Introduction

MATLAB is a mathematical software package which is especially useful for signal processing applications because of its exclusive use of vectors and matrices. In fact, 'MATLAB' itself apparently stands for Matrix Laboratory. Start it up on the LRC machines by simply typing `matlab` at the UNIX prompt. MATLAB will fire up and offer you the command prompt. MATLAB accepts input directly from the keyboard, and from files with the extension `.m`, called script files or functions. For the moment, we will enter commands directly.

Some basics

We can define a scalar:

```
a=1
```

and MATLAB responds with (spaced a little differently)

```
a = 1
```

It can be annoying to have MATLAB repeat everything like this, so instead you can type

```
a=1;
```

which suppresses the output. Now let's define a matrix instead:

```
a=magic(3);
```

which produces a magic square of size 3×3. Type `a` at the prompt to have a look at it. Here is another matrix:

```
b=rand(3);
```

which gives a 3×3 matrix of numbers chosen randomly between 0 and 1. Now we can add these matrices:

```
a+b
```

or do a matrix multiply:

```
a*b
```

or do an element-by-element multiply:

```
a.*b
```

(notice the period). Verify that the answers are what you expect. Notice that addition and multiplication proceed over the whole matrix, with no need to individually access each element (by a `for` loop, for instance). This is called vectorized code, and is the most powerful feature of MATLAB.

In general, functions in MATLAB expect matrix inputs, that is, it is possible to operate on a whole set of data at once with the function. Some functions, like *, are ambiguous, in that they have a genuine matrix interpretation as well as an element-by-element interpretation. Such functions use the period form, shown above, to denote the element-by-element version. Most functions, however, do not make sense matrix-wise and therefore they automatically operate on each element individually. For instance, try typing

```
ang=[pi/2 pi/4; pi pi/3];
sin(ang)
```

This returns the sine of every element in the matrix `ang`. Notice that matrices are defined in square brackets, with a semicolon separating rows.

MATLAB deals inherently with complex numbers of double precision, although it understands when a matrix is pure real and does not store a complex part. (Note: MATLAB 5 also allows other data types.) The use of complex numbers is seamless; try `sqrt([3 -3])` for instance. You can find the state of your variables by typing `whos`; this will give you a listing of the existing variables, what size they are, and so on. If you are running out of space, you can use `clear <variable name>` to remove a variable, but beware that MATLAB does not perform garbage collection; this can leave unused holes in memory. To recover these holes, use `pack`.

Defining common vectors

Commonly one wants an index vector (of time, for instance), defined as integers from a lower to an upper limit. These can be defined easily using the colon notation:

```
t=1:100;
```

for instance, defines `t` to be a vector of integers from 1 to 100. The spacing of elements need not be one:

```
u=1:0.1:2;
```

produces an eleven-element vector between 1 and 2 spaced every 0.1. Sometimes one knows the limits of the range and the number of elements required; instead of calculating the inter-element spacing, use the form

```
v=linspace(1,2,11);
```

which produces a vector `v` equal to vector `u` defined previously. `logspace` returns a vector whose elements are spaced by a multiplicative, rather than an additive, factor. To define a vector, or matrix, of all zeros use:

```
a=zeros(4,5);
```

which defines a matrix of size 4 rows by 5 columns of all zeros. Use a 1 for one of the arguments to define a row, or column, vector. Similarly, `ones(x,y)` produces a matrix of all ones. Multiplying this matrix by two gives a matrix of all twos, and so on.

Useful matrix operations

We have already met the matrix operations + and *, and the element-by-element operation `.*`. Since MATLAB deals naturally with vectors and matrices, it has a rich set of manipulation operations on these data types. Some are given here.

Operator	Effect
`a'`	Conjugate transpose
`a.'`	Transpose
`a(:)`	Write out matrix as a vector
`inv(a), det(a)`	Matrix inverse and determinant (square matrices only)
`a+scalar`	Add scalar to each element of `a` (also `-`, `*` and `/`)
`a./b`	Element-by element division of `a` and `b`
`a/b`	Pseudo-inverse (solution of simultaneous equations)
`fft(a)`	Fast Fourier transform (also `ifft(a)`, the inverse)
`abs(a), angle(a)`	Magnitude and phase
`real(a)`	Real part (also `imag(a)`, imaginary part)
`conj(a)`	Complex conjugate
`fftshift(a)`	Flip two halves of vector (or four quarters of matrix)

Indexing matrices

Sometimes one wants to operate on part of a vector or matrix, rather than the whole thing. MATLAB allows you to access parts of a vector or matrix easily without using a `for` loop. Generate a vector:

```
v=linspace(0,1,21);
```

and take part of it:

```
v_part=v(10:15);
```

This extracts elements 10 through 15 of `v`. (Note that MATLAB indexes from 1, not 0.) Similarly, a rectangular area of a matrix can be extracted:

```
m_part=m(3:7,8:20);
```

extracts the rectangular region from (and including) rows 3 to 7 and columns 8 to 20. To extract a list of elements from a vector, define a vector containing the index of the elements you want:

```
wanted_elements=[3 5 9];
```

and then index the data vector using this list vector:

```
elements=data_vector(wanted_elements);
```

which returns the 3rd, 5th and 9th elements of `data_vector`. Unfortunately, this technique does not work with matrices; for instance, executing

```
data_matrix=rand(10,10);
row_list=[1 3 6];
```

```
col_list=[2 2 3];
wanted_elements=data_matrix(row_list, col_list);
```

does not return elements (1, 2), (3, 2) and (6, 3). To accomplish this, you need instead:

```
wanted_elements=data_matrix(size(data_matrix,1)*(col_list-1)+row_list);
```

which uses the vector indexing mode of a matrix. A matrix can be accessed as if it were a vector. For instance, if a matrix `a` has 5 rows, then `a(1)` is equivalent to `a(1,1)`, `a(2)` is equivalent to `a(2,1)` (that is, row 2, column 1), `a(6)` is equivalent to `a(1,2)` and so on. This is rarely useful but is, as far as I know, the only way to efficiently extract non-rectangular data from a matrix.

Plotting data

MATLAB offers a number of ways to visualize data. Let us first generate some data:

```
t=0:0.01:1                                  % time index (every 10 ms)
ws=2*pi*t/0.01;                             % sampling frequency
signal=sin(ws/32)+0.5*sin(ws/16);           % sine wave with a bit of interest
```

Just type `plot(signal)` to look at it. MATLAB selects a good range for the plot and displays it in a new window. The values on the x-axis are the index of elements in the `vector` signal; they do not correspond to the time index. To see the signal against time, type `plot(t, signal)`. The two vectors must be the same length. You can change the axes after plotting by using `axis([xmin xmax ymin ymax])` where `xmin...` are the values at the edges of the plot. Use `title 'string'`, `xlabel 'string'` and `ylabel 'string'` to label your plots. You can make multiple plots on the same axes by using `plot(x1,y1,x2,y2,...)`.

Sometimes I will ask you to turn in a page of several plots. You can do this using `subplot`, which divides the figure window into several independent parts, each of which can hold a plot. For instance, using the data defined above, try the following:

```
subplot(211)                                % top half of window
plot(t, signal); title 'Original signal'
subplot(212)                                % bottom half of window
plot(abs(fft(signal))); title 'FFT of signal'
```

The three arguments of `subplot` are the number of rows into which the window is divided, the number of columns, and the current active section (numbering from left to right, and from top to bottom). Each plot can be scaled and titled individually.

You can print a figure window in UNIX by typing `print -dps <filename>` to generate a PostScript file which can then be dumped to a printer with `lp` (or whatever your machine uses). On the Mac and PC a simple `print` will send the figure to the printer directly. MATLAB prints the *current* figure, that is, the one you last changed. If you have used `figure` to open several figure windows, you can make an old figure active by typing `figure(x)`, where `x` is the number of the figure. UNIX note: You can execute shell commands from within MATLAB by prefacing them with `!` (exclamation point, or bang as I learned as a kid). Thus, for instance, `print -dps graph1.ps; !lp graph1.ps` will print the current figure.

Sometimes data will have a large range that demands a logarithmic plot. For this, you can either manually `log` the data and plot it normally, or you can use `semilogx` (x is logged), `semilogy` (y is logged) or `loglog` (both are logged) in place of the usual `plot`. `semilogy` is often useful for a quick dB-style plot; I prefer something like `plot(x, 20*log10(y))` ultimately because you know you are getting genuine dBs.

Two-dimensional data can be visualized in several ways; `image` is used to display data as an intensity map, `mesh` as a surface, and `contour` as a contour map. Other exotica that are sometimes useful include `bar`, `hist` and `stairs` for one-dimensional data, and `quiver` for two-dimensional data.

Functions and scripts

Up to now, we have been entering commands at the MATLAB prompt. Sooner or later, you are going to want to be able to save and edit strings of commands. MATLAB has two ways of dealing with this. The simpler method is the script, which is a file of commands that are executed as if they had been typed at the keyboard. For instance, the seven lines of MATLAB code given in the previous section can be typed into a text editor and saved as `signal_demo.m`; these lines can then be executed within MATLAB by typing `signal_demo` at the prompt. Once the script has executed, all variables defined inside it remain and are accessible.

Some sequences of commands are not self-contained; rather, they are called by other scripts and require data to be passed to them. They may also return data to the calling script. Such sequences are called functions. They should always be preferred over scripts if they are called by another script, since all variables defined inside the function are destroyed on exit; there is therefore no chance of a clash of variable names. Furthermore, functions have a well-defined and simple method of transferring data. Let us define a simple function that returns the hyperbolic cosine of an input:

```
function [result]=my_cosh(input)
%MY_COSH  Hyperbolic cosine.
%
%     Y = MY_COSH(X) finds the hyperbolic cosine of each element in matrix X
%     and returns the result in Y.

%     T Kite 1997
result=(exp(input)+exp(-input))/2;
```

Save these lines in a file called `my_cosh.m`. From the command prompt (or from a script), try out the function with `my_cosh(1:5)` or `x=my_cosh(magic(3))` or anything else. Notice that the function works with vectors and matrices as well as scalars; this is because every operation inside the function is defined for a matrix argument. Sometimes a function will need to find the square of the input, for instance; you should use '`x.*x`' (element-by-element multiplication) for this, rather than '`x*x`' (matrix multiplication). If you use the latter form, the function will work correctly for scalar inputs, will work incorrectly for square matrix inputs, and will fail with an error for all other inputs, because matrix multiplication is not defined for these.

From the command line, type `help my_cosh`. You will see the commented lines (up to the line break). Thus a function has a simple help facility; you can try this with any built-in command too. `help plot`, for instance, will return a large list of features that can be used. Much of learning to use MATLAB consists of hearing about commands and then using `help` to find out how to use them, or by looking at a list of related commands in a help file. Because the help file consists of all contiguous commented lines below the `function` declaration, leaving a space allows you to write comments to yourself that will not be displayed as help, as shown above.

Because functions are called by name and do not have to be declared other than in the first line of the file, MATLAB is an extensible language. You will find, after using it for a while, that you will write functions that you use over and over again. These functions have now effectively become part of the language. Indeed, many of the functions supplied with MATLAB are actually function files rather than code built into the kernel. You can find this out by typing `type <function name>` at the prompt; an external function is echoed to the screen, while '`<function name> is a built-in function`' is returned otherwise. `type` can be helpful for learning MATLAB, as it enables you to examine other (hopefully well-written) code.

MATLAB has a `path` command to help you organise your files and functions. When you first fire up MATLAB, it looks for a file called `startup.m` in the directory you were in when you invoked it. This file can contain any MATLAB code, but it is normally used to define the path that MATLAB will use to look for functions. Thus your `startup.m` file might look like

```
path(path, 'ee381k')
path(path, 'ee381k/homework')
path(path, 'matlab/my_functions')
```

in which case MATLAB will look in these directories for functions when you call them. Type `path` at the prompt to examine the current path. The path is printed in order, that is, the paths are searched in the order shown for a function. That means that if you give a function a name that already exists in a path that is searched before the path containing your function, your function will not be found. Call it something else.

A final note about `for` loops

Remember, MATLAB is a vectorized language. It was designed to operate on matrix arguments. Most of the time, you will not need to use a `for` loop (although MATLAB provides it, as well as `while`). If you find yourself doing a `for` loop over a vector to do the same thing to each element, that code can almost certainly be vectorized. Vectorized code runs enormously faster (often by a factor of 50) than looped code because it is compiled rather than interpreted. Get into the habit of writing vectorized code—although it is difficult at first, it will soon become habitual and will save you time in the long run. It also means you won't get the dreaded 'Don't use FOR loops!' on your homework from me, in large red letters. Good luck!