoom in on the place(s) where  $f$  goes to zero. To an accuracy of two significant digits, what value(s) of  $x$  correspond to  $f = 0$ ?*

The graphical method of solution is effective and simple, but it requires that a person manually looks at a plot and pick off the points where the curve intersects the  $f = 0$  axis. If we need to solve several equations, this manual process can be very time consuming and it is better to have a computer do all the work with no user intervention. In the sections below, we will discuss some of these automated methods of solution.

### 5.1.2 The Bisection Method

The bisection method requires us to start with an  $x$  range containing one and *only* one root of the function. Since we’ve plotted the function, it is easy to choose a range of  $x$ . Consider the far right root of  $f(x)$ . Choose  $a_0$  as an  $x$  value to the left of the solution and  $b_0$  as an  $x$  value to the right of the solution such that  $b_0 - a_0 = 3$ .

We carry out the bisection method by splitting the interval in half repeatedly, always keeping the two points that straddle the zero crossing of the function.

#### 5.1.2.1 Manual bisection calculation

Before we write a MATLAB program to carry out Bisection, we will carry out a few steps of the bisection algorithm by hand. Fill in the blanks for the first step through the procedure:

1. The midpoint of the interval  $(a_0, b_0)$  is  $x = (a_0 + b_0)/2 =$

2. Find the value of the function at the midpoint  $f(x) =$

3. Is  $f(a_0)f(x) < 0$ ?
4. Based on your answer to item 3, pick values for  $a_1$  and  $b_1$  that straddle the zero of the function:  
 $a_1 =$    $b_1 =$

This procedure would repeat until some desired accuracy (or other condition) is reached.

Imagine how many times you would have to repeat the Bisection process to get a highly accurate solution. Compared to simply plotting the function, Bisection seems like an arduous process, but it is a process that a computer is very well suited for.

### 5.1.2.2 Automated Bisection

Let's try to write an m-file that will perform the Bisection algorithm. Since we don't know how many steps it will take, a FOR loop isn't the best thing. Let's use a WHILE loop instead.

We want to repeat all our steps until either

- $|b_i - a_i| < 1 \times 10^{-4}$ , or
- the number of times we've been through everything is  $> 20$ .

The first condition is a convergence criteria. The second condition will stop the program if convergence is not met within a maximum allowable number of steps. We don't need to keep the old versions of  $a_{i-1}$  and  $b_{i-1}$  around, so let's plan to use **a** and **b** to store the current endpoints in the bisection method. Also, the last condition can be written as  $i > 20$ , assuming we use **i** as a counting variable inside the **while** loop.

**Problem 5.3** Complete the MATLAB routine below to perform Bisection according to the description above. Hint: (a) fill in the conditions for the while loop, (b) assign the midpoint to **x** on the line just after the **while** command, (c) on the next line, assign the value of  $f(\mathbf{x})$  to **f\_at\_x**, (d) on the next few lines, create an IF statement that reassigns **a** and **b** appropriately for the two cases  $f\_at\_x * f(\mathbf{a}) < 0$  and  $f\_at\_x * f(\mathbf{a}) \geq 0$ . Run your bisection routine and verify that the result agrees with the graphical solution obtained earlier. Print out and turn in your Bisection routine and the calculated answer.

```
a = yourLoGuess ;
b = yourHiGuess ;
i = 0 ;
while (...
    ...
    ...
```

```

    if ( f_at_x * f(a) < 0 )
        ...
    else
        ...
    end
end
x % This line prints out final result in Command Window

```

**Problem 5.4** *How can we determine if the value in  $\mathbf{x}$  that's left after running the loop is a good solution or if the algorithm quit because the loop had been run too many times?*

### 5.1.2.3 Pitfalls of bisection

Bisection is a pretty robust process for finding roots, but if you aren't aware of its restrictions, you can get bad answers. If you haven't heard it before, let me tell you now:

⚠ A wrong answer is worse than no answer!

**Problem 5.5** *Modify the bisection routine to work on  $g(x)$  above and try it out with  $\mathbf{a}=4$  and  $\mathbf{b}=6$ . What do you get for  $\mathbf{x}$ ? What is the value  $\mathbf{x}$  returned by the algorithm? Is the value of the function close to 0? Why not?*

Remember, the bisection method requires the function to be continuous. Otherwise, there may not be a point where the function is zero. Also, it is a good idea to check that  $f(b_i)f(b_a) < 0$  before starting the bisection algorithm and it is also a good idea to check that (1)  $|b_i - a_i| < \delta$ , (2)  $i < 20$ , and (3)  $|f(\frac{a_i+b_i}{2})| < \epsilon$  when the loop ends. If all three aren't satisfied, there's a problem.

Another problem with bisection is that you have to give it *two* starting points. That was easy to do with  $f(x)$  for the root to the far right, but what about the area to the left? Is there even a solution to  $f(x) = 0$  there? One way to find out is to just start plugging in  $x$  values to see if the sign of  $f(x)$  is negative over the whole range. Of course, you may miss a spot. Newton's method is one way to avoid this problem.

### 5.1.3 Newton's Method

Rather than make you program Newton's method yourselves, we just want you to see how it works. Below is a copy of `fig_newton.m`. Copy that m-file into your folder that contains `f.m` and `g.m`. (A note of caution: Although you can use the text tool in Adobe to C&P the the below text into an m-file, not all characters may be recognized by MATLAB. Be sure and look over the m-file to ensure it copied 100% correctly.)

```

% fig_newton.m
function root = fig_newton( func, x0, delta, epsilon, ax )
    axis ;
    maxstep = 40 ;
    width = ax(2)-ax(1) ;

    xdata = ax(1):(ax(2)-ax(1))/200:ax(2) ;
    for i=1:length(xdata)
        ydata(i) = feval( func, xdata(i) ) ;
    end

    steps = 0 ;
    curx = x0 ;
    curfx = feval( func, curx ) ;
    curdlt = 2*delta ;

    while ( steps < maxstep & curdlt > delta & abs(curfx) > epsilon )
        % Estimate df/dx
        fpdlt = feval( func, curx+delta ) ;
        dfdx = (curfx - fpdlt)/delta ;
        % Find new guess
        dx = curfx/dfdx ;
        newx = curx + dx ;
        newfx = feval( func, newx ) ;

        ep = [ curx curfx ; newx 0 ] ;
        curdlt = abs(curx-newx) ;
        plot(xdata,ydata,'-',ep(:,1),ep(:,2),'-',curx,curfx,'x',newx,newfx,'x');
        grid on ;
        curx = newx
        curfx = newfx ;
        figure(1); pause

        steps = steps + 1 ;
    end
    root = curx ;
    axis ;

```

You should be familiar with most of the functions used in the above m-file. If not, do not worry about it too much. One of the advantages of function programming is that you can use someone else's code and treat it like a block box as long as you know what to put it in and can interpret what is coming out. If you are curious be sure and use MATLAB's help to learn more

about the commands you are unfamiliar with.

At the MATLAB prompt, run your new script file like this:

```
>>fig_newton( 'f', 10, 0.001, 0.001, [ -5 20 ] ) ;
```

It should draw the function for you in the range from -5 to 20 (that's what the last argument is for). You'll see the starting point  $x = x_0 = 10$  (that's the second argument),  $y = f(x_0)$  marked with an "x". And there's a line from that pointing up to the horizontal axis. Press return and **fig\_newton** will draw another figure, this time with the "x" in the new spot decided by the intersection of the line approximating  $f(x)$  and the horizontal axis. There will be a new line, again approximating  $f(x)$ , but about  $x_1$  instead of about  $x_0$ . Each time you HIT RETURN, the function will print a new approximation for the root of the function until finally it is within the tolerances you set.

The third and fourth arguments to **fig\_newton** are values for  $\delta$  (the convergence criteria) and  $\epsilon$  (the accuracy criteria), respectively.

### 5.1.3.1 Running the function

**Problem 5.6** Run *fig\_newton* three times using  $f(x)$  as the function. Use the starting values given below and write down what it converged to (that's the return value of the function).

$x_0$	$x^*$
3	_____
7	_____
20	_____

### 5.1.3.2 Pitfalls of Newton's method

Of course, Newton's method doesn't always converge. Try running **fig\_newton** with the command below and watch what happens:

```
>>fig_newton( 'f', 12.0, 0.001, 0.001, [ -5 20 ] ) ;
```

**Problem 5.7** What happens when you execute the above line? Repeat using  $x_0 = 11.5$  and discuss the difference between the two results.

### 5.1.4 MATLAB's Method

MATLAB has a function named `fzero` (short for “find zero”) to quickly (i.e., with very few function evaluations) find the zero of a function. To understand the command syntax, type

```
>>help fzero
```

in the Command Window. The `fzero` function can take different numbers of arguments to do fancy things, but we are only worried about the most basic way to use it. That's discussed in the first paragraph of the help. Then, at the bottom of the help are some examples. These examples use a command named `inline` we haven't discussed yet. But if you don't understand something in an example, you can always ask for help on that, too:

```
>>help inline
```

Once you've read all this, simply try `fzero`:

```
>>fzero( 'f', 1.0 )
```

**Problem 5.8** Using MATLAB's `fzero` command, write down the zeros for the functions and starting points below.

Function	$x_0$	$x^*$
$f(x)$	1.0	_____
$g(x)$	2.4	_____
$g(x)$	-1.8	_____
$g(x)$	-4.2	_____

## 5.2 Solving Systems of Linear Equations

Sets of linear equations can arise in many physical applications including, for example, calculation of static stresses on a bridge. Gaussian elimination is an algorithm for solving systems of  $n$  linear equations in  $n$  unknowns:  $Ax = b$ . In this exercise, we will consider the following set of three equations and three unknowns:

$$\begin{bmatrix} 9 & 2 & 5 \\ 27 & 6 & 34 \\ 7 & 11 & 3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 8 \\ 3 \end{bmatrix} \quad \text{or} \quad \begin{bmatrix} 9 & 2 & 5 & 1 \\ 27 & 6 & 34 & 8 \\ 7 & 11 & 3 & 3 \end{bmatrix}$$

The matrix equation on the left is of the form  $[A]x = b$ . On the right, this equation has been re-expressed as a single augmented matrix. In general, any  $n$  linear equations with  $n$  unknowns can be written in augmented matrix form as

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} & \dots \\ a_{21} & a_{22} & a_{23} & a_{24} & \dots \\ a_{31} & a_{32} & a_{33} & a_{34} & \dots \\ \vdots & \vdots & \vdots & \vdots & \vdots \end{bmatrix}$$

### 5.2.1 Gaussian Elimination

The Gaussian elimination algorithm solves the problem as follows:

*For each element on the diagonal  $a_{ii}$*   
*For each row  $j$  beneath this element (i.e., for all  $i < j \leq n$ )*  
*Multiply row  $i$  by  $-\frac{a_{ji}}{a_{ii}}$  and add it to row  $j$ . (This puts 0 in elements below  $a_{ii}$ )*  
*end*  
*end*

These steps create an upper diagonal matrix (all zeros below the diagonal). The upper diagonal matrix is of the form

$$\begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} & \dots \\ 0 & b_{22} & b_{23} & b_{24} & \dots \\ 0 & 0 & b_{33} & b_{34} & \dots \\ \vdots & \vdots & \vdots & \vdots & \vdots \end{bmatrix}$$

Notice that the bottom row now represents an equation of the form  $b_{33}x_3 = b_{34}$  with solution  $x_3 = b_{34}/b_{33}$ . We finish solving all  $n$  equations with “back substitution.” The method is as follows:

*For each element,  $a_{ii}$  on the diagonal, starting with  $a_{nn}$  and moving to  $a_{11}$ ,*  
*Solve for  $x_i$  using the  $i^{\text{th}}$  equation – it should have  $x_i$  as the only unknown.*  
*end*

Clear out MATLAB’s memory with `clear` and then assign the  $3 \times 4$  augmented matrix into the MATLAB variable `M`. An element in the matrix can be displayed by specifying the row number and column number in parentheses to the right of the variable name. For example, type `M(1,2)` and see what you get in the Command Window. If you place a colon (`:`) inside the parentheses MATLAB will return all of the elements in the column or row.

**Problem 5.9** Use MATLAB commands to display (a) the entire third column of  $M$ , (b) the entire first row of  $M$ , (c) the element in the second column, third row of  $M$ . Show you commands and MATLAB’s response.



Now that we have this matrix, let's eliminate all the numbers beneath the diagonal elements (9, 6, and 3 in this example). MATLAB has an easy way to perform the row operations. The first row operation is to multiply the first equation by  $-27/9$  and add it to the second equation. To do this in MATLAB:

```
>>M(2,:) = M(2,:) -27/9*M(1,:)  
MATLAB response: M =   9   2   5   1  
                   0   0  19   5  
                   7  11   3   3
```

**Problem 5.10** Now, you enter the command to eliminate the 7 (in the third row). Show the MATLAB command and response.

The next step in the algorithm is to eliminate the numbers below the next element on the diagonal. But wait! There's a problem. Now the second element on the diagonal is 0! If we try to use the formula in the algorithm above, we'll be dividing by 0. To solve these equations, we'll have to change the algorithm. We'll do this with a technique called *pivoting*.

## 5.2.2 Pivoting

In order to get a non-zero element on the diagonal, all we have to do is realize that it doesn't matter what order we write the equations in. If we swap equations 2 and 3, then the second diagonal element will be non-zero. To do this in MATLAB, use these commands:

```
tmp = M(3,:); % we create a temporary variable  
M(3,:) = M(2,:);  
M(2,:) = tmp
```

If you leave the semicolon off the last statement, you'll see the matrix printed out with rows 2 and 3 swapped.

Now, we can continue with the algorithm. Since every element in the matrix **M** that's below the diagonal is 0, we are done with the first loop of the algorithm. Now it's time to perform back substitution.

**Problem 5.11** Finish writing the code below to perform back-substitution. (That is, figure out what to put in the two blank lines.) The vector **x** will store the solution to the system of equations. Turn in the completed code.

```
x = [0 0 0];  
for i=3:-1:1  
    % first, solve the ith equation for x(i)
```

```

x(i) =
% substitute x(i) into the matrix eqn. and move constant to 4th column
M(:,4) = M(:,4) - M(:,i)*x(i);
M(:,i) =          ;
end

```

Your answer should be  $x = [-0.092879 \ 0.260062 \ 0.263158]$ .

### 5.2.3 MATLAB's Routines

MATLAB has a very powerful linear system solver that takes care of Gaussian elimination, pivoting, and back solving. In order to understand how MATLAB's routine works, we need to look at matrix multiplication a little more carefully.

When you multiply two numbers together and get 1, we say those numbers are the *inverses* of each other:  $a \cdot b = 1 \implies b = \frac{1}{a}$ . Also, the number 1 is special because when we multiply any number by 1, we get the same number.

There are special matrices called *identity* matrices. When you multiply any matrix that's the correct size by an identity matrix, you get the original matrix back. An identity matrix has 1's on the diagonal and 0's everywhere else. For instance,

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

So, solving a linear systems of equation is like multiplying by the inverse of matrix A from the left to get an identity matrix I times  $x$  on the left hand side:

$$\begin{aligned} A^{-1}Ax &= A^{-1}b \\ Ix &= A^{-1}b \\ x &= A^{-1}b \end{aligned}$$

MATLAB can find the inverse of a matrix with the INV function.

```
>>INV(A)
```

In the example we are considering

$$A = \begin{bmatrix} 9 & 2 & 5 \\ 27 & 6 & 34 \\ 7 & 11 & 3 \end{bmatrix} \quad x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \quad b = \begin{bmatrix} 2 \\ 8 \\ -3 \end{bmatrix}$$

Run the command above to see what the inverse of A. Now, multiply the inverse of A by A. Remember, since matrix multiplication isn't commutative, it matters which side you multiply on!

```
>> INV(A) * A
```

Whoa, that's not exactly the identity matrix is it? I mean, it's got 1's on the diagonal, but the numbers off the diagonal aren't exactly zero. That's because matrix inversion is hard to do numerically. The results are often slightly off. Although you can get  $x$  using this command

```
>> x = INV(A) * b
```

it is generally better to have MATLAB combine the matrix inversion with the multiplication by the right hand side ( $b$ ) into one step. That way, MATLAB can reduce the error. To let MATLAB know that's what you want, use this command:

```
>> x = A \ b
```

The backslash is called the “division from the left” operator.

**Problem 5.12** Calculate  $x$  this way and check your answer by multiplying  $A$  and  $x$  (this should equal  $b$ ). Show the commands and MATLAB's responses.

#### 5.2.4 Ill-Conditioned Matrices

Finally, there are some linear systems that even MATLAB's advanced numerical techniques will have a hard time with. That's because the linear system of equations is close to degenerate. For instance, when a set of two equations end up being linearly dependent (i.e., when one row of  $A$  is a multiple of another), the problem cannot be solved. It's easy to visualize this: two rows that are multiples of each other represent parallel lines. Either they don't intersect anywhere, or the intersect everywhere, depending on the right hand sides of the equation.

Lack of linear independence rarely occurs *exactly*. Usually, because we've rounded off some numbers somewhere, the lines end up being *close* to parallel. That's a problem because it's not always easy to detect and can lead to *wrong* answers, not just no answer. Consider this system:

$$\begin{bmatrix} 1.001 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$$

Because the lines are close to parallel, it's hard to find out exactly where they intersect.

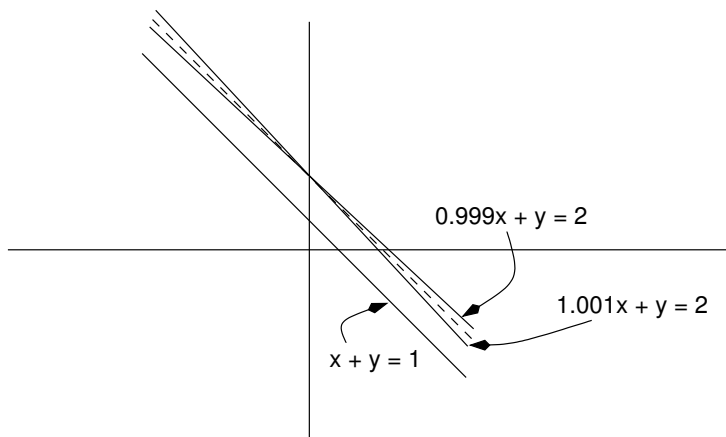
**Problem 5.13** Use MATLAB to solve the system of equations. (MATLAB will get the correct solution because it can deal with this kind of ill-conditioning.) Show your commands and MATLAB's response.

Now, let's take a look at what happens if we rounded differently and got this matrix instead:

$$\begin{bmatrix} 0.9999 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$$

**Problem 5.14** Use MATLAB to solve the system of linear equations above. Show your commands and MATLAB's response.

See how far off the answers are? That's because the slope of first line went from being pointed slightly below  $-45^\circ$  to slightly above it, as the figure below shows.



The problem is, computers normally only store 15 or 16 significant digits. So, every calculation you make rounds off the matrix. A matrix that's close to degenerate can easily have its solution changed to some point that's not even close to a solution! MATLAB tries to detect these problems and will let you know when a matrix is ill-conditioned. But MATLAB isn't perfect and you may have rounded off the matrix before you even entered it into MATLAB, so if you get unexpected results, be sure and check the *condition* of your coefficient matrix. The MATLAB `cond` function will return a number that is low for "well-posed" or "good" matrices and high for "ill-conditioned" or "bad" matrices.

**Problem 5.15** Use MATLAB to find the condition of the two matrices above and compare that to `cond(A)` from our earlier example. Show your commands and MATLAB's response.