An Introduction to MATLAB

1 MATLAB Introduction and Basics

MATLAB is an interpreted programming language that includes a command-line interpreter (like the various UNIX shells and many implementations of LISP). This has many advantages in terms of ease of use, speed of algorithm prototyping, and so on. On the downside, this means that a piece of code in MATLAB is usually slower than an equivalent in a compiled programming language like C++. To start with, we'll just be using the command-line interpreter so that we can become familiar with some of MATLAB's commands and conventions. The exact same commands, though, will be used later to write *functions* and *scripts*.

The first step to using MATLAB is to bring up the program on your computer system. At a UNIX system, simply typing "matlab" should be sufficient. Starting MATLAB on a Windows machine or a Macintosh generally involves finding the appropriate icon either on the desktop or in the Start menu. Note that you can run MATLAB remotely on UNIX servers through telnet or ssh, but you generally won't be able to use its graphical capabilities. When MATLAB is finished loading, you'll find yourself with a command prompt that looks something like this:

```
>>
```

So now what do you do? Well, the first step is to make use of MATLAB's single most useful command:

```
>> help
```

See that list of categories? You can call help on any of these categories to get an organized list of commands with brief discussions. Then, you can call `help` on any of the commands for a complete description of that command. Select a category that looks interesting and call `help` on it. Do the same for whichever command strikes your fancy. For instance:

```
>> help elmat
>> help why
```

Most often you'll use `help` in this last capacity. Unfortunately it isn't so helpful if you don't know the name of the command you're looking for. One way around this is to use the `lookfor` command. For instance, if you know you're looking for a function that deals with time, you can try:

```
>> lookfor time
```

This searches the first line of the every help description for the word "time." This can take a while, though (depending upon your system's configuration). You should get into the habit of reading the `help` on every new command that you run across. So call `help` on both `help` and `lookfor`. There's some useful information there.

Not surprisingly, you can use MATLAB to do arithmetic. It operates very much like you might expect, employing infix arithmetic like that used on standard calculators. MATLAB can evaluate simple expressions or arbitrarily complicated ones with parentheses used to enforce a particular order of operations.

```
>> 6 * 7
>> (((12 + 5) * 62/22.83) - 5)^2.4
```

(The ^ operator performs exponentiation). Notice that when you execute these commands, MATLAB indicates that `ans = 7.4977e+003` (or whatever). This indicates that the result has been stored in a variable called `ans`. We can then refer to this quantity like this:

```
>> 0 * ans
>> ans + 1
```

It is important to note that the each of these commands overwrites `ans`. If we want to save an answer, we can simply perform assignment like this:

```
>> my_variable = 42
```

This is the only declaration of `my_variable` that is needed, and we can use this variable later just as we could with `ans`. Further, `my_variable` will retain its value until we explicitly assign something else to it. We can `clear` variables, and typing `who` or `whos` will list what variables we have in our workspace.

We also have access to a wealth of standard mathematical functions. Thus, we can if we want to calculate the sine of the square-root of two and store it in a variable called `var`, we simply type:

```
>> var = sin(sqrt(2))
```

Type `help elmat` to see how to call most of the elementary mathematical functions like these. There are also a number of constants built into MATLAB that are very useful. The number π is referred to as `pi` (note that MATLAB is case sensitive!). Both `i` and `j` default to $\sqrt{-1}$, but you can still use either (or both) as variable names if you like. You should glance at `help i` so that you can see the various options for building complex numbers.

2 Vectors, Matrices, and Arrays

So far, we've been using MATLAB to deal with *scalar* numbers. The real power of MATLAB, though, comes from its ability to handle *vectors* and *matrices*. In MATLAB, vectors and matrices are simply one-dimensional and two-dimensional *arrays*, respectively. An array is simply a collection of numbers, each of which is *indexed* by an n-tuple. For an array, the dimension is equal to the length of the n-tuple. For instance, consider the following vector and matrix:

$$\mathbf{v} = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} \qquad M = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \tag{1}$$

To access the 3 from vector $\mathbf{v}$, we simply need to know that it is in the third row. Thus the vector is one-dimensional. To access the 6 in the matrix M, though, we need to know that it is in the second row and the third column. We index the 6 using the pair (2,3), and so matrix is two-dimensional. MATLAB arrays can have any number of dimensions. In practice, though, we will only need vectors and matrices.

There are many different ways to build up and manipulate arrays in MATLAB. For instance, consider (and execute) the following commands:

```
>> a = [1 2 3 4 5 6 7]
>> b = [1, 2, 3, 4, 5, 6, 7]
>> c = [1; 2; 3; 4; 5; 6; 7]
>> d = [1 2 3 4; 5 6 7 8; 9 10 11 12]
```

The first two commands both build up the same vector, a 1x7 *row-vector*. The third command builds up a 7x1 *column-vector* with the same elements. The fourth command builds a 3x4 matrix. Notice that the comma (or the space) within the square brackets concatenates horizontally and the semicolon concatenates vertically. The elements being concatenated do not need to be scalars, either:

```
>> e = [a b]
>> f = [a; b]
>> g = [c d]
```

Oops! That last command produced an error. When concatenating arrays, the concatenated arrays must have sizes such that the resulting array is rectangular.

The single apostrophe, `'`, is MATLAB's transposition operator. It will turn a row-vector into a column-vector and vice versa. Similarly, it will make an nxm matrix into an mxn matrix. To see how this works, type `d'` and look at the results. (Warning: `'` is actually a complex conjugate transpose, so complex numbers will have the sign of their imaginary parts changed. To perform a straight transposition, use the `.'` operator. For real arrays, both operators are identical.) Other useful commands for matrix manipulation include `flipud` and `fliplr`, which mirror matrices top-to-bottom and left-to-right, respectively. Look at `help elmat` for more useful functions.

Building small arrays by hand is fine, but it can become very tedious for larger arrays. There are a number of commands to facilitate this. The `ones` and `zeros` commands build matrices that are populated entirely with ones or zeros. The `eye` command builds identity matrices. `repmat` is especially useful for making matrices out of vectors. `diag` builds diagonal matrices from vectors, or returns the diagonal (vector) of a matrix. Check the `help` for all of these commands.

Note that sometimes you don't need or want to see what MATLAB returns in response to a particular command (especially if you're dealing with very large arrays). To suppress the output, simply add a semicolon, `;`, after the command.

The colon operator is one way of creating long vectors that are useful for indexing (see the next section). Execute the following commands:

```
>> 1:7
>> 1:2:13
>> 0.1:0.01:2.4
```

Each of these commands defines a row-vector. With only two arguments, as in the first command, the colon operator produces a row vector starting with the first argument and incrementing by one until the second argument has been reached. The optional middle argument (seen in the second two commands) provides a different increment amount. The colon operator is extremely useful, so it is recommended that you check out `help colon` for more details. Play with some other combinations of parameters to familiarize yourself with the behavior of this operator.

3 Array Arithmetic

MATLAB allows you to perform mixed arithmetic between scalars and arrays as well as two different types of arithmetic on vectors and arrays. Mixed scalar/array arithmetic is the most straightforward. Adding, subtracting, multiplying or dividing a scalar from an array (or an array from a scalar) is equivalent to performing the operation on every element of the array. It is also useful to note that most of the provided mathematical functions (like `sqrt` and `sin`) operate in a similar element-by-element fashion. Thus, the command

```
>> sin(d)
```

returns an array with the same size as `d` containing the sine of each element of `d`.

If we have two arrays, addition and subtraction is also straightforward. Provided that the arrays are the same size, adding and subtract them performs the operation on an element-by-element basis. Thus, the (3,4) element in the output (for instance) is the result of the operation being performed on the (3,4) elements in the input arrays. If the arrays are *not* the same size, MATLAB will generate an error message.

For multiplication, division, exponentiation, and a few other operations, there are two different ways of performing the operation in question. The first involves matrix arithmetic, which you may have studied previously. You may recall that the product of two matrices is only defined if the "inner dimensions" are the same; that is, we can multiply an mxn matrix with an nxp matrix to yield an mxp matrix, but we cannot reverse the order of the matrices. Then, the (p,q) element of the result is equal to the sum of the element-by-element product of the p^{th} row of the first matrix and the q^{th} column of the second. Division and exponentiation are defined with respect to this matrix product. It is not imperative that you recall matrix multiplication here (most likely you will see it in a linear algebra course in the future); however, it *is* important that you note that in MATLAB the standard mathematical operators (`*`, `/`, and `^`) default to these forms of the operations.

A form of multiplication, division, and exponentiation for arrays that is more useful for our purposes is the element-by-element variety. To use this form, we must use the "dot" forms of the respective operators, `.*`, `./`, and `.^`). Once again, the arrays must have the same dimensions or MATLAB will return an error. Thus, the commands

```
>> [1 2 3 4].*[9 8 7 6]
>> [7; 1; 4]./(1:3)'
>> 2.^[1 2 3 4 5 6]
```

perform element-by-element multiplication, division, and exponentiation. Note that the `.^` form is necessary even for scalar-to-array exponentiation operations.

The array arithmetic capabilities of MATLAB contribute greatly to its power as a programming language. Using these operators, we can perform mathematical operations on hundreds or thousands of numbers with a single command. This also has the side effect of simplifying MATLAB code, making it shorter and easier to read (usually).

4 Indexing

To make arrays truly useful, we need to be able to access the elements in those arrays. Consider the following commands:

```
>> a(6)
>> a(3) = 12
>> d(2,3)
```

The first command retrieves the sixth element from a vector. The second assigns a number to the third element of the same vector. For the third command, the order of the dimensions is important. In MATLAB, the first dimension is rows and the second dimension is columns. Note particularly that this is the opposite of (x, y) indexing. Thus, the third command retrieves the element from row two, column three. We can also index into matrices using single numbers. In this case, the numbers count *down the columns*. This is called "column-major" and is the opposite of array indexing in C or C++.

It is not necessary to index arrays only with scalars. One of the most powerful features of MATLAB is the ability to use one array to index into another one. For instance, consider the following commands:

```
>> a([1 4 6])
>> b(3:7)
>> c(2:2:end)
```

These commands return a subset of the appropriate vector, as determined by the indexing vector. For instance, the first command returns the first, fourth, and sixth elements from the vector `a`. Notice the use of the `end` keyword in the third command. In an indexing context, `end` is interpreted as the length of the currently indexed dimension. This is particularly useful because MATLAB will return an error if you try to access the eighth element of a seven-element vector, for instance. In general, indices must be strictly positive integers less than the length of the dimension being indexed. Thus, unlike C or C++, the indices begin at one rather than at zero.

Using multiple indices into multi-dimensional array is slightly more complicated than doing so with vectors. Consider the following commands:

```
>> d([1 3],2)
>> d([2 3],[1 4])
>> d(2,:)
```

The first command, as you might expect, returns the first and third elements of the second column. The second command returns the second and third rows from the first and fourth columns. Note particularly that this command does *not* return the individual elements at (2,1) and (1,4). (To index individual elements in this manner, we need to use single-index method along with the `sub2ind` command). The colon operator in the second command is a shortcut for `1:end`; thus, the third command returns all of the second row.

Two very useful commands that can be used to facilitate indexing are `size` and `length`. `size` returns a vector containing the length of each dimension of an array. Alternately, `size` can be used to request the length of a single dimension. `length` is primarily useful for vectors when you're not sure about their orientation. `length` returns the length of the longest dimension. Thus, `length(v)` is the same whether `v` is a row-vector or a column-vector, but `size(v,1)` will only properly return the length of a column-vector.

It is important to note that all of these indexing techniques are used not only to retrieve many elements from an array but also to set them. When performing array assignment, you must be careful to make sure the array being assigned has the same size as the array to which it is being assigned. For instance, consider the following command:

```
>> d([1 3],[2 4]) = [9 8; 7 6]
```

Note that both of the matrixes on the left and right of the equal sign are 2x2, so the assignment is valid. Look at the results of this command and make sure you understand why it does what it does.

One last command that is extremely useful in context of indexing in MATLAB is `find`. `find` will return a vector containing the indices of any nonzero elements in an array. Note that `find` uses the single-index indexing scheme that was mentioned earlier. At first glance, this has relatively few uses; however, it is in fact extremely useful because of the behavior of conditional statements in MATLAB (i.e., `>`, `<`, and `==`). The command `a > 5` will return an array with the same size as `a`, but with each element either 1 or 0 depending on whether or not it is greater than 5. Using `find` on this array will provide the indices of elements greater than 5. One particularly good use of the `find` command is the following contexts. Suppose you wish to set all negative elements in a matrix to zero. You can do this with a single command like so:

```
>> m = [-1 5 10; 3 -8 2; -4 -9 -1];
>> m(find(m < 0)) = 0;
```

Alternately, if you wish to square every element that is greater or equal to 4, you can use the find command twice in a single line, like this:

```
>> m(find(m >= 4)) = m(find(m >= 4)).^2;
```

5 Data Visualization

So now we know how to build arbitrarily large arrays, populate them with interesting things, and get individual elements or sections of them. However, pouring over pages and pages of numbers is generally not much fun. Instead, we would really like to be able to visualize our data somehow. Of course, MATLAB provides many options for this. Let's start by building a vector we can use throughout this section, and then looking at it. Execute the following commands:

```
>> x = sin(2*pi*(1:200)/25);
>> plot(x);
>> zoom on;
```

The first command builds up a sine wave, and the second command plots it. A window should have popped up with a sine wave in it. Notice the y-axis extents from -1 to 1 as we would expect. Using this form of `plot`, the x-axis is labeled with the index number; that is, our vector has 200 elements, and so the data is plotted from 1 to 200. The third command turned on MATLAB's zooming capabilities. If you left-click on the figure, it will zoom in; right-clicking will zoom out. You can also left-click and drag to produce a zoom box that lets you control what where the figure zooms. Experiment with this zoom tool until you're comfortable with it. Depending on the version of MATLAB that you are using, there may also be an icon of a magnifying glass with a `+` in it above the figure; clicking this icon will also enable and disable zoom mode.

If you zoomed in closely enough on the plot, you probably noticed that the signal isn't perfectly smooth. Instead, it is made up of line segments. This is because our vector, `x`, is made up of a finite collection of numbers. MATLAB defaults to *interpolating* between these points on the plot. You can tell MATLAB to show you where the data points are, or to not interpolate, by changing the line and point styles. Try each of these commands and look at the results before executing the next one:

```
>> plot(x,'x-')
>> plot(x,'o')
>> plot(x,'rd:')
```

`help plot` lists the various combinations of characters that you can use to define line style and color. There are a few similar commands for plotting vectors as well. Try these commands, and make sure you zoom in on each one so you can see the results:

```
>> stem(x)
>> stairs(x)
>> bar(x)
```

When you checked the `help` for `plot` (you *did* look at the `help`, didn't you?), most likely you noticed that there are some more explicit ways to use the function. There is an optional first parameter that gives the x-position of each data point. Thus, we use plot for x-y scatter plots and other things. Calling plot without the first parameter is equivalent to the following command:

```
>> plot(1:length(x),x,'x-');
```

We can do other interesting things by adjusting this first parameter. Try this:

```
>> plot(log2(1:length(x)),x);
```

A signal that looks like this is called a "chirp." We'll see why later on when we listen to some signals.

It is also possible to plot multiple vectors simultaneously. One way requires two parameters for each vector. A second makes use of the fact that `plot` will plot the columns of a matrix as separate lines. Execute the following commands. The first builds a second vector, while the second and third show alternative ways of plotting them.

```
>> y = .8*sin(2*pi*(1:200)/14 + 0.5);
>> plot(1:length(x),x,1:length(y),y);
>> plot([x; y]');
```

You're probably not surprised by now that MATLAB also has facilities for visualizing two dimensional arrays. Let's look at some of them. First, we need an interesting matrix to look at. Execute the following command:

```
>> z = membrane(1,50);
```

We now have a 101x101 matrix of numbers. The most straightforward way to look at this data is using the `imagesc` command, which displays the matrix as though it were an image. Execute the following commands:

```
>> imagesc(z); axis xy; colorbar;
```

Our surface has been displayed in color. Notice the colorbar along the right side of the image, which tells what values the various colors map to. This type of display, where different colors are used to represent differing values, is known as a pseudocolor display. If we look at the image we've got a "high" spot in the lower right that tapers off to "low" regions around the outside. The surface also has an overall L-shape. Another way to visualize this uses the `contour` command. Try this:

```
>> contour(z,20); colorbar;
```

This display, the *contour plot*, shows us lines of constant height. This is the way that meteorologists usually display atmospheric pressure on weather maps.

We also have some more interesting options. Try the following commands:

```
>> mesh(z); rotate3d on
>> surf(z); rotate3d on
```

Now we have some "3-D" visualizations of our surface. If you click-and-drag the plot, you should be able to rotate the surface so that you can see it from various directions. Experiment with this until you're comfortable with how it works. Notice what happens if you look at the surface from directly above.

MATLAB has some very powerful tools for data visualization; here, you've seen only a small sampling. There many more. If you're interested in exploring this topic further, check `help graph2d`, `help graph3d`, and `help specgraph`.

6 Programming in MATLAB

As has been mentioned, programming in MATLAB is really just like using the environment at the command line. The only difference is that commands are placed in a file (called an *M-file*) so that they can be executed by simply calling the file's name. We'll also see that MATLAB has many of the same control flow structures, like loops and conditionals, as many other programming languages.

Before we jump into programming in MATLAB I need to make a few comments about files in MATLAB. MATLAB has access to machine's file system in roughly the same way a command line based operating system like DOS or UNIX. It has a "present working directory" (which you can see with the command `pwd`); any files in the present working directory can be seen by MATLAB. You can change the present working directory in in roughly the same way that you do in DOS or UNIX, using the `cd` command (for "change directory"). MATLAB also has a "path," like the path in DOS or UNIX, which lists other directories that contain files that MATLAB can see. The `path` command will list the directories in the path. We'll be making a few files in this tutorial, and you'll need to store commands in files when doing the laboratories. You'll probably want to make a directory somewhere in

your personal workspace, `cd` to that directory, and store your files there. Unless you're working on your own system, do not store them in the main MATLAB directory; if you do, the system's administrator will probably become very irritated with you.

There are two types of files containing commands that MATLAB can call. Both use the ".m" file extension. The first type is a *script*, which contains nothing but a list of commands. When you call the script (by simply typing in the script's filename), MATLAB will execute all of the commands in the file and return to the command line exactly as if you had typed the commands in by hand. Let's write a short script.

First, we need an editor. If you're using a Windows or Macintosh system, or MATLAB 6.0 on a UNIX, just type in `edit` at the MATLAB prompt and text editor will pop up. Older versions of MATLAB under UNIX do not have a built-in editor, so you'll need to use some other editor, like emacs or vi. Use the system to open a new terminal and open your favorite text editor (`pico` is a UNIX text editor that is easy to use if you don't already have a favorite). Place the following lines in the text file and save it as "hello.m".

```
% hello.m  --  Introductory 'Hello World' script
% These lines are comments, because they start with '%'
hw = 'Hello World!';
disp(hw);
```

Now execute it by typing `hello` at the MATLAB prompt. (Remember that the file needs to be in your present working directory or on the path for MATLAB to see it – `cd` to the correct directory if necessary). As a result of executing the script, you should now have a variable 'hw' in your workspace (remember, `who` lists variables in your workspace).

The second type of file that we can put commands in is called a *function*. A function communicates with the current workspace by passing parameters. It also creates a separate workspace so that it's variables don't get mixed up with whatever variables you have in your current workspace. Using your text editor, make a new file that contains the following lines and save it as "hello2.m."

```
% hello2.m  --  Introductory 'Hello World' function
% Try typing 'help hello2' when you're done, and see what happens
%
% function output = hello2(input)

function output = hello2(input)
% The line above tells MATLAB that this is a function
%   with one input and one output parameter

hw2 = ['Hello World! x' num2str(input)];
disp(hw2);
output = hw2;
```

To call this function, type `hello2(2)`. Note that once you've done this, the variable `hw2` does not show up in your workspace. However, the data that was stored in `output` (the output parameter) has been placed in `ans`. This is exactly what happens if you called a MATLAB built-in function without supplying an output parameter. Similarly, the '2' is an input parameter which is passed into the function. When a function is executed, it will not have any variables defined except those defined inside the function itself and the input parameters. For further help on these two types of M-file, type `help function` or `help script` at the MATLAB prompt.

In MATLAB we also have a number of programming constructs at our disposal. While primarily used in M-files, these constructs can also be used at the command line. However, anything complicated enough to need a loop or an if-statement is usually worth putting into an M-file. Let's look at the most typical types of programming constructs.

The `for` loop is used to execute a set of commands a certain number of times, while also providing an index variable. Consider the simple loop here:

```
for index = 1:10
   disp(index);
end
```

This loop executes the `disp` command ten times. The first time it is executed, `index` is set to 1. Thereafter, it is incremented by one each time the commands in the loop are executed. Note that the colon form of the `for` loop is not mandatory; any *row-vector* can be used in its place, and the index (which, of course, can be renamed) will be sequentially set to each of the elements in the vector from left to right. We can use `while` loops in a similar manner. Consider this:

```
ct = 10;
while ct > 0.5
   ct = ct/2;
   disp(ct);
end
```

As long as the conditional after `while` is true, the loop will be executed.

A more traditional method of conditional execution comes from the `if-else` statement. Consider this:

```
if pi > 4
   disp('Pi is too big!');
elseif pi < 3
   disp('Pi is too small!');
else
   disp('Pi is just about right.');
end
```

Here, MATLAB will first check the conditional, `pi > 4`. If this is true, the first display command will be executed and then the remainder of the `if-else` statement will be skipped (that is, none of the other conditionals will be tested). If the first conditional is false, MATLAB will begin to check the remaining conditionals. There can be any number of `elseif` statements in this construct (including none), and the `else` statement is entirely optional. If you have a large number of chained conditionals, you might consider using the `switch-case` construct (type `help switch` or `help case`).

7 Debugging your MATLAB code

Inevitably, when you put MATLAB commands into a file as a script or a function, you will make mistakes and need to locate them. Because of its interpreted environment, MATLAB is actually one of the most pleasant languages to debug. And, as is always the case when debugging code, there are many ways to accomplish this.

If you are executing a script or function and MATLAB encounters an error, it will immediately print the line number of the function on which the error occurred. If the error occurs in a file other than the calling file, a call stack will be printed. This listing shows which file called which other files and on what line number. This allows us to pinpoint the source of the error quickly.

One of the simplest ways to debug is a method you are probably familiar with from other programming languages. We can force MATLAB to print strings or variables using the `disp` command. This way, we can display `for` loop counters or other relevant variables to determine what they contain and exactly when in the program flow the code "breaks."

The real power of MATLAB debugging comes from our ability to "break" at any point in the code and then proceed to execute any MATLAB commands. There are a number of ways to do this. For instance, you can tell MATLAB to stop and enter "debug mode" whenever an error is encountered. When you're in debug mode, the command line changes to `K>>`. You will then have access to all of the variables that are in scope at the time. Turn on this option with the command

```
>> dbstop if error
```

To turn it off again, use the command

```
>> dbclear if error
```

We can also set and clear breakpoints elsewhere in the code using the same commands. To set (and then clear) a breakpoint in `hello2.m` at line 11, call

```
>> dbstop in hello2 at 11
>> dbclear in hello2 at 11
```

`dbstatus` will show all breakpoints that are currently active. Note that if you try to set a breakpoint at a non-command line (like a comment), the breakpoint will be set at the next valid command.

Another useful command is `dbstep`, which advances one command in the m-file. If you call `dbstep in` or `dbstep out`, you can step into and out of called functions (that is, you traverse up and down the call stack, which contains a list of which functions have been called to reach the current point in the code). `dbstack` lists the current call stack including your current file and the line number in this file. `dbtype` types all or parts of an m-file. Eventually, you'll want to get out of debug mode, so you can call `dbquit` to halt execution of the file or `dbcont` to continue execution until the end of the file or the next breakpoint. In general, `help debug` is the starting point in the help system for learning about the MATLAB command line debugger.

If you are running MATLAB on a Windows system (or possibly a Macintosh), the debugger is also available through the built in editor. The exact implementation depends on your system and the version of MATLAB, but usually breakpoints will show up as red circles next to commands. In debug mode, the current command will be pointed to with an arrow, so you can follow where you are in the code. There are typically shortcut keys and menu items to insert and remove breakpoints, step through the code, and toggle flags such as stop-if-error.

If you save a file that has breakpoints, you may find that your breakpoints disappear. This can be very annoying, so there is an alternative method of entering debug mode. Placing the command `keyboard` into your code is effectively the same as placing a breakpoint in the code, such that you can execute commands before returning to program execution (with the command `return`).

There are a number of error types that you are likely to encounter. One very good rule of thumb says that if an error occurs inside a MATLAB function, the error is almost assuredly in the calling function. Usually this means that the function is being passed improper parameters; check the call stack or `dbstep` out until you find the line in your program which is causing problems. Other common errors include indexing errors (indexing with 0 or a number greater than the length of the indexed dimension of a variables) and assignment size mismatches. MATLAB is usually pretty descriptive with its error messages once you figure out how to interpret what it is saying. As is usually the case when debugging, an error message at a particular line may in fact indicate an error that has occurred several lines before.

List of Commonly Used MATLAB Commands

Elementary Math Functions

abs	atan	exp	log	rem	sqrt
acos	ceil	fix	log10	round	tan
angle	conj	floor	mod	sign	
asin	cos	imag	real	sin	

Graphing and Plotting Functions

axis	figure	line	print	stem	xlabel
bar	grid	loglog	semilogx	subplot	ylabel
clf	hold	plot	semilogx	text	zoom
close	legend	polar	shg	title	

Relational and Logical Functions

all	eq (==)	ge (>=)	isempty	isnan	not (~)
and (&)	exist	gt (>)	isfinite	le (<=)	or (|)
any	find	ischar	isinf	lt (<)	strcmp

Programming and Control Flow

break	disp	end	if	pause	try
case	else	error	input	return	warning
catch	elseif	for	otherwise	switch	while

General Purpose Functions

who	length	exit	which	exist	
whos	clear	quit	lookfor		
size	why	help	helpdesk		

File and Directory Functions

cd	fclose	load	path	save	what
dir	fopen	mkdir	rmdir	type	

Debugging Commands

dbclear	dbquit	dbstatus	dbstop	dbup	
dbcont	dbstack	dbstep	dbtype	keyboard	