Matlab Tutorial-1

Jim George
Colorado State University

September 27, 2005

1 Introduction

Matlab is a numerical computation tool, which can be used to solve problems involving vectors matrices (hence the name). Results obtained from these computations can be visualized with a variety of graphing utilities. Matlab may be used in either interactive- or batch-mode. This tutorial introduces the basic vector/matrix manipulation syntax, some higher-level vector manipulation commands and the 2D graphing commands.

2 Getting Started

2.1 Starting up Matlab

On a Windows machine, Matlab may be started from the Start menu, under Engineering Applications. Alternately, you can type `matlab` from a command prompt or from the "Run Application" window.

On a Unix workstation (HP-UX, Sun or Linux), open a shell window and type `matlab` at the prompt. This works if Matlab is locally installed on the workstation. If not, you can log into one of the Linux workstations at Andersen Lab (named `linux1` through `linux6`), the Sun Java workstations (`cae1` through `cae12`), Solaris workstations (`cae13` through `cae25`), Solaris compute server (`sunray`) or Linux compute servers (`sunflare`, `sunglare`, `sunshine` or `sunrise`) by using the command

```
ssh -X <remote host>
```

Once you are logged in to the remote machine, you can type `matlab` to begin. Note that when running Matlab in this manner, it is actually running on the remote machine and your local machine is merely displaying the results. Sometimes, time-consuming operations can be done quickly by using one of the compute servers, which are significantly faster than the average workstation.

2.2 The Matlab GUI

Once Matlab has started up, you should see a GUI similar to the one shown in Figure 1. By default, Matlab divides the GUI into three panes, the two panes on the left contain the Workspace Browser and Command History. The larger pane is where you can type in commands.

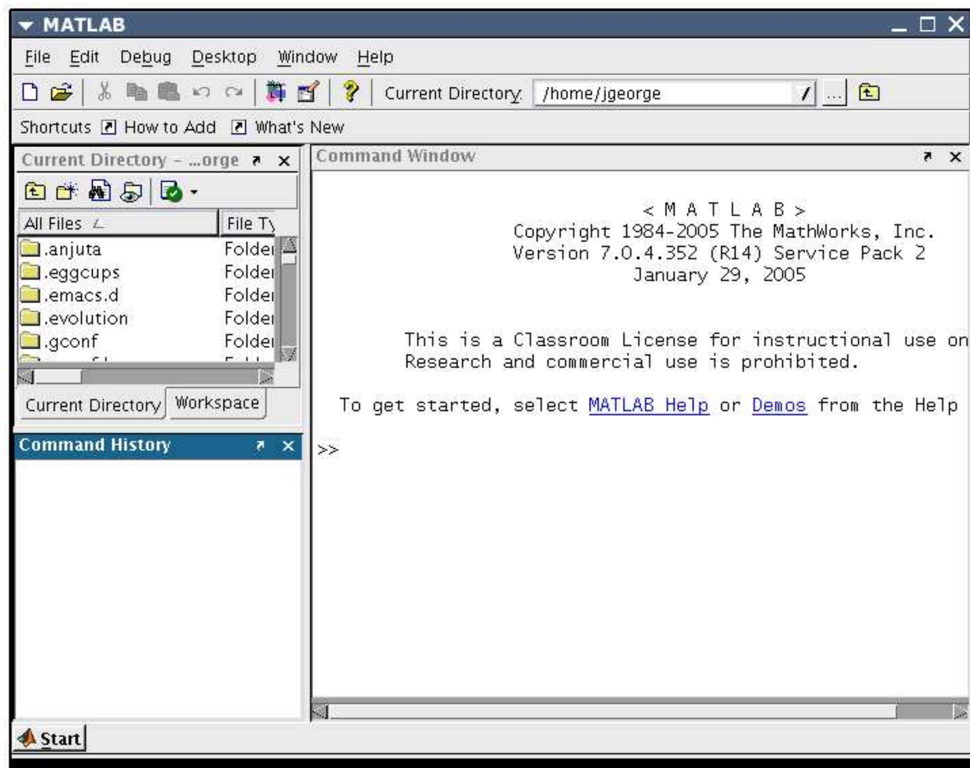

Figure 1: Matlab GUI

2.2.1 Command Window

The command window (by default, the large pane on the right) is where you can type in commands and see the results of both interactive commands and batch-mode scripts. The prompt >> indicates that Matlab is ready to accept commands typed in interactive mode. One of the first commands you should get familiar with is `help`, which allows you to search for a quick-reference style help text on any Matlab command. Another useful command to know is `doc`, which pulls up a more elaborate help document which can often include examples.

2.2.2 Workspace Browser

The workspace browser is useful to quickly examine the contents of the current workspace, and also to perform simple commands like plotting, without having to type in anything. It comes in handy when you are debugging scripts, to find out how big a given variable is.

2.2.3 Current Directory Browser

The current directory browser shows the contents of the current directory. By default (on Windows systems), Matlab starts up with the current working directory set to the Matlab install directory. It's always advisable to switch to using your `U:` drive (under Windows) or your `/home` directory under Unix when working with Matlab.

2.3 Variables and the Workspace

Variables in Matlab can be of several types, including scalars, vectors, matrices and structures. The Matlab Workspace contains all the variables you have created. To clear a variable from the workspace, use the `clear <var>` command, where `var` is the name of a variable. Issuing the `clear all` command clears all variables from the workspace.

Variables don't need to be explicitly created in Matlab, they are automatically created the first time they are used. However, a variable must exist for it to be *readable*.

To examine the contents of any variable, simply type its name at the Matlab command window.

Built-in variables in Matlab include `i` ($\sqrt{-1}$) and `pi` (π). A common beginners' mistake is to accidentally assign a new value to one of the built-ins (this is not flagged as an error).

2.3.1 Scalars

Scalars are the simplest variables in Matlab. Scalars store a single value, usually a real number. A scalar my be created by using the command

```
scalar = 10
```

This creates a new scalar called "scalar" in the workspace, and assigns it a value of 10.0. Note that typing in this command makes Matlab print out the variable name and it's value. To suppress this printout, add a semicolon to the end of the assignment statement. Note that the scalar shows up in the Workspace Browser window on the left.

2.3.2 Vectors

Vectors in Matlab are stored as one-dimensional arrays of real numbers (actually, they are double-precision floating point). To create a simple vector, you can use the box-bracket operators, as shown:

```
vec = [1; 2; 3];
```

This creates a new vector called `vec` and assigns it a vector

$$\text{vec} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

Note that in this example, `vec` is a column vector. To create a row vector, the syntax is

```
vec = [1 2 3];
```

A vector's components can be independently accessed by using an index value. For example, to check the contents of the vector `vec`'s second element, type `vec(2)` at the command window. Note that the indices in Matlab, unlike most programming languages, start with 1, not zero.

Creating vectors of sequences is simple.

```
vec_seq = [1:2:30];
```

This example creates a vector with numbers ranging from 1 to 30, with an increment of two. If the increment is omitted, it is assumed to be one. Thus, `vec_seq = [1:30]` will have numbers ranging from 1 to 30 with an increment of one.

The function `length` gives the length of a vector, typing `length(vec_seq)` should give you a result of 30.

Other ways of creating vectors includes using the `linspace(x1,x2,N)` function, which generates a length-N vector of equally-spaced values between x1 and x2. `logspace(x1,x2,N)` returns a length-N vector with lograrithmically-spaced values between x1 and x2.

Vector concatenation allows two or more vectors to be spliced together to form a new vector. The syntax for this is

```
vec_concat = [vec1 vec2];
```

This works if `vec1` and `vec2` are row vectors. If they are both column vectors, then insert a semicolon between `vec1` and `vec2` in the example.

Vector Slicing allows a portion of a vector to be extracted out. For example, to get elements 15 to 20 from a larger vector, the syntax to use is

```
small_vec = large_vec(15:20);
```

Note that `small_vec` has elements 15 to 20 *inclusive*. Thus, it is six elements long, not five. The subscript of the source vector (`large_vec` in the example) can have an increment specified, such as `small_vec = large_vec(15:2:20);`. This creates a new vector with elements 15, 17 and 19 from large_vec.

2.3.3 Matrices

The syntax to create and assign matrices is very similar to those for vectors. Enclose the matrix elements in box-brackets, using spaces to separate values on the same row and a semicolon to separate values belonging to different columns. All the rows and columns must have the same number of elements. An example of matrix creation is

```
array = [1 2 3; 4 5 6; 7 8 9];
```

This creates the following array:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

The `size` function returns the size of the array in rows and columns.

```
new_array = [1 2 3; 4 5 6];
[rows cols] = size(new_array);
```

This code fragment creates a new array, then uses the `size` function to get it's size. Note that this function returns an array, and to get the results into two scalars, the syntax `[a b]=size(array);` may be used.

Functions exist to create some commonly used matrices. The `zeros(M,N)` and `ones(M,N)` functions return matrices of all zeros and ones respectively, with N rows and M columns. The `eye(N)` function returns an $N \times N$ identity matrix ($\mathbf{I}$).

The function `meshgrid(x,y)` can be used to create an array, with the rows spanning the values specified in `rows` and the columns spanning the values in `cols`. For example,

```
[x,y] = meshgrid(-2:2, -1:1)
```

generates the following matrices

$$x = \begin{bmatrix} -2 & -1 & 0 & 1 & 2 \\ -2 & -1 & 0 & 1 & 2 \\ -2 & -1 & 0 & 1 & 2 \end{bmatrix}$$
$$y = \begin{bmatrix} -1 & -1 & -1 & -1 & -1 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

Matrix Slicing is analogous to slicing vectors. For example, `x(1:3,2:4)` is

$$\mathtt{x}(1:3, 2:4) = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}$$

2.4 Basic Expression Evaluation

Matlab can evaluate expressions typed at the command line and within script files. Expressions can contain numerical constants, scalars, vectors and matrices, operators and function calls. The result of the expression evaluation is displayed in the command window, unless the expression has a trailing semicolon.

Here are some examples of expressions with descriptions of what they do

Expression	Description
`2+sin(2*pi*0.6)`	Evalutates the expression, assigns it to the built-in variable `ans` and also displays the result
`x = [-1:0.1:1];` `cplxsine = exp(i*pi*x*2);`	Creates a vector `x`, filled between -1 and 1, with an increment of 0.1. Next, the function $e^{i \cdot 2\pi x}$ is evaluated for each value of `x` and stored in the variable cplxsine.
`mag = cplxsine .* cplxsine;`	Performs an element-by-element multiplication of the vector `cplxsine` by itself and stores the result in `mag`. Note that the multiplication sign (`*`) is preceded by a period.

`I = eye(3);`
`M = [0 0 1; 1 2 1; 1 0 0];`
`Minv = inv(M);`
`Y = Minv*M - I;`
	Generates an identity matrix `I`, a matrix `M`, computes the inverse of `M` and finally multiplies `M` with it's inverse, subtracts the result and stores it in `Y`.

`Msq = M .^ 2;`
`r1 = cplxsine(3:5) * Msq;`
`r2 = Msq * cplxsine(3:5)';`
	Does an element-by-element squaring of the matrix `M` (note the period before the exponent operator) and stores the result in `Msq`. Next, elements 3 to 5 of the vector `cplxsine` (a row-vector) are assembled into a vector and pre-multiplied with matrix `Msq`. Finally, the same three elements of `cplxsine` are transposed (the transpose operator is the single-quote ') and post-multiplied with `Msq`.

2.5 Matrix Operations

Certain matrix operations like mulitiplication exist in two forms, true matrix multiplication and element-by-element multiplication. Element-by-element operations are distinguished by a leading period (such as `.*` or `.^`), and are used in Matlab where most other high-level languages would use a looping construct (like C's `for` loop) to perform the same operation on an array of data. Note that while Matlab still supports looping constructs, scripts making use of matrix operations execute faster.

The table lists some matrix functions

inv(X)	Computes the inverse of matrix X
rank(X)	Computes the rank of X
det(X)	Find the determinant of X
triu(X), tril(X)	Triangular decompositions (upper/lower) of X
[Q R] = qr(X)	QR decomposition of X

3 Visualization

Matlab provides several in-built functions to produce high-quality graphics, suitable both for on-screen presentation as well as for inclusion in papers, reports and theses.

3.1 Concepts

Matlab outputs graphics to one of many *figure windows*. A figure window by itself only has a title bar and a drawing area, as shown in Figure 2. A figure window can be opened by using the `figure` command. `figure` by itself will open a new figure window, leaving all existing ones in place. Using the `figure(n)` command creates a figure window numbered n, if it does not exist, otherwise the existing figure window is selected for output, so the next graphics output command will write to that figure. This command will return a *figure handle*, which is typically not used except with the more advanced plotting routines.

Subplots are a convenient way to plot more than one graph on the same figure window. Subplots can be created using the `subplot(x,y,idx)` command. Here, `x` and `y` are the number of subdivisions along the X and Y axes, respectively. These subdivisions are numbered as `x_index + y_index` × `y` , where `x_index` and `y_index` are the XY coordinates of the subdivision. By specifying the index in the `subplot` command, the corresponding subdivision is selected for graphics output. Figure 3 shows the effect of running the command `subplot(2,1,1); subplot(2,1,2);`

3.2 2-D plots

3.2.1 Simple Cartesian plots

This may be used to visualize a vector of real data. The command `plot(vector)` plots the contents of `vector` to the currently open plot window. If no plot window is open, a new one is created. Matlab will automatically scale the x and y axes to display the entire vector on the plot window. To change these axes, you can use the `axis([x_low x_high y_low y_high]);` command. Note that the x axis is merely the index into the vector for a given data point. This can be changed to a more meaningful number by using the `plot(x,y)` command, which assumes `x` and `y` are vectors of equal lengths, and treats them as a pair of coordinates describing the plot.

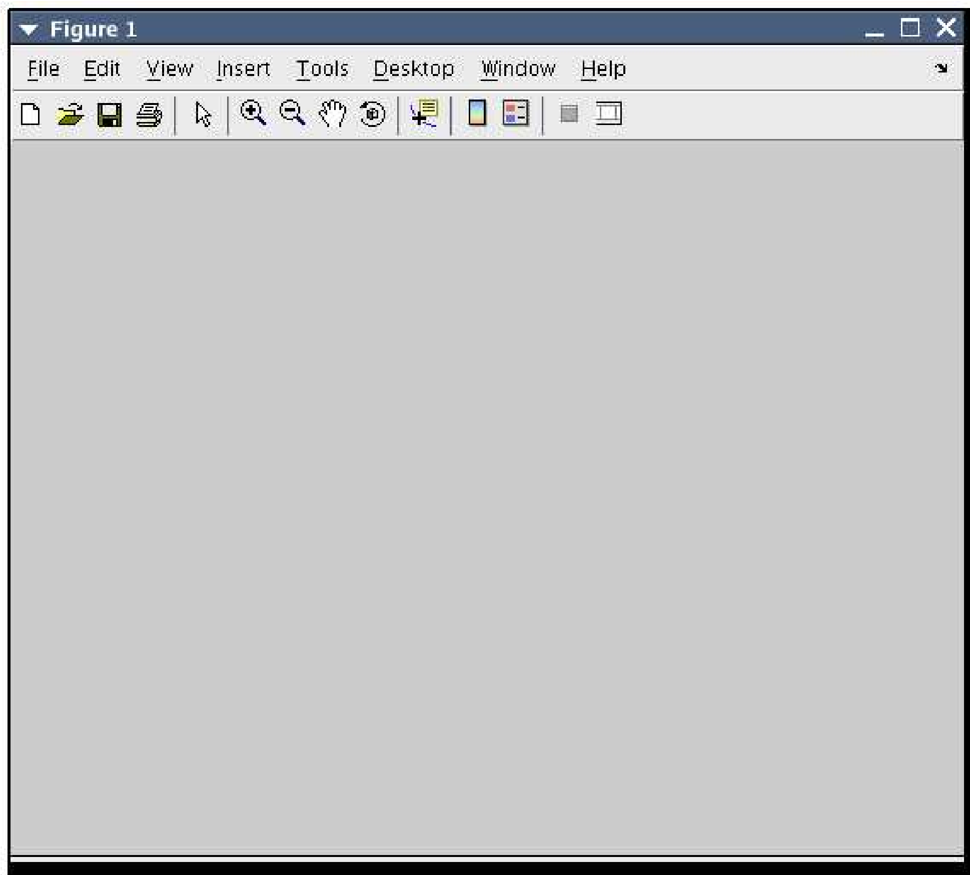

Figure 2: Matlab Figure Window

Plot also takes a third optional parameter to format the displayed line. The line specifier can control the line style, color and data markers (if any). The parameter is a character string, divided into three parts. The first part specifies the line style, and is one of the following

`-`	Solid Line
`--`	Dashed Line
`:`	Dotted Line
`-.`	Dash-dot Line
`(none)`	No Line

The second part describes the data markers. There are many marker styles available (type `doc LineSpec` for details), some commonly used ones are listed below

`(none)`	No Marker
`o`	Circle
`s`	Square
`x`	Cross

The third part describes the color of the trace and associated markers. Again, several colors are available (see `doc LineSpec`), some common ones are listed in the table

`b`	Blue
`r`	Red
`g`	Green
`k`	Black

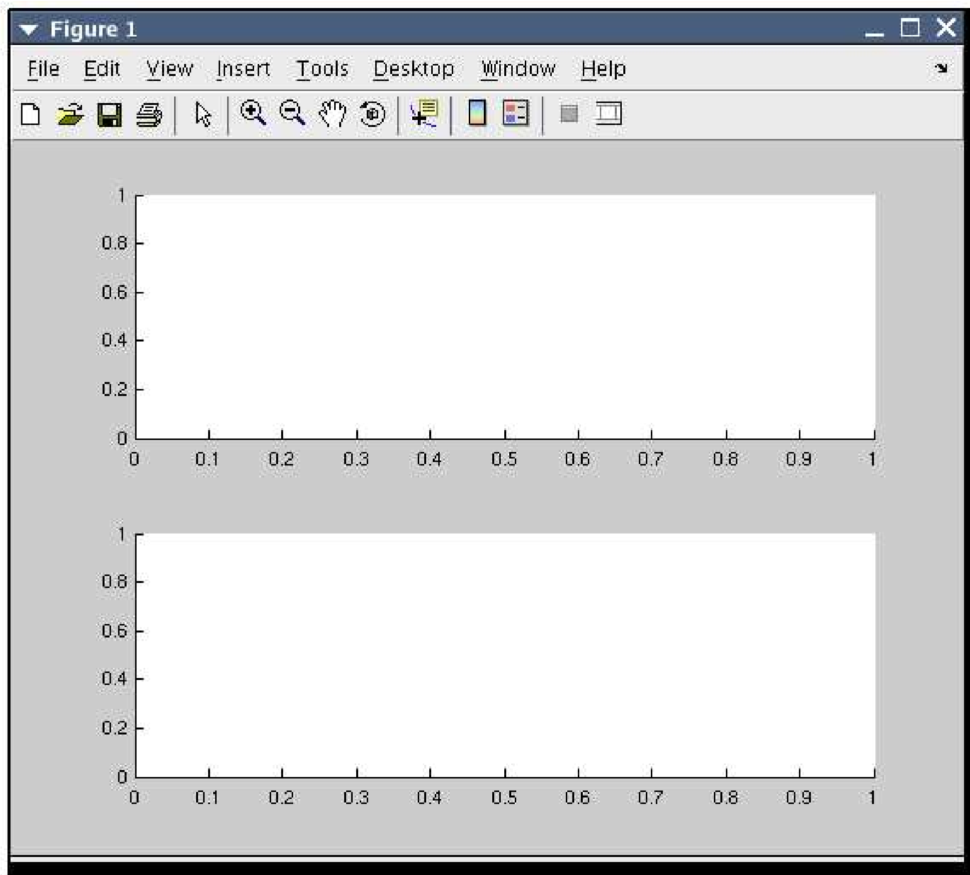

Figure 3: Subplots

Other variants of the `plot` command exist, which add various features. These are summarized in the table below

`semilogx, semilogy`	Plot a semilog graph, along x or y
`plotyy`	Takes in two pairs of x-y coordinates, and plots them with two y axes, on either side of the graph
`loglog`	Plots a log-log graph
`stem, stairs`	Plots a stem-graph or a stairstep graph, useful for visualizing DSP sequences
`bar`	Plots a bar-graph using the specified data

3.2.2 Polar 2D plots

Polar plots are useful to display data which is naturally representable in polar coordinates as opposed to cartesian coordinates. As an example, we can plot the antenna radiation pattern of an ideal dipole antenna (normalized with the power output from an ideal isotropic radiator), which is represented as

$$\left|\frac{P}{P_i}\right| = |\sin\theta|$$

This can be plotted on a polar plot by using the following commands

```
theta = [-1:0.01:1] * pi;
polar(theta,abs(sin(theta)),'r:');
```

Polar plots using the `polar(theta,rho)` command must have the theta axis scaled to be within the range $-\pi \leq \theta \leq \pi$.

3.2.3 2-D Image plots

Images within Matlab can be described by a 2D array or matrix. An image can be displayed as a 2D color map through the use of the `pcolor(x,y,z)` function. Here, `x` and `y` are the vectors defining the extents of the plot (they are optional), and `z` is the actual data to be plotted (a matrix). `pcolor` can also interpolate color values between points, to allow plotting of small datasets without pixellation. Using the `colorbar(dir)` function enables the display of a colorbar by the side of the image map. The `dir` parameter specifies the colorbar direction. If it's not specified, it defaults to `'vert'`. The actual colors used to plot the image are selected by the `colormap(name)` command. You can use one of the built-in color maps, such as `jet`, `bone` or `hot`.

An example of 2D color map plotting is shown below

```
[x y] = meshgrid([-1:0.01:1], [-1:0.01:1]);
z = abs(cos(2*pi*x)^2 + cos(2*pi*y)^2);
figure(1);
pcolor(x,y,z);
shading interp;
colorbar('horiz');
colormap('bone');
```

The code starts by creating a mesh of evenly spaced values from -1 to 1 on each axis. Then, it computes the function $\left|\cos(2\pi x)^2 + \cos(2\pi y)^2\right|$ and plots the results as an image. A horizontal color bar is also added to the image. The colormap is set to `bone`.

Matlab can also generate contour plots from the datasets used for 2D images. This is done using the `contour(x,y,z,contourlist)` function. The `x`, `y` and `z` parameters have the same meaning as for pcolor. The `contourlist` is a vector of values for which contours should be drawn. The `clabel(cs,h)` function may be used to label each contour line. `clabel` has a lot of options, be sure to read the online help!

3.3 3-D plots

3.3.1 Surface Plots

Matlab can plot surfaces from 2D data, where the z-axis (height) is provided by the data array. A family of functions exist to do this type of plot, the most basic of which is the `mesh(x,y,z)` function. The `x` and `y` parameters provide the labels and extents for the x and y axes, while the `z` parameter is a matrix containing the actual data to be plotted.

Here, the same data from the 2D image plotting example is plotted in 3D.

```
[x y] = meshgrid([-1:0.1:1], [-1:0.1:1]);
z = abs(cos(2*pi*x)^2 + cos(2*pi*y)^2);
figure(1);
mesh(x,y,z);
```

Note that the granularity of the mesh grid has been reduced, to make the 3D plot easier to understand. The output of the mesh function is a 3D mesh, with hidden surface removal. The mesh lines are colored using the current color map, based on the Z value.

The `surf` command is very similar, except that the spaces between the mesh are filled in with an appropriate color. The algorithm used to perform the filling may be set using the `shading <mode>` command. The mode can be set to `flat`, `interp` or `faceted`. The `interp` option gives the smoothest graph quality. Faceted is similar to flat, except that a black border is drawn around each mesh polygon.

The `surfc` command will draw the contours of the plotted mesh below the actual mesh itself.

The `surfl` command draws a mesh, with 3D lighting enabled. This requires a colormap with smooth shades, such as `copper`, `bone` or `jet`.

The following example draws a sample dataset with all three surface plotting methods.

```
[x y] = meshgrid([-1:0.1:1], [-1:0.1:1]);
z = abs(cos(2*pi*x)^2 + cos(2*pi*y)^2);
figure(1);
surf(x,y,z);
shading flat;
figure(2);
surfc(x,y,z);
shading interp;
figure(3);
surfl(x,y,z);
shading interp;
```

3.4 Annotating Graphs

Graphs created with any of the functions mentioned above can be annotated using a variety of Matlab functions. The simplest of these is the `title(text)` function, which sets the title of the current graph to `text`. Note that the string provided in `text` supports a subset of LaTeX notation. A typical use for LaTeX notation would be to include subscripts or Greek characters. Some of the commonly used commands are shown in the table.

\sub{text}	Text within the braces is subscripted
\sup{text}	Text within the braces is superscripted
\pi	π
\Lamdba	Λ
\int	$\int$

The `xlabel(text)`, `ylabel(text)` and `zlabel(text)` commands label the axes of the currently selected figure window.

The `legend(string1, string2, ...)` function is used to insert a legend box for graphs which have more than one trace.

`text(x,y,string)` places `string` at the location (x,y) on the graph.

4 Matlab Scripting

Matlab includes a powerful scripting language, which allows for batch-mode execution of expressions and Matlab commands. The scripts are stored in plain-text files with the `.m` extension. Typically, scripts are executed in linear order, but Matlab also has branching, conditional statements and procedure calls.

4.1 Matlab Editor

Since Matlab scripts are plain text files, any editor (such as vi or notepad) can be used to edit the script files. Using Matlab's inbuilt editor, however, has the benefit of syntax highlighting, smart indenting and integration with the Matlab debugger. Note that the keybindings used are different under Unix (follows Emacs keybindings) and Windows (follows CUI keybindings). If you are unfamiliar with Emacs, several tutorials are available online.

4.2 General Information

Matlab scripts have automatic variable creation (there is no need to declare variables) upon first use. Once created, variables stay within the workspace until they are cleared out using the `clear` command. They may be examined with the Workspace Browser or by using commands in the Command Window. It is generally a good idea to issue a `clear all` command at the top of your scripts. This way, all scripts start off with a known state.

Comments in Matlab scrips are indicated using the percent symbol.

4.3 Conditionals

Matlab scripts can make use of the `if-else` branching statements to control program execution order. The syntax for the `if-else` statement is shown in the example below

```
if num_values < 25
    if manual_input
        disp('Number of inputs is too low');
    else
        re_read = 1;
    end
end
```

As with most languages, `if-else` statements can be nested, and the `else` is optional. The `if` block is delimited by `end;`.

4.4 Loops

Matlab provides the standard `while` and `for` loops found in most programming languages. The `for` loop syntax deserves some mention, since the sequence over which it loops can be completely arbitrary, since it's argument is a vector. Thus, it closely resembles Perl's `foreach` loop. An example using `do-while` and `for` loops is shown below

```
# Bubble sort the list
done = 1;
while(~done)
    done = 1;
    for ctr = 2:length(invec)
        if invec(ctr) > invec(ctr - 1)
            temp = invec(ctr);
            invec(ctr) = invec(ctr + 1);
            invec(ctr + 1) = temp;
            done = 0;
        end;
    end;
end;
```

4.5 Function calls

Matlab allows you to expand it's set of in-built functions and toolboxes by creating new .m files which contain functions. These .m files are different from regular scripts, since they can be called from other scripts and passed arguments. To turn a script file into a function, include the following line at the top

```
function [return_val_1 return_val_2] = function_name(arg1, arg2, arg3)
```

This tells Matlab that this file defines a function named `function_name`, which takes three arguments and returns a list of two return values. Note that the file name with which the .m file is saved must match the function name. Matlab uses the file name to search for the function when it first encounters it, and will issue an error if the defined function is not found in the file. Matlab searches through it's installed toolbox list,

then in the current working directory for `.m` files, so make sure the function file is in the same directory as the script which calls it.

Function calls may take a variable argument list, this is specified by using the reserved word `varargin` as the last (or only) argument name. This is similar to the ellipses (...) used by C/C++ to indicate a variable argument list. Then, within the function body, you can use `varargin` as a variable of type "cell array" to get the arguments. The variable `nargin` holds the number of arguments passed. An example of a function which takes variable arguments is given below

```
function [s] = get_sum(varargin)
s = 0;
for ctr = 1:nargin
    s = s + varargin{ctr};
end
```

Note that the subscript for varargin is in braces, not brackets.

Returning from a function is accomplished by "falling off the edge" of the function. There is no equivalent to C's `return` statement.

5 Importing Data into Matlab

Matlab can be used to analyze or visualize results obtained from experiments. Data import functions are available to help you along with this process. Two methods are available for data import, one is to use the Import Wizard, the other is to read in files directly.

5.1 Import Wizard

The Data Import Wizard can be used to read in files from one of the standard data interchange formats, or from a text file. One form of data which it understands is the CSV (comma-separated value) format, which can be read and written by many different programs including spreadsheets like Excel. You can access the wizard by using the File ▷ Import Data menu in the main Matlab window.

Using the wizard is self-explanatory, and is usually trouble-free unless the source data is not in a standard format.

5.2 File Reading

This method can be used when reading in files which contain data in a non-standard format. An example is a CSV file which has comments inserted, which tends to confuse the Data Import Wizard. Another use for file-reading is when the data is in a binary file.

5.2.1 File Handles

Matlab provides an `fopen(filename, mode)` function, similar to the one used in the C runtime library. `filename` is a string containing the filename to be opened. `mode` is a string used to describe the mode in which the file is opened, using character codes. Commonly used codes are `'r'` for read mode, `'w'` for write mode, `'+'` for update mode and `'t'` for text mode (required on Windows systems, ignored under Unix).

These can be combined, as in `'r+t'` for read/update mode with text mode conversion. The `fopen` function returns a *file handle*, a way of identifying the file. The file handle is used when reading/writing to the file.

Files opened with `fopen` must be closed in order to free up resources and also to commit any changes if the file was opened for writing. This is done using the `fclose` function.

5.2.2 Text Files

Text file reading is done using the `[var1[,var2,...]] = fscanf(fid,fmt_string,size)` function. Here, `fid` is the file handle returned by `fopen`, and `fmt_string` is a format control string, similar in syntax to the C/C++ version. The optional `size` parameter tells fscanf how many elements to read from the file. If it is not specified, file reading continues until the end of file is reached. You can also write to a text file using `fprintf(fid,fmt_string,...)` function. A major difference between the `fscanf` and `fprintf` functions and their C versions is that in Matlab, they support vectorization. This means that if the arguments are vectors, then the entire string is written multiple times, once for each element in the vector for `fprintf`, and multiple lines are read and converted in the case of `fscanf`.

An example of reading data from a text file is shown below.

```
fid = fopen('myfile.txt','rt');
[col1 col2] = fscanf(fid, '%d %f', 10);
fclose(fid);
```

This opens the file myfile.txt in text mode for reading, then reads in two columns of data from the file. The first column contains integers, the second column contains floating-point numbers. File reading stops after 10 records are read. The variables `col1` and `col2` will contain the data read in from the file.

5.2.3 Binary Files

Importing binary files into Matlab is not as straightforward as with text files, it requires a knowledge of the internal structure of the file, and the format in which the data is saved. The function `fread(fid, size, precision)` is used to read in binary data. Here, `fid` is the file handle returned by `fopen`, `size` is the number of records to read and precision is the size of each data element. These are strings, containing one of the Matlab data types (such as `uchar`, `int16`, `uint32`, etc). If `size` is of the form `[M N]`, then the elements are treated as an `MxN` matrix. Here, `N` can be the special value `inf`, which makes `fread` read data to the end of the file.

5.2.4 `.MAT` files

Sometimes, it is useful to save certain variables from the workspace (such as the results of a long computation) into a file for later use. `.MAT` files may be used to do this in a simpler manner than using binary or text files. You can use the `save('filename', var1, var2, ...)` to save the specified variables to a disk file named `filename`. The file (and the variables it contains) can later be loaded back using the `load('filename')` command. The `load` function returns a list of variables stored in the file, so you can assign them to new variables when reading back the data using the syntax `[var1 var2 ...] = load('filename');`

6 Polynomial Operations

6.1 Polynomial Representation

Matlab's support for polynomials allows for a variety of computations. Polynomials are represented as a vector containing the coefficients of the polynomial in descending powers. For example, the polynomial $3x^2 + 12x + 5$ is represented as the vector `[3 12 5]`.

Polynomials may be evaluated by using the `polyval(poly, x)` function. `poly` is the polynomial to be evaluated, and x is a vector of values at which to evaluate `poly`.

6.2 Polynomial Arithmetic

Polynomial addition and subtraction can be done using the usual vector add and subtract functions. Note that this works only if both polynomials are of equal order. If they are not, then they must be zero-padded until they are of equal length. A function to do this is shown below

```
function [s]=polyadd(x,y)
lx = length(x);
ly = length(y);
if (lx < ly)
    s = y + [zeros(1,ly-lx) x];
else
    s = x + [zeros(1,lx-ly) y];
end
```

Multiplication of polynomials is accomplished by using the `conv(x,y)` function, which convolves the sequences `x` and `y`. You can use the `deconv(x,y)` function divides polynomial `x` by polynomial `y`.

The `roots(c)` function returns a list of roots for the specified polynomial. The inverse function is `poly(r)`, which returns the polynomial coefficients for a given list of roots.

The `[r p k] = residue(b,a)` function converts the polynomial $\frac{b_1x^m+b_2x^{m-1}+...+b_{m+1}}{a_1x^n+a_2x^{n-1}+...+a_{n+1}}$ into it's equivalent partial fraction version $\frac{r_1}{1-p_1} + \frac{r_2}{1-p_2} + ... + \frac{r_n}{1-p_n} + k$, where r_n are the residues, p_n are the poles and k is a row-vector of direct terms. The `[b,a] = residue(r,p,k)` function does the inverse operation.

The `p = polyfit(x,y,n)` function finds a polynomial p such that `p(x(i))` is fit to `y(i)` in the least-squares sense. The maximum order of `p` is `n`.

An example of polynomial fitting is shown below.

```
x = [-1:0.1:1];
y = sin(2*pi*x);
p = polyfit(x,y,6);
nx = [-1.3:0.1:1.3];
plot(nx, polyval(p,nx), nx, sin(nx);
```

This generates a sine wave and tries to fit a sixth-order polynomial to the sine wave over the interval $-1 \leq x \leq 1$. Plotting the results shows that the fit is quite good within the specified range, but outside that range, the behaviour becomes increasingly chaotic.

Polynomials may be differentiated or integrated, this is done using the `polyder(p)` and `polyint(p,k)` functions. The argument `p` is the polynomial, while `k` is the constant of integration. Polynomial integration may be demonstrated by extending the previous example as follows

```
pc = polyint(p,0);
plot(nx, polyval(pc, nx), nx, -cos(nx));
```

Since we know that $\int \sin(x)dx = -\cos(x)$, we can show that the polynomial integration has worked by comparing the two plots.

7 Elementary Statistics

7.1 Basic Statistical Functions

Matlab can be used to perform basic statistical analysis on data sets, through the use of the statistics toolbox. Some basic functions are listed below

max(A)	Computes the largest component in `A`. If `A` is a vector, then the largest component is returned. If `A` is a matrix, `max(A)` returns a row vector with each row containing the maximum value for the column vectors in `A`.
mean(A)	Returns the mean value of `A` for a vector, and mean value of each column vector if `A` is a matrix
median(A)	Returns the median value of `A`
std(A)	Returns the standard deviation of `A`
cov(A)	Returns the covariance of vector `A`. For a matrix `A`, the covariance matrix is returned. If each column of `A` is a vector, then diag(cov(A)) returns the covariance of each column
xcorr(A,B)	Compute the cross-correlation between `A` and `B`
diff(X)	Compute the finite differences between successive elements

The `hist(x,n)` function is useful to plot a histogram of a set of data points. `x` is the input data and `n` is the number of bins in which to compute the histogram.

7.2 Distribution Fitting

Often, it is useful to fit experimental data to a known distribution. Matlab provides a GUI-based tool called the Distribution Fitting Tool (invoked using `dfittool`), which allows you to fit one of several distributions to the dataset provided. To try out this tool, first generate a data set using the command `x = randn(10000,1);` The `randn(m,n)` function returns an `mxn` matrix of normally distributed data, with $\overline{x} = 0$ and $\sigma = 1$. Start up the tool by typing `dfittool` and import the data by using File ▷ Import Data. You can now create a normal fit by choosing New Fit and choosing "Normal" in the list of distributions. The values of $\overline{x}$ and σ are printed and a gaussian (normal) fit is generated by clicking "Apply".

7.3 Minimum Value Search

Matlab can find the minimum of a function over a given range. This is done using the `fminbnd` and `fminsearch` functions.

`x = fminbnd(@function, x1, x2)` searches for a minimum of the named function, between `x1` and `x2`. The function name is specified as `@functionname`. It must be a function which takes one parameter and returns at least one result. An example of this is shown below

```
x = fminbnd(@sin, -5, 5);
disp(x);
myfunc = @(x)(x^2 - x);
x = fminbnd(myfunc, -5, 5);
disp(x);
```

The first call finds the minimum value of sin(x) in the range -5 to 5. The second call declares an "anonymous function" called myfunc and finds it's minimum between -5 and 5.

The function fminsearch is similar, but allows the function to have multiple variables. It then performs an unconstrained non-linear optimization to find the minimum value of the function. `[x,fval,exitflag] = fminsearch(@fun,x0)` will search for the minimum of fun, with an initial guess of x0. The return value `x` is the value of `x` at which the minimum was found, `fval` is the minimum found and `exitflag` describes if the function was successfull (1) or failed to converge(0). If the function takes in multiple variables, then it must accept them as a vector, and both x0 and the return value x are then vectors.

8 Differential Equation Solvers

Matlab provides a range of functions with which Differential Equations may be solved. These "solver" functions take, as a parameter, a function which describes the differential equation in a standard form, as shown below

$$\begin{aligned} \dot{y} &= f(t,y) \\ y(t_0) &= y_0 \end{aligned}$$

These are known as "initial value problems", since the value of $y(t)$ at time t_o and the time-derivative of y are specified. As an example of an initial value problem, we will use the van der Pol equation, which can be written as

$$\ddot{x} - \mu\left(1 - x^2\right)\dot{x} + x = 0$$

This differential equation describes self-sustaining oscillations in which energy is fed into small oscillations and removed from larger ones. It is used to describe the behavior of an LC tank across which a negative resistance is connected, which is typical of an LC tank oscillator.

To solve this differential equation, we must first bring the van der Pol equation to the standard form. This is done by setting $y_1 = x$ and $y_2 = \frac{dx}{dt}$. Thus, we can write

$$\begin{aligned} \dot{y_1} &= y_2 \\ \dot{y_2} &= \mu(1 - y_1^2) \cdot y_2 - y_1 \end{aligned}$$

The function describing the van der Pol equation can be written as

```
function [dy] = vdp(t,y)
mu = 15;
dy = zeros(2,1);
dy(1) = y(2);
dy(2) = mu * (1-y(1)^2) * y(2) - y(1);
```

This is saved to the file `vdp.m` in the current working directory. In order to solve the differential equation, we will use the `ode45` solver. Other solvers are available, they are sometimes required by the specific differential equation being solved. In general, the `ode45` solver is a good start, but if it fails to converge or the output has discontinuities, another solver can be tried. See the `ode45` help for details. `ode45` (and all other solvers) have the following syntax `[t,y] = ode45(@fn, tspan, y0)`. Here, `fn` is the function describing the ODE, `tspan` is a vector specifying either the range of integration (if there are only two values specified) or a set of values at which the solution must be found. `y0` is a vector specifying initial conditions. The return value `t` is a vector of time instants at which the solution was found, and `y` is a vector containing the solutions at the time instants in `t`. The example below demonstrates the use of the `ode45` solver.

```
[y t] = ode45(@vdp, [0 3000], [2 0]);
figure(1);
plot(t,y(:,1));
figure(2);
plot(y(:,2), y(:,1));
```

You will notice that this program takes a very long time to execute, because the van der Pol equation is an example of a "stiff" problem, where the time increments required to compute a portion of the solution is very small, and at other times the time increment need not be so small. The `ode45` solver maintains a time increment which only decreases. Thus, the non-stiff portions of the problem will still take a long time to solve. Using one of the stiff solvers such as `ode15s` will try to increase the time increment if it detects a non-stiff region. Try solving the problem again using `ode15s`, this time it takes much less time to solve. If you enable data markers, it becomes evident that the points are not evenly spaced in time.

Note that the first figure shows the solution to the van der Pol equation (y vs. t), while the second figure shows the phase-space of the solution to the van der Pol equation ($\dot{y}$ vs y). The phase-space gives an idea of the stability of the system. Chaotic systems generally have a phase space which does not settle down to a closed path (an example is the famous Lorentz Attractor).

9 Examples of Practical Problems

An example of practical data imported and processed through Matlab is shown in Figure 4. This data was collected from the CHILL weather radar, with the antenna pointed at a 45 degree angle. The figure shows the doppler spectrum of the received signal, plotted using the `contour` function with filled in contour lines. The bright band running down the middle is echo from nearby rainfall. It can be distinguished from ground echoes because it has a doppler shift, due to random motion of the water droplets in the atmosphere. This image is courtesy of Dmitri Moisseev, a Post Doctoral candidate at the Radar Lab.

The second figure shows similar data, this time it was recorded with a much shallower antenna position. The CHILL radar is located near Greeley, the storm from which these echoes were measured was over Cheyenne. The y axis shows the range from CHILL in km, the x axis shows the doppler shift, in Hz.

References and Further Reading

1. Mastering MATLAB 6 - Duane Hanselman
2. www.engr.colostate.edu/EE192/laboratory/Lab_3_Using_Matlab.pdf - Bob Pownall
3. www.engr.colostate.edu/EE192/laboratory/Lab_4_MATLAB.pdf - Bob Pownall

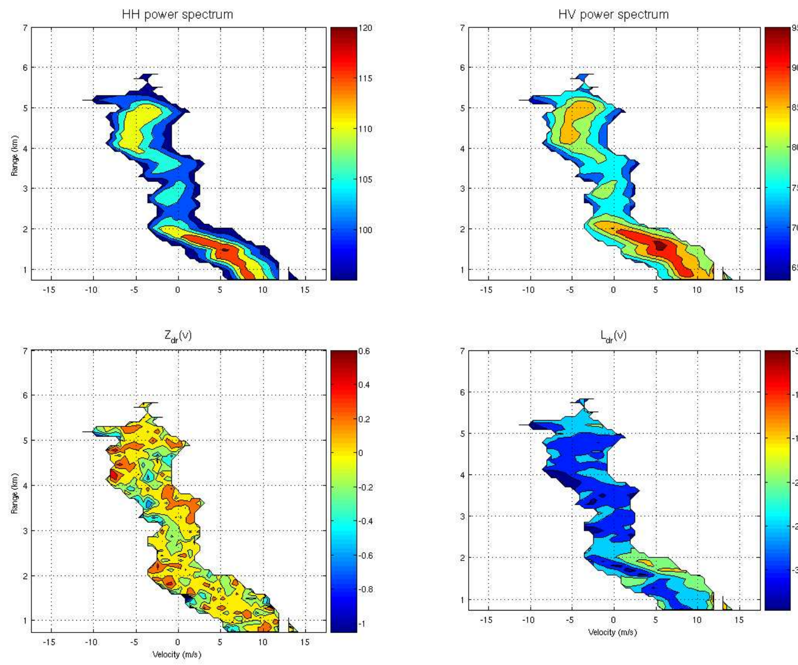

Figure 4: Velocity Distribution of Precipitation

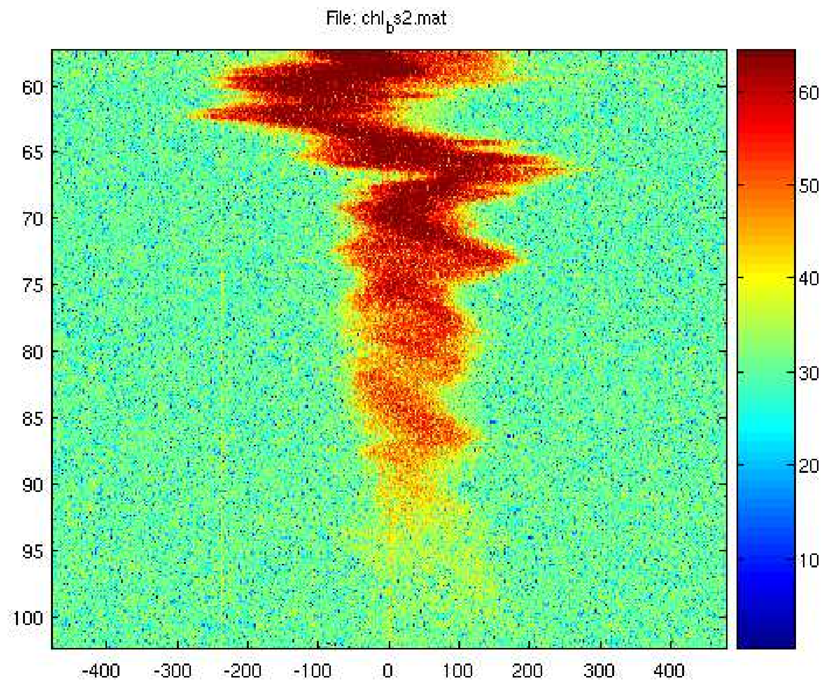

Figure 5: Velocity Distribution of Precipitation