Financial Econometrics
MFE MATLAB Notes

Kevin Sheppard
University of Oxford

September 28, 2007

Contents

Chapter 1

Introduction to MATLAB

These notes provide a brief introduction to MATLAB. All topics relevant to the MFE curriculum should be covered at some basic level but if some important topic is missing or under-explained, please let me know and I'll add examples as necessary.

This set of notes follows a few conventions. `Typewriter` font is used to denote MATLAB commands and code snippets. The double arrow symbol >> is used to indicate MATLAB input (Note: This is the symbol used in MATLAB in the command window). *Math* font is used to denote algebraic expressions.

MATLAB is available on the Economics department servers, either `xlbs.econ.ox.ac.uk` or `nlbs.econ.ox.ac.uk`, using Microsoft's Remote Desktop Client. For help using the RDC, consult the information that accompanied your username, or consult the IT help desk.

For more information on programming in MATLAB, I recommend the book *Mastering MATLAB 7* by Bruce L. Littlefield and Duane C. Hanselman (ISBN: 0131857142). It was the first book I used – back when it was MATLAB 5 – and it is comprehensive with many examples ranging from the basics to more advanced topics.

1.1 The Interface

Figure 1.1 contains an image of the main MATLAB window. There are three sub-windows visible. The command window, labeled **1**, is commands are entered, functions are called and m-files are run (MATLAB batch files). The current directory window, labeled **2**, shows the files located in the current directory. Normally these will include m-files and data. Along the bottom of this window there is a second tab labeled workspace. Clicking on workspace reveals a list of the variables in memory, for example data loaded or variables declared. The final window, labeled **3**, contains the command history where MATLAB records commands recently executed. The history can be copied and pasted into the command window to re-run commands. The history can also be scrolled through in the command window by pressing the up arrow (↑) key.

1.2 The Editor

MATLAB contains an editor which is aware of MATLAB syntax, highlights code to improve the its readability and provides limited error checking. The editor can be launched from the main window in one of two

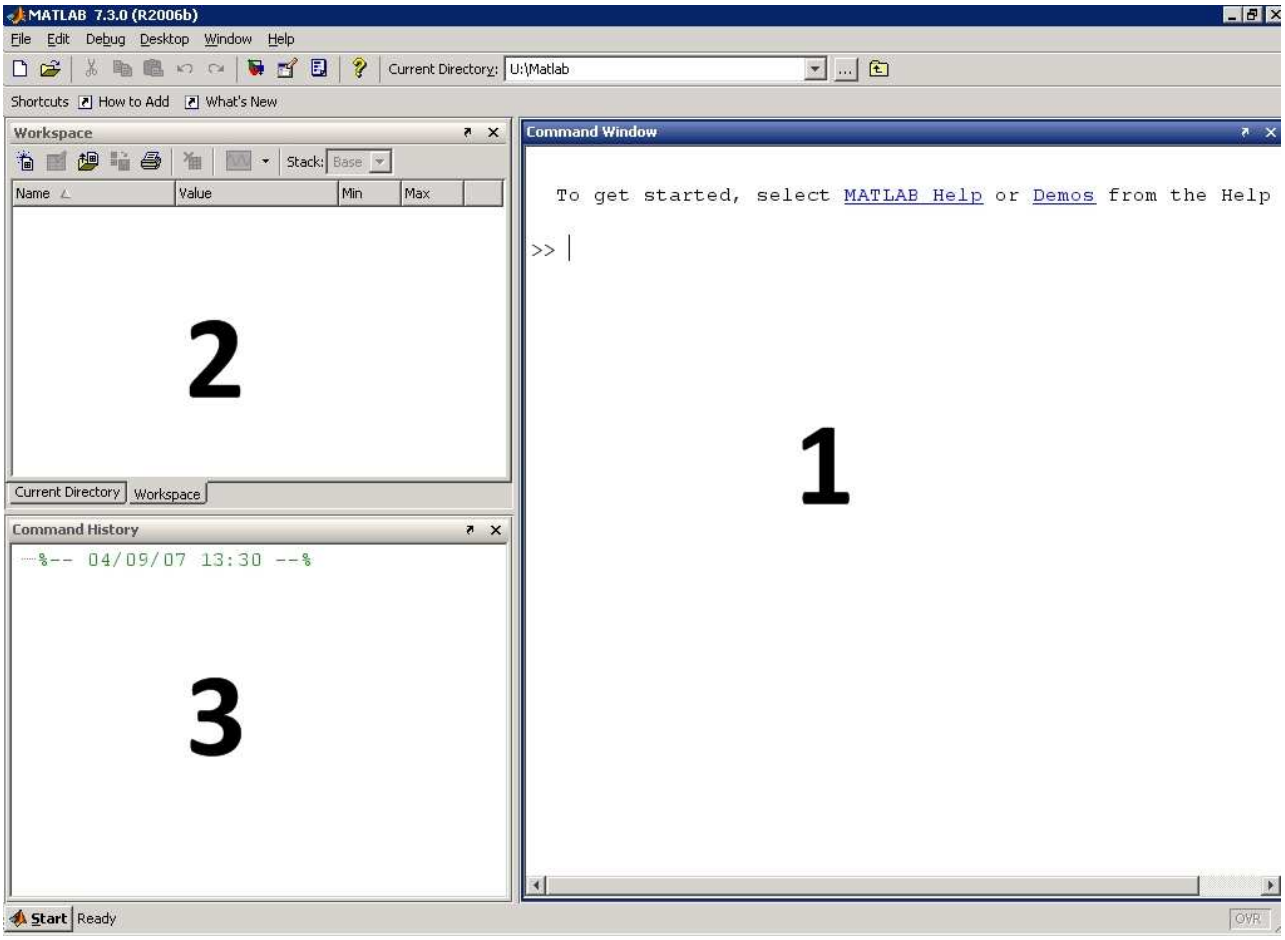


Figure 1.1: Basic MATLAB Window. The standard setup has three pains. **1**: The command window, **2**: Current Directory and Workspace, **3**: Command History

ways, either by clicking File>New>M-File or entering `edit` into the command window directly. Figure 1.2 contains an example of the editor and syntax highlighting.

M-files can contain batches of commands or complete functions. While m-file names can include letters, numbers and underscores, they must begin with a letter and it is important to avoid reserved words (`if, else, for, end, while, ...`) and existing function names (`mean, std, var, cov, sum, ...`). To verify whether a name is already in use, the MATLAB command `which` *filename* can be used to list the file MATLAB would use if *filename* was entered in the command window.

```
>> which for
built-in (C:\MATLAB\R2006b\toolbox\MATLAB\lang\for)
>> which mean
C:\MATLAB\R2006b\toolbox\MATLAB\datafun\mean.m
>> which mymfile
'mymfile' not found.
```

To check whether a files already created is using duplicating the name of another function, use the

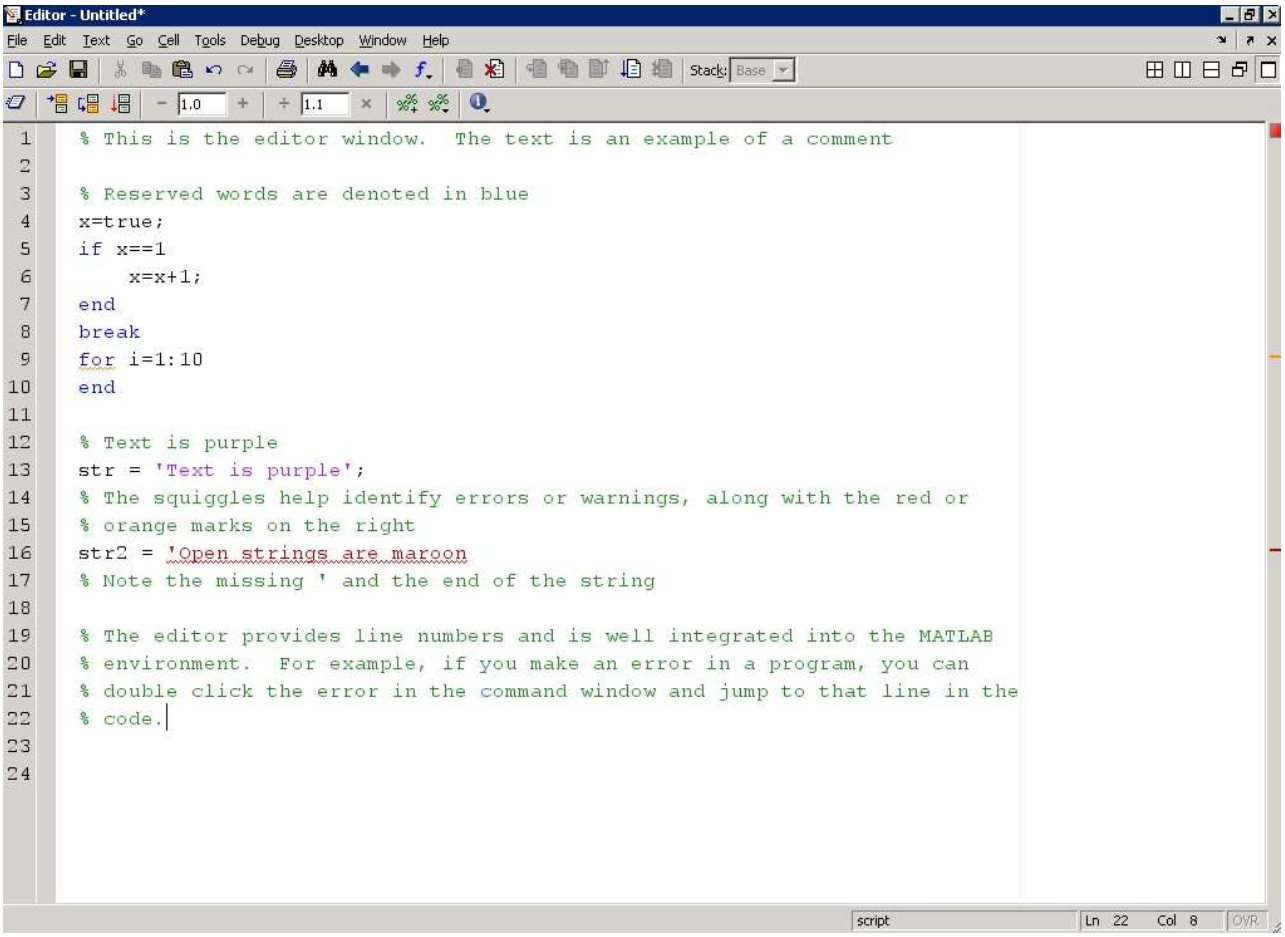


Figure 1.2: MATLAB editor. The editor is a useful tool for programming in MATLAB. It can be used to create batch files or custom functions (both called m-files). Note the syntax highlighting.

command `which` *filename* `-all` to produce a list of all files matching that name.

```
>> which mean -all
C:\MATLAB\R2006b\toolbox\MATLAB\datafun\mean.m
C:\MATLAB\R2006b\toolbox\MATLAB\timeseries\@timeseries\mean.m  % timeseries method
C:\MATLAB\R2006b\toolbox\finance\ftseries\@fints\mean.m        % fints method
```

1.2.1 ;

MATLAB uses a semicolon (`;`) at the end of a line to suppress output. The semicolon instructs MATLAB to process a line *without* returning anything to the command window. To get a feel for the effect of a `;`, examine the result of these two commands,

```
>> x=ones(3,1);
>> x=ones(3,1)

x =

     1
     1
     1
```

It is generally a good idea to suppress the output of commands, although in certain cases, such as debugging or examining the output of a particular command, it can be useful to leave them off until the code is functioning as expected.

1.2.2 Comments

Writing clear comments is an essential practice when coding. Comments assist in tracking completed tasks, documenting unique approaches to solving a difficult problem and are useful if the code needs to be shared. In MATLAB, the percentage symbol, `%`, is used to signify a comment. MATLAB will stop processing anything on that line to the right of the `%` symbol and will resume with the next line. MATLAB, unfortunately, doesn't support block comments and so any comment blocks must use a `%` in front of each line.

```
% This is the start of a
% comment block.
% Every line must have a %
% symbol before the first text
```

1.2.3 ... (dot-dot-dot)

... is a special expression that can be used to continue a long expression which is not easily expressed on a single line. ... instructs MATLAB to concatenate the next line onto the present line when processing. It exists purely to improve the readability of code.

These two expressions are identical to MATLAB:

```
x = 7;
x = x + x * x - x + exp(x) / log(x) * sqrt(2*pi);
```

```
x = 7;
x = x + x * x - x ...
    + exp(x) / log(x) * sqrt(2*pi);
```

1.3 Help

MATLAB has an extensive and thorough help system which is available both in the command window and in a separate browser. The browser-based help is generally more complete and has the added advantage that it is both indexed and searchable.

Two types of help are available from the command line: toolbox and function. Toolbox help produces a list of available functions in a toolbox. It can be called by `help` *toolbox* where *toolbox* is the MATLAB name of the toolbox (e.g. `stats`, `optim`, etc.). `help`, without a second argument, will produce a list of toolboxes. while function specific help can be accessed by calling `help` *function*, for example `help mean`.

The help browser can be accessed by hitting the F1 key, selecting Help>Full Product Family Help a the top of the command window, or entering `helpbrowser` in the command window. The documentation of a function can be jumped to directly by entering `doc` *function* in the command window (e.g. `doc mean`).

1.4 Demos

MATLAB contains an extensive selection of demos. To access the list of available demos, simply enter `demo` in the command window.

1.5 Exercises

1. Become familiar with the MATLAB Command Window.
2. Launch the help browser and read the section MATLAB, Getting Started, Introduction.
3. Launch the editor and explore its interface.
4. Enter `demo` in the command window and play with some of the demos. The demos in the Graphics section are particularly entertaining.

Chapter 2

Basic Input

MATLAB doesn't require any memory management and variables can be input with no setup. The generic form of an expression in MATLAB is

$$\textit{Variable Name} = \textit{Expression}.$$

and expressions are processes by assigning the value on the right to variables on the left. For instance,

```
x = 1;
x = y;
x = somefunction(y);
```

are all valid assignments for `x`. The first assigns 1 to `x`, the second assigns the value of another variable, `y`, to `x` and the third assigns the output of `somefunction(y)` to `x`. Assigning one variable to another assigns the *value* of that variable; not the variable itself. Thus, in the lines `y = 1;` and `x = y;`, any changes to `y` will not be reflected in the value of `x`.

```
>> y = 1;
>> x = y;
>> x
x =
     1
>> y = 2;
>> x
x =
     1
>> y
y =
     2
```

2.1 Variable Names

Variable names can take many forms, although they can only contain numbers, letters (both upper and lower), and underscores (_). They must begin with a letter and are `CaSe SeNsItIve`. For example,

```
x
X
X1
X_1
x_1
dell
dell_returns
```

are all legal and distinct variable names, while

```
x:
1X
X-1
_x
```

are not.

2.2 Entering Vectors

All data in MATLAB are matrices by construction, even if they are 1 by 1 (scalar), K by 1 or 1 by K (vectors). Vectors, both row (1 by K) and column (K by 1) can be entered directly into the command window. The mathematical notation

$$x = [1\ 2\ 3\ 4\ 5]$$

is entered into MATLAB in a natural way:

```
>> x=[1 2  3 4 5];
```

In the above input, `[` and `]` are reserved MATLAB symbols which are interpreted as *begin array* and *end array*, respectively. The column vector,

$$x = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{bmatrix}$$

has a slightly less intuitive structure in MATLAB:

```
>> x=[1; 2;  3; 4; 5];
```

When inside an array, `;` is interpreted as *new row*.

2.3 Entering Matrices

Matrices are just column vectors of row vectors. For instance, to input

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

in to MATLAB, enter the matrix one row at a time, separating the rows with a `;`:

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

2.4 Higher Dimension Arrays

MATLAB is capable of working with N dimensional arrays where N can be a very large number (up to about 30, depending on the size of each matrix dimension). Unlike scalars, vectors and matrices, higher dimension arrays can only be constructed by calling functions and cannot be directly allocated, such as `zeros(2, 2, 2)`. Higher dimensional arrays can be useful for tracking matrix valued functions over time, such as a conditional covariance.

2.5 Empty

There is one unusual matrix worth mentioning. The empty matrix is one with no elements, `x = [ ];`. Empty matrices can be returned from functions in certain cases (e.g. if some criteria is not met) and can cause problems, although have some useful applications. First, they can be used for lazy vector construction using repeated concatenation. For example

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

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

Second, they are needed for calling some functions when multiple inputs are required but do not with to specify all, for example `somefunction(x,[ ],y)`.

2.6 Concatenation

Concatenation is the process by which one vector or matrix is appended to another. Both horizontal and vertical concatenation are possible. For instance, suppose

$$x = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \text{ and } y = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix};$$

and

$$z = \begin{bmatrix} x \\ y \end{bmatrix}.$$

needs to be constructed. This can be accomplished by treating x and y as elements of a new matrix.

```
>> x=[1 2; 3 4];
>> y=[5 6; 7 8];
```

z can be defined in a natural way:

```
>> z=[x; y];
```

This is an example of vertical concatenation. x and y can be horizontally concatenated in a similar fashion:

```
>> z=[x  y];
```

Note: Concatenating is *exactly* like using block-matrix forms in standard matrix algebra.

2.7 Accessing Elements of Matrices

Once data have been entered into a vector or matrix, it is important to be able to access the elements individually. MATLAB stores matrices in a form known as *column major*. This means elements are indexed by first counting down rows and then across columns. For instance, in the matrix

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

the first element of x is 1, the second element is 4, the third is 7, the fourth is 2, and so on.

Elements can be accessed by element number using parenthesis (`x(#)`). After defining `x`, the elements of `x` can be accessed

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

x =
     1     2     3
     4     5     6
     7     8     9

>> x(1)
ans =
    1
>> x(2)
ans =
    4
>> x(3)
ans =
    7
>> x(4)
ans =
    2
>> x(5)
ans =
    5
```

The single index notations works well if `x` is a vector, in which case the indices correspond directly to the order of the elements in `x`. However, in the matrix case, single index notation is confusing. Fortunately, double indexing of matrices is available using the notation `x(#,#)`.

```
>> x(1,1)
ans =
    1
>> x(1,2)
ans =
    2
>>x(1,3)
ans =
```

```
    3
>> x(2,1)
ans =
    4
>> x(3,3)
ans =
    9
```

Higher dimension matrices can also be accessed in a similar manner, `x(#, #, #)`. For example, `x(1, 2, 3)` would return the element in the first row of the second column of the third panel of a 3-D matrix `x`.

The colon operator `(:)` plays a special role in accessing elements. When used, it is interpreted as *all elements in that dimension*. For instance, `x(:,1)`, is interpreted as all elements from matrix `x` in column 1. Similarly, `x(2,:)` is interpreted as all elements from `x` in row 2. Double `:` notation produces all elements of the original matrix; naturally, `x(:,:)` returns `x`. Finally, vectors can be used to access elements of `x`. For instance, `x([1 2],[1 2])`, will return the elements from `x` in rows 1 and 2 and columns 1 and 2, while `x([1 2],:)` will returns all columns from rows 1 and 2 of `x`.

```
>> x(1,:)
ans =
    1     2     3
>> x(2,:)
ans =
    2     5     8
>> x(:,:)
ans =
    1     2     3
    4     5     6
    7     8     9
>> x
ans =
    1     2     3
    4     5     6
    7     8     9
>> x([1 2],[1 2])
ans =
     1     2
     4     5
>> x([1 3],[2 3])
ans =
     2     3
     8     9
>> x([1 3],:)
ans =
    1     2     3
    7     8     9
```

2.8 Calling functions

Functions calls have slightly different conventions other expressions in MATLAB. The biggest difference is that functions can take more than one input and return more than one output. The generic structure of a

function call is `[`*out1*, *out2*, *out3*, ...`]=function(`*in1*, *in2*, *in3*, ...`)`. The important aspects of this structure are

- If only one output is needed, brackets ([ ]) are optional, for example `y=mean(x)`.
- If multiple outputs are required, the outputs **must** be encapsulated in brackets, such as in `[y, index] = min(x)`.
- The number of output variables determines how many outputs will be returned. Asking for more outputs than the function provides will generally result in an error.
- Both inputs and outputs must be separated by commas (,)
- Inputs can be the result of other functions as long as only the first output is requires (e.g. `mean(var(x))`).
- Inputs can contain only selected elements of a matrix or vector (e.g. `mean(x([1 2] ,[1 2]))`).

The usage of function calls will be clarified as they are described throughout the notes.

2.9 Exercises

1. Enter the following mathematical expressions into MATLAB:

$$u = [1\ 1\ 2\ 3\ 5\ 8]$$

$$v = \begin{bmatrix} 1 \\ 1 \\ 2 \\ 3 \\ 5 \\ 8 \end{bmatrix}$$

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

$$y = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

$$z = \begin{bmatrix} 1 & 2 & 1 & 2 \\ 3 & 4 & 3 & 4 \\ 1 & 2 & 1 & 2 \end{bmatrix}$$

$$w = \begin{bmatrix} x & x \\ y & y \end{bmatrix}$$

2. What command would pull x would of w? (Hint: `w([?],[?])` is the same as x.)

3. What command would pull `[x; y]` out of `w`? Is there more than one? If there are, list all alternatives.
4. What command would pull y out of z? List all alternatives.

Chapter 3

Entering and Saving Data

The first *real* challenge is getting data into and out of MATLAB.

3.1 Getting Data Into MATLAB

Getting data into MATLAB ranges from moderate to very difficult, depending on the data. First, a few general pointers:

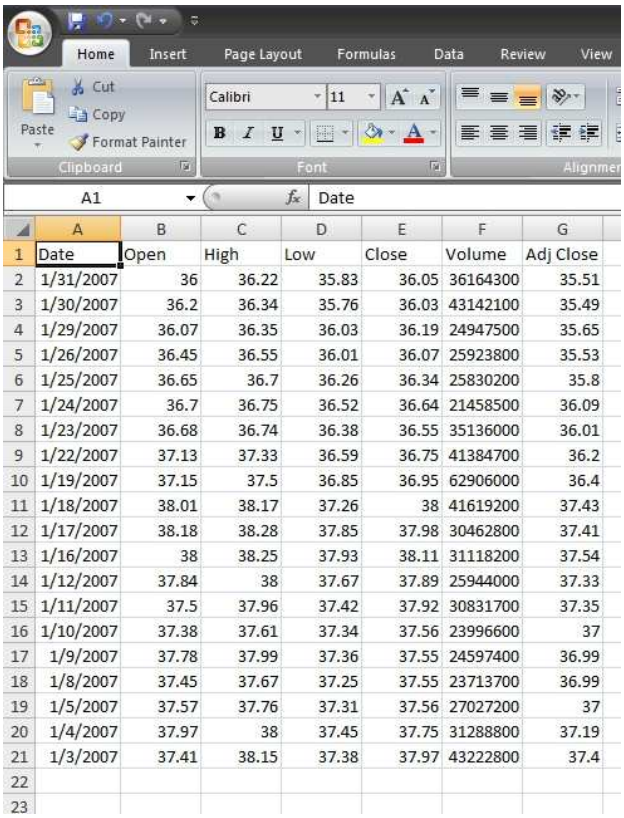
- The file imported should contain numbers *only*, with the exception of the first row which should contain the variable name.
- Use another program, such as Microsoft Excel, to manipulate the data before importing.
- Each column of the spreadsheet should contain a single variable.
- Dates should be imported as numbers by first formatting the columns as Excel Dates and then reformatting as a number (dates with base year 1900). For example, January 1, 2000 would be 36526.

3.2 Robust Data Importing

The simplest and most robust method to import data into MATLAB is to use a correctly formatted Excel file and the import wizard. The key to the import is to make certain the Excel file has a very particular structure:

- One variable per column
- The variable name for the column in the first position
- *All* other data in the column must be numeric, especially dates.

As an example, consider importing a month of GE prices downloaded from Yahoo! Finance historical prices. The original data can be found in `GEPrices.xls` and is presented in figure 3.1. This data file is nearly fits the requirement although the first column, containing the dates, falls short. To prepare this data for import, only the date, close and volume columns were chosen. The key step is to convert the dates from Excel dates



	A	B	C	D	E	F	G
1	Date	Open	High	Low	Close	Volume	Adj Close
2	1/31/2007	36	36.22	35.83	36.05	36164300	35.51
3	1/30/2007	36.2	36.34	35.76	36.03	43142100	35.49
4	1/29/2007	36.07	36.35	36.03	36.19	24947500	35.65
5	1/26/2007	36.45	36.55	36.01	36.07	25923800	35.53
6	1/25/2007	36.65	36.7	36.26	36.34	25830200	35.8
7	1/24/2007	36.7	36.75	36.52	36.64	21458500	36.09
8	1/23/2007	36.68	36.74	36.38	36.55	35136000	36.01
9	1/22/2007	37.13	37.33	36.59	36.75	41384700	36.2
10	1/19/2007	37.15	37.5	36.85	36.95	62906000	36.4
11	1/18/2007	38.01	38.17	37.26	38	41619200	37.43
12	1/17/2007	38.18	38.28	37.85	37.98	30462800	37.41
13	1/16/2007	38	38.25	37.93	38.11	31118200	37.54
14	1/12/2007	37.84	38	37.67	37.89	25944000	37.33
15	1/11/2007	37.5	37.96	37.42	37.92	30831700	37.35
16	1/10/2007	37.38	37.61	37.34	37.56	23996600	37
17	1/9/2007	37.78	37.99	37.36	37.55	24597400	36.99
18	1/8/2007	37.45	37.67	37.25	37.55	23713700	36.99
19	1/5/2007	37.57	37.76	37.31	37.56	27027200	37
20	1/4/2007	37.97	38	37.45	37.75	31288800	37.19
21	1/3/2007	37.41	38.15	37.38	37.97	43222800	37.4
22							
23							

Figure 3.1: The raw data as taken from Yahoo! Finance historical prices. Most of the columns are well formatted with variable names in the first row and numeric content. However the date column contains Excel dates and not numbers. This structure will prevent MATLAB from correctly parsing the excel file.

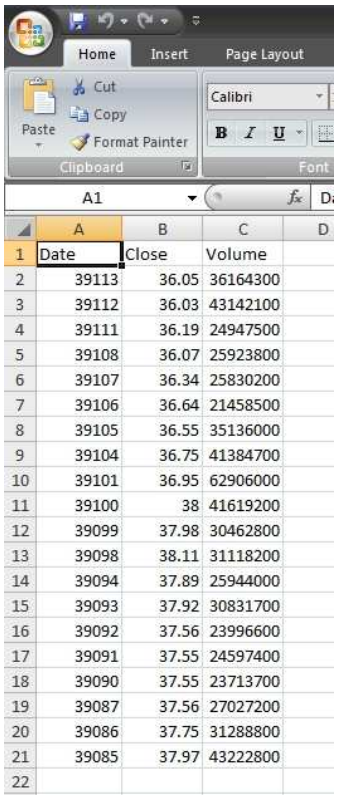
to numbers. To perform the conversion, select the dates, right click and choose format. Select "Number" from the dialog box the pops up. If the conversion was performed correctly, the output should be similar to figure 3.2. This "clean" file can be found in `GEPricesClean.xls`.

Once the excel file has been formatted, the final step is to import it. First, change the Current Directory to the directory with the Excel file to be imported. Next, select the Current Directory browser in the upper left pane of the main window.[1] The Excel file should be present in this view. To import the file, right click on the filename and select Import (see figure 3.3). This will trigger the dialog in figure 3.4. To complete the import, make sure **Create vectors from each column using column names** is chosen and click finish. If the import fails the most likely cause is the format of the Excel file. Make certain this conforms to the rules above.

3.3 Reading Excel Files

Data in excel sheets can be imported using the function `xlsread` from the command window. Accompanying this set of notes is an excel file, *deciles.xls* which contains returns for the 10 CRSP deciles from January

[1] If this pane is absent, it can be enabled in the `Desktop` tab along the top of the MATLAB window.



	A	B	C	D
1	Date	Close	Volume	
2	39113	36.05	36164300	
3	39112	36.03	43142100	
4	39111	36.19	24947500	
5	39108	36.07	25923800	
6	39107	36.34	25830200	
7	39106	36.64	21458500	
8	39105	36.55	35136000	
9	39104	36.75	41384700	
10	39101	36.95	62906000	
11	39100	38	41619200	
12	39099	37.98	30462800	
13	39098	38.11	31118200	
14	39094	37.89	25944000	
15	39093	37.92	30831700	
16	39092	37.56	23996600	
17	39091	37.55	24597400	
18	39090	37.55	23713700	
19	39087	37.56	27027200	
20	39086	37.75	31288800	
21	39085	37.97	43222800	
22				

Figure 3.2: This correctly formatted file contains only the variables to import: date, close and volume. Note that the date column has been converted from Excel date to a number so January 3, 2007 appears as 39085.

1, 2000 to December 31, 2004. The first column contains the dates while columns 2 through 11 contain the portfolio returns from decile 1 through decile 10 respectively. To load the data into MATLAB, use the command

```
>> data = xlsread('deciles.xls');}
```

This command will read the data in *sheet1* of file *deciles.xls* and assign it to `data` in MATLAB. `xlsread` can handle a number of other situations, including reading sheets other than *sheet1* or reading only specific blocks of cells. For more information, see `help xlsread`. Data can be exported to an Excel file using `xlswrite`. Extended information about an excel file, such as sheet names and can be read using the command `xlsflinfo`.

Note: MATLAB and Excel do not agree about dates. MATLAB tracks dates as days past January 1, 0000 (inclusive) while Excel tracks dates as day past January 1, 1900. Thus, to convert imported Excel dates into MATLAB dates, `datenum('30DEC1899')` must be added to the column of data representing the dates. Returning to the example above,

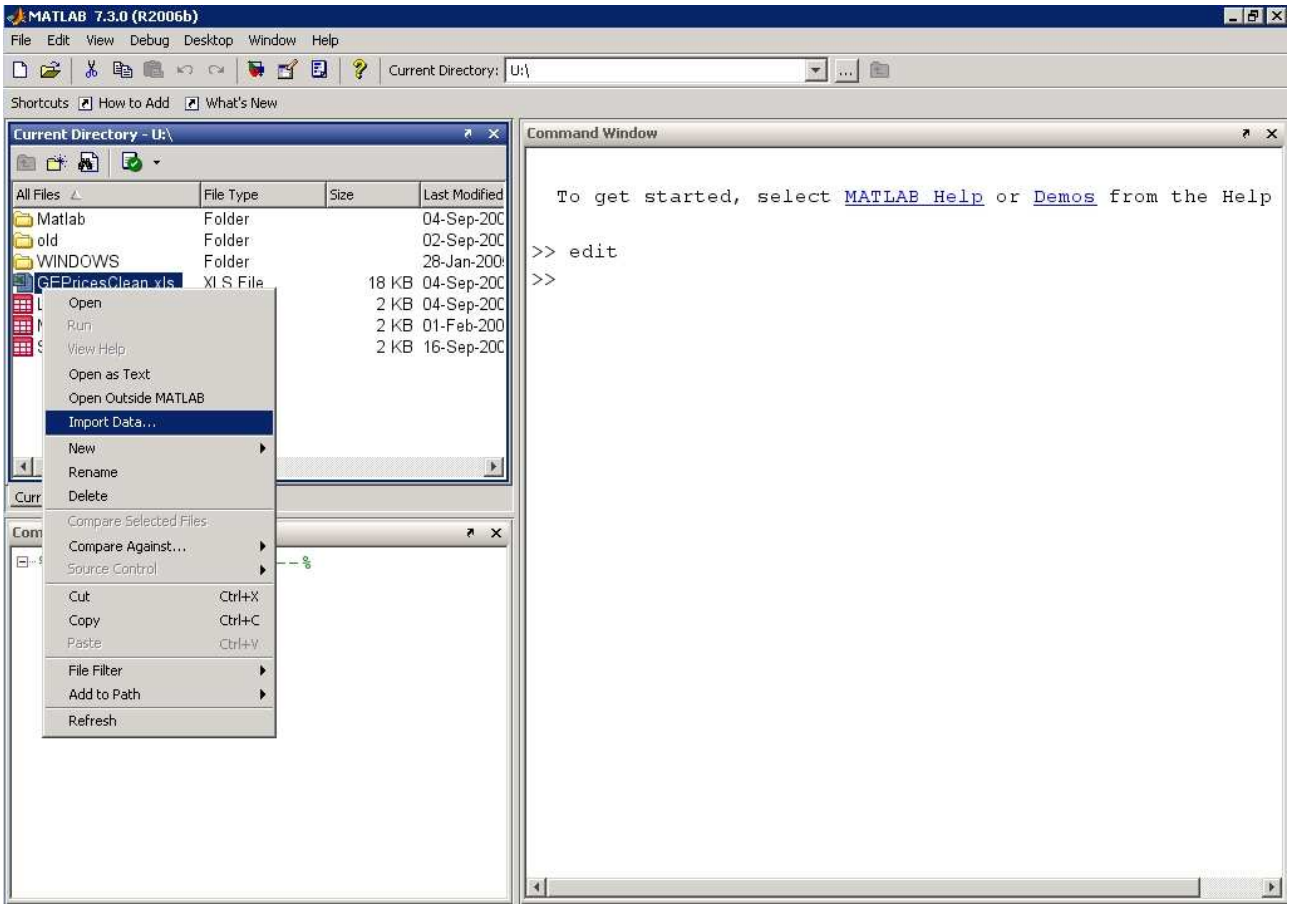


Figure 3.3: To import data, select the Current Directory view, right click on the Excel file to be imported, and select Import. This will trigger the import wizard in figure 3.4.

```
>> [A,finfo]=xlsfinfo('deciles2.xls')
A =
Microsoft Excel Spreadsheet
finfo =
    'Cleaned Data'    'Original Data'
>> data  = xlsread('deciles2.xls','Cleaned Data','A2:K1257');
>> dates = data(:,1);
>> datestr(dates(1))
ans =
03-Jan-0100
>> dates = dates + datenum('30DEC1899');
>> datestr(dates(1))
ans =
03-Jan-2000
```

Alternatively, the function `x2mdate` can be used to convert the dates.

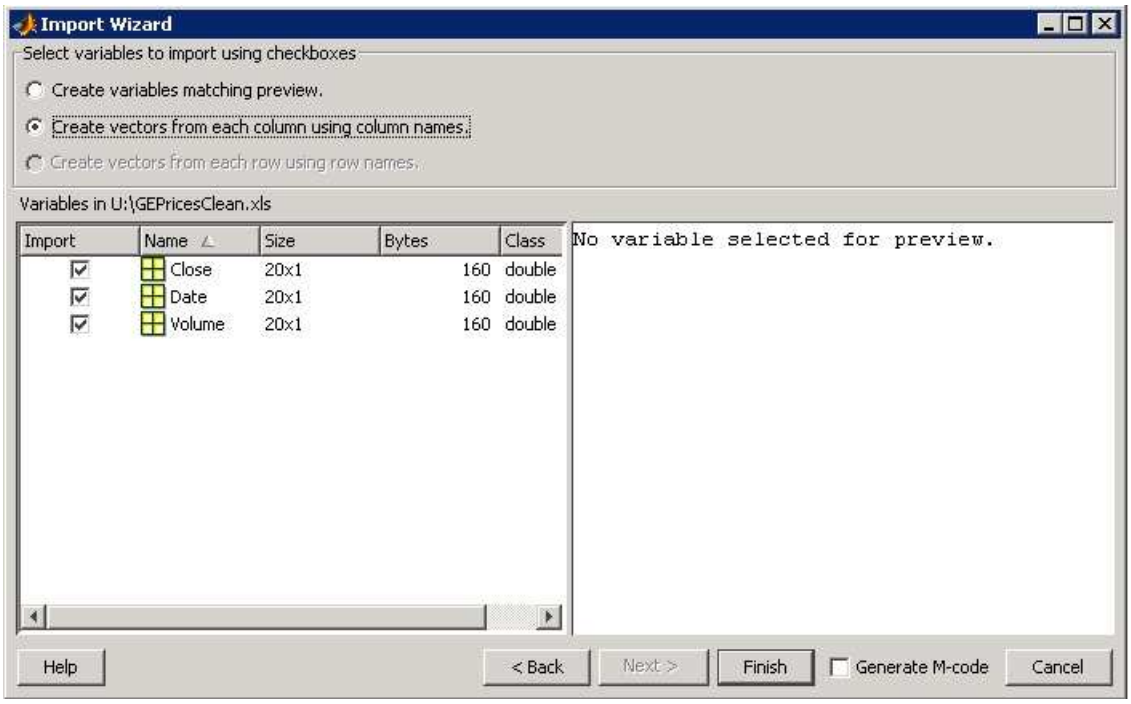


Figure 3.4: As long as the data is correctly formatted (see figure 3.2), the import wizard should pull the data into MATLAB and create variable with the same name as the column headers. To complete this step, make sure the second radio button is selected (Create vectors from each column using column names) and then select Finish.

```
>> data  = xlsread('deciles2.xls','Cleaned Data','A2:K1257');
>> dates = data(:,1);
>> datestr(dates(1))
ans =
03-Jan-0100
>> dates = x2mdate(dates);
>> datestr(dates(1))
ans =
03-Jan-2000
```

This example uses a files *deciles2.xls* which contains two sheets, *Original Data* and *Cleaned Data*. Opening the files in Excel shows that *Cleaned Data* contains column labels as well as the data. To import data from this file, `xlsread` needs to know to pull data from *Cleaned Data* from cells *A2:K1275* (upper left and lower right corners of block). `xlsread('deciles2.xls', 'Cleaned Data', 'A2:K1257')` does exactly this. Finally, the dates disagreement is illustrated and the correction is shown to work. For more on dates, see the Chapter 17 on date manipulation.

3.4 CSV Data

CSV data, or comma-separated values, is much like Excel data. Note that CSV files *must not* contain anything but numeric values. If the file contains strings, such as variable names, the import will fail. To over come this limitation, use the Import Wizard as described in Section 3.2. The command to read CSV data is essentially identical,

```
>> data = csvread('deciles.csv');
```

Like xlsread, other forms can also begin reading at a specific cell

```
>> data = csvread('deciles.csv',0,1);
```

or to read specific blocks of cells

```
>> data = csvread('deciles.csv',1,0,[1 0 1256 10]);
```

Data can be exported to csv using `csvwrite`.

3.5 Text

Reading in text, if it only contains numbers, is also fairly straight forward. The standard command is `textread` and is called identically to `xlsread`,

```
data = textread('deciles.txt');
```

`textread` can handle a variety of data formats, but it is recommend to keep data files as simple as possible and to only use tab delimited text files (like the example *deciles.txt*). See `help textread` for further information.

3.6 MATLAB Data Files (.mat)

The native format for MATLAB data are known as MATLAB data files, or .mat files. These are the easiest to work with and data is loaded simply by entering

```
load deciles.mat
```

There is no need to specify an input variable as the .mat file contains both variable names and data. See below for saving data in .mat format.

3.7 Reading Poorly Formatted Text

MATLAB can be convinced to read just about any text format. MATLAB has functions for reading arbitrary text which can then be converted to numbers. This is an advanced technique and should be avoided if possible. However, there are some situations where this is the only viable alternative. For instance, suppose the raw data is in a very large file (too large for Excel) and is poorly formatted. In this case, the simplest procedure is to write a program to read the file line by line and process each line separately.

The file *IBM_TAQ.txt* contains a simple example of data that is difficult to get into MATLAB. This file was downloaded from WRDS and contains all prices for IBM from the TAQ database in the interval January 1,2001 through January 31, 2001. It has too many lines to use in Excel and has both numbers, dates and text on each line. The following code block shown how the data in this file can be parsed using MATLAB:

```
fid=fopen('IBM_TAQ.csv','rt');
%Count number of lines
count=0;
while 1
    line=fgetl(fid);
    if ~ischar(line)
        break
    end
    count=count+1;
end
%Close the file
fclose(fid);

%Pre-allocate the data
dates = zeros(count-1,1);
time  = zeros(count-1,1);
price = zeros(count-1,1);
%Reopen the file
fid=fopen('IBM_TAQ.csv','rt');
%Get one line to throw away since it contains the column labels
line=fgetl(fid);
%Use count to index the lines this pass
count=1;
%while 1 and break work well when reading test
while 1
    line=fgetl(fid);
    %If the line is not a character value we've reached the end of the file
    if ~ischar(line)
        break
    end
    %Find all the commas, they delimit the file
    commas = strfind(line,',');
```

```
    %Dates are places between the first and second commas
    dates(count)=datenum(line(commas(1)+1:commas(2)-1),'yyyymmdd');
    %Times are between the second and third
    temptime=line(commas(2)+1:commas(3)-1);
    %Times are colon separates, so they need further parsing
    colons=strfind(temptime,':');
    %Convert the text representing the hours, minutes or and seconds to numbers
    hour=str2double(temptime(1:colons(1)-1));
    minute=str2double(temptime(colons(1)+1:colons(2)-1));
    second=str2double(temptime(colons(2)+1:length(temptime)));
    %Convert these values to seconds past midnight
    time(count)=hour*3600+minute*60+second;
    %Read the price from the last comma to the end of the line and convert to number
    price(count)=str2double(line(commas(3)+1:commas(4)-1));
    %Increment the count
    count=count+1;
end
fclose(fid);
```

This block of code does a few thing:

- Open the file directly using `fopen`
- Reads the file line by line using `fgetl`
- Counts the number of lines in the file
- Pre-allocates the `dates`, `times` and `price` variables using `zeros`
- Rereads the file parsing each line by the location of the commas using `strfind` to locate the delimiting character
- Uses `datenum` to convert string dates to numerical dates
- Uses `str2double` to convert strings to numbers

To read poorly formatted data file, see the documentation for `fopen`, `fscanf`, `fread`, `fgetl`, `dlmread`, and `textscan` and consult Chapter 18 on basic string manipulation.

3.8 Stat Transfer

There is one final method worth mentioning to import data. *StatTransfer* is available on the servers and is capable of reading and writing approximately 20 different formats, including MATLAB, GAUSS, Stata, SAS, Excel, CSV and text files. It allow users to load data in one format and output some or all of it in another. *StatTransfer* can make some hard-to-manage situations (e.g. poorly formatted data) substantially easier. *StatTransfer* has a comprehensive help file to provide assistance.

3.9 Getting Data Out of MATLAB

3.9.1 Saving Data

Once the data has been loaded into MATLAB, save it and any changes in the native MATLAB data format. This is easily accomplished by calling

```
>> save filename
```

This will produce a file *filename.mat* containing all variables in memory. *filename* can be replaced with any valid filename. To save a subset of those variables in memory, entering

```
>> save filename var1 var2 var3
```

which produces a file *filename.mat* containing `var1`, `var2` and `var3` .

3.9.2 Exporting Data

One easy method to get data out of MATLAB is to call `save` with the arguments `-double -ascii`. This will produce a tab delimited file of the variables listed. It is generally a good practice to only export one variable at a time using this method. Exporting more than one results in a poorly formatted file that may be hard to import into another program. For example,

```
>> save filename var1 -ascii -double
```

would save `var1` in a tab delimited text file. The restriction to a single variable should not be seen as a severe limitation as another variable, `var1`, can always be constructed from other variables (e.g. `var1=[var2 var3];`. Alternative methods to export data include `xlswrite`, `csvwrite` and `dlmwrite`.

3.10 Exercises

1. The file *exercise3.xls* contains three columns of data, the date, the return on the S&P 500, and the return on XOM (Exxon Mobil). Using Excel, convert the date to a number and save the file. (Hint: Format the cells with dates as numbers. They should be 30000ish).
2. Use `xlsread` to read the file saved in the previous exercise. Load in the three series into a new variable names `returns`.
3. Parse `returns` into three variables, `dates`, `SP500` and `XOM`. (Hint, use the : operator).
4. Save a MATLAB data file *exercise3* with all three variables.
5. Save a MATLAB data file *dates* with only the variable `dates`.

6. Construct a new variable, `sum_returns` as the sum of `SP500` and `XOM`. Create another new variable, `output_data` as a horizontal concatenation of `dates` and `sum_returns`.
7. Export the variable `output_data` to a new .xls file using `xlswrite`. See the help available for `xlswrite`.

Chapter 4

Basic Math

Math in MATLAB closely follows the rules of linear algebra. Anything that can be done in linear algebra can be done in MATLAB; most thing that aren't allowed in linear algebra aren't allowed in MATLAB. For instance, to multiply two matrices together, they must conform along their inside dimensions; attempting to multiple nonconforming matrices produces an error.

4.1 Operators

MATLAB has the standard operators:

Operator	Meaning	Example	Algebraic
+	Addition	`x + y`	$x + y$
-	Subtraction	`x - y`	$x - y$
*	Multiplication	`x * y`	xy
/	Division (Left divide)	`x / y`	$\frac{x}{y}$
\	Right divide	`x \ y`	$\frac{y}{x}$
^	Exponentiation	`x ^ y`	x^y

When `x` and `y` are scalars, the behavior of these operators is obvious. When `x` and `y` are matrices, things are a bit more complex.

4.2 Matrix Addition (+) and Subtraction (-)

Addition and substraction require `x` and `y` to have the same dimensions *or* to be scalar. If they are both matrices, `z=x+y` produces a matrix with `z(i,j)=x(i,j)+y(i,j)`. If `x` is scalar and `y` is a matrix, `z=x+y` results in `z(i,j)=x+y(i,j)`.

Suppose `z=x+y`:

x \ y	Scalar	Matrix
Scalar	Any $z = x + y$	Any $z_{ij} = x + y_{ij}$
Matrix	Any $z_{ij} = y + x_{ij}$	Both Dimensions Match $z_{ij} = x_{ij} + y_{ij}$

Note: These conform to the standard rules of matrix addition and substraction.

4.3 Matrix Multiplication (∗)

Multiplication requires the inside dimensions to be the same or for one input to be scalar. If x is N by M and y is K by L and both are non-scalar matrices, x∗y requires $M = K$. Similarly, y∗x requires $L = N$. If x is scalar and y is a matrix, then z=x∗y produces z(i,j)=x∗y(i,j).
Suppose z=x∗y:

x \ y	Scalar	Matrix
Scalar	Any $z = xy$	Any $z_{ij} = xy_{ij}$
Matrix	Any $z_{ij} = yx_{ij}$	Inside Dimensions Match $z_{ij} = \sum_{k=1}^{M} x_{ik}y_{kj}$

Note: These conform to the standard rules of matrix multiplication. x_i is row i of x and y_j is column j of y.

4.4 Matrix Division (/)

Matrix division is not generally defined in linear algebra and its use is slightly tricker. The intuition for matrix division comes from thinking about a set of linear equations. Suppose there is some z, a M by L vector, such that

$$yz = x$$

where x is N by M and y is N by L. Division finds z as the solution to the linear equations by least squares, and so $z = (y'y)^{-1}(y'x)$.
Suppose z=x/y:

x \ y	Scalar	Matrix
Scalar	Any $z = \frac{x}{y}$	Left Dimensions Match $z = (y'y)^{-1}y'x$
Matrix	Any $z_{ij} = \frac{x_{ij}}{y}$	Left Dimensions Match $z = (y'y)^{-1}y'x$

Note: Like linear regression, matrix division is only well defined if y is nonsingular and thus has full rank.

4.5 Matrix Right Divide (\)

Matrix right division is simply the opposite of matrix division.
Suppose z=x\y:

		y: Scalar	y: Matrix
x	Scalar	Any $z = \frac{y}{x}$	Any $z_{ij} = \frac{y_{ij}}{x}$
	Matrix	Right Dimensions Match $z = (x'x)^{-1}x'y$	Right Dimensions Match $z = (x'x)^{-1}x'y$

Note: Like linear regression, matrix division is only well defined if x is nonsingular.

4.6 Matrix Exponentiation (^)

Matrix exponentiation is only defined if at least one of x or y are scalars.
Suppose z=x^y:

		y: Scalar	y: Matrix
x	Scalar	Any $z = x^y$	y Square Strange, Do Not Use [1]
	Matrix	x Square $z = x^y$	N/A

Note: In the case where x is a matrix and y is an integer, and z=x*x*...*x (y times). If y is not integer, this function involves eigenvalues (see `help mpower`).

4.7 Parentheses

Parentheses can be used in the usual way to control the order mathematical expressions are evaluated. Parentheses can be nested to create complex expressions. See Operator Precedence for more information on the order MATLAB evaluates mathematical expressions.

4.8 . operator

The . operator (read dot operator) changes usual operations into element by element operations. For instance, suppose x and y are N by N matrices. z=x*y results in usual matrix multiplication where `z(i,j) = x(i,:) * y(:,j)`, while `z = x .* y` produces z where `z(i,j) = x(i,j) * y(i,j)`. Multiplication (.*), division (./), right division (.\), and exponentiation (.^) all have "." forms.

```
z=x.*y    z(i,j)=x(i,j)*y(i,j)
z=x./y    z(i,j)=x(i,j)/y(i,j)
z=x.\ y   z(i,j)=x(i,j)\y(i,j)
z=x.^y    z(i,j)=x(i,j)^y(i,j)
```

Note: These are sometimes called the Hadamard operators, especially `.*`.

4.9 Transpose

Matrix transpose is available MATLAB, and is expressed using the `'` operator. For instance, is `x` is an M by N matrix, `x'` is it's transpose with dimensions N by M.

4.10 Operator Precedence

Computer math, like standard math, has operator precedence. This determines how mathematical expressions like

```
2^3+3^2/7*13
```

are evaluated. The order of evaluation is:

Operator	Name	Rank
( )	Parentheses	1
', ^, .^	Transpose, All Exponentiation	2
~	Negation (Logical)	3
+,−	Unary Plus, Unary Minus	3
, ., / , ./ , \, .\	All multiplication and division	4
+, -	Addition and subtraction	5
:	Colon Operator	6
<, <=, >, >=, ==, =	Logical operators	7
&	Element-by-Element AND	8
|	Element-by-Element OR	9
&&	Short Circuit AND	10
||	Short Circuit OR	11

In the case of a tie, operations are executed left-to-right. For example, `x^y^z` is interpreted as `(x^y)^z`.

Note: Unary operators are + or - operations that apply to a single element. For example, consider the expression `(-4)`. This is an instance of a unary - since there is only 1 operation. `(-4)^2` produces 16. However, `-4^2` produces `-16` since MATLAB interprets this as `-(4^2)` since - is no longer unary.

4.11 Exercises

1. Using the matrices entered in exercise 1 of chapter 2, compute the values of $u + v'$, $v + u'$, vu, uv and xy

2. Is `x\ 1` legal? If not, why not. What about `x/1`?

3. Compute the values `(x+y)^2` and `x^2+x*y+y*x+y^2`. Are they the same?

4. Is `x^2+2*x*y+y^2` the same as either above?

5. When will `x^y` and `x.^y` be the same?

6. Is a*b+a*c the same as a*b+c? If so, show it, if not, how can the second be changed so they are equal.

7. Suppose a command `x^y*w+z` was entered. What restrictions on the dimensions of `w`, `x`, `y` and `x` must be true for this to be a valid statement?

8. What is the value of `-2^4`? What about `(-2)^4`?

Chapter 5

Basic Functions

This section provides a reference for a set commonly used functions with a discussion of how they behave.

`length`

To find the size of the maximum dimension of `x`, use `z=length(x)`. Note: If y is T by K, $T > K$, $z = T$. If $K > T$, $z = K$. `length` is a risky command because the value it returns can be the number of columns or the number of rows, depending on which is larger. It is better practice to use `size(y,1)` and `size(y,2)` depending on whether the number of rows of the number of columns is required.

```
>> x=[1 2 3; 4 5 6]
x =
     1     2     3
     4     5     6
>> length(x)
ans =
     3
>> length(x')
ans =
     3
```

`size`

To find the size of a dimension of a matrix, use `z=size(x,`*DIM*`)`, where *DIM* is the dimension. Note that dimension 1 is the number of rows while dimension 2 is the number of columns, so if x is T by K, `z=size(x,1)` returns T while `z=size(x,2)` returns K. Alternatively, `s=size(x)` returns a vector `s` with the size of each dimension.

```
>> x=[1 2 3; 4 5 6]
x =
     1     2     3
```

```
     4     5     6
 >> size(x,1)
 ans =
     2
 >> size(x,2)
 ans =
     3
 >> s=size(x)
 s =
     2     3
```

sum

To compute the sum matrix,

$$z = \sum_{t=1}^{T} x_t$$

use the command `sum(x)`. `z=sum(x)` returns a K by 1 vector of the sum of each column, so `z(i) = sum(x(:,i)) = x(1,i) + x(2,i) + ... + x(T,i)`. Note: If `x` is a vector, `sum` will add all elements of `x` whether it is a row ro column vector.

```
>> x=[1 2 3; 4 5 6]
x =
     1     2     3
     4     5     6
>> sum(x)
ans =
     5     7     9
>> sum(x')
ans =
     6    15
```

min

To find the minimum element of a vector or the rows of a matrix,

$$\min x_{it},\ i = 1, 2, \ldots, K$$

use the command `min(x)`. If `x` is a vector, `min(x)` is scalar. If `x` is a matrix, `min(x)` is a K by 1 vector of the minimum values of each column.

```
>> x=[1 2 3; 4 5 6]
x =
     1     2     3
```

```
     4     5     6
>> min(x)
ans =
     1     2     3
>> min(x')
ans =
     1     4
```

max

To find the maximum element of a vector or the rows of a matrix,

$$\max x_{it},\ i = 1, 2, \ldots, K$$

use the command `max(x)`. If `x` is a vector, `max(x)` is scalar. If `x` is a matrix, `max(x)` is a K by 1 vector of the maximum values of each column.

sort

To sort the values of a vector or the rows of a matrix from smallest to largest, use the command `sort(x)`. If `x` is a vector, `sort(x)` is vector where `x(1)=min(x)` and `x(i)`$\leq$`x(i+1)`. If `x` is a matrix, `sort(x)` is a matrix of the same size where each column is sorted from smallest to largest.

```
>> x=[1 2 3; 4 5 6]
x =
     1     2     3
     4     5     6
>> sort(x)
ans =
     1     2     3
     4     5     6
>> sort(x')
ans =
     1     4
     2     5
     3     6
```

exp

To take the exponential of a vector or matrix (element by element),

$$e^x$$

use `exp`. `z=exp(x)` returns a vector or matrix the same size as `x` where `z(i,j)=exp(x(i,j))`.

```
>> x=[1 2 3; 4 5 6]
x =
     1     2     3
     4     5     6
>> exp(x)
ans =
    2.7183    7.3891   20.0855
   54.5982  148.4132  403.4288
```

log

To take the natural logarithm of a vector or matrix,

$$\log x$$

use `log`. `z=log(x)` returns a vector or matrix the same size as `x` where `z(i,j)=log(x(i,j))`.

```
>> x=[1 2 3; 4 5 6]
x =
     1     2     3
     4     5     6
>> log(x)
ans =
         0    0.6931    1.0986
    1.3863    1.6094    1.7918
```

sqrt

To compute the element-by-element square root of a vector or matrix,

$$\sqrt{x_{ij}}$$

use `sqrt`. `z=sqrt(x)` returns a vector or matrix the same size as `x` where `z(i,j)=sqrt(x(i,j))`.

```
>> x=[1 2 3; 4 5 6]
x =
     1     2     3
     4     5     6
>> sqrt(x)
ans =
    1.0000    1.4142    1.7321
    2.0000    2.2361    2.4495
```

Note: This command produces the same result as `z=x.^(1/2)`.

mean

To compute the mean of a vector or matrix,

$$z = \frac{\sum_{t=1}^{T} x_t}{T}$$

use the command `mean(x)`. `z=mean(x)` is a K by 1 vector containing the means of each column, so `z(i) = sum(x(i,:)) / length(x(i,:))`. Note: If `x` is a vector, `mean` will compute the mean of `x` whether it is a row ro column vector.

```
>> x=[1 2 3; 4 5 6]
x =
     1     2     3
     4     5     6
>> mean(x)
ans =
    2.5000    3.5000    4.5000
>> mean(x')
ans =
     2     5
```

var

To compute the sample variance of a vector or matrix,

$$\hat{\sigma}^2 = \frac{\sum_{t=1}^{T}(x_t - \bar{x})^2}{T-1}$$

use the command `var(x)`. If `x` is a vector, `var(x)` is scalar. If `x` is a matrix, `var(x)` is a K by 1 vector containing the sample variances of each column. Note: This command uses $T-1$ in the denominator unless an optional second argument is used.

```
>> x=[1 2 3; 4 5 6]
x =
     1     2     3
     4     5     6
>> var(x)
ans =
    4.5000    4.5000    4.5000
>> var(x')
ans =
     1     1
```

cov

To compute the sample covariance of a vector or matrix

$$\hat{\Sigma} = \frac{\sum_{t=1}^{T}(\mathbf{x}_t - \bar{\mathbf{x}})'(\mathbf{x}_t - \bar{\mathbf{x}})}{T-1}$$

use the command `cov(x)`. If `x` is a vector, `cov(x)` is scalar (and is identical of `var(x)`). If `x` is a matrix, `cov(x)` is a K by K matrix with sample variances on the diagonals and sample covariances on the off diagonals. Note: This command uses $T-1$ in the denominator unless an optional second argument is used.

```
x =
     1     2     3
     4     5     6
>> cov(x)
ans =
    4.5000    4.5000    4.5000
    4.5000    4.5000    4.5000
    4.5000    4.5000    4.5000
>> cov(x')
ans =
     1     1
     1     1
```

std

To compute the sample standard deviation of a vector or matrix,

$$\hat{\sigma} = \sqrt{\frac{\sum_{t=1}^{T}(x_t - \bar{x})^2}{T-1}}$$

use the command `std(x)`. If `x` is a vector, `std(x)` is scalar. If `x` is a matrix, `std(x)` is a K by 1 vector containing the sample standard deviations of each column. Note: This command always uses $T-1$ in the denominator, and is equivalent to `sqrt(var(x))`.

```
>> x=[1 2 3; 4 5 6]
x =
     1     2     3
     4     5     6
>> std(x)
ans =
    2.1213    2.1213    2.1213
>> std(x')
ans =
     1     1
```

skewness

To compute the sample skewness of a vector or matrix,

$$skew = \frac{\frac{\sum_{t=1}^{T}(x_t - \bar{x})^3}{T}}{\hat{\sigma}^3}$$

use the command `skewness(x)`. If `x` is a vector, `skewness(x)` is scalar. If `x` is a matrix, `skewness(x)` is a K by 1 vector containing the sample skewness of each column. Note: This command uses $T - 1$ in the denominator unless an optional second argument is used.

```
>> x=[1 2 3; 4 5 6]
x =
     1     2     3
     4     5     6
>> skewness(x)
ans =
     0     0     0
>> skewness(x')
ans =
     0     0
```

kurtosis

To compute the sample kurtosis of a vector or matrix,

$$\kappa = \frac{\frac{\sum_{t=1}^{T}(x_t - \bar{x})^4}{T}}{\hat{\sigma}^4}$$

use the command `kurtosis(x)`. If `x` is a vector, `kurtosis(x)` is scalar. If `x` is a matrix, `kurtosis(x)` is a K by 1 vector containing the sample kurtosis of each column. Note: This command always uses $T - 1$ to compute the sample variance.

```
>> x=[1 2 3; 4 5 6]
x =
     1     2     3
     4     5     6
>> kurtosis(x)
ans =
     1     1     1
>> kurtosis(x')
ans =
    1.5000    1.5000
```

: operator

The `:` operator has two uses. One, to access elements in a matrix of vector (e.g. `x(1,:)`), has already been described. The other is to create a row vector with evenly spaced points. In this context, the `:` operator has two forms, *first*:*last* and *first*:*increment*:*last*. The basic form, *first*:*last*, produces a row vector of the form

$$[first,\ \ first+1, \ldots first+N]$$

where N is the largest integer such that *first*+$N \leq$*last*. In common usage, *first* and *last* will be integers and $N =$*last-first*. Three examples to show how this construction works:

```
>> x=1:5
x =
     1     2     3     4     5
>> x=1:3.5
x =
     1     2     3
>> x=-4:6
x =
    -4    -3    -2    -1     0     1     2     3     4     5     6
```

The second form for the : operator includes an increment. The resulting sequence will have the form

$$[first,\ \ first+increment,\ \ first+2(increment), \ldots first+N(increment)]$$

where N is the largest integer such that *first*+*N*(*increment*)$\leq$*last*. Consider these two simple examples:

```
>> x=0:.1:.5
x =
         0    0.1000    0.2000    0.3000    0.4000    0.5000
>> x=0:pi:10
x =
         0    3.1416    6.2832    9.4248
```

The *increment* does not have to be positive. If a negative increment is used, the general form is unchanged but he stopping condition changes to N is the largest integer such that *first*+N(*increment*)$\geq$*last*. For example,

```
>> x=-1:-1:-5
x =
    -1    -2    -3    -4    -5
>> x=0:-pi:-10
x =
         0   -3.1416   -6.2832   -9.4248
```

Note: *first*:*last* is the same as *first*:1:*last* where 1 is the *increment*.

linspace

`linspace` is similar to the : operator. Rather than producing a row vector with a predetermined increment, `linspace` produces a row vector with a predetermined number of nodes. The generic form is `linspace(`*lower*`,`*upper*`,`N`)`

where *lower* and *upper* are the two bounds of the series and N is the number of points to produce.

If *inc* is defined as *inc*=(*upper-lower*)/(N-1), the resulting sequence will have the form

$$[lower,\ \ lower + inc,\ \ lower + 2(inc), \ldots lower + (N-1)(inc)]$$

and the command `linspace(lower,upper,N)` will produce the same output as `lower: (upper-lower)/(N-1): upper`.

Note: Remember `:` is a low precedence operator so operations involving `:` should always enclosed in parenthesis if there is anything else on the same line. Failure to do so can result in undesirable or unexpected behavior. For instance, consider:

```
>> N=4;
>> lower=0;
>> upper=1;
>> linspace(lower,upper,N)-(lower:(upper-lower)/(N-1):upper)
ans =
  1.0e-015 *
         0         0   -0.1110         0
>> linspace(lower,upper,N)-lower:(upper-lower)/(N-1):upper
ans =
         0    0.3333    0.6667    1.0000
```

MATLAB (correctly, based on its rules) interprets the second line as

```
>> (lower:(upper-lower)/(N-1):upper-lower):(upper-lower)/(N-1):upper
```

`logspace`

`logspace` produces points uniformly distributed in `log10` space. `logspace( lower, upper, N)` is the same as `10.^linspace( lower, upper, N)`.

```
>> logspace(0,1,4)
ans =
    1.0000    2.1544    4.6416   10.0000
```

5.1 Exercises

1. Load the MATLAB data file created in the Chapter 4 exercises and compute the mean, standard deviation, variance, skewness and kurtosis of both returns (`SP500` and `XOM`).
2. Create a new matrix, `returns = [SP500 XOM]'`. Repeat exercise 1 on this matrix.

3. Compute the mean of `returns'`.
4. Using both the `:` operator and `linspace`, create the sequence 0, 0.01, 0.02, . . . , .99, 1.
5. Create a custom logspace using the natural log (base e) rather than the logspace created in base 10 (which is what `logspace` uses). Hint: Use `linspace` AND `exp`.
6. Find the `max` and `min` of `SP500`. Create a new variable `SP500sort` which contains the sorted values of this series. Verify that the min corresponds to the first value of this sorted series and the max corresponds to the last Hint: `length` or `size`.

Chapter 6

Special Matrices

MATLAB contains commands to produce a number of useful matrices.

`ones`

`ones` does exactly what it appears to do: generate a matrix of 1's. Ones is generally called with two arguments, the number of rows and the number of columns.

```
oneMatrix = ones(N,M)
```

will generate a matrix of 1's with N rows and M columns. Note: To use the function call above, `N` and `M` must have been previously defined (e.g. `N=10; M=7`).

`eye`

`eye` generates an identity matrix (matrix with ones on the diagonal, zeros every where else). An identity matrix is always square so it only takes one argument.

```
In = ones(N)
```

`zeros`

`zeros` produces a matrix of 0's in the same way `ones` produces a matrix of 1's, and is useful for initializing a matrix to hold values produces by another procedure.

```
x = zeros(N,M)
```

6.1 Exercises

1. Produce two matrices, one containing all zeros and one containing only ones, of size 10×5.
2. Multiply these two matrices in both possible ways.
3. Produce an identity matrix of size 5. Take the exponential of this matrix, element-by-element.
4. How could these be replaced with `repmat`?

Chapter 7

Matrix Functions

MATLAB has a number of functions specifically designed to take matrix inputs. Some are mathematical in nature, for instance computing the eigenvalues and eigenvectors of a matrix, while other are simply functions for manipulating the elements of a matrix.

repmat

`repmat`, along with `reshape`, are among the most useful non-mathematical functions available in MATLAB. `repmat` replicates a matrix according to a specified size vector. To understand how repmat functions, imagine forming a matrix composed of blocks. The generic form of `repmat` is `repmat(`X`, `M`, `N`)` where X is the matrix to be replicated, M is the number of rows in the new block matrix, and N is the number of columns in the new block matrix. For example, suppose we had a matrix

$$X = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

and we needed to form the block matrix

$$Y = \begin{bmatrix} \mathrm{X} & \mathrm{X} & \mathrm{X} \\ \mathrm{X} & \mathrm{X} & \mathrm{X} \end{bmatrix}$$

This could be accomplished by manually constructing `y` as

```
>> x = [1 2; 3 4];
>> y = [x x x; x x x];
```

However, `y` could also be constructed using `repmat`

```
>> x = [1 2; 3 4];
>> y = repmat(x,2,3);
```

`repmat` has two clear advantages over manual allocation: (1) it can be executed based on some parameters determined at run-time, such as the number of explanatory variables in a model and (2) it can be used for arbitrary dimensions. Manual matrix construction becomes tedious and error prone with as few as 5 rows and columns.

reshape

`reshape` transforms a matrix with one set of dimensions and to one with a different set, preserving the number of elements. `reshape` can transform an M by N matrix x into an K by L matrix y as long as $MN = KL$ – and thus the number of elements does not change. The most useful call to reshape switches a matrix into a vector or vice versa. For example

```
>> x = [1 2; 3 4];
>> y = reshape(x,4,1)
y =
     1
     3
     2
     4
>> z = reshape(y,1,4)
z =
     1     3     2     4
>> w = reshape(z,2,2)
w =
     1     2
     3     4
```

The crucial detail to remember when using `reshape` is that MATLAB will always use the column-major notation to determine the shape of the new matrix. MATLAB always counts down then across, and will place elements of the old matrix into their same position in the new matrix. In other words, $x(1) = y(1)$, $x(2) = y(2)$, and so on.

diag

`diag` can produce one of two results depending on the form of the input. If the input is a square matrix, it will return a column vector of the elements along the diagonal of a matrix. If the input is a vector, it will return a matrix with the elements of the diagonal along the vector. Consider the following example:

```
>> x = [1 2; 3 4];
x =
     1     2
     3     4
>> y = diag(x)
y =
     1
```

```
     4
>> z=diag(y)
z =
    1     0
    0     4
```

det

`det` computes the determinant of a square matrix.

trace

`trace` computes the trace of a square matrix (sum of diagonal elements) and so `trace(x)` equals `sum(diag(x))`.

chol

`chol` computes the Cholesky factor of a positive definite matrix. The Cholesky factor is a lower triangular matrix and is defined as C in

$$C'C = \Sigma$$

where Σ is a positive definite matrix.

inv

`inv` computes the inverse of a matrix. `inv(x)` can alternatively be computed using `x^(-1)`.

eig

`eig` computes the eigenvalues and eigenvector of a square matrix. To get both the eigenvalues and eigenvectors, two output arguments are required, `[vec,val]=eig(x)`.

kron

`kron` computes the Kronecker product of two matrices,

$$z = x \otimes y$$

and is written as `z = kron(x,y)` in MATLAB code.

Chapter 8

`Inf`, `NaN` and Numeric Limits

MATLAB has three special expressions reserved to indicate certain non-numerical "values".

`Inf` standard for infinity. MATLAB understands both `Inf` and `-Inf`. `Inf` can be constructed in a number for ways, for instance `1/0` or `exp(1000)`.

`NaN` stands for not-a-number. `NaN`s are created when ever a function produces a result that cannot be clearly defined as a number of infinity. For instance, `inf/inf` produces a `NaN`.

MATLAB has some numeric limits. The easiest to understand are the upper and lower limits, which are `1.7977e+308` and `-1.7977e+308`. Numbers larger (in absolute value) than these are `Inf` in MATLAB. The smallest non-zero number MATLAB can express is `2.2251e-308`. Numbers between `-2.2251e-308` and `2.2251e-308` are 0 in MATLAB.

However, the hardest concept to understand about numerics is the limited precision. MATLAB has a precision of `2.2204e-016` (MATLAB command `eps` gives this number, and it may vary based on the type of CPU that is running MATLAB). Numbers which are outside of a *relative* range of `2.2204e-016` are considered the same. To explore the role of `eps`, examine the results of the following:

```
>> x=1
 x =
     1
 >> x=x+eps/2
 x =
      1
 >> x-1
 ans =
      0
 >> x=x+2*eps
 x =
     1.0000
 >> x-1
 ans =
     4.4409e-016
```

To understand what is meant by *relative* range, examine the following output

```
>> x=10
x =
    10
>> x+2*eps
ans =
    10
>> x-10
ans =
     0
```

In the first example, `eps/2` < `eps` so it has no effect while `2*eps` > `eps` so it does. However in the second example, `2*eps/10` < `eps`, it has no effect when added. This is a very tricky concept to understand, but failure to understand numeric limits can results in errors in code that appears to be otherwise correct.

8.1 Exercises

1. What is the value of `log(exp(1000))` both analytically and in MATLAB? Why do these differ?
2. What is the value of `eps/10`?
3. Is `.1` different from `.1+eps/10`?
3. Is `1e120` (1×10^{120}) different from `1e120+1e102`? (Hint: Test with `==`)

Chapter 9

Logical Operators and Find

Logical operators are useful when writing batch files or scripts. Logical operators, when combined with flow control, allow for complex choices to be compactly expressed.

9.1 >, >=, <, <=, == , ~=

The core logical operators are

>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less then or equal to
==	Equal to
~=	Not equal to

Logical operators can operate on scalars, vector or matrices. All comparisons are done element-by-element and return either 1 (logical true) or 0 (logical false). For instance, suppose x and y are matrices. `z=(x<y);` will be a matrix of the same size as x and y composed of 0's and 1's. Alternatively, if one is scalar, say y, then the elements of z are `z(i,j)=(x(i,j)<y);`.

Logical operators can be used to access elements of a vector or matrix. For instance, suppose `z = x` L y where L is one of the logical operators above such as <. The following table examines the behavior when x and/or y are scalars or matrices.
Suppose `z=x<y`:

		y: Scalar	y: Matrix
x: Scalar		Any $z = x < y$	Any $z_{ij} = x < y_{ij}$
x: Matrix		Any $z_{ij} = x_{ij} < y$	Same Dimensions $z_{ij} = x_{ij} < y_{ij}$

Logical operators are used in portions of programs known as flow control (for example `if ... else ... end` blocks), which will be discussed later. It is important to remember that vector or matrix logical operations return vector or matrix output and that flow control blocks *require scalar* logical expressions.

9.2 & (AND), | (OR) and ∼ (NOT)

Logical expressions can be combined using three logical devices,

&	AND
|	OR
∼	NOT

Recall that ∼ (NOT) has higher precedence than & (AND) and | (OR). They follow the same as other logical operators. If used on two matrices, the dimensions must be the same. If used on a scalar and a matrix, the effect is the same as calling the logical device on the scalar and each element of the matrix.

Suppose `x` and `y` are logical variables (1 or 0's). Suppose `z=x&y`:

		y: Scalar	y: Matrix
x	Scalar	Any $z = x\&y$	Any $z_{ij} = x\&y_{ij}$
	Matrix	Any $z_{ij} = x_{ij}\&y$	Same Dimensions $z_{ij} = x_{ij}\&y_{ij}$

9.3 `logical`

The MATLAB command `logical` is use to convert non-logical elements to logical. MATLAB treats logical values and regular numbers differently. Logical elements only take up 1 byte of memory (The smallest unit of memory MATLAB can address) while regular numbers require 8 bytes. In certain situations, a logical value is required. One such example is in indexing an array. As previously demonstrated, the elements of a matrix `x` can be accessed by `x(#)` where `#` can be a vector of indices. Since the elements of `x` are indexed 1,2,..., an attempt to retrieve `x(0)` will return an error. However, if `#` is not a number but a logical value, this behavior changes. MATLAB interprets logical indices as *indicator* functions. Consider the behavior in the following code:

```
>> x = [1 2 3 4];
>> y = [1 1];
>> x(y)
ans =
   1 1
>> y = logical([1 1]);
>> x(y)
ans =
   1 2
```

```
>> y = logical([1 0 1 0]);
>> x(y)
ans =
   1 3
```

The effect of logical is clear: It forces MATLAB to interpret the indices as indicator variables when deciding what to return. For another example, consider the following block of code

```
>> x = [1 2 3 4];
>> y = x<=2;
>> x(y)
ans =
   1 2
>> y
ans =
   1 1 0 0
```

Note: `logical` turns any non-zero value into logical true (1), although MATLAB will generate a warning if the values differ from 0 or 1. For example

```
>> x=[0 1 2 3]
x =
     0     1     2     3
>> logical(x)
Warning: Values other than 0 or 1 converted to logical 1.
ans =
    0     1     1     1
```

9.4 `all` and `any`

The MATLAB commands `all` and `any` take logicals as input and are self descriptive. `all` returns `logical(1)` if all logical elements in a vector are 1. If `all` is called on a matrix of logical elements, it works column-by-column, returns 1 if all elements of the column are logical true and 0 otherwise. `any` returns `logical(1)` if any element of a vector is logical true. Again, if called on a matrix, `any` operates column-by-column, returning logical true if any element of that column is true.

```
>> x = [1 2 3 4];
>> y = x<=2;
y =
   1 1
   0 0
>> all(y)
ans =
0
```

```
>> any(y)
ans =
1
>> x = [1 2 ; 3 4];
>> y = x<=3;
y =
   1 1
   1 0
>> all(y)
ans =
1 0
>> any(y)
ans =
1 1
```

9.5 find

`find` is an useful function for working with multiple data series. It isn't logical itself, but it takes logical inputs and returns matrix indices where the logical statement is true. There are two primary ways to call find. `indices = find (x < y)` will return indices (1,2,...,`numel(x)`) while `[i,j] = find (x < y)` will return pairs of matrix indices what correspond to the places where x<y.

```
>> x = [1 2 3 4];
>> y = x<=2
y =
   1 1 0 0
>> find(y)
ans =
   1 2
>> x = [1 2 ; 3 4];
>> y = x<=3
ans =
   1 1 1 0
>> find(y)
ans =
   1 2 3
>> [i,j] = find(y)
i =
   1 2 1
i =
   1 1 2
```

9.6 is*

MATLAB provides a number of special purpose logical tests to determine if a matrix has special characteristics. Some operate element-by-element and produce a matrix of the same dimension as the input matrix while other produce only scalars. These function all begin `is`.

isnan	1 if NaN	element-by-element
isinf	1 if Inf	element-by-element
isfinite	1 if not Inf	element-by-element
isreal	1 if not complex valued.	scalar
ischar	1 if input is a character array	scalar
isempty	1 if empty	scalar
isequal	1 if all elements are equal	scalar
islogical	1 if input is a logical matrix	scalar
isscalar	1 if scalar	scalar
isvector	1 if input is a vector ($1 \times K$ of $K \times 1$).	scalar

There are a number of other special purpose `is`$*$ expressions. For more details, search for is* in the help file.

```
>> x=[4 pi Inf Inf/Inf]
 x =
     4.0000    3.1416       Inf       NaN
 >> isnan(x)
 ans =
      0     0     0     1
 >> isinf(x)
 ans =
      0     0     1     0
 >> isfinite(x)
 ans =
      1     1     0     0
```

Note: `isnan(x)+isinf(x)+isfinite(x)` always equals 1, implying any element falls into one (and only one) of these categories.

9.7 Exercises

1. Using the MATLAB data file created in Chapter 4, count the number of negative returns in both the S&P 500 and Exxon Mobile.
2. For both series, create an indicator variable that takes the value 1 is the return is larger than 2 standard deviations or smaller than -2 standard deviations. What is the average return conditional on falling into this range for both returns.
3. Construct an indicator variable that takes the value of 1 when both returns are negative. Compute the correlation of the returns conditional on this indicator variable. How does this compare to the correlation of all returns?
4. What is the correlation when at least 1 of the returns is negative?
5. What is the relationship between `all` and `any`. Write down a logical expression that allows one or the other to be avoided (i.e. write `myany = ???` and `myall = ????`).

Chapter 10

Flow Control

The previous chapter explored one use of logical variables, selecting elements from a matrix. Logical variables have another important use: flow control. Flow control allows different code to be executed depending on whether certain conditions are met. Two flow control structures are available: `if ... elseif ... else` and `switch ... case ... otherwise`.

10.1 If Elseif Else

`if ... elseif ... else` blocks always begin with an `if` statement immediately followed by a **scalar** logical expression and must be terminated with `end`. `elseif` and `else` are optional and can always be replicated using nested `if` statements at the expense of more complex logic. The generic form of an `if ... elseif ... else` block is

```
if logical_1
        Code to run if logical_1 true
elseif logical_2
        Code to run if logical_2 true and logical_1 false
elseif logical_3
        Code to run if logical_3 true and logical_j false, j < 3
...
...
else
        Code to run all logical's false
end
```

However, simpler forms are more frequently used:

```
if logical_1
        Code to run if logical_1 true
end
```

or

```
if logical_1
       Code to run if logical_1 true
else
       Code to run if logical_1 false
end
```

Note: Do not forget that all *logical*'s must be scalar logical values.

A few simple examples

```
>> x = 5;
>> if x<5
     x=x+1;
   else
     x=x-1;
   end
>> x
ans =
    4
```

and

```
>> x = 5;
>> if x<5
     x=x+1;
   elseif x>5
     x=x-1;
   else
     x=2*x;
   end
>> x
ans =
    10
```

These examples have all used simple logical expressions. However, any *scalar* logical expressions, such as `(x<0 || x>1) && (y<0 || y>1)` or `isinf(x) || isnan(x)`, can be used in `if ... elseif ... else` blocks.

10.2 Switch Case Otherwise

`switch...case...otherwise` blocks allow advanced flow control although they can be completely replicated using only `if ... elseif ... else` flow control blocks. Do not feel obligated to use these if not comfortable in their application. The basic structure of this block is to find some variable whose value can be used to choose a piece of code to execute (the `switch` variable). Depending on the value of this variable (its `case`), a particular piece of code will be executed. If no cases are matched (`otherwise`), a default block of

code is executed (`otherwise` can safely be omitted. In this case, nothing is done if one of the `cases` is not matched). However, *at most* one block is matched. Matching a `case` causes that code block to execute then the program continues running on the next line after the `switch ... case ... otherwise` block. The generic form of an `switch ... case ... otherwise` block is

```
switch variable

case value_1
        Code to run if variable=value_1 true

case value_2
        Code to run if variable=value_2 true

case value_3
        Code to run if variable=value_3 true

...

...

otherwise
        Code to run if variable ≠ value_j ∀j

end
```

Note: There is an equivalence between `switch ... case ... otherwise` and `if ... elseif ... else` blocks. However, if the logical expressions contain inequalities, logical variables must be created prior to using a `switch ... case ... otherwise` block . These blocks differ from standard C behavior since only one case can be matched per block. After the first match, the block is exited and the program resumes with the next line after the block.
A simple `switch ... case ... otherwise` example:

```
x=5;
switch x
    case 4
        x=x+1;
    case 5
        x=2*x;
    case 6
        x=x-2;
    otherwise
        x=0;
end
>> x
ans =
   10
```

`cases` can include multiple values for the switch variable using the notation `case {`$case_1$`,`$case_2$`,... }`. For example,

```
x=5;
switch x
    case {4}
        x=x+1;
    case {1,2,5}
        x=2*x;
    otherwise
        x=0;
end
>> x
ans =
   10
```

10.3 Exercises

1. Write a code block that would take a different path depending on whether the returns on two series are simultaneously positive, both are negative, or they have different signs using an `if ... elseif ... else` block.
2. Construct a variable which takes the values 1, 2 or 3 depending on whether the returns in exercise 1 are both positive (1), both negative (2) or different signs (3). Repeat exercise 1 using a `switch ... case ... otherwise` block.

Chapter 11

Loops

Loops are the most useful programming structure in MATLAB. They make many problems, particularly when combined with flow control blocks, simple (and in many cases, possible). MATLAB has two loop blocks: `for...end` and `while...end`. `for` blocks loop over a predetermined iterator and `while` blocks loop as long as some logical expression is satisfied. All `for` loops can be expressed as `while` loops although the opposite is **not** true. They are nearly equivalent when `break` is used, although it is generally preferable to use a `while` loop to a `for` loop and a `break` statement.

11.1 `for`

`for` loops begin with `for` *iterator=vector* and end with `end`. The generic structure of a `for` loop is

```
for iterator=vector
        Code to run
end
```

iterator is a variable name the loop is iterating over. For example, `i` is a common iterator. *vector* is a vector of data. It can be an existing vector or it can be generated on the fly using `linspace` or `a:b:c` syntax (e.g. `1:10`). One subtle aspect of loops in MATLAB is that the iterator can contain any vector data, including non-integer and/or negative values. Consider these three examples:

```
count=0;
for i=1:100
   count=count+i;
end

count=0;
for i=linspace(0,5,50)
   count=count+i;
end
```

```
count=0;
x=linspace(-20,20,500);
for i=x
   count=count+i;
end
```

The first loop will iterate over $i = 1, 2, \ldots, 100$. The second loops over the values produced by the function `linspace` which creates 50 uniform points between 0 and 5, inclusive. The final loops over `x`, a vector constructed from a call to `linspace`. Loops can also iterate over decreasing sequences:

```
count=0;
x=-1*linspace(0,20,500);
for i=x
    count=count+i;
end
```

or vector with no order:

```
count=0;
x=[1 3 4 -9 -2 7 13 -1 0];
for i=x
    count=count+i;
end
```

The key to understanding `for` loop behavior is that MATLAB always iterates over the elements of *vector* in the order they are presented (i.e. *vector*(1), *vector*(2), ...). Loops can also be nested:

```
count=0;
for i=1:10
    for j=1:10
        count=count+j;
    end
end
```

and can contain flow control variables:

```
returns=randn(100,1);
count=0;
for i=1:length(returns)
    if returns(i)<0
        count=count+1;
    end
end
```

One particularly useful loop construct is to loop over the `length` of a vector, which allows each element to be modified one at a time.

```
trend=zeros(100,1);
for i=1:length(trend)
    trend(i)=i;
end
```

Finally, these ideas can be combined to produce nested loops with flow control.

```
matrix=zeros(10,10);
for i=1:size(matrix,1)
    for j=1:size(matrix,2)
        if i<j
            matrix(i,j)=i+j;
        else
            matrix(i,j)=i-j;
        end
    end
end
```

of loops containing nested loops that are executed based on a flow control statement.

```
matrix=zeros(10,10);
for i=1:size(matrix,1)
    if (i/2)==floor(i/2)
        for j=1:size(matrix,2)
            matrix(i,j)=i+j;
        end
    else
        for j=1:size(matrix,2)
            matrix(i,j)=i-j;
        end
    end
end
```

Note: The iterator variable must **NOT** be modified inside the for loop. Changing the iterator can produce undesirable results. For instance,

```
for i=1:100;
    i
    i=1;
    i
end
```

Produces the output

```
...
i =
    99
i =
     1
i =
   100
i =
     1
```

which can lead to unpredictable results if `i` is used inside the loop.

11.2 `while`

`while` loops are useful when the number of iterations needed is unknown. `while` loops are commonly used when a loop should only stop if a certain condition is met, such as the change in some parameter is small. The generic structure of a `while` loop is

```
while logical

      Code to run

      Update to logical inputs
end
```

Two things are crucial when using a `while` loop: 1. *logical* should be true when the loop begins (or the lop will be ignored) and 2. The inputs to the *logical* variable must be updated inside the loop. If they are not, the loop will continue for ever (hit CTRL+C to break an errant loop). The simplest while loops are drop-in replacements for `for` loops:

```
count=0;
i=1;
while i<=10
    count=count+i;
    i=i+1;
end
```

which produces the same results as

```
count=0;
for i=1:10
    count=count+i;
end
```

`while` loops should generally be avoided when `for` loops will do. However, there are situations where no `for` loop equivalent exists.

```
mu=1;
index=1;
while abs(mu)>.0001
    mu=(mu+randn)/index;
    index=index+1;
end
```

In the block above, the number of iterations required is not known in advance and since `randn` is a standard normal pseudo-random number, it may take many iterations until this criteria is met. Any finite `for` loop cannot be guaranteed to meet the criteria.

11.3 `break`

`break` can be used to break out of a loop and can make `for` loops behave nearly like a while loop:

```
for iterator=vector
       Code to run
       if logical
             break
       end
end
```

The only difference between this loop and a standard while loop is that the while loop could potentially run for more iterations and *it0erator*. `break` can also be used to end a while loop*before* running the code inside the loop. Consider this slightly strange loop:

```
while 1
    x = randn;
    if x<0
        break
    end
    y = sqrt(x);
end
```

The use of `while 1` will produce a loop, if left alone, that will run indefinitely. However, the `break` command will stop the loop if some condition is met. More importantly, the `break` will prevent the code after it from being run, which is useful if the operations after the `break` will create errors if the condition is not true.

11.4 `continue`

`continue`, when used inside a loop, has the effect of advancing the loop to the next iteration and skipping any remaining code in the body of the loop. It use can always be avoided using `if...else` blocks, but it can make code tidier. It's effect is best seen through a block of code:

```
for i=1:10
    if (i/2)==floor(i/2)
        continue
    end
    i
end
```

which produces output

```
...
...
i =
     7
i =
     9
```

demonstrating that `continue` is forcing the loop to the next iteration when ever `i` is even (and `(i/2)==floor(i/2)` is logical true).

11.5 Exercises

1. Simulate 1000 observations from an ARMA(2,2) with normal innovations. The process of an ARMA(2,2) is given by

$$y_t = \phi_1 y_{t-1} + \phi_2 y_{t-2} + \theta_1 \epsilon_{t-1} + \theta_2 \epsilon_{t-2} + \epsilon_t$$

Use the values $\phi_1 = 1.4$, $\phi_2 = -.8$, $\theta_1 = .4$ and $\theta_2 = .8$. **Note**: When simulating a process, always simulate more data then needed and throw away the first block of observations to avoid start-up biases. This process is fairly persistent, at least 100 extra observations should be computed. As a rule, I always compute at least 2000 extra data points to throw away, even when simulating a short series.
2. Simulate a GARCH(1,1) process with normal innovations. A GARCH(1,1) process is given by

$$y_t = \epsilon_t \sqrt{h_t}$$

$$h_t = \omega + \alpha \epsilon_{t-1} + \beta h_{t-1}$$

Use the values $\omega = 0.05$, $\alpha = 0.05$ and $\beta = 0.9$.

3. Simulate a GJR-GARCH(1,1,1) process with normal innovations. A GJR-GARCH(1,1) process is given by

$$y_t = \epsilon_t \sqrt{h_t}$$

$$h_t = \omega + \alpha\epsilon_{t-1} + \gamma\epsilon_{t-1} I_{[\epsilon_{t-1}<0]} + \beta h_{t-1}$$

Use the values $\omega = 0.05$, $\alpha = 0.02\,\gamma = 0.07$ and $\beta = 0.9$. Hint: Some form of logical expression is needed in the loop. $I_{[\epsilon_{t-1}<0]}$ is an indicator variable that takes the value 1 if the expression inside the [ ] is true.

4. Simulate a ARMA(1,1)-GJR-GARCH(1,1)-in-mean process,

$$y_t = \phi_1 y_{t-1} + \theta_1 \epsilon_{t-1}\sqrt{h_{t-1}} + \lambda h_t + \epsilon_t \sqrt{h_t}$$

$$h_t = \omega + \alpha\epsilon_{t-1} + \gamma\epsilon_{t-1} I_{[\epsilon_{t-1}<0]} + \beta h_{t-1}$$

Use the values from Exercise 3 for the GJR-GARCH model and use the $\phi_1 = -0.1$, $\theta_1 = 0.4$ and $\lambda = 0.03$.

5. Using a `while` loop, write a bit of code that will do a bisection search to invert a normal CDF. A bisection search cuts the interval in half repeatedly, only keeping the sub interval with the target in it. Hint: keep track of the upper and lower bounds of the random variable value and use flow control. This problem requires `normcdf`.

6. Test out the loop using by finding the inverse CDF of 0, -3 and `pi`. Verify it is working by taking the absolute value of the difference between the final value and the value produced by `norminv`.

Chapter 12

Plotting Data

MATLAB has extensive plotting facilities and that a wide range of graphical data representations. Despite the broad capabilities of the graphics system in MATLAB, this chapter will emphasize the basics.

12.1 Support Functions

All plotting functions have a set of support functions which are useful for providing labels for various portions of the plot or making adjustments to the range. Remember to fully label plots so others can clearly tell which series are being plotted and the units of the plot.

- `legend` labels the various elements on a graph. The specific behavior of legend depends on the type of plot and the order of the data. `legend` takes as many strings as unique plot elements. Standard usage is `legend('Series 1','Series 2')` where the number of series is figure dependent.
- `title` places a title at the top of a figure. Standard usage is `title('Figure Title')`.
- `xlabel`, `ylabel` and `zlabel` produce text labels on the x, y and z –if 3D – axes respectively. Standard usage is `xlabel('X Data Name')`.
- `axis` can be used to both get the axis limits and set the axis limits. To retrieve the current axis limits, enter `AX = axis();`. `AX` will be a row vector of the form [xlow xhigh ylow yhigh (zlow) (zhigh)] where zlow and zhigh are only included if the figure is 3D. The axis can be changed by calling `axis([xlow xhigh ylow yhigh (zlow) (zhigh)])` where the z-variables are only allowed if the figure is 3D. `axis` can also be used to tighten the axes to include only the minimum space required to express the data using the command `axis tight`.

These four are the most important, but there are many additional functions available to tweak figures.

12.2 Plot

`plot` is the most basic plotting command. Like most commands, it can be used many ways. However, the most straight forward is

```
plot(x1,y1,format1,x2,y2,format2,...)
```

where `xi` and `yi` are vector of the same size and $format_i$ is a format string of the form *color shape linespec*. *color* can be any of

b	blue
g	green
r	red
c	cyan
m	magenta
y	yellow
k	black

shape can be any of

o	circle
x	x-mark
+	plus
*	star
s	square
d	diamond
v	triangle (down)
^	triangle (up)
<	triangle (left)
>	triangle (right)
p	pentagram
h	hexagram

and *linespec* can be any of

-	solid
:	dotted
-.	dashdot
- -	dashed
(non)	no line

The three arguments are combined to form a format string. For instance `'gs-'` will produce a green solid line with squares at every data point while `'r+ '` will produce a set of red + symbols at every data point. Arguments which are not needed can be left out. For instance, to produce a green dotted line with no symbol, use the format string `'g:'`. If no format string is provided, MATLAB will use an automatic color scheme and plot solid lines with no markers. Suppose the following x and y data were created in MATLAB.

```
x = linspace(0,1,100);
y1 = 1-2*abs(x-0.5);
y2 = x;
y3 = 1-4*abs(x-0.5).^2;
```

Calling `plot(x,y1,'rs:',x,y2,'bo-.',x,y3,'kp-')` will produce the plot in figure 12.1. A line's color information is lost when documents printed are in black and white. Always use physical characteristics to distinguish multiple series – either different line types or different markers, or both.

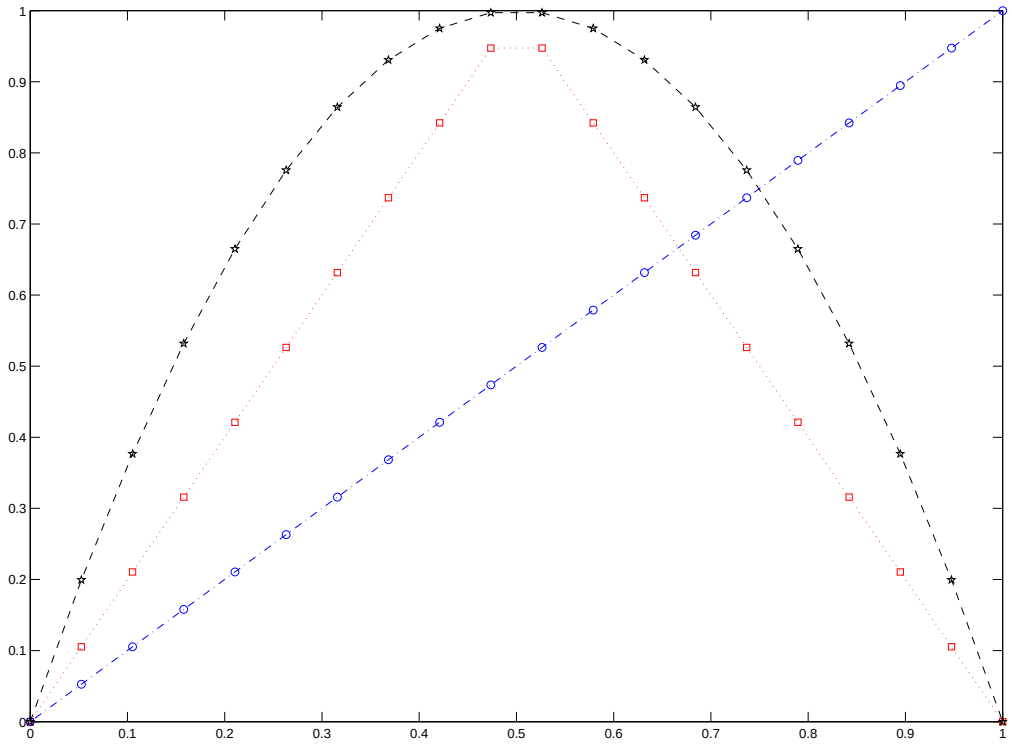


Figure 12.1: Simple plot of three lines. The lines were plotted with the command `plot(x,y1,'rs:',x,y2,'bo-.',x,y3,'kp- -')`.

All plots should be clearly labeled and this one is no exception. The following code block labels the axes, gives the figure a title, and provides a legend. The results of running the code along with the `plot` command above can be seen in figure 12.2.

```
xlabel('x');
ylabel('f(x)');
title('Plot of three series');
legend('f(x)=1-|x-0.5|','f(x)=x','f(x)=1-4*abs(x-0.5).^2');
```

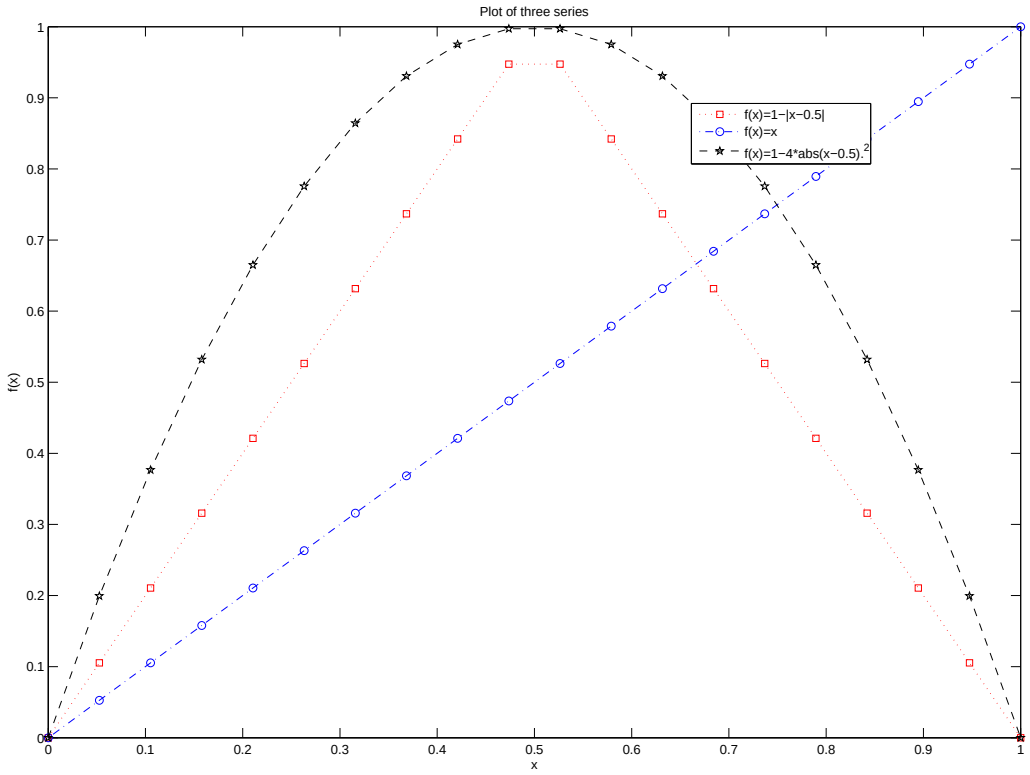


Figure 12.2: Labeled plot of three lines. Be certain to clearly label axes and provide a title and legend so other can comprehend the contents of a figure.

One other form of the plot command is worth mentioning. `plot(y)` will plot the data in vector `y` against a simple series which labels each observation 1, 2, ..., `length(y)`. In fact, `plot(y)` is equivalent to `plot(1:length(y),y)` when `y` is a vector. If `y` is a matrix, `plot` will draw each column of `y` as if it was a separate series. When `y` is a matrix, `plot(y)` is equivalent to `plot(1:length(y(:,1)), y(:,1), 1:length(y(:,2)), y(:,2), ...)`.

12.3 Plot3

`plot3` is behaves similarly to `plot` with the exception it plots a series against two other series in 3-space. All arguments are the same and the generic form is

```
plot3(x1,y1,z1,format1,x2,y2,z2,format2,...)
```

The following code block demonstrates the use of `plot3`.

```
figure(2)
N=200;
x=linspace(0,8*pi,N);
x=sin(x);
y=linspace(0,8*pi,N);
y=cos(y);
z=linspace(0,1,N);
plot3(x,y,z,'rs:');
xlabel('x');
ylabel('y');
zlabel('z');
title('Spiral');
legend('Spiraling Line')
```

The results of this block of code can be seen in figure 12.3.

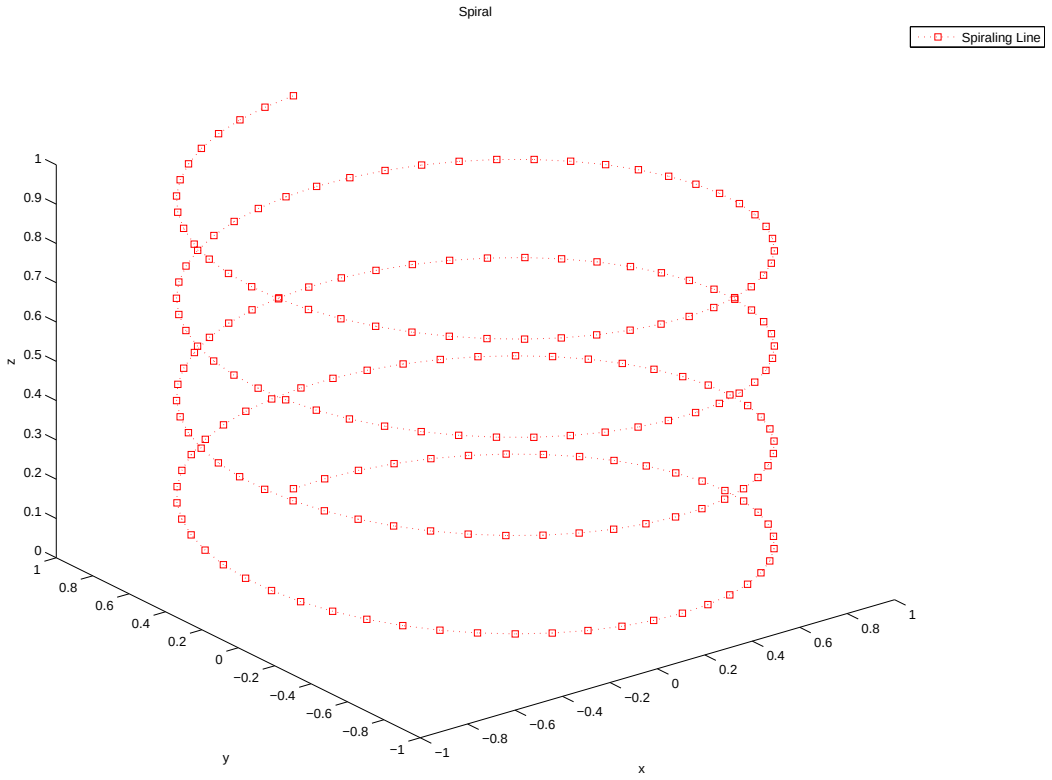


Figure 12.3: 3D Spiral plot. 3D lines can be plotted using the `plot3` command. This line was plotted by calling `plot3(x,y,z,'rs:');`.

12.4 Scatter

`scatter`, like most graphing functions in MATLAB, is self descriptive. It produces a scatter plot of the elements of a vector x against the elements of a vector y. Formatting, such as color or market shape can only be changed by either using *handle graphics* or manually editing the plot. Simple example of these are included at the end of this chapter. Consult the MATLAB help file for `scatter` for further information.

The following code produces a scatter diagram of 1000 pseudo-random numbers from a normal distribution, each with unit variance and correlation of 0.5. The output of this code can be seen in figure 12.4.

```
figure(4)
x=randn(1000,2);
Sigma=[2 .5;.5 0.5];
x=x*Sigma^(0.5);
scatter(x(:,1),x(:,2),'rs')
xlabel('x')
ylabel('y')
legend('Data point')
title('Scatter plot of correlated normal random variables')
```

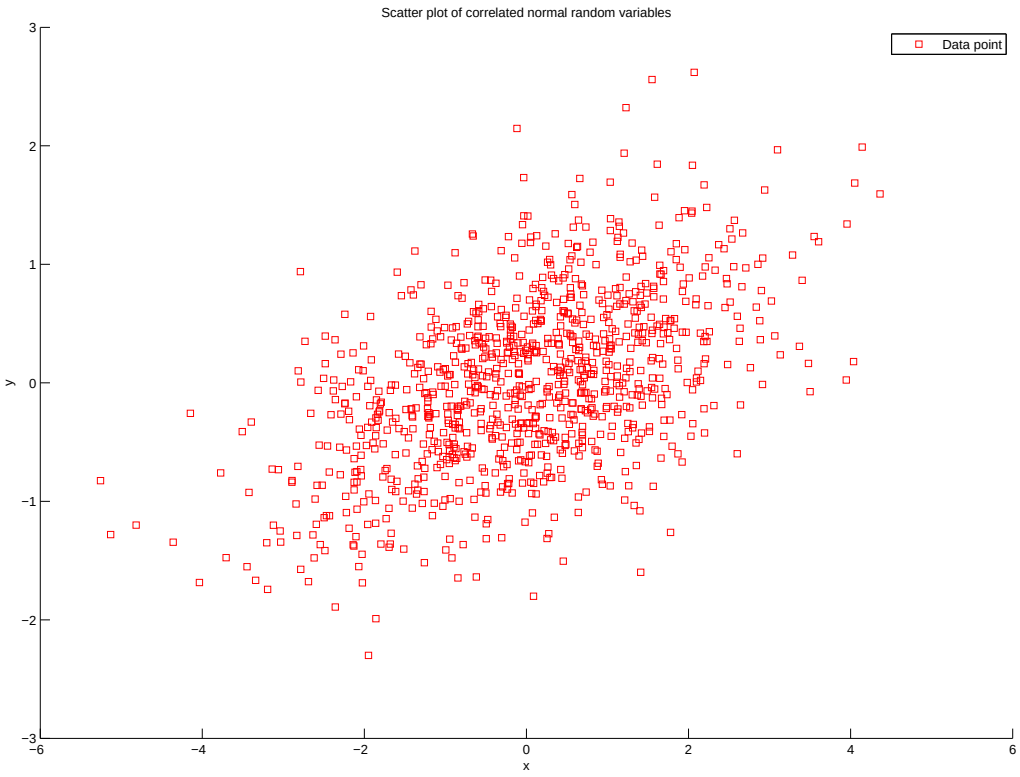


Figure 12.4: Scatter plot. This plot contains a scatter plot of a bivariate normal random deviations with unit variance and correlation of 0.5. This line was plotted by calling `scatter(x(:,1),x(:,2),'rs');`.

12.5 Surf

The next three graphics tools all plot a matrix of z data against vector of x and y data. All three uses the results from a bivariate normal probability density function. The PDF of a bivariate normal with mean 0 is given by

$$f_X(x) = -\frac{1}{2\pi|\Sigma|^{\frac{1}{2}}} \exp(-\frac{1}{2}x'\Sigma^{-1}x)$$

In this example, the covariance matrix, Σ, was chosen

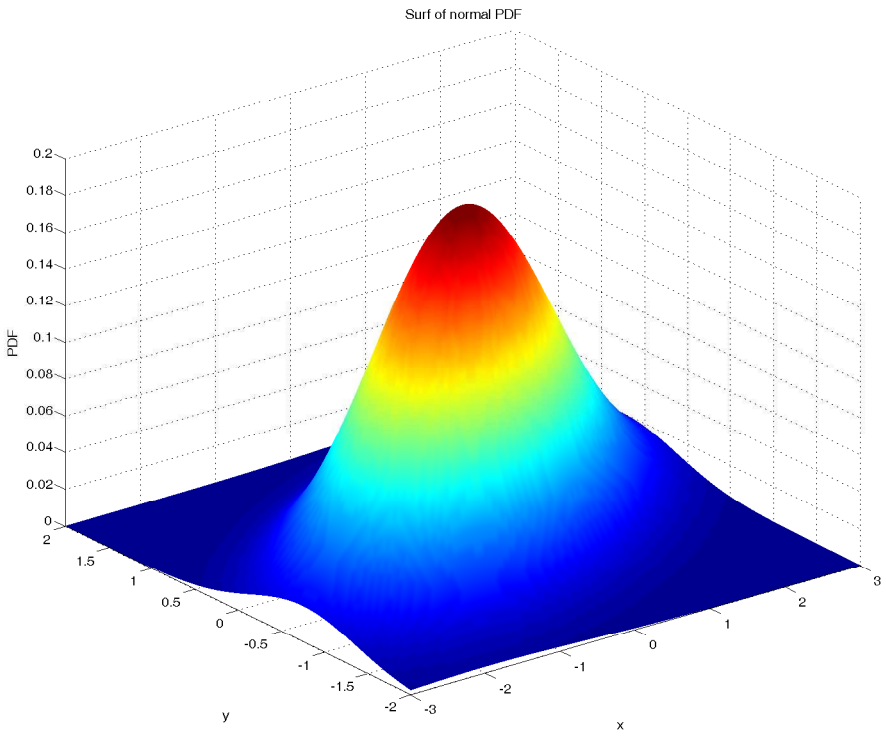


Figure 12.5: Surface plot. `surf` plots a 3D surface from vectors of x and y data and a matrix of z data. This `surf` contains the PDF bivariate of a bivariate normal, and was created using `surf(x,y,pdf)` where `x`, `y` and `pdf` are defined in the text.

$$\Sigma = \begin{bmatrix} 2 & 0.5 \\ 0.5 & 0.5 \end{bmatrix}$$

A matrix of pdf values, `pdf` was created with the following code:

```
N = 100;
x = linspace(-3,3,N);
y = linspace(-2,2,N);
```

```
pdf=zeros(N,N);
for i=1:length(y)
   for j=1:length(x)
      pdf(i,j)=exp(-0.5*[x(j) y(i)]*Sigma^(-1)*[x(j) y(i)]')/sqrt((2*pi)^2*det(Sigma));
   end
end
```

The first two lines initialize the `x` and `y` values. Since `x` has a higher variance, it has a larger range. The `surf` (figure 12.5) was created by

```
surf(x,y,pdf)
xlabel('x')
ylabel('y')
zlabel('PDF')
title('Surf of normal PDF')
shading interp
```

The command `shading interp` changes how the colors are applied form a discrete to grid to a continuous grid.
Note: The x and y arguments of `surf` must match the dimensions of the z argument. If `[M,N]=size(z)`, then `length(y)` must be `M` and `length(x)` must be `N`. This is true of all 3D plotting functions that draw matrix data. In the code above, `i` is the row iterator which corresponds to `y` and `j` is the column iterator, corresponding to `x`.

12.6 Mesh

`mesh` produces a graphic similar to `surf` but with empty space between grid points. Mesh has the advantage that the *hidden* side can be seen, potentially revealing more from a single graphic. It also produces much smaller files which can be important when including multiple graphics in a presentation or repot. Using the same bivariate normal setup, the following code produces the `mesh` plot evidenced in figure 12.6.

```
mesh(x,y,pdf)
xlabel('x')
ylabel('y')
zlabel('PDF')
title('Mesh of normal PDF')
```

12.7 Contour

Contour is similar to `surf` and `mesh` in that it takes three arguments, x, y and z. However, it differs in that it produces a 2D plot. `contour` plots, while not as eye-catching as `mesh` plots, are often better at conveying meaningful information. Contour plots can be either called as `contour(x,y,z)` or `contour(x,y,z,N)`

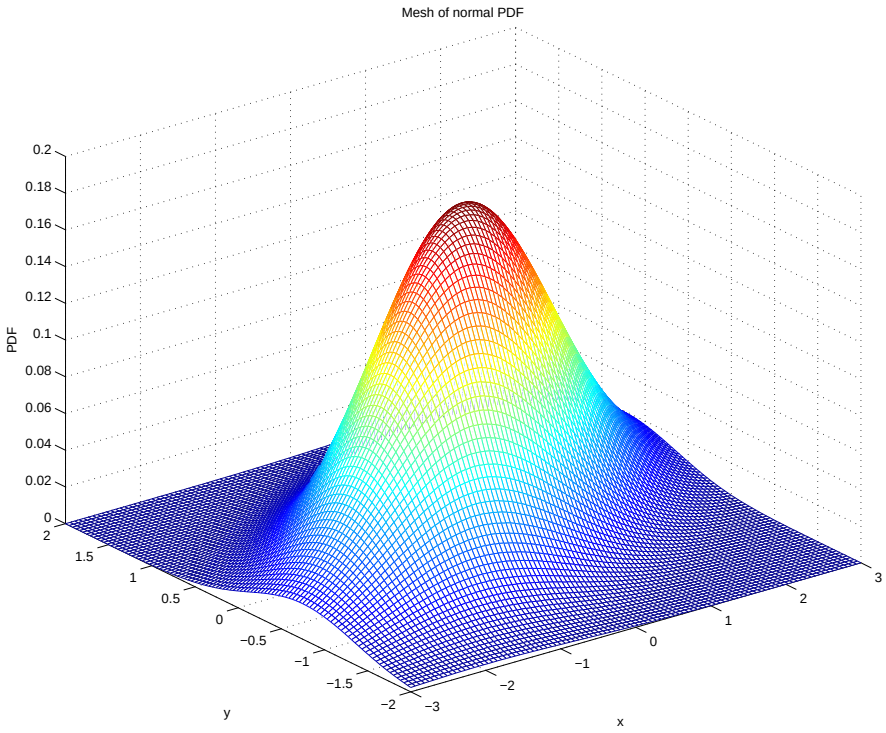


Figure 12.6: Mesh plot. `mesh` produce a figure similar to `surf` but with gaps between grid points, allowing the backside of a figure to be seen in a single view. This `mesh` contains the PDF bivariate of a bivariate normal, and was created using `mesh(x,y,pdf)` where `x`, `y` and `pdf` are defined in the text.

where `N` instructs MATLAB how many contours to produce. If omitted, MATLAB will determine the number of contours based on the variance of the `z` data. The code below and figure 12.7 demonstrate the use of `contour`.

```
contour(x,y,pdf);
xlabel('x')
ylabel('y')
title('Contours of normal PDF')
```

12.8 Subplot

Subplots allow for multiple plots to be placed in the same figure. All calls to subplot must specify three arguments, the number of rows, the number of columns, and which cell to place the graphic. The generic form is

`subplot(`M`,`N`,#)`.

where M is the number of rows, N is the number of columns, and # indicates the cell to place the graphic. Cells in a subplot are counted across then down For instance, in a call to `subplot(3,2,#)`, the #'s would be

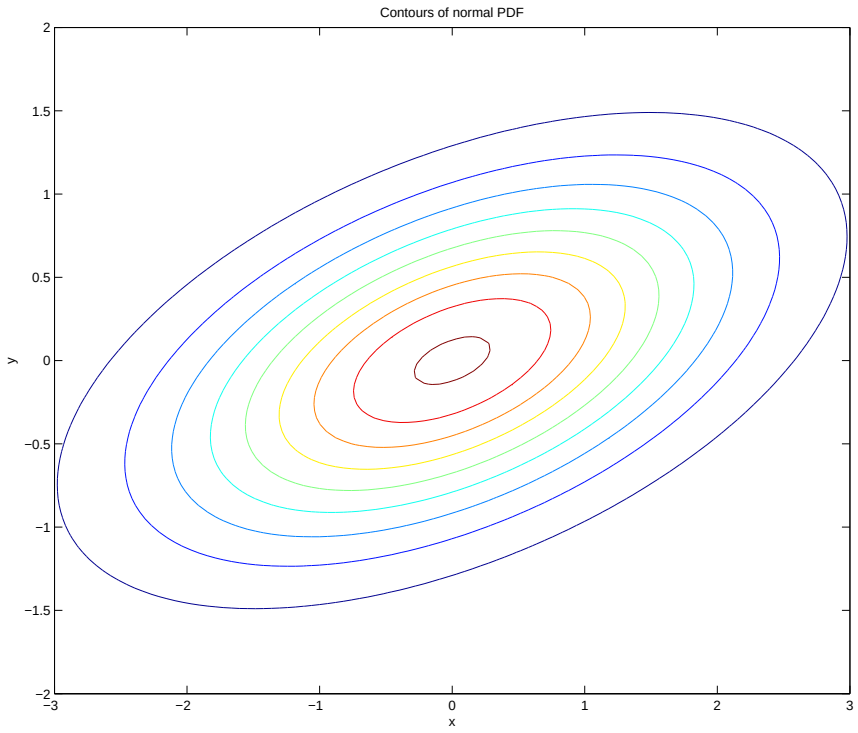


Figure 12.7: Contour plot. A `contour` plot is a set of slices through a `surf` plot. This particular contour plot contains iso-probability lines from a bivariate normal distribution with mean 0, variances of 2 and 0.5, and correlation of 0.5.

1 2
3 4
5 6

A call to `subplot` should be immediately followed by some plotting function. In the simplest case, this would be a call to plot. However, any graphic function in MATLAB can be used in a subplot. The code below and output in figure 12.8 demonstrates how different data visualizations may be used in every cell. These also show a few of the available graphics function that are not described in these notes.

```
subplot(2,2,1);
x = [5 3 0.5 2.5 2];
explode = [0 1 0 0 0];
pie(x,explode)
colormap jet
title('pie function')
axis tight

subplot(2,2,2);
Y = cool(7);
bar3(Y,'detached')
```

```
title('Detached')
title('bar3, ''Detached''')
axis tight

subplot(2,2,3)
bar3(Y,'grouped')
title('bar3, ''Grouped''')
axis tight

subplot(2,2,4);
x = 1:10;
y = sin(x);
e = std(y)*ones(size(x));
errorbar(x,y,e)
title('errorbar')
axis tight
```

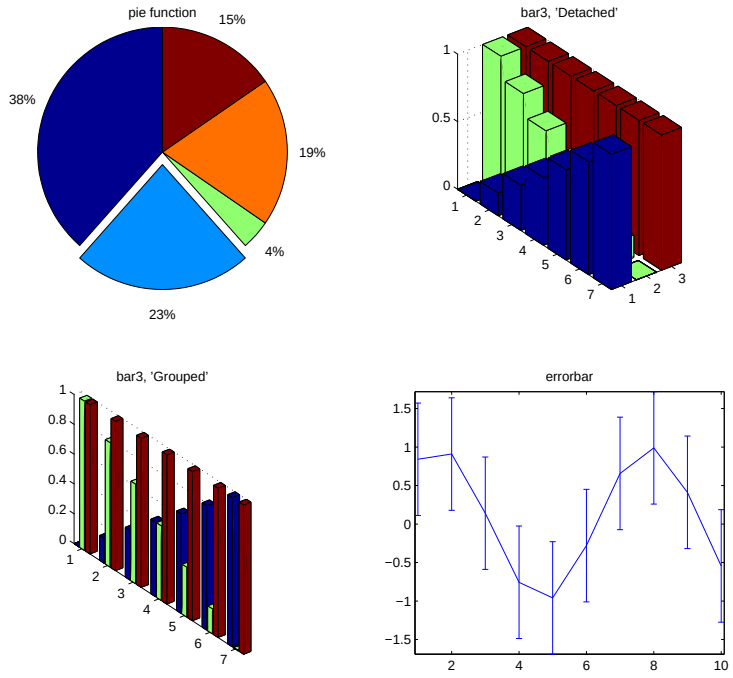


Figure 12.8: Subplot example. Subplots allow for more than one graphic to be included in a figure. This particular subplot contains three different types of graphics with two variants on the 3D bar. The upper left contains a call to `pie`, the upper right contains a call to `bar3` specifying the option `'grouped'`, the lower left contains a call to `bar3` specifying the options `'detached'` and the lower right contains the results to a call to `errorbar`.

Note: The graphics code in each subplot was taken straight form the MATLAB help files. The help system is very comprehensive and illustrates most functions with example code.

12.9 Advanced Graphics

While the standard graphics functions of MATLAB are very powerful and can directly accomplish many tasks, sometimes they are not sufficiently general. For instance, if is often useful to change the thickness of a line in order to improve its appearance or to add an arrow to highlight a particular feature of a graph.

Fortunately, MATLAB provides two mechanisms to add elements to a plot. The first, which will be referred to as "point-and-click", involves manually editing the plot in the figure window. The second, and more general of the two, is known as *handle graphics*. Handle graphics provides a mechanism to *programmatically* change anything about a graph.

12.9.1 Point-and-click

The simplest method to improve plots is to use the editing facilities of the figure windows directly. A number of buttons are available along the top edge of a plot. One the of these is an arrow, (**1**) in figure 12.9. Clicking on the arrow will highlight it and allow any element, such as a line, to be selected. Double-clicking on a line will bring up a Property Editor (**2**) dialog which contains elements of the selected item that can be changed. These include color, line width, and marker (**3**). For more information in editing plots, search for *Editing Plots* in the MATLAB help.

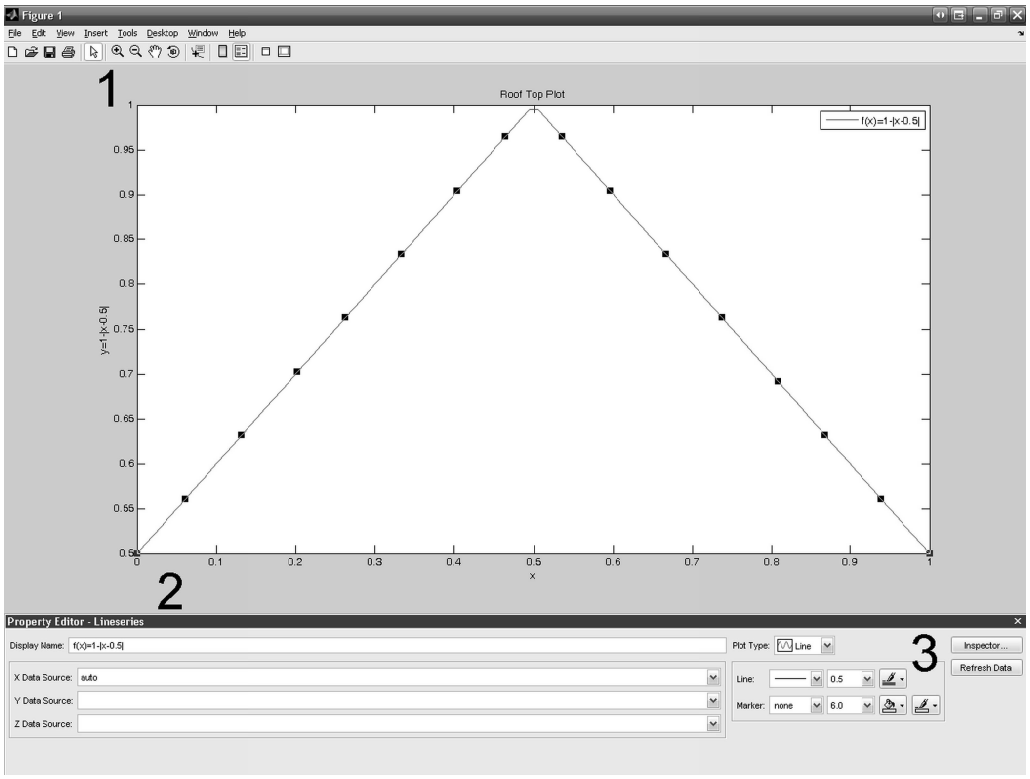


Figure 12.9: point-and-click editing. Most features of a plot can be editing using the interactive editing tools of a figure window. Interactive editing is started by first selecting the arrow icon along the top of the figure (**1**), then clicking on the element to be edited (e.g. the line, the axes, any text label). This will bring up the Property Editor (**2**) where the item specific properties can be changed (**3**).

12.9.2 Handle Graphics

Part of the power of MATLAB graphics is that it is fully programmable. Anything that can be accomplished through manual editing of a plot can be accomplished by using *handle graphics*. Every graphical element is assigned a handle. The handle contains everything there is to know about the particular graphic, such as colors or line widths. Once familiar with handle graphics, they can be used to create spectacularly complex data visualizations. However, their use will be illustrated through a simple example.

The example will illustrate the use of handle graphics by showing both before and after plots using subplot.

```
e  = randn(100,2);
y  = cumsum(e);
subplot(2,1,1);
plot(y);
legend('Random Walk 1','Random Walk 2')
title('Two Random Walks')
xlabel('Day')
ylabel('Level')

subplot(2,1,2);
h  = plot(y);
l  = legend('Random Walk 1','Random Walk 2')
t  = title('Two Random Walks')
xl = xlabel('Day')
yl = ylabel('Level')
set(h(1),'Color',[1 0 0],'LineWidth',2,'LineStyle',':')
set(h(2),'Color',[1 .6 0],'LineWidth',2,'LineStyle','-.')
set(t,'FontSize',14','FontName','Bookman Old Style','FontWeight','demi')
set(l,'FontSize',14','FontName','Bookman Old Style','FontWeight','demi')
set(xl,'FontSize',14','FontName','Bookman Old Style','FontWeight','demi')
set(yl,'FontSize',14','FontName','Bookman Old Style','FontWeight','demi')
parent = get(h(1),'Parent');
set(parent,'FontSize',14','FontName','Bookman Old Style','FontWeight','demi')
```

Most things that can be accomplished through handle graphics can be accomplished using the point-and-click editing method outlined above. However, the advantage of handle graphics is are more apparent when a figure needs to be updated or redrawn. When redrawing a figure, ff using handle graphics, only the code needs to be updates code and rerun. If using the point-and-click editing method, each change must be manually reapplied after any every change in the data.

For more on handle graphics, please consult the *Handle Graphics Properties* in the MATLAB help file.

12.10 Exercises

1. Generate two random walks using a loop and `randn`. Plot these two on a figure and provide all of the necessary labels.
2. Generate a 3D plot from

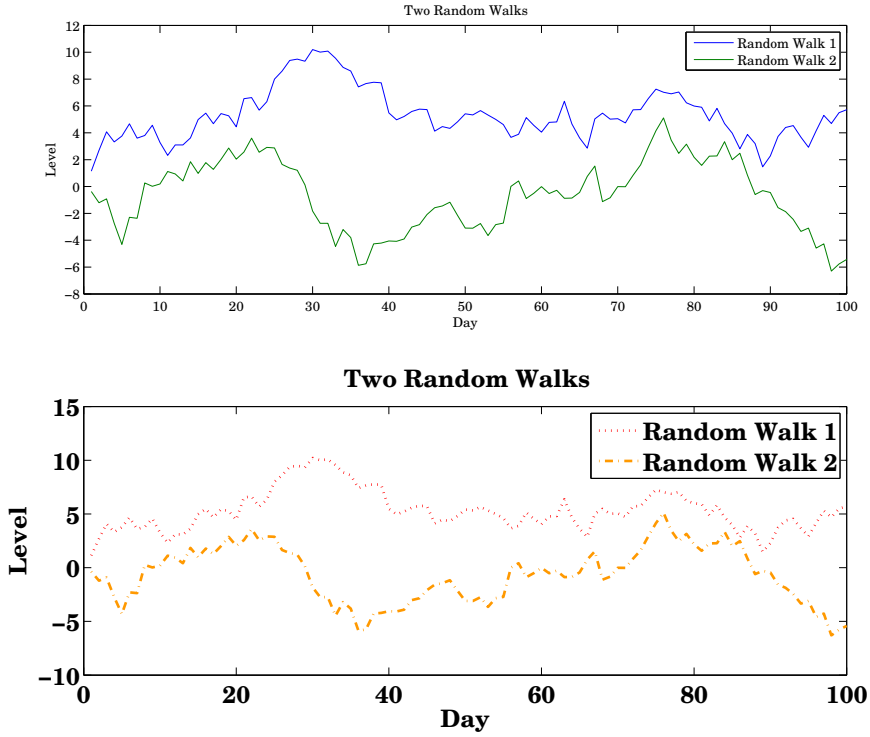


Figure 12.10: Handle graphics. The top subplot is a standard call to `plot` while the bottom highlight some of the possibilities when using handle graphics. It is worth nothing that all of these changes evidenced int eh bottom subplot can be reproduces using the point-and-click method.

```
x = linspace(0,10*pi,300);
y = sin(x);
z = x*y;
```

Label all axes, title the figure and provide a legend.

3. Generate 1000 draws from a normal. Plot a histogram with 50 bins of the data.
4. Using the Dell and S&P 500 data, produce a subplot with 4 windows containing:

- A scatter plot of the two series
- Two histograms of the series
- One plot of the two series against the dates. Change the axis labels to text using `datetick`.

Chapter 13

Exporting Plots

Once a plot has been finalized, it must be exported to be included in an assignment, report or project. Exporting is straight forward. On the figure, click File, Save As (**1** in figure 13.1). In the Save as type box, select the desired format (TIFF for Microsoft Office uses, EPS file for LaTeX(**2** in figure 13.2)), enter a file name (**1** in figure 13.2) and save. Figures 13.1 and 13.2 contain representations of the steps needed to export from a figure box.

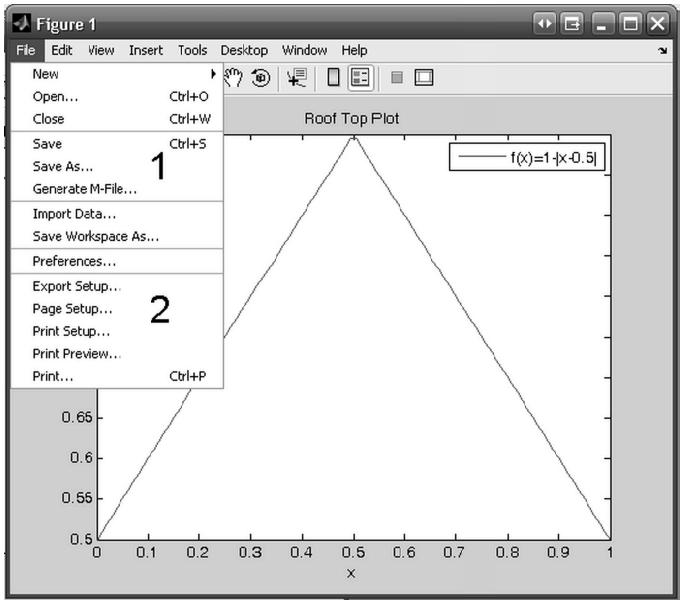


Figure 13.1: Steps to export a figure. To export a figure, click *Save As...* in the file menu of a figure (**1**). The dialog in figure 13.2 will appear.

If the exported figure does not appear as desired, some options in Page Setup may need to be altered (**2** in figure 13.1). Specifically, it may be useful to change the paper orientation to Landscape (Paper tab) and then hit Fill Page, Fix aspect ratio and Center (**1**, **2** and **3** in figure 13.4) on the Size and Position Tab. Figures 13.3 and 13.4 contain representation of the Page Setup screens needed to change the page size for exporting.

Note: Figures can be exported programmatically using the `print` command.

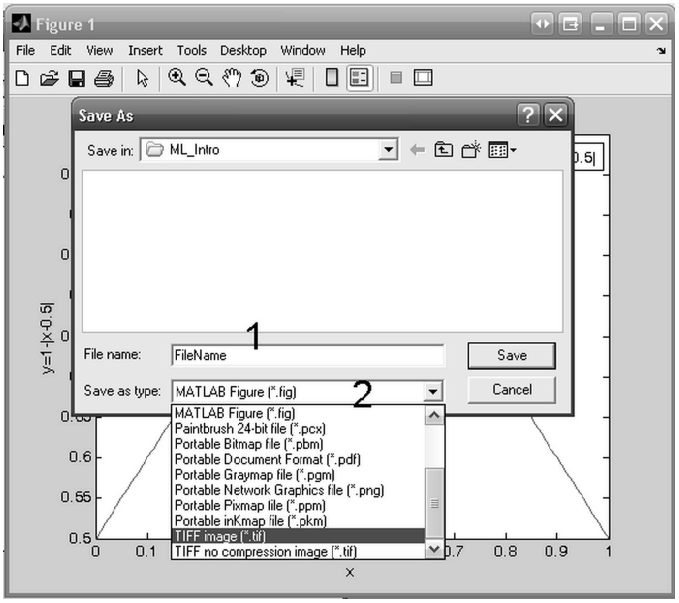


Figure 13.2: Save as dialog. To export a figure, enter a file name and use the drop-down box to select a file type. Select TIFF image if using Microsoft Office or EPS File (Encapsulated Postscript) if using LaTeX.

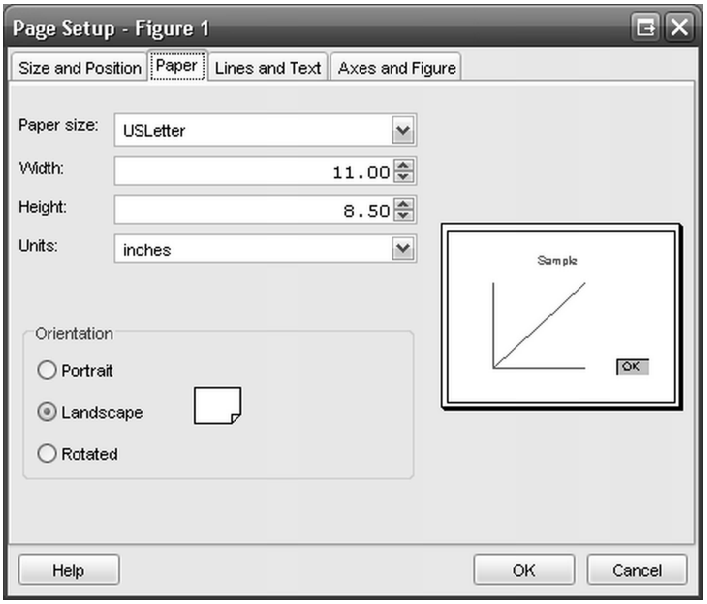


Figure 13.3: Page Setup, Paper tab. If the exported figure is too small, change the paper to Landscape and then change the size using the Size and Position tab (figure 13.4).

13.1 Exercises

1. Export the plot from exercise 1 of the previous chapter as a TIFF and an EPS. View the files created outside of MATLAB.

2. Use page setup to change the orientation and dimensions as described in this chapter. Re-export the

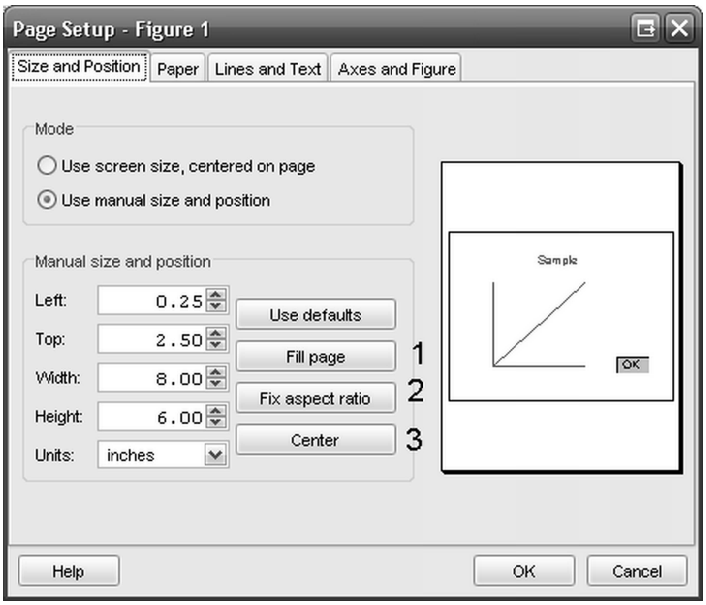


Figure 13.4: Page Setup, Size and Position tab. After changing the paper orientation to Landscape (figure 13.3), make sure the page is filled by clicking on (**1**) Fill Page, (**2**) Fix aspect ratio and then (**3**) Center. This will produce large, high quality exported figures which can be resized in Office or LaTeX.

figure as both a TIFF and EPS (using different names) and compare the new images to the old versions.

Chapter 14

Custom Functions

In addition to writing batch files and calling predefined functions, MATLAB allows custom functions to perform repeated tasks or to use as the objective of an optimization routine. All functions in MATLAB begin with the line of the form `function [out1, out2, ...] = functionname(in1,in2,...)` where *out1*, *out2*, ... are variables the function returns to the command window, *functionname* is the name of the function (which should be unique and not a reserved word) and *in1*, *in2*, ... are input variables. Obviously functions can take multiple inputs sand return multiple outputs. However, to get started, consider this simple function:

```
function y = func1(x)
x = x + 1;
y = x;
```

This function, which isn't particularly well written[1], takes one input and returns one output, incrementing the input variable (whether a scalar, vector or matrix) by one.

Functions have a few important differences from standard m-file scripts.

- Functions operate on a copy of the original data. Thus, the same variable names can be used inside and outside of a function without risking any data.[2]
- Any variables created when the function is running, or any copies of variables made for the function, are lost when the function completes.

In the function above, this means that only the value of y is returned and everything else is lost. Specifically changes in x do not persist. For example, suppose the following was entered in MATLAB

[1] It has no comments and superfluous commands. The function should only contain `y = x +1;` and a description.

[2] For anyone with a programming background, MATLAB uses a copy-on-change model where data is only copied if modified. If unmodified, variables passed to functions behave as if passed by reference.

```
>> x = 1;
>> y = 1;
>> z = func1(x);
>> x
ans =
    1
>> y
ans =
    1
>> z
ans = 2
```

Thus, despite the function using variables names `x` and `y`, the values of `x` and `y` in the workspace do not change when the function is called.

Functions with multiple inputs and outputs can also be constructed. A simple example is given by

```
function [xpy, xmy] = func2(x,y)
xpy = x + y;
xmy = x - y;
```

This function takes two inputs and returns two outputs. It is important to note that despite the two outputs of this function, it does not need to be called using both. For example, consider the following use of this function

```
>> x = 1;
>> y = 1;
>> z1 = func2(x, y);
>> z1
ans =
    2
>> [z1, z2] = func2(x ,y);
>> z1
ans =
    2
>> z2
ans =
    0
```

There are a number of advanced function specific variables available to convey environmental parameters such as how many input variables were provided to the function (`nargin`), how many output were requested (`nargout`), that allow variable numbers of input and outputs (`varargin` and `varargout`, respectively) and allow for early termination of the function (`return`). This course ca be completed without using any of theses. However, they are available if needed them other research.

14.1 Comments

Like batch m-files, comments in custom functions are also made using the % symbol. However, comments have an additional purpose in custom functions. Whenever `help` *function* is entered in the command window, MATLAB will display the first continuous block of comments in the command window. For instance, if the function `func` is given by

```
function y = func(x)
% This function returns
% the value of the input squared.

% The next block of comments will not be returned when
% 'help func' is entered in the Command Window
% This line does the actual work.
y=x.^2;
```

entering `help func` would return

```
>> help func

    This function returns
    the value of the input squared.
```

Initial comments usually contain the possible combinations of input and output arguments as well as a description of the function. While comments are strictly optional, they should be included both to assist in reading the function and to assist others if the function is shared.

14.2 Debugging

Since the data modified in the function is not available when the function is run, debugging can be somewhat difficult. There are three basic strategies to debug a function:

- Write the "function" as a script and then convert it to a proper function.
- Leave off `;` as needed to write out the value of variables to the command window. Alternatively, use `disp`.
- Use `keyboard` and `return` to interrupt the function in order to inspect the values.

The first of these methods is often the easiest. Consider a script version of the function above,

```
x = 1;
y = 2;
%function [xpy, xmy] = func2(x,y)
xpy = x + y;
xmy = x - y;
```

Running this script would be equivalent to calling the function `func2(1,2)`. However, when calling it as a script, variables can be examined as they change. The second method can be useful but is clumsy. Often the output window becomes filled with numbers and find the problematic code may be difficult. The third options is the most advanced. Adding `keyboard` to a function instructs MATLAB to interrupt the function and return control to the keyboard. When in this situation, the usual `>>` prompt changes to a `K>>`. When in keyboard mode, variables inside the function are treated as if they were script variables. Once finished inspecting the variables, enter `return` to continue the execution of the function. A simple example of `keyboard` can be adapted to the function above,

```
function [xpy, xmy] = func3(x,y)
keyboard
xpy = x + y;
xmy = x - y;
keyboard
```

Calling this function will result in an immediate keyboard session (note the `K>>`. Entering `whos` will list two variables, `x` and `y`. When `return` is entered, a second keyboard session open. Entering `whos` will now list four variables, the original two and `xpy` and `xmy`. When a function has been completely debugged, either comment out the `keyboard` commands or remove them entirely.

14.3 Exercises

1. Write a function `summstat` that take one input, a T by K matrix, and returns a matrix of summary statistics of the

```
mean(x(:,1))  std(x(:,1))  skewness(x(:,1))  kurtosis(x(:,1))
mean(x(:,2))  std(x(:,2))  skewness(x(:,2))  kurtosis(x(:,2))
     ⋮            ⋮              ⋮                 ⋮
mean(x(:,K))  std(x(:,K))  skewness(x(:,K))  kurtosis(x(:,K))
```

2. Rewrite the function so that it outputs 4 vectors, one each for `mean`, `std`, `skewness` and `kurtosis`.

3. Write a function called `normloglikihood` that takes two arguments, `params` and `data` (in that order) and returns the log-likelihood of a vector of data. Note: `params` $=$ `[mu sigma2]`$'$ consists of two elements, the mean and the variance.

4. Append to the previous function a second output that returns the score of the log-likelihood (a 2×1 vector) evaluated at `params`.

Chapter 15

Probability and Statistics Functions

MATLAB, through the statistics toolbox, contains an extensive range of statistics function.

15.1 `quantile`

`quantile` returns the empirical quantile of a vector. However, it's function is simple and can easily be replaced using `sort`, `length` and `floor` or `ceil`.

15.2 `prctile`

`prctile` is identical to `quantile` except it expects an argument from 0 to 100 rather than an argument between 0 and 1.

15.3 `regress`

`regress` performs basic regression and returns key regression statistics. I'm not a big fan of MATLAB implementation and recommend writing a custom regression function as an exercise.

15.4 `*cdf`, `*pdf`, `*rnd`, `*inv`

The most valuable code in the statistics toolbox are the CDFs, PDFs, random number generators and inverse CDFs contained within. All common distributions have the complete set of four provided, including

- χ^2 (`chi2-`)
- β (`beta-`)
- Exponential (`exp-`)
- Extreme Value (`ev-`)

- F (`f-`)
- Γ (`gam-`)
- Lognormal (`logn-`)
- Normal (Gaussian) (`norm-`)
- Poisson (`poiss-`)
- Student's t (`t-`)
- Uniform (`unif-`)

15.5 The JPL Toolbox

The JPL toolbox, available from `http:\\www.spatial-ecnometics.com`, contains many econometric functions written by academics. Best of all, it is free. It also has a number of useful plotting functions such as `pltdens` which plots an kernel smooth of an empirical density. I suggest downloading this toolbox before writing a custom own functions to avoid needless reinvention.

15.6 Exercises

1. Have a look through the statistics toolbox in the help browser and explore the functions available.
2. Download the JPL toolbox and extract its contents. Have a look through the list of functions available.

Chapter 16

Optimization

The optimization toolbox contains a number of routines that use numerical techniques to find extremum of user-supplied functions. Most of these implement a form of the Newton-Raphson algorithm which uses derivatives to find the **minimum** of a function. **Note**: MATLAB can *only* find minimums. However, if f is a function to be maximized, $-f$ is a function with the minimum at the same point as the maximum of f.

To use MATLAB to optimize function (for example a log-likelihood or a GMM quadratic form) a custom function that returns the function value at a set of parameters must be constructed. All optimization targets must have the parameters as the first argument. For example consider finding the minimum of x^2. A function which would allows MATLAB's optimizer to work correctly would have the form

```
function x2 = optim_target1(x)

x2=x^2;
```

When multiple parameters (a parameter vector) are used, the objective function must take the form

```
function obj = optim_target2(params)

x=params(1);
y=params(2);

obj= x^2-3*x+3+y*x-3*y+y^2;
```

Optimization targets can have additional inputs,

```
function obj = optim_target3(params,hyperparams)

x=params(1);
y=params(2);

c1=hyperparams(1);
```

```
c2=hyperparams(2);
c3=hyperparams(3);
obj= x^2+c1*x+c2+y*x+c3*y+y^2;
```

This form is particularly useful in econometrics where optimization targets typically require at least two inputs: parameters and data. Once an optimization target function has been specified, the next step is to use one of the MATLAB optimizers find the minimum.

16.1 fminunc

`fminunc` performs derivative based unconstrained minimization. Derivatives can be provided by the user or approximated numerically by MATLAB. The generic form of `fminunc` is
`[p,fval,exitflag]=fminunc('`*fun*`',`p_0`,`*options*, var_1, var_2,...)
where *fun* is the optimization target, p_0 is the vector of starting values, *options* is a user supplied optimization options structure (see 16.5), and var_1, var_2, ... are (optional) variables containing data or other constant values. Typically, three outputs are requested, the parameters at the optimum (`p`), the function value at the optimum (`fval`) and a flag to determine whether the optimization was successful (`exitflag`). For example, suppose

```
function obj = optim_target4(params,hyperparams)

x=params(1);
y=params(2);

c1=hyperparams(1);
c2=hyperparams(2);
c3=hyperparams(3);
obj= x^2+c1*x+c2+y*x+c3*y+y^2;
```

was our objective function (and was saved as *optim_target.m*). To minimize the function, call

```
>> options = optimset('fminunc');
>> options = optimset(options,'Display','iter');
>> p0 = [0 0];
>> hyper = [-3 3 -3];
>> [p,fval,exitflag]=fminunc('optim_target4',p0,options,hyper)
```

which produces

```
[x,fval,exitflag]=fminunc('optim_target4',[0 0],options,hyper)
                                                    First-order
 Iteration  Func-count       f(x)        Step-size       optimality
```

```
     0           3              -3                             3
     1           6              -8       0.333333              2
     2           9             -12              1      1.19e-007
Optimization terminated: relative infinity-norm of gradient less
than options.TolFun.
 x =
     1 1
 fval =
    0
 exitflag =
     1
```

`fminunc` has minimized this function and returns the optimum value of 0 at $x = (1, 1)$ and the `exitflag` has the value 1, indicating the optimization was successful.

16.2 fminsearch

`fminsearch` also performs unconstrained optimization but uses a derivative free method (using a simplex). `fminsearch` uses a virtual amoeba to crawl around in the parameter space that will always move to lower objective function values. `fminsearch` has the same generic form as `fminunc`

`[p,fval,exitflag]=fminsearch('`*fun*`',`p_0`,`*options*, var_1,var_2,...)

where *fun* is the optimization target, p_0 is the vector of starting values, *options* is a user supplied optimization options structure (see 16.5), and var_1, var_2, ... are (optional) variables of data or other constant values. Returning to the previous example but using fminsearch,

```
>> options = optimset('fminsearch');
>> options = optimset(options,'Display','iter');
>> [x,fval,exitflag]=fminsearch('optim_target4',[0 0],options,hyper)
 Iteration   Func-count     min f(x)         Procedure
     0           1               3
     1           3          2.99925         initial simplex
     2           5          2.99775         expand
     3           6          2.99775         reflect
     4           8          2.99475         expand
     5           9          2.99475         reflect
    $\ldots$
    $\ldots$
    $\ldots$
    57          107     8.93657e-009         contract inside
    58          109     3.71526e-009         contract outside
    59          111     1.99798e-009         contract inside
    60          113     5.82712e-010         contract inside
Optimization terminated:
 the current x satisfies the termination criteria using OPTIONS.TolX of 1.000000e-004
 and F(X) satisfies the convergence criteria using OPTIONS.TolFun of 1.000000e-004
 x =
     1.0000    1.0000
 fval =
   5.8271e-010
 exitflag =
     1
```

`fminsearch` requires more iterations and many more function evaluations and should not be used if `fminunc` works satisfactorily. However, for certain problems, such as objective functions which are not differentiable, `fminsearch` may be the only option.

16.3 `fminbnd`

`fminbnd` performs minimization of single parameter problems over a bounded interval using a golden section algorithm. The generic form is

`[p,fval,exitflag]=fminbnd('`*fun*`',`*lb*`,`*ub*`,`*options*`,` var_1`,`var_2`,...)`

where *fun* is the optimization target, *lb* and *ub* are the lower and upper bounds of the parameter, *options* is a user supplied optimization options structure (see 16.5), and var_1, var_2, ... are (optional) variables containing data or other constant values.

Since `fminbnd` only minimizes univariate objectives, consider finding the minimum of

```
function obj = optim_target5(params,hyperparams)

x=params(1);

c1=hyperparams(1);
c2=hyperparams(2);
c3=hyperparams(3);
obj= c1*x^2+c2*x+c3;
```

and optimizing using `fminbnd`

```
>> options = optimset('fminbnd');
>> options = optimset(options,'Display','iter');
>> hyper=[1 -10 21];
>> [x,fval,exitflag]=fminbnd('optim_target5',-10,10,options,hyper)
Func-count     x          f(x)         Procedure
   1       -2.36068      50.1796        initial
   2        2.36068      2.96601        golden
   3        5.27864     -3.92236        golden
   4              5           -4        parabolic
   5        4.99997           -4        parabolic
   6        5.00003           -4        parabolic
Optimization terminated:
 the current x satisfies the termination criteria using OPTIONS.TolX of 1.000000e-004
x =
     5
fval =
    -4
exitflag =
     1
```

16.4 fmincon

`fmincon` performs constrained optimizations using linear and/or nonlinear constraints which can be either equality of inequality constraints. `fmincon` minimizes $f(x)$ subject to any combination of

- $A^{EQ}x = b^{EQ}$
- $Ax \leq b$
- $C^{NEQ}(x) = d^{NEQ}$
- $C(x) \leq d$

where x is K by 1 parameter vector, A is a $Q \times K$ matrix and b is a $Q \times 1$ vector. In the second set of constraints, $C(\cdot)$ is a function from $\mathbb{R}^K$ to $\mathbb{R}^P$ where P is the number of nonlinear constraints and d is a $P \times 1$ vector (EQ and NEQ are simply used to distinguish the equality constraints form the inequality constraints). Note that any $\geq$ constrain can be transformed into a $\leq$ constraint by multiplying by -1.

The generic form of `fmincon` is

`[p,fval,exitflag]=fmincon('`*fun*`',` p_0,*A*,*b*, A^{EQ},b^{EQ},*LB*, *UB*,*nlcon*,*options*,var_1,var_2,...)

where *fun* is the optimization target, p_0 is the vector of starting values, A and A^{EQ} are matrices as described above for inequality and equality constraints, respectively and b and b^{EQ} are conformable vectors as described above. *LB* and *UB* are vectors with the same size as p_0 that contain upper and lower bounds, respectively. **Note**: *LB* and *UB* can always be represented in A and b. For instance, suppose the constraint was $-1 \leq p \leq 1$, then A and b would be

$$A = \begin{bmatrix} -1 \\ 1 \end{bmatrix} \qquad b = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

which are MATLAB expressions for $-p \leq 1$ (which is equivalent to $p \geq -1$) and $p \leq 1$. *nlcon* is a nonlinear constraint function that returns the value of $C(x) - d$ and $C^{NEQ}(x) - d^{NEQ}$ (This is tricky function. See `doc fmincon` for specifics). *options* is a user supplied optimization options structure (see 16.5), and var_1, var_2, ... are (optional) variables of data or other constant values.

Consider the problem of optimizing a CRS Cobb-Douglas utility function of the form $U(x_1, x_2) = x_1^{\lambda} x_2^{1-\lambda}$ subject to a budget constraint $p_1 x_1 + p_2 x_2 \leq 1$. This is a nonlinear function subject to a linear constraint (note, we need $x_1 \geq 0$ and $x_1 \geq 0$. First, specify the optimization target

```
function u = crs_cobb_douglas(x,lambda)

x1=x(1);
x2=x(2);

u=x1^(lambda)*x2^(1-lambda);
u=-u %Must change max problem to min!!!
```

The optimization problem can be formulated in MATLAB by

```
>> options = optimset('fmincon');
>> options = optimset(options,'Display','iter');
>> prices  = [1 1]; %Change this set of parameters as needed
>> lambda  = 1/3;  %Change this parameter as needed
>> A = [-1 0; 0 -1; prices(1) prices(2)]
A =
   -1     0
    0    -1
    1     1
>> b=[0; 0; 1]
b =
    0
    0
    1
>> p0=[.4; .4]; %Must start from a feasible position, usually off the constraint
>> [x,fval,exitflag]=fmincon('crs_cobb_douglas',x0,A,b,[],[],[],[],[],options,lambda)
                              max                   Directional   First-order
 Iter F-count        f(x)   constraint    Step-size   derivative   optimality Procedure
    0      3         -0.4         -0.2
    1      6    -0.529134            0            1       -0.106        0.129
    2      9    -0.529134            0            1   -4.14e-025    2.01e-009
Optimization terminated: first-order optimality measure less
 than options.TolFun and maximum constraint violation is less
 than options.TolCon.
Active inequalities (to within options.TolCon = 1e-006):
  lower      upper     ineqlin   ineqnonlin
                         3
x =
    0.3333
    0.6667
fval =
   -0.5291
exitflag =
    1
```

the `exitflag` value of 1 indicates success. Suppose that dual to the original problem, that of cost minimization, is used instead. In this alternative formulation, the optimization problems becomes

$$\min_{x_1,x_2} p_1x_1 + p_2x_2 \text{ subject to } U(x_1,x_2) \geq \bar{U}$$

Again, to solve this problem in MATLAB, first specify an objective function

```
function cost = budget_line(x,prices,lambda,Ubar)

x1=x(1);
x2=x(2);

p1=prices(1);
p2=prices(2);

cost = p1*x1+p2*x2;
```

Since this problem has a nonlinear constraint, we must specify a *nlcon* function,

```
function [C, Ceq] = compensated_utility(x,prices,lambda,Ubar)

x1=x(1);
x2=x(2);

 u=x1^(lambda)*x2^(1-lambda);

 con=u-Ubar; %Note this is a >= constraint
 C=-con; %This turns it into a <= constraint
 Ceq = []; %No equality constraints
```

Note: The constraint function and the optimization *must* take the same optional arguments in the same order, even if they do not need them. This problem can be solved in MATLAB using

```
>> options = optimset('fmincon');
>> options = optimset(options,'Display','iter');
>> prices  = [1 1]; %Change this set of parameters as needed
>> lambda  = 1/3;  %Change this parameter as needed
>> A = [-1 0; 0 -1] %Note, we still need x1 and x2>=0
A =
   -1     0
    0    -1
>> b=[0; 0]
b =
    0
    0
>> Ubar = .5291;
>> x0 = [1.5;1.5]; %Start with all constraints satisfied, since -1.5+1<0 (-u+ubar).
>> [x,fval,exitflag]=fmincon('budget_line',x0,A,b,[],[],[],[],'compensated_utility',...
                     options,prices,lambda,Ubar)
                                  Max     Line search  Directional  First-order
 Iter F-count        f(x)   constraint   steplength   derivative   optimality Procedure
    0      3           3      -0.9709
    1      6     1.05238   6.451e-005            1        -1.95        0.982
    2     10    0.952732      0.02503          0.5       -0.199        0.083  Hessian modified
    3     13    0.999469    0.0004091            1       0.0467       0.0365
    4     16    0.999653    0.0001502            1     0.000184      0.00127
    5     19    0.999936   1.615e-007            1     0.000283    2.34e-005  Hessian modified
    6     22    0.999936   2.535e-011            1    3.05e-007    1.31e-008  Hessian modified
Optimization terminated: first-order optimality measure less
 than options.TolFun and maximum constraint violation is less
 than options.TolCon.
Active inequalities (to within options.TolCon = 1e-006):
  lower      upper     ineqlin   ineqnonlin
                                     1
x =

    0.3333
    0.6666

fval =
    0.9999

exitflag =
     1
```

These two examples are relatively simple problems where the answers can be analytically verified. Unfortunately, in many it is impossible to verify that the global optimum was been found (for instance, if there are local minima). In these cases, the standard practice is to try many different starting values and use the lowest `fval`. If things are working well, many of the starting values should produce parameter estimates near the others with similar `fval`s.

Note: Many aspects of constrained optimization (and optimization in general) are more black magic than science. Worse, most are problem class specific so general rules are hard to derive. The only way to become proficient at minimizing function is to practice.

16.5 `optimset`

`optimset` sets optimization options and has two distinct forms. The initial call to `optimzet` should always be of the form `options = optimset('fmin`*type*`')` which will return the default options for that type. Once the options structure has been initialized, individual options can be changes by calling
`options = optimset(options,'`*option*$_1$`','`*option value*$_1$`','`*option*$_2$`','`*option value*$_2$`,...)`

For example, to set options for `fmincon`,

```
>> options = optimset('fmincon');
>> options = optimset(options,'MaxFunEvals',1000,'MaxIter',1000);
>> options = optimset(options,'TolFun',1e-3);
```

For help on the available options or their specific meaning, see `doc optimset`.

16.6 Other Optimization Routines

MATLAB's Optimization toolbox contains a number of other optimization algorithms:

`fseminf`	Multidimensional constrained minimization, semi-infinite constraints
`fgoalattain`	Multidimensional goal attainment optimization
`fminimax`	Multidimensional minimax optimization
`lsqlin`	Linear least squares with linear constraints
`lsqnonneg`	Linear least squares with nonnegativity constraints
`lsqcurvefit`	Nonlinear curve fitting via least squares (with bounds)
`lsqnonlin`	Nonlinear least squares with upper and lower bounds
`bintprog`	Binary integer (linear) programming
`linprog`	Linear programming
`quadprog`	Quadratic programming

Chapter 17

Dates and Times

Keeping track of dates is crucial when working with time-series data. MATLAB stores dates as *days since January 1, 0000* known as MATLAB serial dates. For example, January 1, 0000 is 1 in MATLAB date format while January 1, 2000 is 730,486. MATLAB serial dates store hours as fractional days, so 12:00 January 1, 2000 is 730,486.5. The standard method to get dates into MATLAB is to use Excel to produce Excel dates and to add a constant to map Excel dates (which are based on January 1, 1900) to MATLAB dates. However, this isn't always possible, and MATLAB provides a number of useful functions to manipulating date data.

17.1 datenum

`datenum` converts either string dates (`'01JAN2000'`) or numeric dates (`[2000 01 01]`) into MATLAB serial dates. To call the function with string dates, use either `datenum(stringdate)` or `datenum(stringdate,`*format*`)` where format is composed of blocks from

yyyy	Four digit year.
yy	Two digit year, risky since it can assume the wrong century.
mmmm	Full name of month (e.g. January)
mmm	First three letters of month (e.g. JAN)
mm	Numeric month of year
m	Capitalized first letter of month
dddd	Full name of weekday
ddd	First three letters of weekday
dd	Numeric day of month
d	Capitalized first letter of weekday
HH	Hour, should be 24 hour format and padded with 0 if single digit
MM	Minutes, must be padded with extra 0 if single digit
SS	Seconds, must be padded with extra 0 if single digit

MATLAB will automatically recognize most reasonable string date formats. However, the format strings above can be used to handle non-standard cases. They are particularly useful if the arguments appear in a strange order, such as yyyyddmm (e.g. 20000101), or if the dates are delimited using nonstandard characters, such as a ; or , (e.g. 2000;01;01).

A few examples

```
>> datenum('01JAN2000')
ans =
      730486
>> datenum('01JAN2000','ddmmmyyyy')
ans =
      730486
>> datenum('01;JAN;2000','dd;mmm;yyyy')
ans =
      730486
>> datenum('01012000','ddmmyyyy')
ans =
      730486
```

`datenum` also works on string arrays. For example

```
>> strdates=strvcat('01JAN2000','02JAN2000','03JAN2000')
strdates =
01JAN2000 02JAN2000 03JAN2000
>> datenum(strdates)
ans =
      730486
      730487
      730488
```

`datenum` can also be used to convert numeric dates, such as `[2000 01 01]` to MATLAB serial date format. For example,

```
>> datenum([2000 01 01])
ans =
     730486
>> years=[2000;2000;2000];
>> months=[01;01;01];
>> days=[01;02;03];
>> [years months days]
ans =
        2000           1           1
        2000           1           2
        2000           1           3
>> datenum(years,months,days)
ans =
     730486
     730487
     730488
```

`datenum` can also be used to translate hours, minutes and seconds to fractional days, although it should be easy to write code to handle this.

17.2 datestr

`datestr` is the "inverse" of `datenum`. It produces a human readable string of a MATLAB serial date. By default, it will return string dates of the form 'dd-mmm-yyyy'. However, it also knows a number of standard formats such as 'mm/dd/yy' or 'mmm.dd,yyyy'. To produce one of the nonstandard date formats, use `datestr(serial_date, #)` where # corresponds to one of the format strings (see `help datestr` for a list). `datestr` can also produce strings with arbitrary formats using the syntax detailed above (e.g. 'dd; mm; yyyy' to produce a date string with ; delimiters).

```
>> serial_date=datenum('01JAN2000')
serial_date =
      730486
>> datestr(serial_date)
ans =
01-Jan-2000
>> datestr(serial_date,0)
ans =
01-Jan-2000 00:00:00
>> datestr(serial_date,'mmm;dd;yyyy')
ans =
Jan;01;2000
```

Like `datenum`, `datestr` can take vector input and return vector output.

```
>> serial_date=datenum(strvcat('01JAN2000','02JAN2000','03JAN2000'))
serial_date =
     730486
     730487
     730488
>> datestr(serial_date)
ans =
01-Jan-2000
02-Jan-2000
03-Jan-2000
```

17.3 datevec

`datevec` converts MATLAB serial dates into human parsable *numeric* formats. Specifically, given a MATLAB serial date, `datevec` will produce a 1×6 vector of the form [Year Month Day Hour Minute Second]. For example,

```
>> serial_date=datenum(strvcat('01JAN2000','02JAN2000','03JAN2000'))
serial_date =
      730486
      730487
```

```
      730488
>> datevec(serial_date)
ans =
       2000           1           1           0           0           0
       2000           1           2           0           0           0
       2000           1           3           0           0           0
```

which correspond to 0:00 (midnight) on the January 1-3, 2000.

17.4 `now` and `clock`

`now` returns the a MATLAB serial date representation of the computer clock. Clock returns a 1×6 vector (same format as `datevec`) of the computer clock. `datevec(now)` produces the same output as `clock`.

17.5 `tic` and `toc`

`tic` and `toc` can be used for timing when optimizing code. For example,

```
>> tic
>> j=1; for i=1:1000000; j=j+1; end
>> toc
Elapsed time is 0.4867477 seconds.
```

17.6 `etime`

The elapsed time between two calls to `clock` can be computed using `etime`.

```
>> c=clock;
>> j=1; for i=1:10000000; j=j+1; end;
>> e=etime(clock,c)

e =
    0.4830
```

17.7 datetick

`datetick` is not a function for explicitly working with dates. `datetick` converts an axis of a plot expressed in MATLAB serial dates to text dates. For example,

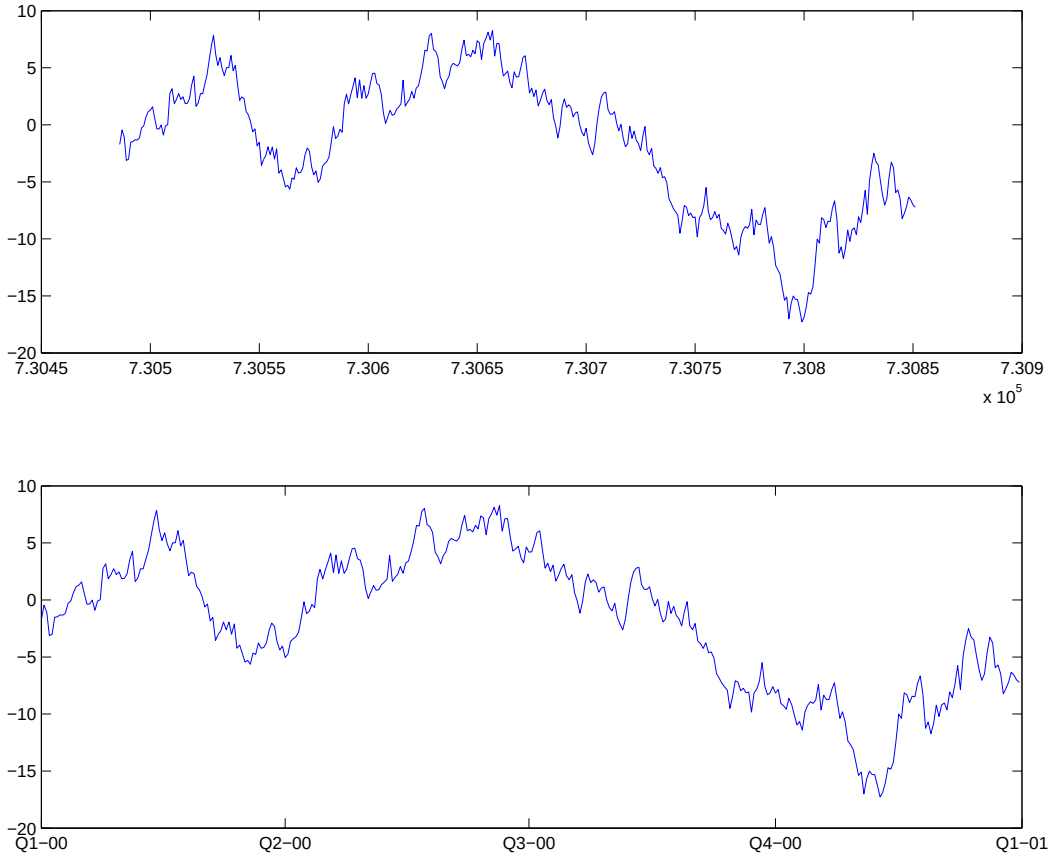


Figure 17.1: Example of `datetick`. `datetick` converts MATLAB serial dates into text strings. Unfortunately, it typically changes the location of points and makes fairly bad choices. The solution is to use `datetick('x','keepticks','keeplimits')`.

```
>> dates = datenum('01Jan2000'):datenum('31Dec2000');
>> rw    = cumsum(randn(size(dates)));;
>> subplot(2,1,1);
>> plot(dates, rw);
>> subplot(2,2,1);
>> plot(dates, rw);
>> datetick('x')
```

produces the two plots in figure 17.1. The top plot contains MATLAB serial dates along the x-axis while the bottom contains sting dates. `datetick` also understands both standard formatting commands (see `datestr`) and custom formatting commands (see `datenum`). This function has an unfortunate tendency to produce few x-labels. The solution is to first choose the axis label points (in serial dates) and than use `datetick('x','keepticks','keeplimits')` as illustrated in figure 17.2.

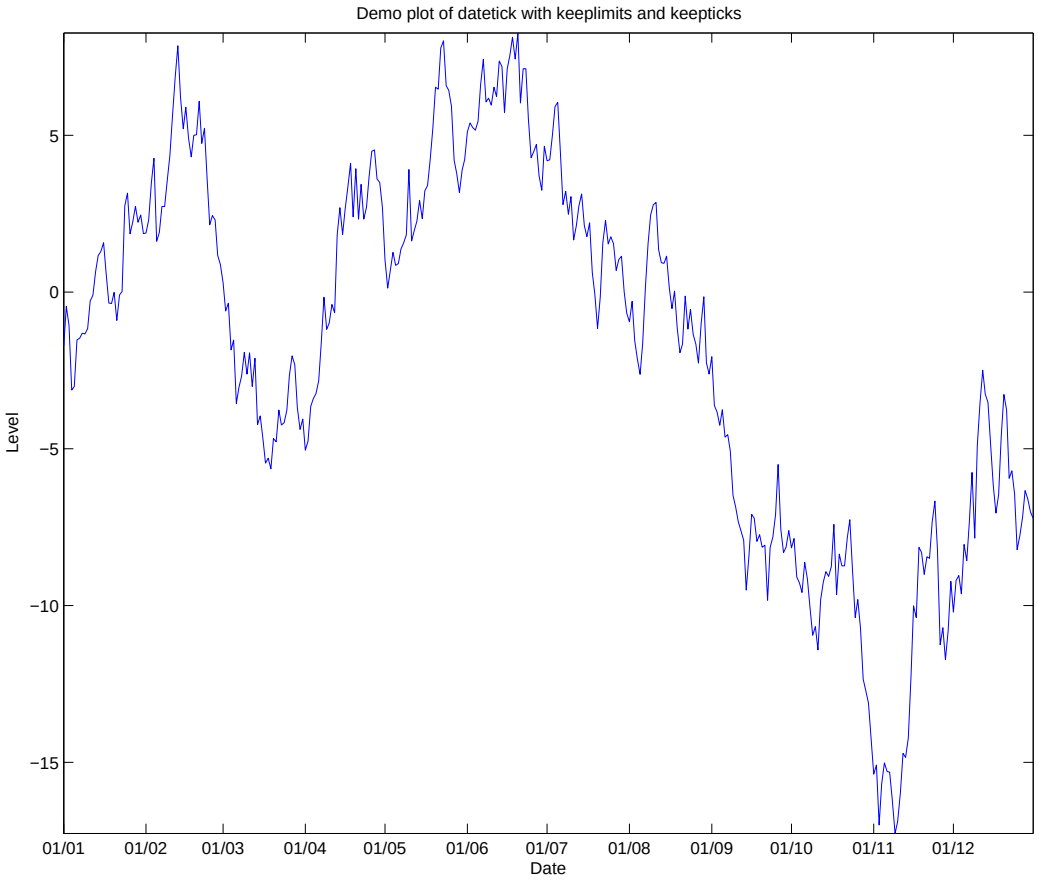


Figure 17.2: `datetick` with `keepticks` and `keeplimits`. These two arguments ensure `datetick` behaves sanely. To use them, set up the figure as is should look but with serial dates, and then call `datetick('x','keepticks','keeplimits')`.

```
>> h=plot(dates, rw);
>> axis tight
>> serial_dates=datenum(strvcat('01/01/2000','01/02/2000','01/03/2000','01/04/2000',...
                               '01/05/2000','01/06/2000','01/07/2000','01/08/2000',...
                               '01/09/2000','01/10/2000','01/11/2000','01/12/2000'),...
                               'dd/mm/yyyy');
>> parent=get(h,'Parent');
>> set(parent,'XTick',serial_dates);
>> datetick('x','dd/mm','keeplimits','keeplimits');
>> xlabel('Date')
>> ylabel('Level')
>> title('Demo plot of datetick with keeplimits and keepticks')
```

Chapter 18

String Manipulation

While manipulating text is not MATLAB's strong suit, it does provide a complete set of tools for working with strings. MATLAB treats strings as matrices of character data. Simple strings can be input from the command line

```
str = 'Econometrics is my favorite subject.';
```

Since character data are contained in matrices, they respect the standard behavior of most commands (e.g. `str(1:10)`). However, using commands designed for numerical data is tedious and MATLAB contains special purpose string functions to assist.

The primary use of string functions in MATLAB is for parsing data. In chapter 3, an example of parsing a poorly formatted file. It uses a number of MATLAB's string manipulation functions to manipulate and parse the text of a file.

`char`

`char` changes integer numerical values between 1 and 255 into their ASCII equivalent characters. Other values produce a nonsense result.

```
>> char(65:100)
ans =
ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_‘abcd
>> char([1.32 pi 256])
ans =
□□□
```

`double`

`double` changes character strings into their numerical values.

```
>> double('MATLAB')
ans =
   77    97   116   108    97    98
```

strvcat

`strvcat` vertically concatenates two strings. In normal math mode, to matrices `x` and `y` can be vertically concatenated by `[x;y]`. However, strings often have different widths which makes concatenation of the difficult. `strvcat` makes this easy.

```
>> strvcat('apple','banana','cherry')
ans =
apple
banana
cherry
>> x=strvcat('alpha','beta');
>> y=strvcat('delta','gamma');
>> strvcat(x,y)
ans =
alpha
beta
delta
gamma
```

strcat

`strcat` horizontally concatenates strings. `z=strcat(x,y)` is the same as `z=[x y]` when `x` and `y` have the same number of rows. If one has a single row, `strcat` concatenates it to every row of the other vector.

```
>> strcat(strvcat('a','b'),strvcat('c','d'))
ans =
ac
bd
>> strcat(strvcat('a','b'),'c')
ans =
ac
bc
```

strfind

`strfind` returns all beginnings of any block of text in another string. It is useful in finding delimiting characters in a block of text to assist in parsing. For example, consider a single line from WRDS TAQ output

```
>> str = 'IBM,02JAN2001,9:30:07,84.5';
>> strfind(str,',')
ans =
     4    14    22
```

`strfind` returns all of the location of `','`. If more than one character is searched for, `strfind` can produce overlapping blocks.

```
>> str = 'abababababa'
str =
abababababa
>> strfind(str,'aba')
ans =
     1     3     5     7
```

`strcmp` and `strcmpi`

`strcmp` compares two strings and returns (logical) 1 if they are the same. It is case sensitive. `strcmpi` does the same but is not case sensitive.

```
>> strcmp('a','a')
ans =
     1
>> strcmp('a','A')
ans =
     0
>> strcmpi('a','A')
ans =
     1
```

`strncmp` and `strncmpi`

`strncmp` compares the first `n` characters of two strings and returns (logical) 1 if they are the same. It is case sensitive. `strcnmpi` does the same but is not case sensitive.

```
>> strncmp('apple','apple1',5)
ans =
     1
>> strncmp('apple','apple1',6)
ans =
     0
>> strncmp('apple','Apple1',5)
ans =
     0
>> strncmpi('apple','Apple1',5)
ans =
     1
```

strmatch

strmatch compares rows of a character matrix with a string and returns the index of all rows that begin with the string. To match only the entire row, use the optional command 'exact'

```
>> str = strvcat('alpha','beta','alphabeta');
>> strmatch('alpha',str)
ans =
     1
     3
>> strmatch('alpha',str,'exact')
ans =
     1
```

regexp and regexpi

regexp is similar to strfind but takes standard regular expression syntax commands to find matches. It is case sensitive. regexpi does the same but is not case sensitive. For examples of regexp, see doc regexp.

str2num

str2num converts string values into numerical varies. The input can be either vector or matrix valued.

```
>> strvcat('1','2','3')
ans =
1
2
3
>> str2num(strvcat('1','2','3'))
ans =
    1
    2
    3
>> str2num(['1 2 3';'4 5 6'])
ans =
     1     2     3
     4     5     6
```

Note the different spacing of strings (no str2num) and numbers.

num2str

num2str converts numerical values into strings. The input can be either vector or matrix valued.

```
>> num2str([1;2;3])
ans =
1 2 3
>> num2str([1 2 3;4 5 6])
ans =
1  2  3
4  5  6
```

18.1 Exercises

1. Load the file hardtoparsetext.mat and inspect the variable string_data. The data in this file are ; delimited and contain stock name, date of observation, shares out standing, and price. Write a program that will loop over the rows and parse the data into four variables: `ticker`,`date`, `shares` and `price`.

Note: Ticker should be a string, date should be a MATLAB serial data, and shares outstanding and price should be numerical. For values of 'N/A', use NaN. For help converting the dates to serial dates, see chapter **??**.

Chapter 19

File System and Navigation

MATLAB uses standard DOS file system commands to change working directories. For instance, to change directory, type

```
cd c:\MyDirectory
```

Other standard DOS file navigation commands, such as `dir` and `mkdir` are also available from within MATLAB.

Alternatively, the current directory can be changed by clicking the button with ... next to the Current Directory box at the top of the command window (see figure 1.1).

19.1 The MATLAB path

While this section sounds like a Buddhist rite of passage, the path is an important set of locations to MATLAB. The path tells MATLAB where to look for files. All of the toolbox directories are automatically on the path, but it may be necessary to add new directories to use custom or a non-standard toolbox.

To see the current path, enter `path` in the command window. Alternatively, there is a GUI path browser available under File>Set Path.... The path is sorted from most important directory to least, with the present working directory (what `pwd` returns in the command window) silently atop the list. The path controls which files MATLAB will use when evaluating a function or running a batch file.

Suppose a custom function is accidentally titled `mean`. When `mean` is entered in the command window, MATLAB will find all occurrences of `mean` on the path and rank them based on their order. It will then execute the highest ranking one. Naturally, existing function names should be avoided. However, mistakes do happen and if worried that the wrong function is being called, `which` *function* `-all` will show all files that match *function* (function, m-files and mat files), returning them in the order they appear on the path.

New directories can be appended to the path using `addpath` or File>Set Path.... The GUI tool provide additional functionality as it can be used to re-rank directories on the path. To save any changes, use the command `savepath` or click on Save Path in the Path GUI.

19.2 Setting up a Custom Path in a Shared Environment

In most shared environments, the MATLAB program directory will be read only and the original MATLAB path cannot be directly altered. To work around this issue, create and save a file named STARTUP.M in U:\MATLAB. MATLAB will automatically look in the startup directory for this file and run it whenever it is launched. This file should contain the following:

```
addpath('U:\Path to Add');
addpath('U:\Second Path to Add');
```

where 'Path to Add' and 'Second Path to Add' are directories to be added to the base path.

19.3 Exercises

1. Use the command window to create a new directory, *chapter 13* (`mkdir`).
2. Change into this directory using `cd`.
3. Create a new file names *tobedeleted.m* using the editor in this new directory (It can be empty).
4. Get the directory listing using `dir`.
5. Add this directory to the path using either `addpath` or the Path GUI. Save the changes using either `savepath` or the Path GUI.
6. Delete the newly created m-file, and then delete this directory from the command line.
7. Remove this folder from the path using either `rmpath` or the Path GUI.

Chapter 20

Quick Function Reference

This is a brief summary of functions that are useful in the course. It only scratches the surface of what MATLAB offers. There are about 100 functions listed here; MATLAB and the Statistics Toolbox combine to produce more than 1400.

20.1 General Math

`abs`

Returns the absolute value of the elements of a vector or matrix. If used on a complex data, returns the complex modulus.

`diff`

Returns the difference between two adjacent elements of a vector. The if the original vector has length T, vector returned has length $T-1$. If used on a matrix, returns a matrix of differences of each column. The matrix returned has one less row than the original matrix.

`exp`

Returns the exponential function (e^x) of the elements of a vector or matrix.

`log`

Returns the natural logarithm of the elements of a vector or matrix. Returns complex values for negative elements.

`log10`

Returns the logarithm base 10 of the elements of a vector or matrix. Returns complex values for negative elements.

max

Returns the maximum of a vector. If used on a matrix, returns a row vector containing the maximum of each column.

mean

Returns the arithmetic mean of a vector. If used on a matrix, returns a row vector containing the mean of each column.

min

Returns the minimum of a vector. If used on a matrix, returns a row vector containing the minimum of each column.

mod

Returns the remainder of division of the elements of a vector or matrix by a scalar.

roots

Returns the roots of a polynomial.

sign

Returns the sign of the elements of a vector or matrix. The sign is defined as $x/|x|$ and 0 if $x = 0$.

sum

Returns the sum of the elements of a vector. If used on a matrix, produces a row vector containing the sum of each column.

20.2 Rounding

ceil

Returns the next larger integer. Output is same size as input and `ceil` operates element-by-element.

floor

Returns the next smaller integer. Output is same size as input and `floor` operates element-by-element.

round

Rounds to the nearest integer. Output is same size as input and `round` operates element-by-element.

20.3 Statistics

corrcoef and corr

Computes the correlation of a matrix. If a matrix x is N by M, returns the M by M correlation treating the columns of x as realizations from separate random variables.

cov

Computes the covariance of a matrix. If a matrix x is N by M, returns the M by M covariance treating the columns of x as realizations from separate random variables. If used on a vector, produces the same output as `var`.

kurtosis

Computes the kurtosis of a vector. If used on a matrix, a row vector containing the kurtosis of each column is returned.

median

Returns the median of a vector. If used on a matrix, a row vector containing the median of each column is returned.

quantile

Computes the quantiles of a vector. If used on a matrix, a row vector containing the quantiles of each column is returned.

skewness

Computes the skewness of a vector. If used on a matrix, a row vector containing the skewness of each column is returned.

std

Computes the standard deviation of a vector. If used on a matrix, a row vector containing the standard deviation of each column is returned.

var

Computes the variance of a vector. If used on a matrix, a row vector containing the variance of each column is returned.

*DIST***pdf**

Returns the probability density function values for a given *DIST*, where *DIST* takes one of many forms such as `t` (`tpdf`), `norm` (`normpdf`), or `gam` (`gampdf`). Inputs vary by distribution.

*DIST***cdf**

Returns the cumulative distribution function values for a given *DIST*, where *DIST* takes one of many forms such as `t` (`tcdf`), `norm` (`normcdf`), or `gam` (`gamcdf`). Inputs vary by distribution.

*DIST***inv**

Returns the inverse cumulative distribution value for a given *DIST*, where *DIST* takes one of many forms such as `t` (`tinv`), `norm` (`norminv`), or `gam` (`gaminv`). Inputs vary by distribution.

*DIST***rnd**

Produces pseudo-random numbers for a given *DIST*, where *DIST* takes one of many forms such as `t` (`trnd`), `norm` (`normrnd`), or `gam` (`gamrnd`). Inputs vary by distribution.
Note: Most *DIST* function are available for the following distributions: Beta, Binomial, χ^2, Exponential, Extreme Value, F, Gamma, Generalized Extreme Value, Generalized Pareto, Geometric, Hypergeometric, Lognormal, Negative Binomial, Noncentral F, Noncentral t, Noncentral χ^2, Normal, Poisson, Rayleigh, t, Uniform, Discrete, Uniform, Weibull,

20.4 Random Numbers

rand

Uniform pseudo-random number generator.

randn

Standard normal pseudo-random number generator.

random

Generic psuedo-random number generator. Can generate random numbers for the following distributions: Beta, Binomial, χ^2, Exponential, Extreme Value, F, Gamma, Generalized Extreme Value, Generalized Pareto, Geometric, Hypergeometric, Lognormal, Negative Binomial, Noncentral F, Noncentral t, Noncentral χ^2, Normal, Poisson, Rayleigh, t, Uniform, Discrete, Uniform, Weibull,

20.5 Logical

all

Returns logical true (1) if all elements of a vector are logical true. If used on a matrix, returns a row vector containing logical true if all elements of each column are logical true.

any

Returns logical true (1) if any elements of a vector are logical true. If used on a matrix, returns a row vector containing logical true if any elements of each column are logical true.

find

Returns the indices of the elements of a vector or matrix which satisfy a logical condition.

20.6 Special Values

ans

`ans` is a special variable that contains the value of the last *unassigned* operation.

eps

`eps` is the numerical precision of MATLAB. Numbers differing by more the `eps` are the same to MATLAB.

Inf

`Inf` represents infinity in MATLAB.

NaN

`NaN` represents not-a-number in MATLAB. It occurs as a results of performing an operation which produces in indefinite result, such as `Inf/Inf`.

20.7 Special Matrices

eye

`z=eye(`N`)` returns a N by N identity matrix.

linspace

`z=linspace(`L`,`U`,`N`)` returns a 1 by N vector of points uniformly spaced between L and U.

logspace

`z=linspace(L,U,N)` returns a 1 by N vector of points logarithmically spaced between 10^L and 10^U.

ones

`z=ones(N,M)` returns a N by M matrix of ones.

zeros

`z=zeros(N,M)` returns a N by M matrix of zeros.

toeplitz

`z=toeplitz(x)` returns a Toeplitz matrix constructed from a vector `x`.

20.8 Matrix Functions

chol

Computes the Cholesky factor of a positive definite matrix.

det

Computes the determinant of a square matrix.

diag

Returns the elements along the diagonal of a square matrix. If the input to `diag` is a vector, returns a matrix with that diagonal.

eig

Returns the eigenvalues and eigenvectors of a square matrix.

inv

Returns the inverse of a square matrix.

kron

Kronecker product of two matrices.

trace

Returns the trace of a matrix, equivalent to `sum(diag(x))`.

tril

Returns a lower triangular version of the input matrix.

triu

Returns a upper triangular version of the input matrix.

20.9 Matrix Manipulation

cat

Concatenates two matrices along some dimension. If `x` and `y` are conformable matrices, `cat(1,x,y)` is the same as `[x; y]` and `cat(2,x,y)` is the same as `[x y]`.

length

Length of the longest dimension of a matrix. In the 2D case, is equivalent to `max(size(x, 1), size(x, 2))`.

numel

Returns the number of elements in a matrix. If the matrix is 2D with dimensions N and M, `numel` returns NM.

repmat

Replicates a matrix according to the dimensions provided.

reshape

Reshapes a matrix to have a different size. The product of the dimensions must be the same before and after, hence the number of elements cannot change.

size

Returns the dimension of a matrix. Dimension 1 is the number of rows and dimension 2 is the number of columns.

20.10 Set Functions

intersect

Returns the intersection of two vectors. Can be used with optional `'rows'` argument and same-sized matrices to produce an intersection of the rows of the two matrices.

setdiff

Returns the difference between the elements of two vectors. Can be used with optional `'rows'` argument and same-sized matrices to produce a matrix containing difference of the rows of the two matrices.

union

Returns the union of two vectors. Can be used with optional `'rows'` argument and same-sized matrices to produce an union of the rows of the two matrices.

unique

Returns the unique elements of a vector. Can be used with optional `'rows'` argument and a matrix to produce the unique rows of the matrix.

sort

Produces a sorted vector from smallest to largest. If used on a matrix, operates column-by-column.

sortrows

Sorts the rows of a matrix using lexicographic ordering, similar to alphabetizing words.

20.11 Flow Control

case

Command which can be evaluated to logical true or false in a `switch ... case ... otherwise` flow control block.

else

Command that is the default in `if ... elseif ... else` flow control blocks. If none of the `if` or `elseif` statement are evaluated to logical true, the `else` path is followed.

elseif

Command that is used to continue a `if ... elseif ... else` flow control block. Should be immediately followed by a statement that can be evaluated to logical true or false.

end

Command indicating the end of a flow control block. Both `if ... elseif ... else` and `switch ... case ... otherwise` must be terminated with an end. Also ends loops.

if

Command that is used to begin a `if ... elseif ... else` flow control block. Should be immediately followed by a statement that can be evaluated to logical true or false.

switch

Command signalling the beginning of a `switch ... case ... otherwise` flow control block. Switch should be followed by a variable contained in the case.

20.12 Looping

continue

Command that exits the current loop and continues the program at the next (outside of loop) line.

end

All loop blocks in MATLAB must be terminated by an end command. Also ends flow control blocks.

for

One of two types of loops available in MATLAB. For loops loop over a predefined vector unless prematurely ended by a `break` or `continue` command.

while

One of two types of loops available in MATLAB. While loops continue until some logical condition is evaluated to logical false (0) unless prematurely ended by a `break` or `continue` command.

break

Can be used to prematurely break out of a loop before the remainder of the code in the loop has been executed.

continue

Can be used to proceed to the next iteration of a loop while bypassing any code occurring after the continue statement.

20.13 Optimization

`fmincon`

Constrained function minimization using a gradient based search. Constraints can be linear or non-linear and equality or inequality.

`fminbnd`

Function minimization with bounds. Find the minimum of a function that exists between L and U.

`fminsearch`

Function minimization using a simplex (derivative-free) search.

`fminunc`

Unconstrained function minimization using a gradient based search.

`optimget`

Gets options structure for optimization.

`optimset`

Sets options structure for optimization.

20.14 Graphics

`axis`

Sets or gets the current axis limits of the active figure. Can also be used to tighten limits using the command `axis tight`.

`bar`

Produces a bar plot of a vector or matrix.

`bar3`

Produces a 3D bar plot of a vector or matrix.

`contour`

Produces a contour plot of the levels of z data against vectors of x and y data.

errorbar

Produces a plot of x data against y data with error bars (confidence sets) around each point.

figure

Opens a new figure window. When used with a number, for example `figure(1)` opens a window with label Figure 1. If a windows with label Figure 1 is already open, sets that figure as the active figure.

hist

Produces a histogram of data. Can also be used to compute bin centers and height.

legend

Produces a legend of elements of a plot.

mesh

Produces a 3D mesh plot of a matrix of z data against vectors of x and y data.

plot

Plots x data against y data.

plot3

Plots z data against x and y data in a 3D setting.

scatter

Produces a scatter plot of x data against y data.

subplot

Command that allows for multiple plots to be graphed on the same figure. Used in conjunction with other plotting commands, such as `subplot(2,1,1); plot(x,y); subplot(2,1,2); plot(y,x);`

surf

Produces a 3D surface plot of a matrix of z data against vectors of x and y data.

title

Produces a text title at the top of a figure.

xlabel

Produces a text label on the x-axis of a figure.

ylabel

Produces a text label on the y-axis of a figure.

zlabel

Produces a text label on the z-axis of a figure.

20.15 Date Functions

date

Returns string with current date.

datenum

Converts string dates, such as 1-Jan-1900, to MATLAB serial (numeric) dates.

datestr

Converts serial dates to string dates.

datetick

Converts axis labels in serial dates to string labels in plots.

datevec

Parses date numbers and date strings and returns date vectors of the form `[YEAR MONTH DATE HOUR MIN SEC]`.

tic

Begins a tic-toc timing loop. Useful for determining the amount of time required to run a section of code.

toc

Ends a tic-toc timing loop.

clock

Returns the current date and time as a 6 by 1 numeric vector of the form [YYYY MM DD HH MM SS].

etime

Can be used to compute the elapsed time between two readings from clock.

now

Returns the current time as a MATLAB serial date.

20.16 File System

cd

Change directory. When used with a directory, changes the working directory to that directory. When called as `cd ..`, changes the working directory to its parent. If the desired directory has a space, use the function version `cd('c:\dir with space\dir2\dir3')`.

delete

Deletes a file from the present working directory. Warning: This command is dangerous; any deleted file is permanently gone and **not** in the Recycle Bin.

dir

Returns the contents of the current working directory.

mkdir

Creates a new child directory in the present working directory.

pwd

Returns the path of the present working directory.

rmdir

Removes a child directory in the present working directory. Child directory must be empty.

20.17 MATLAB Specific

clc

Clears the command window.

clear

Clears variables from memory. `clear` and `clear all` remove all variables from memory, while `clear var1 var2 ...` removes only those variables listed.

clf

Clears the contents of a figure window.

close

Closes figure windows. Can be used to close all figure windows by calling `close all`.

format

Changes how numbers are represented in the command windows. `format long` shows all decimal places while `format short` only shows up to 5. `format short` is the default.

help

Displays inline help for calling a function. Also can be used to list the function in a toolbox (`help` *toolbox*) or to list toolboxes (`help`).

keyboard

Allows functions to be interrupted for debugging. After verifying function operation, use `return` to continue running.

helpbrowser

Opens the integrated help system for MATLAB at the last viewed page.

helpdesk

Opens the integrated help system for MATLAB at the home page.

doc

When used as `doc` *function name*, opens the help browser to the documentation of *function name*.

realmax

Returns the largest number MATLAB is capable of represented. Larger numbers are `Inf`.

realmin

Returns the smallest positive number MATLAB is capable of representing. Numbers closer to 0 are 0.

which

When used in combination with a function name, returns full path to function. Useful if there may be multiple functions with same name on the MATLAB path.

whos

Returns a list of all variables in memory along with a description of type and information on size and memory requirements.

profile

Built-in MATLAB profiler. Reports code dependencies, timing of executed code and provides tips for improving the performance of m-files.

20.18 Input/Output

csvread

Reads variables in .csv files into MATLAB. Requires all data be numeric.

csvwrite

Saves variables from MATLAB to a .csv file.

load

Generally used to load the contents of a MATLAB data file (.mat) into the current workspace. Can also be used to load simple text files.

save

Generally used to save variables to a MATLAB data file (.mat). Can also be used to save tab delimited text files. Can be combined with `-ascii -double` to produce tab delimited text files.

xlsfinfo

Returns information about an .xls file, such as sheet names.

xlsread

Reads variables in .xls files into MATLAB. All data should be numeric, although it does contain methods which allow for text to be read.

`xlswrite`

Saves variables from MATLAB to an .xls file.

Index