MATLAB 7.0 Basics

P. Howard

Spring 2005

Contents

1 Introduction

MATLAB, which stands for MATrix LABoratory, is a software package developed by MathWorks, Inc. to facillitate numerical computations as well as some symbolic manipulation.[1] It strikes me as being slightly more difficult to begin working with than such packages as Maple, Mathematica, and Macsyma, 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.[2]

Beyond these notes (and other class hand-outs), I can suggest three useful resources for help with MATLAB: 1. the references listed at the end of these notes, the best of which for our purposes is probably the short book by Gilat, 2. MATLAB's extensive built-in help, which you can access from the MATLAB prompt with the command *helpdesk*, and 3. the Calclab help sessions in which you can typically find at least one or two people who not only know MATLAB but have been through this modeling course.

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

[1] The collection of programs (primarily in Fortran) that eventually became MATLAB were developed in the late 1970s by Cleve Moler, who used them in a numerical analysis course he was teaching at the University of New Mexico. Jack Little and Steve Bangert later reprogrammed these routines in C, and added M-files, toolboxes, and more powerful graphics (original versions created plots by printing asterisks on the screen). Moler, Little, and Bangert founded MathWorks in California in 1984.

[2] 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.

screen. Go to **Mathematics** and choose **Matlab**. Congratulations! (Alternatively, click on the surface plot icon at the foot of your screen.)[3]

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.

3 Simple Computations with MATLAB

3.1 What you're looking at

The (default) MATLAB screen is divided into three windows, with a large Command Window on the right, and two smaller windows stacked one atop the other on the left. The Command Window is where calculations are carried out in MATLAB, while the smaller windows display information about your current MATLAB session, your previous MATLAB sessions, and your computer account. Your options for these smaller windows are *Command History*, which displays the commands you've typed in from both the current and previous sessions, *Current Directory*, which shows which directory you're currently in and what files are in that directory, and *Workspace*, which displays information about each variable defined in your current session. You can choose which of these options you would like to have displayed by selecting **Desktop** from the main MATLAB window. Occasionally, it will be important that you are working in a certain directory. Notice that you can change MATLAB's working directory by double-clicking on a directory in the *Current Directory* window. In order to go backwards a directory, click on the folder with a black arrow on it in the top left corner of the *Current Directory* window.

3.2 Basic Computations

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

```
>>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:[4]

```
>>2+3+4+...
+5+6
ans =
    20
```

[3]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.

[4]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.* In order to conserve space, I have, for the most part, eliminated the vertical spacing MATLAB uses between expressions.

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).

3.2.1 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 $X * Y$:

```
>>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.

3.3 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*.

3.3.1 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
```

In general, the *eval()* command evaluates text strings or symbolic objects by executing them as MATLAB commands.

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
```

3.3.2 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 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)]
```

3.3.3 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
```

3.4 Basic Calculus

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

3.4.1 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
```

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
```

3.4.2 Integration

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

```
>>syms x;
>>int(x^2)
ans =
1/3*x^3
```

or

```
>>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 $\int_0^{\pi}\int_0^{\sin x}(x^2+y^2)dydx$, 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*, *quadv*, 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.

3.4.3 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
```

3.4.4 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
```

3.4.5 Taylor series

Certainly one of the most useful tools in mathematics is the Taylor expansion, whereby for certain functions 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.[5] 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
```

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

3.4.6 Maximization and Minimization

MATLAB has several built in tools for maximization and minimization. One of the most direct ways to find the maximum or minimum value of a function is directly from a MATLAB plot. In order to see how this works, create a simple plot of the function $f(x) = \sin x - \frac{2}{\pi}x$ for $x \in [0, \frac{\pi}{2}]$:

```
>>x=linspace(0,pi/2,25);
>>f=sin(x)-(2/pi)*x;
>>plot(x,f)
```

Now, in the graphics menu, choose **Tools, Zoom In**. Use the mouse to draw a box around the peak of the curve, and MATLAB will automatically redraw a refined plot. By refining carefully enough (and choosing a sufficient number of points in our *linspace* command), we can determine a fairly accurate approximation of the function's maximum value and of the point at which it is achieved.

In general, we will want a method more automated than manually zooming in on our solution. MATLAB has a number of built-in minimizers: *fminbnd()*, *fminunc()*, and *fminsearch()*. For straightfoward examples of each of these, use MATLAB's built-in help. For a more complicated example of *fminsearch()*, see Example 2.7 of our course notes *Modeling Basics*. In either case, we first need to study MATLAB M-files, so we will consider that topic next.

3.5 M-Files

3.5.1 Script M-Files

The heart of MATLAB lies in its use of M-files. We will begin with a *script* M-file, which is simply a text file that contains a list of valid MATLAB commands. 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):

```
%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. In this case, save the file as *junk.m*, and then close or minimize your editor window. 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.)

3.5.2 Working in Script M-files

Even if you're not writing a program, you will often find that the best way to work in MATLAB is through script M-files. Suppose, for example, that we would like to find that point at which the function $f(x) = xe^{-\frac{x^4}{1+x^2}}$ reaches its maximum value. Certainly, this is a straightforward calculation that we could carry out step by step in the Command Window, but instead, we will work entirely in an M-file (see template.m below). We begin our M-file with the commands *clear*, which clears the workspace of all variables, *clc*, which clears the Command Window of all previously issued commands, and *clf*, which clears the figure window. In addition, we delete the (assumed) diary file junk.m and restart it with the command *diary junk.out*. Finally, we set *echo on* so that the commands we type here will appear in the command window. We now type in all commands required for plotting and maximizing the function $f(x)$.

```
%TEMPLATE: MATLAB M-file containing a convenient
%workplace template.
clear; clc; clf;
delete junk.out
diary junk.out
echo on
%
fprime = diff(x*exp(-x^4/(1+x^2)))
pretty(fprime)
r = eval(solve(fprime))
%
diary off
echo off
```

The only new command in *template.m* is *pretty(),* which simply instructs MATLAB to present the expression *fprime* in a more readable format. Once *template.m* has been run, the diary file *junk.out* will be created. Observe that the entire session is recorded, both inputs and outputs. It's also worth noting that while *pretty()* makes expressions look better in the MATLAB Command Window, its output in diary files is regrettable.

```
%
fprime = diff(x*exp(-x^4/(1+x^2)))
fprime =
exp(-x^4/(1+x^2))+x*(-4*x^3/(1+x^2)+2*x^5/(1+x^2)^2)*exp(-x^4/(1+x^2))
pretty(fprime)
4 / 3 5 \ 4
x | x x | x
exp(- ——) + x |-4 —— + 2 ———| exp(- ——)
2 | 2 2 2| 2
1 + x \ 1 + x (1 + x ) / 1 + x
r = eval(solve(fprime))
r =
0.8629 - 0.0000i
-0.8629 + 0.0000i
0.0000 - 1.3745i
-0.0000 + 1.3745i
-0.0000 - 0.5962i
0.0000 + 0.5962i
%
diary off
```

3.5.3 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.

Function M-files need neither accept input nor return output. For example, the function M-file *noout.m*, below, displays output to the screen, but does not actually return a value.

```
function noout(x)
%NOOUT: Function M-file that takes input
%but does not return output.
x^2
```

On the other hand, the function M-file *noin.m* does not accept any input, but returns a value.

```
function value = noin
%NOIN: Function M-file that takes no input
%but does return output.
x = 2;
value = x^2;
```

Function M-files can have subfunctions (script M-files cannot have subfunctions). In the following example, the subfunction *subfun* simply squares the input x.

```
function value = subfunex(x)
%SUBFUNEX: Function M-file that contains a subfunction
value = x*subfun(x);
%
function value = subfun(x)
%SUBFUN: Subfunction that computed x^2
value = x^2;
```

For more information about script and function M-files, see Section 6 of these notes, on Programming in MATLAB.

3.5.4 Debugging M-files

Since MATLAB views M-files as computer programs, it offers a handful of tools for debugging. First, from the M-file edit window, an M-file can be saved and run by clicking on the icon with the white sheet and downward-directed blue arrow (alternatively, choose **Debug, Run** or simply type **F5**). By setting your cursor on a line and clicking on the icon with the white sheet and the red dot, you can set a marker at which MATLAB's execution will stop. A green arrow will appear, marking the point where MATLAB's execution has paused. At this point, you can step through the rest of your M-file one line at a time by choosing the *Step* icon (alternatively **Debug, Step** or **F6**).

Unless you're a phenomenal programmer, you will occasionally write a MATLAB program (M-file) that has no intention of stopping any time in the near future. You can always abort your program by typing **Control-c**, though you must be in the MATLAB Command Window for MATLAB to pay any attention to this.

3.6 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 11 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.

3.7 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*.

3.8 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.

3.9 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

The primary tool we will use for plotting in MATLAB is *plot()*. In order to see how this function works, suppose we would like to plot the line that passes through the points $\{(1,4),(3,6)\}$. We first define the x values as the vector $x = (1,3)$ and the y values as the vector $y = (4,6)$, and then we plot these points, connecting them with a line. The following commands (accompanied by MATLAB's output) suffice:

```
>>x=[1 3]
x =
    1     3
>>y=[4 6]
y =
    4     6
>>plot(x,y)
```

The output we obtain is the plot given as Figure 1.

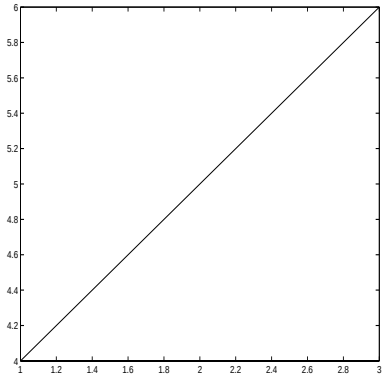

Figure 1: A very simple linear plot.

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:

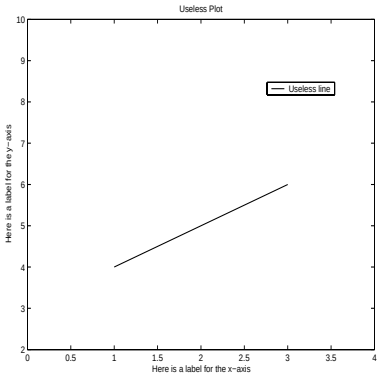

Figure 2: A still pretty much ridiculously simple linear plot.

```
>>X=1:9
X =
    1    2    3    4    5    6    7    8    9
>>X=0:2:10
X =
    0    2    4    6    8   10
```

4.2 Plotting Functions

4.2.1 Plotting functions with the plot command

Suppose 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 we have two functions of time $x(t) = t^2 + 1$ and $y(t) = e^t$, and we want to suppress t and plot y versus x (you probably recall that we refer to this as *parametrizing* our equations). 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*, we would expect MATLAB to report that *ans = 2.7183*, the correct value of *e* to five digits. Instead, we get *ans = 0.3679*, which is the first entry in the vector *y*. In order to define the exponential as a function, we type

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

We 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
```

4.2.2 Ezplot

Another method for plottting functions in MATLAB is with the command *ezplot().* In this case, we need only specify the function we would like to plot as an argument of *ezplot().* For example, we have the following:

```
>>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 the function $f(x) = sin(e^x)$:

```
function y = ezexample(x)
%EZEXAMPLE: MATLAB function M-file that returns values
%of the function sin(exp(x)).
y = sin(exp(x));
```

We can now plot this function with the command

```
>>ezplot(@ezexample,[0 pi/2])
```

4.3 Juxtaposing One Plot On Top of Another

Suppose in the example in the previous section in which $x(t) = t^2 + 1$ and $y(t) = e^t$, we wanted to plot $x(t)$ and $y(t)$ on the same figure, both versus t. We 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,

```
>>plot(t,x)[6]
>>hold on
>>plot (t,y)
>>title('One plot over the other')
>>u=[-1 0 1];
>>v=[1 0 -1]
>>plot(u,v)
```

4.4 Multiple Plots

Often, we will want MATLAB to draw two or more plots at the same time so that we 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 we can plot functions with multiple variables, of which I'll only mention one at this point.

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

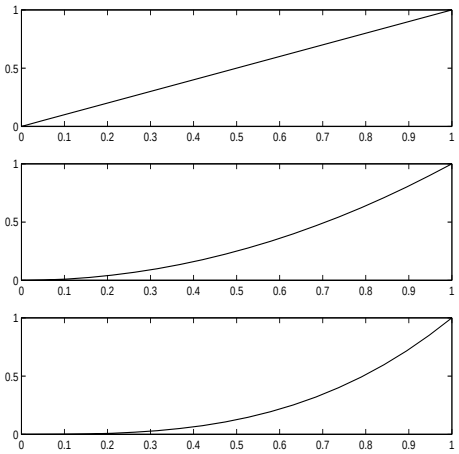

Figure 3: Algebraic functions on parade.

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
```

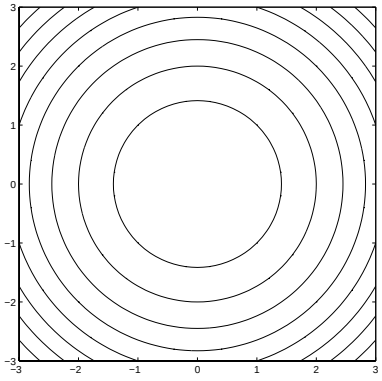

Figure 4: A very simple contour plot.

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 (from the options in your graphics box) **File**, **Save As**, and change **Save as type** to **EPS file**. Finally, click on the **Save** button. 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 (which is MATLAB's default).

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)
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 Programming in MATLAB

6.1 Overview

Perhaps the most useful thing about MATLAB is that it provides an extraordinarily convenient platform for writing your own programs. Every time you create an M-file you are writing a computer program using the MATLAB programming language. If you are familiar with C or C++, you will find programming in MATLAB very similar.[7] And if you are familiar with any programming language—Fortran, Pascal, Basic, even antiques like Cobol—you shouldn't have much trouble catching on. In this section, I will run through the basic commands you will need to get started in programming. Some of these you have already seen in one form or another on previous pages.

6.2 Loops

6.2.1 The For Loop

One of the simplest and most fundamental structures is the *for*-loop, exemplified by the MATLAB code,

```
f=1;
for n=2:5
f=f*n
end
```

[7] In fact, it's possible to incorporate C or C++ programs into your MATLAB document.

The output for this loop is given below.

```
f =
     2
f =
     6
f =
    24
f =
   120
```

Notice that I've dropped off the command prompt arrows, because typically this kind of structure is typed into an M-file, not in the Command Window. I should point out, however, that you can type a for-loop directly into the command line. What happens is that after you type *for n=2:5*, MATLAB drops the prompt and lets you close the loop with *end* before responding. By the way, if you try typing this series of commands in MATLAB's default editor, it will space things for you to separate them and make the code easier to read. One final thing you should know about *for* is that if you want to increment your index by something other than 1, you need only type, for example, *for k=4:2:50,* which counts from 4 (the first number) to 50 (the last number) by increments of 2.

6.2.2 The While Loop

One problem with for-loops is that they generally run a predetermined set of times. While-loops, on the other hand, run until some criterion is no longer met. We might have

```
x=1;
while x<3
x=x+1
if x > 100
break
end
end
```

The output for this loop is given below.

```
x =
     2
x =
     3
```

Since while-loops don't necessarily stop after a certain number of iterations, they are notorious for getting caught in infinite loops. In the example above I've stuck a *break* command in the loop, so that if I've done something wrong and x gets too large, the loop will be broken.

6.3 Branching

Typically, we want a program to run down different paths for different cases. We refer to this as branching.

6.3.1 If-Else Statements

The most standard branching statement is the *if-else*. A typical example, without output, is given below.

```
if x > 0
    y = x;
elseif x == 0
    y = -1;
else
    y = 0;
end
```

The spacing here is MATLAB's default. Notice that when comparing x with 0, we use == instead of simply =. This is simply an indication that we are comparing x with 0, not setting x equal to 0. The only other operator that probably needs special mention is ~= for *not equal*. You probably wouldn't be surprised to find out what things like <=, <, >=, > mean. Finally, observe that *elseif* should be typed as a single word. MATLAB will run files for which it is written as two words, but it will read the *if* in that case as beginning an entirely new loop (inside your current loop).

6.3.2 Switch Statements

A second branching statement in MATLAB is the *switch* statement. *Switch* takes a variable—x in the case of the example below—and carries out a series of calculations depending on what that variable is. In this example, if x is 7, the variable y is set to 1, while if x is 10 or 17, then y is set to 2 or 3 respectively.

```
switch x
case 7
   y = 1;
case 10
   y = 2;
case 17
   y = 3;
end
```

6.4 Subfunctions

Function files can have their own subfunction files. We could write the M-file,

```
function y = sumcuberoots(x)
y=sum(cuberoot(x));
%Subfunction starts here
function z = cuberoot(x)
z=sign(x).*abs(x).^(1/3)
```

One thing you might want to know about subfunctions is that script M-files cannot contain them.

6.5 Input and Output

6.5.1 Parsing Input and Output

Often, you will find it useful to make statements contingent upon the number of arguments flowing in to or out of a certain function. For this purpose, MATLAB has *nargin* and *nargout*, which provide the number of input arguments and the number of output arguments respectively. The following function, *addthree*, accepts up to three inputs and adds them together. If only one input is given, it says it cannot add only one number. On the other hand, if two or three inputs are given, it adds what it has. We have

```
function s=addthree(x, y, z)
%ADDTHREE: Example for nargin and nargout
if nargin < 2
   error('Need at least two inputs for adding')
end
if nargin == 2
   s=x+y;
else
   s = x+y+z;
end
```

Working at the command prompt, now, we find,

```
>>addthree(1)
??? Error using ==> addthree
Need at least two inputs for adding
>>addthree(1,2)
ans =
     3
>>addthree(1,2,3)
ans =
     6
```

Notice that the error statement is the one we supplied.

I should probably mention that a function need not have a fixed number of inputs. The command *varargin* allows for as many inputs as the user will supply. For example, the following simple function adds as many numbers as the user supplies:

```
function s=addall(varargin)
%ADDALL: Example for nargin and nargout
s=sum([varargin{:}]);
```

Working at the command prompt, we find

```
>>addall(1)
ans =
     1
>>addall(1,2)
ans =
     3
>>addall(1,2,3,4,5)
ans =
    15
```

6.5.2 Screen Output

Part of programming is making things user-friendly in the end, and this means controlling screen output. MATLAB's simplest command for writing to the screen is *disp*.

```
>>x = 2+2;
>>disp(['The answer is ' num2str(x) '.'])
The answer is 4.
```

In this case, *num2str()* converts the number x into a string appropriate for printing.

6.5.3 Screen Input

Often, you will want the user to enter some type of data into your program. Some useful commands for this are *pause*, *keyboard*, and *input*. *Pause* suspends the program until the user hits a key, while *keyboard* allows the user to enter MATLAB commands until he or she types *return*. As an example, consider the M-file

```
%EXAMPLE: A script file with examples of
%pause, keyboard, and input
disp('Hit any key to continue...')
pause
disp('Enter a command.  (Type "return" to return to the script file)')
keyboard
answer=input(['Are you tired of this yet (yes/no)?'], 's');
if isequal (answer,'yes')
   return
end
```

Working at the command prompt, we have,

```
>>example
Hit any key to continue...
Enter a command.  (Type "return" to return to the script file)
>>3+4
ans =
    7
>>return
Are you tired of this yet (yes/no)?yes
```

6.5.4 Screen Input on a Figure

The command *ginput* can be used to put input onto a plot or graph. The following function M-file plots a simple graph and lets the user put an x on it with a mouse click.

```
function example
%EXAMPLE: Marks a spot on a simple graph
p=[1 2 3];
q=[1 2 3];
plot(p,q);
hold on
disp('Click on the point where you want to plot an x')
[x y]=ginput(1);    %Gives x and y coordinates to point
plot(x,y,'Xk')
hold off
```

7 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.

8 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.

9 SIMULINK

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

10 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.

11 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*.)

11.1 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.

11.2 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*.

12 FAQs

1. How do I create and save a MATLAB diary session? Believe it or not, this question is answered in these notes. Unfortunately, finding that answer is a little bit like playing a game of *Where's Waldo?* See footnote 3 on page 3.

References

[G] A. Gilat, *MATLAB: An Introduction with Applications*, John Wiley & Sons, Inc. 2004.

[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