MATLAB 9.9 Basics

Peter Howard

Spring 2022

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.

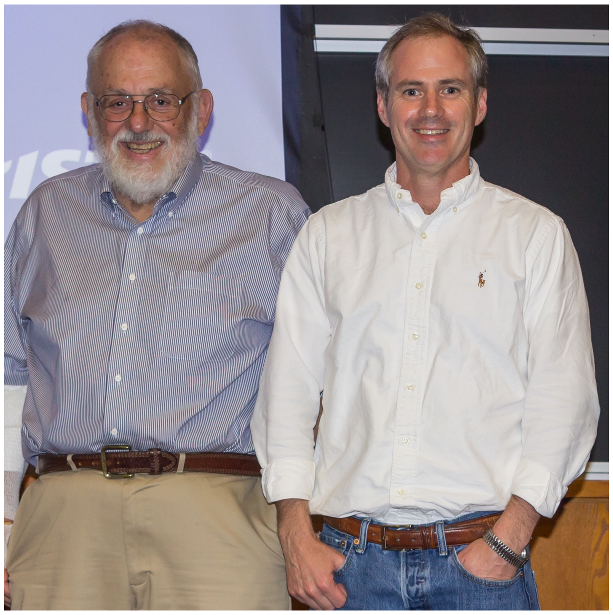

Figure 1: The author of these notes with Cleve Moler in 2016 (Moler on left).

1.2 The Goal of These Notes

These notes are intended to provide readers with a quick introduction to some of MATLAB's basic functionality. They cover symbolic calculations, rudimentary plotting features, anonymous functions, script and function M-files, matrices, and some elements of programming.

1.3 Using MATLAB at Texas A&M University

MATLAB is available on all Texas A&M open access lab accounts, and can be selected from the Microsoft start menu. In addition, students can obtain a free copy of MATLAB through the University at *software.tamu.edu*.

1.4 The MATLAB Interface

The (default) MATLAB screen is divided into four windows, with a large *Command Window* in the middle, two smaller windows on the left (*Current Folder* and *Details*), and one narrow window on the right (*Workspace*). This default layout can be changed with the **Layout** option at the top of the screen. The Command Window is where calculations are carried out in MATLAB, while the smaller windows display information about the current MATLAB session, the previous MATLAB sessions (if the *Command History* option is in use), and the user's local directory. In particular: the Command History displays the commands that have been typed in during either the current or previous sessions; the Current Folder shows which folder the user is currently in and which files are in that folder; the Workspace window displays information about each variable defined in the current session; and the Details option provides additional information about a selected file (the type of file it is, its header description etc.) The user can change MATLAB's working directory by double-clicking on a directory in the Current Directory window. The user can return to the parent directory by clicking on the folder with a black arrow on it in the top left corner of the Current Directory window.

1.5 Basic Computations

At the prompt, designated by two arrows, >>, if we type *2 + 2* and press **Enter** we find that the answer has been assigned to the default variable *ans.* Next, if we type *2+2;* (including the semicolon this time) and press **Enter**, we 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, if we type the keyboard's up arrow key, we see that the command *s=sin(t)* comes back up on the screen. Likewise, if we hit the up arrow key again *t=4;* will appear at the prompt. Using the down arrow, we can scroll back the other way, giving us 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, we will find that an expression we are typing is getting inconveniently long and needs to be continued to the next line. We can accomplish this by putting in three dots and typing **Enter**. For example, we can type the following:[1]

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

Notice that *2+3+4+...* was typed at the Command Window prompt, followed by **Enter**. MATLAB then proceeded to the next line, but without producing a new prompt. The remainder of the sum *+5+6* was then typed in.

[1]For the MATLAB examples of these notes, typed commands will always be indicated by the command line prompt, >>, so that they can be distinguished from MATLAB's output.

1.6 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, we can type *format long* at the beginning of a session. All subsequent numerical output will show the greater precision. We can then type *format short* to return to shorter display. There are numerous data types in MATLAB, including *floating point* (which we've just been discussing), *symbolic* (see Section 2), and *character strings*. A list of all active variables—along with size and type—is given in the *Workspace.*

1.7 Diary Files

For many of the assignments this semester, and also for the projects, students will need to turn in a log of MATLAB commands typed, along with 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.*

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.* △

1.8 Live Scripts

As an alternative to diary files, *Live Scripts* can be used to create files containing both user input and MATLAB output. The Live Scripts option can be selected at the top left of the screen, and it will generate a workspace in which a series of commands can be typed, in this case without prompts. Once the commands have been typed in, the Live Script can be run by selecting the green arrow at the top of the page, and it will then print both the input and the output. This completed script can then be exported via the **Save** option as a PDF, HTML, or LaTeX document.

1.9 Clearing and Saving the Command Window

The Command Window can be cleared with the command *clc*, which leaves all variable definitions in place. The variable definitions can be cleared with the command (you guessed it) *clear*. All variables in a MATLAB session can be saved to a .mat file with the menu option **Save Workspace**. Later, this file can be opened via the **Open** command. A word of warning, though: This does not save every command that has been typed into the; it only saves the variable assignments. For bringing all commands from a session back, see the discussion under *Command History.*

1.10 The Command History

The Command History window displays a list of recent commands issued at the prompt. Often, it's convenient 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. Alternatively, if the Command History isn't docked, it will come up as a pop-up menu when the up-arrow option is used, and commands can be selected from the list.

1.11 File Management from MATLAB

There are certain commands in MATLAB that will manipulate files on its primary directory. For example, if we have the file *junk.m* in our working MATLAB directory, we can delete it simply by typing *delete junk.m* at the MATLAB command prompt. Much more generally, if we precede a command with an exclamation point, MATLAB will read it as a unix shell command. So, for example, the three commands *!ls*, *!cp junk.m morejunk.m*, and *!ls* serve to list the contents of the directory we 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.

1.12 Getting Help

As with any other software package, the most important MATLAB commands are those that can be used to get additional help. In MATLAB, the most useful of these are *help*, *doc*, and *helpdesk.* For help on a particular topic such as the integration command *int*, we can type >>*help int* or >>*doc int*, and in either case we will obtain information about the command, though *doc* gives more information. If the screen's output flies by too quickly, it can be stopped 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, we type >>*helpdesk.* Next, in the top right of the helpdesk screen, we type *inverse* in and select the magnifying glass to carry out the search. A pop-up menu comes up, and we can select the function *asin()* as the inverse for *sine.* We then 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: We saw above that $\sin(4) = -.7568$, so if *asin()* is the inverse of *sin()*, why isn't the answer 4?[2])

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)
```

△

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:

[2]The sine function must be inverted on an appropriate interval of length π, and MATLAB chooses the interval $[-\frac{\pi}{2}, \frac{\pi}{2}]$. The values $\sin(-.8584)$ and $\sin(4)$ are both $-.7568$, but MATLAB only returns the value on the interval $[-\frac{\pi}{2}, \frac{\pi}{2}]$.

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

2.1.1 Complex Numbers

We 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.)

2.2 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
```

2.3 Solving Algebraic Equations

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

Example 2.8. 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 =
-(b + (b^2 - 4*a*c)^(1/2))/(2*a)
-(b - (b^2 - 4*a*c)^(1/2))/(2*a)
```

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 command:

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

Here, the *syms* command has been used again because a has been redefined in the code above. △

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

Example 2.9. 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 =
d*x*y - c*y
>>[prey pred]=solve(Rx,Ry)
prey =
0
c/d
pred =
0
a/b
```

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 =
struct with fields:
x: [2×1 sym]
y: [2×1 sym]
>>pops.x
ans =
0
c/d
>>pops.y
ans =
0
a/b
```

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* command

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

Example 2.10. 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);
>>double(r)
ans =
0.4534 + 0.0000i
-0.2267 - 1.4677i
-0.2267 + 1.4677i
```

△

2.4.2 The Subs Command

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

Example 2.11. 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 =
-N*a*(N/K - 1)
>>R=subs(R,a,.1)
R =
-(N*(N/K - 1))/10
>>R=subs(R,K,10)
R =
-(N*(N/10 - 1))/10
```

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

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.

2.5 Basic Calculus

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

2.5.1 Differentiation

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

```
>>syms x;
>>diff(x^3)
ans =
3*x^2
```

Alternatively, we can first define x^3 as an anonymous function:

```
>>syms x;
>>f = @(x) x^3;
>>diff(f(x))
ans =
3*x^2
```

Remark. *Anonymous functions* are discussed more systematicallly below in Section 4.

Higher order derivatives can be computed simply by putting the order of differentiation after the function, separated by a comma. Continuing with the anonymous function $f(x)$ from just above, we can use:

```
>>diff(f(x),2)
ans =
6*x
```

Finally, MATLAB can compute partial derivatives. For example, suppose we would like to compute $\frac{\partial g}{\partial x}$ and $\frac{\partial g}{\partial y}$ for the function of two variables

$$g(x, y) = x^2 y^4.$$

We can use the following code:

```
>>syms x y;
>>g = @(x,y) x^2*y^4;
>>gx=diff(g(x,y),y)
gx=
2*x*y^4
>>gy = diff(g(x,y),y)
gy =
4*x^2*y^3
```

Example 2.12. In introductory calculus courses, we often identify critical points of a function $f(x)$ by finding the values x_* at which $f'(x_*) = 0$. In this example, we will use MATLAB to identify the critical points of

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

on $\mathbb{R}$. In addition, we will find the values $x_{**} \in \mathbb{R}$ at which $f''(x_{**}) = 0$. For both steps, we can use the following code:

```
>>syms x
>>fp=diff(x/(x^2+1))
fp =
1/(x^2 + 1) - (2*x^2)/(x^2 + 1)^2
>>solve(fp)
ans =
-1
1
>>fpp=diff(fp)
fpp =
(8*x^3)/(x^2 + 1)^3 - (6*x)/(x^2 + 1)^2
>>solve(fpp)
ans =
0
3^(1/2)
-3^(1/2)
```

We see that the critical points of f are ± 1, and the values x_{**} at which $f''(x_{**}) = 0$ are 0, $\pm\sqrt{3}$.

2.5.2 Integration

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

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

or

```
>>syms x;
>>f = @(x) x^2;
>>int(f(x))
ans =
x^3/3
```

The integration with limits $\int_0^1 x^2 dx$ can easily be computed similary. Continuing with the anonymous function $f(x)$ is defined just above, we can use:

```
>>syms x;
>>f = @(x) x^2;
>>int(f(x),0,1)
ans =
1/3
```

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

```
>>syms x y;
>>g = @(x,y) x^2+y^2;
>>int(int(g(x,y),y,0,sin(x)),0,pi)
ans =
pi^2-32/9
```

Numerical integration is accomplished through the commands *quad*, *quadv*, and *quadl* (*quad* abbreviated *quadrature*). Suppose we would like to integrate the function $h(x) = e^{-x^4}$ numericallly from -1 to $+1$. We can use the following code:

```
>>h=@(x) exp(-x.^4);
>>quad(h,-1,1)
ans =
   1.6897
```

In this case, notice that we did not need to specify x as symbolic, and also that x does not explicitly appear in the call to *quad.m*. In addition, notice that the anonymous function $h(x) = e^{-x^4}$ was specified with an array operation. This is necessary if the function is going to be integrated with one of the quad functions.

Finally, suppose we would like to carry out a double integral such as the previous one in this subsection numerically. We can accomplish this with *quad2d.m*, using the following code.

```
>>g = @(x,y) x.^2+y.^2;
>>yupper = @(x) sin(x);
>>quad2d(g,0,pi,0,yupper)
ans =
6.3140
```

Alternatively to *quad.m* and *quad2.m*, single, double, and triple integrals can be computed numerically respectively with *integral1.m*, *integral2.m*, and *integral3.m*. Integration of functions of more than three variables requires some form of nesting of these commands, similarly as with the *int.m* nesting above.

2.5.3 Limits

MATLAB can also compute limits, such as

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

We can use:

```
>>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 can use

```
>>syms x;
>>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 use

```
>>syms x;
>>limit((x^4 + x^2 - 3)/(3*x^4 - log(x)),x,Inf)
ans =
1/3
```

2.5.4 Sums and Products

We often want to compute 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 (with X as defined just above)

```
>>prod(X)
ans =
      5040
```

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

$$\sum_{k=1}^{n} k = \frac{n(n+1)}{2}.$$

We can use

```
>>syms k n;
>>symsum(k,1,n)
ans =
(n*(n + 1))/2
```

Likewise, we can evaluate the infinite sum

$$\sum_{k=1}^{\infty} \frac{1}{k^2} = \frac{\pi^2}{6}$$

with

```
>>symsum(1/kˆ2,1,Inf)
ans =
piˆ2/6
```

2.5.5 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 $\sin x$ about $x = 0$ up to tenth order can be obtained through the commands

```
>>syms x;
>>taylor(sin(x),x,0,'Order',10)
ans =
xˆ9/362880 - xˆ7/5040 + xˆ5/120 - xˆ3/6 + x
```

Likewise, for a function of two variables such as

$$f(x) = e^x \sin(x) \cos(y),$$

we can compute the Taylor expansion about a point $(x_0, y_0) = (1, 2)$ to second order with the following code.

```
>>syms x y;
>>taylor(exp(x)*sin(x)*cos(y),[x y],[1,2],'Order',2)
ans =
cos(2)*(cos(1)*exp(1) + exp(1)*sin(1))*(x - 1) + cos(2)*exp(1)*sin(1) - exp(1)*sin(1)*sin(2)*(y
- 2)
```

2.5.6 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, we can 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, we choose **Tools, Zoom In**. Next, we 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.

Remark. The *plot.m* function M-file is discussed in more detail in the next section.

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.m*, *fminunc.m*, and *fminsearch.m*. We will use *fminsearch.m* quite a bit in the M442 course notes *Modeling Basics*, so we won't discuss it here.

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 $\{(1, 4), (2, 5), (3, 7)\}$.

We first define the x values (1 for the first point, 2 for the second, and 3 for the third) as a single variable $x = (1, 2, 3)$ (typically referred to as a *vector*) and the y values as the vector $y = (4, 5, 7)$, and then we plot these points, connecting them with a line. The following commands (accompanied by MATLAB's output) suffice:

```
>>x=[1 2 3]
x =
1 2 3
>>y=[4 5 7]
y =
4 5 7
>>plot(x,y)
```

The output we obtain is the plot given as Figure 2.

In MATLAB it's particularly easy to decorate a plot. For example, if we minimize the plot in Figure 2, we can add labeling with the following lines:

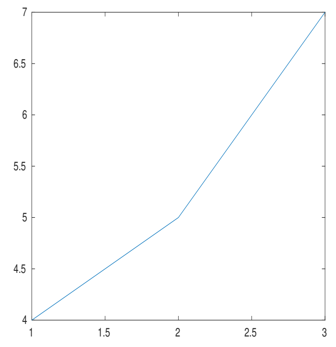

Figure 2: First plot for Example 3.1.

```
>>xlabel('Here is a label for the x-axis')
>>ylabel('Here is a label for the y-axis')
>>title('Simple 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 3.

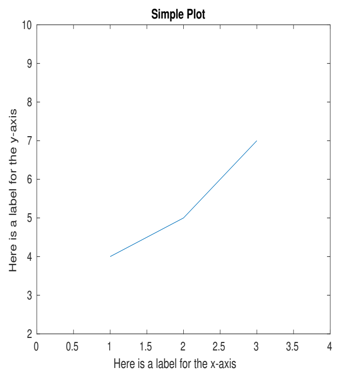

Figure 3: Second plot for Example 3.1.

As an alternative to our use of commands to add labeling to this plot, we could use options in the figure window. For example, under the **Insert** option, the first four choices are X Label, Y Label, Z Label, and Title (Z Label is not available unless the plot has three dimensions).

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 to notice here 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 we 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 4.

```
>>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. △

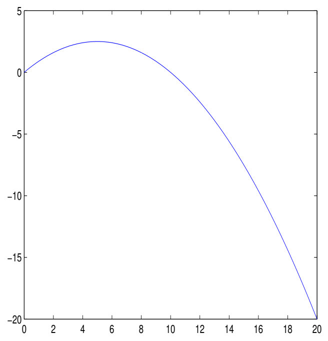

Figure 4: Growth rate for the logistic model.

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 5.

```
>>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);
```

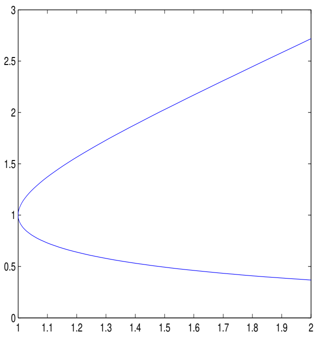

Figure 5: Plot of $x(t) = t^2 + 1$ and $y(t) = e^t$ for $t \in [-1, 1]$.

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--')
```

This creates the figure depicted in Figure 6. For more information on the various options, type *help plot*.

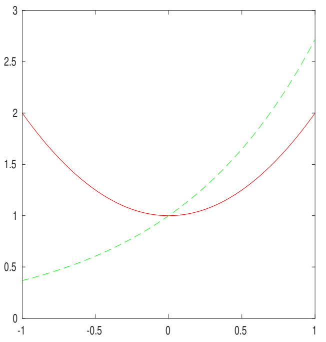

Figure 6: Figure for Example 3.5.

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)[3]
>>hold on
>>plot (t,y)
```

△

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

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.

Example 3.6. Plot the three functions $f(x) = x$, $g(x) = x^2$, and $h(x) = x^3$ in a stack of three different plots.

The following sequence of commands produces the plot given in Figure 7.

```
>>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);
```

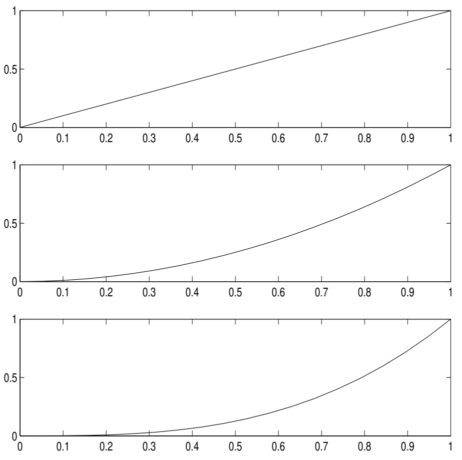

Figure 7: Figure for Example 3.6.

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 Saving Plots as Encapsulated Postscript Files

In order to print a plot or incorporate it into a document, the plot first needs to be saved in a convenient format such as an encapsulated postscript (.eps) file. In order to do this, we can select **File**, **Save As** from the figure window and change **Files of type** to **EPS file**. We then name the file and click on the **Save** button. 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).

3.6 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.

3.6.1 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 8.

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

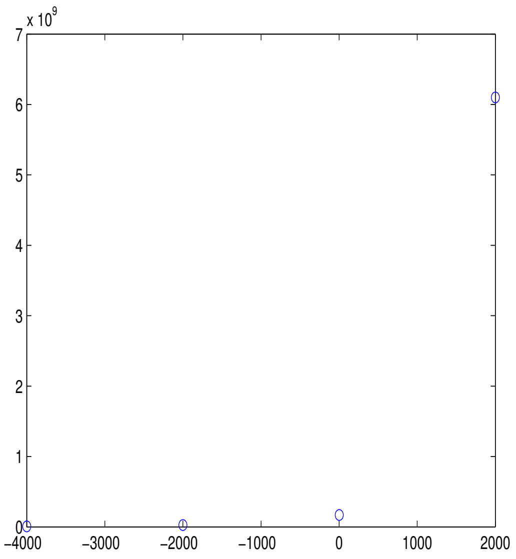

Figure 8: Standard plot for populations versus year.

Looking at Figure 8, 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 9.

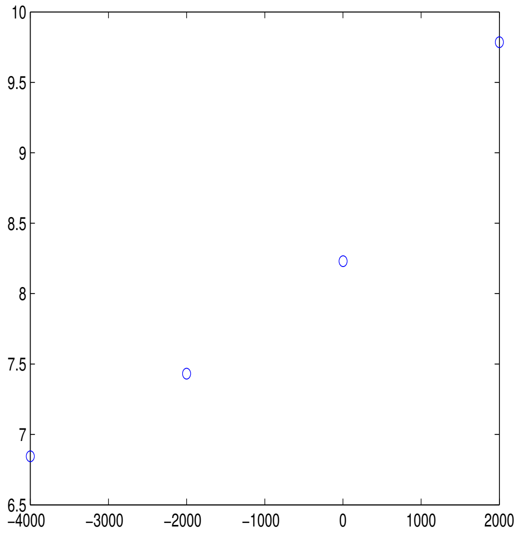

Figure 9: Plot of the log of populations versus years.

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 10. Observe that there are precisely eight marks in Figure 10 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.

3.6.2 Deriving Functional Relations from a Semilog Plot

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

$$N = f(x),$$

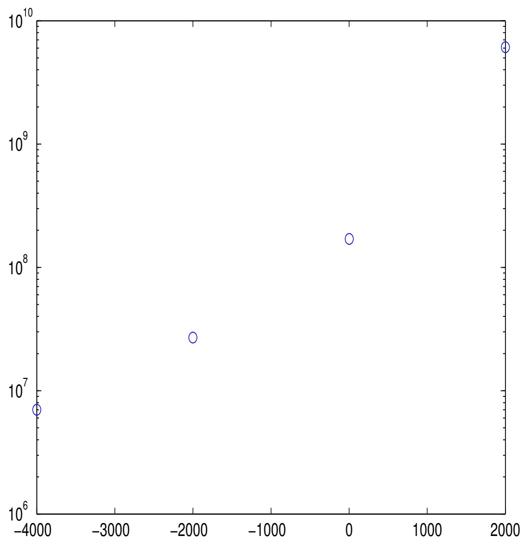

Figure 10: Semilog plot of world population data.

where N denotes the number of individuals in the population during year x. We proceed by observing that the four points in Figure 10 all lie fairly close to the same straight line. In the lecture notes *Modeling Basics*, 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 9 (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 11.

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.

3.7 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.

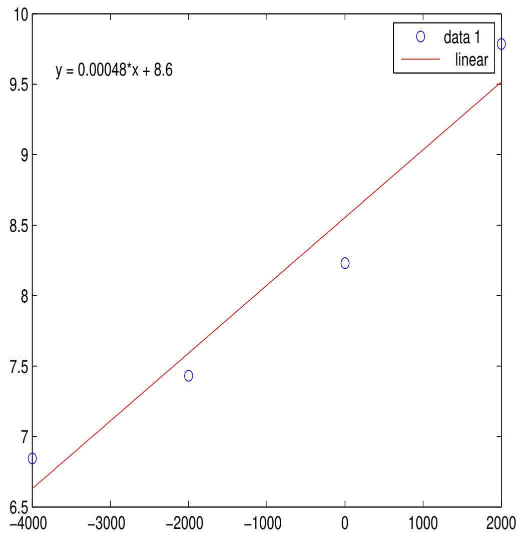

Figure 11: Best line fit for the population data.

Example 3.7. 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 12.

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

Since the graph of the data is a straight line in this case,[4] we can compute the slope and intercept from standard formulas. In standard slope-intercept form, we can write the

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

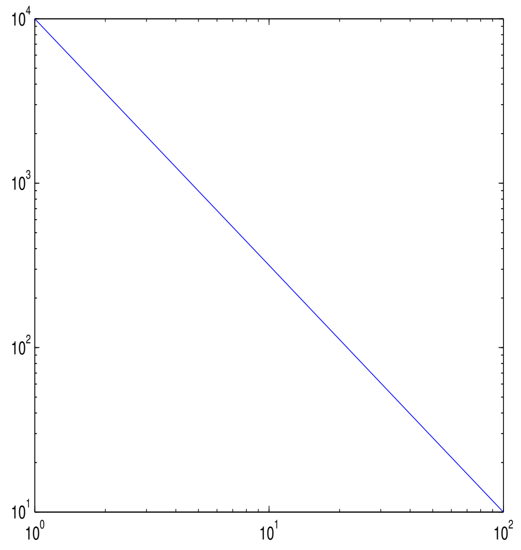

Figure 12: Double-log plot of average plant size S versus number of plants N.

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

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

△

4 Anonymous 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).

4.1 Anonymous Functions

Example 4.1. Define the function $f(x) = xe^x$ in MATLAB and compute $f(1)$.

We can accomplish this, as follows, with an *anonymous* function.

```
>>f = @(x) x*exp(x);
>>f(1)
ans =
2.7183
```

Observe, in particular, the difference between $f(1)$ when f is an anonymous 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 4.2. Define the function $f(x, y) = x^2 + y^2$ in MATLAB and compute $f(1, 2)$.

In this case, we use

```
>>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. △

In many cases we would like to define functions that use MATLAB's array operations .ˆ, .*, and ./.

Example 4.3. 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 $x = [1, 2, 3]$.

We use

```
>>f=@(x) x.^2;
>>f([1 2 3])
ans =
1 4 9
```

△

4.2 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 **New** at the upper left corner of your MATLAB window, then select **Script**. A window will appear, docked or undocked, depending on your preferences, with MATLAB's default editor. (M-file can be created and edited with any text editor, but MATLAB's built-in editor is typically the most convenient one to use.) As a first example, let's enter the following lines:

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

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

It's often useful to add comment lines to the top of an M-file to describe what it does. For example, this information becomes available via the help command, just as with MATLAB's built-in M-files. Since we've named the M-file above sineplot.m, we might add comments lines so that it takes the following form:

```
%SINEPLOT: MATLAB script M-file that plots
%sin(x) on [0, 2*pi].
x = linspace(0,2*pi,50);
f = sin(x);
plot(x,f)
```

The percent signs indicate the comments, and MATLAB will not try to evaluate those lines as code.

4.3 Function M-files

The second type of M-file is called a *function* M-file and typically (though not inevitably) this type of M-file will involve one or more variables being taken as input and one or more variables being returned. As a 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, we can now revise sineplot.m so that it has the following form:

```
%SINEPLOT: MATLAB function M-file that plots
%sin(x) on [0, 2*pi].
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. We can save this file as before and then run it with 5 points by typing

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

In this case, the plot is fairly poor, but we can improve it by adding more points. For example, *sineplot(50)* looks fairly smooth. (These plots have been omitted from the notes, because I think it's clear what they look like.)

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 (leaving off the comment lines)

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

Here, we observe that order is important, so when this function is called the inputs must be 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 we enter

```
>>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)
```

4.4 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. We can accomplish this by revising *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, we typically assign the returned value a designation. Here, we 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 can 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, we 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.

Alternatively to the previous example, we can write *sineplot.m* in such a way that the output is specified as a vector. 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.

As a final note, we mention that it's possible to write a function M-file that neither takes input nor returns output. At first glance, this might sound a lot like a script M-file, but there are (at least) two good reasons for doing it. First, as opposed to script M-files, function M-files use a memory location different from the Workspace, and variables defined by the function M-file won't conflict with active variables in the Workspace. And, second, function M-files allow the incorporation of subfunctions, which can be a useful way of organizing code.

4.5 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.

4.6 Debugging M-files

Once an M-file becomes sufficiently complicated, it can be useful to step through it line-by-line in search of possible errors. As an example, let's return to the final version of *sineplot.m* (from the end of Section 4.4). With this file active in the MATLAB editor, we can set the cursor at the beginning of the line *x = linspace(v(1),v(2),v(3));* and from the menu at the top of the screen choose **Breakpoints**, **Set/Clear**. This will set a break point at the indicated line, marked with a red dot at the left. For a script M-file, we can simply run the file at this point (with the big green button), but for a function M-file we run it as usual in the Command Window by typing, for example,

```
>>sineplot([0,2*pi,50]);
```

The function will run until it arrives at the indicated line, without evaluating the indicated line. In this case, all that happens is that the vector $v = [0, 2\pi, 50]$ is specified, and in the Command Window, the cursor changes to K>>. In order to run the next step, we can either choose Step from the menu at the top of the screen or press F10. In either case, the line *x = linspace(v(1),v(2),v(3));* is now evaluated, x is identified in the Workspace, and MATLAB's editor proceeds to the next line. Continuing in this way, we can step through the file, and see precisely what happens at each step.

In the process above, the line *x = linspace(v(1),v(2),v(3));* was viewed as a single line of code, and evaluated with a single step. But, of course, this isn't really a single line of code, it's really however many lines of code were required to write the built-in M-file *linspace.m*. If the program is running, and the next step is this line, the user can choose **Step In** from the top of the screen (or press F11) to actually proceed step-by-step into the M-file *linspace.m*. Then, in the usual way, the user can step through as this M-file is evaluated. The option **Step Out** can then be used at any time to return to the flow of the program, just past the line in which the M-file under further scrutiny was called.

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 we used for plotting in Section 3). A column vector can be defined similarly by

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

5.1 Matrix Operations

The matrix operations in MATLAB are quite intuitive. For example:

```
>>A=[1 2 3; 4 5 6; 7 8 9]
A =
    1    2    3
    4    5    6
    7    8    9
>>det(A)
ans =
    -9.5162e-16
>>B=[1 2 2; 1 1 2; 0 3 3]
B =
    1    2    2
    1    1    2
    0    3    3
>>det(B)
ans =
   -3
>>Bˆ(-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 $A. * B$.

5.2 Elements of Matrices

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
```

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.[5] And if you are familiar with any programming language—Python, Java, Fortran, Pascal, Basic, even antiques like Cobol—you shouldn't have much trouble catching on. In this section, we will run through the basic commands needed in order to get started programming in MATLAB.

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
```

The output for this loop is given below.

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

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

Notice that we've dropped off the command prompt arrows, because typically this kind of structure is typed into an M-file, not in the Command Window. It's worth noting, however, that a for-loop can be typed directly into the command line. What happens is that after we type *for n=2:5*, MATLAB drops the prompt and lets us close the loop with *end* before responding. By the way, if we type this series of commands in MATLAB's default editor, it will space things for us to separate them and make the code easier to read. One final thing to notice about the *for* command is that if we want to increment our index by something other than 1, we 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 we've included a *break* command in the loop, so that if we'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*. Finally, we 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 the 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.

```
x = input('Enter a number ');
switch x
case 7
y = 1;
case 10
y = 2;
case 17
y = 3;
otherwise
disp('Invalid choice')
end
disp(['The result is y = ',num2str(y)])
```

6.4 Input and Output

6.4.1 Parsing Input and Output

Often, we 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 (line 4)
Need at least two inputs for adding
>>addthree(1,2)
ans =
    3
>>addthree(1,2,3)
ans =
    6
```

We emphasize that the error statement is the one we supplied.

We 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
```

6.4.2 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.

6.4.3 Screen Input

Often, we will want the user to enter some type of data into our 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 *dbcont*. As an example, consider the M-file

```
%INPUTEG1: A script file with examples of
%pause, keyboard, and input
disp('Hit any key to continue...')
pause
disp('Enter a command.  (Type "dbcont" and hit Enter 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,

```
>>inputeg1
Hit any key to continue...
Enter a command.  (Type "dbcont" and hit Enter to return to the script file)
>>3+4
```

```
ans =
    7
>>return
Are you tired of this yet (yes/no)?yes
```

6.4.4 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 inputeg2
%INPUTEG2: 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
```

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