

LIFE, THE UNIVERSE AND MATLAB[®]

A Brief Guide to Programming in MATLAB

Kyle D. Dippery
Department of Engineering Mechanics
University of Kentucky
Lexington, KY 40506-0046

kdip@engr.uky.edu

Copyright © 1995 by Kyle D. Dippery

Last Revised November 10, 1995

CONTENTS

1.0 INTRODUCTION	1
2.0 USING MATLAB	2
2.1 Programming	2
2.1.1 Entering Scalars	2
2.1.2 Entering Arrays	2
Vectors	2
Matrices	3
Strings	4
Array Sizes	5
Dealing With Arrays	5
Variable Names	6
Managing Memory	7
2.1.3 Data Input and Output	7
Keyboard Input	7
Screen Output	7
Capturing Screen Output	8
File I/O	8
2.1.4 Mathematical Operators	9
2.1.5 Loops and Logic	10
Logic	12
2.2 Using m-files: Scripts and Functions	14
Argument Lists	15
Stopping Functions	16
Suspending Functions	17
Global Variables	17
2.3 Comments	18
3.0 FUN WITH FIGURES	19
Subplots	19
Plot Formats	19
Scaling Plots	20
Labelling Plots	20
Labelling Plots Interactively	20
Adding to Plots	21
3.1 Handle Graphics	22
3.1.1 Object Properties	23
4.0: WHAT NOW?	24
APPENDIX A: A BRIEF CATALOGUE OF FUNCTIONS	25
Operators and Punctuation	25

Some Useful Scalars	26
Some Useful Matrix Functions	26
Important Things We Can't Find a Good Category For	27
Logic and Loops	27
Graphing Functions	28
File Input/Output Functions	29
Screen/Keyboard Input/Output	30
Miscellaneous	30
APPENDIX B: SOME HANDLE GRAPHICS PROPERTIES	32
Figure Properties	32
Axes Properties	33
Line Properties	34
Text Properties	35
EXERCISES	37
FURTHER READING	38

1.0 INTRODUCTION

MATLAB[®] is a comprehensive engineering and mathematical computing environment. It has its own programming language, but it is much more than a programming language. It is also a library of routines for solving matrix problems, and a graphics package that is easy to use, but can be as flexible as the user needs it to be.

This document is intended to be a brief introduction to MATLAB, its language, and some of its capabilities. It is not intended to be a complete reference guide. Although some commands are described in a reference section, many more have not been included. Much more information is available about all of MATLAB's commands through the on-line help feature and through the official MATLAB reference guide.

2.0 USING MATLAB

MATLAB can be used interactively, with commands input directly from the keyboard, or through previously-written programs, called *m-files*. Each of these modes has certain advantages over the other; the interactive mode, for example, is much more flexible than the script mode, while *m-files* allow for easy repetition of long sequences of commands. These modes are not mutually exclusive, however; scripts are generally run when they are called from the command line, and it is possible to enter commands from the keyboard while a running script is temporarily suspended.

2.1 Programming

MATLAB programs do not need to be compiled before they can be used. They do not, in fact, have to be *programs* at all. Rather, a MATLAB session could consist of a number of commands entered directly from the keyboard. These commands can be assignments, loops, logical operations, comments, or any other legal MATLAB expression.

2.1.1 Entering Scalars

A scalar value can be assigned to a variable simply by entering it. For example, typing `a=5` will result in the statement:

```
a =  
5.
```

Similarly, typing `cos(pi)` will yield,

```
ans =  
-1.
```

The first lesson to be learned, then, is this: MATLAB interprets any keyboard input as some type of command. If the keyboard input contains an equal sign (=), MATLAB reads it as an assignment statement; the numerical result on the right hand side of the equality is assigned to the variable on the left hand side. If there is no variable and no equal sign, MATLAB reads it as an implicit assignment, and assigns the input value to the variable `ans` (so, `cos(pi)` is equivalent to `ans=cos(pi)`). If the statement begins with a percent sign (%), MATLAB will read it as a comment, and will disregard that line completely.

A minor lesson to be learned from this is that MATLAB implicitly recognizes some of the better-known mathematical constants. `pi`, for instance, is given as `3.14159265358979` (MATLAB defaults to double-precision arithmetic). Similarly, `i` and `j` are recognized as the imaginary unit, `sqrt(-1)` (displayed as `0 + 1.0000i`).

2.1.2 Entering Arrays

MATLAB treats every variable as an array of complex numbers. In other words, there is no difference between the procedures for entering real scalars, real arrays, complex scalars, and complex arrays.

Vectors

A vector of numbers ($m \times 1$ or $1 \times n$ array) is entered in square brackets:
`a=[1 3 5 6 7]` yields,

```
a =  
1    3    5    6    7.
```

If the vector is to consist of a sequence of regularly-spaced values, a colon may be used to make the entry more concise: both `b=[1 4 7 10 13 16]` and `b=1:3:16` ("start at 1, skip by three, stop at 16") yield

```
b =  
    1     4     7    10    13    16
```

The value by which a vector's elements are incremented does not need to be an integer. It does not even need to be positive. Both `c=0.5:0.0025:15` and `d=10:-pi:-15` are legal vector assignments. If no increment is specified, the increment is 1. Invalid sequences will return an empty answer (eg, `10:-25` returns `[]`, because the increment defaults to 1, and a positive increment, starting at 10, will never reach -25).

The final value in a vector assignment does not need to be an element of the sequence being assigned; the last element of the sequence that falls within the specified bound will be the last element of the vector. For example, both `b=1:3:16` and `b=1:3:17` will yield the vector shown above.

A column vector is most easily entered as the transpose of a row vector:
`b = [1:3:16]'` will yield

```
b =  
     1  
     4  
     7  
    10  
    13  
    16.
```

Square brackets are needed in this case because of the presence of the transpose operator. The transpose operator (`'`) will be discussed in Section 2.1.4, *Mathematical Operators*.

Alternatively, either a semicolon or a carriage return may be used to separate elements of a column vector: `b=[1;4;7;10;13;16]` will also yield the column vector shown above.

Matrices

A matrix ($m \times n$ array) is entered as a series of row vectors, separated by semicolons, or a series of column vectors, separated by spaces:

`A=[1 3 5 6 7; 2 4 8 9 10]` yields,

```
A =  
     1     3     5     6     7  
     2     4     8     9    10
```

`A2=[[1;4;7] [10;13;16]]` yields,

```
A2 =  
     1    10  
     4    13  
     7    16
```

In the first example, every row vector used in `A` must contain the same number of elements. In the second example, every column vector must contain the same number of elements. It is important to remember this when arrays are constructed from other arrays, whose dimensions may vary.

The vectors used to build an array may themselves consist of arrays: `B=[A ones(2,2); zeros(2,2) A]` yields the 4x7 array,

```
B =  
  
     1     3     5     6     7     1     1  
     2     4     8     9    10     1     1  
     0     0     1     3     5     6     7  
     0     0     2     4     8     9    10.
```

Two special arrays have been used in this example. `ones(m,n)` produces an $m \times n$ array of 1's. `zeros(m,n)` produces an $m \times n$ array of 0's. Some other built-in array generators are `rand`, `diag`, and `eye`. These will be discussed briefly in Appendix A.

Strings

Character strings may be assigned to variables by enclosing them in single quotation marks. If an apostrophe is needed within a string, it is denoted by two single quotes. For example, `a='Hi there, I am an apteryx'` would return,

```
a =  
  
Hi there, I am an apteryx  
  
b='Eddie''s place' would return  
  
b =  
  
Eddie's place
```

Character strings are essentially treated as row vectors. Strings may be concatenated by placing them side by side, within square brackets. For example, `['Eddie''s' 'place']` would return the string `b`, above. Strings may also be combined into rectangular arrays by using the `str2mat` command. `str2mat(a, 'at', b)`, where `a` and `b` are as shown above, would return,

```
Hi there, I am an apteryx  
at  
Eddie's place
```

where the smaller strings, `'at'` and `b`, have been padded with spaces to make them the same length as the longer string, `a`.

Numbers may be converted into strings by the `num2str` command. `num2str` may be used with square brackets to concatenate strings with numeric quantities. For instance, if `q` is defined as 17.5, the command `disp(['q = ' num2str(q)])` would produce the display,

```
q = 17.5
```

This can be useful for constructing header strings or graph labels from variables in the workspace.

Similarly, `str2num` may be used to rescue a numeric value that is trapped in the form of a string variable. For instance, if the string `'q = 17.5'` is assigned to the variable `trap`, `q = str2num(trap(5:8))` would assign the numeric value 17.5 to the variable `q`.

Strings that look like MATLAB commands can be executed through use of the `eval` command. For example, `eval(['load ' datafile])` would load the data file whose name is contained in the `datafile` variable. Or, in the above example, `eval(trap)` would assign the value 17.5 to `q`.

A string may be converted to ASCII codes by using the `abs` function. `abs('Hi!')` would return the three-element row vector `[72 105 33]`, where each element is the ASCII code of the corresponding element of `'Hi!'`. A

set of ASCII codes may be converted to a string by using the `setstr` function. `setstr([72 105 33])` would return the string `'Hi!'`.

Mathematical operations on a string are actually performed on the string's ASCII codes. `'Hello'/2`, for instance, is the same as `abs('Hello')/2`.

Note: `abs` is the same function that returns the magnitudes of complex numbers, or the absolute value of real numbers.

Array Sizes

The size of a matrix can be found by entering `size(B)`. This returns a two-element row vector, `[Nrows, Ncols]`. For `B` as defined above, the size is returned as,

```
ans =  
     4     7.
```

A similar command, `length`, can be used to determine the size of a vector. In essence, `length` returns the largest element of the `size` vector. For example, `length(a)` (where `a=[1 3 5 6 7]` from before) returns

```
ans =  
     5.
```

`size(a)`, by comparison, returns the two-element vector,

```
ans =  
     1     5.
```

`length` may also be applied to matrices. `length(B)`, for example, yields

```
ans =  
     7.
```

Dealing With Arrays

It is not always necessary, or desirable, to operate on an entire array. In such instances, it is possible to extract data from one array and either operate on it, or assign it to another variable and operate on that. This is accomplished through the use of indices, where the first index indicates the row(s) to extract and the second index identifies the column(s).

Indices can be scalars or vectors. For example, `A(1,3)` returns the third element of the first row of `A`. Similarly, `A(2,[1 3 4])` returns a 3-element row vector consisting of the first, third and fourth elements of the second row of `A`.

Because indices are essentially vectors, colons may sometimes be used to write them more concisely. As an example, `A(1,1:3)` returns the first three elements of the first row of `A`, while `A(1,1:2:5)` returns the first, third, and fifth elements of the first row.

Column vectors may be extracted by indexing multiple rows of a single column (`A(1:4,1)`, for example). Submatrices are extracted by indexing multiple rows and columns. A colon by itself as an index means either "all the rows" or "all the columns", depending on its position. (`A(:,5:7)` returns the entire fifth, sixth and seventh columns of `A`; `A(5:7,:)` returns all of the fifth, sixth and seventh rows of `A`.)

Vectors will accept two indices, but only one is required. If both are given, care must be taken to ensure that the order of the indices matches the dimension of the vector. For example, if `b` is a row vector, `b(:,5)`, `b(1,5)`, and `b(5)` will all return the fifth element, but `b(5,1)` will cause an error.

Elements of an existing array may be overwritten using indices. If, for instance, A is defined as the array,

```

5     7     3     4     0
7     4     1     1     3
8     5     4     8     7
8     9     0     0     5

```

the 8's in the third row may be changed to 2's by the assignment,

```
A(3,[1 4]) = [2 2];
```

to yield

```

5     7     3     4     0
7     4     1     1     3
2     5     4     2     7
8     9     0     0     5

```

Similarly, the 2x2 block in the upper left-hand corner,

```

5     7
7     4

```

may be replaced by a multiple of itself,

```
A(1:2,1:2) = A(1:2,1:2) .* 2;
```

giving,

```

10    14     3     4     0
14     8     1     1     3
 2     5     4     2     7
 8     9     0     0     5

```

Finally, the colon can be used to turn an array into a vector. Writing `A(:)` is the same as writing `[A(:,1); A(:,2); A(:,3); A(:,4); A(:,5)]`. Both of these expressions result in,

```

10
14
 2
 8
14
 8
...
 0
 3
 7
 5

```

This can simplify things a bit, in some applications (such as `min`, `max`, `any` and `all`). For instance, `min(min(A))` is the same as `min(A(:))`.

Variable Names

Variable names in MATLAB can be up to 19 characters in length. Upper-case and lower-case letters are treated as distinct, so `A` (used as a matrix, above) and `a` (used as a vector) are treated as separate arrays. Any array given a name longer than 19 characters will have its name truncated; if two arrays have long names, with the same first 19 characters, they will be treated as the same array.

Before assigning a name to an array, it is good practice to make sure that name is not already being used for some other purpose. This can be done by using the `exist` function. `exist('A')` will check the current workspace and all

directories in the search path to determine whether `A` is currently in use as either a variable or a file name. It is not good to use the name of a function (m-file or built-in) as a variable.

Managing Memory

A listing of the variables currently in memory can be obtained by entering the command `who`. A similar command, `whos`, shows a list of the variables and displays some key properties (size, real/complex, sparse/full) of each one. These can be useful in accounting for memory usage.

If memory becomes a problem, `clear` can be used to free up some space. `clear` removes variables from the workspace. Entering `clear name`, where `name` represents a variable, removes that variable from memory. If `name` represents a global variable, it remains in memory for all functions declaring it global but is removed from the current function's workspace. Global variables (see Section 2.2) may be removed by entering `clear global name`. `clear global` removes all global variables. `clear` removes all variables.

2.1.3 Data Input and Output

Input and output of data is important to any programming language. MATLAB provides several methods for data I/O to and from the keyboard and/or disk files.

Keyboard Input

Keyboard input can take the form of the direct assignment (discussed above) or can use the `input` statement. The `input` statement, which is more useful in m-file implementations than interactive sessions, takes the form:

```
variable=input('prompt');
```

`input` must be given a string input argument; this string is then echoed as the prompt for data.

When an `input` statement is encountered during m-file execution, MATLAB stops and waits for a carriage return. Whatever text is entered before the carriage return is evaluated as a MATLAB command, and the result is stored in `variable`. If nothing is entered, `variable` will be empty. Arrays may be entered with `input` statements, if they are enclosed in square brackets.

String variables may be assigned using `input`, as well; the syntax then becomes:

```
variable=input('prompt','s');
```

Anything entered with this form of the `input` statement is simply assigned to `variable`; no evaluation takes place in this instance.

Input lines that run off the edge of the screen may be continued on the next line if the long line ends with the continuation mark (...). Continuation marks are not always necessary (when inputting arrays, for instance), but are always legal. Use them if there is any doubt about syntax.

Screen Output

The semicolon may be used to control output to the screen. Whenever a command does not end with a semicolon, MATLAB echoes the results to the screen. Any command that does end with a semicolon will have its output suppressed.

The simplest method of obtaining output from MATLAB is therefore to never end a line with a semicolon; every operation will then echo its result to the screen. If `diary` is on, results will be saved in the current diary file, as well.

Another, perhaps cleaner, way of obtaining output is to use the `disp` command. `disp` evaluates its input argument and writes the result to the screen, without assigning it to any variable. If the input argument is the name of a variable

in the workspace, the contents of that variable are displayed; if the input argument is a valid MATLAB expression (`disp(cos(1.2*pi)/log(14))`), for instance, then the expression is evaluated and the result (`-0.3066`) is written.

The `format` command may be used to change the precision of data displayed on the screen. `format long` displays numeric results as 15-digit fixed point values. `format short` displays values as 5-digit fixed points (the default format). Some other useful scientific displays are `format short e` (exponential) and `format long e` (exponential with lots of digits).

`format` does not affect the precision of mathematical calculations; it only affects the precision of the display.

Other commands help to manage the amount of information displayed on a screen at one time. `clc`, for instance, clears the command window and places the cursor at the first line of the window. `more` forces data to be displayed one page (screen) at a time. `more` by itself toggles the paging routine. `more on` turns the pager on; `more off` turns the pager off.

Capturing Screen Output

Anything that is written to the command screen can also be sent to a file. `diary` is the command that accomplishes this; `diary on` turns the `diary` function on, and anything written to the screen is also written to the file `diary`. `diary off` turns the `diary` function off. `diary filename` turns the `diary` function on and sends all output to the specified file.

If the specified diary file already exists when `diary` is turned on, any new information is appended to the old file. No marker is written to the file to show where the old information leaves off and the new stuff begins.

File I/O

Files may be read or written using the `load` and `save` commands. By themselves, these commands read and write (respectively) the file `matlab.mat`. If a filename without an extension is included with the `load` or `save` command, MATLAB will append a `.mat` extension to that name. If an extension is included with the filename on a `load` command, MATLAB will attempt to load the file as ASCII; the contents of the file will then be assigned to a variable with the same name as the file. If an extension is included with the filename on a `save` command, the file must be saved as ASCII format or MATLAB will be unable to reload it.

Any file that has a `.mat` extension is assumed to be in the MATLAB format (binary); all variables are preserved within a `mat-file`. Any file saved as ASCII format will not preserve the variable structure.

Some sample load and save commands:

<code>save</code>	saves entire workspace to <code>matlab.mat</code>
<code>load</code>	loads file <code>matlab.mat</code>
<code>save eddie</code>	saves entire workspace to <code>eddie.mat</code>
<code>save eddie A B</code>	saves arrays A and B to file <code>eddie.mat</code> .
<code>load eddie</code>	loads file <code>eddie.mat</code>
<code>save eddie.dat</code>	saves entire workspace to <code>eddie.dat</code> . File is in MATLAB format and will be difficult to reload.
<code>save eddie.dat -ascii</code>	saves entire workspace to file <code>eddie.dat</code> , in ASCII format. Variables are saved in 8-digit precision. No distinction is made between the variables within the file.
<code>save eddie.dat A B -ascii</code>	saves arrays A and B to file <code>eddie.dat</code> , in 8-digit ASCII format.

`save eddie.dat A B -ascii -double` saves A and B to `eddie.dat` in 16-digit ASCII format.

`load eddie.dat` loads ASCII file `eddie.dat`, assigns contents to variable `eddie`. All rows of the file must contain the same number of columns. String data is not allowed in this format.

If specific file formats are needed, file input/output can also be accomplished through commands such as `fprintf`, `fwrite`, `fread`, and `fscanf`. `fprintf` and `fscanf` are used for writing and reading ASCII files; `fread` and `fwrite` are used for writing and reading binary files. For more information, see the MATLAB manual or `help` on any of these items.

2.1.4 Mathematical Operators

Much of MATLAB's power derives from its treatment of variables as arrays. For basic operations, this makes MATLAB somewhat independent of the `for` loop. While some loops are unavoidable, as the `for` or `while` loops become larger and more complex, MATLAB's performance suffers and programs run slower. "Proper" MATLAB programming uses the array nature of variables to minimize dependence on `for` and `while` loops.

For example, a program might include the following loop to add 3 to a list of numbers:

```
for i=1:5,
    b(i)=a(i)+3
end
```

While this is perfectly legal within MATLAB, it is, at the least, the long way to do it. A much more concise method would be to say, simply,

```
b=a+3.
```

This works for almost all of the elementary operations, and most of the functions found in MATLAB. `a-3` subtracts 3 from each element of `a`; `a+b` adds each element of `a` to the corresponding element of `b` (if `a` and `b` are the same size); `sin(a)` returns the sine of each element of `a`; `log(a)` returns the natural logarithm of each element of `a`, and so on.

Problems can arise, however, with the multiplication (`*`), division (`/`), or power (`^`) operations. These are defined within MATLAB to be inherently matrix operations. `A*B` is interpreted as the matrix `A` times the matrix `B`, where `A` is assumed to be $m \times n$ and `B` is assumed to be $n \times p$. `A/B` is read as `A * inv(B)`, or `A * B-1`, the solution to the equation

$$x * B = A,$$

where `x` is a $p \times m$ matrix, `B` is an $m \times n$ matrix (a least squares solution is applied if `B` is not square), and `A` is a $p \times n$ matrix.

`A\B` is similarly interpreted as `inv(A) * B`, or `A-1 * B`, or the solution to

$$A * x = B,$$

where `A` is an $m \times n$ matrix (a least squares solution is applied if `A` is not square), `x` is an $n \times p$ matrix, and `B` is an $m \times p$ matrix.

While this is all very nice when dealing with matrices, sometimes it doesn't work very well when scalars are involved. For example, while `a/2` nicely divides each element of our vector `a` by 2, `2/a` attempts to solve the equation,

$$x * a = 2,$$

rather than dividing 2 by each element of `a`.

If `a` is a 1×5 array, as defined above (`[1 3 5 6 7]`), and 2 is 1×1 (as it is, usually), MATLAB ends up trying to multiply a 1×1 matrix into a 5×1 matrix, and crashing on a dimension disagreement. The way around this is to precede

the / by a period (.). This tells MATLAB to regard the division as a scalar operation, acting on each element of a. So, while 2/a crashes, 2 ./ a will yield the desired result. (Note to the curious: a space is required between the 2 and the period. Otherwise, the period would be read as a decimal point, and we'd have the matrix dimension disagreement all over again.)

Multiplication of a matrix by a scalar doesn't have this problem. 2*a and a*2 would each yield a 1x5 array whose elements were the elements of a multiplied by the element of 2.

Arrays may be multiplied by the operator "dot-star" (. *). This is an element-by-element multiplication of two arrays of the same dimension. That is, if A and B are defined as

```
A = [ 1    2;
      2    3;
      3    4]

B = [ 2    3;
      3    4;
      4    5]
```

then C = A .* B (no space is required, this time, but it doesn't hurt to have it there) would yield the array,

```
C = [ 2    6;
      6   12;
     12   20].
```

Arrays may be divided element-by-element using the "dot-divide" operations, ./ and ./\ . Using the arrays defined above, A ./ B would result in an array whose elements are the elements of A divided by the elements of B. A ./\ B would yield an array whose elements are the elements of B divided by the elements of A. (A ./\ B and B ./ A are equivalent statements.)

Powers are just plain ugly. To generate an array whose elements are the elements of A raised to the nth power, the proper expression is

```
A .^ n
```

(again, when only variables are involved, the spaces aren't required, but they make the expression look nice). For anything else, either A or n has to be a square matrix. For more information on this, see help arith.

When dealing with the transpose operator ('), the period takes on a slightly different meaning. A transpose operator by itself denotes the complex conjugate (*Hermitian*) transpose. A "dot transpose" (.') is a simple transpose. For real-valued matrices, there is no difference between ' and .' .

An example: A = [1+2i 2+3i; 1-3i 3-2i]' yields

```
A =
    1.0000 - 2.0000i    1.0000 + 3.0000i
    2.0000 - 3.0000i    3.0000 + 2.0000i,
```

while A = [1+2i 2+3i; 1-3i 3-2i].' yields

```
A =
    1.0000 + 2.0000i    1.0000 - 3.0000i
    2.0000 + 3.0000i    3.0000 - 2.0000i.
```

2.1.5 Loops and Logic

Although the array operations discussed above remove some of the dependence on loops, some looping will be impossible to avoid. Looping is accomplished through the for and while commands. The syntax for each of these

types of loops is essentially the same, except that `for` loops generally run a specific number of times (determined by the length of the index vector), while `while` loops run an indefinite number of times (generally until a specified condition is no longer true).

A simple `for` loop would be structured as,

```
for Index=[1 2 3 4 5],  
    disp(Index)  
end
```

Similar output can be obtained from a `while` loop:

```
while Index <= 5,  
    disp(Index)  
    Index=Index+1;  
end
```

The primary difference between these two loops is that, in the second loop, the variable `Index` is changed within the loop. This is necessary for `while` loops; if the index variable never changed, the `while` condition would always be either true or false, and the loop would either never end or never begin. In `for` loops, by contrast, it is not generally good practice to alter the index variable within the loop. While this isn't strictly illegal (it probably won't result in an error message), it could cause a great deal of confusion (and lead to incorrect results).

Some popular choices for the index variable are `i` and `j`. Sometimes, these might actually be poor choices, because `i` and `j` are recognized by MATLAB as `sqrt(-1)`. Using either of them as an index variable overwrites the implicit definition, which could then lead to difficulties if the implicit definition should happen to be needed afterward. Better choices for index variables might therefore be `Index`, as above, or dummies such as `ii` or `jj`.

It is possible to nest loops. Each loop must have its own index variable and its own `end` statement. Loops are terminated in the opposite order from that in which they are opened. For example, the nested loops

```
for ii=1:5,  
    for jj=1:3,  
        A(ii,jj)=(ii+jj)/3;  
    end  
end
```

% ends jj loop
% ends ii loop

fill the 5x3 array `A`, row by row.

Loops can be prematurely ended by the `break` command. If loops are nested, `break` ends the innermost loop only.

Some notes about loops:

1. It is not necessary to indent the contents of a loop, as shown above. This does make a program easier to read, however.
2. In the nested loop example, the matrix `A` must be regenerated each time the loop is entered. Initially, `A` is undefined. After the first pass through the inner loop, `A=2/3`. After the second pass through the inner loop, `A` has been resized and now `A=[2/3 1]`. This constant resizing of the matrix is time-consuming and, as `A`

grows, takes longer each time. A much faster method of filling A is to define it first, then fill it:

```
A=zeros(5,3);
for ii=1:5,
    for jj=1:3,
        A(ii,jj)=(ii+jj)/3;
    end
end
```

Logic

Branching can occur within a MATLAB session by means of the logical operators and the `if` command. The syntax of the `if` statement is very similar to that of the `while` loop:

```
if Something >= 0,
    disp(Something)
end
```

Multiple branches are obtained by combining `if` with `else`:

```
if Something >= 0,
    disp(Something)
elseif Something == -5,
    disp(Something / 2)
else
    disp(['Not now, I''m right in the middle of a Rothschild'])
end
```

The first line of any `if` branch should be some kind of comparison or a check for some property. Comparisons and property checks return the logical value 1 for true statements, 0 for false statements. Comparisons are made by use of the relational operators:

<code>==</code>	equal to
<code>~=</code>	not equal to
<code><</code>	less than
<code><=</code>	less than or equal to
<code>></code>	greater than
<code>>=</code>	greater than or equal to

It is important to notice that there is a distinction between `=`, the assignment statement, and `==`, the comparison symbol.

Some property checks that may be made include:

<code>isstr(a)</code>	true if a is a string variable
<code>isnan(a)</code>	true if a is “not-a-number” (NaN)
<code>isempty(a)</code>	true if a is an empty array (one or both dimensions is zero)

`isinf(a)` true if a is infinite (Inf)

`issparse(A)` true if A is designated as a sparse matrix

`exist('a')` true if either a function in the search path or a variable in the workspace has the name 'a'. (Notice that the input argument in this case must be a string variable. Also, this function differs from the others in that values other than 1 may be returned. For more on this, see `help exist`.)

A tilde (logical *not*) may be combined with any of these checks to create checks for the absence of a particular property. `~isempty(a)`, for instance, is true if neither dimension of a is 0.

Several comparisons and property checks may be combined into single `if` statements by using the logical *and* (`&`), logical *or*, (`|`), and *exclusive or* (`xor`) operators. To illustrate, `a == 2 & ~isnan(b)` is true only if a contains the value 2 and b does not contain the “not-a-number” value.

When any of these comparisons, or some of the property checks (`isnan`, for example) are applied to arrays, they are in fact applied to each element of those arrays, resulting in an array of ones and zeros, of the same size as the original arrays.

For example, if A is defined as the 2x3 array,

2	3	4
8	2	5

and B is the 2x3 array,

5	4	3
6	2	2

then the comparison `A <= B` compares each element of A to the corresponding element of B, returning the 2x3 array,

1	1	0
0	1	0

There are two commands, `any` and `all`, that reduce arrays of 1's and 0's (as above) to vectors of 1's and 0's, and reduce vectors of 1's and 0's to a single 1 or 0. `any` returns 1 if an input vector contains any nonzero elements (an *or* operation, in other words). If the input is an array, each column of the array is checked. The output is then a row vector containing the results from each column. `all` operates in a similar fashion, except that 1 is returned only if all elements of the input vector are nonzero (an *and* operation). If the input is an array, then each column is checked independently, and the results are returned in a row vector.

In the above example, `any(A<=B)` would return the 1x3 vector,

1	1	0
---	---	---

while `all(A<=B)` would return the vector,

0	1	0
---	---	---

To further condense these results, `any(any(A<=B))` yields 1, while `all(all(A<=B))` returns 0.

The `find` command can be used to determine which elements, if any, of a particular matrix satisfy a given condition. For example, suppose we wanted to know which elements of A, from above, were equal to 4. `[X,Y]=find(A==4)` would return the arrays X and Y, which contain the row and column indices of any elements of A which happened to be 4 (X=1, Y=3 in this case). Any valid logical expression, or combination of logical expressions, can be entered as the `find` criterion.

Some notes:

1. As mentioned above, results of property checks are associated with a logical value (0 or 1). When these checks are made in conjunction with an `if` statement, a common approach is to check this numerical value:

```
if isnan(b) == 0,
    c=b/2;
end
```

There is nothing strictly illegal about this statement; it is, however, just a little redundant. Since `isnan(b) == 0` is true when `isnan(b)` is false, the same result can be obtained by writing

```
if ~isnan(b)
    c=b/2;
end
```

2. `elseif` is one word, not two. `elseif` branches do not require their own `end` statements.
3. `ifs` may be nested. The result is similar to combining `ifs` with an `&`, but may be more useful in some circumstances. Each nested `if` may have its own `elseifs`, and must have its own `end`.

2.2 Using m-files: Scripts and Functions

Any sequence of MATLAB commands may be written into an ASCII file. This file must be given a `.m` extension; the command sequence can then be executed by entering the name of the file. Many MATLAB functions, such as `pinv` or `rank`, are actually m-files.

There are two primary classifications of m-files: scripts, and functions. The main difference between the two classes is in how they treat variables. Scripts are executed at the command level, and have access to any variable currently in the workspace. Functions are executed below the command level and can only access a limited subset of the variables in the current workspace.

One advantage of using scripts over using functions is just that: scripts can access any variable currently in memory, while any variable needed by a function must be passed in as an argument, or included in the global variable list. However, if a command sequence contains many calculations, with lots of intermediate variables, writing the sequence as a script and executing it at the command level will clutter up the workspace with all of the intermediate results. If all that is required is the end result of the calculations, writing the command sequence as a function will keep the workspace much cleaner.

For example, suppose the $n \times n$ arrays `A` and `B` are currently in memory. Now suppose that we want to know what happens when we pre- and post-multiply `A` by `B`'s eigenvectors, and then pre- and post-multiply `B` by the eigenvectors of that result. We can do that with the following script:

```
% Script to multiply A by B's eigenvectors
% then multiply B by the eigenvectors of the
% result.
% A and B are square matrices

[Evec, Eval]=eig(B);
C=Evec' * A * Evec;

[Cvec, Cval]=eig(C);
D=Cvec' * B * Cvec;
```

A look at the current variable list (enter who or whos) now shows that, in addition to A, B, and D, we have the intermediate results C, Evec, Eval, Cvec, and Cval residing in memory. If the commands are programmed as a function, however, the variable list can be kept to a minimum:

```
function D=funtest(A,B)
% Function to multiply A by B's eigenvectors,
% then multiply B by the eigenvectors of the
% result.
% A, B, and D are square matrices

[Evec,Eval]=eig(B);
C=Evec' * A * Evec;

[Cvec,Cval]=eig(C);
D=Cvec' * B * Cvec;
```

Notice that the only difference between this function and the preceding script is the first line: all functions must begin with a function declaration. Also, notice that there is no specific end-of-file marker. An m-file (whether script or function) simply stops running when it runs out of commands or finds an error.

The syntax of the function statement shows the calling syntax of the function: to execute the above function, the proper command is

```
D=funtest(A,B);
```

A look at the variable list now shows only A, B, and D in memory.

Argument Lists

Functions may have any number of input or output arguments. Input arguments are the arrays contained in parentheses after the name of the function. Output arguments are requested to the left of the equal sign. If there are multiple output arguments, they must be contained in square brackets (as in the eig call above).

Input arguments do not need to be definite arrays; any valid MATLAB expression, numeric or string, should be acceptable as an argument. For instance, the function funtest could be written as:

```
function D=funtest(A,B)
% Function to multiply A by B's eigenvectors,
% then multiply B by the eigenvectors of the
% result.
% A, B, and D are square matrices

[Evec,Eval]=eig(B);
[Cvec,Cval]=eig(Evec' * A * Evec);
D=Cvec' * B * Cvec;
```

where the first product, $C=Evec' * A * Evec$, is given as the input argument for the second eig call.

It is possible to request fewer return variables from a function than the number shown in the function's output argument list. It is also possible to supply fewer input arguments than the function thinks it needs. These two conditions may be checked for by using the nargin (number of arguments in) and nargsout (number of arguments out) parameters. For example, the function funtest may be rewritten to provide the Cvec and Cval arrays in addition to the end result:

```
function [D,Cvec,Cval]=funtest(A,B)
% Function to multiply A by B's eigenvectors,
% then multiply B by the eigenvectors of the
% result.
% A and B are square matrices
```

```

if nargin == 1,
    B = eye(size(A));
end

[Evec, Eval]=eig(B);
[Cvec, Cval]=eig(Evec' * A * Evec);
D=Cvec' * B * Cvec;

if nargout == 2,
    Cvec=Cval;
end

```

Now, if only the end result of the calculation is required, the function may be invoked as

```
D=funtest(A,B);
```

If, however, the end result and the eigenvalues of the intermediate array (Cval) are required, the function call becomes

```
[D,Cval]=funtest(A,B);
```

Notice that the nargout check at the end of the function reverses the output arguments Cvec and Cval in this case.

If all three return variables are desired, the function call becomes

```
[D,Cvec,Cval]=funtest(A,B);
```

Finally, notice that, if only one matrix is input, B is set to be the identity matrix of the same size as the input matrix.

Stopping Functions

A function can be interrupted without causing an error by the `return` command. Alternatively, the error command will interrupt the function and display an error message in the command window. For example, `funtest` could be written as:

```

function [D,Cvec,Cval]=funtest(A,B)
% Function to multiply A by B's eigenvectors,
% then multiply B by the eigenvectors of the
% result.
% A and B are square matrices

if nargin == 1,
    disp('Must have two input arguments');
    return
end

[Evec, Eval]=eig(B);
[Cvec, Cval]=eig(Evec' * A * Evec);
D=Cvec' * B * Cvec;

if nargout == 2,
    Cvec=Cval;
end

```

Then, if only one input argument is given in the function call, the message “Must have two input arguments” will be shown on the screen, and the function will stop. Or, `funtest` may be written as

```
function [D,Cvec,Cval]=funtest(A,B)
```

```

% Function to multiply A by B's eigenvectors,
% then multiply B by the eigenvectors of the
% result.
% A and B are square matrices

if nargin == 1,
    error('Needs two input arguments');
end

[Evec, Eval]=eig(B);
[Cvec, Cval]=eig(Evec' * A * Evec);
D=Cvec' * B * Cvec;

if nargin == 2,
    Cvec=Cval;
end

```

In this case, if only one input argument is given, MATLAB will sound a beep, display the name of the function in which the error occurred, and display the message, “Needs two input arguments”. This could be useful if several functions are nested, and it wouldn’t otherwise be immediately obvious where the error was occurring.

The primary difference between this and the previous example is that, if `funtest` was called from another m-file rather than from the keyboard, the `return` statement would allow the calling m-file to continue running; the `error` statement stops everything that is currently running.

Suspending Functions

A function may be temporarily suspended by use of the `keyboard` command. When a keyboard is encountered, function execution stops and keyboard input is allowed. The double-arrow prompt (`>>`) changes to the keyboard prompt (`K>>`) to indicate that the function is only temporarily stopped.

Any input that is allowed at the normal prompt is allowed at the keyboard prompt. One of the main uses of the `keyboard` statement is program debugging; commands entered at the keyboard prompt can access only the information that would normally be available to the function which called the `keyboard`, so this is a useful means of ensuring that all necessary variables are available, and of checking the syntax of lines that use variables local to the function.

Entering `return` at the keyboard prompt resumes execution of the function.

Global Variables

In addition to input arguments, functions have access to global variables. Global variables are stored separately from other variables; they must be declared `global` at the command level (through a keyboard command or in a script), and must be declared `global` by any function that uses them. Only functions that declare a variable to be `global` may use that `global` variable. Any function that changes a `global` variable changes it for every function that uses it.

An important note: **DO NOT** pass `global` variables to a function as either input or output arguments.

The following m-files provide a brief example of usage of `global` variables:

The first file runs at the command level:

```

global TEST

TEST=[];

glob2
disp(TEST)

```

```
glob3
disp(TEST)
```

The second file runs at the function level:

```
function glob2
global TEST
disp(TEST)
TEST=3
```

The third file also runs at the function level:

```
function glob3
TEST=2;
disp(TEST)
```

The first file declares `TEST` to be a global variable and initializes it as an empty array. `glob2` then accesses `TEST`, displays its current value, and changes it. The next line of the command-level script displays the new value of `TEST` and calls `glob3`. Because `glob3` does not declare `TEST` a global variable, however, the change to `TEST` is strictly local, and the value of `TEST` stays at 3 at the command level.

It is good practice to initialize global variables immediately after first declaring them. In many cases, as above, simply setting them equal to an empty array will do nicely.

A popular convention, when naming global variables, is to name them all in capital letters (as `TEST`, above). Other variables are then named in all lower case letters, except when two or more words are combined to form a name. In that case, the initial letter of every word after the first is capitalized. For example, `EIGENVALUES` might be a global variable containing a set of eigenvalues, while `singVals` might be a local variable containing a set of singular values. (For the record, this document's author (that's me) has never fully subscribed to this convention.)

2.3 Comments

Although some of the commands discussed in the preceding sections (`if` or `keyboard`, for example) are probably more useful when programmed into `m`-files, virtually every MATLAB command can be used directly from the keyboard. And, though `m`-files are part of what makes MATLAB powerful, many tasks (such as labelling a graph) are easier to accomplish interactively, especially if they are only going to be done a few times.

For most applications, MATLAB is case-sensitive. A few exceptions are the `inf` and `nan` values. Both `inf` and `Inf` refer to infinity; `nan` and `NaN` both refer to the not-a-number value. Another application in which case-insensitivity occurs is in setting property values of graphical objects (see Section 3.1).

Functions are compiled by MATLAB when a program begins execution. A function can be changed while a program is running (at a `keyboard` prompt, for instance), but the changes are not registered until the program ends. Entering `clear name`, where `name` is the name of the edited function, removes that function from memory and forces MATLAB to recompile it the next time it is called.

3.0 FUN WITH FIGURES

Graphic output is obtained from MATLAB by use of figure windows. In the Handle Graphics™ philosophy, every aspect of a figure is given a numeric label, called a *handle*. Although it is possible to produce a plot without making use of these handles, they can be a powerful tool in producing and labelling graphs. They can, in fact, allow any graph to be tailored to almost any format.

The simplest method of obtaining a graph is to enter `plot(y)`. This command graphs the values of y (y -axis) vs their position in the array (x -axis) in the current figure window. Any previous graph in the window is deleted. If no figure window was open before the command was issued, one will be created. (Additional figure windows may be opened by entering `figure`; an existing figure window may be made current by entering `figure(n)`, where n is the number of the figure to make current; `gcf` is used to determine which figure window is current.)

In this example, y does not need to be a pre-defined vector. It can be any valid MATLAB expression resulting in a rectangular numeric array. If there is more than one column in y , each column will be plotted as a separate curve. For example, `plot(rand(3,8))` produces 8 curves of 3 random data points each.

If an independent x -axis is available for the graph, the `plot` command can take the form, `plot(x,y)`. If x is a vector of length n , then at least one of y 's dimensions must also be n . Each vector of y (either row or column, whichever is length n) is then plotted vs x . (If y is $n \times n$, then each column of y is plotted vs x .) If x is an $m \times n$ array, then y must either be an $m \times n$ array (so each column of y is plotted vs each column of x), or a vector (so y is plotted vs each column of x).

By default, MATLAB plots each curve as a sequence of straight lines connecting the data points given by x and y (or just y). If more than one curve is plotted, each curve (up to 8) is given a different color (beyond 8, the colors repeat). Linestyles and colors may be set from the command line, using the expanded `plot` syntax, `plot(x,y,sc)`. sc in this form of the command is a string which contains the color preference, the linestyle choice, or both. For a list of color and linestyle options, see `help plot`.

Three-dimensional plots are created by using the `plot3` command. Syntax is the same as for `plot`, except that three numeric input arguments are required (for the x -, y -, and z -axes).

Subplots

Use of the `subplot` command allows a single figure window to contain multiple graphs. Entering `subplot(m,n,p)` divides the window into an $m \times n$ grid, making the p th grid current. (m and n must not be greater than 9; p is found by counting across the rows, starting at the top.) Each grid element has its own set of axes and may therefore be treated as a separate figure.

Plots created by `subplot` do not need to be all of one size. The following set of commands will create an irregular grid of 7 plot axes:

```
subplot(3,3,1)
subplot(3,3,2)
subplot(3,3,3)
subplot(3,1,2)
subplot(3,4,9)
subplot(3,4,10)
subplot(3,2,6)
```

The same command that creates a particular grid element can be used to make that element current. If, once all seven of these plots have been generated, the third one needs to be edited in some way, `subplot(3,3,3)` will make that one current without destroying any of its settings.

Plot Formats

By default, `plot` produces linear axes for all displayed data. Log scales may be created for the x -axis, the y -axis, or both, by the alternate commands `semilogx`, `semilogy`, and `loglog`. Each of these commands replaces the `plot` command; syntax and options for these three commands are identical to the syntax and options for `plot`.

Scaling Plots

There are two methods for scaling the axes on a given plot. The first, `axis`, is perhaps the easier of the two to use. Entering `axis(scale)`, where `scale` is the 4-element vector $[X_{\min} \ X_{\max} \ Y_{\min} \ Y_{\max}]$, freezes the current axis at the specified limits. Entering `axis(axis)` (no quotes) freezes the current axis at its current limits. Several string options can also be applied to `axis`, to set the axes aspect ratio, reverse the y-axis (for inspecting matrices, etc), and perform a few other miscellaneous functions.

The second scaling method involves the `set` command and the axes properties `XLim` and `YLim` (discussed in Section 3.1). If the handle of the axes is known, then the axes limits can be set by entering

```
set(handle, 'XLim', [x_min x_max], 'YLim', [y_min y_max])
```

If only the x-axis is to be changed, then the `YLim` property/value pair may be left out; if only the y-axis is to be changed, the `XLim` property/value pair may be left out.

For 3-D plots, the z-axis can be frozen by setting the `ZLim` property the same way.

The handle of the current graph axes may always be found by entering `gca`.

Labelling Plots

Plots can be labelled with the `title`, `xlabel`, and `ylabel` (and `zlabel`, for 3-D plots) commands. These place their string input arguments in the appropriate locations on the plot: `title` places its string above the plot, `xlabel` places its string below the x-axis, `ylabel` places its string to the left of the y-axis.

Arbitrary labels may be placed on the plot using the `text` command. In its simplest form, `text(x,y,string)` places `string` at the location specified by the `x`, `y` coordinates. `string` appears in white text and is aligned so that the lower left-hand corner is located at `x`, `y`. Color and alignment are among the properties that may be specified on the command line in the expanded form, `text(x,y,string,Property,value)`. Color may be changed by setting the property `Color`; alignment may be changed by setting `HorizontalAlignment` and/or `VerticalAlignment`. Detailed information on setting text properties is contained in Section 3.1; the command to place a yellow text object horizontally centered on `x`, `y` is `text(x,y,string,'Color',[1 1 0], 'Horizontal','center')`.

If an output argument is requested, `text` returns the handle of the text object. This handle can then be used to alter properties after the string has been placed on the figure. Again, see Section 3.1 for more information.

Labelling Plots Interactively

Text can be placed on figures interactively, using the command `gtext`. `gtext(string)` waits for a mouse button click somewhere in the current figure window, then places `string` at the location of the click. `string` will be aligned such that the lower left-hand corner of the text object is at the location of the click.

If `string` contains multiple rows, `gtext` treats each row as a separate label. A click is then required to place each row of `string` individually.

Like `text`, `gtext` can return the handles of its text objects (one handle per row of `string`), if requested. These can be used for changing text color, alignment, and position (since `gtext`'s cursor isn't incredibly precise), among other things.

Figure 1 shows a figure produced and labelled within MATLAB. This figure was generated by issuing the command `plot(x,y)`, with `t=[0:5*360-1].*pi/180`, `x=t.*sin(t)`, and `y=t.*cos(2*t)`. The text object was placed using `gtext`, and could be adjusted using `set`, if necessary. The axis labels and the title were placed by `xlabel`, `ylabel`, and `title`, respectively.

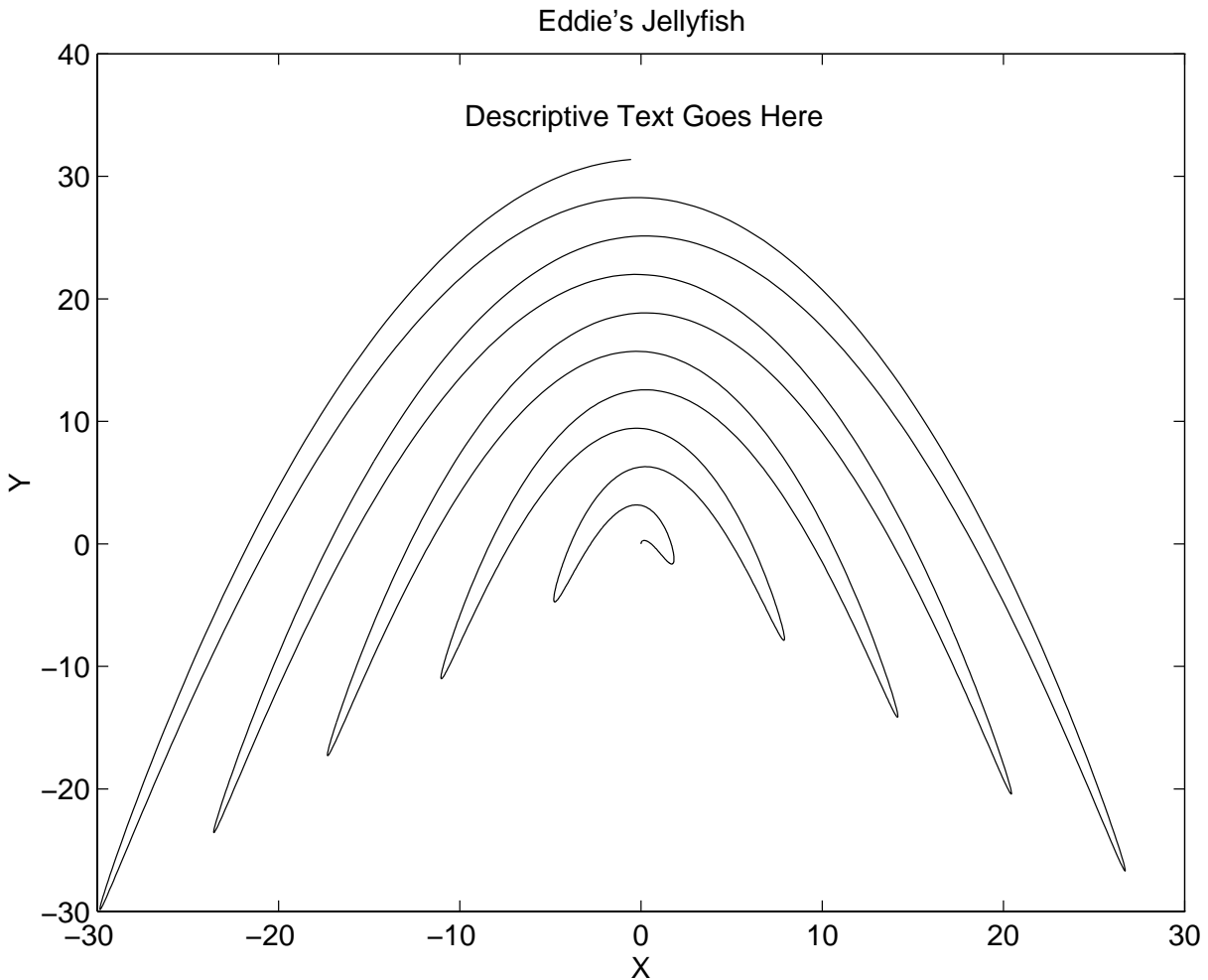


Figure 1: A MATLAB Figure.

Adding to Plots

Graphing data by the `plot` command destroys any data that was previously contained in the current figure window. It is possible, through use of the `line` command, to add data to an existing graph without destroying anything.

`line` syntax is very similar to `text` syntax; both are essentially graphic objects, and can be manipulated by setting object properties. `line(x,y)` or `line(x,y,z)` adds the line specified by the `x` and `y` (or `x`, `y`, and `z`) vectors to the existing plot. `line(x,y,'Color',[0 1 1],'LineStyle','+')` adds the line specified by `x` and `y` to the existing plot, marked by cyan plusses.

Another method of adding to a plot without affecting existing data is to use the `hold` command. `hold`, by itself, toggles the current hold state. `hold on` sets the `NextPlot` property of the current figure and current axes to *add*, so that future plots are added to the current one. `hold off` sets the figure and axes `NextPlot` properties to *replace*, so that future plots replace the current one. The function `ishold` can be used to determine whether the current hold state is on (1) or off (0).

3.1 Handle Graphics

In the Handle Graphics philosophy, almost every aspect of a figure is controlled by a set of properties. These properties can be read or changed by the commands `set` and `get`. The syntax for these commands is

```
get(handle,property)
set(handle,property,value)
```

`property` is always a string, and must be enclosed in single quotation marks. `value` is sometimes a string and sometimes a numeric array, depending upon the needs of the specific property.

A word about handles: they're ugly. It is good practice to always assign handles to variables, then refer to them by those variable names.

For a complete list of properties for a particular object, enter `get(handle)`, where `handle` is the handle of the object in question. For example, if `q` is the handle of an existing line object, the command `get(q)` might result in,

```
Color = [1 1 0]
EraseMode = normal
LineStyle = -
LineWidth = [0.5]
MarkerSize = [6]
Xdata = [0.13 0.28 0.28 0.13 0.13]
Ydata = [0.69 0.69 0.37 0.37 0.69]
Zdata = []

ButtonDownFcn =
Children = []
Clipping = on
Interruptible = no
Parent = [49.0006]
Type = line
UserData = []
Visible = on
```

```
ans =
```

```
[]
```

For a complete list of properties, and legal settings, enter `set(handle)`. For the above line object, `set(q)` would yield,

```
Color
EraseMode = [{normal}|background|xor|none]
LineStyle = [{-}|--|:|-.|+|o|*|.|x]
LineWidth
MarkerSize
Xdata
Ydata
Zdata

ButtonDownFcn
Clipping = [{on}|off]
Interruptible = [{no}|yes]
Parent
UserData
Visible = [{on}|off]
```

Options enclosed in brackets ({}) are the current values of that property.

3.1.1 Object Properties

All (or most) of the properties for specific objects are explained in the MATLAB reference manual. A brief description of some of the more useful properties of figures, axes, lines, and texts is given in Appendix B as an introduction to the wonderful world of Handle Graphics.

When setting properties and values, the property name need not be capitalized exactly as it is shown in Appendix B. Also, the property name need not be spelled out completely; however, enough of the property name must be spelled to ensure that there is no ambiguity about which property is being set. For example, when finding the figure property `Position`, `get(gcf, 'Po')` would be illegal (because it could be `Position` or `Pointer`), while `get(gcf, 'pos')` would be perfectly legal.

Once again, the lists in Appendix B are NOT complete; for the complete lists see the MATLAB reference manual.

4.0: WHAT NOW?

It is not possible, even in 39 pages, to cover every aspect of an environment as complex as MATLAB. This document has attempted to provide a brief introduction to the joys of working in this environment, but much has naturally been left out. Of the commands that are described here, many are not fully explained. This is partly by design; probably the best way to really learn MATLAB is to play with it and see what it can or cannot do. In fact, a large portion of what is contained in this document was learned by experience, or by direct experimentation.

There are certain features built into MATLAB to make the task of exploring a bit less intimidating. `demo`, for instance, provides some interesting demonstrations of MATLAB's graphics and data analysis capabilities. And, the `help` feature is always available to provide more information about any aspect of the MATLAB environment.

Help is available on any MATLAB function, whether built-in or written as an m-file. For m-files, help displays any comment lines that are located at the beginning of the file. `help`, by itself, provides a list of directories in MATLAB's search path. Asking for help on one of these directories will yield a list of the m-files in that directory.

In addition to this, certain commands can aid in determining which function to use for a given task. `lookfor` scans the help text of every m-file in the search path and echos the filenames of every file in which it finds a specified string. `which` displays the pathname of a specified m-file, if it exists in the search path. `what` lists all m-files, mat-files, and mex-files (not discussed here) in a specified directory. If no directory is specified, all directories in the search path are listed.

APPENDIX A: A BRIEF CATALOGUE OF FUNCTIONS

A Glossary of Useful MATLAB Commands

Operators and Punctuation

=	Assignment. Gives the variable named on the left the value specified on the right.
+	Addition. Easy stuff.
-	Subtraction. No problem.
*	Matrix multiplication.
.*	Array multiplication (not the same!). $A.*B$ is the (i,j)th element of A times the (i,j)th element of B.
/	Matrix division. A/B is (almost) the same as $A*inv(B)$.
./	Array division. $A./B$ is the (i,j)th element of A divided by the (i,j)th element of B.
\	Matrix left-division. $A\B$ is (almost) the same as $inv(A)*B$.
.\	Array left-division. $A.\B$ is the (i,j)th element of A divided into the (i,j)th element of B.
^	Matrix power operator. Either the base or the exponent (but not both) has to be square. The other has to be a scalar.
.^	Array power. $A.^B$ is the (i,j)th element of A raised to the (i,j)th element of B. If A is a scalar, then it is A raised to the (i,j)th element of B. If B is a scalar, then it is the (i,j)th element of A raised to the Bth power.
:	Builds vectors. $1:3:13$ is a compact way of writing $[1\ 4\ 7\ 10\ 13]$. If the increment is omitted, it defaults to 1.
;	Suppresses echo to screen. Nice thing to know... Also separates rows of a matrix. Another nice thing to know...
'	Conjugate (hermitian) transpose.
.'	Transpose.
!	Execute system command.
%	Comment (ignore rest of line)

Some Useful Scalars

Inf	infinity.
NaN	not-a-number
i	$\sqrt{-1}$
j	$\sqrt{-1}$
eps	machine epsilon (a very small number)
flops	flop count. flops(n) sets flop count to n.

Some Useful Matrix Functions

diag	If input is a vector, returns a diagonal matrix with input along primary diagonal. if input is a matrix, returns a vector of the elements along the primary diagonal.
rand	Matrix of random numbers.
eye	Identity matrix.
ones	Matrix of 1's.
zeros	Matrix of 0's.
inv	Inverse of a matrix.
det	Determinant of a matrix.
trace	Trace of a matrix.
svd	Singular value decomposition of a matrix.
rank	Rank of a matrix.
cond	Condition number of a matrix.
pinv	Pseudo-inverse of a matrix (from svd).
eig	Eigenvalues and eigenvectors of a matrix.
poly	If input is a vector, returns coefficients of polynomial whose roots were the input vector. If input is an $n \times n$ matrix, returns a vector whose elements are the coefficients of the characteristic polynomial.
sum	Sum of elements of a vector. If input is a matrix, returns row vector of sums of each column.

prod	Product of elements of a vector. If input is a matrix, returns row vector of products of each column.
abs	Absolute value of real numbers. Magnitude of complex numbers. ASCII codes of strings.
min	Minimum value of a vector. If input is a matrix, returns row vector of minimum values of each column.
max	Maximum value of a vector. If input is a matrix, returns row vector of maximum values of each column.
norm	Norm of vectors or matrices. Default is 2-norm, but others may be specified.
chol	Cholesky factorization of symmetric positive definite matrix.
qr	QR decomposition of a matrix.
schur	Schur decomposition of a matrix.

Important Things We Can't Find a Good Category For

find	Find non-zero elements of a vector (or matrix). Good with comparisons.
any	True if its input vector has any nonzero elements.
all	True if its input vector has all nonzero elements.
sort	Sort a vector.
eval	Evaluate a MATLAB command that's hidden in a string variable.
...	Line continuation mark.

Logic and Loops

for	Loop at most a specified number of times.
while	Loop until a condition is met.
if	Starts an if structure. ifs may be nested.
else	For branching within an if structure.
elseif	The proper form of an else for multiple branches within a single if structure.

end	Closes an if structure or substructure. Must have one end for each nested if. elseifs do not require separate ends.
	Also closes for or while loops. Each nested for or while loop must have a separate end.
==	Requires exact match.
~=	Not equal to.
<	Less than.
<=	Less than or equal to.
>	Greater than.
>=	Greater than or equal to.
~	Not. Returns zero or one. ~0=1, ~1=0. Nonzero values are read as 1's.
&	And. Must satisfy all criteria.
	Or. Must satisfy at least one of criteria.
xor	Exclusive or. Must satisfy one criterion, but not both.

Graphing Functions

plot	Plot a graph.
plot3	Plot a graph in 3 dimensions.
semilogx	Plot a graph on log-x scale.
semilogy	Plot a graph on log-y scale.
loglog	Plot a graph on log-log scale.
axes	Produce an axes to graph on.
axis	Freeze axis scale.
hold	Freeze current plot.
line	Add a line to a graph.
text	Add a text object to a graph.
mesh	Draw a wire-mesh 3-D representation of a matrix.
waterfall	Draw a waterfall representation of a matrix.

<code>gplot</code>	Draw a graph—theoretic representation of a matrix.
<code>spy</code>	Display sparsity pattern of a matrix.
<code>image</code>	Create an image of a matrix.
<code>gtext</code>	Interactively place text on a graph.
<code>ginput</code>	Interactively read values from a graph.
<code>set</code>	Set properties of graphical objects.
<code>get</code>	Read properties of graphical objects.
<code>figure</code>	Open a new figure window, or make an existing figure window current.
<code>cla</code>	Clear current axes. Deletes all axes children (lines, texts, patches, etc).
<code>clf</code>	Clear current figure. Deletes all figure children (axes, uicontrols, etc). <code>clf(n)</code> clears figure number <code>n</code> .
<code>clg</code>	Old way of clearing figures. Use <code>clf</code> , instead.
<code>close</code>	Close a figure window.
<code>delete</code>	Delete a graphical object.
<code>uicontrol</code>	Build user interface object.
<code>uimenu</code>	Build user interface menubar.

File Input/Output Functions

<code>diary</code>	Echos screen display to a disk file. Default filename is 'diary'. Filename can be set by appending it to the command line: <code>diary filename</code> . Alternatively, <code>set(0, 'diaryfile', filename)</code> will change the active diary file.
<code>load</code>	Load a data file. If the file is in MATLAB (binary) format, all defined variables are loaded. If file is in ASCII format, data is assigned to an array whose name is the same as the filename. If no filename is given, the file <code>matlab.mat</code> is searched for.
<code>save</code>	Save a data file. Default is <code>matlab.mat</code> , in MATLAB (binary) format. ASCII format may be specified. Variables to save may be specified on the command line (default is everything).

<code>fopen</code>	Open a file for read/write.
<code>fclose</code>	Close a file.
<code>fwrite</code>	Write data to a specific file, in a user-defined format. (For binary files).
<code>fprintf</code>	Write data to a specific file, in a user-defined format (ASCII, as near as we can tell).
<code>fread</code>	Read data from a specified file, in a user-defined format.
<code>fscanf</code>	Read data from a specified ascii file, in a user-defined format.
<code>ftell</code>	Determine current pointer location within a file.
<code>fseek</code>	Move pointer within a file.

Screen/Keyboard Input/Output

<code>input</code>	Input an array (scalar, vector or matrix, number or string).
<code>disp</code>	Display (on the screen) the contents of an array.
<code>format</code>	Change display format.

Miscellaneous

<code>clc</code>	Clear the command window.
<code>clear</code>	Remove variable from workspace. When applied to a function name, forces MATLAB to recompile that function the next time it's called.
<code>demo</code>	Demonstration of some of MATLAB's capabilities.
<code>lookfor</code>	Find all m-files whose help text contains a specified keyword.
<code>puzzle</code>	Something to do to look busy without accomplishing anything serious.
<code>tic</code>	Place marker on clock.
<code>toc</code>	Read time elapsed (in seconds) since last <code>tic</code> .
<code>what</code>	List m-files, mat-files, and mex-files.
<code>which</code>	Display pathname of m-files and mex-files.

who	List variables currently in memory.
whos	Display information about memory usage.
why	Succinct answers and/or inspirational guidance.

APPENDIX B: SOME HANDLE GRAPHICS PROPERTIES

Figure Properties

<code>gcf</code>	Get Current Figure. Command to return handle of the current figure window.
<code>Color</code>	Color of figure background. Color is specified by a three-element vector, indicating the fractions of red, green, and blue, respectively, that appear in the color. Some examples are [0 0 0] (black) [1 0 0] (red) [0 1 0] (green) [0 0 1] (blue) [1 1 1] (white) [1 0.5 0] (orange)
<code>CurrentPoint</code>	Location of last mouse-button click. First value is measured from left edge of window, second value measured from bottom, in the figure's current units.
<code>Name</code>	Name of the window. Name appears in the window frame, not in the figure.
<code>NextPlot</code>	Action taken on next <code>plot</code> command.
<code>NumberTitle</code>	Display or suppress "Figure No." in window frame.
<code>Pointer</code>	Shape of mouse pointer when over figure window.
<code>Position</code>	Location of figure window on screen, in figure's current units. Values are measured [from left, from bottom, horizontal length, vertical length].
<code>SelectionType</code>	Type of mouse click last registered. Types are normal (button 1), extend (button 2), alt (button 3), and open (double click, any button).
<code>Units</code>	Units in which to measure <code>Position</code> and <code>CurrentPoint</code> .
<code>WindowButtonDownFcn</code>	Command to execute whenever a mouse button is pressed within the figure window.
<code>WindowButtonMotionFcn</code>	Command to execute whenever the mouse is moved inside the figure window.
<code>WindowButtonUpFcn</code>	Command to execute whenever a mouse button is released inside the figure window.

ButtonDownFcn	Command to execute whenever a mouse button is pressed within the figure window, but outside the current axes. ButtonDownFcn is executed after WindowButtonDownFcn.
Children	Objects belonging to the figure. Children can be axes, uicontrols, or uimenu.
UserData	A handy pocket for stashing important information in.
Visible	Figure window visibility.

Axes Properties

Every property that applies to the x-axis has a y-axis and a z-axis counterpart. The y- and z-axis properties are omitted for brevity.

gca	Get Current Axes. Command to return handle of the current graph axes.
Box	Draw or not draw a box around the figure.
CurrentPoint	Location of last mouse-button click, measured from the origin of the current axes. The first column is the x value, the second column is y, the third column is z. CurrentPoint actually returns the endpoints of a line drawn through a three-dimensional figure, at the location of the mouse click.
GridLineStyle	LineStyle in which to draw gridlines.
NextPlot	Action taken on next plot command.
LineWidth	How big to draw axes borders.
Position	Location of axes within figure window, in axes' current units. Values are measured [from left, from bottom, horizontal length, vertical length].
TickLength	How long to draw tickmarks.
TickDir	Draw ticks on inside or outside of axes box.
Title	Handle of text object containing axes title. (Title's string property is set using title command.)
Units	Units in which to measure Position and CurrentPoint.
View	Location of point from which origin is viewed: [Angle from z-axis, angle from xz-plane].
XColor	Color in which to draw the x-axis.

XDir	Draw x-axis right-to-left or left-to-right.
XGrid	Draw grid lines on x-axis.
XLabel	Handle of text object containing x-axis label. (XLabel's <code>string</code> property is set when <code>xlabel</code> command is used.)
XLim	Limits of x-axis display.
XLimMode	Set x-axis limits manually or automatically.
XScale	Linear or log scale for x-axis.
XTick	Where to place tick marks on x-axis.
XTickLabels	What to call tick marks on x-axis.
XTickLabelMode	Label XTicks manually or automatically.
XTickMode	Set XTicks manually or automatically.
ButtonDownFcn	Command to execute whenever a mouse button is pressed on the current axes. <code>ButtonDownFcn</code> is executed after the current figure's <code>WindowButtonDownFcn</code> .
Children	Objects belonging to the axes. Children can be line, surface, patch, or text objects.
Parent	Figure to which axes belongs.
UserData	A handy pocket for stashing important information in.
Visible	axes visibility.

Line Properties

Color	Color of line object.
LineStyle	How to display data (solid line, dashed line, x-mark, plus, etc).
LineWidth	How big to draw the line.
MarkerSize	How big to mark the data points.
XData	Values plotted on the x-axis.
YData	Values plotted on the y-axis.
ZData	Values plotted on the z-axis (if any).

ButtonDownFcn	Command to execute whenever a mouse button is pressed on the line object. ButtonDownFcn is executed after the current figure's WindowButtonDownFcn.
Children	Objects belonging to the line.
Parent	axes to which the line object belongs.
UserData	A handy pocket for stashing important information in.
Visible	Line object visibility.

Text Properties

Color	Color of text object.
Extent	How long the text object is, measured in the object's current units.
FontAngle	Straight or slanted text.
FontSize	How big to write the text.
FontWeight	How heavily to write the text.
HorizontalAlignment	How to align the text on the specified x-value.
Position	Where to write the text. Position is [right from y-axis, up from x-axis], in text's current units.
Rotation	Counterclockwise angle through which to rotate text (measured in degrees).
String	What to write.
Units	Units in which to measure position and extent.
VerticalAlignment	How to align the text on the specified y-value.
ButtonDownFcn	Command to execute whenever a mouse button is pressed on the text object. ButtonDownFcn is executed after the current figure's WindowButtonDownFcn.
Children	Objects belonging to the text.
Parent	axes to which the text object belongs.

UserData	A handy pocket for stashing important information in.
Visible	Text object visibility.

EXERCISES

1. Using the `input` function, develop another function to use the same syntax, but to wait for a non-empty input.
2. Write a function to calculate (using `eig`) the eigenvalues and eigenvectors of a square matrix, then sort them by the magnitude of the eigenvalues. Include the sort order (descending or ascending) as an input argument, defaulting to ascending.

FURTHER READING

MATLAB Reference Guide, The MathWorks, Inc, 1992.

MATLAB Primer, Kermit Sigmon, University of Florida, 1989.

MATLAB is a registered trademark, and Handle Graphics is a trademark, of The MathWorks, Inc.,

More information about MATLAB, Handle Graphics, and other products is available from

The MathWorks, Inc.
Cochituate Place
24 Prime Park Way
Natick, Mass. 01760.

(508) 653-1415 Fax: (508) 653-2997

email: info@mathworks.com (general information)
tech@mathworks.com (technical support)