MATLAB 7.8 Basics

P. Howard

Fall 2009

Contents

1 Introduction

1.1 The Origin of MATLAB

MATLAB, which stands for MATrix LABoratory, is a software package developed by MathWorks, Inc. to facillitate numerical computations as well as some symbolic manipulation. 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

1.2 Starting MATLAB at Texas A&M University

1.2.1 MATLAB in a calclab room

Rooms such as Blocker 123 are designated calclab rooms, and your NetID and password should directly access your calclab account. Log in and click on the green sphere 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.)

1.2.2 MATLAB in an Open Access Computer Lab

Though MATLAB is available directly on your open access account it is often more convenient to work on your calclab account. You can access your calclab account from any open access lab with the NX client. From the Start menu, choose **All Programs**, **Communications**, **NX Client for Windows**, then select as your host calclab1.math.tamu.edu. In the process of setting up this connection you can create an icon on your desktop that will automatically connect to calclab.

1.3 The MATLAB Interface

The (default) MATLAB screen is divided into four windows, with a large *Command Window* in the middle, a *Current Directory* window on the left, and two smaller windows (*Workspace* and *Command History*) stacked one atop the other on the right. 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. The Command History displays the commands you've typed in from both the current and previous sessions; the Current Directory shows which directory you're currently in and what files are in that directory, and the Workspace window 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 and left-clicking on the option. (MATLAB will place a black check to the left of this option.) Occasionally, it will be important that you are working in a certain directory. 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.

1.4 Basic Computations

At the prompt, designated by two arrows, >>, type *2 + 2* and press **Enter**. You should find that the answer has been assigned to the default variable *ans*. Next, type *2+2;* and hit Enter. Notice that 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.) Next, 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. (The left and right arrow keys will move the cursor left and right along the current line.)

Occasionally, you will find that an expression you are typing is getting inconveniently long and needs to be continued to the next line. You can accomplish this by putting in three dots and typing **Enter**. Try the following:[1]

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

[1]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, >>.

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.

1.5 Variable Types

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 2), *character string*, and *inline function* .

A list of all active variables—along with size and type—is given in the *Workspace*. Observe the differences, for example, in the descriptions given for each of the following variables.

```
>>t=5;
>>v=1:25;
>>s='howdy'
>>y=solve('a*y=b')
```

1.6 Diary Files

For many of the assignments this semester, and also for the projects, you will need to turn in a log of MATLAB commands typed and of MATLAB's responses. This is straightforward in MATLAB with the *diary* command.

Example 1.1. Write a MATLAB script that sets $x = 1$ and computes $\tan^{-1} x$ (or $\arctan x$). Save the script to a file called *script1.txt* and print it.

In order to accomplish this, we use the following MATLAB commands.

```
>>diary script1.txt
>>x=1
x =
1
>>atan(1)
ans =
0.7854
>>diary off
```

In this script, the command *diary script1.txt* creates the file *script1.txt,* and MATLAB begins recording the commands that follow, along with MATLAB's responses. When the command *diary off* is typed, MATLAB writes the commands and responses to the file *script1.txt.* Commands typed after the *diary off* command will no longer be recorded, but the file *script1.txt* can be reopened either with the command *diary on* or with *diary script1.txt.* Finally, the diary file *script1.txt* can be deleted with the command *delete script1.txt.*

In order to print *script1.txt*, follow the *xprint* instructions posted in the Blocker lab. More precisely, open a terminal window by selecting the terminal icon from the bottom of your screen and use the *xprint* command

xprint -d blocker script1.txt

You will be prompted to give your NetID (neo account ID) and password. The file will be printed in Blocker 130. △

1.7 Clearing and Saving the Command Window

The Command Window can be cleared with the command *clc*, which leaves your variable definitions in place. You can delete your variable definitions with the command *clear*. All variables in a MATLAB session can be saved 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*.

1.8 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. An easy way to accomplish this is as follows: right-click on the command in the Command History, and while holding the right mouse button down, choose **Evaluate Selection**. This is exactly equivalent to typing your selection into the Command Window.

1.9 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 13 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.

1.10 Getting Help

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?)

2 Symbolic Calculations in MATLAB

Though MATLAB has not been designed with symbolic calculations in mind, it can carry them out with the Symbolic Math Toolbox, which is standard with student versions. (In order to check if this, or any other toolbox is on a particular version of MATLAB, type *ver* at the MATLAB prompt.) In carrying out these calculations, MATLAB uses Maple software, but the user interface is significantly different.

2.1 Defining Symbolic Objects

Symbolic manipulations in MATLAB are carried out on symbolic variables, which can be either particular numbers or unspecified variables. The easiest way in which to define a variable as symbolic is with the *syms* command.

Example 2.1. Suppose we would like to symbolically define the logistic model

$$R(N) = aN(1 - \frac{N}{K}),$$

where N denotes the number of individuals in a population and R denotes the growth rate of the population. First, we define both the variables and the parameters as symbolic objects, and then we write the equation with standard MATLAB operations:

```
>>syms N R a K
>>R=a*N*(1-N/K)
R =
a*N*(1-N/K)
```

Here, the expressions preceded by >> have been typed at the command prompt and the others have been returned by MATLAB. △

Symbolic objects can also be defined to take on particular numeric values.

Example 2.2. Suppose that we want a general form for the logistic model, but we know that the carrying capacity K is 10, and we want to specify this. We can use the following commands:

```
>>K=sym(10)
K =
10
>>R=a*N*(1-N/K)
R =
a*N*(1-1/10*N)
```

2.1.1 Complex Numbers

You can also define and manipulate symbolic complex numbers in MATLAB.

Example 2.3. Suppose we would like to define the complex number $z = x + iy$ and compute z^2 and $z\bar{z}$. We use

```
>>syms x y real
>>z=x+i*y
z =
x+i*y
>>square=expand(z^2)
square =
x^2+2*i*x*y-y^2
>>zzbar=expand(z*conj(z))
zzbar =
x^2+y^2
```

Here, we have particularly specified that x and y be real, as is consistent with complex notation. The built-in MATLAB command *conj* computes the complex conjugate of its input, and the *expand* command is required in order to force MATLAB to multiply out the expressions. (The *expand* command is discussed more below in Subsubsection 2.2.2.)

2.1.2 The *Clear* Command

You can clear variable definitions with the *clear* command. For example, if x is defined as a symbolic variable, you can type *clear x* at the MATLAB prompt, and this definition will be removed. (*Clear* will also clear other MATLAB data types.) If you have set a symbolic variable to be *real*, you will additionally need to use *syms x unreal* or the Maple kernel that MATLAB calls will still consider the variable real.

2.2 Manipulating Symbolic Expressions

Once an expression has been defined symbolically, MATLAB can manipulate it in various ways.

2.2.1 The *Collect* Command

The *collect* command gathers all terms together that have a variable to the same power.

Example 2.4. Suppose that we would like organize the expression

$$f(x) = x(\sin x + x^3)(e^x + x^2)$$

by powers of x. We use

```
>>syms x
>>f=x*(sin(x)+x^3)*(exp(x)+x^2)
f =
x*(sin(x)+x^3)*(exp(x)+x^2)
>>collect(f)
ans =
x^6+exp(x)*x^4+sin(x)*x^3+sin(x)*exp(x)*x
```

△

2.2.2 The *Expand* Command

The *expand* command carries out products by distributing through parentheses, and it also expands logarithmic and trigonometric expressions.

Example 2.5. Suppose we would like to expand the expression

$$f(x) = e^{x+x^2}.$$

We use

```
>>syms x
>>f=exp(x+x^2)
f =
exp(x+x^2)
>>expand(f)
ans =
exp(x)*exp(x^2)
```

△

2.2.3 The *Factor* Command

The *factor* command can be used to factor polynomials.

Example 2.6. Suppose we would like to factor the polynomial

$$f(x) = x^4 - 2x^2 + 1.$$

We use

```
>syms x
>f=x^4-2*x^2+1
f =
x^4-2*x^2+1
>factor(f)
ans =
(x-1)^2*(x+1)^2
```

△

2.2.4 The *Horner* Command

The *horner* command is useful in preparing an expression for repeated numerical evaluation. In particular, it puts the expression in a form that requires the least number of arithmetic operations to evaluate.

Example 2.7. Re-write the polynomial from Example 6 in Horner form.

```
>>syms x
>>f=x^4-2*x^2+1
f =
x^4-2*x^2+1
>>horner(f)
ans =
1+(-2+x^2)*x^2
```

2.2.5 The *Simple* Command

The *simple* command takes a symbolic expression and re-writes it with the least possible number of characters. (It runs through MATLAB's various manipulation programs such as *collect*, *expand*, and *factor* and returns the result of these that has the least possible number of characters.)

Example 2.8. Suppose we would like a reduced expression for the function

$$f(x) = (1 + \frac{1}{x} + \frac{1}{x^2})(1 + x + x^2).$$

We use

```
>>syms x f
>>f=(1+1/x+1/x^2)*(1+x+x^2)
f =
(1+1/x+1/x^2)*(x+1+x^2)
>>simple(f)
simplify:
(x+1+x^2)^2/x^2
radsimp:
(x+1+x^2)^2/x^2
combine(trig):
(3*x^2+2*x+2*x^3+1+x^4)/x^2
factor:
(x+1+x^2)^2/x^2
expand:
2*x+3+x^2+2/x+1/x^2
combine:
(1+1/x+1/x^2)*(x+1+x^2)
```

```
convert(exp):
(1+1/x+1/x^2)*(x+1+x^2)
convert(sincos):
(1+1/x+1/x^2)*(x+1+x^2)
convert(tan):
(1+1/x+1/x^2)*(x+1+x^2)
collect(x):
2*x+3+x^2+2/x+1/x^2
mwcos2sin:
(1+1/x+1/x^2)*(x+1+x^2)
ans =
(x+1+x^2)^2/x^2
```

In this example, three lines have been typed, and the rest is MATLAB output as it tries various possibilities. In returns the expression in *ans*, in this case from the *factor* command. △

2.2.6 The *Pretty* Command

MATLAB's *pretty* command simply re-writes a symbolic expression in a form that appears more like typeset mathematics than does MATLAB syntax.

Example 2.9. Suppose we would like to re-write the expression from Example 3.8 in a more readable format. Assuming, we have already defined f as in Example 3.8, we use *pretty(f)* at the MATLAB prompt. (The output of this command doesn't translate well into a printed document, so I won't give it here.)

2.3 Solving Algebraic Equations

MATLAB's built-in function for solving equations symbolically is *solve*.

Example 2.10. Suppose we would like to solve the quadratic equation

$$ax^2 + bx + c = 0.$$

We use

```
>>syms a b c x
>>eqn=a*x^2+b*x+c
eqn =
a*x^2+b*x+c
>>roots=solve(eqn)
roots =
1/2/a*(-b+(b^2-4*a*c)^(1/2))
1/2/a*(-b-(b^2-4*a*c)^(1/2))
```

Observe that we only defined the expression on the left-hand side of our equality. By default, MATLAB's *solve* command sets this expression to 0. Also, notice that MATLAB knew which variable to solve for. (It takes x as a default variable.) Suppose that in lieu of solving for x, we know x and would like to solve for a. We can specify this with the following commands:

```
>>a=solve(eqn,a)
a =
-(b*x+c)/x^2
```

In this case, we have particularly specified in the *solve* command that we are solving for a. Alternatively, we can type an entire equation directly into the solve command. For example:

```
>>syms a
>>roots=solve(a*x^2+b*x+c)
roots =
1/2/a*(-b+(b^2-4*a*c)^(1/2))
1/2/a*(-b-(b^2-4*a*c)^(1/2))
```

Here, the *syms* command has been used again because a has been redefined in the code above. Finally, we need not first make our variables symbolic if we put the expression in *solve* in single quotes. We could simply use *solve('a*x^2+b*x+c')*. △

MATLAB's *solve* command can also solve systems of equations.

Example 2.11. For a population of prey x with growth rate R_x and a population of predators y with growth rate R_y, the the Lotka–Volterra predator–prey model is

$$\begin{aligned} R_x &= ax - bxy \\ R_y &= -cy + dxy. \end{aligned}$$

In this example, we would like to determine whether or not there is a pair of population values (x, y) for which neither population is either growing or decaying (the rates are both 0). We call such a point an equilibrium point. The equations we need to solve are:

$$\begin{aligned} 0 &= ax - bxy \\ 0 &= -cy + dxy. \end{aligned}$$

In MATLAB

```
>>syms a b c d x y
>>Rx=a*x-b*x*y
Rx =
a*x-b*x*y
>>Ry=-c*y+d*x*y
Ry =
-c*y+d*x*y
>>[prey pred]=solve(Rx,Ry)
```

```
prey =
0
1/d*c
pred =
0
1/b*a
```

Again, MATLAB knows to set each of the expression Rx and Ry to 0. In this case, MATLAB has returned two solutions, one with $(0, 0)$ and one with $(\frac{c}{d}, \frac{a}{b})$. In this example, the appearance of *[prey pred]* particularly requests that MATLAB return its solution as a vector with two components. Alternatively, we have the following:

```
>>pops=solve(Rx,Ry)
pops =
x: [2x1 sym]
y: [2x1 sym]
>>pops.x
ans =
0
1/d*c
>>pops.y
ans =
0
1/b*a
```

In this case, MATLAB has returned its solution as a MATLAB *structure*, which is a data array that can store a combination of different data types: symbolic variables, numeric values, strings etc. In order to access the value in a *structure*, the format is

```
structure_name.variable_identification
```
△

2.4 Numerical Calculations with Symbolic Expressions

In many cases, we would like to combine symbolic manipulation with numerical calculation.

2.4.1 The *Double* and *Eval* commands

The *double* and *eval* commands change a symbolic variable into an appropriate double variable (i.e., a numeric value).

Example 2.12. Suppose we would like to symbolically solve the equation $x^3 + 2x - 1 = 0$, and then evaluate the result numerically. We use

```
>>syms x
>>r=solve(x^3+2*x-1);
>>eval(r)
ans =
0.4534
-0.2267 + 1.4677i
-0.2267 - 1.4677i
>>double(r)
ans =
0.4534
-0.2267 + 1.4677i
-0.2267 - 1.4677i
```

MATLAB's symbolic expression for r is long, so I haven't included it here, but you should take a look at it by leaving the semicolon off the *solve* line. △

2.4.2 The Subs Command

In any symbolic expression, values can be substituted for symbolic variables with the *subs* command.

Example 2.13. Suppose that in our logistic model

$$R(N) = aN(1 - \frac{N}{K}),$$

we would like to substitute the values $a = .1$ and $K = 10$. We use

```
>>syms a K N
>>R=a*N*(1-N/K)
R =
a*N*(1-N/K)
>>R=subs(R,a,.1)
R =
1/10*N*(1-N/K)
>>R=subs(R,K,10)
R =
1/10*N*(1-1/10*N)
```

Alternatively, numeric values can be substitued in. We can accomplish the same result as above with the commands

```
>>syms a K N
>>R=a*N*(1-N/K)
R =
a*N*(1-N/K)
>>a=.1
```

```
a =
0.1000
>>K=10
K =
10
>>R=subs(R)
R =
1/10*N*(1-1/10*N)
```

In this case, the specifications $a = .1$ and $K = 10$ have defined a and K as numeric values. The *subs* command, however, places them into the symbolic expression.

3 Plots and Graphs in MATLAB

The primary tool we will use for plotting in MATLAB is *plot()*.

Example 3.1. Plot the line that passes through the points $\{(1,4),(3,6)\}$.

We first define the x values (1 for the first point and 3 for the second) as a single variable $x = (1,3)$ (typically referred to as a *vector*) 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.

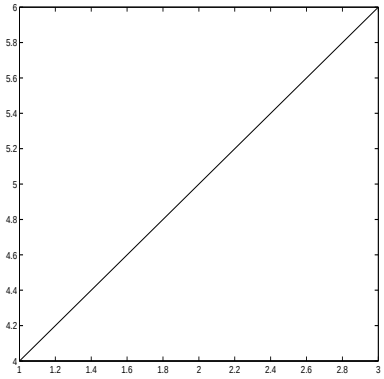

Figure 1: A very simple linear plot.

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

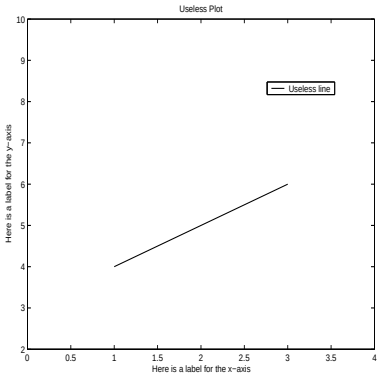

Figure 2: A still pretty much ridiculously simple linear plot.

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 3.6 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 =
    1    2    3    4    5    6    7    8    9
>>X=0:2:10
X =
    0    2    4    6    8   10
```

3.1 Plotting Functions with the *plot* command

In order to plot a function with the *plot* command, we proceed by evaluating the function at a number of x-values $x_1, x_2, ..., x_n$ and drawing a curve that passes through the points $\{(x_k, y_k)\}_{k=1}^{n}$, where $y_k = f(x_k)$.

Example 3.2. Use the *plot* command to plot the function $f(x) = x^2$ for $x \in [0, 1]$.

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 left endpoint a and terminating with right 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 we have used the *array operation* .^. This operation .^ 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. △

Example 3.3. In our section on symbolic algebra, we encountered the logistic population model, which relates the number of individuals in a population N with the rate of growth of the population R through the relationship

$$R(N) = aN(1 - \frac{N}{K}) = -\frac{a}{K}N^2 + aN.$$

Taking $a = 1$ and $K = 10$, we have

$$R(N) = -.1N^2 + N.$$

In order to plot this for populations between 0 and 20, we use the following MATLAB code, which creates Figure 3.

```
>>N=linspace(0,20,1000);
>>R=-.1*N.^2+N;
>>plot(N,R)
```

Observe that the rate of growth is positive until the population achieves its "carrying capacity" of $K = 10$ and is negative for all populations beyond this. In this way, if the population is initially below its carrying capacity, then it will increase toward its carrying capacity, but will never exceed it. If the population is initially above the carrying capacity, it will decrease toward the carrying capacity. The carrying capacity is interpreted as the maximum number of individuals the environment can sustain. △

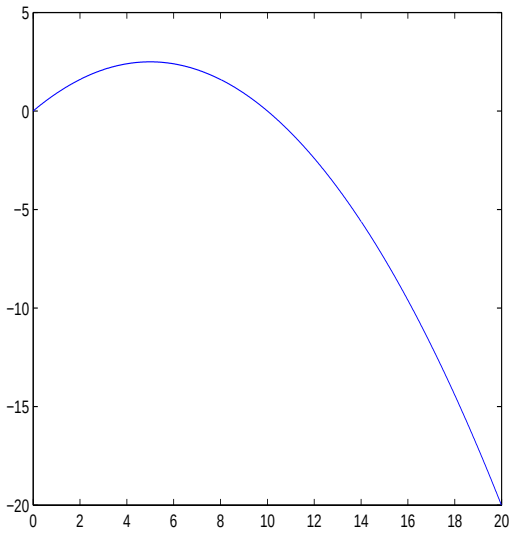

Figure 3: Growth rate for the logistic model.

3.2 Parametric Curves

In certain cases the relationship between x and y can be described in terms of a third variable, say t. In such cases, t is a parameter, and we refer refer to a plot of the points (x, y) as a parametric curve.

Example 3.4. Plot a curve in the x-y plane corresponding with $x(t) = t^2 + 1$ and $y(t) = e^t$, for $t \in [-1, 1]$. 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 use the following script, which creates Figure 4.

```
>>t=linspace(-1,1,100);
>>x=t.^2 + 1;
>>y=exp(t);
>>plot(x,y)
```

△

3.3 Juxtaposing One Plot On Top of Another

Example 3.5. For the functions $x(t) = t^2 + 1$ and $y(t) = e^t$, plot $x(t)$ and $y(t)$ on the same figure, both versus t.

The easiest way to accomplish this is with the single command

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

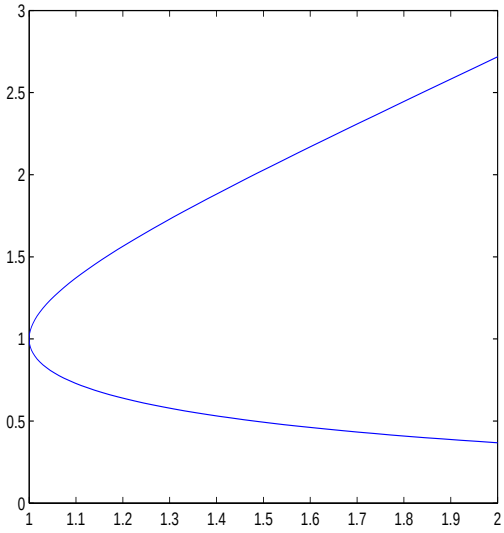

Figure 4: Plot of $x(t) = t^2 + 1$ and $y(t) = e^t$ for $t \in [-1, 1]$.

The color and style of the graphs can be specified in single quotes directly after the pair of values. For example, if we would like the plot of $x(t)$ to be red, and the plot of $y(t)$ to be green and dashed, we would use

```
>>plot(t,x,'r',t,y,'g–')
```

For more information on the various options, type *help plot*.

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)[2]
>>hold on
>>plot (t,y)
>>title('One plot over the other')
>>u=[-1 0 1];
>>v=[1 0 -1]
>>plot(u,v)
```

△

3.4 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.

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

Example 3.6. Plot the three functions $f(x) = x$, $g(x) = x^2$, and $h(x) = x^3$.
The following sequence of commands produces the plot given in Figure 5.

```
>>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);
```

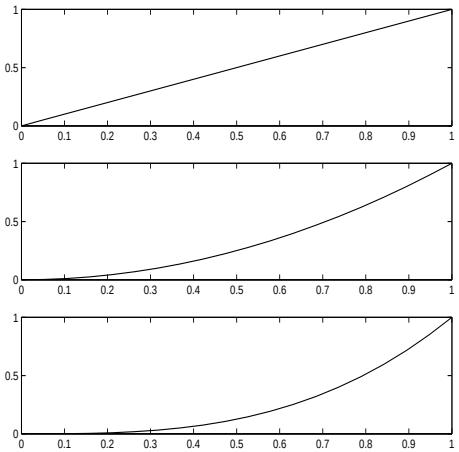

Figure 5: Algebraic functions on parade.

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).

3.5 Ezplot

In most of our plotting for M151, we will use the *plot* command, but another option is the built-in function *ezplot*, which can be used along with symbolic variables.

Example 3.7. Plot the function

$$f(x) = x^4 + 2x^3 - 7x^2.$$

We can use

```
>>syms f x
>>f=x^4+2*x^3-7*x
f =
x^4+2*x^3-7*x
>>ezplot(f)
```

In this case, MATLAB chooses appropriate axes, and we obtain the plot in Figure 6.

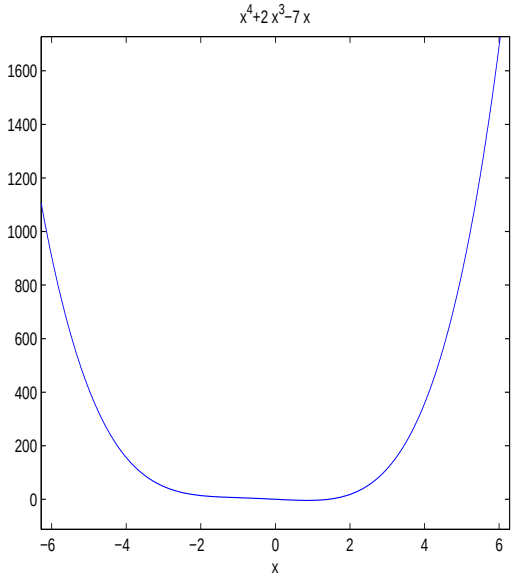

Figure 6: Default plot from *ezplot*.

We can also specify the domain on which to plot with *ezplot(f,xmin,xmax).* For example, *ezplot(f,-1,1)* creates Figure 7.

Alternatively, the variables need not be defined symbolically if they are placed in single quotes. We could also plot this example using respectively

```
>>ezplot('x^4+2*x^3-7*x')
```

or

```
>>ezplot('x^4+2*x^3-7*x',-1,1)
```

△

The ezplot command can also be a good way for plotting implicitly defined relations, by which we mean relations between x and y than cannot be solved for one variable in terms of the other.

Example 3.8. Plot y versus x given the relation

$$\frac{x^2}{9} + \frac{y^2}{4} = 1.$$

This is, of course, the equation of an ellipse, and it can be plotted by separately graphing each of the two solution curves

$$y = \pm 2\sqrt{1 - \frac{x^2}{9}}.$$

Alternatively, we can use the following single command to create Figure 8.

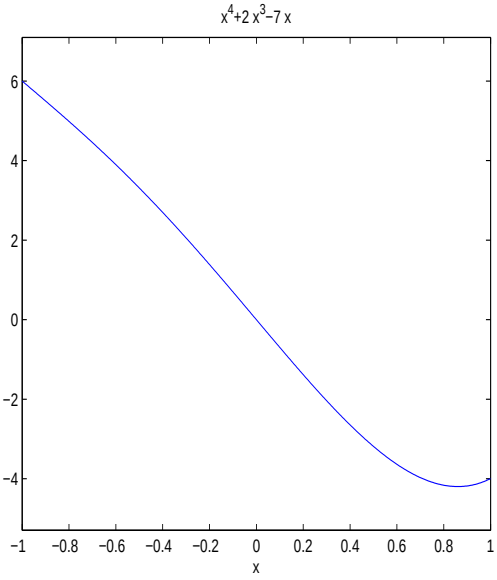

Figure 7: Domain specified plot with *ezplot*.

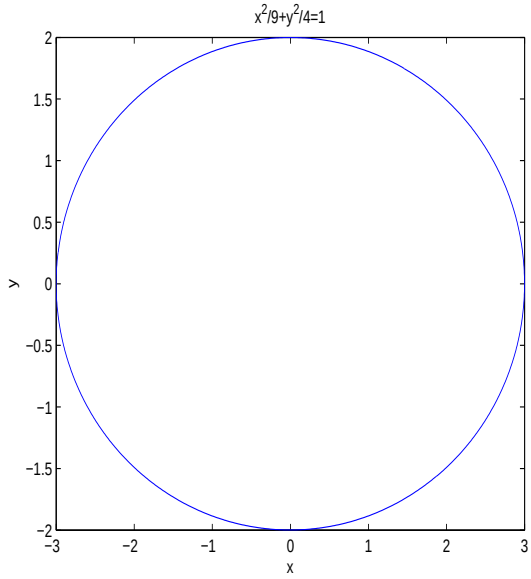

Figure 8: The ellipse described by $\frac{x^2}{9} + \frac{y^2}{4} = 1$.

```
>>ezplot('xˆ2/9+yˆ2/4=1',[-3,3],[-2,2])
```

Here, observe that the first interval specifies the values of x and the second specifies the values for y. △

Finally, we can use *ezplot* to plot parametrically defined relations.

Example 3.9. Use *ezplot* to plot y versus x, given $x(t) = t^2+1$ and $y(t) = e^t$, for $t \in [-1, 1]$.

We can accomplish this with the single command

```
>>ezplot('tˆ2+1','exp(t)',[-1,1])
```

△

3.6 Saving Plots as Encapsulated Postscript Files

In order to print a plot, first save it as an encapsulated postscript file. From the options in your graphics box, choose **File**, **Save As**, and change **Save as type** to **EPS file**. Finally, click on the **Save** button. The plot can now be printed using the *xprint* command.

Once saved as an encapsulated postscript file, the plot cannot be edited, so it should also be saved as a MATLAB figure. This is accomplished by choosing **File**, **Save As**, and saving the plot as a .fig file (which is MATLAB's default).

3.7 More on Basic Plotting

Two more useful built-in functions that have been omitted in this section are the *line* function and the *fplot* function. Interested readers can obtain information about these functions from MATLAB's help menu.

4 Semilog and Double-log Plots

In many applications, the values of data points can range significantly, and it can become convenient to work with $\log_{10}$ values of the original data. In such cases, we often work with *semilog* or *double-log* (or *log-log*) plots.

4.1 Semilog Plots

Consider the following data (real and estimated) for world populations in certain years.

Year	Population
-4000	7×10^6
-2000	2.7×10^7
1	1.7×10^8
2000	6.1×10^9

We can plot these values in MATLAB with the following commands, which produce Figure 9.

```
>>years=[-4000 -2000 1 2000];
>>pops=[7e+6 2.7e+7 1.7e+8 6.1e+9];
>>plot(years,pops,'o')
```

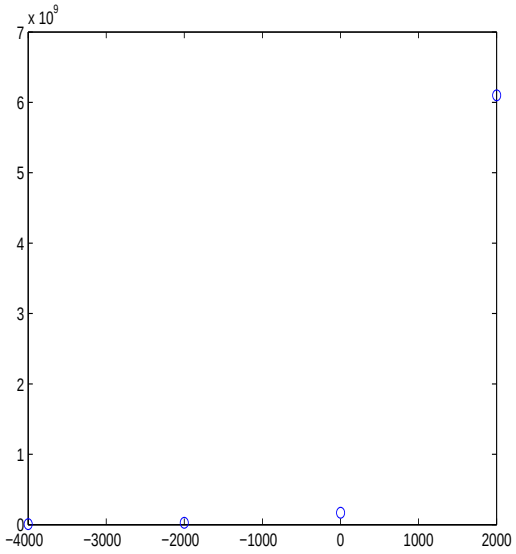

Figure 9: Standard plot for populations versus year.

Looking at Figure 9, we immediately see a problem: the final data point is so large that the remaining points are effectively zero on the scale of our graph. In order to overcome this

problem, we can take a base 10 logarithm of each of the population values. That is,

$$\begin{aligned}
\log_{10} 7 \times 10^6 &= \log_{10} 7 + 6 \\
\log_{10} 2.7 \times 10^7 &= \log_{10} 2.7 + 7 \\
\log_{10} 1.7 \times 10^8 &= \log_{10} 1.7 + 8 \\
\log_{10} 6.1 \times 10^9 &= \log_{10} 6.1 + 9.
\end{aligned}$$

We can plot these new values with the following commands.

```
>>logpops=log10(pops);
>>plot(years,logpops,'o')
```

In this case, we obtain Figure 10.

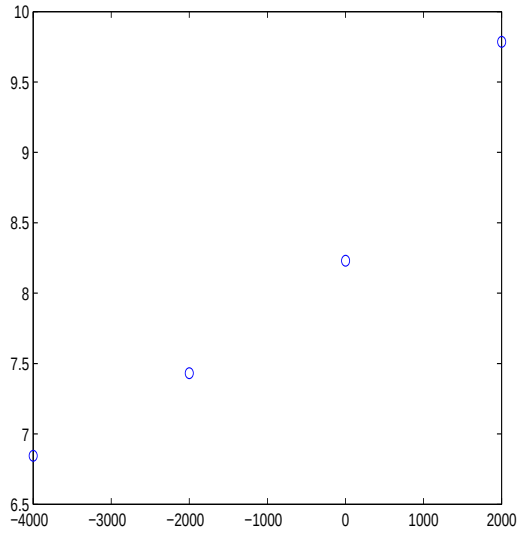

Figure 10: Plot of the log of populations versus years.

We can improve this slightly with MATLAB's built-in function *semilogy*. This function carries out the same calculation we just did, but MATLAB adds appropriate marks on the vertical axis to make the scale easier to read. We use

```
>>semilogy(years,pops,'o')
```

The result is shown in Figure 11. Observe that there are precisely eight marks in Figure 11 between 10^7 and 10^8. The first of these marks 2×10^7, the second 3×10^7 etc. up to the eighth, which is 9×10^7. At that point, we have reached the mark for 10^8.

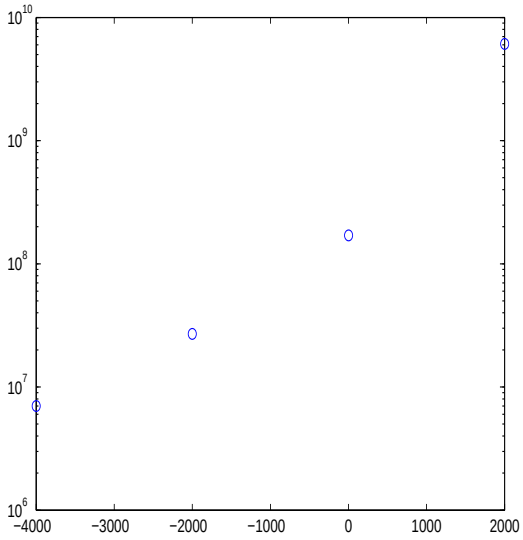

Figure 11: Semilog plot of world population data.

4.1.1 Deriving Functional Relations from a Semilog Plot

Having plotted our population data, suppose we would like to find a relationship of the form

$$N = f(x),$$

where N denotes the number of individuals in the population during year x. We proceed by observing that the four points in Figure 11 all lie fairly close to the same straight line. In Section **??**, we will discuss how calculus can be used to find the exact form for such a line, but for now we simply allow MATLAB to carry out the computation. From the graphics window for Figure 10 (the figure created prior to the use of *semilogy*), choose **Tools**, **Basic Fitting**. From the **Basic Fitting** menu, choose a **Linear** fit and check the box next to **Show Equations**. This produces Figure 12.

This line suggests that the relationship between N and x is

$$\log_{10} N = .00048x + 8.6.$$

(Recall that we obtained this figure by taking $\log_{10}$ of our data.) Taking each side of this last expression as an exponent for the base 10, we find

$$10^{\log_{10} N} = 10^{.00048x+8.6} = 10^{.00048x}10^{8.6}.$$

We conclude with the functional relation

$$N(x) = 10^{.00048x}10^{8.6},$$

which is the form we were looking for.

Finally, we note that MATLAB's built-in function *semilogx* plots the x-axis on a logarithmic scaling while leaving the y-axis in its original form.

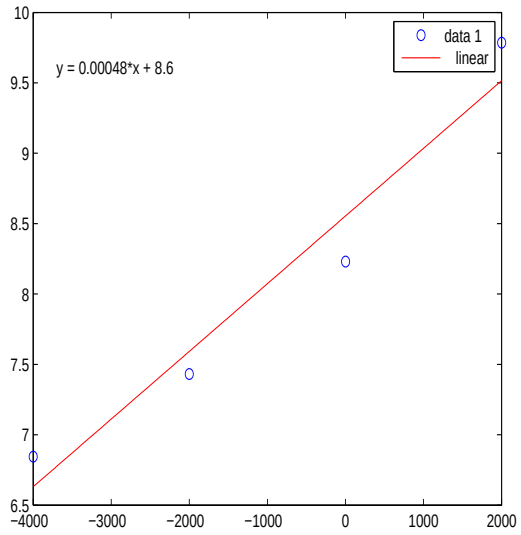

Figure 12: Best line fit for the population data.

4.2 Double-log Plots

In the case that we take the base 10 logarithm of both variables in the problem, we say that the plot is a *double-log* or *log-log* plot.

Example 4.1. In certain cases, the number of plants in an area will decrease as the average size of the individual plants increases. (Since each plant is using more resources, fewer plants can be sustained.) In order to find a quantitative relationship between the number of plants N and the average plant size S, consider the data given in Table 1.

N	S
1	10000
10	316.23
50	28.28
100	10

Table 1: Number of plants N and average plant size S.

In this case, we will find a relationship between N and S of the form

$$S = f(N).$$

We proceed by taking the base 10 logarithm of all the data and creating a double-log plot of the resulting values. The following MATLAB code produces Figure 13.

```
>>N=[1 10 50 100];
>>S=[10000 316.23 28.28 10];
>>loglog(N,S)
```

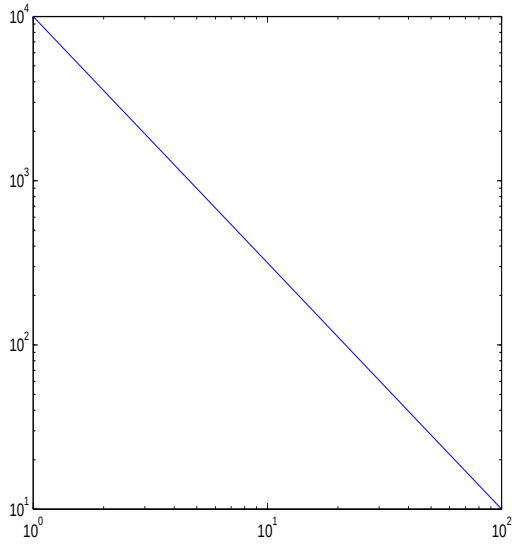

Figure 13: Double-log plot of average plant size S versus number of plants N.

Since the graph of the data is a straight line in this case,[3] we can compute the slope and intercept from standard formulas. In standard slope-intercept form, we can write the equation for our line as

$$\log_{10} S = m \log_{10} N + b.$$

The slope is

$$m = \frac{y_2 - y_1}{x_2 - x_1},$$

where (x_1, y_1) and (x_2, y_2) denote two points on the line, and b is the value of $\log_{10} S$ when $N = 1$ (because $\log_{10} 1 = 0$). In reading the plot, notice that values 10^k should be interpreted simply as k. That is,

$$m = \frac{4 - 1}{0 - 2} = -\frac{3}{2},$$

and

$$b = 4.$$

We conclude

$$\log_{10} S = -\frac{3}{2} \log_{10} N + 4.$$

In order to get a functional relationship of the type we are interested in, we take each side of this last expression as an exponent for the base 10. That is,

$$10^{\log_{10} S} = 10^{-\frac{3}{2} \log_{10} N + 4} = 10^{\log N^{-\frac{3}{2}}} 10^4 \Rightarrow S = 10^4 N^{-\frac{3}{2}}.$$

In practice, the multiplication factor 10^4 varies from situation to situation, but the power law $N^{-\frac{3}{2}}$ is fairly common. We often write

$$S \propto N^{-\frac{3}{2}}.$$

[3]Cooked up, admittedly, though the relationship we'll get in the end is fairly general.

△

5 Inline Functions and M-files

Functions can be defined in MATLAB either in line (that is, at the command prompt) or as M-files (separate text files).

5.1 Inline Functions

Example 5.1. Define the function $f(x) = e^x$ in MATLAB and compute $f(1)$.

We can accomplish this, as follows, with MATLAB's built-in *inline* function.

```
>>f=inline('exp(x)')
>>f(1)
ans =
2.7183
```

Observe, in particular, the difference between $f(1)$ when f is a function and $f(1)$ when f is a vector: if f is a vector, then $f(1)$ is the first component of f, *not* the function f evaluated at 1. △

In a similar manner, we can define a function of several variables.

Example 5.2. Define the function $f(x, y) = x^2 + y^2$ in MATLAB and compute $f(1, 2)$.

In this case, we use

```
>>f=inline('x^2 + y^2','x','y')
f =
    Inline function:
    f(x,y) = x^2 + y^2
>>f(1,2)
ans =
    5
```

Notice that in the case of multiple variables we specify the order in which the variables will appear as arguments of f. Compare the previous code with the following, in which MATLAB expects y as the first input of f and x as the second.[4]

```
f=inline('x^2+y^2','y','x')
f =
Inline function:
f(y,x) = x^2+y^2
```

△

In many cases we would like to define functions that use MATLAB's array operations .^, .*, and ./. This can be accomplished either by typing the array operations in by hand or by using the *vectorize* command.

Example 5.3. Define the function $f(x) = x^2$ in MATLAB in such a way that MATLAB can take vector input and return vector output. Compute $f(x)$ if x is the vector $x = [1, 2]$.

We use

[4] Granted, in this example order doesn't matter.

```
>>f=inline(vectorize('x^2'))
f =
     Inline function:
     f1(x) = x.^2
>>x=[1 2]
x =
     1     2
>>f(x)
ans =
     1     4
```

△

Finally, in some cases it is convenient to define an inline function when the variables are symbolic. Since the *inline* function expects a string, or character, as input, we first convert the symbolic expression into a string expression.

Example 5.4. Compute the inverse of the function

$$f(x) = \frac{1}{x+1}, \quad x > -1,$$

and define the result as a MATLAB inline function. Compute $f^{-1}(5)$.

We use

```
>>finv=solve('1/(x+1)=y')
finv =
-(y-1)/y
>>finv=inline(char(finv))
finv =
Inline function:
finv(y) = -(y-1)/y
>>finv(5)
ans =
-0.8000
```

Observe that the variable *finv* is originally defined symbolically even though the expression MATLAB solves is given as a string. The *char* command converts *finv* into a string, which is appropriate as input for *inline*. △

Inline functions can be plotted with either the *ezplot* command or the *fplot* (function plot) command.

Example 5.5. Define the function $f(x) = x + \sin x$ as an inline function and plot if for $x \in [0, 2\pi]$ using first the *ezplot* command and second the *fplot* command.

The following commands create, respectively, Figure 14 and Figure 15.

```
>>f=inline('x+sin(x)')
f =
Inline function:
```

```
f(x) = x+sin(x)
>>ezplot(f,[0 2*pi])
>>fplot(f,[0 2*pi])
```

△

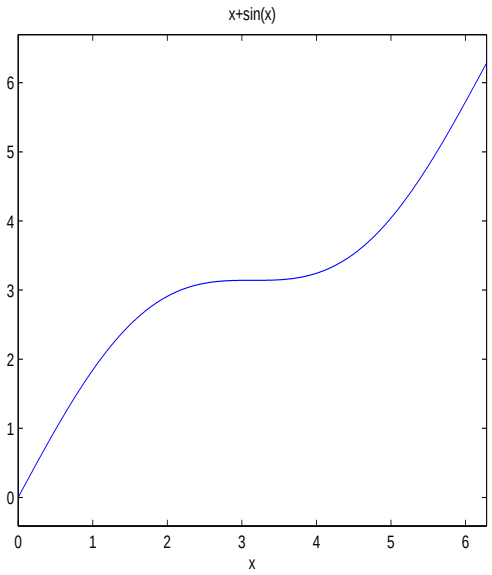

Figure 14: Plot of $f(x) = x + \sin x$ using *ezplot*.

5.2 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.) In this window, type the following lines:

```
x = linspace(0,2*pi,50);
f = sin(x);
plot(x,f)
```

Save this file by choosing **File**, **Save As** from the main menu. In this case, save the file as *sineplot.m*, and then close or minimize your editor window. Back at the command line, type *sineplot* at the prompt, and MATLAB will plot the sine function on the domain $[0, 2\pi]$. It has simply gone through your file line by line and executed each command as it came to it.

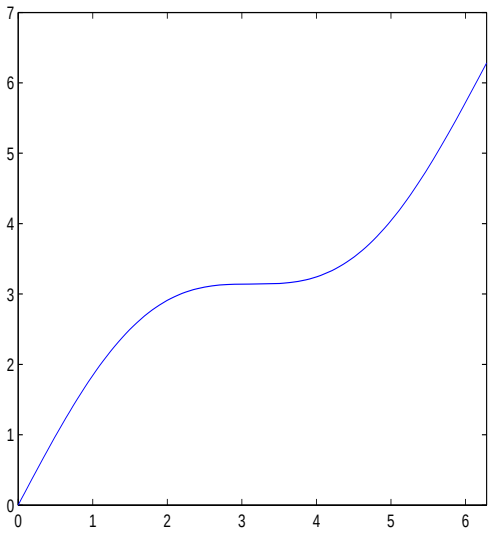

Figure 15: Plot of $f(x) = x + \sin x$ using *fplot*

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 our first example, we will write a function M-file that takes as input the number of points for our sine plot from the previous section and then plots the sine curve. We can begin by typing

```
>>edit sineplot
```

In MATLAB's editor, revise your file sineplot.m so that it has the following form:

```
function sineplot(n)
x = linspace(0,2*pi,n);
f = sin(x);
plot(x,f)
```

Every function M-file begins with the command *function,* and the input is always placed in parentheses after the name of the function M-file. Save this file as before and then run it with 5 points by typing

```
>>sineplot(5)
```

In this case, the plot should be fairly poor, so try it with 50 points (i.e., use *sineplot(50)*).

We can also take several inputs into our function at once. As an example, suppose that we want to take the left and right endpoints of our plotting interval as input (as well as the number of points). We use

```
function sineplot(a,b,n)
x = linspace(a,b,n);
f = sin(x);
plot(x,f)
```

Here, observe that order is important, so when you call the function you will need to put your inputs in the same order as they are read by the M-file. For example, to again plot sine on $[0, 2\pi]$, we use

```
>>sineplot(0,2*pi,50).
```

MATLAB can also take multiple inputs as a vector. Suppose the three values 0, 2π, and 50 are stored in the vector v. That is, in MATLAB you have typed

```
>>v=[0,2*pi,50];
```

In this case, we write a function M-file that takes v as input and appropriately places its components.

```
function sineplot(v)
x = linspace(v(1),v(2),v(3));
f = sin(x);
plot(x,f)
```

5.4 Functions that Return Values

In the function M-files we have considered so far, the files have taken data as input, but they have not returned values. In order to see how MATLAB returns values, suppose we want to compute the maximum value of $\sin(x)$ on the interval over which we are plotting it. Change *sineplot.m* as follows:

```
function maxvalue = sineplot(v)
x = linspace(v(1),v(2),v(3));
f = sin(x);
maxvalue = max(f);
plot(x,f)
```

In this new version, we have made two important changes. First, we have added *maxvalue* = to our first line, specifying that the value we want MATLAB to return is the one we compute as maxvalue. Second, we have added a line to the code that computes the maximum of f and assigns its value to the variable *maxvalue*. (The MATLAB function *max* takes vector input and returns the largest component.) When running an M-file that returns data from the command window, you will typically want to assign the returned value a designation. Here, you might use

```
>>m=sineplot(v)
```

The maximum of $\sin(x)$ on this interval will be recorded as the value of m.

MATLAB can also return multiple values. Suppose we would like to return both the maximum and the minimum of f in this example. We use

```
function [minvalue,maxvalue] = sineplot(v)
x = linspace(v(1),v(2),v(3));
f = sin(x);
minvalue = min(f);
maxvalue = max(f);
plot(x,f)
```

In this case, at the command prompt, type

```
>>[m,n]=sineplot(v)
```

The value of m will now be the minimum of $\sin(x)$ on this interval, while n will be the maximum.

As our last example, we will write a function M-file that takes vector input and returns vector output. In this case, the input will be as before, and we will record the minimum and maximum of f in a vector. We have

```
function w = sineplot(v)
x = linspace(v(1),v(2),v(3));
f = sin(x);
w = [min(f),max(f)];
plot(x,f)
```

This function can be called with

```
>>b=sineplot(v)
```

where it is now understood that b is a vector with two components.

5.5 Subfunctions

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.

5.6 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. If all else fails, **Control-Alt-Backspace** will end your session on a calclab account.

6 Basic Calculus in MATLAB

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

6.1 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
```

6.2 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 $\int_0^{\pi} \int_0^{\sin x} (x^2 + y^2) dy dx$, 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*, *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.

6.3 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
```

6.4 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
```

6.5 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.

6.6 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.

7 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
```

It's also possible to refer to a precise collections of rows and columns of a matrix. For example, suppose we would like to refer to the matrix created by taking only elements that are in the first two rows of A and in the first and third columns. That is, we want

$$B = \begin{pmatrix} 1 & 3 \\ 4 & 6 \end{pmatrix}.$$

In MATLAB, we use

```
>>B=A([1,2],[1,3])
B =
1 3
4 6
```

Here, we have simply used one vector to designate which rows to select and a second vector to designate which columns to select.

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
```

8 Programming in MATLAB

8.1 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.[6] 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.

8.2 Loops

8.2.1 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
```

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

8.2.2 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.

8.3 Branching

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

8.3.1 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).

8.3.2 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
```

8.4 Input and Output

8.4.1 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
```

8.4.2 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.

The *fprintf* function is slightly more complicated, but it gives the user more versatility in creating output. For example,

```
>>fprintf('If we raise %5.2f to the power %i, we get %5.4e \n',pi,10,pi^10)
If we raise 3.14 to the power 10, we get 9.3648e+04
```

In this example, the % designates that a number is to be inserted into the text, and the decimal 5.2 specifies a field width of 5 and a decimal accuracy of 2. The f designates fixed point (i.e., standard decimal) notation. Likewise, %i designates that an integer is to be inserted, and %5.4e designates that a number is to be inserted with field width five (or larger, if necessary, as here), four decimal places of accuracy, and in exponential notation. Finally, the combination \n creates a new line following the print-out.

8.4.3 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
```

8.4.4 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
```

9 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.

10 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.

11 SIMULINK

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

12 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.

13 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*.)

13.1 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.

13.2 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.

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

Index