MATLAB Basics

P. Howard

Fall, 2003

Contents

1 Introduction

MATLAB, which stands for MATrix LABoratory, is a software package developed by MathWorks, Inc. to facillitate numerical computations. It differs from such packages as Maple, Mathematica, and Macsyma in that while these three are primarily symbolic manipulation packages, MATLAB is best suited for the kind of heavy duty numerics that often arise in industrial problems. This is not to say that MATLAB balks entirely at the prospect of symbolic manipulation (or that Maple, Mathematica, or Macsyma are completely hopeless when it comes to numerics), only that its focus is different. MATLAB strikes me as being slightly more difficult to begin working with than the packages mentioned above, though once you get comfortable with it, it offers greater flexibility. The main point of using it in M442 is that it is currently the package you will most likely find yourself working with if you get a job in engineering or industrial mathematics.[1]

If you aren't familiar with MATLAB, you should use these notes as a (relatively) brief tutorial. Before doing so, you should create some subdirectories in your account: *matlab* off your main directory and *examples* off the *matlab* directory. Even if you are well versed in MATLAB, I strongly recommend that a MATLAB subdirectory be created, and that a separate sub-subdirectory be created in *matlab* for each project. (See Section 10 for an extremely brief discussion of Unix issues and, more important, a reference to how you can learn more.)

Finally, it should be pointed out that these notes do not pretend to be any kind of thorough introduction to MATLAB. They are simply meant to get you started on the kinds of things you'll need to know for M442. Significantly more detailed discussions can be found in the references listed at the end.

2 Starting MATLAB at Texas A&M University

You should have a calclab account assigned to you for M442 (I'll pass these out on the first day of class, or as soon as I get them). Log in and click on the six pointed geometric figure in the bottom left corner of your screen. Go to **Mathematics** and choose **Matlab**. Congratulations! (Alternatively, click on the surface plot icon at the foot of your screen.)[2]

For basic information on using calclab accounts at Texas A&M University—printing, access, etc.—get Art Belmonte's *Maple in Texas A&M's Mathematics Courses*, available at the department web site:

http://calclab.math.tamu.edu/~belmonte/calclab/MapleIntro21.pdf.

3 Simple Computations with MATLAB

3.1 Basic Computations

The (default) MATLAB screen is divided into three windows, with a large command window on the right, and two smaller ones that we won't worry about yet on the left. At the prompt, designated by two arrows, >>, type *2 + 2* and press **Enter**. (Yes, I meant *basic* computations.) You should find that the answer has been assigned to the default variable *ans.* Not so hard. Next, type *2+2;* and hit Enter. Notice that unlike Maple, the semicolon suppresses screen output in MATLAB.

We will refer to a series of commands as a MATLAB *script*. For example, we might type

[1] If you get a job in a particular field of engineering or industry (as opposed to engineering or industrial *mathematics*) you will most likely used specialized software.

[2] By the way, since I don't actually have a student account, these specific directions may occasionally be wrong. Ask me if you have any troubles. Not only will your life be easier; you will improve the lives of students for years to come.

```
>>t=4;
>>s=sin(t)
```

MATLAB will report that s = -.7568. (Notice that MATLAB assumes that t is in radians, not degrees.) While we're at it, type the up arrow key on your keyboard, and notice that the command *s=sin(t)* comes back up on your screen. Hit the up arrow key again and *t=4;* will appear at the prompt. Using the down arrow, you can scroll back the other way, giving you a convenient way to bring up old commands without retyping them.

Occasionally, you will find that an expression you are typing in is getting out of hand and needs to be continued to the next line. You can accomplish this by putting in three dots and typing **Enter**. Try the following:[3]

```
>>2+3+4+...
+5+6

ans =

    20
```

Notice that *2+3+4+...* was typed at the Command Window prompt, followed by **Enter**. When you do this, MATLAB will proceed to the next line, but it will not offer a new prompt. This means that it is waiting for you to finish the line you're working on.

As with any other software package, the most important MATLAB command is *help*. You can type this at the prompt just as you did the commands above. For help on a particular topic such as the integration command *int*, type *help int*. If the screen's input flies by too quickly, you can stop it with the command *more on*. Finally, MATLAB has a nice help browser that can be invoked by typing *helpdesk*.

Let's get some practice with MATLAB help by computing the inverse sine of -.7568. First, we need to look up MATLAB's expression for inverse sine. At the prompt, type *helpdesk*. Next, in the left-hand window of the pop-up menu, click on the **index** tab (second from left), and in the data box type *inverse*. In the box below your input, you should now see a list of *inverse* subtopics. Using your mouse, scroll down to *sine* and click on it. An example should appear in the right window, showing you that MATLAB uses the function *asin()* as its inverse for *sine*. Close help (by clicking on the upper right X as usual), and at the prompt type *asin(-.7568)*. The answer should be -.8584. (Pop quiz: If *asin()* is the inverse of *sin()*, why isn't the answer 4?)

Final comment: MATLAB uses double-precision floating point arithmetic, accurate to approximately 15 digits. By default, only a certain number of these digits are shown, typically five. To display more digits, type *format long* at the beginning of a session. All subsequent numerical output will show the greater precision. Type *format short* to return to shorter display. MATLAB's four basic data types are *floating point* (which we've just been discussing), *symbolic* (see Section 3.2), *character string*, and *inline function* (see Section 3.2).

3.1.1 Array Operations

Section 5 of these notes is devoted to matrices, but I want to jump ahead while we're talking about basic computations and warn you about something early on. Define the vector $X = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$ by typing *X=[1; 2; 3]* at the command prompt. Define a second vector $Y = \begin{pmatrix} 4 & 5 & 6 \end{pmatrix}$ by typing *Y=(4 5 6)* at the command prompt. Notice in particular that while X is a *column* vector (each row was ended by a semicolon), Y is a *row* vector. Let's examine the significance of this by computing first $Y * X$, then $X * Y$:

[3] In the MATLAB examples of these notes, you can separate the commands I've typed in from MATLAB's responses by picking out those lines that begin with the command line prompt, >>. All such examples have been created directly from a MATLAB session using the *diary* command. Typing *diary filename* begins a session, and typing *diary off* ends it. The session is then stored as a text file under the name *filename*.

```
>>Y*X
ans =
32
>>X*Y
ans =
4 5 6
8 10 12
12 15 18
```

Notice that while $Y * X$ is the usual vector dot product, the dimensions of X and Y are such that $X * Y$ forms a matrix. *Here's what I actually want to warn you about.* Very often, you will want to multiply two vectors together element by element: the first entry of X by the first entry of Y, the second entry of X by the second entry of Y and so on. To accomplish this, MATLAB has the odd-looking operator .* (also ./ and .^). To see how this works, type at the command prompt *X=[1 2 3],* so that X and Y are both row vectors. First—and, be warned, this won't work—try typing $X * Y$. MATLAB should inform you that your matrix dimensions must agree (it can create neither a dot product nor a matrix product from this combination). (By the way, you will get this message a lot, that your matrix dimensions don't agree, so keep this example in mind and remember how to fix it.) Try now $X. * Y$. Notice that MATLAB returns a vector in which the elements of X have been multiplied by the elements of Y.

3.2 Basic Algebra

Variables can be manipulated algebraically in MATLAB if they are declared as symbols by the *syms* command. Try, for example,

```
>>syms x y
>>z=(x - y)*(x+y)
>>expand(z)
```

Type *help symbolic* to learn more about this. It may happen that during a particularly long MATLAB session, you lose track of what variables have been assigned. Type *whos* to get a list. To clear the assignment of a variable x, type *clear x*.

3.2.1 Solving Algebraic Equations in MATLAB

There are two basic methods for solving algebraic equations in MATLAB, in line and with function files. As we will see, the function file method is considerably more robust, but the inline method is fairly easy. First, let's solve the simple algebraic equation $x^2 - 2x - 4 = 0$. The command is *solve* and the equation must appear in single quotes.

```
>>solve('x^2 - 2*x - 4 = 0')

ans =

[ 5^(1/2)+1]
[ 1-5^(1/2)]
```

Alternatively, if we simply have an expression between single quotes, rather than an equation, MATLAB will set the expression to 0 by default. That is, the MATLAB command *solve('x^2-2*x-4')* also solves the equation $x^2 - 2x - 4 = 0$. Next, let's solve the system of equations

$$x^2 - y = 2$$
$$y - 2x = 5.$$

We use

```
>>[x,y]=solve('x^2 - y = 2','y-2*x=5')

x =

[ 1+2*2^(1/2)]
[ 1-2*2^(1/2)]


y =

[ 7+4*2^(1/2)]
[ 7-4*2^(1/2)]
```

Incidentally, you might decide that what you require are decimal representations of x and y. These can be obtained with the *eval()* command. Continuing the code above, we have,

```
>>eval(x)
ans =
3.8284
-1.8284
>>eval(y)
ans =
12.6569
1.3431
```

Finally, we solve the more difficult equation, $e^{-x} - \sin x = 0$, using the numerical solver *fzero*.

```
>>fzero(inline('exp(-x)-sin(x)'),.5)

ans =

    0.5885
```

In the command *fzero*, the value .5 is an initial guess as to the solution of

$$e^{-x} - \sin(x) = 0.$$

While *solve* is an algebraic function, *fzero* is numerical. In this last example, we could also have defined our function beforehand:

```
>>f=inline('exp(-x)-sin(x)','x')

f =

     Inline function:
     f(x) = exp(-x)-sin(x)

>>fzero(f,1)

ans =

    0.5885
```

3.2.2 Inline Functions

In the examples above, we defined functions with the command *inline()*. Later, we will see that most functions in MATLAB are defined through M-files, but simple functions can be defined with *inline*. Once a function has been defined with *inline*, it can easily be solved, evaluated etc. In the following MATLAB code, the function $f(x) = x^2 + \sin x$ is defined and evaluated at the point $x = 1$.

```
>>f=inline('x^2+sin(x)','x')
f =
Inline function:
f(x) = x^2+sin(x)
>>f(1)
ans =
1.8415
```

3.3 Basic Calculus

Of course, MATLAB comes equipped with a number of tools for evaluating basic calculus expressions.

3.3.1 Differentiation

Symbolic derivatives can be computed with *diff()*. To compute the derivative of x^3, type:

```
>>syms x;
>>diff(x^3)

ans =

3*x^2
```

or

```
>>diff('x^3','x')

ans =

3*x^2
```

Alternatively, you can first define x^3 as a function of f.

```
>>f=inline('x^3','x');
>>diff(f(x))

ans =

3*x^2
```

Higher order derivatives can be computed simply by putting the order of differentiation after the function, separated by a comma.

```
>>diff(f(x),2)

ans =

6*x
```

Finally, MATLAB can compute partial derivatives. See if you can make sense of the following input and output.

```
>>syms y;
>>g=inline('x^2*y^2','x','y')

g =

     Inline function:
     g(x,y) = x^2*y^2

>>diff(g(x,y),y)

ans =

2*x^2*y
```

3.3.2 Integration

Symbolic integration is similar to symbolic differentiation. To integrate x^2, use

```
>>int('x^2','x')

ans =

1/3*x^3
```

or

```
>>syms x;
>>int(x^2)

ans =

1/3*x^3
```

or, finally,

```
>>f=inline('x^2','x')

f =

     Inline function:
     f(x) = x^2

>>int(f(x))

ans =

1/3*x^3
```

The integration with limits $\int_0^1 x^2 dx$ can easily be computed if f is defined inline as above:

```
>>int(f(x),0,1)

ans =

1/3
```

For double integrals, such as $\int_0^\pi \int_0^{\sin x} (x^2 + y^2) dy dx$, simply put one *int()* inside another:

```
>>syms y
>>int(int(x^2 + y^2,y,0,sin(x)),0,pi)

ans =

pi^2-32/9
```

Numerical integration is accomplished through the commands *quad*, *quad8*, and *quadl*. For example,

```
quadl(vectorize('exp(-x^4)'),0,1)

ans =

    0.8448
```

(If x has been defined as a symbolic variable, you don't need the single quotes.) You might also experiment with the numerical double integration function *dblquad*. Notice that the function to be numerically integrated must be a vector; hence, the *vectorize* command. In particular, the vectorize command changes all operations in an expression into array operations. For more information on *vectorize*, type *help vectorize* at the MATLAB Command Window.

3.3.3 Limits

MATLAB can also compute limits, such as

$$\lim_{x\to 0} \frac{\sin x}{x} = 1.$$

We have,

```
>>syms x;
>>limit(sin(x)/x,x,0)

ans =

1
```

For left and right limits

$$\lim_{x\to 0^-} \frac{|x|}{x} = -1; \qquad \lim_{x\to 0^+} \frac{|x|}{x} = +1,$$

we have

```
>>limit(abs(x)/x,x,0,'left')

ans =

-1

>>limit(abs(x)/x,x,0,'right')

ans =

1
```

Finally, for infinite limits of the form

$$\lim_{x\to\infty} \frac{x^4 + x^2 - 3}{3x^4 - \log x} = \frac{1}{3},$$

we can type

```
>>limit((x^4 + x^2 - 3)/(3*x^4 - log(x)),x,Inf)

ans =

1/3
```

3.3.4 Sums and Products

We often want to take the sum or product of a sequence of numbers. For example, we might want to compute

$$\sum_{n=1}^{7} n = 28.$$

We use MATLAB's *sum* command:

```
>>X=1:7

X =

     1     2     3     4     5     6     7

>>sum(X)

ans =

    28
```

Similarly, for the product

$$\prod_{n=1}^{7} n = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \cdot 6 \cdot 7 = 5040,$$

we have

```
>>prod(X)

ans =

        5040
```

MATLAB is also equipped for evaluating sums symbolically. Suppose we want to evaluate

$$\sum_{k=1}^{n} (\frac{1}{k} - \frac{1}{k+1}) = 1 - \frac{1}{n+1}.$$

We type

```
>>syms k n;
>>symsum(1/k - 1/(k+1),1,n)

ans =

-1/(n+1)+1
```

3.4 Taylor series

Certainly one of the most useful tools in mathematics is the Taylor expansion, whereby local information (at a single point) can be used to obtain global information (in a neighborhood of the point and sometimes on an infinite domain). The Tayor expansion for $\sin x$ up to tenth order can be obtained through the commands

```
>>syms x;
>>taylor(sin(x),x,10)

ans =

x-1/6*x^3+1/120*x^5-1/5040*x^7+1/362880*x^9
```

We can also employ MATLAB for computing the Taylor series of a function about points other than 0.[4] For example, the first four terms in the Taylor series of e^x about the point $x = 2$ can be obtained through

```
>>taylor(exp(x),4,2)

ans =

exp(2)+exp(2)*(x-2)+1/2*exp(2)*(x-2)^2+1/6*exp(2)*(x-2)^3
```

3.5 The Subs Command

An extremely useful command in MATLAB is *subs()*, which can be used to evaluate an expression at a particular variable value. For example, suppose we would like to evaluate the second Taylor expansion of Section 3.4 at $x = 2.1$. We use simply *subs(ans,2.1)*. More generally, suppose we have a symbolic expression of multiple variables. If any parameter values are defined in the workspace, the *subs* command substitutes them for the parameters. In the following example, a general quadratic equation is solved, and then one of the parameters in the root is evaluated.

```
>>r=solve('a*x^2+b*x+c=0','x')
r =
[ 1/2/a*(-b+(b^2-4*a*c)^(1/2))]
[ 1/2/a*(-b-(b^2-4*a*c)^(1/2))]
a=1;
subs(r)
ans =
[ -1/2*b+1/2*(b^2-4*c)^(1/2)]
[ -1/2*b-1/2*(b^2-4*c)^(1/2)]
```

3.6 M-Files

3.6.1 Script M-Files

The heart of MATLAB lies in its use of M-files. We will begin with a *script* M-file, which as we will see, simply contains a list of commands like the one given in Sections 3.1 and 3.2, above. To create an M-file, click on **File** at the upper left corner of your MATLAB window, then select **New**, followed by **M-file**. A window will appear in the upper left corner of your screen with MATLAB's default editor. (You are free to use an editor of your own choice, but for the brief demonstration here, let's stick with MATLAB's. It's not the most powerful thing you'll ever come across, but it's not a complete slouch either.) In this window, type the following lines (MATLAB reads everything following a % sign as a comment to be ignored):

[4] You may recall that the Taylor series of a function about the point 0 is also referred to as a Maclaurin series.

```
%JUNK: A script file that computes sin(4),
%where 4 is measured in degrees.
t=4;      %Define a variable for no particularly good reason
radiant=pi*t/180;  %Converts 4 degrees to radians
s=sin(radiant)  %No semicolon means output is printed to screen
```

Save this file by choosing **File**, **Save As** from the main menu. Choose your directory *matlab* and your subdirectory *examples*, and save this M-file as *junk.m*. Close or minimize your editor window. Finally, go to the top middle of your screen, where you'll find a box marked *Current Directory*. Change the path to .../*username*/matlab/examples, where by *username* I mean **your** username. Back at the command line, type simply *help junk*, and notice that the description you typed in as the header of your script file appears on the screen. Now, type *junk* at the prompt, and MATLAB will report that s=.0698. It has simply gone through your file line by line and executed each command as it came to it. One more useful command along these lines is *type*. Try the following:

```
>>type junk
```

The entire text of your file *junk.m* should appear on your screen. Since you've just finished typing this stuff in, this isn't so exciting, but try typing, for example, *type mean*. MATLAB will display its internal M-file *mean.m*. Perusing MATLAB's internal M-files like this is a great way to learn how to write them yourself. In fact, you can often tweak MATLAB's M-files into serving your own purposes. Simply use *type filename* in the Command Window, then choose and copy the M-file text using the **Edit** option. Finally, copy it into your own M-file and edit it. (Though keep in mind that if you ever publish a routine you've worked out this way, you need to acknowledge the source.)

3.6.2 Function M-files

The second type of M-file is called a *function* M-file and typically (though not inevitably) these will involve some variable or variables sent to the M-file and processed. As an example, let's suppose we want to solve the algebraic equation

$$e^x = x^2. \tag{3.1}$$

We begin by writing a function $f(x)$ that has zeros at solutions of (3.1). Here,

$$f(x) = e^x - x^2.$$

We now define the function $f(x)$ as a function M-file. To accomplish this, go back through the **File**, **New**, **M-file** mumbo jumbo as before, and create an M-file entitled *fname.m* with the following lines:

```
function f = fname(x);
%FNAME: computes f(x) = exp(x) - x^2
%call syntax: f=fname(x);
f = exp(x) - x^2;
```

Every function M-file begins with the command *function*. The next expression, *f* here, is the value the function returns, while the file name on the right-hand side of the equality is the name of the function file (with *.m* omitted). Finally, the input variable appears in parentheses. First, let's evaluate the function *fname* at a few values. At the command line prompt, type *fname(1)*, and MATLAB should return the value of the function at $x = 1$. Next, type $a = 0$, followed by *fname(a)*. MATLAB should return the value of the function at $x = 0$. Okay, enough of that. To solve this equation, (3.1), we will use a built-in MATLAB function *fzero*() (see also Section 3.2, above). At the command line prompt, type

```
x = fzero('fname', -.5)
```

to which MATLAB should respond by informing you that $x = -.7035$. Notice that -.5 served as an initial guess, and could have been found, for example, from a graph.

3.7 File Management from MATLAB

There are certain commands in MATLAB that will manipulate files on its primary directory. For example, if you happen to have the file *junk.m* in your working MATLAB directory, you can delete it simply by typing *delete junk.m* at the MATLAB command prompt. Much more generally, if you precede a command with an exclamation point, MATLAB will read it as a unix shell command (see Section 10 of these notes for more on Unix shell commands). So, for example, the three commands *!ls*, *!cp junk.m morejunk.m*, and *!ls* serve to list the contents of the directory you happen to be in, copy the file *junk.m* to the file *morejunk.m*, and list the files again to make sure it's there. Try it.

3.8 The Command Window

Occasionally, the Command Window will become too cluttered, and you will essentially want to start over. You can clear it by choosing **Edit**, **Clear Command Window**. Before doing this, you might want to save the variables in your workspace. This can be accomplished with the menu option **File**, **Save Workspace As**, which will allow you to save your workspace as a .mat file. Later, you can open this file simply by choosing **File**, **Open**, and selecting it. A word of warning, though: This does not save every command you have typed into your workspace; it only saves your variable assignments. For bringing all commands from a session back, see the discussion under *Command History*.

3.9 The Command History

The Command History window will open with each MATLAB session, displaying a list of recent commands issued at the prompt. Often, you will want to incorporate some of these old commands into a new session. A method slightly less gauche than simply cutting and pasting is to right-click on a command in the Command History window, and while holding the right mouse button down, to choose **Evaluate Selection**. This is exactly equivalent to typing your selection into the Command Window.

3.10 The MATLAB Workspace

As we've seen, MATLAB uses several types of data, and sometimes it can be difficult to remember what type each variable in your session is. Fortunately, this information is all listed for you in the MATLAB Workspace. Look in the upper left corner of your MATLAB window and see if your Workspace is already open. If not, choose **View**, **Workspace** from the main MATLAB menu and it should appear. Each variable you define during your session will be listed in the Workspace, along with its size and type. Observe the differences, for example, in the following variables.

```
>>t=5;
>>v=[1 2];
>>s='howdy'
>>y=solve('a*y=b')
```

4 Plots and Graphs in MATLAB

4.1 Simple x–y Plots

I really like the way MATLAB's graphics work. They're convenient, they're clear, and they're fast. Let's take a look. Have you ever heard the old saying, "To a guy with a hammer, everything's a nail." ? Well, as you might imagine from its name, for MATLAB everything's a matrix. Once you get used to it, this is pretty nice. Suppose, for example, that you know some values for y and you know some values for x and you want to make a plot of y versus x. The following commands (accompanied by MATLAB's output) suffice:

```
>>x=[1 2 3]

x =
```

```
     1     2     3

>>y=[4, 5, 6]

y =

     4     5     6

>>plot(x,y)
```

Observe that the elements in x are separated by spaces, while the elements in y are separated by commas. MATLAB will take either to mean the same thing. (Though remember from Section 3.1.1 that semicolons are a different matter entirely.) The output we obtain is the plot given as Figure 1.

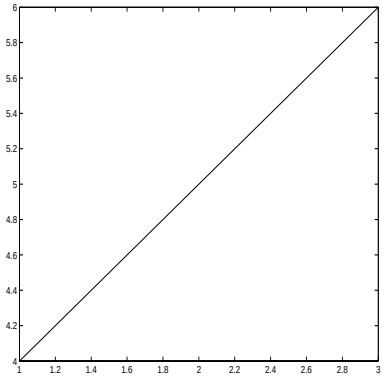

Figure 1: A very simple linear plot.

In MATLAB it's particularly easy to decorate a plot. For example, minimize your plot by clicking on the left button on the upper right corner of your window, then add the following lines in the Command Window:

```
>>xlabel('Here is a label for the x-axis')
>>ylabel('Here is a label for the y-axis')
>>title('Useless Plot')
>>axis([0 4 2 10])
```

The only command here that needs explanation is the last. It simply tells MATLAB to plot the x-axis from 0 to 4, and the y-axis from 2 to 10. If you now click on the plot's button at the bottom of the screen, you will get the labeled figure, Figure 2.

I added the legend after the graph was printed, using the menu options. Notice that all this labeling can be carried out and edited from these menu options. After experimenting a little, your plots will be looking great (or at least better than the default-setting figures displayed here). Not only can you label and detail your plots, you can write and draw on them directly from the MATLAB window. One warning: If you retype *plot(x,y)* after labeling, MATLAB will think you want to start over and will give you a clear figure with nothing except the line. To get your labeling back, use the up arrow key to scroll back through your commands and re-issue them at the command prompt. (Unless you labeled your plots using menu options, in which case you're out of luck, though this might be a good time to consult Section 4.6 on saving plots.)

Defining vectors as in the example above can be tedious if the vector has many components, so MATLAB has a number of ways to shorten your work. For example, you might try:

```
>>X=1:9

X =
```

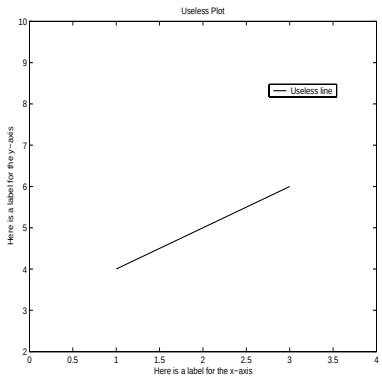

Figure 2: A still pretty much ridiculously simple linear plot.

```
    1     2     3     4     5     6     7     8     9

>>X=0:2:10

X =

    0     2     4     6     8    10
```

4.2 Plotting Functions

Plotting functions in MATLAB is almost as easy as plotting sets of points.

4.2.1 Ezplot

By far the easiest way to make simple plots in MATLAB is with the command *ezplot*. It's so easy to use, in fact, that I'm not going to say much about it. To get the idea, try the following commands:

```
>>ezplot('x^2-x')
>>ezplot('x^2 + 1',[-2 2])
>>ezplot('cos(t)','sin(t)',[0 2*pi])
```

(This last command plots *cos(t)* along the x-axis and *sin(t)* along the y-axis.) The function *ezplot* also works with function M-files. Suppose we have the following M-file, storing a function of the variable o (which you might find interesting for the ballistics project):

```
function d = dist(o);
%DIST: Computes distance as a function
%of angle for no air resistance.
v = 10.26;
H = .18;
g = 9.81;
d = (v*cos(o)/g).*(v*sin(o) + sqrt(v^2*sin(o).^2 + 2*g*H));
```

Notice, in particular, that I have vectorized it with .* and .^ where appropriate. We can now plot this function with the command

```
>>ezplot('dist',[0 pi/2])
```

I should stress, however, that for most of the figures you will want to create, the more general command *plot()* will be more useful. Speaking of which...

4.2.2 More General Methods of Plotting

Typically, you will want quite a bit more control over your plots than *ezplot* allows. Suppose, for example, we want to plot the function $f(x) = x^2$, say, for x in the domain [0,1]. First, we will partition the interval [0,1] into twenty evenly spaced points with the command, *linspace(0, 1, 20)*. (The command *linspace(a,b,n)* defines a vector with n evenly spaced points, beginning with right endpoint a and terminating with left endpoint b.) Then at each point, we will define f to be x^2. We have

```
>>x=linspace(0,1,20)

x =

  Columns 1 through 8

         0    0.0526    0.1053    0.1579    0.2105    0.2632    0.3158    0.3684

  Columns 9 through 16

    0.4211    0.4737    0.5263    0.5789    0.6316    0.6842    0.7368    0.7895

  Columns 17 through 20

    0.8421    0.8947    0.9474    1.0000

>>f=x.^2

f =

  Columns 1 through 8

         0    0.0028    0.0111    0.0249    0.0443    0.0693    0.0997    0.1357

  Columns 9 through 16

    0.1773    0.2244    0.2770    0.3352    0.3989    0.4681    0.5429    0.6233

  Columns 17 through 20

    0.7091    0.8006    0.8975    1.0000

>>plot(x,f)
```

Only three commands have been typed; MATLAB has done the rest. One thing you should pay close attention to is the line f=x.^2, where I've used one of the array operations from Section 3.1.1. This odd operation .^ is so critical to understand that I'm going to go through it one more time. It signifies that the vector x is not to be squared (a dot product, yielding a scalar), but rather that each component of x is to be squared and the result is to be defined as a component of f, another vector. Similar commands are .* and ./. These are referred to as *array operations,* and you will need to become comfortable with their use. (Or I'll just keep on nagging you.)

Suppose you have two functions of time $x(t)$ and $y(t)$ and you want to suppress t and plot y versus x (in calculus, you called this *parametrizing* your equations). For this example, we will take $x(t) = t^2 + 1$ and $y(t) = e^t$. One way to accomplish this is through solving for t in terms of x and substituing your result into $y(t)$ to get y as a function of x. Here, rather, we will simply get values of x and y at the same values of t. Using semicolons to suppress MATLAB's output, we have,

```
>>t=linspace(-1,1,40);
>>x=t.^2 + 1;
>>y=exp(t);
>>plot(x,y)
```

It is critical to notice that in the examples above, *f*, *x*, and *y* are *expressions* rather than *functions*. Here's a good way to see the difference: Try

```
>>y(1)
```

If *y* were a function of *t*, you would expect MATLAB to report that *ans = 2.7183*, the correct value of *e* to five digits. Instead, you get *ans = 0.3679*, which is the first entry in the vector *y*. In order to define the exponential as a function, type

```
>>f=inline('exp(x)','x')
>>f(1)
```

You can similarly define functions of multiple variables

```
>>g=inline('u^2 + v^2','u','v')

g =

    Inline function:
    g(u,v) = u^2 + v^2

>>g(1,2)

ans =

    5
```

or functions of vectors

```
>>f1=inline(vectorize('x^2'),'x')

f1 =

    Inline function:
    f1(x) = x.^2

>>x=[1 2]

x =

    1     2

>>f1(x)

ans =

    1     4
```

4.3 Juxtaposing One Plot On Top of Another

Suppose in the example above, you wanted to plot $x(t)$ and $y(t)$ on the same figure, both versus t. You need only type

```
>>plot(t,x,t,y);
```

Another way to accomplish this same thing is through the *hold on* command. After typing *hold on*, further plots will be typed one over the other until the command *hold off* is typed. For example, try

```
>>plot(t,x)^5
>>hold on
>>plot (t,y)
>>title('One plot over the other')
>>u=[-1 0 1];
>>v=[1 0 -1]
>>plot(u,v)
```

Click on the figure's button to bring it up. Now, try

```
>>hold off
>>plot(x,y)
```

4.4 Multiple Plots

Often, you will want MATLAB to draw two or more plots at the same time so that you can compare the behavior of various functions. For example, we might want to plot, $f(x) = x$, $g(x) = x^2$, and $h(x) = x^3$. The following sequence of commands produces the plot given in Figure 3.

```
>>x = linspace(0,1,20);
>>f = x;
>>g = x.^2;
>>h = x.^3;
>>subplot(3,1,1);
>>plot(x,f);
>>subplot(3,1,2);
>>plot(x,g);
>>subplot(3,1,3);
>>plot(x,h);
```

The only new command here is *subplot(m,n,p)*. This command creates m rows and n columns of graphs and places the current figure in position p (counted left to right, top to bottom).

4.5 Plotting Functions of Multiple Variables

MATLAB has many, many ways in which you can plot functions with multiple variables, of which I'll only mention one at this point.

4.5.1 Contour Plots

Contour plots are obtained when a sketch is made of the various regions in domain space along which some function is constant. Here, we will consider the function $f(x, y) = x^2 + y^2$. We observe that the region in domain space for which $f(x, y) \equiv 1$ corresponds with the circle $x^2 + y^2 = 1$. We type

```
>>[x y]=meshgrid(-3:0.1:3,-3:0.1:3);
>>contour(x,y,x.^2+y.^2)
>>axis square   %same scale on both axes
```

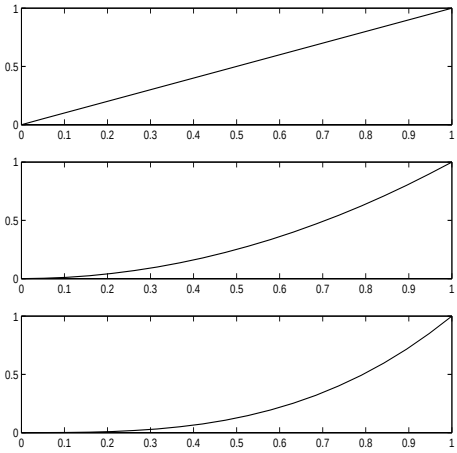

Figure 3: Algebraic functions on parade.

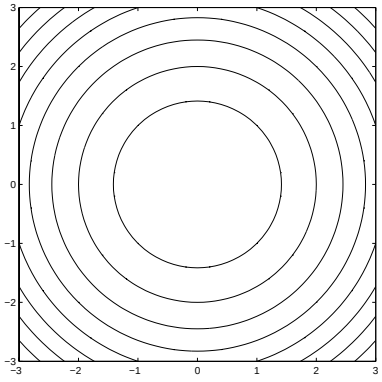

Figure 4: A very simple contour plot.

Observe that while Figure 4 is in black and white, MATLAB's different colors indicate different constants C for which $f(x, y) = C$.

4.6 Saving Plots as Encapsulated Postscript Files

Plots and graphs will constitute a large portion of your reports for M442. Fortunately, combining MATLAB and LyX, you will find that they are quite easy to incorporate. Once you have your plot sufficiently labeled, choose **File**, **Export** from the plot menu and save the plot as an *encapsulated postscript* file, with a .eps extension. Our course notes on LyX will explain how to draft .eps files into your reports.

Once saved as an encapsulated postscript file, you won't be able to edit your graph, so it's also a good idea to save it as a MATLAB figure, by choosing **File**, **Save As**, and saving it as a .fig file.

5 Matrices

We can't have a tutorial about a MATrix LABoratory without making at least a few comments about matrices. We have already seen how to define two matrices, the scalar, or 1×1 matrix, and the row or $1 \times n$ matrix (a row vector, as in Section 4.1). A column vector or matrix can be defined similarly by

```
>>x=[1; 2; 3]
```

This use of semicolons to end lines in matrices is standard, as we see from the following MATLAB input and output.

```
>>A=[1 2 3; 4 5 6; 7 8 9]

A =

    1    2    3
    4    5    6
    7    8    9

>>A(2,2)

ans =

    5

>>det(A)

ans =

    0

>>B=[1 2 2; 1 1 2; 0 3 3]

B =

    1    2    2
    1    1    2
    0    3    3

>>det(B)
```

[5] If a plot window pops up here, minimize it and bring it back up at the end.

```
ans =

    -3

>>B^(-1)

ans =

    1.0000   -0.0000   -0.6667
    1.0000   -1.0000         0
   -1.0000    1.0000    0.3333

>>A*B

ans =

     3    13    15
     9    31    36
    15    49    57

>>A.*B

ans =

     1     4     6
     4     5    12
     0    24    27
```

Note in particular the difference between $A * B$ and $A. * B$.

A convention that we will find useful while solving ordinary differential equations numerically is the manner in which MATLAB refers to the column or row of a matrix. With A still defined as above, $A(m, n)$ represents the element of A in the m^{th} row and n^{th} column. If we want to refer to the first row of A as a row vector, we use $A(1, :)$, where the colon represents that all columns are used. Similarly, we would refer to the second column of A as $A(:, 2)$. Some examples follow.

```
>>A(1,2)
ans =
2
>>A(2,1)
ans =
4
>>A(1,:)
ans =
1 2 3
>>A(:,2)
ans =
2
5
8
```

Finally, adding a prime (') to any vector or matrix definition transposes it (switches its rows and columns).

```
>>A'
ans =
1 4 7
```

```
2 5 8
3 6 9
>>X=[1 2 3]
X =
1 2 3
>>Y=X'
Y =
1
2
3
```

6 Miscellaneous Useful Commands

In this section I will give a list of some of the more obscure MATLAB commands that I find particularly useful. As always, you can get more information on each of these commands by using MATLAB's *help* command.

- *strcmp(str1,str2)* (string compare) Compares the strings *str1* and *str2* and returns logical true (1) if the two are identical and logical false (0) otherwise.
- *char(input)* Converts just about any type of variable *input* into a string (character array).
- *num2str(num)* Converts the numeric variable *num* into a string.
- *str2num(str)* Converts the string *str* into a numeric variable. (See also *str2double()*.)
- *strcat(str1,str2,...)* Horizontally concatenates the strings *str1*, *str2*, etc.

7 Graphical User Interface

Ever since 1984 when Apple's Macintosh computer popularized Douglas Engelbart's mouse-driven graphical computer interface, users have wanted something fancier than a simple command line. Unfortunately, actually coding this kind of thing yourself is a full-time job. This is where MATLAB's add-on GUIDE comes in. Much like Visual C, GUIDE is a package for helping you develop things like pop-up windows and menu-controlled files. To get started with GUIDE, simply choose **File**, **New**, **GUI** from MATLAB's main menu.

8 SIMULINK

SIMULINK is a MATLAB add-on taylored for visually modeling dynamical systems. To get started with SIMULINK, choose **File**, **New**, **Model**.

9 M-book

M-book is a MATLAB interface that passes through Microsoft Word, apparently allowing for nice presentations. Unfortunately, my boycott of anything Microsoft precludes the possibility of my knowing anything about it.

10 Useful Unix Commands

In Linux, you can manipulate files, create directories etc. using menu-driven software such as Konqueror (off the **Internet** sub-menu). Often, the fastest way to accomplish simple tasks is still from the Unix shell. To open the Unix shell on your machine, simply click on the terminal/seashell icon along the bottom of your

screen (or from the **System** sub-menu choose **Terminal**). A window should pop up with a prompt that looks something like: *[username]$*. Here, you can issue a number of useful commands, of which I'll discuss the most useful (for our purposes). (Commands are listed in bold, filenames and directory names in italics.)

- **cat** *filename* Prints the contents of a file *filename* to the screen.
- **cd** *dirname* Changes directory to the directory *dirname*
- **mkdir** *dirname* Creates a directory called *dirname*
- **cp** *filename1 filename2* Copies a file named *filename1* into a file name *filename2* (creating *filename2*)
- **ls** Lists all files in the current directory
- **rm** *filename* Removes (deletes) the file *filename*
- **quota** Displays the number of blocks your currently using and your quota. Often, when your account crashes, it's because your quota has been exceeded. Typically, the system will allow you to long into a terminal screen to delete files.
- **du -s *** Summarizes disk usage of all files and subdirectories
- **find . -name** **.tag* Finds all files ending .tag, in all directories
- **man ls** Opens the unix on-line help manual information on the command *ls*. (Think of it as typing *help ls*.) Of course, *man* works with any other command as well. (Use *q* to exit.)
- **man -k** *jitterbug* Searches the unix manual for commands involving the keyword jitterbug. (Oddly, there are no matches, but try, for example, *man copy*.)

10.1 Creating Unix Commands

Sometimes you will want to write your own Unix commands, which (similar to MATLAB's M-files) simply run through a script of commands in order. For example, use the editor of your choice (even MATLAB's will do) to create the following file, named *myhelp*.

```
#Unix script file with a list of useful commands
echo "Useful commands:"
echo
echo "cat: Prints the contents of a file to the screen"
echo "cd: Changes the current directory"
echo
echo "You can also issue commands with a Unix script."
ls
```

Any line in a Unix script file that begins with # is simply a comment and will be ignored. The command *echo* is Unix's version of *print*. Finally, any command typed in will be carried out. Here, I've used the list command. To run this command, type either simply *myhelp* if the Unix command path is set on your current directory or *~/myhelp* if the Unix command path is not set on your current directory.

10.2 More Help on Unix

Unix help manuals are among the fattest books on the face of the planet, and they're easy to find. Typically, however, you will be able to find all the information you need either in the on-line manual or on the Internet. One good site to get you started is *http://www.mcsr/olemiss/edu/unixhelp*.

References

[P] R. Pratap, *Getting Started with MATLAB 5: A Quick Introduction for Scientists and Engineers,* Oxford University Press, 1999.

[HL] D. Hanselman and B. Littlefield, *Mastering MATLAB 5: A Comprehensive Tutorial and Reference,* Prentice Hall, 1998.

[UNH] http://spicerack.sr.unh.edu/~mathadm/tutorial/software/matlab.

[HLR] B. R. Hunt, R. L. Lipsman, and J. M. Rosenberg (with K. R. Coombes, J. E. Osborn, and G. J. Stuck), *A Guide to MATLAB: for beginners and experienced users*, Cambridge University Press 2001.

[MW] http://www.mathworks.com

Index