Econometric and Statistical Analysis in MATLAB: Revision 3.1 (R2017b)[1]

Kevin Sheppard
University of Oxford

Tuesday 10th September, 2019

[1] Formerly *Financial Econometrics MFE MATLAB Notes*.

©2019 Kevin Sheppard

Changes in Version 3.1

- Verified code works correctly in R2017b.

Changes in Version 3

- Working with heterogeneous data using `table`s, which provides a data structure that can easily handle heterogeneous data (e.g., strings, numbers and dates)
- `varfun`, `rowfun`, `findgroups` and `splitapply` for computing function on grouped data
- The new `datetime` format as a replacement for serial dates, including `duration`s and `calendarDuration`s for working with `datetime`s and `NaT` (not a time)
- `categorical` arrays for optimized storage or repetitive strings
- `strsplit`, `strjoin` for splitting and joining strings.
- `timeit` for simple measuring of performance
- Moving Statistics Functions: Calculate moving statistics using the `movmean`, `movsum`, `movmedian`, `movmax`, `movmin`, `movvar`, and `movstd` functions
- Additional functions that work cumulatively in an array, `cummin` and `cummax`

Changes in Version 2

- bsxfun, which provides a high-performance method to perform math on arrays with different dimensions, has been added to chapter 6.
- nan, which similar to zeros and ones, has been added to chapter 5.
- The use of ~ to suppress outputs of functions is discussed in chapter 17.
- A new chapter (22) containing an extensive set of complete examples has been added.
- Removed references to textread which is now depreciated.
- A new chapter covering the basics of parallel programming in MATLAB has been added. This chapter covers two scenarios. The first considers parallel coding when the parallel toolbox is available and the second discusses methods to achieve simple parallelism using the file system.
- All code has been tested on R2012a, the current release at the time of writing.
- A major rewrite of chapter 14 on importing data. MATLAB's importer has improved substantially over the past few years and importing data is now much simpler. The chapter also covers some useful improvements to load which allow for selective loading from a mat file containing more than one variable.
- Combined the chapters covering control flow with the chapter covering loops for both improved organization and a reduction in the chapter count.
- Combined the exporting graphics chapter with the plotting chapter.
- Discussion of the performance benefits of global variables in chapter 21.
- Added sscanf to chapter 12 as a high-performance alternative to str2double.
- Added the string formatting functions sprintf and fprintf to chapter 12

Contents

1 Introduction to MATLAB **1**

1.1 The Interface 1
1.2 The Editor 1
1.3 Help 4
1.4 Demos 5
1.5 Exercises 5

2 Basic Input **7**

2.1 Variable Names 7
2.2 Entering Vectors 8
2.3 Entering Matrices 9
2.4 Higher Dimension Arrays 9
2.5 Empty Matrices ([]) 9
2.6 Concatenation 10
2.7 Accessing Elements of Matrices 10
2.8 Calling Functions 12
2.9 Exercises 13

3 Basic Math **15**

3.1 Operators 15
3.2 Matrix Addition (+) and Subtraction (-) 15
3.3 Matrix Multiplication (*) 16
3.4 Matrix Left Division (\) 16
3.5 Matrix Right Division (/) 17
3.6 Matrix Exponentiation (^) 17
3.7 Parentheses 17
3.8 Dot (.) Operations 17
3.9 Transpose 18
3.10 Operator Precedence 18
3.11 Exercises 18

4 Basic Functions **21**

4.1 Moving window functions 29

4.2 Exercises . . . 29

5 Special Vectors and Matrices **31**
5.1 Exercises . . . 33

6 Matrix Functions **35**
6.1 Matrix Manipulation . . . 35
6.2 Broadcastable Operations: `bsxfun` . . . 37
6.3 Linear Algebra Functions . . . 38

7 Inf, NaN and Numeric Limits **39**
7.1 Exercises . . . 40

8 Logical Operators **41**
8.1 >, >=, <, <=, == , ~= . . . 41
8.2 & (AND), | (OR) and ~ (NOT) . . . 42
8.3 Logical Indexing . . . 42
8.4 Logical Functions . . . 43
8.5 Exercises . . . 46

9 Control Flow **47**
9.1 Choice . . . 47
9.2 Loops . . . 50
9.3 Exception Handling . . . 54
9.4 Exercises . . . 55

10 Graphics **57**
10.1 Support Functions . . . 57
10.2 2D Plotting . . . 57
10.3 3D Plotting . . . 63
10.4 Multiple Graphs . . . 65
10.5 Advanced Graphics . . . 66
10.6 Exporting Plots . . . 69
10.7 Exercises . . . 73

11 Dates and Times **75**
11.1 MATLAB `datetimes` . . . 75
11.2 MATLAB Serial Dates . . . 77
11.3 Converting between `datetimes` and Serial Dates . . . 81
11.4 Dates on Figures . . . 81

12 String Manipulation **85**
12.1 String Functions . . . 85
12.2 String Conversion . . . 88

12.3 Exercises 90

13 Structures and Cell Arrays **91**
13.1 Structures 91
13.2 Cell Arrays 92

14 Importing and Exporting Data **95**
14.1 Robust Data Importing 95
14.2 Importing Data in Code 96
14.3 MATLAB Data Files (.mat) 101
14.4 Advanced Data Import 102
14.5 Exporting Data 104
14.6 Exercises 105

15 Working with Heterogeneous Data **107**
15.1 Creating `tables` 107
15.2 Features of `tables` 110
15.3 Column data types 111
15.4 Selection 113
15.5 Table-specific features 114

16 Probability and Statistics Functions **119**
16.1 Distributions: `*cdf`, `*pdf`, `*rnd`, `*inv` 119
16.2 Selected Functions 119
16.3 The MFE Toolbox 120
16.4 Exercises 120

17 Custom Functions **121**
17.1 Function-specific functions 122
17.2 Comments 124
17.3 Debugging 124
17.4 Exercises 125

18 Simulation and Random Number Generation **129**
18.1 Core Random Number Generators 129
18.2 Replicating Simulation Data 129
18.3 Considerations when Running Simulations on Multiple Computers 130
18.4 Advanced Random Number Generator 130

19 Optimization **131**
19.1 Unconstrained Derivative-based Optimization 132
19.2 Unconstrained Derivative-free Optimization 133
19.3 Bounded scalar optimization 133
19.4 Constrained Derivative-based Optimization 134

19.5 Optimization Options . . . 138
19.6 Other Optimization Routines . . . 138

20 Accessing the File System 139
20.1 Addressing the File System Programmatically . . . 139
20.2 Running Other Programs . . . 141
20.3 The MATLAB Path . . . 142
20.4 Exercises . . . 142

21 Performance and Code Optimization 145
21.1 Just-in-time Compilation . . . 145
21.2 Suppress Printing to Screen Using ; . . . 145
21.3 Pre-allocate Data Arrays . . . 145
21.4 Avoid Operations that Require Allocating New Memory . . . 146
21.5 Use Vector and Matrix Operations . . . 147
21.6 Vectorize Code . . . 148
21.7 Use Pre-computed Values in Optimization Targets . . . 148
21.8 Use M-Lint . . . 149
21.9 timeit . . . 149
21.10 Profile Code to Find Hot-Spots . . . 149
21.11 Using Global Variables . . . 150
21.12 In-place Evaluation . . . 151

22 Examples 153
22.1 Estimating the Parameters of a GARCH Model . . . 153
22.2 Estimating the Risk Premia using Fama-MacBeth Regressions . . . 157
22.3 Estimating the Risk Premia using GMM . . . 160
22.4 Outputting LaTeX . . . 162

23 Parallel MATLAB 167

24 Quick Function Reference 169
24.1 General Math . . . 169
24.2 Rounding . . . 170
24.3 Statistics . . . 171
24.4 Random Numbers . . . 172
24.5 Logical . . . 173
24.6 Special Values . . . 174
24.7 Special Matrices . . . 174
24.8 Vector and Matrix Functions . . . 175
24.9 Matrix Manipulation . . . 176
24.10 Set Functions . . . 177
24.11 Flow Control . . . 177
24.12 Looping . . . 178

24.13 Optimization . . . 178
24.14 Graphics . . . 179
24.15 Date Functions . . . 181
24.16 String Function . . . 182
24.17 Trigonometric Functions . . . 183
24.18 File System . . . 184
24.19 MATLAB Specific . . . 184
24.20 Input/Output . . . 186

Chapter 1

Introduction to MATLAB

These notes provide an introduction to MATLAB with an emphasis on the tools most useful in econometrics and statistics. All topics relevant to the MFE curriculum should be covered but if any relevant 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. MATLAB keywords such as `if`, `for` and `break` are highlighted in blue and existing MATLAB functions such as `sum`, `abs` and `plot` are highlighted in cyan. In general, both keywords and standard function names should not be used for variable names, although only keywords are formally excluded from being redefined. Strings are highlighted in purple, and comments are in green. The double arrow symbol >> is used to indicate the MATLAB command prompt – it is also the symbol used in the MATLAB command window. *Math* font is used to denote algebraic expressions.

For more information on programming in MATLAB, see *MATLAB: An Introduction with Applications* by Amos Gilat (ISBN:0470873736), *Matlab: A Practical Introduction to Programming and Problem Solving* by Stormy Attaway (ISBN: 0128045256) or *Mastering MATLAB 8* by Bruce L. Littlefield and Duane C. Hanselman (ISBN: 0136013309). The first book provides more examples for beginners, the second is similar to this set of notes while the final is comprehensive, ranging from basic concepts to advanced applications, and was the first book I used – back when the title was *Mastering MATLAB 5*.

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 **A**, is where commands are entered, functions are called and m-files – batches of MATLAB commands – are run. The current directory window, labeled **B**, shows the files located in the current directory. Normally these will include m- and data-files. On the left side of the command window is the workspace (**C**), which contains a list of the variables in memory, such as data loaded or variables entered in the command window. The workspace can be customized using the Home>Layout, and other available panes include command history or an integrated editor.

1.2 The Editor

MATLAB contains a syntax-aware editor that highlights code to improve readability, provides limited error checking and encourages best practices, such as using a semicolon at the end of each statement. The

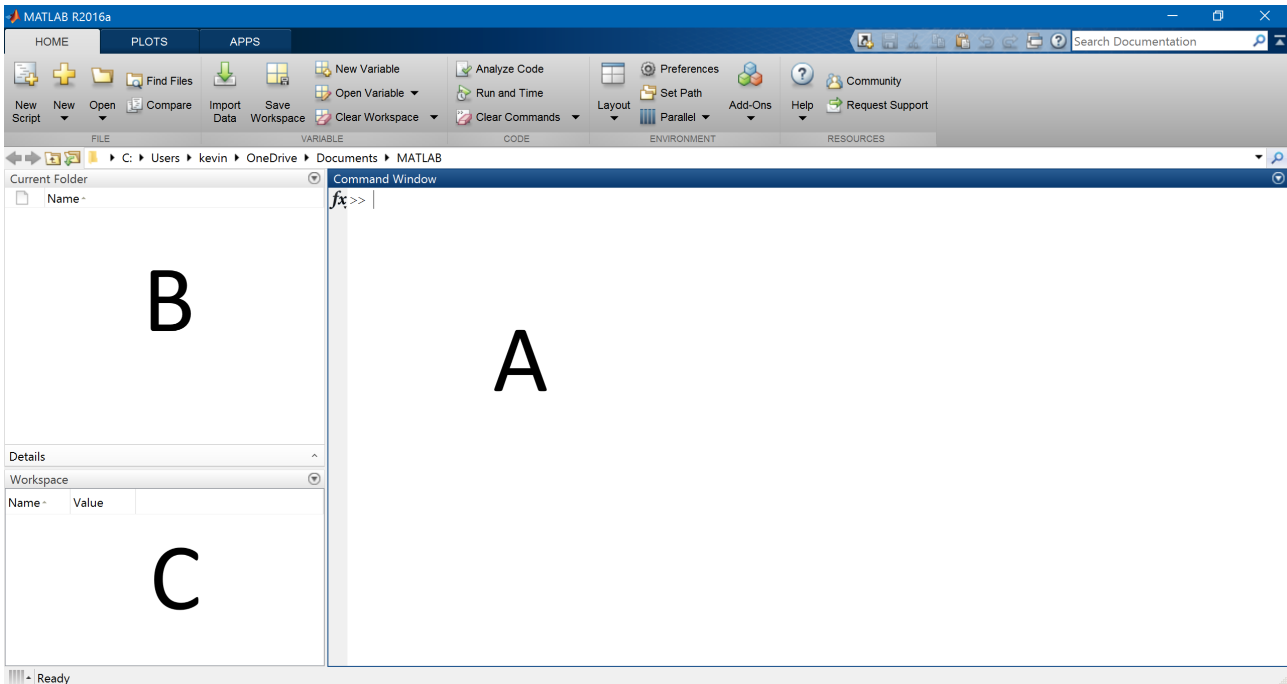

Figure 1.1: Basic MATLAB Window. The standard setup has four panes. **1**: The Command Window, **2**: Current Directory, **3**: Workspace, and **4**: Command History

editor can be launched from the main window in one of two ways, either by clicking Home>New Script or entering `edit` into the command window directly. Figure 1.2 contains an example of the editor and shows the syntax highlighting.

M-files may contain either lists of commands or complete functions (but not both).[1] M-file names can include letters, numbers, and underscores, although they must begin with a letter. Names should be distinct from 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 command `which` *filename* can be used to list the file which would be executed use if *filename* was entered in the command window.[2]

```
>> which for
  built-in (C:\Program Files\MATLAB\R2012a\toolbox\matlab\lang\for)
>> which mean
  C:\Program Files\MATLAB\R2012a\toolbox\matlab\datafun\mean.m
>> which mymfile
  'mymfile' not found.
```

To check whether an existing file duplicates the name of an existing function, use the command `which` *filename* `-all` to produce a list of all matching files.

```
>> which mean -all
C:\Program Files\MATLAB\R2012a\toolbox\matlab\datafun\mean.m
C:\Program Files\MATLAB\R2012a\toolbox\finance\ftseries\@fints\mean.m
```

[1] MATLAB also supports the object-oriented programming paradigm, which allows for richer structure within an m-file. OOP, while useful in large, complex code bases, requires a deeper understanding of programming and is not essential for solving econometric problems.

[2] The exact path will depend on both the the version of MATLAB used and the underlying operating system.

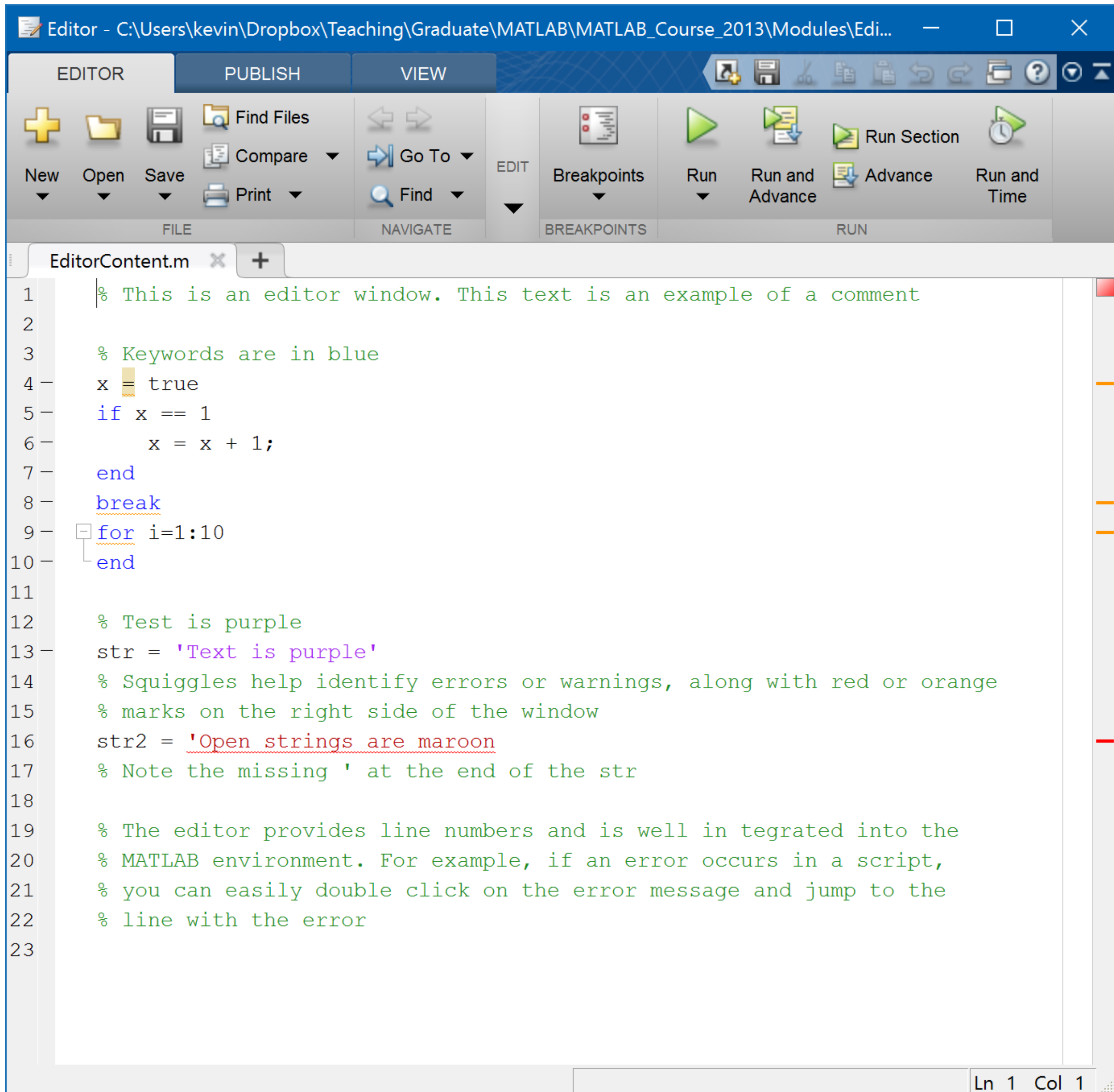

Figure 1.2: MATLAB editor. The editor is a useful programming tool. It can be used to create batch files or custom functions (both called m-files). Note the syntax highlighting emphasizing the different types of commands and data.

```
C:\Program Files\MATLAB\R2012a\toolbox\stats\stats\@ProbDistUnivParam\mean.m
C:\Program Files\MATLAB\R2012a\toolbox\matlab\timeseries\@timeseries\mean.m
```

When multiple files exist on the MATLAB path with the same name, the first listed will be executed.

;

The semicolon (;) is used at the end of a line to suppress the display of the result of a command. To understand 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 may be useful to omit the semicolon until the code is performing as expected.

Comments

Comments assist in tracking completed tasks, documenting unique approaches to solving a difficult problem and are useful if the code needs to be shared. The percentage symbol (%) is used to identify a comment. When a % is encountered, processing stops on the current line and continues on the next line. Block comments are not supported and so 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 comment
```

... (dot-dot-dot)

... is a special expression that can be used to break a long code expression across multiple lines in an m-file. ... concatenates the next line onto the end of the present line when processing, and exists purely to improve the readability of code. These two expressions are identical to the MATLAB interpreter.

```
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 contains a comprehensive help system which is available both in the command window and in a separate browser. The browser-based help is typically more complete and is both indexed and searchable.

Two types of help are available from the command line: toolbox and function. Toolbox help returns a list of available functions in a toolbox. It can be called by `help` *toolbox* where *toolbox* is the short 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* (e.g. `help mean`).
The help browser can be accessed by hitting the F1 key, selecting Help>Full Product Family Help at the top of the command window, or entering `doc` in the command window. The documentation of a specific function can be directly accessed using `doc` *function* (e.g. `doc mean`).

1.4 Demos

MATLAB contains an extensive selection of demos. To access the list of available demos, 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 does not require users to directly manage memory and so variables can be input with no setup. The generic form of a MATLAB expression is

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

and expressions are processed by assigning the value on the right to the variable on the left. For instance,

```
x = 1;
y = x;
x = exp(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 `exp(y)` to `x`. Assigning one variable to another assigns the *value* of that variable, not the variable itself – changes to `y` will not be reflected in the value of `x` in `y = 1` and `x = y`.

```
>> 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 legal names.

2.1.1 Keywords

Like all programming languages, MATLAB has a list of reserved keywords which cannot be used as variable names. The current list of keywords is

```
break case catch classdef continue else elseif end for function
global if otherwise parfor persistent return spmd switch try while
```

2.2 Entering Vectors

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

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

is entered as

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

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

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

is entered using a less intuitive structure

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

where `;` is interpreted as *new row* when used inside square brackets ([ ]).

[1] An important exception to the "everything is a matrix" rule occurs in cell arrays, which are matrices composed of other matrices (formally arrays of arrays or ragged (jagged) arrays). See chapter 13 for more on the use of and caveats to cell arrays.

2.3 Entering Matrices

Matrices are essentially a column vector composed of row vectors. For instance, to construct

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

enter the matrix one row at a time, separating the rows with semicolons,

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

Note that it is *not* necessary to use brackets to denote the inner row vectors.

2.4 Higher Dimension Arrays

Multi-dimensional (N-dimensional) arrays are available for N up to about 30, depending on the size of each matrix dimension. Higher dimensional arrays are particularly useful for storing panel data – time series of cross-sections, such as a time-varying covariance. Unlike scalars, vectors and matrices, higher dimension arrays cannot be directly allocated and are typically constructed by calling functions such as `zeros(2, 2, 2)`.

2.5 Empty Matrices (`[]`)

An empty matrix contains no elements, `x = []`. Empty matrices may be returned from functions in certain cases (e.g. if some criteria is not met). Empty matrices often cause problems, occasionally in difficult to predict ways, although they do 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
```

is a legal operation that builds a 3-element vector by concatenating the previous value with a new value. This type of concatenation is bad from a code performance point-of-view and so it should generally be avoided by pre-allocating the data array using `zeros` (see page 33), if possible. Second, empty matrices are needed for calling functions when multiple inputs are required but some are not used. For example, `std(x,[],2)` uses `x` as the first argument, `2` as the third and leaves the second empty.

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

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

Suppose

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

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

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

z can be defined in a natural way:

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

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

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

Note that concatenating is the code equivalent of block-matrix forms in standard matrix algebra.

2.7 Accessing Elements of Matrices

Once a vector or matrix has been constructed, it is important to be able to access the elements individually. Data in matrices is stored in *column-major order*. This means elements are indexed by first counting down rows and then across columns. For example, 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 notation works well if `x` is a vector, in which case the indices correspond directly to the order of the elements. However single index notation can be tedious when `x` is a matrix, and double indexing of matrices is available using the notation `x(r,c)` where `r` and `c` are the row and column indices, respectively.

```
>> 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 using one index for each dimension, `x(#, #, #)`. For example, `x(1,2,3)` would return the element in the first row of the second column of the third panel.

The colon operator (`:`) plays a special role in accessing elements. It is interpreted as *all elements in that dimension*. For example, `x(:,1)`, returns all elements from matrix `x` in column 1. Similarly, `x(2,:)`

returns all elements from x in row 2. Double : notation produces all elements of the original matrix – 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 =
     4     5     6
>> x(:,:)
ans =
    1     2     3
    4     5     6
    7     8     9
>> x
x =
     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
```

end

end is a keyword which has a number of uses. One of the uses is to automatically select the final element in an array when using a slice. Suppose x is a 2 by 3 matrix. x(1,2:end) is the same as x(1,2:3). The advantage of end is that it will automatically select the last index in a particular dimension without needing to know the array size.

2.8 Calling Functions

Functions calls have different conventions other expressions. The most important 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*, ...]=functionname(*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 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 required. For example, the following are equivalent,

  ```
  y = var(x);
  mean(y)
  ```

  and

  ```
  mean(var(x))
  ```

- Inputs can contain only selected elements of a matrix or vector (e.g. `mean(x([1 2] ,[1 2]))`).

Details of important function calls will be clarified as they are encountered.

2.9 Exercises

1. Input the following mathematical expressions into MATLAB.

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

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

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

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

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

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

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

Basic Math

Mathematical operations in MATLAB code closely follow the rules of linear algebra. Operations legal in linear algebra are legal in MATLAB; operations that are not legal in linear algebra are not legal in MATLAB. For example, matrices must be conformable along their inside dimensions to be multiplied – attempting to multiply nonconforming matrices produces an error.

3.1 Operators

These standard operators are available:

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.

3.2 Matrix Addition (+) and Subtraction (-)

Addition and subtraction 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 $z_{ij} = x_{ij} + y_{ij}$

These conform to the standard rules of matrix addition and subtraction. x_{ij} is the element from row i and column j of x.

3.3 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.

3.4 Matrix Left Division (\)

Matrix division is not defined in linear algebra. The intuition for the definition of matrix division in MATLAB follows from solving a set of linear equations. Suppose there is some z, a M by L vector, such that

$$xz = y$$

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

x \ y	Scalar	Matrix
Scalar	Any $z = \frac{y}{x}$	Any $z_{ij} = \frac{y_{ij}}{x}$
Matrix	N/A –	Left Dimensions Match $z = (x'x)^{-1}x'y$

Note: Like linear regression, matrix left division is only well defined if x is nonsingular (has full rank).

3.5 Matrix Right Division (/)

Matrix right division is simply the opposite of matrix right division, and `z = y/x` is identical to `z = (x'\y')'`, and so there is little reason to use matrix right division. Suppose `z = y/x`:

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

3.6 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 Not useful
x	Matrix	x Square $z = x^y$	N/A

In the case where x is a matrix and y is an integer, and `z=x*x* ... *x` (y times). If y is not an integer, this function involves eigenvalues and eigenvalues.[1]

3.7 Parentheses

Parentheses can be used in the usual way to control the order in which mathematical expressions are evaluated, and can be nested to create complex expressions. See section 3.10 on Operator Precedence for more information on the order mathematical expressions are evaluated.

3.8 Dot (.) Operations

The . operator (read dot operator) changes matrix operations into element-by-element operations. 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(i,j) = x(i,j) * y(i,j)`. Multiplication (.*), division (./), right division (.\), and exponentiation (.^) all have dot 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)
```

[1] If x is a scalar and y is a real symmetric matrix, then `x^y` is defined as `V * diag(x.^diag(D))*V'` where V is the matrix of eigenvectors and D is a diagonal matrix containing the corresponding eigenvalues of y.

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

3.9 Transpose

Matrix transpose is expressed using the `'` operator. For instance, if `x` is an M by N matrix, `x'` is its transpose with dimensions N by M.

3.10 Operator Precedence

Computer math, like standard math, has operator precedence which determined how mathematical expressions such as

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

are evaluated. The order of evaluation is:

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

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

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. `-4^2` produces -16 since ^ has higher precedence than unary negation and so is interpreted as `-(4^2)`. `-4 * -4` produces `16` since it is interpreted as `(-4) * (-4)` because unary negation has a higher precedence than multiplication.

3.11 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 4

Basic Functions

This chapter discusses a set of core functions which are frequently encountered.

`length`

`length` returns the size of the *maximum* dimension of a matrix. If y is T by K, $T > K$, then `length(x)` is T. If $K > T$, the length is K. Using `length` is risky since the value returned can be either the number of columns or the number of rows, depending on which is larger.[1] In practice, `size` should be used since the dimension can be explicitly provided.

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

`size`

`size` returns the size of either a particular dimension or the entire array. To determine the size of a particular dimension, use `z=size(x,`*DIM*`)`, where *DIM* is the dimension. Dimension 1 corresponds to rows and dimension 2 is 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. `size` can also be used with as many outputs as dimensions (the jth output contains the length of the jth dimension).

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

[1] When used on higher dimensional arrays, `length(x)` is the same as `max(size(x))` and so it returns the maximum dimension size across the entire array.

```
     4     5     6
>> size(x,1)
ans =
     2
>> size(x,2)
ans =
     3
>> size(x',1)
ans =
     3
>> s=size(x)
s =
     2     3
>> [m,n] = size(x)
m =
     2
n =
     3
```

sum

sum computes the sum of the columns of a matrix,

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

z=sum(x) returns a K by 1 vector containing the sum of each column, so that
z(i) = sum(x(:,i)) = x(1,i) + x(2,i) + ... + x(T,i).
Warning: If x is a vector, sum will add all elements of x irrespective of whether it is a row or column vector.
sum can be used with an optional second argument which specifies the dimension to sum over using the 2-input form, z=sum(x,*DIM*). If x and a T by K matrix, then sum(x,1) (identical to sum(x)) returns a 1 by K vector of column sums, while sum(x,2) returns a T by 1 vector of row sums.

```
>> 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
>> sum(x,2)
ans =
     6
    15
```

min, max

min(x) computes the minimum of a matrix,

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

column-by-column (max is identical to min, only computing the maximum). If x is a vector, min(x) is scalar. If x is a matrix, min(x) is a K by 1 vector containing 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
```

min(x,[],*DIM*) can be used with an optional 3rd input to indicate the dimension to compute the minimum across (e.g. min(x,[],1) for columns, min(x,[],2) for rows). The *DIM* argument occurs in the 3rd position since there is a rarely used 2-input form of min which computes the minimum of 2 matrices (with the same size) or of a matrix and a scalar. Both min and max can be used with a 2nd output to return the index or indices of the smallest and largest elements, respectively.

```
>> x=[1 5 3; 4 2 6];
>> [minX, ind] = min(x)
minX =
     1     2     3

ind =
     1     2     1
```

prod

prod computes the product of the elements of a column of a matrix,

$$z = \prod_{t=1}^{T} x_t.$$

z=prod(x) returns a K by 1 vector containing the product of each column, so that
z(i) = prod(x(:,i)) = x(1,i) * x(2,i) * ...* x(T,i).
Warning: If x is a vector, prod will multiply all elements of x irrespective of whether it is a row or column vector. prod can be used with an optional second argument which specifies the dimension to multiply over using the 2-input form, z=prod(x,*DIM*). If x and a T by K matrix, then prod(x,1) (identical to prod(x)) returns a 1 by K vector of column products, while prod(x,2) returns a T by 1 vector of row products.

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

```
x =
     1     2     3
     4     5     6
>> prod(x)
ans =
     4    10    18
>> prod(x')
ans =
     6   120
>> prod(x,2)
ans =
     6
   120
```

`cumsum`, `cumprod`, `cummax`, `cummin`

`cumsum` computes the cumulative sum of a vector of a matrix (column-by-column),

$$x_{ij} = \sum_{k=1}^{i} x_{kj}.$$

`cumsum(x,`*DIM*`)` changes the dimension used to compute the cumulative sum. `cumprod` is identical to `cumsum`, only computing the cumulative product,

$$x_{ij} = \prod_{k=1}^{i} x_{kj}.$$

`cummax(x)` and `cummin(x)` compute the cumulative max and minimum of an array, respectively.

`sort`

`sort` orders the values in a vector or the rows of a matrix from smallest to largest. If `x` is a vector, `sort(x)` is vector where `x(1)=min(x)` and `x(i)`≤`x(i+1)`. If `x` is a matrix, `sort(x)` is a matrix of the same size where the sort is performed column-by-column.

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

```
     5     6
```

sort(x, *DIM*) can be used to change the dimension of the sort. sort can be used with a second input to output a lit of the indices used to sort. This is especially useful when one matrix needs to be sorted according to the data in another matrix.

```
>> x=[9 1 8 2 7 3 6 4 5];
>> [sortedX,ind] = sort(x)
sortedX =
     1     2     3     4     5     6     7     8     9

ind =
     2     4     6     8     9     7     5     3     1
>> y = x;
>> y(ind)
y =
     1     2     3     4     5     6     7     8     9
```

The related command sortrows can be used to perform a lexicographic sort or a matrix, which first sorts the first column, then the second column for those rows with the same value in the first column, and so on.

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

Like sort, sortrows can be used with a 2nd output to produce a vector containing the indices used in the sort.

exp

exp computes the exponential of a vector or matrix (element-by-element),

$$e^x.$$

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

`log` computes the *natural* logarithm of a vector or matrix (element-by-element),

$$\ln x.$$

`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

`sqrt` computes the square root of a vector or matrix (element-by-element),

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

`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 dot-operator command `z=x.^(1/2)`.

mean

`mean(x)` computes the mean of a vector or matrix,

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

If `x` is a T by K matrix, `z=mean(x)` is a K by 1 vector containing the means of each column, so `z(i) = sum(x(:,i)) / size(x,1)`. `mean(x, DIM)` can be used to alter the dimension used.
Warning: When `x` is a vector, `mean` behaves like `sum` and so will compute the mean of the 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

`var` computes the sample variance of a vector or matrix,

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

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. `var(x,[],` *DIM*`)` can be used to alter the dimension used. **Note**: This command uses $T-1$ in the denominator by default. This behavior can be altered using an optional second argument.

```
>> 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

`cov` computes the sample covariance of a vector or matrix

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

If x is a vector, `cov(x)` is scalar (and is identical to `var(x)`). If x is a matrix, `cov(x)` is a K by K matrix with sample variances in the diagonal elements and sample covariances in the off-diagonal elements. **Note**: Like `var`, `cov` 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

std compute the sample standard deviation of a vector or matrix (column-by-column),

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

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. std(x,[], *DIM*) can be used to alter the dimension used. **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

skewness computes the sample skewness of a vector or matrix (column-by-column),

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

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. skewness(x,[],*DIM*) changes the dimension 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

kurtosis computes the sample kurtosis of a vector or matrix,

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

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. kurtosis(x,[],*DIM*) changes the dimension used.

```
>> 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
```

4.1 Moving window functions

The most common statistics functions are available in moving window versions which will compute the function using all blocks of data within an array. There are moving window versions of mean, median,var, std, max, min and sum. These all follow the patterm mov*func* where *func* is one of the previously names functions, for example, movmean. The basic use of these functions requires the specification of the window length, and the function will be computed for all contiguous blocks with this length.

```
>> x=[1 7 2 10 0 -1];
>> movmean(x, 3)
ans =
   4.0000 3.3333 6.3333 4.0000 3.0000 -0.5000
>> movmax(x, 4)
ans =
    7     7    10    10    10    10
```

4.2 Exercises

1. Load the MATLAB data file created in the Chapter 14 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. Find the max and min of the variable SP500 (see the Chapter 14 exercises). 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: Use length or size.

Chapter 5

Special Vectors and Matrices

MATLAB contains a number of commands to produce structured vectors and matrices.

: operator

The : operator has multiple uses. The first allows elements in a matrix or vector to be accessed ((e.g. `x(1,:)` as previously described). The second allows a matrix to be collapsed into a column vector (e.g. `x(:)`, which is identical to `reshape(x,prod(size(x)),1)`). The final constructs row vectors 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

$$[\mathit{first}, \quad \mathit{first}+1, \ldots \mathit{first}+N]$$

where N is the largest integer such that *first*+N $\leq$ *last*. When *first* and *last* are both integers and *first*$\geq$*last*, then N =*last*–*first*. These examples demonstrate the use of the : operator.

```
>> 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

$$[\mathit{first}, \quad \mathit{first}+\mathit{increment}, \quad \mathit{first}+2(\mathit{increment}), \ldots \mathit{first}+N(\mathit{increment})]$$

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

Note that *first*:*last* is the same as *first*:1:*last*.

The *increment* does not have to be positive. If a negative increment is used, the general form is unchanged but the stopping condition changes so that 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
```

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 δ=(*upper*-*lower*)/(N-1), the resulting sequence will have the form

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

where *lower*+(N − 1)δ is by construction equal to *upper*. This, the command `linspace(lower,upper,N)` will produces the same output as `lower:(upper-lower)/(N-1):upper`.

Recall that : is a low precedence operator, and so operations involving : should always be enclosed in parenthesis when used with other mathematical expressions. Failure to do so can result in undesirable or unexpected behavior. For example, consider

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

The second line is interpreted (correctly, based on its rules) as

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

which first generates a sequence, and then uses the colon operator with the sequence as the first argument – which is not the correct method to produce a sequence using :.

logspace

logspace produces points uniformly distributed in log10 space.

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

Note that `logspace(lower, upper, N)` is the same as `10.^linspace(lower, upper, N)`.

zeros

zeros generates a matrix of 0s and is generally called with two arguments, the number of rows and the number of columns.

```
>> M = 2; N = 5;
>> x = zeros(M,N)
```

will generate a matrix of 0s with N rows and M columns. `zeros(M,N)` and `zeros([M N])` are equivalent – the latter is more useful if the number of dimensions depends on data or some other input. zeros can also be used with more than 2 inputs to create 3- or higher-dimensional arrays.

ones

ones produces a matrix of 1s in the same way zeros produces a matrix of 0s.

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

eye

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

```
In = eye(N)
```

nan

nan produces a matrix populated with NaNs (see Ch. 7) in the same way zeros produces a matrix of 0s. nan is useful for initializing a matrix for storing missing values where the missing values are left as NaNs.

5.1 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`?
5. Using both the : operator and `linspace`, create the sequence 0, 0.01, 0.02, . . . , .99, 1.
6. 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`.

Chapter 6

Matrix Functions

Some functions operate exclusively on matrix inputs. These functions can be broadly categorized as either matrix manipulation functions – functions which alter that structure of an existing matrix – or mathematical functions which are only defined for matrices such as the computation of eigenvalues and eigenvectors.

6.1 Matrix Manipulation

repmat

`repmat` replicates a matrix according to a specified size vector. 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 X was a matrix

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

and the block matrix

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

was needed. This could be accomplished by manually constructing `y` as

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

Alternatively, `y` can also be constructed with `repmat` by

```
>> y = repmat(x,2,3);
```

`repmat` has two clear advantages over manual allocation. First, `repmat` can be executed using on some parameters determined at run-time, such as the number of explanatory variables in a model. Second, `repmat` can be used for arbitrary dimensions. Manual matrix construction is tedious and error prone with as few as 4 rows or columns.

reshape

reshape transforms a matrix with one set of dimensions to one with a different set as long and the number of elements does not change (and *cannot* change). reshape can transform an M by N matrix x into an K by L matrix y as long as $MN = KL$. 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) % same as y'
z =
     1     3     2     4
>> w = reshape(z,2,2)
w =
     1     2
     3     4
```

The crucial implementation detail of reshape is that matrices are stored using column-major notation. Elements in matrices are indexed first down the rows of a column and then across columns. reshape will place elements of the old matrix into the same position in the new matrix and so after calling reshape, $x(1) = y(1)$, $x(2) = y(2)$, and so on.

diag

diag can be used to convert a vector to a diagonal matrix or to extract the leading diagonal from a matrix. The behavior depends on the format of the input. If the input is a vector, diag will return a matrix containing the elements of the vector along the diagonal. If the input is a matrix, diag will return a column vector containing the elements of the leading diagonal (i.e. positions (1,1), (2,2) ... up to the smaller of the number of rows or columns). 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
```

x	y	Compatible	Common Size
$(10,1)$	$(10,200)$	Yes	$(10,200)$
$(1,1,200)$	$(10,10)$	Yes	$(10,10,200)$
$(10,2)$	$(10,200)$	No	–
$(10,1)$	$(1,10)$	Yes	$(10,10)$
$(1,10)$	$(1,10)$	Yes	$(1,10)$
$(10,1,100)$	$(1,10,100)$	Yes	$(10,10,100)$

Table 6.1: Example of compatible dimensions for singleton expansion.

6.2 Broadcastable Operations: bsxfun

`bsxfun` is a convenience and performance function which allows basic mathematical operations on vectors which are not compatible under the rules of chapter 3. For example, suppose `x` is a T by K matrix of data, and the studentized – mean 0 and variance 1 – data are needed. The first step in studentizing a matrix, subtracting the mean from each column, can be accomplished using a variety of functions, for example using `repmat` or `ones`:

```
>> x = randn(100,10);
>> meanX = mean(x);
>> demeanedX_1 = x - repmat(meanX,100,1);
>> demeanedX_2 = x - ones(100,1) * meanX
```

`bsxfun` simplifies this code by automatically performing *singleton expansion*. Singleton expansion expands all arrays dimensions which are 1 to be compatible with a dimension sizes which are not 1. Formally, singleton expansion is only possible for two arrays `x` and `y` when either statement is true for all dimensions:

- $\texttt{dim(x,}i\texttt{)} = \texttt{dim(y,}i\texttt{)}$
- If $\texttt{dim(x,}i\texttt{)} \neq \texttt{dim(y,}i\texttt{)}$, then $\texttt{dim(x,}i\texttt{)} = 1$ or $\texttt{dim(y,}i\texttt{)} = 1$

Note that if the number of dimensions of the two arrays differ, that all “missing” dimensions have size 1. Table 6.1 contains some examples with different array dimensions.

When two arrays are compatible, `bsxfun` requires 3 inputs. The first is function to use, and can either be a string or a function handle. Common functions are `'plus'`, `'minus'`, `times'`, `'rdivide'` and `'ldivide'` (or `@plus`, `@minus`, etc.). The previous example can be rewritten using `bsxfun` in a single line:

```
>> demeanedX_3 = bsxfun(@minus,x,mean(x));
```

While this example produces the same output as the two previous examples, the `bsxfun` version is actually higher performing since `bsxfun` avoids allocating the full T by K matrix of the means prior to computing the difference. When the input is small, these two will perform similarly. However, when the input is large, `bsxfun` is substantially higher performing.

6.3 Linear Algebra Functions

chol

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

$$C'C = \Sigma$$

where Σ is a positive definite matrix.

det

`det` computes the determinant of a square matrix,

$$|x|$$

eig

`eig` computes the eigenvalues and eigenvector of a square matrix. When used with one output (`val=eig(x)`), the vector of eigenvalues is returned. When used with two (`[vec,val]=eig(x)`), matrices containing the eigenvectors and eigenvalues (diagonal) are returned so that `vec*val*vec'` is the same as `x`.

inv

`inv` computes the inverse of a matrix. `inv(x)` can alternatively be computed using `x^(-1)` or `x\eye(length(x))` – the latter form is preferred for both performance and precision.

kron

`kron` computes the Kronecker product of two matrices. `z = kron(x,y)` implements the mathematical expression

$$z = x \otimes y.$$

trace

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

Chapter 7

Inf, NaN and Numeric Limits

Three special expressions are reserved to indicate certain non-numerical "values". `Inf` represents infinity and `Inf` is distinct from `-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 whenever a function produces a result that cannot be clearly defined as a number or infinity. For instance, `inf/inf` produces a `NaN`.

All numeric software has limited precision and MATLAB is no different. The easiest limits to understand are the upper and lower limits – 1.7977×10^{308} and -1.7977×10^{308} (`realmax`). Numbers larger (in absolute value) than these are `Inf`. The smallest non-zero number that can be expressed is 2.2251×10^{-308} (`realmin`). Numbers between -2.2251×10^{-308} and 2.2251×10^{-308} are numerically 0.

The most difficult concept to understand about numerical accuracy is the limited relative precision. The relative precision of MATLAB is 2.2204×10^{-16}. This value is returned from the command `eps` and may vary based on the type of CPU and/or the operating system used. Numbers which differ by a *relative* range of 2.2204×10^{-16} are numerically 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
>> x-1
ans =
   4.4408e-016
```

Next, consider how the order of execution matters to the final result:

```
>> x=1-1+eps/2
x =
   1.1102e-16
>> x=1-(1+eps/2)
```

```
x =
    0
```

The difference in these two expressions arises since, in the first, 1 is subtracted from 1, and then eps/2 is added (which is distinct from 0), while in the second 1+eps/2 is numerically identical to 1, and so after the expression in the parentheses is evaluated, the intermediate result is 1, which is the subtracted from 1 producing 0.

To better understand what is meant by *relative* range, consider the following output:

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

In the initial example, eps/2 < eps so it has no effect (relative to 1) while 2*eps > eps so it does. However in the second example, 2*eps/10 < eps, and so it has no effect when added. In other words, 2*eps is sufficiently "big" relative to 1 to create a difference, while it is not relative to 10. This is a very tricky concept to understand, but failure to understand numeric limits can results in errors or surprising results from that is otherwise.

7.1 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 8

Logical Operators

Logical operators, when combined with flow control (such as `if` ... `else` ... `end` blocks, chapter 9), allow for complex choices to be compactly expressed. They are additionally useful for selecting subsets of vectors of matrices which satisfy some range restrictions.

8.1 $>$, $>=$, $<$, $<=$, $==$, $\sim=$

The core logical operators are

Mathematical Expression	MATLAB Expression	Definition
$>$	`>`	Greater than
$\geq$	`>=`	Greater than or equal to
$<$	`<`	Less than
$\leq$	`<=`	Less then or equal to
$=$	`==`	Equal to
$\neq$	`~=`	Not equal to

Logical operators can be used on scalars, vector or matrices. All comparisons are done element-by-element and return either logical true (which has numeric value 1) or false (0).[1] For instance, suppose `x` and `y` are matrices of the same size. `z=x < y` will be a matrix of the same size as `x` and `y` composed of 0s and 1s. Alternatively, if one is scalar, say `y`, then the elements of `z` are `z(i,j)= x(i,j) < y`. The following table examines the behavior when `x` and/or `y` are scalars or matrices. Suppose `z = x < y`:

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

[1] Note that true and false are known as Boolean variables and are not standard numerical values. Boolean variables are stored using 1 byte of computer memory, while typical numerical values require 8 bytes to store. Ch. 21 provides a more detailed description of the data types available.

8.2 & (AND), | (OR) and ~ (NOT)

Logical expressions can be combined using three logical devices,

Logical Expression	Standard Operator	Short-circuit Operator
AND	&	&&
OR	|	||
NOT	~	N/A

Aside from the different level of precedence (NOT (~) has higher precedence than AND (&) and OR ()), these operators follow the same rules as other logical operators, and so when used on matrices, all dimensions must be identical. When one of the inputs is a scalar and the other is a matrix, the operator is applied to the scalar and each element of the matrix.

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

		y: Scalar	y: Matrix
x	Scalar	Any $z = x\&y$[2]	Any $z_{ij} = x\&y_{ij}$
x	Matrix	Any $z_{ij} = x_{ij}\&y$	Both Dimensions Match $z_{ij} = x_{ij}\&y_{ij}$

AND and OR (but not NOT), can be used in both standard and short-circuit forms. Short-circuit operators terminate as soon as the statement can be correctly evaluated and so offer higher performance, although they can only be used with scalar logical expressions. In general, short-circuit operators should be used when applicable.

8.3 Logical Indexing

Logical operators can be used to access a subset of the elements of a vector or matrix. Standard indexing operates by using the numerical location (e.g. 1,2,...) of elements in a matrix. In contrast, logical indexing essentially is a series of *yes* or *no* indicating whether a value should be selected. Logical indexing uses Boolean values – true or false (0 or 1, but must be logical 0 or 1, not numeric 0 or 1) – as opposed to the numeric values when using standard indexing. In essence, Logical indices behave like a series of light switches indicating which elements to select: 1 for on (selected) and 0 for off (not selected).

```
>> x=[-2 0 1 2];
>> y = x<=0
y =
     1     1     0     0
>> x(y)
ans =
    -2     0
>> x(x~=0)
ans =
```

[2] When both inputs are scalar, short-circuit operators (&& and) should be used.

```
    -2     1     2
>> x(x>0) = -1
x =
    -2     0    -1    -1
```

Logical indexing is very powerful when combined with other logical functions. For example, suppose `nan` is used to represent missing data in an array. `any(isnan(x),2)` will return a logical vector indicating whether any of the values in a row are `nan`, and so the negative of this statement indicates all values are not `nan`. This expression can be used to remove rows with nans so that mathematical operations will not be `nan`.

```
>> x= ones(3,3);
>> x(2,2) = nan;
>> sum(x)
ans =
     3   NaN     3
>> sum(x(~any(isnan(x),2),:))
ans =
     2     2     2
```

8.4 Logical Functions

8.4.1 logical

The command `logical` is used to convert non-logical elements to logical. Logical values and regular numerical values are not exactly the same. Logical elements only take up 1 byte of memory (The smallest unit of memory MATLAB can address) while regular numbers require 8 bytes. `logical` is useful to convert the standard numerical data type in MATLAB to logical values.

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 instead is a logical value, this behavior changes. The following code shows how numeric indices differ from logical ones,

```
>> x = [1 2 3 4];
>> y = [1 1];
>> x(y) % Element number 1 twice
ans =
   1 1
>> y = logical([1 1]); % True for elements 1 & 2
>> x(y)
ans =
   1 2
>> y = logical([1 0 1 0]); % True for elements 1 & 3
>> x(y)
ans =
   1 3
```

Note that `logical` turns any non-zero value into logical true (1), although a warning is generated 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
```

8.4.2 all and any

The commands `all` and `any` are useful for aggregating logical values. `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. When used with a matrix input, `any` operates column-by-column, returning logical true if any element of that column is true.

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

8.4.3 find

`find` is a useful function for working with multiple data series. `find` is not logical itself, although 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 (i, j) that 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
y =
    1  1
    1  0
>> find(y)
ans =
   1
   2
   3
>> [i,j] = find(y)
i =
    1
    2
    1
j =
    1
    1
    2
```

8.4.4 `is*`

A number of special purpose logical tests are provided 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 functions all begin with `is`.

Function	Description	Mode of Operation
`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 input is 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.

8.5 Exercises

1. Using the data file created in Chapter 14, count the number of negative returns in both the S&P 500 and ExxonMobil.
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 = ?` with out using `any` and `myall = ?` without using `all`).

Chapter 9

Control Flow

9.1 Choice

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`.

9.1.1 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
elseif  logical_2
    Code to run if logical_2
elseif  logical_3
    Code to run if logical_3
...
...
else
    Code to run if all previous logicals are false
end
```

However, simpler forms are more common,

```
if logical
    Code to run if logical true
end
```

or

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

Note: Remember 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
x =
    4
```

and

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

```
>> x
x =
    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.

9.1.2 switch...case...otherwise

`switch`...`case`...`otherwise` blocks allow for more 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 and if not present no code is run if none of the `case`s are 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 a `switch` ... `case` ... `otherwise` block is

```
switch variable
    case value_1
        Code to run if variable=value_1
    case value_2
        Code to run if variable=value_2
    case value_3
        Code to run if variable=value_3
```

```
...
...
    otherwise
        Code to run if variable not matched
end
```

There is an equivalence between switch ... case ... otherwise and if ... elseif ... else blocks, although if the logical expressions in the if ... elseif ... else block contain inequalities, variables must be created prior to using a switch ... case ... otherwise block. switch ... case ... otherwise blocks also differ from standard C behavior since only one case can be matched per block. The switch ... case ... otherwise block is exited after the first match 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
x =
   10
```

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

```
x=5;
switch x
    case {4,5}
        x=x+1;
    case {1,2}
        x=2*x;
    otherwise
        x=0;
end
>> x
x =
   6

x = 9;
switch x
    case {4}
        x=x+1;
    case {1,2,5}
        x=2*x;
    otherwise
```

```
        x=0;
end
>> x
x =
    0
```

9.2 Loops

Loops make many problems, particularly when combined with flow control blocks, simple and in many cases, feasible. Two types of loop blocks are available: `for` . . . `end` and `while`. . . `end`. `for` blocks iterate over a predetermined set of values 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 than a `for` loop and a `break` statement.

9.2.1 for loops

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

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

iterator is the variable that the loop will iterate over. For example, `i` is a common name for an 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 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 `for` 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 construct is to loop over the `length` of a vector, which allows each element to accessed individually.

```
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
```

or 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 should not be modified inside the for loop. Changing the iterator can produce undesirable results. For instance,

```
for i=1:10
    i
    i=2*i;
    i
end
```

Produces the output

```
i =
     1
i =
     2
i =
     2
i =
     4
i =
     3
i =
     6
...
i =
    10
i =
    20
```

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

9.2.2 while loops

`while` loops are useful when the number of iterations needed depends on the outcome of the loop contents. `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: first, the *logical* expression should evaluate to true when the loop begins (or the loop will be ignored) and second the inputs to the *logical* expression must be updated inside the loop. If they are not, the loop will continue indefinitely (hit CTRL+C to break an interminable loop). The simplest while loops are drop-in replacements of `for` loops, and

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

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 criterion is met. Any finite `for` loop cannot be guaranteed to meet the criteria.

9.2.3 break

`break` can be used to terminate a `for` loop and, as a result, `for` loops can be constructed to behave similarly to `while` loops.

```
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 than *iterator* contains. `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 logical condition is not true.

9.2.4 continue

`continue`, when used inside a loop, has the effect of advancing the loop to the next iteration while skipping any remaining code in the body of the loop. While `continue` can always be avoided using `if`...`else` blocks, its use typically results in tidier code. The effect of `continue` 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 whenever `i` is even (and `(i/2)== floor(i/2)` evaluates to logical true).

9.3 Exception Handling

Exception handling is an advanced tool which allows programs to be tolerant of errors. It is not necessary for most numerical applications since data values which would produce the error, such as dividing by 0, can be checked, and if encountered, an alternative code path can be executed. Exception handling is more useful when performing input/output (especially if over a network)

9.3.1 try...catch

`try`...`catch` blocks can be used to execute code which may not always complete. They should not usually be used in numeric code since it is better to anticipate and explicitly handled issues when they occur to ensure correct results. `try` statement allow subsequent statements to be run, and, more importantly, for

continuation even if they code contains an error. `catch` blocks execute at the point where the error occurs, and so if the code in the `try` block does not produce an error, the `catch` block is skipped. `catch` blocks can be used with a special syntax to capture the error, which may be useful for debugging or cleaning up any resources which were used in the `try` block. Note that when an error occurs, the code in the try block *before* the error is executed and any code *after* the error is skipped.

One scenario for using a `try` . . . `catch` block is when reading or writing data to a network drive if there is some chance that the network drive may be temporarily down. The following code shows one method to accomplish this. The `catch` block uses `matlabError` to capture the error so that information can be displayed. It also checks to see if the file is open, in which case `fid` would be positive, and closes it if needed.

```
notRead = true;
while notRead
   try
      fid = fopen('data.txt','rt');
      data = fgetl(fid);
      fclose(fid);
      notRead = false;
   catch matlabError
      if fid>0
          fclose(fid);
      end
      disp(matlabError.identifier)
      disp(matlabError.message)
      % Pause for 30 seconds before retrying
      pause(30)
   end
end
```

9.4 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.

3. Simulate 1000 observations from an ARMA(2,2) where ϵ_t are independent standard 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**: A T by 1 vector containing standard normal random variables can be simulated using `e = randn(T,1)`. 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.

4. Simulate a GARCH(1,1) process where ϵ_t are independent standard normal innovations. A GARCH(1,1) process is given by

$$y_t = \sigma_t \epsilon_t$$

$$\sigma_t^2 = \omega + \alpha y_{t-1}^2 + \beta \sigma_{t-1}^2$$

Use the values $\omega = 0.05$, $\alpha = 0.05$ and $\beta = 0.9$, and set $h_0 = \omega / (1 - \alpha - \beta)$.

5. Simulate a GJR-GARCH(1,1,1) process where ϵ_t are independent standard normal innovations. A GJR-GARCH(1,1) process is given by

$$y_t = \sigma_t \epsilon_t$$

$$\sigma_t^2 = \omega + \alpha y_{t-1}^2 + \gamma y_{t-1}^2 I_{[y_{t-1}<0]} + \beta \sigma_{t-1}^2$$

Use the values $\omega = 0.05$, $\alpha = 0.02$ $\gamma = 0.07$ and $\beta = 0.9$ and set $h_0 = \omega / \left(1 - \alpha - \frac{1}{2}\gamma - \beta\right)$. Note that 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.

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

$$y_t = \phi_1 y_{t-1} + \theta_1 \sigma_{t-1} \epsilon_{t-1} + \lambda \sigma_t^2 + \sigma_t \epsilon_t$$

$$\sigma_t^2 = \omega + \alpha \sigma_{t-1}^2 \epsilon_{t-1}^2 + \gamma \sigma_{t-1}^2 \epsilon_{t-1}^2 I_{[\epsilon_{t-1}<0]} + \beta \sigma_{t-1}^2$$

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

7. 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`.

8. 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 10

Graphics

Extensive plotting facilities capable of producing a virtually limitless range of graphical data representations are available. This chapter will emphasize the basics of the most useful graphing tools.

10.1 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.

- `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 the plot is 3-D) 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 3-D. 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 3-D. `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 support functions, but there are many additional functions available to customize figures (see section 10.5).

10.2 2D Plotting

10.2.1 `plot`

`plot` is the most basic plotting command. Like most commands, it can be used many ways. the standard usage for a single series is

```
plot(x1,y1,format1)
```

where `x1` and `y1` are vector of the same size and *format1* is a format string of the form *color shape linespec*. *color* can be any of

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

shape can be any of

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

and *linespec* can be any of

-	solid	-.	dashdot
:	dotted	--	dashed
(none)	no line		

The three arguments are combined to produce 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 (note that the string is r-plus-space). 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, an automatic color scheme will be used with marker-less solid lines. Suppose the following x and y data were created,

```
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 10.1. A line's color information is lost when documents printed are in black and white, and so it is important to use physical characteristics to distinguish multiple series – either different line types or different markers, or both.

All plots should be clearly labeled. The following code 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 10.1.

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

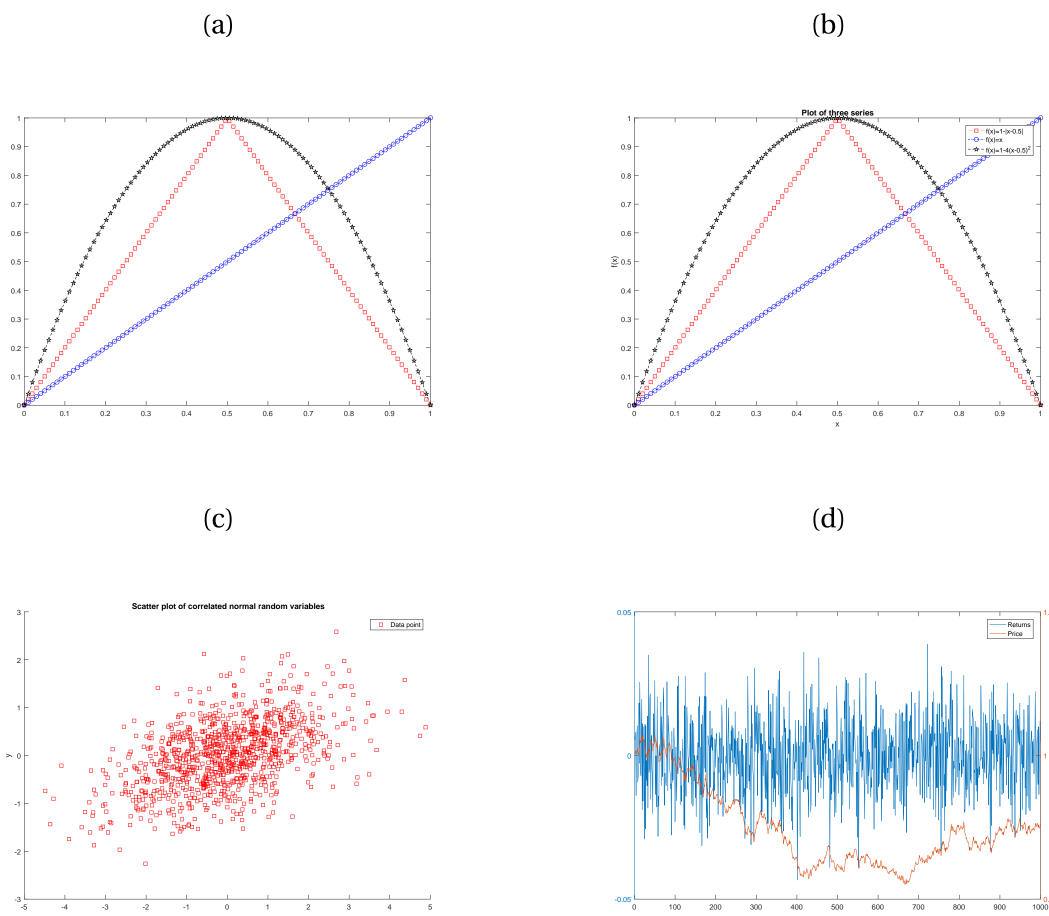

Figure 10.1: The lines in panel (a) were plotted with the command `plot(x,y1,'rs:',x,y2,'bo-.',x,y3,'kp- -')`. Panel (b) shows that same plot only with clearly labeled axes, a title and legend. Panel (c) contains a scatter plot of a bivariate normal random deviations with unit variance and correlation of 0.5 produced by calling `scatter(x(:,1),x(:,2),'rs')`. Panel (d) contains a plot with two different axes produced using `plotyy(x,y1,x,y2)`.

One final method for calling `plot` is worth mentioning. `plot(y)` will plot the data in vector `y` against a simple series which labels each observation 1, 2, . . ., `length(y)`. `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 and `plot(y)` is equivalent to `plot(1:length(y(:,1)), y(:,1), 1:length(y(:,2)), y(:,2), ...)`.

10.2.2 `plotyy`

`plotyy` is a special version of `plot` which allows two series to be plotted on the same graph using different axes - a left and a right one. The basic use is `plotyy(x1,y1,x2,y2)`. The following code plots a set of simulated returns, `y1`, and the corresponding log-price, `y2`, which is just the exponential of the cumulative sum of the returns. The output of this code can be seen in panel (d) of figure 10.1.

```
x = 1:1000;
y1 = .08/365 + randn(1000,1)*.2/sqrt(250);
```

```
y2 = exp(cumsum(y1));
plotyy(x,y1,x,y2)
legend('Returns','Price')
```

10.2.3 scatter

scatter, like most graphing functions, 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 marker shape can be provided using a format string as plot. Other options, such as marker size, must be set using handle graphics or interactive plot editing. A simple example of handle graphics is included at the end of this chapter. Consult scatter's help file for further information. The following code produces a scatter plot 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 panel (c) of figure 10.1.

```
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')
```

10.2.4 bar

bar produces vertical bar chart, and can be used as bar(y) or bar(x,y) – the first form uses 1:length(y) as the values for x, which are the bar locations. The following code produces a bar chart with only selected columns present. The output of this code can be seen in panel (a) of figure 10.2.

```
x = [1 2 4 5 9];
y = 20-(5-x).^2;
bar(x,y)
title('Bar Chart')
```

Other bar charts can be produced using an optional style argument (bar(x,y,'style')), where style is one of:

- 'grouped' - Produces a bar chart where values in each column of y are grouped together, but appear in different colors.
- 'stacked' - Produces a bar chart by stacking the values in the each column of y. This is only useful if y is n by k where $n > 1$.
- 'hist' - produces a bar chart with no space between bars.

Examples of the three styles appear in panels (b) – (d) of figure 10.2. These were generated (in order) using

```
bar(1:3,[1 2 3;2 3 4;3 4 5],'grouped')
bar(1:3,[1 2 3;2 3 4;3 4 5],'stacked')
bar(1:3,[1 2 3;2 3 4;3 4 5],'hist')
```

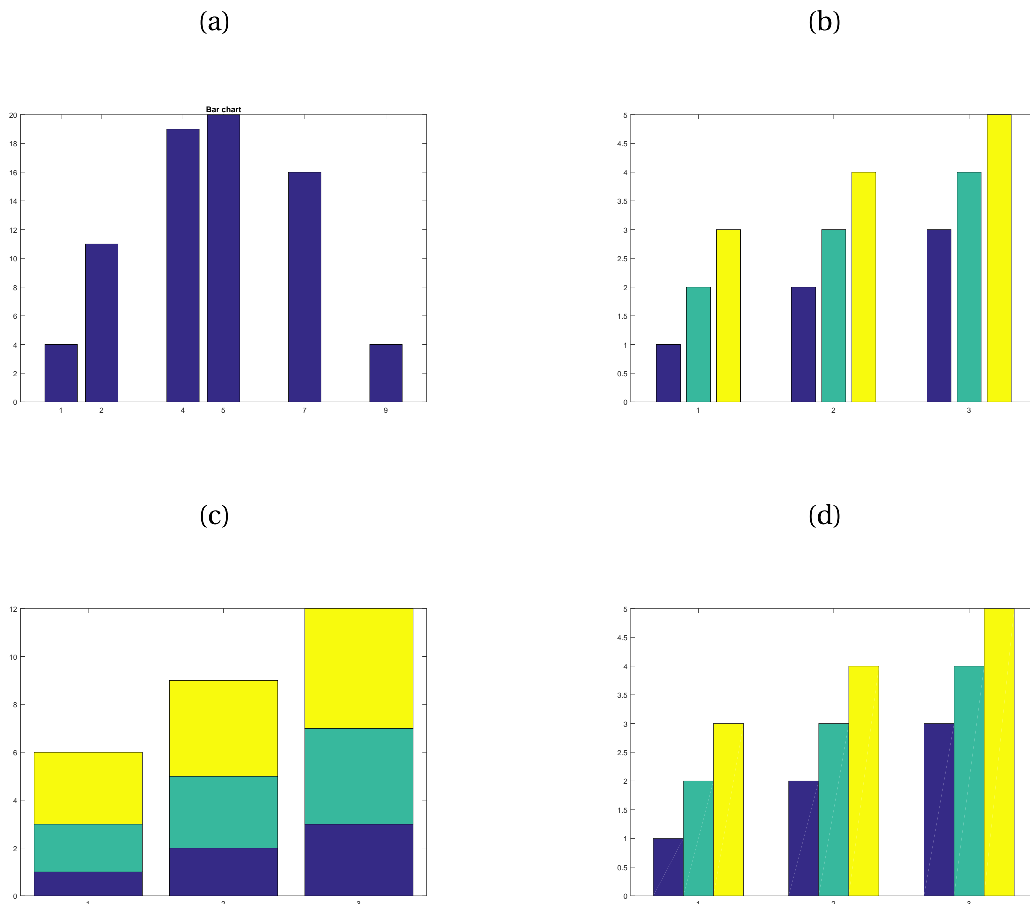

Figure 10.2: This plot contains four bar charts produced using variances of `bar(x,y)`.

10.2.5 hist

`hist` constructs a histogram – a rough empirical PDF – of a vector of data. The following code simulates 10,000 χ^2_4 random variables and produces a histogram of the simulated values using 50 bins in the histogram (10 bins are used by default).

```
x = chi2rnd(4,10000,1);
hist(x, 50)
```

The results of running this code is presented in panel (a) of figure 10.3.

10.2.6 stairs

`stairs` produces a plot which is appropriate for discrete data - such as high-frequency price data. The primary difference between `stairs` and `plot` is the mechanism used to connect the data points plotted. `stairs` uses a step method to connect the points while `plot` uses simple linear interpolation. Panel (b) of figure 10.3 shows the result of running the following code.

```
price = cumsum(randn(20,1));
```

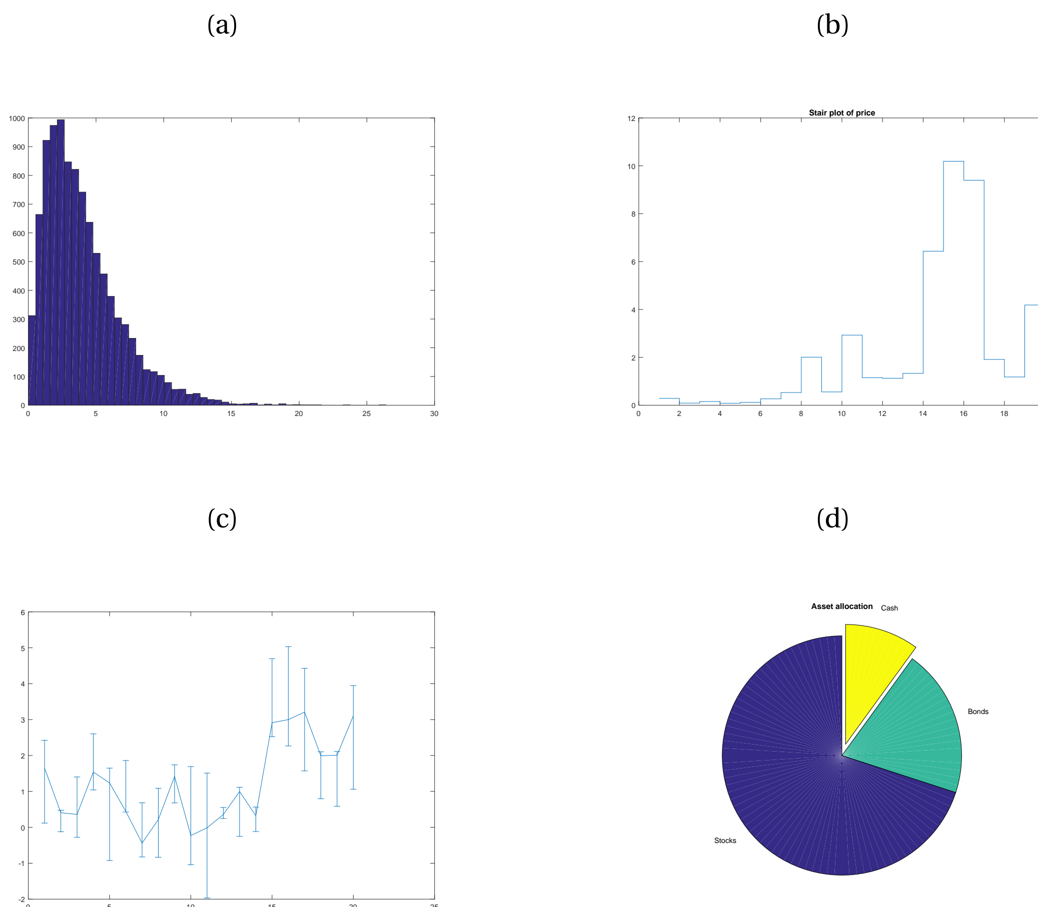

Figure 10.3: Panel (a) demonstrates the use of `hist`. Panel (b) shows the use of `stairs` to plot discrete data. Panel (c) demonstrates the use of `errorbar` and panel (d) shows the use of `pie`.

```
stairs(price)
title('Stair plot of price')
```

10.2.7 errorbar

`errorbar` adds error bars to a basic `plot`. The error bars can be provided using either a scalar, in which case the error bars are plotted using 2 times the scalar – it is similar to a standard deviation when the data is normal – or using vectors `L` and `U` which specify the lower and upper bounds (in deviation from the data). The following code produces an `errorbar` plot using (random) lower and upper bounds for the error bars.

```
x = 1:20;
l = -abs(randn(1,20));
u = abs(randn(1,20));
y = cumsum(randn(1,20));
errorbar(x,y,l,u)
```

The resulting plot can be seen in panel (c) of figure 10.3.

10.2.8 pie

pie can be used to produce a pie chart. The basic structure is pie(y,explode,label) where y is the data to use in the pie chart, explode is a vector with the same size as y which describes how far from a center a slice should appear (default is 0), and label is a cell array of strings which can be used to provide labels for each slice (See Chapter 13 for more on cell arrays). The following code produces the pie chart in panel (d) of figure 10.3.

```
pie([.7 .2 .1],[.1 0 0],{'Stocks','Bonds','Cash'})
title('Asset allocation')
```

10.3 3D Plotting

10.3.1 plot3

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

```
plot3(x1,y1,z1,format1)
```

The following code block demonstrates the use of plot3.

```
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 panel (a) of figure 10.4.

10.3.2 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

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

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

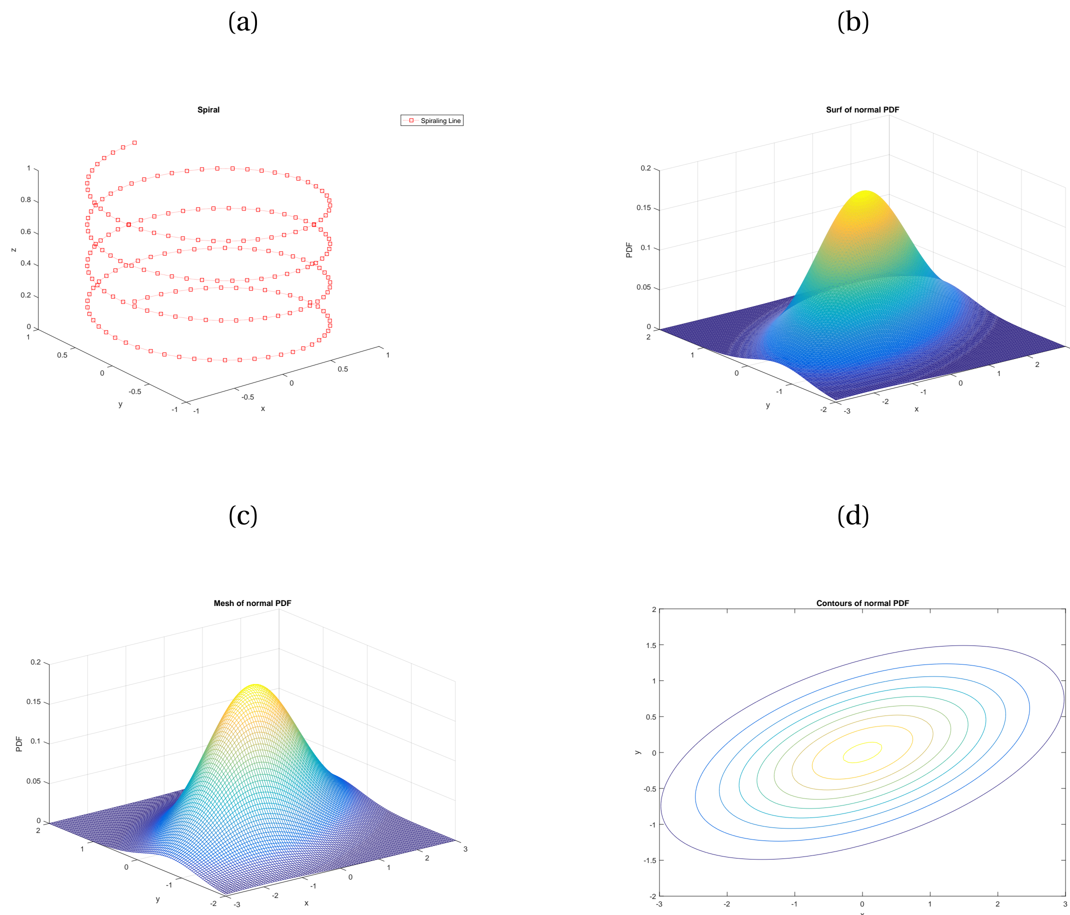

Figure 10.4: 3-D lines can be plotted using the `plot3` command. This line was plotted by calling `plot3(x,y,z,'rs:');`. `surf` plots a 3-D surface from vectors of x and y data and a matrix of z data. This `surf` contains the PDF bivariate of a bivariate normal, and was created using `surf(x,y,pdf)` where `x`, `y` and `pdf` are defined in the text. `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 of a bivariate normal, and was created using `mesh(x,y,pdf)` where `x`, `y` and `pdf` are defined in the text. 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.

```
N = 100;
x = linspace(-3,3,N);
y = linspace(-2,2,N);
Sigma = [2 .5; .5 .5];

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 (panel (b) of figure 10.4) 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 from a discrete 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 3-D 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.

10.3.3 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 report. Using the same bivariate normal setup, the following code produces the mesh plot evidenced in panel (c) of figure 10.4.

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

10.3.4 contour

contour is similar to surf and mesh in that it takes three arguments, x, y and z. contour differs in that it produces a 2D plot. contour plots, while not as eye-catching as surf or 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) where N determines the number of contours drawn. If omitted, the number of contours is automatically determined based on the variance of the z data. The code below and panel (d) of figure 10.4 demonstrate the use of contour.

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

10.4 Multiple Graphs

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

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 can be used in a subplot. The code below and output in figure 10.5 demonstrates how different data visualizations may be used in every cell. These also show some of the available plotting 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
```

Note: The graphics code in each subplot was taken from the function's help file (see `doc` *function*). The help system is comprehensive and most functions are illustrated with example code.

10.5 Advanced Graphics

While the standard graphics functions are powerful, these functions are not flexible enough to express all available options. For example, it is often useful to change the thickness of a line in order to improve its

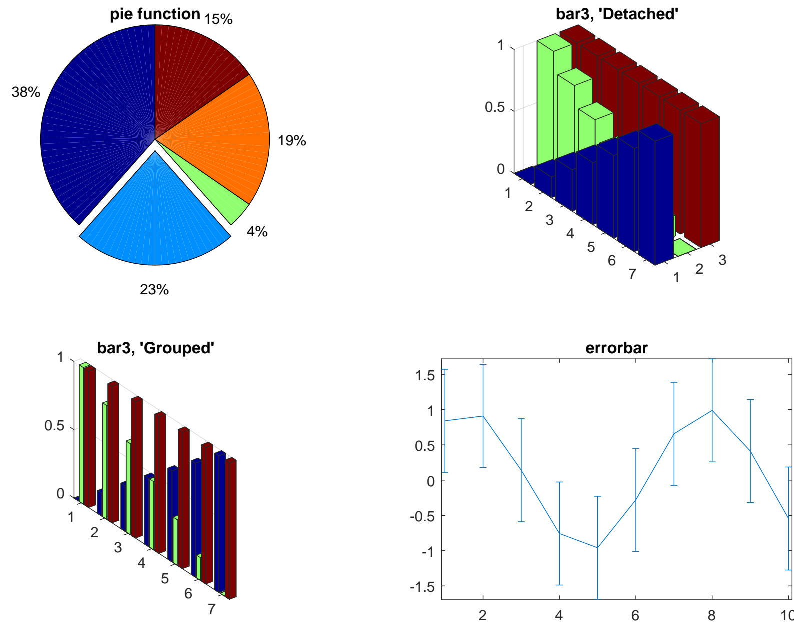

Figure 10.5: Subplots allow for more than one graphic to be included in a figure. This particular subplot contains three different types of graphics with two variants on the 3-D bar. The upper left contains a call to `pie`, the upper right contains a call to `bar3` specifying the option `'grouped'`, the lower left contains a call to `bar3` specifying the options `'detached'` and the lower right contains the results to a call to `errorbar`.

appearance or to add an arrow to highlight a particular feature of a graph.

Two mechanisms are provided 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.

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 10.6. 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 help browser.

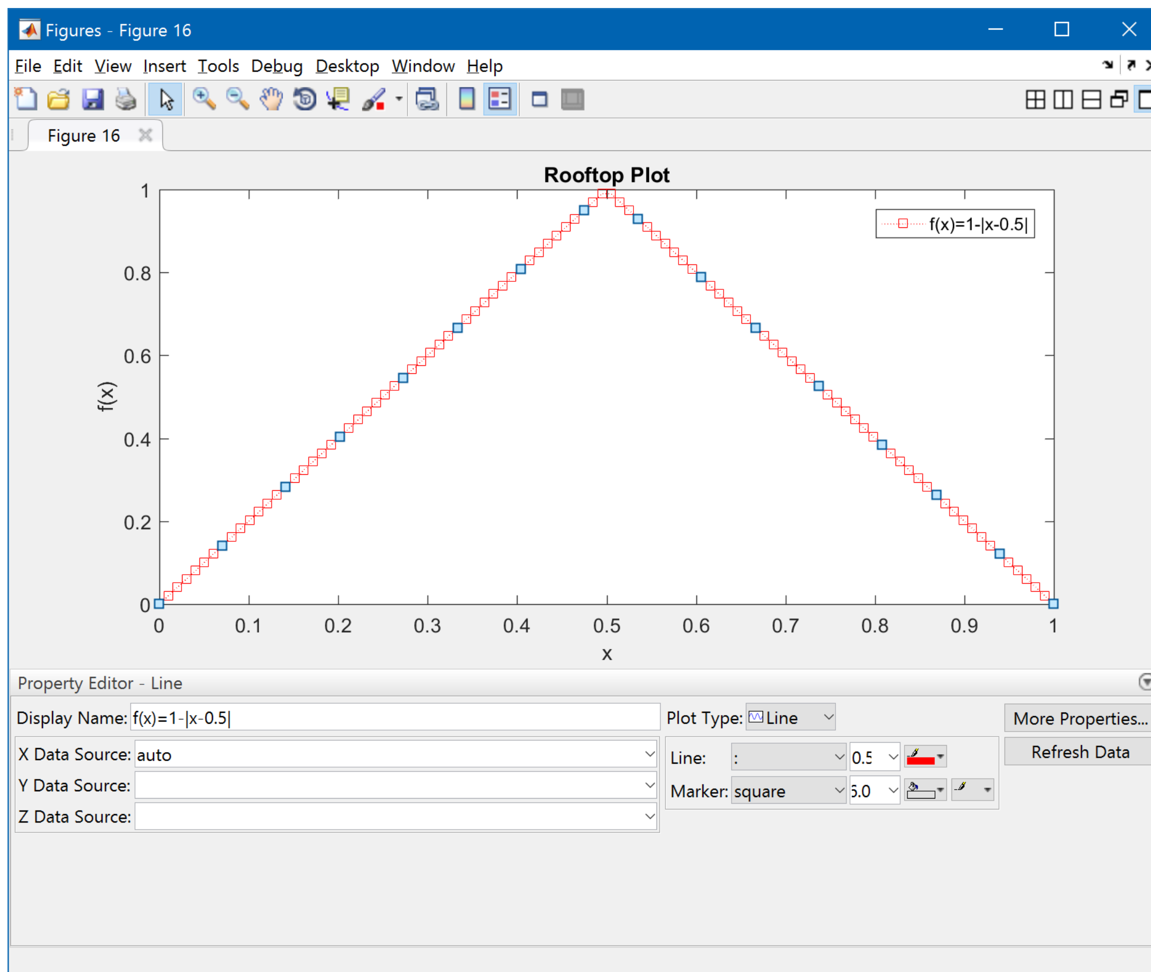

Figure 10.6: Most features of a plot can be editing using the interactive editing tools of a figure window. Interactive editing can be started by first selecting the arrow icon along the top of the figure (**1**), then clicking on the element to be edited (e.g. the line, the axes, any text label). This will bring up the Property Editor (**2**) where the item-specific properties can be changed (**3**). Alternatively, the interactive editing features can be enabled by selecting Edit>Figure Properties.

Handle Graphics

The MATLAB graphics system is fully programmable. Anything that can be accomplished through manual editing of a plot can be accomplished by using *handle graphics* since every graphical element is assigned a handle. The handle contains everything there is to know about the particular element, such as the color or line width. Once familiar with handle graphics, they can be used to create spectacularly complex data visualizations. The use of handle graphics will be illustrated through an 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','Location','Southwest');
t  = title('Two Random Walks');
xl = xlabel('Day');
yl = ylabel('Level');
set(h(1),'Color',[1 0 0],'LineWidth',3,'LineStyle',':')
set(h(2),'Color',[1 .6 0],'LineWidth',3,'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 modifications that can be made using handle graphics can be implemented using the point-and-click editing method previously outlined. The advantage of handle graphics is only apparent when a figure needs to be updated or redrawn. If handle graphics have been used, it is only necessary to change the data and the re-run the code. If using the point-and-click editing method, any change in the data or model requires manually reapplying the edits. For more on handle graphics, please consult the *Handle Graphics Properties* in the help file.

10.6 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 10.8). In the Save as type box, select the desired format (TIFF for Microsoft Office, EPS or PDF file for LaTeX(**2** in figure 10.9)), enter a file name (**1** in figure 10.9) and save. Figures 10.8 and 10.9 contain representations of the steps needed to export from a figure box.

If the exported figure does not appear as desired, it may be necessary to alter the shape of the figure's window. Exported figures are What-You-See-Is-What-You-Get (WYSIWYG). Figure 10.10 contains an example of a figure with reasonable proportions while the axes in Figures 10.11 and 10.12 poorly scaled. The following code will the three figures.

```
fig = figure(1);
x = linspace(0,1,100);
y = 1-abs(x-0.5);
plot(x,y,'r')
xlabel('x');
ylabel('y=1-|x-0.5|');
title('Roof-top plot');
legend('f(x)=1-|x-0.5|');
set(fig,'Position',[445  -212   957   764]);
```

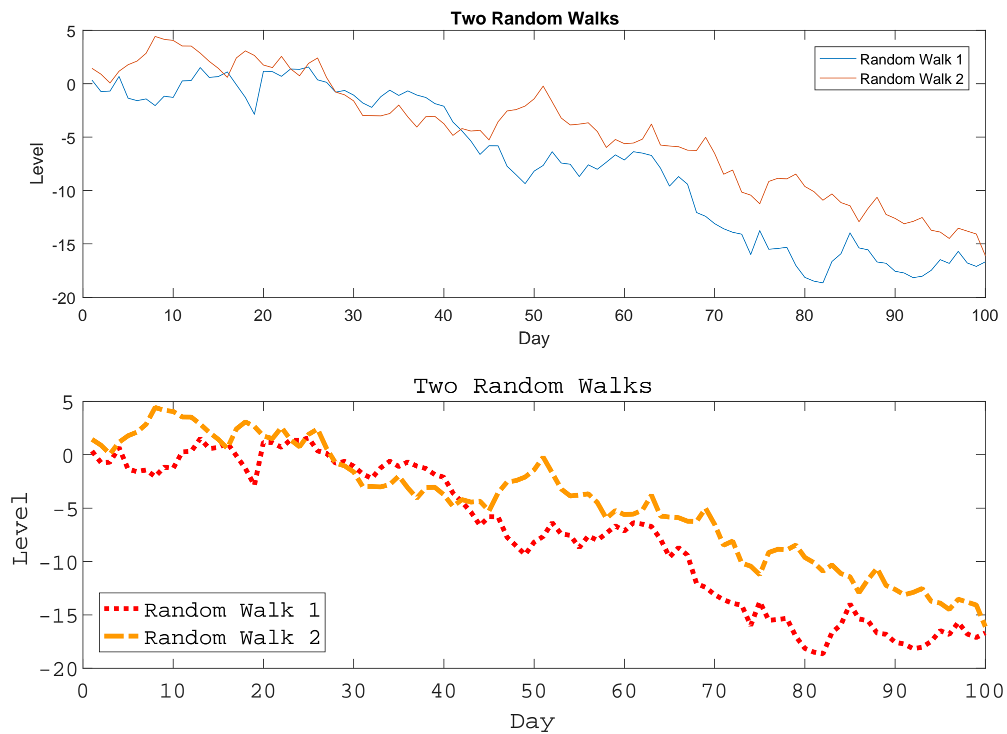

Figure 10.7: The top subplot is a standard call to `plot` while the bottom highlight some of the possibilities when using handle graphics. It is worth nothing that all of these changes evidenced in the bottom subplot can be reproduces using the point-and-click method.

```
fig = figure(2);
x = linspace(0,1,100);
y = 1-abs(x-0.5);
plot(x,y,'r')
xlabel('x');
ylabel('y=1-|x-0.5|');
title('Roof-top plot');
legend('f(x)=1-|x-0.5|');
set(fig,'Position',[ 445  -212   461   764]);

fig = figure(3);
x = linspace(0,1,100);
y = 1-abs(x-0.5);
plot(x,y,'r')
```

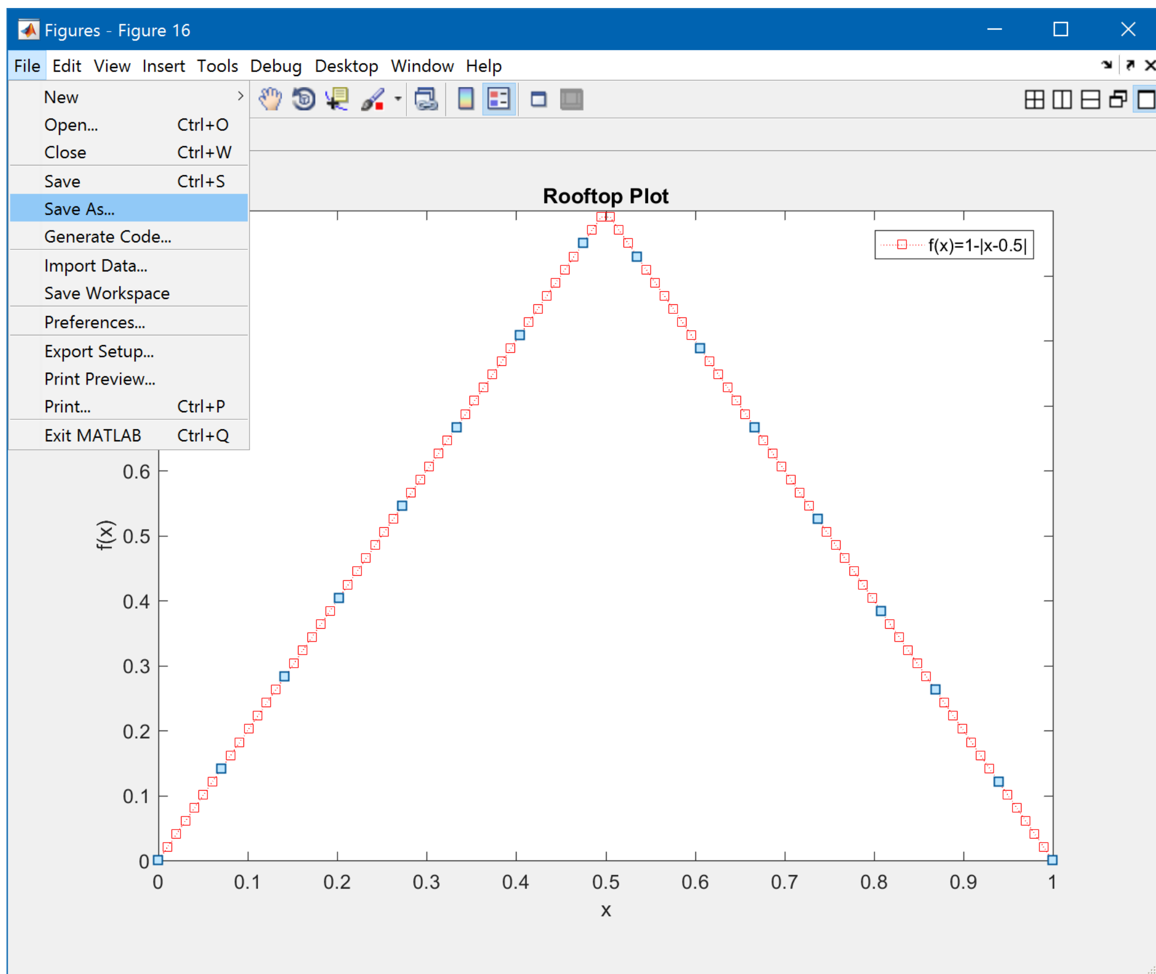

Figure 10.8: To export a figure, click *Save As...* in the file menu of a figure (**1**). The dialog in figure 10.9 will appear.

```
xlabel('x');
ylabel('y=1-|x-0.5|');
title('Roof-top plot');
legend('f(x)=1-|x-0.5|');
set(fig,'Position',[  445    216    957    336]);
```

10.6.1 print

Figures can be programmatically exported using the `print` command. The basic structure of the command is `print -d`*format filename* where *format* is `epsc2` for color encapsulated postscript (EPS, LaTeX or Microsoft Office), `pdf` for portable document format (LaTeX) or `tiff` for TIFF (Microsoft Office). When explorint to PDF, it is a good idea to use the additional flag `-fillpage`. Figures exported in EPS or PDF formats are vector images and scale both up and down well. TIFF images are static and become blurry when scaled.
Note: It is necessary to call `set(gcf,'Color',[1 1 1],'InvertHardcopy','off')` before `print` to remove the gray border surrounding the figure.

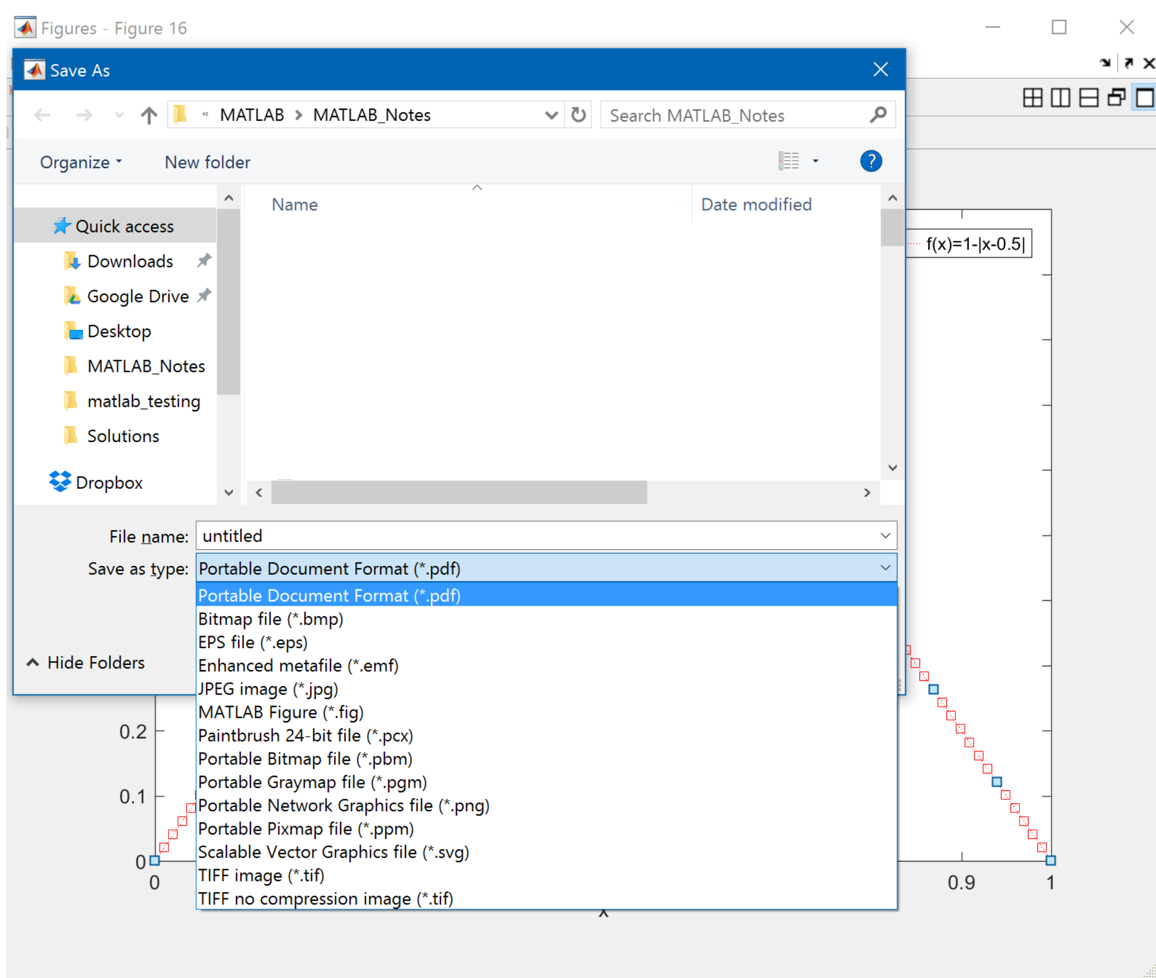

Figure 10.9: To export a figure, enter a file name and use the drop-down box to select a file type. Select TIFF image if using Microsoft Office or EPS File (Encapsulated Postscript) if using LaTeX.

```
fig = figure(1);
x = linspace(0,1,100);
y = 1-abs(x-0.5);
plot(x,y,'r')
xlabel('x');
ylabel('y=1-|x-0.5|');
title('Roof-top plot');
legend('f(x)=1-|x-0.5|');
set(fig,'Position',[445  -212   957   764]);
set(gcf,'Color',[1 1 1],'InvertHardcopy','off')
print -depsc2 ExportedFigure.eps
print -dtiff ExportedFigure.tiff
print -dpdf -fillpage ExportedFigure.pdf
```

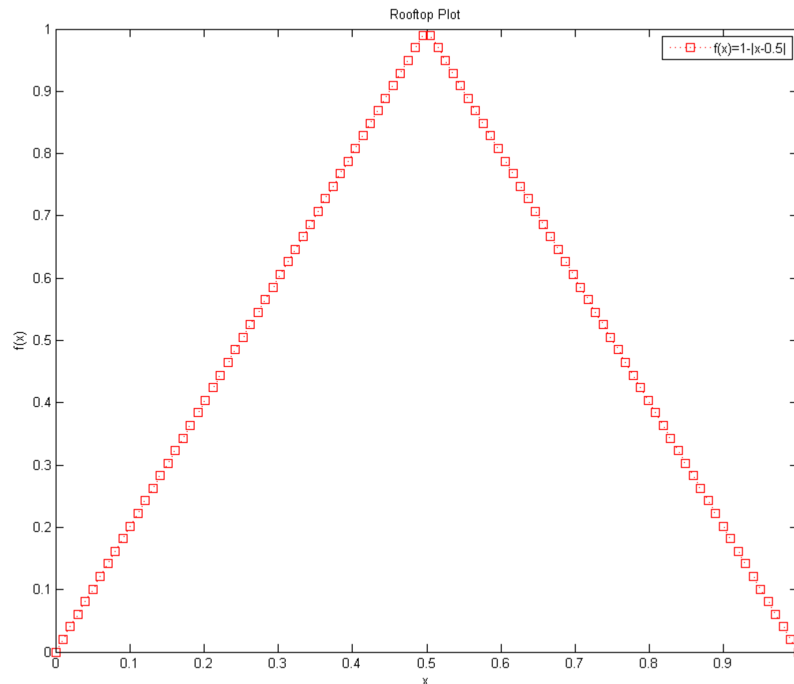

Figure 10.10: Exporting figures is What-You-See-Is-What-You-Get. The axes in this figure are appropriately scaled.

10.7 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 3-D plot from

```
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 ExxonMobil and S&P 500 data (see the Chapter 14 exercises), produce a 2×2 subplot containing:

   - A scatter plot of the two series
   - Two histograms of the series
   - One plot of the two series against the dates using both MATLAB series dates and a `datetime` array. Change the axis labels to text using `datetick` for the serial date solution.

5. Export the plot from exercise 1 as a TIFF, an EPS and a PDF. View the files created outside of MATLAB.

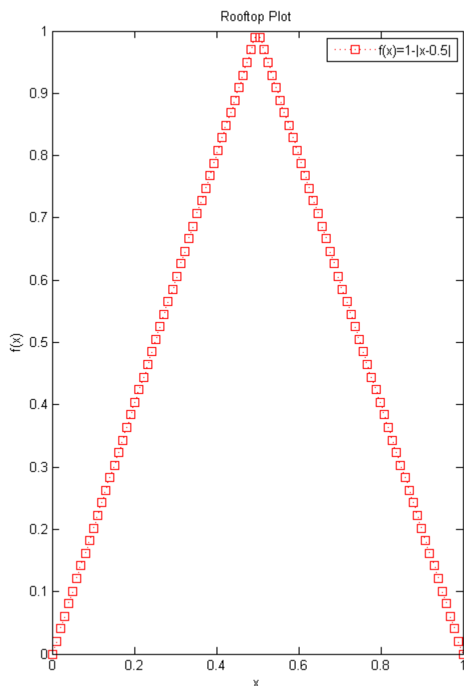

Figure 10.11: Exporting figures is What-You-See-Is-What-You-Get. The axes in this figure are poorly scaled and the height is too large for the width.

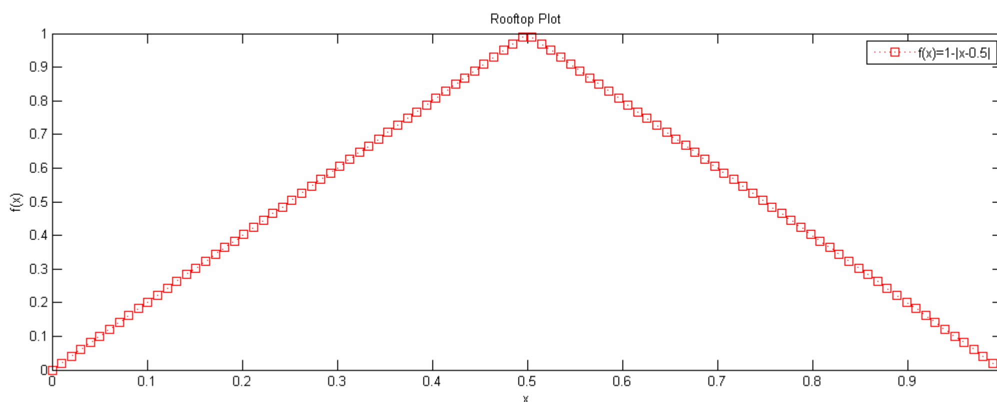

Figure 10.12: Exporting figures is What-You-See-Is-What-You-Get. The axes in this figure are poorly scaled and the width is too large for the height.

6. Use page setup to change the orientation and dimensions as described in this chapter. Re-export the figure as both a TIFF and EPS (using different names) and compare the new images to the old versions.

Chapter 11

Dates and Times

Tracking dates is crucial when working with time-series data. MATLAB provides two methods to store dates and times. The modern implementation is known as a `datetime`. `datetime`s are optimized format that is human readable in the console and provides support extended information such as time zones. The legacy date format is known as MATLAB serial dates where dates are stored as *days since January 0, 0000*.[1] For example, January 1, 0000 is 1 in MATLAB date format and January 1, 2000 is 730,486. Serial dates store hours as fractional days, and so 12:00 January 1, 2000 is 730,486.5.

11.1 MATLAB `datetime`s

MATLAB `datetime`s are a modern, flexible data type for storing dates and times. Theey provide nanosecond resolution and support time zone information. The latter is important in many domains since 18:15:32 GMT on January 31, 2016 is the same as 13:15:32 in New York.

The standard method to create a `datetime` array is to call `datetime` on a cell array of strings. This will produce an array of `datetime`.[2]

```
>> dates = {'12/31/1999','1/1/2000','1/2/2000'}
>> datetimes = datetime(dates)
datetimes =
  31-Dec-1999   01-Jan-2000   02-Jan-2000
```

`datetime`s can also be created from numeric values containing year, month, day, and optionally hour, minute, second and millisecond.

```
>> year = [1999 2000 2000];
>> month = [12 1 1];
>> day = [31 1 2];
>> datetimes = datetime(year, month, day)
datetimes =
  31-Dec-1999   01-Jan-2000   02-Jan-2000
>> hours = [23 6 18]
>> minutes = [59 0 30]
```

[1] Serial dates in MATLAB are numeric values and do not require special treatment.

[2] While the implementation of a `datetime` is not directly exposed to users, in 2016a each `datetime` is stored using 16 bytes of memory. This is twice as much storage as a MATLAB Serial date and allows for additional information about the date and time, such as a time zone, to be stored.

```
>> seconds = [59 0 18]
>> ms = [999, 0, 300]
>> datetimes = datetime(year, month, day, hours, minutes, seconds, ms)
datetimes =
  31-Dec-1999 23:59:59 01-Jan-2000 06:00:00 02-Jan-2000 18:30:48
```

datetimes can be created from other formats using the optional argument 'ConvertFrom' followed by a supported format such as 'datenum' (MATLAB Serial dates) or 'excel' (Excel dates). Finally, a small number of frequently used dates can be created from string arguments, including 'now', 'today', 'tomorrow' and 'yesterday'.

```
>> datetime('now')
ans =
  07-Oct-2016 15:47:59
>> datetime('today')
ans =
  07-Oct-2016
>> datetime('yesterday')
ans =
  06-Oct-2016
```

11.1.1 datetime properties

datetimes are objects and properties of a datetime can be accessed using dot notation. Available properties include the components of the date such as year, month, day, the components of the time such as hour or second, and information about the timezone of the datetime.

```
>> n = datetime('now');
>> n.Year
ans =
       2016
>> n.Hour
ans =
   15
>> n.TimeZone
ans =
    ''
>> n.SystemTimeZone
ans =
   Europe/London
```

11.1.2 durations and calendarDurations

durations arise naturally through differencing datetimes. durations are expressed in terms of hours, minutes, seconds.

```
>> datetime('now')
ans =
   07-Oct-2016 15:58:20
>> datetime('now') - datetime('today')
```

```
ans =
   15:58:20
>> datetime('now') - datetime('yesterday')
ans =
   39:58:20
```

durations can be directly created by passing in the number of hours, minutes, seconds and milliseconds. duration can also be used in mathematical expressions to construct datetimes.

```
>> oneday = duration(24,0,0)
ans =
   24:00:00
>> datetime('today') + (0:2) * oneday
ans =
  07-Oct-2016 00:00:00 08-Oct-2016 00:00:00 09-Oct-2016 00:00:00
```

calendarDurations are similar to durations except that are expressed in terms of calendar units such as years, months and days. They are a convenience function for generating sequences that are regular in terms of a calendar but do not have a uniform duration in terms of hours.

```
>> oneyear = calendarDuration(1,0,0)
oneyear =
  1y
>> datetime('today') + (0:2) * oneyear
ans =
  07-Oct-2016   07-Oct-2017   07-Oct-2018
```

11.1.3 NaT

Like NaN for numeric values, datetimes support a specific missing value – NaT (not a time). Importing unrecognizable date strings will produce NaTs.

```
>> datetime({'12/31/1999','12/32/1999'})
ans =
  31-Dec-1999   NaT
```

Like NaNs, operations involving NaTs will produce NaTs.

```
>> dt = datetime({'12/31/1999','12/32/1999'});
>> dt(1) - dt(2)
ans =
  01-Jan-2000 00:00:00   NaT
```

11.2 MATLAB Serial Dates

Serial dates store dates as numbers based on the relative distance to January 0, 0000. Since these are simply numbers, special purpose functions are required to convert to a human readable format. datetimes are preferred to serial dates, and these are primarily retained for legacy compatibility.

11.2.1 Core Date Functions

11.2.1.1 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 (padded with 0 if single digit)
MM	Minutes (padded with extra 0 if single digit)
SS	Seconds (padded with extra 0 if single digit)

While common string formats are automatically recognized, format strings allow virtually any date format to be converted to MATLAB serial dates. Format strings 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). Consider the following examples showing both automatic detection and the use of format strings.

```
>> 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=char('01JAN2000','02JAN2000','03JAN2000')
strdates =
  01JAN2000
  02JAN2000
  03JAN2000

>> datenum(strdates)
```

```
ans =
      730486
      730487
      730488
```

datenum can additionally 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 (using [year month day hour minute second] format).

11.2.1.2 datestr

datestr is the "inverse" of datenum – it produces a human readable string from a MATLAB serial date. By default, datestr returns string dates of the form 'dd-mmm-yyyy'. datestr also provides a number of standard formats such as 'mm/dd/yy' or 'mmm.dd,yyyy'. To produce one of standard date formats, use datestr(serialdate, #) where # corresponds to one of the format strings (see doc datestr for a list). datestr can also produce strings with arbitrary formats by providing a format string (e.g. use '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,'dd;mm;yyyy')
ans =
01;01;2000
```

Like datenum, datestr can take a vector input and return a 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
```

11.2.1.3 datevec

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

```
>> serial_date=datenum(strvcat('01JAN2000','02JAN2000','03JAN2000 12:00:00'))
serial_date =
    730486
    730487
    730488.5
>> datevec(serial_date)
ans =
       2000          1          1          0          0          0
       2000          1          2          0          0          0
       2000          1          3         12          0          0
```

corresponds to 0:00 (midnight) on January 1 and 2, 2000 and 12:00 (noon) on January 3, 2000.

11.2.1.4 Additional Date and Time Functions

11.2.1.5 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.

11.2.1.6 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 =
    0.0630
```

11.2.1.7 tic and toc

tic and toc can be used for timing code to find hot spot – segments of code which take the majority of the computational time. For example,

```
>> tic
>> j=1; for i=1:1000000; j=j+1; end
>> toc
Elapsed time is 0.010740 seconds.
```

11.3 Converting between datetimes and Serial Dates

MATLAB series dates are numbers while datetime requires string dates. Serial dates can be converted to datetimes using the optional arguments 'ConvertFrom','datenum' when calling datetime. datetimes can be directly converted to serial dates using datenum.

```
>> dates = {'12/31/1999','1/31/2000','2/29/2000'}'
>> serial = datenum(dates)
serial =
     730485
     730516
     730545
>> datetimes = datetime(serial,'ConvertFrom','datenum')
datetimes =
   31-Dec-1999 00:00:00
   31-Jan-2000 00:00:00
   29-Feb-2000 00:00:00
>> serial = datenum(datetimes)
serial =
     730485
     730516
     730545
```

11.4 Dates on Figures

Plotting with dates can be implemented using either datetimes or serial dates. When plotting with datetimes, the plot will automatically show human readable dates. When plotting with serial dates (which are just numbers), datetick is required to converts an axis of a plot expressed in MATLAB serial dates to text dates. For example,

```
>> dates = datenum('01Jan2000'):datenum('31Dec2000');
>> rw    = cumsum(randn(size(dates)));
>> subplot(3,1,1);
>> plot(dates, rw);
>> subplot(3,1,2);
>> plot(dates, rw);
>> datetick('x')
>> subplot(3,1,3);
```

```
>> datetimes = datetime(dates, 'ConvertFrom', 'datenum');
>> plot(datetimes, rw);
```

produces the two plots in figure 11.4. The top plot contains MATLAB serial dates along the x-axis while the bottom contains string 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 then use `datetick('x','keepticks','keeplimits')` as illustrated in figure 11.4.

```
>> figure()
>> 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')
```

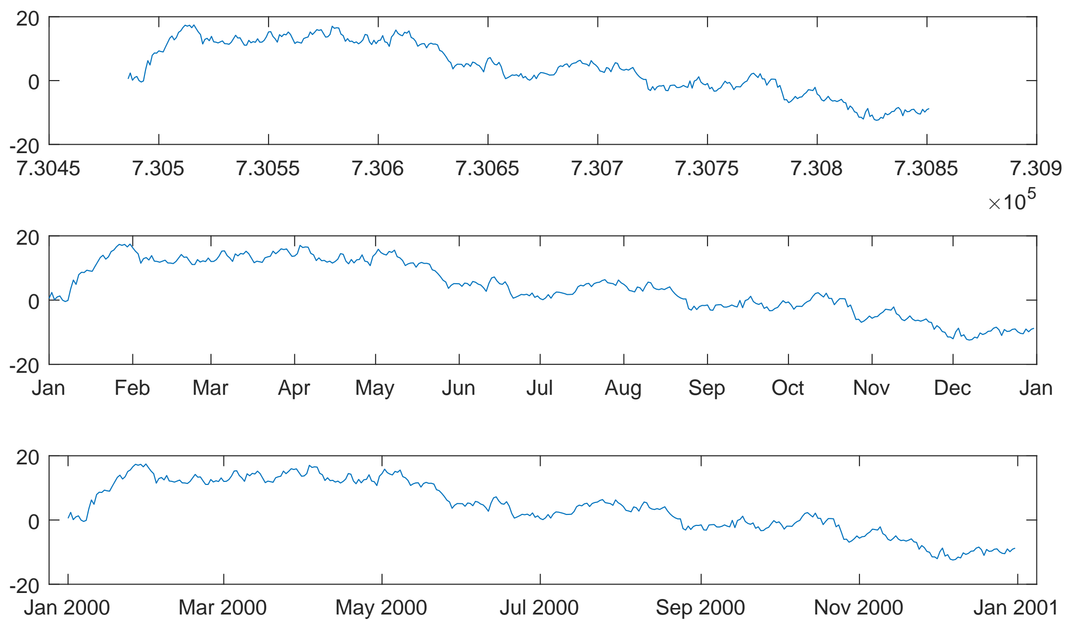

Figure 11.1: `datetick` converts MATLAB serial dates into text strings. Unfortunately, `datetick` changes the location of points and makes fairly bad choices. The solution is to use `datetick('x','keepticks','keeplimits')`. The bottom panel uses `datetimes` and so does not require a call to `datetick`.

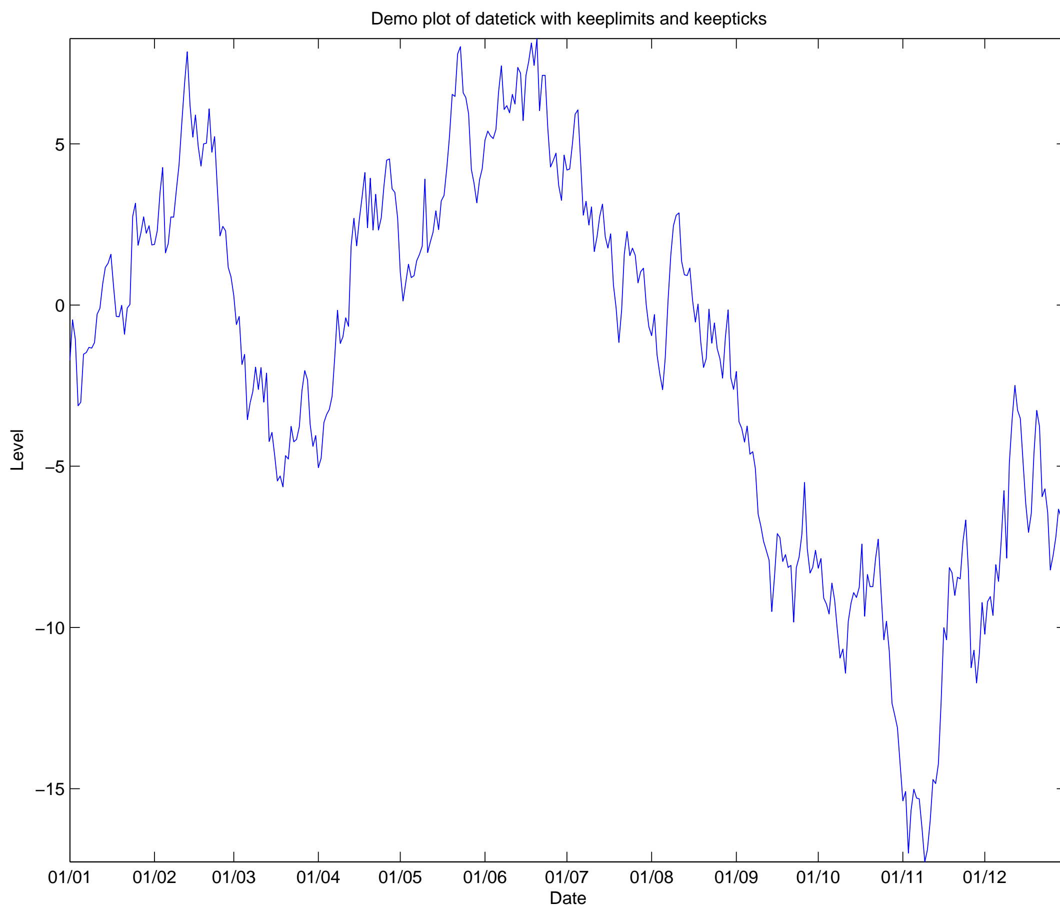

Figure 11.2: `datetick` with `keepticks` and `keeplimits`. These two arguments ensure `datetick` behaves in a consistent manner. To use them, set up the figure as is should look but with serial dates on the axis, and then call `datetick('x','keepticks','keeplimits')`.

Chapter 12

String Manipulation

While manipulating text is not MATLAB's forté, the programming environment does provide a complete set of tools for working with strings. Simple strings can be input from the command line

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

Strings are treated as matrices of character data, and so they respect the standard behavior of most commands (e.g. `str(1:10)`). However, using commands designed for numerical data is tedious and special purpose functions are provided to assist with string data.

The primary application of string functions is to parse data. Chapter 14 contains an example of parsing a poorly formatted file. It uses a number of string functions to manipulate and parse the text of a file.

12.1 String Functions

char

`char` has two uses. The first is to convert integer numerical values between 1 and 127 into their ASCII equivalent characters.[1] Non-integer values are truncated to integers using `floor` and then converted.

```
>> char(65:100)
ans =
ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_‘abcd
>> char(25*pi)
ans =
N
```

The second use of `char` is to vertically concatenate strings (stack) which do not (necessarily) have the same length.

```
>> s1 = 'A string';
>> s2 = 'A longer string';
>> s3 = 'An even longer string';
>> char(s1,s2,s3)
ans =

A string
```

[1]Values up to 65535 are permitted to allow unicode characters.

```
A longer string
An even longer string
```

Note that `char` works similarly to `strvcat`, although the latter deprecated and should not be used.

double

`double` converts character strings into their numerical values.

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

upper and lower

`upper` and `lower` convert strings to all upper case and lower case, respectively.

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

strfind

`strfind` returns the index of the all matching strings in a text block, such as delimiting characters in a block of text. 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, and 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, and is case sensitive. strncmpi 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
```

strsplit and strjoin

strsplit allows a string to be split into a cell array using a character as a delimiter. strjoin is the inverse function and will join a cell array containing strings into a single string separated by a user provided character.

```
>> str = 'Econometrics is my favorite subject.';
>> split = strsplit(str, ' ')
split =
   'Econometrics'  'is'  'my'  'favorite'  'subject.'
>> joined = strjoin(split, '-')
joined =
   Econometrics-is-my-favorite-subject.
```

regexp and regexpi

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

12.2 String Conversion

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
```

str2double

str2double converts string values into numerical varies. Unlike str2num it only operates only on scalars or cell arrays, and when used on a cell array, each cell must contain only a single string to convert. str2double offers better performance when it is applicable.

num2str

num2str converts numerical values into strings. The input can be scalar, 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
```

sscanf

sscanf can be used to convert strings to text, and is by far the fastest method to convert large text blocks to numbers. The generic form of sscanf is
sscanf(*text* ,*format*)
where *text* is a numeric character string and *format* contains information about the format of the values in *text*. sscanf operates column-by-column so that lines must be stored in columns (or if stored in rows, the input can be transposed). The space character is used to delimit the end of an entry and so it is essential that the input string must be padded by a space.[2] The format string can handle a wide variety of cases, although the most important are %d, which converts a string to a base-10 (32-bit) integer, and %f, which converts a string to a floating point. Consider the following example which generates 10,000 random numeric strings using randi and then parses the text using sscanf, str2num and str2double.

```
>> text = char(47+randi(10,10000,6)); % Random numeric string
>> text = [text repmat(' ',10000,1)];  % Pad with space
>> tic; numericValues = sscanf(text','%d'); toc
Elapsed time is 0.005850 seconds.

>> tic; numericValues = str2num(text); toc
Elapsed time is 0.234914 seconds.

>> tic; for i=1:10000; numericValues(i) = str2double(text(i,:)); end; toc
Elapsed time is 0.597951 seconds.
```

sscanf is about 100 times faster than str2num and str2double. Format strings can include multiple elements in which case the formats are sequentially applied until the end of the text string is reached.

[2]Technically, sscanf operates on `text(:)` (which is a single column vector constructed by stacking the input text). This is why it is essential that lines are padded by a space.

```
>> text = num2str([pi floor(exp(1)) (1+sqrt(5))/2])
text =
3.1416      2       1.618

>> sscanf(text','%f %d %f')
ans =
       3.1416
            2
        1.618
```

Note that sscanf terminate without an error when an unexpected string is encountered.

```
>> text = [num2str([pi floor(exp(1))]) ' A ' num2str((1+sqrt(5))/2)]
text =
3.1416           2 A 1.618

>> sscanf(text','%f')
ans =
                   3.1416
                        2
```

In the example above, sscanf stops when it encounters the A and returns the first two values. It is important to verify that the strings contain only the expected data (e..g. only numeric types, including .) prior to the command.

fprintf

fprintf allows formatted text to be output to the screen or to files.

12.3 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 11.

Chapter 13

Structures and Cell Arrays

Structures and cell arrays are advanced data storage formats that often provide useful scaffolding for working with mixed (i.e. string and numeric) or structured data.

13.1 Structures

Structures allow related pieces of data to be organized into a single variable. Structures are constructed using

variable_name.field_name

syntax where both *variable_name* and field_name must be valid variable names. One application of structures is to organize data. Consider the case of working with data that comes in triples which correspond to x-, y- and z-data. One alternative would be to store the data as a 3 by 1 vector. Alternatively, a structure could be used with field names x, y and z to provide added guidance on what is expected.

```
>> coord.x = 0.5
coord =
    x: 0.5000
>> coord.y = -1
coord =
    x: 0.5000
    y: -1
>> coord.z = 2
coord =
    x: 0.5000
    y: -1
    z: 2
```

Structures can also be used in arrays (array of structures), which can either be constructed using the command `struct` or lazily initialized by concatenation. Continuing from the previous example,

```
>> coord(2).x = 3
coord =
1x2 struct array with fields:
    x
    y
    z
```

```
>> coord(2).y = 2
coord =
1x2 struct array with fields:
    x
    y
    z
>> coord(2).z = -1
coord =
1x2 struct array with fields:
    x
    y
    z
```

The elements of the array of structures can be accessed like any other array with the caveat that the assignment will itself be a structure.

```
>> newCoord = coord(1)
newCoord =
    x: 0.5000
    y: -1
    z: 2
```

Structures can also be used to store mixed data.

```
>> contact.phoneNumber = 441865281165
contact =
    phoneNumber: 4.4187e+011
>> contact.name = 'Kevin Sheppard'
contact =
    phoneNumber: 4.4187e+011
           name: 'Kevin Sheppard'
```

13.1.1 The Problem with Structures

The fundamental problem with structures in MATLAB is that they are difficult to work with, and that operating on structures requires operating on the fields one-at-a-time. Structures are also difficult to preallocate and so performance issues arise when used in large arrays. Structures are still commonly used (for example, in `optimset`), although they have been supplanted by a more useful object, the cell array. It is tempting to use structures to push large collections of data, parameters and other values into and out of functions. This is generally a bad practice and should be avoided.

13.2 Cell Arrays

Cell arrays are a powerful alternative to the "everything is a matrix" model of classic MATLAB. Cell arrays are formally jagged (or ragged) arrays and are collections of other arrays (possibly other cell arrays). Cell arrays can be thought of as generic containers where the final elements are one of the MATLAB primitive data types (e.g. a matrix). The are most useful when handling either pure string data or mixed data which contains both string values and numbers. Cell arrays manipulation is similar to matrix manipulation although there are some important caveats.

Cell arrays can be initialized using the `cell` command or directly using braces (`{}`). In either case, braces are used to access elements within a cell array. The example below shows how cell arrays can be pre-allocated using `cell` and then populated using braces.

```
>> cellArray = cell(2,1) % Initialize a cell array
cellArray =
        []
        []
>> cellArray{1} = 'cell' % Add an element using braces { }
cellArray =
    'cell'
        []
>> cellArray{2} = 'array'
cellArray =
    'cell'
    'array'
```

Initially the variable was an empty cell array. After the string vector `'cell'` was added in the first position, only the second was empty. Finally, the string vector `'array'` was placed into the second position. This simple example show the ease with which cell arrays can be used to handle strings as opposed to using matrices of characters which becomes problematic when some of the rows may not have the same number of characters, which are required to be padded with blank characters (and then deblanked before being used).

Cell arrays are also adept at handling mixed data, as the next example shows.

```
% Initialize a cell array
>> cellArray = cell(2,1);
>> cellArray{1} = 'string'
cellArray =
    'string'
        []
>> cellArray{2} = [1 2 3 4 5]
cellArray =
    'string'
    [1x5 double]
>> cellArray{2}
ans =
     1     2     3     4     5
```

The cell array above has a string in the first position and a 5 by 1 numeric vector in the second. Cell arrays can even contain other cell arrays, and so can be used to store virtually any data structure by nesting.

```
% Initialize a cell array
>> cellArray{3} = cell(2,1)
cellArray =
    'string'
    [1x5 double]
    {2x1 cell  }
```

13.2.1 Accessing Elements of Cell Arrays

The primary method for accessing cell arrays is through the use of braces ({}) as the two previous examples demonstrated. Selecting an element using braces returns the contents of the cell and can be used to assign the values for processing using functions that are not designed for cell arrays. Continuing from the previous example,

```
>> x = cellArray{1}
x =
string
>> y = cellArray{2}
y =
     1     2     3     4     5
```

Cell arrays can also be accessed using parentheses although this type of access is markedly different from accessing cell arrays with braces. Unlike braces which access the contents of a cell, parentheses access the cell itself and not its contents. The difference in behavior means that subsets of a cell array can be assigned to another variable without iterating across the contents of the cell array.

```
>> cellArray = cell(3,1);
>> cellArray{1} = 'one';
>> cellArray{2} = 'two';
>> cellArray{3} = 'three';
cellArray =
    'one'
    'two'
    'three'

% Correct method to reassign elements of a cell array to a new array using parentheses ( )
>> newCellArray = cellArray(1:2)
newCellArray =
    'one'
    'two'

% Incorrect method to reassign elements of a cell array to a new array using braces { }
>> newCellArray = cellArray{1:2}
newCellArray =
one
```

In the correct example above, `newCellArray` contains the first elements of `cellArray`. Also note the incorrect attempt to assign the first two elements using braces which does not produce the desired result.

13.2.2 Considerations when Using Cell Arrays

Cell arrays, like structures, are useful data structures for working with strings or mixed data. Cell arrays are generally superior to structures and there are many functions which can operate directly on cell arrays of strings (e.g. `sort`, `unique`, `ismember`). They do come with some overhead and so are not appropriate for every use. For example, a 2 by 1 vector containing `[1 2]'` requires 16 bytes of memory. A cell array with 1 in its first cell and 2 in its second requires 240 bytes of memory, a 15 fold increase. Due to this overhead cell arrays are undesirable in situations where data is highly regular and where the contents of each cell is small.

Chapter 14

Importing and Exporting Data

Importing data ranges from simple to very difficult, depending on the data size and format. A few principles can simplify this task:

- Use the import wizard to import mixed data in Excel files.
- Variables in Excel files should have one variable per column with a distinct variable name in the top cell of each column.
- Use `readtable` to import either delimited text or Excel files into MATLAB `table`s.
- When importing data using one of the fast but legacy import functions (e.g., `csvread` or `xlsread`), the file imported should contain numbers *only*. The sole exception to this rule is that Excel files can also contain dates (in Excel date format).

14.1 Robust Data Importing

The simplest and most robust method to import data is to use a correctly formatted Excel file and the import wizard. The key to the import is to make certain the data in the Excel file has been formatted according to a simple set of rules:

- One variable per column
- A valid, distinct variable name for the column in the first row
- *All* data in the column must be either numeric or contain dates.

As an example, consider importing a month of GE prices downloaded from Yahoo! Finance. The original data can be found in GEPrices.xlsx and is presented in Figure 14.1. This data file fits the requirements since all columns contain either dates or numbers.

This file can be imported using the following steps. 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 file name and select Import Data... (see figure 14.1). This will trigger the dialog in figure 14.1. To complete the

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

	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

Figure 14.1: The raw data as taken from Yahoo! Finance. All of these columns are well formatted with variable names in the first row and numeric (or date) content.

import, make sure Column Vectors is chosen (top left of Import Wizard) and click Import. If the import fails the most likely cause is the format of the Excel file – make certain the file conforms to the rules above and try again. Alternatively, select Table which will read the data into a single MATLAB `table` (see 15 for more on `table`s).

14.2 Importing Data in Code

The preferred method to import date from files through code is to use `readtable` which will import the data into a `table` (see 15 for more on `table`s). Other methods to read in data include `xlsread`, `csvread`, `textread` and `textscan` which provide data-type specific readers. These lower level readers can be faster than `readtable` although they are also more fragile in the sense that they produce errors when data is not well formatted.

14.2.1 Importing Using `readtable`

For many datasets stored as either delimited text or in an Excel file, `readtable`(*filename*) will import data into a MATLAB `table` without any further options. The imported data will consist of columns, each with a datatype optimized to hold the type of data in the file. In particular, columns of numbers will be imported

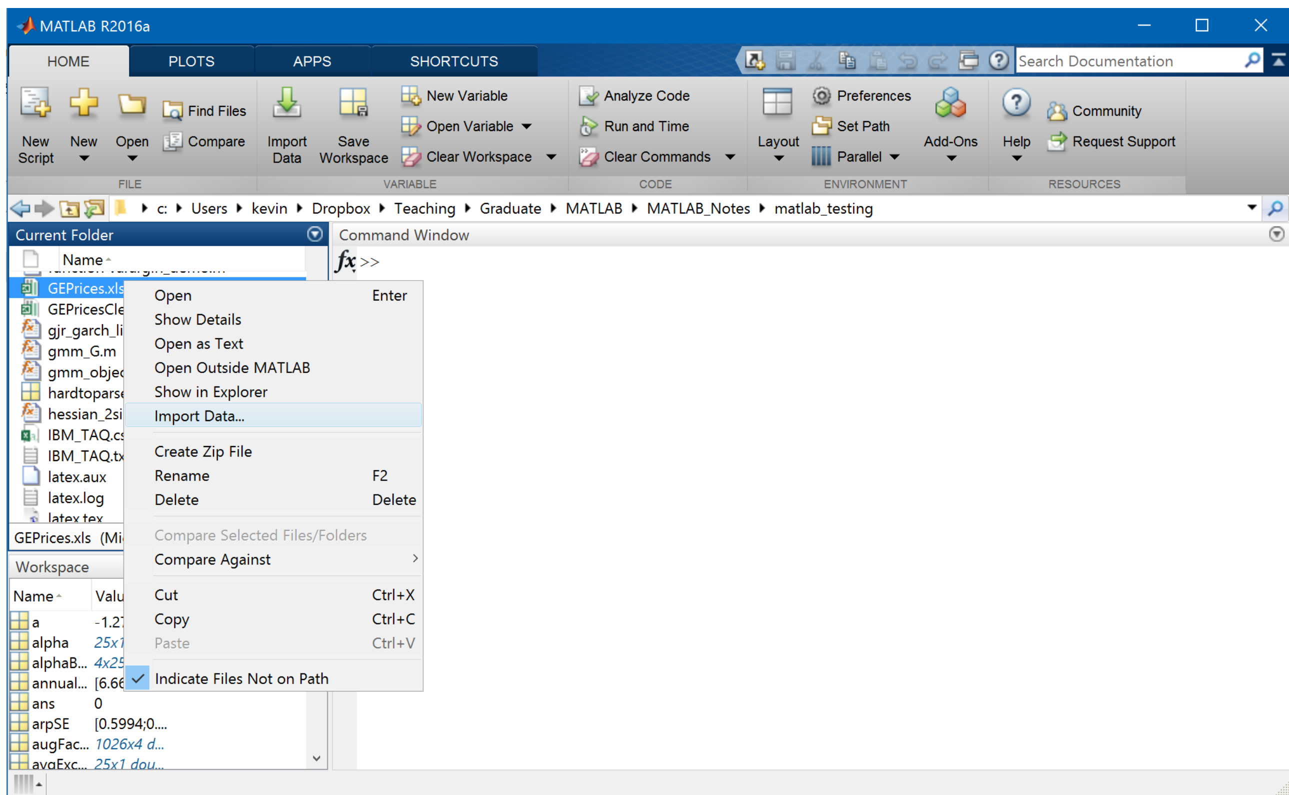

Figure 14.2: To import data, select the Current Directory view, right click on the Excel file to be imported, and select Import. This will trigger the import wizard in figure 14.1.

as numeric arrays while columns of strings or string dates will be imported in a cell array. If the file contains variable names in the first row, the `table` will read these in automatically and use them as the column names. `readtable` supports delimited text, Excel files, and Open Document Spreadsheets, and attempts to infer the type of file from the file's extension.

The CSV below contains the first 10 rows to IBM_TAQ.txt and contains Trade-and-Quote data for IBM on one day.

```
SYMBOL,DATE,TIME,PRICE,SIZE
IBM,20070103,9:30:03,97.18,100
IBM,20070103,9:30:08,96.6,373200
IBM,20070103,9:30:08,97.17,1000
IBM,20070103,9:30:08,97.17,100
IBM,20070103,9:30:08,96.61,200
IBM,20070103,9:30:08,96.75,200
IBM,20070103,9:30:08,97.15,100
IBM,20070103,9:30:08,97.15,100
IBM,20070103,9:30:08,97.15,100
IBM,20070103,9:30:08,97.15,100
```

This file can be imported using `readtable` and columns that are not numbers are imported as strings.

```
>> t = readtable('IBM_TAQ_top_10.csv')
t =
    SYMBOL       DATE          TIME        PRICE       SIZE
```

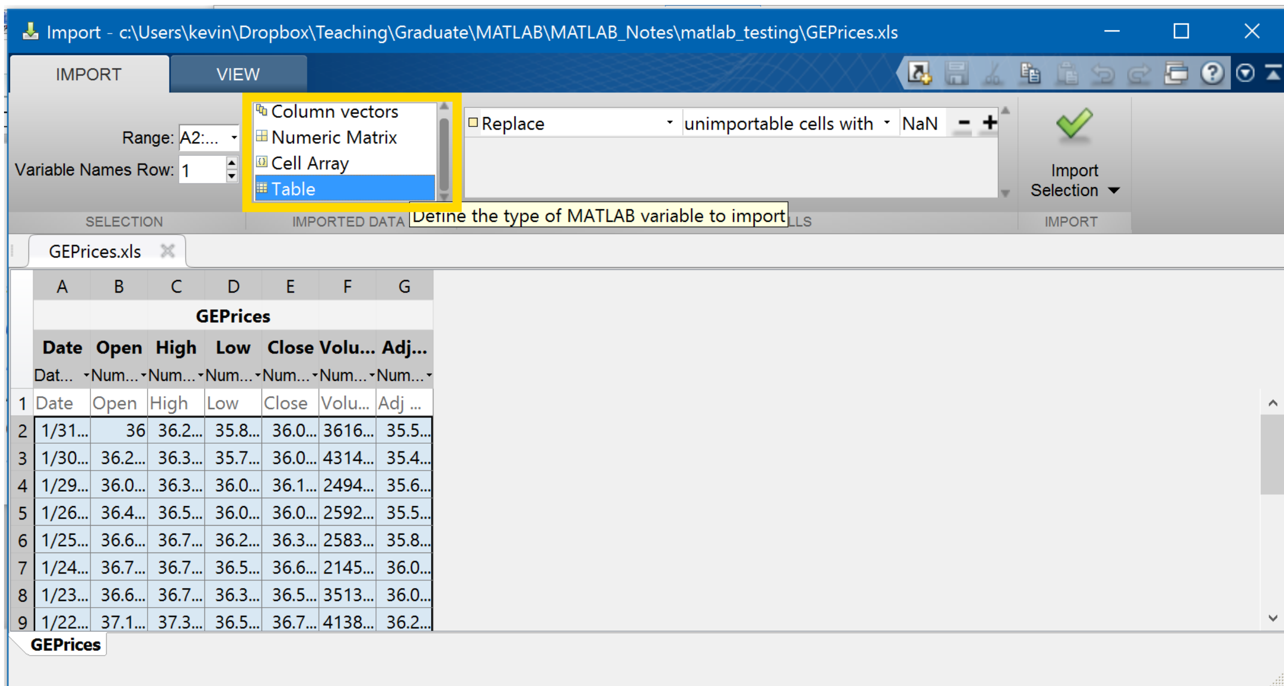

Figure 14.3: As long as the data is correctly formatted, the import wizard should import the data and create variables with the same name as the column headers. To complete this step, make sure that Column vectors is selected from the Import as drop-down box and then select Import.

```
    ______    _________    _________    _____    _________

    'IBM'     2.007e+07    '9:30:03'    97.18          100
    'IBM'     2.007e+07    '9:30:08'     96.6    3.732e+05
    'IBM'     2.007e+07    '9:30:08'    97.17         1000
    'IBM'     2.007e+07    '9:30:08'    97.17          100
    'IBM'     2.007e+07    '9:30:08'    96.61          200
    'IBM'     2.007e+07    '9:30:08'    96.75          200
    'IBM'     2.007e+07    '9:30:08'    97.15          100
    'IBM'     2.007e+07    '9:30:08'    97.15          100
    'IBM'     2.007e+07    '9:30:08'    97.15          100
    'IBM'     2.007e+07    '9:30:08'    97.15          100
```

Some basic reformatting can be used to reformat the DATE and TIME columns as a datetime.

```
>> times = datetime(t.TIME) - datetime('today');
>> dates = datetime(t.DATE,'ConvertFrom','yyyymmdd');
>> t.datetimes = dates + times;
>> t(:,'datetimes')
ans =
        datetimes
    ____________________

    03-Jan-2007 09:30:03
    03-Jan-2007 09:30:08
    03-Jan-2007 09:30:08
```

```
    03-Jan-2007 09:30:08
    03-Jan-2007 09:30:08
    03-Jan-2007 09:30:08
    03-Jan-2007 09:30:08
    03-Jan-2007 09:30:08
    03-Jan-2007 09:30:08
    03-Jan-2007 09:30:08
```

Optional parameters that provide additional information can be passed using the format `readtable(`*filename*`,` *param*`,` *value*`)`. The most useful options for *param* are `'ReadVariableNames'` which takes a value of true or false (1 or 0) and instructs `readtable` to read the names from the file, `'ReadRowName'` which indicates that row names should be read, and `'TreatAsEmpty'` which takes a cell array of strings that `readtable` should treat as missing values (e.g., `{'', 'N/A]}`). When reading text files, optional inputs all the the delimiter to be set (`'Delimiter'`) and lines at the top of the file containing non-date to be skipped (`'HeadLines'`). When reading Excel or OpenDocument Spreadsheet files, the sheet to read can be set (`'Sheet'`) and the range to read, a rectangular region in the sheet, can be set (`'Range'`).

14.2.2 Excel Files Using `xlsread`

Data in Excel sheets can be also 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, 2004, to December 31, 2007. 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, use the command

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

This command will read the data in *sheet1* of file *deciles.xls* and assign it to the variable `data`. `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 `doc 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 `xlsfinfo`.

MATLAB and Excel *do not* agree about dates. MATLAB dates are measured as days past January 0, 0000 while Excel dates are measured relative to December 31, 1899. In MATLAB serial date 1 corresponds to January 1, 0000 while in Excel day 1 corresponds to January 1, 1900. 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,

```
>> [A,finfo]=xlsfinfo('deciles2.xls')
A =
 Microsoft Excel Spreadsheet
finfo =
   'deciles'
>> data  = xlsread('deciles2.xls','deciles','A2:K1257');
>> dates = data(:,1);
>> datestr(dates(1))
ans =
 03-Jan-0104
>> dates = dates + datenum('30DEC1899');
```

```
>> datestr(dates(1))
ans =
 02-Jan-2004
```

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

```
>> data  = xlsread('deciles2.xls','deciles','A2:K1257');
>> dates = data(:,1);
>> datestr(dates(1))
ans =
 03-Jan-0104
>> dates = x2mdate(dates);
>> datestr(dates(1))
ans =
 02-Jan-2004
```

This example uses a files *deciles2.xls* which contains the sheet *deciles*. Opening the files in Excel shows that *deciles* contains column labels as well as the data. To import data from this file, `xlsread` needs to know to take the data from *deciles* in cells *A2:K1275* (upper left and lower right corners of the block). Running the command `xlsread('deciles2.xls', 'deciles', 'A2:K1257')` does this. Finally, the disagreement in the base date is illustrated and the correction is shown to work. For more on dates, see Chapter 11.

14.2.3 Reading CSV Data Using csvread

Comma-separated value (CSV) data is similar to Excel data, although the CSV files *must* contain only numeric values. If the file contains strings, such as variable names, the import will fail. The command to read CSV data is virtually identical to the command to read Excel files,

```
% This command fails since deciles.csv contains variable names in the first row
>> data = csvread('deciles.csv') %Error
Error using dlmread (line 139)
Mismatch between file and format string.
Trouble reading number from file (row 1u, field 1u) ==>
caldt,CAP1RET,CAP2RET,CAP3RET,CAP4RET,CAP5RET,CAP6RET,CAP7RET,CAP8RET,CAP9RET,CAP10RET\n

Error in csvread (line 48)
    m=dlmread(filename, ',', r, c);
```

However, like `xlsread`, `csvread` be given a specific cell to begin reading, and so leading text can be avoided.

```
% This command works since it skips the first row
>> data = csvread('deciles.csv',1,0)
```

or to read specific blocks of cells

```
>> data = csvread('deciles.csv',1,0,[1 0 100 10]);
```

Data can be exported to CSV using `csvwrite`.

14.2.4 Reading Other Delimited Files

`dlmread` allows for text files with other delimiters to be read. These examples show the basic usage of `dlmread` for reading a tab-delimited file and a CSV file.

```
>> textData = dlmread('deciles.txt','\t');
>> csvData = dlmread('deciles.csv',',',1,0); % Skip 1st Row
```

14.3 MATLAB Data Files (.mat)

The native file format is the MATLAB data file or mat file. Data from a mat file is loaded 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. `load` can also be used as a function, which allows for dynamically generated file names.

```
% Function usage
>> load('deciles.mat')
```

Load can be used with a single output which will load all variables in the mat file into a structure (see chapter 13).

```
% Function usage
>> dec = load('deciles.mat')
dec =
    data: [1256x11 double]
>> dec.data(1:2,1)
    20040102
    20040105
```

In recent versions of MATLAB (R2011b or later), it is possible to load only a subset of the variables in a file. Suppose a mat file contained 3 variables, `x`, `y` and `z`. To load only `x` and `y` (but not `z`), use the following command.

```
% Limited import
>> load('datafile.mat','x','y')
```

The related function `whos` can be used to generate a list of the variables in a mat file.

```
% Mat contents
>> contents = whos('-file','deciles.mat')

contents =
       name: 'data'
       size: [1006 11]
       bytes: 88528
       class: 'double'
       global: 0
       sparse: 0
       complex: 0
       nesting: [1x1 struct]
       persistent: 0
```

In this example, `contents` is a structure. If the mat file contains more than 1 variables, an array of structures is returned (see chapter 13 for more on structures).

14.4 Advanced Data Import

14.4.1 Manually Reading Poorly Formatted Text

MATLAB can be programmed to read virtually any text (or even binary) format since it contains functions for parsing and interpreting arbitrary file data. Reading poorly formatted data files is an advanced technique and should be avoided if possible. However, some data is only available in formats where reading in data line-by-line is the best solution.[2] For instance, the standard import method fails if the raw data is very large (too large for Excel) and is poorly formatted. In this case, one solution is to write a program to read and process the file line-by-line.

The file *IBM_TAQ.txt* contains a simple example of data that is difficult to import. 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 is too large to use in Excel and has both numbers, dates and text on each line. The following code block demonstrates one approach to parsing this file.

```
fid=fopen('IBM_TAQ.txt','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);
time  = zeros(count,1);
price = zeros(count,1);
%Reopen the file
fid=fopen('IBM_TAQ.txt','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,',');
```

[2] Line-by-line importing of complex files is slow but relatively straight forward. More advanced users will find that processing complex files in blocks using `fread` is substantially faster. See chapter 21 for more discussion of file importing.

```
    %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`
- Re-reads 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
- Closes the file directly using `fclose`

14.4.2 Reading Poorly Formatted Text Using `textscan`

`textscan` is a relatively fast method to read files that contain mixed numeric and string data. A text file must satisfy some constraints in order for `textscan` to be useful. First, the file must be regular in the sense that it has the same number of columns in every row, and second each column must contain the same type of data – that is, the file must not mix strings with numbers in a column. *IBM_TAQ.txt* is satisfied these two constraints and so can be read using the command block below. `textscan` uses a file handle created using `fopen` as the file input, rather than the file name directly.

```
fid = fopen('IBM_TAQ.csv','rt');
data = textscan(fid, '%s %f %s %f %f', 'delimiter', ',', 'HeaderLines', 1)
fclose(fid);
```

The arguments to textscan instruct the function that the lines are formatted according to string-number-string-number-number where %s indicates string and %f indicates number, that the columns are delimited by a comma, and that the first line is a header and so should be skipped. The data read in by textscan is returned as a cell array, where numeric columns are stored as vectors while string values (the ticker and the time in this example) are stored as cell arrays of strings. The use of curly braces, {} indicates that a cell array is being used. See chapter 13 for more on accessing the values in cell arrays.

```
>> data
data =
  Columns 1 through 3
    {558986x1 cell}    [558986x1 double]    {558986x1 cell}
  Columns 4 through 5
    [558986x1 double]    [558986x1 double]
>> data{1}(1)
ans =
    'IBM'
>> data{2}(1)
ans =
    20070103
>> data{3}(1)
ans =
    '9:30:03'
>> data{4}(1)
ans =
   97.1800
>> data{5}(1)
ans =
   100
```

Note that the time column would need further processing to be transformed into a useful format. For more on reading poorly formatted data file, see the documentation for fopen, fscanf, fread, fgetl, dlmread, and textscan. See chapter 12 for more on string manipulation. textscan is a good solution for files with mixed data which are not excessively large – large files tend to be very slow due to the use of cell arrays.

14.5 Exporting Data

14.5.1 Saving Data

Once the data has been loaded, save it and any changes in the native MATLAB data format using save

```
>> save filename
```

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

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

which saves the file *filename.mat* containing var1, var2, and var3. save, like load, can also be used as a function which allows for using a variable as the file name.

```
>> saveFileName = 'filename';
>> save(saveFileName,'var1','var2','var3')
```

14.5.2 Exporting Data for Use in Other Software

Data can be exported to a tab-delimited text files using `save` with the arguments *-double*-ascii. For example,

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

would save the data in `var1` in a tab-delimited text file. It is generally a good practice to only export one variable at a time using this method. Exporting more than one variable results in a poorly formatted file that may be hard to import into another program. The restriction to a single variable should not be seen as a severe limitation since `var1` can always be constructed from other variables (e.g. `var1=[var2 var3];`.

`tables`s can be exported using `writetable` to either delimited text or Excel. `writetable` facilitates the export of `tables`-specific features such as variable names and row names. Alternative methods to export data include `xlswrite`, `csvwrite` and `dlmwrite`.

14.6 Exercises

1. Use the import wizard to import *exercise3.xls*, which contains three columns of data, the date, the return on the S&P 500, and the return on XOM (ExxonMobil).
2. Use `xlsread` to read the file *exercise3.xls*. Load in the three series into a new variable named `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, `sumreturns` as the sum of `SP500` and `XOM`. Create another new variable, `outputdata` as a horizontal concatenation of `dates` and `sumreturns`.
7. Export the variable `outputdata` to a new .xls file using `xlswrite`. See the help available for `xlswrite`.

Chapter 15

Working with Heterogeneous Data

Traditionally arrays in MATLAB were homogeneous – all data in an array had to have the same type. For example, in the usual case, all values in a numeric array are stored as double precision floating point numbers. Over time MATLAB has added support for arrays with many different data types including integers, unsigned integers, single precision floating points and datatimes. Simultaneously MATLAB has offered support for fully heterogeneous data using cell arrays. Cell arrays support all MATLAB data types and each cell in an array can contain a different data type. While cell arrays are general purpose they are necessarily slow when storing homogeneous data. This arises due to the different way in which data is stored. In a traditional array, data is contiguous so that the second element in the array is adjacent to the first in memory, and the nth element is exactly $(n-1)$ steps away from the first where the step size only depends on the size of the data type used in the array (e.g., 8 bytes for a double precision floating point). Cell arrays, on the other hand, are not contiguous in memory and each cell actually points to a different location where the data is stored. As a result, accessing adjacent elements in a cell array requires additional lookups and in most cases, additional fetches from the main memory of the computer. This makes cell arrays too slow for any serious numeric computations.

Recently, `table`s have been introduced to provide more continuity between fast, homogeneous numeric arrays and slow, heterogeneous cell arrays. `table`s are collection of columns with additional metadata including variable names. With-in each column, the data type is homogeneous[1], while across columns data types may differ. This structure has a number of advantages over purely numeric arrays since data sets containing strings, dates, and numeric data can be aligned while preserving the ability to easily use data in high-performance numeric applications.

15.1 Creating `tables`

15.1.1 Importing

In most cases, `table`s will be created by importing data into MATLAB. Data is imported into a `table` using `readtable`, which can import delimited text files (e.g., comma or tab separated values), Excel or OpenDocument Spreadsheet files. If the data file to be imported is well formatted with variable names in the first row and data in rows below, `readtable` will import into a table and automatically read the variable names. Consider importing the following comma separated value file saved as animals.csv,

[1] With the exception of a column that is actually a cell array, which is permitted. For example, columns in `table`s that store strings will typically be cell arrays.

```
name,species,weight,height,birthday
Seabiscuit,horse,650000,1600,23/5/1933
Callie,dog,32000,550,3/1/2015
Grumpy Cat,cat,4000,240,4/4/2012
Jerry,mouse,19,25,10/2/1940
```

Importing this file will result in the table

```
>> readtable('animals.csv')
ans =
        name         species     weight     height      birthday
    ____________     _______     _______    ______     ___________

    'Seabiscuit'     'horse'     6.5e+05    1600       '23/5/1933'
    'Callie'         'dog'         32000     550       '3/1/2015'
    'Grumpy Cat'     'cat'          4000     240       '4/4/2012'
    'Jerry'          'mouse'          19      25       '10/2/1940'
```

which demonstrated the variable name importing. If a data file does not have obvious variable names, `readtable` will generate automatic variable names using the pattern `Var1`, `Var2`, ...

15.1.2 Direct creation

Tables can also be directly created using the `table` function. When used with existing variables, the variable name will be automatically used in the table.

```
>> special = [3.14, 2.72, 1.61]';
>> dates = {'31/12/1999', '15/9/2008', '23/06/2016'}';
>> names = {'Y2k bug', 'Lehman Collapse', 'Brexit'}';
>> table(special, dates, names)
ans =
    special        dates              names
    _______     ____________     _________________

    3.14        '31/12/1999'     'Y2k bug'
    2.72        '15/9/2008'      'Lehman Collapse'
    1.61        '23/06/2016'     'Brexit'
```

Alternatively, variable names can be set when calling `table` by using the optional argument `'VariableNames'` followed by a cell array containing the variable names.

```
>> table(special, dates, names, 'VariableNames', {'alpha','beta','gamma'})
ans =
    alpha        beta              gamma
    _____     ____________     _________________

    3.14      '31/12/1999'     'Y2k bug'
    2.72      '15/9/2008'      'Lehman Collapse'
    1.61      '23/06/2016'     'Brexit'
```

15.1.3 Conversion from other arrays

Standard MATLAB arrays, cell arrays and arrays of structures can all be converted to tables using `array2table`, `cell2table`, and `struct2table`. Using `array2table` to convert a 2-dimensional array to a table is similar to calling table after splitting the columns into separate variables where each column is named after the parent array and the column numbers.

```
>> x = reshape(1:12,4,3)
>> array2table(x)
ans =
   x1    x2    x3
   __    __    __

   1     5      9
   2     6     10
   3     7     11
   4     8     12
```

`cell2table` is similar except that an attempt to find a homogeneous datatype for each column is attempted. If a homogeneous datatype cannot be detected, then the column will be stored as a cell array. In general, it is not a good idea to store data with mixed types in a table, and in some cases converting cell arrays using tables can result in a loss of information due to the method used by MATLAB for detecting a homogeneous datatype. This issue is demonstrated in column 4 (x4) in the example below where the float is truncated and the third value is truncated to be the maximum value of a `uint8`.

```
>> x = [{'a','b','c'}',{1,2,3}',...
        {'a',2,datetime('12/31/1999')}', {uint8(1), 3.14, 2^31}'];
>> t = cell2table(x)
t =
   x1     x2         x3            x4
   ___    __    _____________    ___

   'a'    1     'a'                1
   'b'    2     [           2]     3
   'c'    3     [31-Dec-1999]    255
>> iscell(t.x1) % String, so still cell array
>> iscell(t.x2) % Homogeneous array now
0
>> iscell(t.x3) % Mixed, so still a cell array
1
>> t.x4 % Note loss of information
ans =
   1
   3
 255
```

Finally, `struct2table` creates a table from an structure containing arrays or an array of structures. Column names are derived from the fields of the structure.

```
>> clear y % Ensure y is clear before assigning fields
>> y.y1 = [1;2;3];
```

```
>> y.y2 = {'cat';'dog';'horse'}
>> t = struct2table(y);
>> y = [struct('y1',1,'y2','cat'),struct('y1',2,'y2','dog'),...
        struct('y1',3,'y2','horse')]'
y =
3x1 struct array with fields:
   y1
   y2
>> t2 = struct2table(y);
>> isequal(t,t2)
ans =
    1
```

15.2 Features of `tables`

In addition to providing a flexible container for storing heterogeneous data, tables have a number of additional features when compared with either numeric arrays or cell arrays. These features are designed to provide additional meaning to the data stored in a table and include both variable and row names.

Variable Names. Descriptions and Units

Three properties of a table are dedicated to storing information about variables. The most useful property is the capability to provide variable names (`VariableNames`). Traditional numeric arrays can only be accessed by column number, and so it was necessary to remember which column contained which data series. `tables` allow variables to have names which are displayed when viewing the data in a table and are used in other table-specific functions. Variable names must be value MATLAB variable names. Variable descriptions (`VariableDescriptions`) provide a matched set of strings which can contain any information required to describe a variable in a table. Variable units (`VariableUnits`) are strings which can be used to store the unit of a variable (e.g. Million USD, $, or Hours).

Row Names

Tables can have named rows. Row names are set using the `RowNames` property. Row names much satisfy two constraints: the row names must be strings and they must be unique.

Table Description and other Information

Three additional fields are available to store table metadata. Description can be used to store a general string description of a table. `DimensionNames` can be used to store the names of each dimension in a table. Finally, `UserData` can be used to store any other information about a table that one wishes to store that doesn't cleanly fit into one of the other categories.

15.2.1 Reading or Setting Properties

Properties can either be set when creating a table using optional arguments of the form `'`*PropertyName*`'`, *PropertyValue* or using the field `.Properties` of a table. All properties can be read using *table* `.Properties`

or specific properties can be read using *table* `.Properties.`*PropertyName*.

```
>> t = table(special, dates, names)
>> t.Properties
ans =
             Description: ''
    VariableDescriptions: {}
           VariableUnits: {}
          DimensionNames: {'Row'  'Variable'}
                UserData: []
                RowNames: {}
           VariableNames: {'special'  'dates'  'names'}
>> t.Properties.Description = 'Some example data';
>> t.Properties.VariableUnits = {'number','date','string'};
>> t.Properties
             Description: 'Some example data'
    VariableDescriptions: {}
           VariableUnits: {'number'  'date'  'string'}
          DimensionNames: {'Row'  'Variable'}
                UserData: []
                RowNames: {}
           VariableNames: {'special'  'dates'  'names'}
```

15.3 Column data types

tables are designed to efficiently handle data that with heterogeneous types across variables but homogeneous within a single variable. Columns can have different data types which allows for efficient storage of large dataset.

15.3.1 Numeric

Numeric is a common format for storing data values. The default numeric columns type will use double precision floating point numbers. Each value requires 8 bytes of storage, and so when data have a more limited range, for example, integer values less than some value, other numeric types can be used to reduce the amount of memory required to store data. For example, 8-bit unsigned integers can hold values between 0 and 255, inclusive, and require only 1 byte of storage per value.

15.3.2 Strings

Strings are usually stored in a table using cell arrays. This allows for strings to have different lengths and for simple manipulations of string values.

15.3.3 `categoricals`

Categoricals are used when a string variable only takes a relatively small number of values. For example, country names in a large dataset of web visitor data can only take around 200 values. Categorical variables efficiently encode these strings to integers while preserving the ease of interpretation of the original

country names. A cell array of strings can be converted to a categorical using the command `categorical(` *cellarray* `)`. In this example, a large list of full country names requires around 1.1 MiB of storage which the categorical version of the same data requires about 1% as much memory.

```
>> names = {'Afghanistan','Albania','Algeria','Andorra','Angola'}'
>> countries = names(randi(5,10000,1))
>> countries_cat = categorical(countries)
>> whos countries*

 Name                    Size              Bytes  Class          Attributes
 countries           10000x1             1272560  cell
 countries_cat       10000x1               10742  categorical
```

Data in an existing table can be converted to a categorical by assigning the output of a call to `categorical` to the original variable.

```
>> t = table(countries)
>> t.countries = categorical(t.countries)
```

15.3.4 datetimes

`datetime`s represent another optimized format. Traditionally MATLAB used a proprietary serial date format that expresses a date as the number of days since January 1, 0000 12:00:00 AM (which was 1.0). These dates are difficult to work with since it is not easy to interpret 730485 as December 31, 1999. `datetime`s offer an alternative storage format that is visibly represented as human-readable dates while using an optimized format for the storage of dates and times. Datetimes also bring support for timezone information, which is missing in the MATLAB serial date format. `datetime`s are created by calling `datetime` on a cell array of string dates and times.

```
>> dates = {'23/5/1933 12:00:00 AM','3/1/2015 6:30:15 PM',...
            '4/4/2012 6:18:18 PM','10/2/1940 12:21:12 AM'}';
>> datetimes = datetime(dates)
>> whos date*
Name             Size              Bytes  Class       Attributes
 dates           4x1                 608  cell
 datetimes       4x1                 169  datetime
```

Like `categorical`s, `datetime`s can be added to an existing table by assigning the output of `datetime`.

```
>> animals = readtable('animals.csv')
animals =
      name         species     weight     height      birthday
   ____________    _______    _______    ______     ___________

   'Seabiscuit'    'horse'    6.5e+05    1600       '23/5/1933'
   'Callie'        'dog'        32000     550       '3/1/2015'
   'Grumpy Cat'    'cat'         4000     240       '4/4/2012'
   'Jerry'         'mouse'         19      25       '10/2/1940'

>> animals.birthday = datetime(animals.birthday)
animals =
```

```
     name          species     weight     height     birthday
    ____________   _______    _______    ______    ___________

    'Seabiscuit'   'horse'    6.5e+05    1600      23-May-1933
    'Callie'       'dog'        32000     550      03-Jan-2015
    'Grumpy Cat'   'cat'         4000     240      04-Apr-2012
    'Jerry'        'mouse'         19      25      10-Feb-1940
```

The difference between dates stored as strings and dates stored as `datetime` is in the representation of the date and the lack of quotation marks.

15.4 Selection

15.4.1 Selecting Subtables

Parentheses are the simplest method to access a table and selections made with parentheses will return a table even if a single column is selected. Two inputs are required, one for rows and one for columns. Selecting rows is identical to selecting rows of a matrix and any of the usual methods, scalar, numeric list of indices, slice (using : notation), or logical array, can be used. Selecting columns supports the same 4 selection types in addition to selection by variable name. When selecting using a single variable name, the name alone can be used. When selecting multiple columns the variable names should be entered using a cell array using {}.

```
>> t = table([1,2,3]',[10,9,8]',[-1,0,1]',['a','b','c']',...
             datetime({'12/31/1999','1/31/2000','2/29/2000'}'),...
             'VariableNames',{'Alpha','Beta','Gamma','Delta','Epsilon'});
>> t(:,1:3) % Cols 1, 2 and 3
>> t(:,3) % Col 3 only
>> t(:,logical([1,1,0,1])) % Select cols 1, 2 and 4
```

It is also possibly to use the names in the second position.

```
>> t(:,{'Alpha','Beta','Delta'}) % Select using variable names
>> t(:,'Gamma') % Select using a single variable name
```

Finally, order matters and so these two commands are not identical.

```
>> t([2,1],{'Alpha', 'Beta'}) % Name order is respected
>> t([2,1],{'Beta', 'Alpha'}) % Name order is respected
```

15.4.2 Selecting arrays

Braces (or curly braces, {}) can be used to extract values from a table. The important difference is that the result is an array *if the data are homogeneous.* Arrays do not support heterogeneous data and so using braces to select multiple columns with different types will produce an error. When using braces, two arguments are required. Aside from the requirement for a homogeneous input and the return of an array, using braces is virtually identical to using parentheses.

```
>> t{:,1:3} % Cols 1, 2 and 3
>> t{:,3} % Col 3 only
>> t{:,logical([1,1,0,1])} % Select cols 1, 2 and 4
```

Similarly variable names can be used.

```
>> t{:,{'Alpha','Gamma'}} % Select using variable names
>> t{:,'Delta'} % Select using a single variable name
```

Finally note that using braces on a mixed table produces another an error.

```
>> t{:,{'Alpha','Delta','Epsilon'}} % Error
```

15.4.3 Selecting single columns

Dot notation allows a single column to be extracted. Generally, the syntax used will be *table*.*variable* as in

```
>> t.Alpha
```

Dot notation can additionally be used to select columns based on numeric position using the syntax table.(#) where # is a number. This dot selection is identical to the previous one.

```
>> t.(1)
```

Note that when using dot selection the column selected is just a standard array and not a table. Dot selection can be chained with other selectors to subset the column selected with the dot. For example,

```
>> t.Alpha(2:3)
```

will select elements 2 and 3 from Alpha.

15.5 Table-specific features

15.5.1 Converting to other data structures

Tables can be exported to other MATLAB data structures including homogeneous arrays, cell arrays and structures containing arrays using `table2array`, `table2cell`, and `table2struct`, respectively. `table2array` can only export tables that are homogeneous (e.g. all numbers).

```
>> table2array(t(:,1:3)) % Only the numbers
 ans =
     1    10    -1
     2     9     0
     3     8     1
>> table2cell(t)
 ans =
    [1]    [10]    [-1]    'a'    [31-Dec-1999]
    [2]    [ 9]    [ 0]    'b'    [31-Jan-2000]
    [3]    [ 8]    [ 1]    'c'    [29-Feb-2000]
>> table2struct(t)
 ans =
 3x1 struct array with fields:
    Alpha
    Beta
    Gamma
    Delta
    Epsilon
```

15.5.2 Saving and Exporting tables

`table`s can be saved to MATLAB data files using the same syntax as any other variable, `save` *matfilename tablename*. `table`s can be exported to either delimited text files or excel files. The file extension determines the file format written. By default, text files will be comma separated, although this can be changed using an optional argument. Excel files can be exported in either old or new Excel file formats (`.xls` for old, `.xlsx` or `.xlsm` for new).

```
>> writetable(t, 'out.csv')   % Commma separated
>> writetable(t, 'out.xlsx')  % Excel
>> writetable(t, 'out.txt',...  % Tab delimited with variable names
              'WriteVariableNames',true,...
              'Delimiter','\t')
```

Options can be passed using additional arguments. The most useful are `'WriteVariableNames'` and `'WriteRowNames'` which determine whether these values will be exported.

15.5.3 Merging tables

Tables support a range of SQL-like operations that allow tables to be merged or joined and which allow for row-based set operations such as intersections or differences of two tables. `join` can be used to join two tables on one or more variables using a SQL-like left join. `join` requires that the table being joined to the existing table has all of the keys in the existing table. The related, and more useful, `innerjoin` and `outerjoin` perform inner (retain only rows in both) or outer (retain if a row in either) joins. Neither `innerjoin` nor `outerjoin` require all values of a key to be available in both tables.

```
>> t1 = table({'dog','cat','horse'}',[1,2,3]',...
              'VariableNames',{'animal','id'});
>> t2 = table({'bird','dog','cat','dolphin'}',...
              [102.2,43.1,13.9,73.3]',...
              'VariableNames',{'animal','weight'});
>> innerjoin(t1,t2,'Keys','animal')
ans =
   animal    id    weight
   ______    __    ______
   'cat'     2     13.9
   'dog'     1     43.1
>> outerjoin(t1,t2,'Keys','animal')
ans =
   animal_t1    id     animal_t2    weight
   _________    ___    _________    ______

   ''           NaN    'bird'       102.2
   'cat'          2    'cat'         13.9
   'dog'          1    'dog'         43.1
   ''           NaN    'dolphin'     73.3
   'horse'        3    ''             NaN
```

A wide range of set operations are also available for finding sub- or super-sets of tables. `intersect` find the intersection of two tables and returns the common rows. `setdiff` returns the rows in one table that

are not in the other table; similarly setxor returns the rows that are in either table except those that are available in both. union returns set of all unique rows in two tables. unique returns the unique rows in a single table while ismember returns a true/false value indicating if a row is in another table.

15.5.4 Grouping

tables support computing statistics or applying other functions across groups. For example, if a data set contains data for individual income and hours worked across states, it is interesting to see how income varies with hours worked. This requires computing the average income and the average hours worked for each state. varfun makes this type of calculation simple since it support computing a function, variable-by-variable, and allow automatic grouping on one or more variables.

```
>> states = {'NY','FL','CA','TX'}';
>> index = randi(4,100000,1);
>> income = 30000 + 2500 * index + 10000 * randn(100000,1);
>> hours = 35 + 2.5 * index + 8 * randn(100000,1);
>> state = states(index);
>> t = table(state, income, hours)
>> varfun(@mean,t,'InputVariables',{'hours','income'},...
   'GroupingVariables', 'state')
ans =
   state    GroupCount    mean_hours    mean_income
   _____    __________    __________    ___________

   'CA'     24657         42.547        37487
   'FL'     25001          40.09        35103
   'NY'     25200         37.453        32449
   'TX'     25142         44.989        40011
```

Other available table-specific function as rowfun, which will compute function across variables in a particular row, findgroups which will generate a set of group indices for a table, and splitapply which allows more generality than varfun for computing grouped statistics.

15.5.5 Table-specific Functions

A number of c-specific functions are available to simplify working with tables. summary can be used to compute a basic summary of the variables in a table.

```
>> summary(t)
   Alpha: 3x1 double
       Values:
           min       1
           median    2
           max       3
   Beta: 3x1 double
       Values:
           min        8
           median     9
           max       10
   Gamma: 3x1 double
```

```
        Values:
            min       -1
            median     0
            max        1
    Delta: 3x1 char
    Epsilon: 3x1 datetime
        Values:
            min       31-Dec-1999
            median    31-Jan-2000
            max       29-Feb-2000
```

`istable` returns `true` (1) if a variable is a `table`. `height` and `width` return the number of rows and columns, respectively, in a `table`. These are mostly redundant since `size` can be used with these `table`s as well.

Chapter 16

Probability and Statistics Functions

The statistics toolbox contains an extensive range of statistical function.

16.1 Distributions: `*cdf`, `*pdf`, `*rnd`, `*inv`

The most valuable code in the statistics toolbox are the CDFs, PDFs, random number generators and inverse CDFs. All distributions commonly encountered in econometrics 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-`)

16.2 Selected Functions

16.2.1 `quantile`

`quantile` returns the empirical quantiles of a vector. It requires two inputs. The first is a vector or matrix (T by K) and the second is an M-element vector of quantiles to compute. When the input is a vector, the

output will have the same dimensions as the list of quantiles used (either 1 by M or M by 1). When the input is a matrix, a M by K matrix is returned where each column of the computed quantiles corresponds to a column of the input matrix. `quantile` is simple and can easily be replaced using `sort`, `length` and `floor` or `ceil`.

```
>> x = randn(100000,1);
>> quantile(x,[.025 .05 .5 .95 .975])
   -1.9567   -1.6430    0.0010    1.6375    1.9488
```

16.2.2 prctile

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

16.2.3 regress

`regress` performs basic regression and returns key regression statistics. The Statistic Toolbox implementation is not robust to many empirical realities in economic or financial data (such as heteroskedasticity) and so is of limited use.

16.3 The MFE Toolbox

The MFE Toolbox contains a set of functions addressing many common problems in financial econometrics. It is available on the course website. Note that the MFE Toolbox has superceeded the UCSD_garch toolbox.

16.4 Exercises

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

Chapter 17

Custom Functions

Custom functions can be written to perform repeated tasks or to use as the objective of an optimization routine. All functions must begin with the line of the form

$$\texttt{function}\ [out1, out2, \ldots] = functionname(in1, in2, \ldots)$$

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.

To begin, consider this simple function `func1`

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

This function, which is not 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 relative to 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 unless they are explicitly returned.[3]

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

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

[1] It has no comments, has superfluous commands and is trivial in nature. The function should only contain `y = x +1;` and a comment that describes the function's purpose.

[2] 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.

[3] MATLAB supports global variables using the keyword `global`. Global variables can be seen both in the standard workspace and inside functions. In general, global variables should be avoided. Use cases of `global` variables are discussed in Chapter 21.

```
x =
    1
>> y
y =
    1
>> z
z = 2
```

Thus, despite the function using variables named 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 i not necessary to call the function with two outputs. For example, consider the following use of this function.

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

The final call shows the use of ~ to suppress leading outputs of functions when they are not used.

17.1 Function-specific functions

There are a number of advanced function specific variables available to determine 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 that allow for early termination of the function (`return`). This course can be completed without using any of these, although they are useful especially when producing code for other users.

17.1.1 nargin

`nargin` is available inside functions to determine the number of inputs provided in the function call. This allows for default values to be used for trailing inputs. Note that an empty input (`[]`) is still an input, and so it may be necessary to check whether an input is empty using `isempty`.

17.1.2 nargout

nargout is available inside functions to determine the number of outputs requested. It is useful to avoid calculating some outputs when if the number of outputs requested is smaller than the maximum number of outputs supported by the function.

17.1.3 varargin

varargin can be used as the last input in a function declaration to capture a variable number of inputs. Consider the following code.

```
function varargin_demo(varargin)
% Iterates across all inputs and displays the contents
for i=1:length(varargin)
    disp(varargin{i})
end
```

This function can accept any number of inputs (including 0) and will iterate across the inputs and display their contents. Note that varargin is a cell array (see chapter 13).

17.1.4 varargout

varargout is similar to varargin, only that it allows for a variable number of outputs. varargout is rarely encountered, but can be used to allow producing as many outputs as the number of inputs when varargin is used.

```
function varargout = varargout_demo(varargin)
% Iterates across all inputs and displays the contents
varargout = cell(size(varargin));
for i=1:length(varargin)
    varargout{i} = varargin{i};
end
```

The following code demonstrated this function using different numbers of inputs.

```
>> [a,b,c] = varargout_demo(1,2,3)
a =
     1
b =
     2
c =
     3

>> [a,b,c,d,e,f] = varargout_demo(pi,exp(1),sqrt(2),10i,-1,inf)
a =
    3.1416
b =
    2.7183
c =
    1.4142
d =
```

```
        0 +10.0000i
e =
    -1
f =
   Inf
```

17.1.5 return

`return` can be used to exit a function before all code has been executed, and is usually used inside an `if` statement.

17.2 Comments

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

```
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;
```

`help` func returns

```
>> 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 optional, they should be included both to improve readability of the function and to assist others if the function is shared.

17.3 Debugging

Since the data modified in the function is not available when the function is run, debugging can be difficult. There are four 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 (or alternatively, use `disp`).
- Use `keyboard` and `return` to interrupt the function to inspect the values.

- Use the editor window to set breakpoints.

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 although clumsy – often the output window is quickly filled with numbers and so locating the problematic code becomes difficult. The third options is more advanced. Adding `keyboard` to a function interrupts the function at the location of `keyboard` and returns control to the command window. 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 `xpy` and `xmy` in addition to the original two. When a function has been debugged, either comment out or remove the `keyboard` commands.

The final option is to set breakpoints in the MATLAB editor. Breakpoints can be added either in the editor or using the command `dbstop in` *file* `at` *lineNumber*. In practice, it is usually simpler to use the editor to set the breakpoint. When using breakpoints, the function is stopped whenever a breakpoint is encountered. This allows for values inside the function to be inspected. In addition, various methods of "stepping" are available when using formal debugging:

- Step - Proceed to the next line
- Step In - Proceed to the next line, and enter any sub-function (also in debugging mode)
- Step Out - Proceed out of the current function to the next line in the main program
- Continue - Resume normal execution, stopping at the end of the main function or when another breakpoint is encountered

Figure 17.1 show how a break point is set in the MATLAB editor.

17.4 Exercises

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

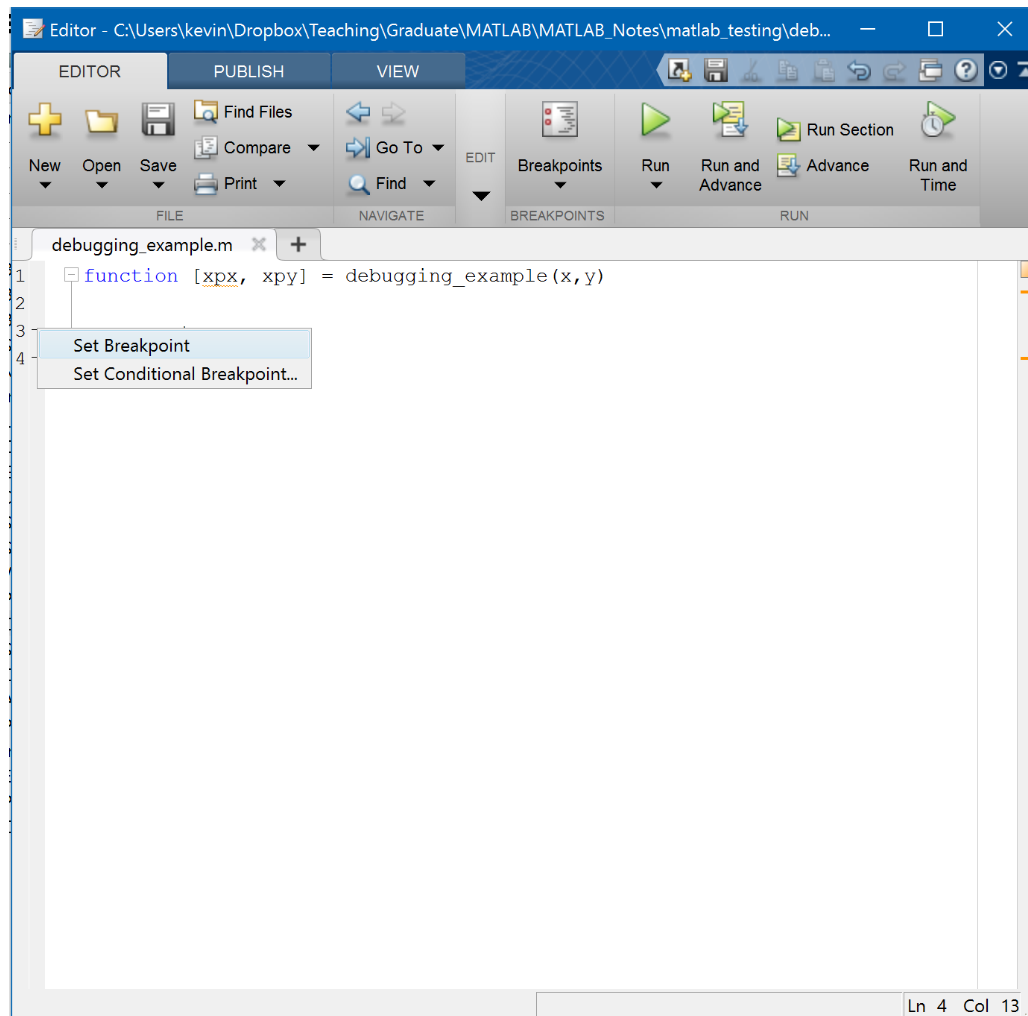

Figure 17.1: Break points can be set in the editor for debugging files by right-clicking in the left column of the window.

```
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 18

Simulation and Random Number Generation

18.1 Core Random Number Generators

All pseudo-random numbers are generated by four core random number generators,

- `rand`: Uniform pseudo-random number generator on the interval (0,1)
- `randn`: Standard Normal pseudo-random number generator
- `randg`: Standard Gamma pseudo-random number generator
- `randi`: Uniform integer pseudo-random number generator

The distribution of pseudo-random number generated will determine which of these are used. For example, Weibull pseudo-random numbers use `rand`. Normal pseudo-random numbers obviously call `randn`. Creating Students-t pseudo-random numbers requires calls to both `randn` and `randg`, and χ^2 uses only `randg`.

18.2 Replicating Simulation Data

The all of the pseudo-random number generators share a common state (by default). The state is a large vector which determines the next pseudo-random number. This state allows a sequence of random numbers to be repeated by first saving the state and then restoring it. The state is saved using `state = rng()`, where `state` is a structure containing information about the type of generator in use, the seed and the actual state vector. The state can be restored using `rng(state)`.

```
>> state = rng()
state =
    Type: 'twister'
    Seed: 0
   State: [625x1 uint32]
>> randn
ans =
          0.5376671395461
>>  randn
ans =
```

```
        1.83388501459509
>> rng(state)
>> randn
ans =
         0.5376671395461
>>  randn
ans =

        1.83388501459509
```

These two sequences are the same since the state was restored to its previous value.

Warning: The state is restored every time MATLAB is initialized. As a result, all of the random number generators will produce the same sequence when starting from a fresh MATLAB session. This default state can be restored using `rng(0)`.

18.3 Considerations when Running Simulations on Multiple Computers

The state of all random number generators is reset each time MATLAB is opened. Thus, two programs drawing pseudo-random numbers on different computers, or in two instance on the same computer, will be identical. Two avoid this problem the state needs to be initialized to a "random" value. This can be accomplished in recent versions of MATLAB by

```
rng('shuffle')
```

which uses the current time to act as a "random" input to generate the state. This will ensure that simulations running in different MATLAB sessions will not use the same sequence of random numbers.

Warning: Do not *over-initialize* the pseudo-random number generators. The generators should be initialized once per session and then allowed to produce the sequence beginning with the state set by `rng('shuffle')`. Repeatedly re-initializing the pseudo-random number generators will produce a sequence that is much less random than the generator was designed to provide.

18.4 Advanced Random Number Generator

MATLAB has substantially overhauled their random number generators over the past decade. Fine-grained control of the random number generator is available using `RandStream`, which is a class that can be used to initialize a random stream.[1] The random stream, in-turn, does the actual generation of the pseudo-random numbers. Recent versions of MATLAB support 6 core random number generators, each with different properties. The default algorithm is known as `mt19937ar`, or the Mersenne Twister. It is a widely used algorithm with good properties. However, other choices may work better when using MATLAB is parallel.

[1] MATLAB supports object-oriented programming (OOP). `RandStream` is an example of a class, one of the core components of OOP. Understanding OOP is not necessary to be a proficient MATLAB programmer where the dominant programming paradigm is imperative programming.

Chapter 19

Optimization

The optimization toolbox contains a number of routines to the find extremum of a user-supplied objective function. Most of these implement a form of the Newton-Raphson algorithm which uses the gradient to find the *minimum* of a function.[1]

A custom function that returns the function value at a set of parameters – for example a log-likelihood or a GMM quadratic form – is required to use one of the optimizers. All optimization targets must have the parameters as the first argument. First, consider finding the minimum of x^2. A function which allows the optimizer to work correctly has 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 that are not parameters (such as data or hyper-parameters).

```
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 useful when optimization targets require at least two inputs: parameters and data. Once an optimization target has been specified, the next step is to use one of the optimizers find the minimum.

[1] MATLAB's optimization routines *only* find minima. However, if f is a function to be maximized, $-f$ is a function with the minimum at located the same point as the maximum of f.

19.1 Unconstrained Derivative-based Optimization

fminunc performs gradient-based unconstrained minimization. Derivatives can optionally be provided by the user and when not supplied are numerically approximated. 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 19.5), 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_target4.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

```
>> [p,fval,exitflag]=fminunc('optim_target4',p0,options,hyper)
                                                         First-order
 Iteration  Func-count       f(x)        Step-size       optimality
     0           3             3                             3
     1           6             0        0.333333       1.49e-008
Optimization terminated: relative infinity-norm of gradient less than options.TolFun.
p =
   1   1
fval =
   0
exitflag =
   1
```

fminunc has minimized this function and returns the optimum value of 0 at $x = (1, 1)$. exitflag has the value 1, indicating the optimization was successful. Values less than or equal to 0 indicate the optimization to not converge successfully.

19.2 Unconstrained Derivative-free Optimization

fminsearch also performs unconstrained optimization but uses a derivative free method called a simplex search. fminsearch uses an "amoeba" to crawl around in the parameter space and 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 19.5), 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
     ...
     ...
    ...
    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 in general should not be used if fminunc works satisfactorily. However, for certain problems, such as when the objective is not continuously differentiable, fminsearch may be the only option.

19.3 Bounded scalar optimization

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 19.5), and var_1, var_2, ... are (optional) variables

containing data or other constant values.

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
```

19.4 Constrained Derivative-based Optimization

`fmincon` performs constrained optimizations using linear and/or nonlinear constraints. The constraints can take the form of either equality or inequality expressions (or both). `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^{EQ} is a $P \times K$ matrix, b^{EQ} is a P by 1 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 $\mathbb{R}^M$ where M is the

number of nonlinear inequality constraints, d is a $M \times 1$ vector, $C^{NEQ}(x)$ is a function from $\mathbb{R}^K$ to $\mathbb{R}^N$ and d^{NEQ} if an $N \times 1$ vector where N is the number of nonlinear equality constraints. Note that any $\geq$ constraint 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 for inequality and equality constraints, respectively, and b and b^{EQ} are conformable vectors. *LB* and *UB* are vectors with the same size as p_0 that contain upper and lower bounds, respectively.[2] *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 19.5), and var_1, var_2, . . . are (optional) variables containing 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 that is must also be that case that $x_1 \geq 0$ and $x_2 \geq 0$). 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 as

```
>> 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',p0,A,b,[],[],[],[],[],options,lambda)
                                max                     Directional   First-order
 Iter F-count          f(x)   constraint    Step-size   derivative    optimality Procedure
```

[2] *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

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

which are expressions for $-p \leq 1$ (which is equivalent to $p \geq -1$) and $p \leq 1$.

```
     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

$$\min_{x_1,x_2} p_1 x_1 + p_2 x_2 \text{ subject to } U(x_1, x_2) \geq \bar{U}$$

Begin by 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, it is necessary to 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 the arguments are not required. The solution to this problem can be found 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, require x1>=0 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
    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
    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
    6     22     0.999936    2.535e-011            1    3.05e-007    1.31e-008
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 problems where the answers can be analytically verified. In many cases it is impossible to verify that the global optimum has been found if there are local minima. The standard practice for addressing the possibility of local minima is to start the optimization from different starting values and then to use the lowest `fval`. If the optimizer is working well on the specified problem, many of the starting values should produce similar parameter estimates and `fvals`.

Note: Many aspects of constrained optimization (and optimization in general) are more black magic than science. Worse, most techniques are problem class specific and so general rules are hard to derive.

Practice is the only method to become proficient at function minimization.

19.5 Optimization Options

`optimset` sets optimization options and has two distinct forms. The initial call to `optimset` should always be of the form `options = optimset('fmin`*type*`')` which will return the default options for the selected optimizer. Once the options structure has been initialized, individual options can be changed 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`.

19.6 Other Optimization Routines

The 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 non-negativity 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 20

Accessing the File System

MATLAB uses standard DOS (or Unix, depending on the platform) file system commands to change working directories. For instance, to change directory, type

```
cd c:\MyDirectory
```

on Windows or

```
cd ~/MyDirectory/
```

on Unix.

Other standard file navigation commands, such as `dir` and `mkdir` are also available. 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).

20.1 Addressing the File System Programmatically

The file system can be accessed in MATLAB code. One common application of programmatic access to the file system is to perform some action on every file in a particular directory, which can be done by looping over the output of `dir`.

```
% Create some files
for i=1:3;
    fid = fopen(['file_' num2str(i) '.demotxt'],'wt');
    fprintf(fid,'Nothing to see');
    fclose(fid);
end
```

The example code below get a list of files that have the extension demotxt and then loops across the files, first displaying the file name and then using `type` to print the contents of the file. This method is very useful for processing multiple data files.

```
>> d = dir('*.demotxt')
d =
3x1 struct array with fields:
    name
```

```
    date
    bytes
    isdir
    datenum
>> for i=1:length(d);
>> disp(d(i).name)
>> type(d(i).name)
>> end
file_1.demotxt
Nothing to see

file_2.demotxt
Nothing to see

file_3.demotxt
Nothing to see
```

MATLAB contains a full set of platform-independent commands to access the file system. The platform-independence is derived from the availability of utility functions such as `filesep` which returns the platform-specific file seperator and `copyfile` which operates like `copy` on Windows and `cp` on Unix platforms.

cd

`cd` can be used to change the current directory. Both relative and absolute paths are supported. `cd` can be used both with a space, as in `cd c:\temp` or as a function, as in `cd('c:\temp')`. `cd` can also be used with string variables containing the path, in which case the function version must be used.

```
% Absolute
cd('c:\temp\')
% Relative, up one then down in temp
cd('..\temp\')
% Relative, up two levels
cd('..\..')
% String input
targetDir = 'c:\temp';
cd(targetDir)
% Non-funciton version
cd c:\temp
```

dir

`dir` can be used to list the contents of a directory. It can be used without any arguments, with a wildcard argument, or with a full path. When used without an output variable, the listing is printed to the screen. Using an output returns a structure containing the contents of the directory. Like `cd`, `dir` can be used either with a space or as a function.

```
% Wildcard
files = dir('*.mat')
% Path
files = dir('c:\temp')
```

mkdir and rmdir

mkdir and rmdir can be used to create and remove directories, respectively. Like cd, both commands can be used with a space or as a function, although only the function version can be used with string inputs.

delete

delete can be used to delete files. It can be used with a single filename, or with a wildcard expression to delete all matches. Like cd, delete can be used either with a space or as a function.

copyfile and movefile

copyfile and movefile can be used to copy and move files, respectively. Both command require two inputs, the *source* and the *destination*. The source can include wildcards in which case the destination must be a directory.

fullfile

fullfile is a useful utility for building full paths including the filename and extensions.

```
% Wildcard
>> fileLoc = fullfile('c:','temp','data.mat')
fileLoc =
c:\temp\data.mat
```

fileparts

fileparts can be used to split a file name into:

- Path (excluding filename)
- File name
- Extension

filesep

filesep can be used to get the platform-specific platform separator. It is useful for manually building full paths, although using fullfile is often easier.

20.2 Running Other Programs

MATLAB can launch other programs using system (or dos on Windows). The basic structure is system(' *command_to_run* ') (which can also be executed using the syntax !*command_to_run*). An optional output can be used to capture any outpur from the command that would have appeared in the DOS window or terminal.

20.3 The MATLAB Path

While this section sounds like a Buddhist rite of passage, the path contains an important set of locations. The path determines where MATLAB searches for files when running programs. All of the MATLAB 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 the most important directory to least, with the present working directory (what `pwd` returns in the command window) silently atop the list. The path determines 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 the order the files appear. The highest ranked filed will then be executed. Because of this, it is crucial that existing function names are avoided when writing m-files. `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. This is useful for detecting duplicate file names.

New directories can be appended to the path using `addpath` or File>Set Path. . .. The GUI tool 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.

`startup.m`

When using MATLAB in a shared environment, the MATLAB path will generally be read-only – hence it cannot be permanently changed. The “work-around” for this issue is to create a file named `startup.m` in the directory where MATLAB initially opens. `startup.m` is a special file that is automatically executed when MATLAB is started and can contain lines with the `addpath` command.

```
% Example startup.m
addpath('c:\temp');
addpath('c:\temp\mytoolbox');
% Change the directory to where I keep my work
cd('c:\users\kevin\Dropbox')
```

20.4 Exercises

1. Use the command window to create a new directory, *chapter20* (`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 21

Performance and Code Optimization

The final step in writing code is to optimize the performance of the code, if needed. Code optimization can produce large improvements in speed over a naïve (but correct) implementation. In some cases the improvements can be 100 times or greater and the largest gains come from removing superfluous memory allocations.

Warning: Be careful not to over-optimize code. Over-optimizing code can produce code that is unreadable and difficult to debug. A good practice is to use a simple, possibly slow, implementation as a starting point. The optimized version can be built from the known-good code and the output from the optimized code can be compared to the known-correct version.

21.1 Just-in-time Compilation

Recent versions of MATLAB (R2015b or later) generate Low Level Virtual Machine intermediate results, which is them compiled to machine code using LLVM (`http://llvm.org/`). This a common strategy used by a number of projects Clang (C/C++), Apple's Swift, Julia and Python's Numba. This produces code that runs quickly and often has performance indistinguishable from code written in C or Fortran and compiled using an optimizing compiler (e.g., GCC, MSOC or ICC/IFort). This feature is known as MATLAB Execution Engine. While traditional code optimizations are still useful, the performance improvements of these optimizations are decidedly lower when using versions of MATLAB that include the Execution Engine.

21.2 Suppress Printing to Screen Using `;`

Displaying results to the screen is a relatively slow action and excess printing to screen can (substantially) reduce performance. Use `;` to suppress output.

21.3 Pre-allocate Data Arrays

Pre-allocating data and pre-generating random numbers in large blocks is the most basic optimization. While recent MATLAB improvements have reduced the performance impact of not pre-allocating it still allows some expensive memory allocation to be avoided in the core of the program. Similarly, pre-generating

random numbers allows function overhead to be avoided. To see the effects of pre-allocating, consider the following code:

```
clear y
y = 0;
tic;
for i=2:100000;
y(i) = y(i-1) + randn;
end;
toc
```

```
Elapsed time is 0.0128 seconds.
```

```
clear y
y = zeros(100000,1);
tic;
for i=2:100000;
y(i) = y(i-1) + randn;
end;
toc
```

```
Elapsed time is 0.0074 seconds.
```

The second version with a pre-allocated `y` is about 2 times faster. To see the effects of pre-generating random numbers, consider the following code:

```
M = 1000000 ;
y = zeros(M,1);
tic;
for i=2:M;
y(i) = y(i-1) + randn;
end;
toc
```

```
Elapsed time is 0.0410 seconds.
```

```
y = zeros(M,1);
e=randn(M,1);
tic;for i=2:M;
y(i) = y(i-1) + e(i);
end;
toc
```

```
Elapsed time is 0.0205 seconds.
```

Pre-allocating random numbers to avoid many tiny function calls produces a doubling in performance.

21.4 Avoid Operations that Require Allocating New Memory

One of the key advantages to using an environment such as MATLAB is that end-users are not required to manage memory. This abstraction comes at the cost of performance and memory allocation is slow. For an example of the penalty, consider the two implementations of the following recursion

$$y_t = .1 + .5y_{t-1} - .2y_{t-2} + 0.8\epsilon_{t-1} + \epsilon_t$$

```
epsilon = randn(10000,1);
y = zeros(10000,1);
parameters = [.1 .5 -.2 .8 1];
tic
for t=3:10000
    y(t) = parameters * [1 y(t-1) y(t-1) epsilon(t-1) epsilon(t)]';
end
toc
```

```
Elapsed time is 0.0127 seconds.
```

```
tic
for t=3:10000
    y(t) = parameters(1);
    for i=1:2
      y(t) = y(t) + parameters(i+1)*y(t-i);
    end
    for i=0:1
      y(t) = y(t) + parameters(5-i)*epsilon(t-i);
    end
end
toc
```

```
Elapsed time is 0.0014 seconds.
```

The second implementation is about 10 times as fast because it avoids allocating memory inside the loop. In the first implementation, `[1 y(t-1) y(t-1) epsilon(t-1) epsilon(t)]` requires a new, empty 5 element vector to be allocated in memory and then for the 5 elements to be copied into this vector *every iteration*. The second implementation uses more loops but avoids costly memory allocation.

21.5 Use Vector and Matrix Operations

Vector and matrix operations are highly optimized and writing code in matrix-vector notation is faster than looping. Consider the problem of computing

$$\mathbf{X}'\mathbf{X} = \sum_{n=1}^{N} x_n x_n'$$

which is the inner product of a matrix.

```
N = 10000;
X = randn(N,10);
op = zeros(10);

tic
for n=1:N
    op = op + X(n,:)'*X(n,:);
```

```
end
toc
```

```
Elapsed time is 0.0214 seconds.
```

```
tic
op_fast = X'*X;
toc
```

```
Elapsed time is 2.2687e-04 seconds.
```

Here the performance difference is very large.

21.6 Vectorize Code

Many operations in MATLAB are amenable to vectorization, not just matrix algebra. For example, logical operators can be used on entire vectors or matrices, and the result can then be used to select the relevant data points. Consider the following example:

```
x = randn(10000,1);
tic
y = x(x<0); % Vectorized select
toc
```

```
Elapsed time is 0.009911 seconds.
```

```
tic
y = zeros(10000,1);
count = 0;
for i=1:10000;
    if x(i)<0;
        count = count + 1;
        y(count) = x(i);
     end
end
y = y(1:count);
toc
```

```
Elapsed time is 0.021515 seconds.
```

In this simple example, the vectorized code requires about half the time as the for-loop code.

21.7 Use Pre-computed Values in Optimization Targets

Many optimization targets depend on parameters, data and functions of data. In most cases, the functions of the data do not depend on the parameter values and so they can be pre-computed. For example, if the optimization target is a likelihood target that depends on the square of the data (e.g. the Gaussian log-likelihood), pre-computing the square of the data and passing it as one of the optional arguments avoids needlessly re-computing these values every time the objective function is called.

21.8 Use M-Lint

The editor provides M-Lint guidance when available. This advice is almost always correct and should only be ignored if known to be wrong.

21.9 `timeit`

The function `timeit` can be used to quickly time and compare alternative versions of a function. Consider these two implementations of a dot product,

```
function dp = dot_1(x,y)

dp = x'*y;
```

and

```
function dp = dot_2(x,y)

dp = 0;
for i=1:length(x)
   dp = dp + x(i) * y(i);
end
```

The execution time can be examined using `timeit` and an anonymous function,

```
>> x = randn(1000000, 1);
>> y = randn(1000000, 1);
>> timeit(@() dot_1(x,y))
ans =
   0.0011
>> timeit(@() dot_2(x,y))
ans =
   0.0119
```

which shows that the manual version is about 10 times slower than the version which uses the built-in multiplication operator.

21.10 Profile Code to Find Hot-Spots

Running through the profiler records every line executed and the time required to execute. This allows hot-spots in code – code segments which require the most time – to be identified so that optimization can be focused on the code that spends the most time running.

The profiler is run using

```
>> profile on
>> code_to_profile
>> profile report
>> profile off
```

The first command turns the profile on. The second run the code to be profiled. The final command turns the profiler off and opens the profile report viewer. The file below uses concatenation which is slow. Profiling will highlight that virtually all of the computational effort is spent in the inner line in the loop.

```
% file: code_to_profile.m
% This is an example of a file that does not use best practices

text = [];
for i = 1:200000
    text = [text char(mod(i,26) + 65)];
end
```

21.11 Using Global Variables

Under normal circumstances, variables are not available in functions unless explicitly passed as inputs. Moreover, even when passed, the value of a passed variable cannot be changed inside the function and changes are discarded when the function returns (unless explicitly passed out). Global variables, on the other hand, are available both in the base MATLAB workspace and in functions. They also can be accessed and modified at any time. As a general rule, global variables should not be used. Using global variables makes debugging more difficult and lowers long-run code maintainability.

Some scenarios where globals are useful include:

- Tracking intermediate values when optimizing a function. The diagnostics available from the optimizers are limited, and using a global will allow any value visible to the optimization target (e.g. parameter values) to be saved.
- Avoiding memory allocation when the memory allocation is an important component of the total run-time of the function.

Global variables are declared using the `global` keyword. `global` should be called *prior* to initializing a variable.

```
>> x = 1;
>> whos x
  Name      Size            Bytes  Class     Attributes
  x         1x1                 8  double

>> clear x
>> global x
>> x = 1;
>> whos x
  Name      Size            Bytes  Class     Attributes
  x         1x1                 8  double    global
```

Global variables can then be accessed inside a function using the `global` keyword in the function.

```
function print_global()

global x
disp(['The value of x is ' num2str(x)])
```

Calling the function prints the value of the global variable. Note that if x is not a global it will be initialized as a global with an empty value.

```
>> print_global()
The value of x is 1
```

Finally, note that a global is only available after using the `global` keyword, and so the existence of a global variable with a particular name *does not* prevent that a variable with the same named from being used in functions in the usual, non-persistent manner.

21.12 In-place Evaluation

In general, when a function is called, `a=f(b)` and `b=f(b)` have the same performance since the output must be allocated from memory. Some functions which operate element-by-element can be evaluated "in-place" so that `a=f(b)` and `b=f(b)` are no longer the same. The reason for the difference is that when `f` operates element-by-element, it can be directly applied to `b` without allocating a new array – but only if the function output is also `b` (otherwise it would overwrite the values in `b`). Functions which support in-place evaluation include `exp` and `log`. To see the memory gains to using in-place evaluation, it is necessary to track the memory usage of MATLAB and use very large matrices (5000 by 5000 or larger). For example, the memory usage of

```
>> x = randn(5000,5000);
>> y = exp(x); % First new memory allocated
>> y = exp(x); % New memory allocated again
>> x = exp(x); % No memory allocation
```

shows that even repeated calls to `y=exp(x)` require memory allocation while `x=exp(x)` does not. Note that it is necessary to overwrite the contents of an array to use in-place operations and so they are only useful in certain situations.

Chapter 22

Examples

These examples are all actual econometric problems chosen to demonstrate the use of MATLAB in an end-to-end manner, form importing data to presenting estimates. A reasonable familiarity with the underlying econometric models and methods is assumed so that the focus can be on the translation of mathematics to MATLAB.

22.1 Estimating the Parameters of a GARCH Model

This example will highlight the steps needed to estimate the parameters of a GJR-GARCH(1,1,1) model with a constant mean. The volatility dynamics in a GJR-GARCH model are given by

$$\sigma_t^2 = \omega + \sum_{i=1}^{p} \alpha_i r_{t-i}^2 + \sum_{j=1}^{o} \gamma_j r_{t-j}^2 I_{[r_{t-j}<0]} + \sum_{k=1}^{q} \beta_k \sigma_{t-k}^2.$$

Returns are assumed to be conditionally normal, $r_t|\mathcal{F}_{t-1} \sim N\left(\mu, \sigma_t^2\right)$, and parameters are estimated by maximum likelihood. To estimate the parameters, it is necessary to:

1. Produce some starting values
2. Estimate the parameters using (quasi-) maximum likelihood
3. Compute standard errors using a "sandwich" covariance estimator (also known as the Bollerslev & Wooldridge (n.d.) covariance estimator)

The first task is to write the log-likelihood function which can be used in an optimizer. The log-likelihood function will compute the recursion and the log-likelihood. It will also, optionally, return the T by 1 vector of individual log-likelihoods which are useful for numerically computing the scores.

The log-likelihood can be defined using the normal distribution,

$$\ln f\left(r_t|\mu, \sigma_t^2\right) = -\frac{1}{2}\left(\ln 2\pi + \ln \sigma_t^2 + \frac{(r_t-\mu)^2}{\sigma_t^2}\right),$$

which is negated in the code since the optimizers only minimize.

```
function [ll,lls,sigma2] = gjr_garch_likelihood(parameters, data, backCast)
```

```
mu = parameters(1);
omega = parameters(2);
alpha = parameters(3);
gamma = parameters(4);
beta = parameters(5);

T = size(data,1);
eps = data - mu;
% Data and sigma2 are T by 1 vectors
sigma2 = zeros(T,1);
% Must use a back cast to start the algorithm
sigma2(1) = backCast;
for t = 2:T
    sigma2(t) = omega + alpha * eps(t-1)^2 ...
                + gamma * eps(t-1)^2 * (eps(t-1)<0) + beta * sigma2(t-1);
end

lls = 0.5*(log(2*pi) + log(sigma2) + eps.^2./sigma2);
ll = sum(lls);
```

The function also returns the conditional variances in the third output since the fit variances are often of interest in addition to the model parameters.

It is necessary to discuss one other function before proceeding with the main block of code. The asymptotic variance takes the "sandwich" form, which is commonly expressed as

$$\mathcal{J}^{-1}\mathcal{I}\mathcal{J}^{-1}$$

where J is the expected Hessian and $\mathcal{I}$ is the covariance of the scores. Both are numerically computed. The strategy for computing the Hessian is to use the definition that

$$\mathcal{J}_{ij} \approx \frac{f\left(\theta + e_i h_i + e_j h_j\right) - f\left(\theta + e_i h_i\right) - f\left(\theta + e_j h_j\right) + f\left(\theta\right)}{h_i h_j}$$

where h_i is a scalar "step size" and e_i is a vector of 0s except for element i, which is 1. A 2-sided version of this approximation, which takes both forward and backward steps and then averages, is below. For more on numerical derivatives, see **?**.

```
function H = hessian_2sided(fun, theta, varargin)

if size(theta,2)>size(theta,1)
    theta = theta';
end

f = feval(fun,theta,varargin{:});
h = 1e-5 * abs(theta);
thetah = theta + h;
h = thetah - theta;
K = size(theta,1);
h = diag(h);
```

```
fp = zeros(K,1);
fm = zeros(K,1);
for i = 1:K
    fp(i) = feval(fun, theta+h(:,i),varargin{:});
    fm(i) = feval(fun, theta-h(:,i),varargin{:});
end

fpp = zeros(K);
fmm = zeros(K);
for i = 1:K
    for j = i:K
        fpp(i,j) = fun(theta + h(:,i) + h(:,j),  varargin{:});
        fpp(j,i) = fpp(i,j);
        fmm(i,j) = fun(theta - h(:,i) - h(:,j),  varargin{:});
        fmm(j,i) = fmm(i,j);
    end
end
hh = diag(h);
hh = hh*hh';

H = zeros(K);
for i=1:K
    for j=i:K
        H(i,j) = (fpp(i,j) - fp(i) - fp(j) + f+ f - fm(i) - fm(j) + fmm(i,j))/hh(i,j)/2;
        H(j,i) = H(i,j);
    end
end
```

Finally, the code that does the actual work can be written. The first block imports the data, flips it so that the oldest observations are first, and computes 100 times returns. Scaling data can be useful to improve optimizer performance since ideally estimated parameters should have similar magnitude (i.e. $\omega \approx .01$ and $\alpha \approx .05$)

```
% Import data
FTSE = readtable('FTSE_1984_2012_clean.xlsx');
% Flip upside down
FTSE = flipud(FTSE);
% Compute returns
FTSE.Date = datetime(FTSE.Date,'ConvertFrom','excel');
FTSE.Ret = [nan; 100*diff(log(FTSE.Close))];
```

Good starting values are important. These are a good guess based on more than a decade of fitting models. An alternative is to implement grid search and use the best (smallest) value from the grid.

```
% Starting values
startingVals = [nanmean(FTSE.Ret),nanvar(FTSE.Ret) * .01,.03, .09, .90];
```

Bounds are used in estimation to ensure that all parameters are ≥ 0, and to set sensible upper bounds in the parameters. A constraint is placed on α, γ and β which is sufficient to ensure stationarity of the process. This is not technically necessary, although it is helpful since it prevents the volatility from exploding which produces numerical issues.

```
% Estimate parameters
LB = [-10*nanmean(FTSE.Ret) realmin 0 0 0];
UB = [10*nanmean(FTSE.Ret) 10*nanvar(FTSE.Ret) 1 2 1];
% Sum constraint
A = [0 0 1 0.5 1];
b = 1;
```

Next, a back cast is constructed to initialize the conditional variance process. This is an example of an exponential weighted moving average, only running backward in time.

```
T = size(FTSE.Ret,1);
w = .06*.94.^(0:T-2);
backCast= w*FTSE.Ret(2:end).^2;
```

The options are then specified, and the main optimization routine can be called. The two options used set the display to be iterative so that the function value at each iteration is displayed, and the set the algorithm to SQP (sequential quadratic programming) which is a good choice for many constrained problems.

```
options = optimset('fmincon');
options.Display = 'iter';
options.Algorithm = 'sqp';
estimates = fmincon(@gjr_garch_likelihood, startingVals, ...
              A, b, [], [], LB, UB, [], options, FTSE.Ret(2:end), backCast);
```

The optimized log-likelihood and the time series of variances are computed by calling the objective using the parameters found by the optimizer.

```
[loglik, logliks, sigma2] = gjr_garch_likelihood(estimates, FTSE.Ret(2:end), backCast);
```

Next, the numerical scores and the covariance of the scores are computed. These exploit the definition of a derivative, so that for a scalar function,

$$\frac{\partial f(\theta)}{\partial \theta_i} \approx \frac{f(\theta + e_i h_i) - f(\theta)}{h_i}.$$

The covariance is computed as the outer product of the scores since the scores should have mean 0 when evaluated at the solution to the optimization problem.

```
% Covariance
step = 1e-5 * estimates;
scores = zeros(T-1,5);
for i=1:5
    h = step(i);
    delta = zeros(1,5);
    delta(i) = h;

    [~, logliksplus]  = gjr_garch_likelihood(estimates + delta, FTSE.Ret(2:end), backCast);
    [~, logliksminus] = gjr_garch_likelihood(estimates - delta, FTSE.Ret(2:end), backCast);
    scores(:,i) = (logliksplus - logliksminus)/(2*h);
end
I = scores'*scores/T;
```

The final block of the numerical code calls `hessian_2sided` to estimate the Hessian and finally computes the asymptotic covariance.

```
% Hessian
J = hessian_2sided(@gjr_garch_likelihood, estimates, FTSE.Ret(2:end), backCast);
J = J/T;
Jinv = J\eye(length(J));
vcv = Jinv*I*Jinv/T;
```

The remaining steps are to pretty print the results and to produce a plot of the conditional variances,

```
% Pretty print parameters, standard error and t-stat
output = [estimates', sqrt(diag(vcv)), estimates'./sqrt(diag(vcv))];
disp(' Parameter   Estimate     Std. Err.      T-stat')
param = {'mu','omega','alpha','gamma','beta'};
for i = 1:length(estimates)
    fprintf('%10s %10.3f %13.3f %11.3f \n',param{i},output(i,1),output(i,2),output(i,3));
end
```

This final code block produce a plot of the annualized conditional standard deviations.

```
% Produce a plot
plot(FTSE.Date(2:end),sqrt(252*sigma2));
axis tight;
ylabel('Volatility')
title('FTSE Volatility (GJR GARCH(1,1,1))')
```

22.2 Estimating the Risk Premia using Fama-MacBeth Regressions

This example highlights how to implement a Fama-MacBeth 2-stage regression to estimate factor risk premia, make inference on the risk premia, and test whether a linear factor model can explain a cross-section of portfolio returns. This example closely follows Cochrane (n.d.) (See also Jagannathan et al. (n.d.)).

First, the data are imported. I formatted the data downloaded from Ken French's website into an easy-to-import CSV which can be read by `readtable`. The data in the `table` is split into different variables (as arrays), and the dimensions are determined using `size`.

```
% Import data
data = readtable('famafrench.csv');
% Split using slices
dates = data.date;
factors = data{:,{'VWMe','SMB','HML'}};
riskfree = data{:,'RF'};
portfolios = data{:,6:end};
% Shape information
[T,K] = size(factors);
[T,N] = size(portfolios);
% Compute excess returns
excessReturns = bsxfun(@minus,portfolios,riskfree);
```

The next block does 2 things:

1. Compute the time-series βs. This is done be regressing the full array of excess returns on the factors (augmented with a constant) using \.

2. Compute the risk premia using a cross-sectional regression of average excess returns on the estimates βs. This is a standard regression where the step-1 β estimates are used as regressors, and the dependent variable is the average excess return.

```
% Time series regressions
X = [ones(T,1) factors];
alphaBeta = X\excessReturns;
alpha = alphaBeta(1,:)';
beta = alphaBeta(2:4,:)';
avgExcessReturns = mean(excessReturns)';
% Cross-section regression
lam = beta\avgExcessReturns;
```

The asymptotic variance requires computing the covariance of the demeaned returns and the weighted pricing errors. The problem is formulated as a 2-step GMM estimation where the moment conditions are

$$g_t(\theta) = \begin{bmatrix} \epsilon_{1t} \\ \epsilon_{1t} f_t \\ \epsilon_{2t} \\ \epsilon_{2t} f_t \\ \vdots \\ \epsilon_{Nt} \\ \epsilon_{Nt} f_t \\ \beta u_t \end{bmatrix}$$

where $\epsilon_{it} = r^e_{it} - \alpha_i - \beta'_i f_t$, β_i is a K by 1 vector of factor loadings, f_t is a K by 1 set of factors, $\beta = [\beta_1\, \beta_2 \ldots \beta_N]$ is a K by N matrix of all factor loadings, $u_t = r^e_t - \beta'\lambda$ are the N by 1 vector of pricing errors and λ is a K by 1 vector of risk premia. The collection of parameters is $\theta = \left[\alpha_1\ \beta'_1\ \alpha_2\ \beta'_2\ \ldots\ \alpha_N\ \beta'_N\ \lambda'\right]'$. In order to make inference on this problem, the derivative of the moments with respect to the parameters, $\partial g_t(\theta)/\partial\theta'$ is needed. With some work, the estimator of this matrix can be seen to be

$$G = E\left[\frac{\partial g_t(\theta)}{\partial\theta'}\right] = \begin{bmatrix} -I_n \otimes \Sigma_X & 0 \\ G_{21} & -\beta\beta' \end{bmatrix}.$$

where $X_t = \left[1\ f'_t\right]'$ and $\Sigma_X = E\left[X_t X'_t\right]$. G_{21} is a matrix with the structure

$$G_{21} = \left[G_{21,1}\ G_{21,2}\ \ldots G_{21,N}\right]$$

where

$$G_{21,i} = \begin{bmatrix} 0_{K,1} & \text{diag}\left(E\left[u_i\right] - \beta_i \odot \lambda\right) \end{bmatrix}$$

and where $E[u_i]$ is the expected pricing error. In estimation, all expectations are replaced with their sample analogues.

```
% Moment conditions
p = alphaBeta;
epsilon = excessReturns - X*p;
moments1 = kron(epsilon,ones(1,K+1));
```

```
moments1 = moments1 .* kron(ones(1,N),X);
u = bsxfun(@minus,excessReturns,lam'*beta');
moments2 = u*beta;
% Score covariance
S = cov([moments1 moments2]);
% Jacobian
G = zeros(N*K+N+K,N*K+N+K);
SigmaX = X'*X/T;
G(1:N*K+N,1:N*K+N) = kron(eye(N),SigmaX);
G(N*K+N+1:end,N*K+N+1:end) = -beta'*beta;
for i=1:N
    temp = zeros(K,K+1);
    values = mean(u(:,i))-beta(i,:).*lam';
    temp(:,2:end) = diag(values);
    G(N*K+N+1:end,(i-1)*(K+1)+1:i*(K+1)) = temp;
end

vcv = inv(G')*S*inv(G)/T;
```

The J test examines whether the average pricing errors, $\hat{\alpha}$, are zero. The J statistic has an asymptotic χ^2_N distribution, and the model is badly rejected.

```
vcvAlpha = vcv(1:4:N*K+N,1:4:N*K+N);
J = alpha'*inv(vcvAlpha)*alpha;
Jpval = 1 - chi2cdf(J,25);
```

The next block formats the output to present all of the results in a readable manner. In particular, `fprintf` is used to print the estimated parameters to screen.

```
riskPremia = lam;
vcvLam = vcv(N*K+N+1:end,N*K+N+1:end);
annualizedRP = 12*riskPremia;
arpSE = sqrt(12*diag(vcvLam));
fprintf('        Annualized Risk Premia\n')
fprintf('           Market       SMB        HML\n')
fprintf('--------------------------------------\n')
fprintf('Premia     %0.4f     %0.4f     %0.4f\n',annualizedRP)
fprintf('Std. Err.  %0.4f     %0.4f     %0.4f\n',arpSE)
fprintf('\n\n')

fprintf('J-test:   %0.4f\n',J)
fprintf('P-value:   %0.4f\n\n\n',Jpval)

i=1;
betaVar = zeros(25,4);
for j=1:5
    for k=1:5
        a = alpha(i);
        b = beta(i,:);
        offset = (K+1)*(i-1)+1:(K+1)*(i);
        variances = diag(vcv(offset,offset))';
        % Lazy concatenation
```

```
        betaVar(i,:) = variances;
        s = sqrt(variances);
        c = [a b];
        t = c./s;
        fprintf('Size: %d, Value:%d   Alpha   Beta(VWM)   Beta(SMB)   Beta(HML)\n',j,k)
        fprintf('Coefficients: %10.4f  %10.4f  %10.4f  %10.4f\n',c);
        fprintf('Std Err.      %10.4f  %10.4f  %10.4f  %10.4f\n',s);
        fprintf('T-stat        %10.4f  %10.4f  %10.4f  %10.4f\n\n',t);
        i = i + 1;
    end
end
```

The final block saves the data and estimates.

```
save('Fama-MacBeth_results','alpha','beta','betaVar','arpSE','annualizedRP','J','Jpval')
```

22.3 Estimating the Risk Premia using GMM

The final numeric example estimates the same problem, only using GMM rather than 2-stage regression. The GMM objective takes the parameters, portfolio returns, factor returns and the weighting matrix and computes the moments, average moments and the objective value. The moments used can be described as

$$\left(r_{it}^2 - \beta_i f_t\right) f_t \quad \forall i = 1, \ldots N$$

and

$$r_{it} - \beta_i \lambda \quad \forall i = 1, \ldots N.$$

```
function [J,moments] = gmm_objective(params, pRets, fRets, Winv)

N = size(pRets,2);
[T,K] = size(fRets);

beta = params(1:N*K);
lam = params(N*K+1:end);
beta = reshape(beta,N,K);
lam = reshape(lam,K,1);
betalam = beta*lam;
expectedRet = fRets*beta';
e = pRets - expectedRet;
instr = repmat(fRets,1,N);
moments1  = kron(e,ones(1,K));
moments1 = moments1 .* instr;
moments2 = bsxfun(@minus,pRets,betalam');
moments = [moments1 moments2];

avgMoment = mean(moments);

J = T * avgMoment*Winv*avgMoment';
```

The final function needed is the Jacobian of the moment conditions. Mathematically it is simply to express the Jacobian using $\otimes$(Kronecker product). This code is so literal that it is simple to reverse engineer the mathematical formulas used to implement this estimator.

$$\hat{G} = \begin{bmatrix} I_N \otimes \Sigma_F & 0 \\ I_N \otimes \lambda & -\beta \end{bmatrix}$$

```
function G = gmm_G(params, pRets, fRets)

N = size(pRets,2);
[T,K] = size(fRets);
beta = params(1:N*K);
lam = params(N*K+1:end);
beta = reshape(beta,N,K);
lam = reshape(lam,K,1);
G = zeros(N*K+K,N*K+N);

ffp = fRets'*fRets/T;
G(1:N*K,1:N*K)=kron(eye(N),ffp);
G(1:(N*K),(N*K)+1:end) = kron(eye(N),-lam);
G((N*K)+1:end,(N*K)+1:end) = -beta';
```

The data import step is virtually identical to that in the previous example – although it shows some alternative functions to accomplish the same tasks. Note that only portfolios in odd-numbered columns are selected in order to speed up the GMM optimization.

```
data = csvread('famafrench.csv',1);
dates = data(:,1);
factors = data(:,2:4);
riskfree = data(:,5);
portfolios = data(:,6:end);
N = size(portfolios,2);
portfolios = portfolios(:,1:2:N);
[T,N] = size(portfolios);
excessRet = bsxfun(@minus,portfolios,riskfree);
K = size(factors,2);
```

Starting values are important in any optimization problem. The GMM problem is closely related to Fama-MacBeth regression, and so it is sensible to use the output from an FMB regression.

```
augFactors = [ones(T,1) factors];
alphaBeta = augFactors\excessRet;
betas = alphaBeta(2:4,:);

avgReturn = mean(excessRet)';
riskPremia = betas'\avgReturn;
```

The GMM objective can be minimized using an identity matrix as the covariance of the moment conditions along with the starting values computed using a Fama-MacBeth regression.

```
startingVals = [betas(:);riskPremia];
```

```
Winv = eye(N*(K+1));

options = optimset('fminunc');
options.Display = 'iter';
options.LargeScale = 'off';
step1opt = fminunc(@gmm_objective,startingVals,options,excessRet,factors,Winv);
```

Once the initial estimates have been computed, these can be used to estimate the covariance of the moment conditions, which is then used to estimate the optimal weighting matrix.

```
[J,moments]= gmm_objective(step1opt, excessRet, factors, Winv);
S = cov(moments);
Winv2 = inv(S);
options.MaxFunEvals = 10000;
step2opt = fminunc(@gmm_objective,step1opt,options,excessRet,factors,Winv2);
```

The final block computes estimates the asymptotic covariance of the parameters using the usual efficient GMM covariance estimator, assuming that the moments are a martingale.

```
[J,moments] = gmm_objective(step2opt, excessRet, factors, Winv2);
G = gmm_G(step2opt, excessRet, factors);
S = cov(moments);
vcv = inv(G*inv(S)*G')/T;
```

22.4 Outputting LaTeX

Automatically outputting results to LaTeX or another format can eliminate export errors and avoid tedious work. This example shows how two of the tables in the previous Fama-MacBeth example can be exported to a LaTeX document, and how, if desired, the document can be compiled to a PDF. The first code block contains code to clear the workspace, clear the window (`clc`) and to set a flag indicating whether the MATLAB code should compile the latex file.

```
clear all
clc
fclose('all');
% Flag to compile output tables
compileLatex = true;
```

The next code block loads the mat file created using the output from the Fama-MacBeth example.

```
% Load variables
load('Fama-MacBeth_results.mat')
```

The document will be stored in a cell array. The first few lines contain the required header for a LaTeX document, including some packages used to improve table display and to select a custom font. Most of this code uses lazy concatenation – that is appending lines to an existing variable. While this is generally a bad practice from a performance perspective, concatenation is a useful technique in situations where performance is not important. The concatenation in the cell array is implemented using `latex{end+1}` which tells MATLAB to place the new information 1 after the last element.

```
% Cell to hold table, initially empty
latex = cell(6,1);
% Initializd LaTeX document
latex{1} = '\documentclass[a4paper]{article}';
latex{2} = '\usepackage{amsmath}';
latex{3} = '\usepackage{booktabs}';
latex{4} = '\usepackage[adobe-utopia]{mathdesign}';
latex{5} = '\usepackage[T1]{fontenc}';
latex{6} = '\begin{document}';
```

Table 1 will be stored in its own cell array, and then concatenated onto the main LaTeX code. Building this table is string manipulation, `num2str` and `sprintf`.

```
% Table 1
table1 = cell(2,1);
table1{1} = '\begin{center}';
table1{2} = '\begin{tabular}{lrrr} \toprule';
% Header
colNames = {'VWM$^e$','SMB','HML'};
header = '';
for cName=colNames
    header = [header ' & '  cName{:}];
end

header = [header '\\ \cmidrule{2-4}'];
table1{end+1} = header;

% Main row
row = '';
for i=1:length(annualizedRP)
    row =[row  ' & $\underset{{(' num2str(arpSE(i),'%0.3f') ...
          ')}}{{' num2str(annualizedRP(i),'%0.3f') '}}$'];
end

table1{end+1}=row;
% Blank row
row = '\\';
table1{end+1} = row;
% J-stat row
row = sprintf('J-stat: $\\underset{{(%0.3f)}}{{%0.1f}}$ \\\\',Jpval,J);
table1{end+1}=row;

table1{end+1} = '\bottomrule \end{tabular}';
table1{end+1} = '\end{center}';
% Extend latex with table 1
latex = [latex; table1];
latex{end+1} = '\newpage';
```

Table 2 is a more complex and uses loops to iterate over the rows of the arrays containing the βs and their standard errors.

```
% Format information for table 2
```

```
sizes = {'S','2','3','4','B'};
values = {'L','2','3','4','H'};
% Table 2 has the same header as table 1, copy with a slice
table2 = table1(1:3);
m = 1;
for i=1:5
    for j=1:5
        row = sprintf('Size: %s, Value: %s',sizes{i},values{j});
        b = beta(m,:);
        s = sqrt(betaVar(m,2:end));
        for k=1:length(b)
            row = [row sprintf(' & $\\underset{{(%0.3f)}}{{%1.3f}}$',s(k),b(k))];
        end
        row = [row ' \\ '];
        table2{end+1}=row;
        m = m + 1;
        if j==5 && i~=5
            table2{end+1}= '\cmidrule{2-4}';
        end
    end
end


table2{end+1} = '\bottomrule \end{tabular}';
table2{end+1} = '\end{center}';
% Extend with table 2
latex = [latex;table2];
```

The penultimate block finished the document, and uses `fprintf` to write the lines to the LaTeX file. `fprintf` does not break lines, so the new line character is added to each (\n). Note that MATLAB treats text starting with a slash (\) as an *escape sequence*, and so it is necessary to escape the slashes in the LaTeX. This means that \ in the output LaTeX must be \\ prior to being written.

```
% Finish document
latex{end+1}= '\end{document}';
% Write to table
fid = fopen('latex.tex','wt');
for i=1:length(latex)
    temp = latex{i};
    % Escape slashes if needed
    slashes = strfind(temp,'\');
    if ~isempty(slashes)
        temp = [temp; repmat(char(0),1,length(temp))];
        temp(2,slashes) = '\';
        temp = temp(temp~=char(0))';
    end
    fprintf(fid,[temp  '\n']);
end
fclose(fid);
```

Finally, if the flag is set, `system` is used to compile the LaTeX. This assumes that `pdflatex` is on the system path.

```
% Compile if needed
if compileLatex
    exitStatus = system('pdflatex latex.tex');
end
```

Chapter 23

Parallel MATLAB

To be completed

Chapter 24

Quick Function Reference

This list contains a brief summary of the functions most useful in the MFE course. It only scratches the surface of what MATLAB offers. There are approximately 100 functions listed here; MATLAB and the Statistics Toolbox combined contain more than 1400.

24.1 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 a division operation where the elements of a vector or matrix are divided by a scalar or conformable vector or matrix.

`roots`

Returns the roots of a polynomial.

`sqrt`

Returns the square root of a number. Operates element-by-element on vectors or matrices.

`sign`

Returns the sign, defined as $x/|x|$ and 0 if $x = 0$, of the elements of a vector or matrix. Operates element-by-element on vectors or matrices.

`sum`

Returns the sum of the elements of a vector. If used on a matrix, operated column-by-column.

24.2 Rounding

`ceil`

Returns the next larger integer. Operates element-by-element on vectors or matrices.

`floor`

Returns the next smaller integer. Operates element-by-element on vectors or matrices.

`round`

Rounds to the nearest integer. Operates element-by-element on vectors or matrices.

24.3 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.

`prctile`

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

`regress`

Estimates a classic linear regression. *Does not* compute White heteroskedasticity-robust standard errors.

`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.

$DIST$**`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.

$DIST$**`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.

$DIST$**`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: $DIST$ 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.

24.4 Random Numbers

`rand`

Uniform pseudo-random number generator. One of three core random number generators that are used to produce pseudo-random numbers from other distributions.

`randg`

Standard gamma pseudo-random number generator. One of three core random number generators that are used to produce pseudo-random numbers from other distributions.

randn

Standard normal pseudo-random number generator. One of three core random number generators that are used to produce pseudo-random numbers from other distributions.

random

Generic pseudo-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.

24.5 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.

ischar

Returns logical true if the argument is a string.

isfinite

Returns logical true if the argument is finite. Operates element-by-element on vectors or matrices.

isinf

Returns logical true if the argument is infinite. Operates element-by-element on vectors or matrices.

isnan

Returns logical true if the argument is not a number (`NaN`). Operates element-by-element on vectors or matrices.

isreal

Returns logical true if the argument is not complex.

logical

Converts non-logical variables to logical variables. Operates element-by-element on vectors or matrices.

24.6 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 numerically identical.

Inf

`Inf` represents infinity.

NaN

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

pi

Returns the value of π.

24.7 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 (inclusive).

logspace

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

ones

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

toeplitz

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

zeros

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

24.8 Vector and 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 the elements of the vector along the 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.

cumprod

Computes the cumulative product of a vector. `y = cumprod(x)` computes $y_i = \prod_{j=1}^{i} x_j$. If used on a matrix, operates column-by-column.

cumsum

Computes the cumulative sum of a vector. `y = cumsum(x)` computes $y_i = \sum_{j=1}^{i} x_j$. If used on a matrix, operates column-by-column.

24.9 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 and is equivalent to `max(size(x))`.

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.

24.10 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.

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).

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 on a matrix to select the set of unique rows.

24.11 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 signaling the beginning of a `switch` ... `case`... `otherwise` flow control block. Switch should be followed by a variable to be used by `case`.

24.12 Looping

continue

Forces a loop to proceed to the next iteration while bypassing any code occurring after the `continue` statement.

break

Prematurely breaks out of a loop before the all iterations have completed.

end

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

for

One of two types of loops. `for` loops iterate over a predefined vector unless prematurely ended by `break`.

while

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

24.13 Optimization

fminbnd

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

fmincon

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

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.

24.14 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 3-D bar plot of a vector or matrix.

colormap

Colors figures according to the selected color file.

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(XX)` opens a window with label Figure XX where XX is some integer. If a windows with label Figure XX is already open, that figure is set as the active figure and any subsequent plot commands will operate on Figure XX.

gcf

Gets the handle of the current figure.

get

Gets of list of properties from a graphics handle or the value of a property if used with an optional second argument.

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

pie

Produces a pie chart.

plot

Plots x data against y data.

print

Saves a figure to disk in a wide range of formats.

plot3

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

scatter

Produces a scatter plot of x data against y data.

set

Sets a property of a graphics handle.

shading

Changes the shading method for 3-D figures.

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 3-D 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.

24.15 Date Functions

clock

Returns the current date and time as a 6 by 1 numeric vector of the form `[YEAR MONTH DATE HOUR MIN SEC]`.

date

Returns string with current date.

datenum

Converts string dates, such as 1-Jan-2000, 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]`.

etime

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

now

Returns the current time in MATLAB serial date format.

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.

x2mdate

Converts excel dates in the MATLAB serial dates.

24.16 String Function

char

Converts numeric values to their ASCII character equivalents.

double

Converts values to double precision from character or other data types.

num2str

Converts numbers to strings for output or inclusion in graphics.

str2double

Converts string numbers to doubles. Limited but fast.

str2num

Converts string numbers to doubles. Flexible but slow.

strcat

Horizontally concatenates two or more strings. Equivalent to `[string1 string2]` for strings with the same number of rows.

strcmp

Compares two string values using using a case-sensitive comparison.

strcmpi

Compares two string values using using a case-insensitive comparison.

strfind

Finds substrings in a string.

strmatch

Finds exact string matches.

strncmp

Compares the first n characters of two strings using a case-sensitive comparison.

strncmpi

Compares the first n characters of two strings using a case-insensitive comparison.

strvcat

Vertically concatenates two or more strings. If the strings have different numbers of columns, right pads the shorter string with blanks.

24.17 Trigonometric Functions

cos

Computes the cosine of a scalar, vector or matrix. Operates element-by-element on vectors or matrices.

sin

Computes the sine of a scalar, vector or matrix. Operates element-by-element on vectors or matrices.

24.18 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; files deleted are *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.

24.19 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`.

doc

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

edit

Launches the built-in editor. If called using `edit` *filename*, opens the editor with *filename.m* or, if *filename.m* does not exist on the MATLAB path, creates the file in the current directory.

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

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.

keyboard

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

profile

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

- `profile on` turns the profiles on
- `profile off` turns the profiles off
- `profile report` opens the profiling report which contains statics on the performance on code executed since `profile on` was called. Does not stop the profiler.
- `profile viewer` turns the profiles off and opens the profiling report which contains statics on the performance on code executed since `profile on` was called

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.

24.20 Input/Output

csvread

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

csvwrite

Saves variables to a .csv file.

fclose

Used to close a file handle opened using `fopen`.

fgetl

Reads the current file until an end-of-line character is encountered, returning a string representing the line without the end-of-line character.

fopen

Opens a file for low level reading (using e.g. `fgetl`) or writing (using e.g. `fprintf`).

fprintf

Writes formatted text to a file.

load

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

save

Saves 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 a tab delimited text file.

textread

Older method to read formatted text. Has been replaced by `textscan`.

textscan

Reads formatted text. Can read into cell arrays and from specific points in a file.

xlsfinfo

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

xlsread

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

xlswrite

Saves variables to an .xls file.

Bibliography

Bollerslev, T. & Wooldridge, J. M. (n.d.), 'Quasi-maximum likelihood estimation and inference in dynamic models with time-varying covariances', **11**(2), 143–172.

Cochrane, J. H. (n.d.), *Asset Pricing*, Princeton University Press.

Jagannathan, R., Skoulakis, G. & Wang, Z. (n.d.), The analysis of the cross section of security returns, *in* Y. AÃ¯t-Sahalia & L. P. Hansen, eds, 'Handbook of financial econometrics', Vol. 2, Elsevier B.V., pp. 73–134.

Index

%, 4
*cdf, 119
*inv, 119
*pdf, 119
*rnd, 119
:, 31
;, 4
~, 122

A
abs, 169
all, 44, 173
AND, 42
ans, 174
any, 44, 173
array2table, 109
axis, 179

B
bar, 60, 179
bar3, 179
Basic Functions
  length, 21
  size, 21
break, 53, 178
bsxfun, 37

C
case, 48, 177
cat, 176
categoricals, 111
cd, 140, 184
cdf, 172
ceil, 170
cell, 93
Cell Arrays, 92–94
cell2table, 109
char, 85, 182
chol, 38, 175
clc, 184
clear, 184
clf, 184
clock, 80, 181
close, 184
colormap, 179
Comments, 4
continue, 54, 178
contour, 65, 179
copyfile, 141
corr, 171
corrcoef, 171
cos, 183
cov, 27, 171
csvread, 100, 186
csvwrite, 100, 105, 186
cummax, 24
cummin, 24
cumprod, 24, 176
cumsum, 24, 176

D
Data Importing
  csvread, 100
  dlmread, 100
  load, 101
  readtable, 96
  textscan, 103
  xlsflinfo, 99
  xlsread, 99
date, 181
Date Functions
  x2mdate, 100
datenum, 78, 103, 181
datestr, 79, 182
datetick, 81, 182

datetimes, 112
datevec, 80, 182
delete, 141, 184
det, 38, 175
diag, 36, 175
diff, 169
dir, 139, 140, 184
disp, 124
dlmread, 100
dlmwrite, 105
doc, 5, 185
dos, 141
double, 86, 182

E
edit, 2, 185
Editor, 1
eig, 38, 175
else, 47, 177
elseif, 47, 177
end, 178
eps, 174
errorbar, 179
etime, 80, 182
exp, 25, 169
Exporting Data, 104, 105
    csvwrite, 105
    dlmwrite, 105
    save, 104, 105
    writetable, 105
    xlswrite, 105
eye, 33, 174

F
fclose, 103, 186
fgetl, 103, 186
figure, 180
filesep, 141
find, 173
floor, 170
fminbnd, 133, 178
fmincon, 134, 179
fminsearch, 133, 179
fminunc, 132, 179
fopen, 103, 186
for, 50, 178
format, 185
fprintf, 90, 186
fullfile, 141
function, 121
Functions, 121–125
    Calling, 12
    Comments, 124
    Custom, 121
    Debugging, 124

G
gcf, 180
get, 180

H
height, 116
Help, 4
help, 4, 185
helpbrowser, 185
helpdesk, 185
hist, 180

I
if, 47, 178
Importing Data, 95–104
Inf, 174
innerjoin, 115
Input, 7–13
intersect, 177
Introduction, 1–5
inv, 38, 172, 175
ischar, 173
isfinite, 173
isinf, 173
isnan, 173
isreal, 174

J
join, 115

K
keyboard, 124, 185
kron, 38, 175
kurtosis, 29, 171

L
. . ., 4
legend, 57, 59, 60, 63, 69, 180
length, 21, 176
linspace, 32, 174
load, 101, 186
log, 26, 169
log10, 169
logical, 43, 174
logspace, 33, 174
lower, 86

M
Math Functions
- cummax, 24
- cummin, 24
- cumprod, 24
- cumsum, 24
- exp, 25
- log, 26
- max, 23
- mean, 26
- min, 23
- prod, 23
- sqrt, 26
- sum, 22

MATLAB Path, 142
Matrices
- Math, 15–18
  - Addition, 15
  - Division, 16
  - Dot Operations, 17
  - Multiplication, 16
  - Operator Precedence, 18
  - Subtraction, 15
- Transpose, 18

max, 23, 170
mean, 26, 170
mesh, 65
min, 23, 170
mkdir, 141, 184
M-Lint, 149
mod, 170
movefile, 141
Moving Window Functions
- movmax, 29
- movmean, 29
- movmedian, 29
- movmin, 29
- movstd, 29
- movvar, 29

movmax, 29
movmean, 29
movmedian, 29
movmin, 29
movstd, 29
movvar, 29

N
NaN, 174
nan, 33
nargin, 122
nargout, 123
NOT, 42
now, 80, 182
num2str, 89, 182
numel, 176

O
ones, 33, 175
Operator Precedence, 18
optimget, 179
Optimization, 131–138
- Bounded, 133
- Constrained, 134
- Derivative-based, 132
- Derivative-free, 133

optimset, 138, 179
OR, 42
otherwise, 48
outerjoin, 115

P
pdf, 172
Performance, 145
pi, 174
pie, 180
plot, 57, 180
plot3, 63, 180

prctile, 119, 120, 171
print, 71, 180
prod, 23
profile, 149, 185
pwd, 184

Q

quantile, 119, 171

R

rand, 129, 172
randg, 129, 172
randi, 129
randn, 129, 173
random, 173
readtable, 96, 108
realmax, 186
realmin, 186
regexp, 88
regexpi, 88
regress, 120, 171
repmat, 35, 176
research, 124
reshape, 36, 176
return, 124
rmdir, 141, 184
rnd, 172
roots, 170
round, 171
rowfun, 116

S

save, 104, 105, 187
scatter, 60, 180
set, 181
Set Functions
    sort, 24
setdiff, 177
shading, 181
sign, 170
sin, 183
size, 21, 176
skewness, 28, 171
sort, 24, 177
sortrows, 24, 177
sqrt, 26, 170
sscanf, 89
Startup M-file, 142
Statistical Functions
    cov, 27
    kurtosis, 29
    skewness, 28
    std, 28
    var, 27
std, 28, 172
str2double, 89, 103, 182
str2num, 88, 183
strcat, 86, 183
strcmp, 87, 183
strcmpi, 87, 183
strfind, 86, 103, 183
strjoin, 88
strmatch, 87, 183
strncmp, 87, 183
strncmpi, 87, 183
strsplit, 88
struct, 91
struct2table, 109
Structures, 91, 92
strvcat, 183
subplot, 65, 181
sum, 22, 170
summary, 116
surf, 63, 180, 181
switch, 48, 178
system, 141

T

table, 107
table2array, 114
table2cell, 114
table2struct, 114
Tables, 107–117
    array2table, 109
    categoricals, 111
    cell2table, 109
    height, 116
    innerjoin, 115
    join, 115

- outerjoin, 115
- readtable, 108
- Row Names, 110
- rowfun, 116
- Selecting Elements, 113
- Special Features, 110
- struct2table, 109
- summary, 116
- table, 107
- table2array, 114
- table2cell, 114
- table2struct, 114
- varfun, 116
- Variable Names, 110
- width, 116
- writetable, 115

textread, 187
textscan, 103, 187
tic, 81, 182
timeit, 149
title, 57, 181
toc, 81, 182
toeplitz, 175
trace, 38, 175
tril, 175
triu, 176
type, 139

U

union, 177
unique, 177
upper, 86

V

var, 27, 172
varargin, 123
varargout, 123
varfun, 116
Variable Names, 7

W

which, 2, 186
while, 52, 178
whos, 186
width, 116
writetable, 105, 115

X

x2mdate, 100, 182
xlabel, 57, 181
xlsfinfo, 187
xlsflinfo, 99
xlsread, 99, 187
xlswrite, 99, 105, 187

Y

ylabel, 57, 181

Z

zeros, 33, 175
zlabel, 57, 181