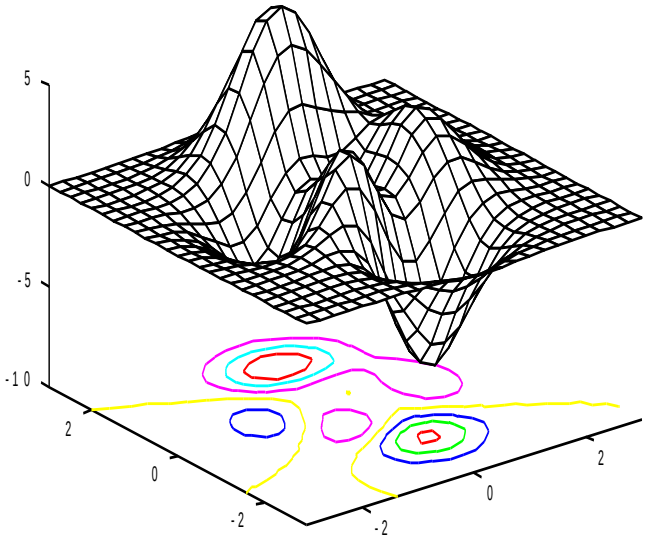
UNIVERSITY OF Southampton

Faculty of Engineering and the Environment

MATLAB REFERENCE BOOK



Prof Simon J Cox

Booklet Production

MATLAB 4 material by SJ Cox.

MATLAB 5 update by SJ Cox.

Second Edition October 1997 (Version 2.0)

Second Edition Reprinting October 1998 (Version 2.1: Minor Corrections)

Second Edition Reprinting October 1999

Reprinted Oct 2000

Reprinted Nov 2001

Reprinted Nov 2010

Key Features	Additional Features	Extras
Basic Mathematics (1.2)	Handling arrays (1.6)	Complex Numbers (1.3)
"Help" (1.4) Array operations (1.5)	Relational Operators (1.8)	Linear Algebra (1.9.2 – 1.9.5)
2D plotting (1.7)	Linear Algebra (1.9.1)	Special Matrices (1.10)
3D plotting (1.18)	Text Handling (1.11)	Polynomials (1.14)
Programming (1.12 and 1.13)	Curve fitting & interpolation (1.15.1-1.15.6)	Numerical Analysis (1.15.7 – 1.15.10)
	Sparse Matrices & Optimisation tips (1.17)	Data Analysis example (1.16)

Contents

1 MATLAB Tutorial *3*

1.1 Introduction **3**

1.2 Basic Features **4**
1.2.1 Simple Mathematics 4
1.2.2 The MATLAB working environment 6
1.2.3 Saving and Retrieving Data 7
1.2.4 Display Precision 7
1.2.5 Variable names 8

1.3 Mathematical Functions **9**
1.3.1 Complex Numbers 9

1.4 Help **11**

1.5 Array Operations **13**
1.5.1 Array Assignments: Rows 13
1.5.2 Array Addressing : Colon Notation I 13
1.5.3 Array Construction: Colon notation II 14
1.5.4 Array Assignments: Columns 16
1.5.5 Scalar-Array Calculations 17
1.5.6 Array-Array Calculations 18
1.5.7 The transpose operator 20
1.5.8 Multidimensional Arrays 21

1.6 Handling Arrays **21**
1.6.1 Accessing Array Subsets and Re-ordering Arrays: Colon Notation IV. 21
1.6.2 Re-assigning array elements 22
1.6.3 Reshaping arrays 23
1.6.4 How Big is a Matrix? 24
1.6.5 Other Operations for Handling Arrays 25

1.7 2-D plotting of functions **25**
1.7.1 Simple use of the `plot` command 25
1.7.2 Linestyles, Markers and Colours 26
1.7.3 Multiple plots 27
1.7.4 Labels and Grids 28
1.7.5 Customising Axes 29
1.7.6 A Multitude of Graph Types 32
1.7.7 Handling Plots: Hold, Subplot, Figure, and Zoom. 38
1.7.8 Producing Hardcopy: Printing to Paper or File. 41

1.8 Relational and Logical Operations **42**
1.8.1 Relational Operators 42
1.8.2 Logical Operators 43
1.8.3 Using Relational Operators to Address Arrays 44
1.8.4 Other Matrix Operators returning True or False 46

1.9 Linear Algebra **46**
1.9.1 Simultaneous Equations 47
1.9.2 Badly-Conditioned Problems 48
1.9.3 Poor Scaling 49
1.9.4 Finding Eigenvalues and Eigenvectors 49
1.9.5 Other Linear Algebra Tools 49

1.10 Special Matrices, and the Rogues' Gallery **50**
1.10.1 Special Matrices 50
1.10.2 The Rogues' Gallery 53

1.11 Text **54**
1.11.1 Character strings 54
1.11.2 Handling Character Strings 54

1.11.3 Cell Arrays 55

1.12 MATLAB Files and Functions 56
1.12.1 File Handling commands 56
1.12.2 Script Files 57
1.12.3 Adding New Functions 58
1.12.4 Summary 59

1.13 MATLAB Programming Structures 59
1.13.1 For Loops 59
1.13.2 While Loops 61
1.13.3 If .. Else Decisions 61

1.14 Polynomials 62
1.14.1 Polynomial Storage 63
1.14.2 Roots of a Polynomial 63
1.14.3 Adding Polynomials 63
1.14.4 Multiplying Polynomials 64
1.14.5 Dividing Polynomials 64
1.14.6 Polynomial Derivatives 65
1.14.7 Polynomial Evaluation 65
1.14.8 Partial Fractions 66

1.15 An Introduction to Numerical Analysis with MATLAB 66
1.15.1 Curve Fitting: Least Squares 67
1.15.2 Interpolation I: Linear 69
1.15.3 Interpolation II: Polynomials 70
1.15.4 Interpolation III: Splines 71
1.15.5 The Humps function 72
1.15.6 Interpolation III: Surface Splines 72
1.15.7 Function Minimization or Maximization 74
1.15.8 Finding Zeros of a Function 74
1.15.9 Numerical Integration 75
1.15.10 Differential Equations 78

1.16 Data Analysis 79
1.16.1 A Worked Example 79
1.16.2 Visualizing the data 80
1.16.3 Determining Statistical Properties of the Data 81
1.16.4 Other Statistical Properties of Data 82
1.16.5 A Linear Relationship for the data 82
1.16.6 An Power Law Relationship for the data 83

1.17 Some Optimisation Tips 84
1.17.1 Vectorisation and Built-In Functions 84
1.17.2 Subscripting 85
1.17.3 Array Operations 85
1.17.4 Boolean Array operations 86
1.17.5 Constructing Matrices from Vectors 87
1.17.6 Constructing Special Matrices 88
1.17.7 Functions of two variables 89
1.17.8 Redundancy 90
1.17.9 Sparse Matrices 93
1.17.10 Conclusion 94

1.18 3-D Graphics 94
1.18.1 An extension of two dimensional plotting 94
1.18.2 Mesh Plots 95
1.18.3 Colour Maps 97
1.18.4 Back to two dimensions 97
1.18.5 A helpful suggestion 98

1.19 About the front cover 98

1 MATLAB Tutorial

1.1 Introduction

MATLAB is a widely used package for performing calculations, analysing results and visualising data. In this chapter, we discuss the main features of MATLAB, and indicate how it might be useful to you in a research environment.

One way to think of MATLAB is as a versatile calculator. Its features include:

- Basic calculations: addition, subtraction, multiplication, and division.
- Scientific calculations: trigonometric functions, complex numbers, square roots and powers, and logarithms.
- Storage and retrieval of data.
- The ability to write custom functions and link to C or FORTRAN programs.
- Sophisticated plotting and data visualisation in two and three dimensions.
- A full suite of built-in functions to handle matrix calculations.

MATLAB provides an efficient way to develop software applications, process data from experimental apparatus, or analyse computational simulations. The environment is user friendly, and the programming language is much easier to use than writing C or FORTRAN code to analyse results. We believe that this high-level approach allows the user to concentrate on "what" rather than "how".

The most recent version of Matlab is version 5, which provides a number of enhancements over 4.2, and important changes are noted in this booklet.

A number of specialized toolboxes for MATLAB exist. These are written by experts in a particular field and offer suites of additional routines in fields as diverse as signal processing, optimization, neural network simulation and symbolic mathematics.

In this chapter we discuss in detail most of MATLAB core functions (including all the functions discussed in Chapter 5 of the MATLAB User's Guide). We have aimed to give insight into "why" you might use MATLAB, rather then simply saying "how". Included is some advanced material, such as a numerical analysis section, and a section discussing how to write efficient MATLAB code. In our examples, we have tried to use built-in data-files, so that the material can be followed without the need for excessive typing. Throughout the chapter we have included screen output as it would appear if you typed in the examples in MATLAB. The `Courier` typeface is used to denote MATLAB input and output. The output lines are what will appear when you type in our examples and press Enter. They often start thus:

```
ans =
    (some numbers)
```

You do not need to type these parts in.

We have provided an Appendix (at the end of the booklet) in which we give an alphabetical listing of most of the MATLAB commands. We refer to this as the 'reference appendix' throughout this chapter.

Although we will not have time to cover all the material in this chapter during the lectures, nor have we had time to discuss every MATLAB function in this tutorial, we hope that you

will be able to find out the rest from the excellent online help system. This is accessed by typing `help <command>` at the MATLAB command prompt (UNIX), or using the Windows help system.

```
help cos
 COS    Cosine.
    COS(X) is the cosine of the elements of X.


 Overloaded methods
    help sym/cos.m
```

If you have not used MATLAB before, we suggest that you concentrate on the earlier chapters first. If you are already familiar with MATLAB, we have included some material in the later chapters, which we hope will interest you.

1.2 Basic Features

The command window is the primary place where you interact with MATLAB. Commands are typed in at the prompt: >> (or EDU>> for the student version). As MATLAB runs, additional windows are opened for the output of plots. To start MATLAB under Windows, double click on the MATLAB icon. In UNIX type `matlab` (having ensured that any system specific file locations are in your current path). To exit MATLAB, either type `quit`, or use (under Windows) the 'Exit MATLAB' option from the File Menu.

To interrupt a MATLAB computation, press `ctrl + c`.

1.2.1 Simple Mathematics

As with many modern calculators, commands are typed in as you would write them down:

```
3*3+4*3+5*10
ans =
    71
```

MATLAB does not care about spaces in calculations, and precedence follows the normal order:

"Expressions are evaluated from left to right with the power operation having the highest order of precedence, followed by both multiplication and division having equal precedence, followed by both addition and subtraction having equal precedence. Parentheses can be used to alter this usual ordering, in which case evaluation initiates within the innermost parentheses and proceeds outward." Here some examples that demonstrate this:

```
8^2 - 3 - 5/2*2
ans =
    56
400/40/(3+2)
ans =
     2
(4+3)^2-12
ans =
    37
```

In each case, MATLAB defines a variable `ans` (short for answer) for the result of the last computation. It is also possible to define variables yourself, and perform calculations with them:

```
x=3
y=4
z=5*x+2*y
x =
     3
y =
     4
z =
    23
```

Here we assigned the value of $5x + 2y$ to the variable `z`. In the above cases, MATLAB returns the current assignment value on the next line. This can be suppressed by adding a semi-colon at the end of the input line. This is useful for intermediate calculations.

```
x=3;
y=4;
z=5*x+2*y
z =
    23
```

Each of the assignment lines is evaluated, but the answer is not returned. However the variable of interest, *z*, is returned, since we did not append a semi-colon to the line. A semi-colon can also be used to put multiple statements on a line:

```
x=10*3 ; y= 6*5 ; z=x/y
z =
     1
```

To put multiple statements on a line, but show some intermediate results, a comma is used:

```
x=10*3 , y= 6*5 ; z=x / y
x =
    30
z =
     1
```

We can also use the left division operator: `\`, which follows the convention of dividing the number above the slash by the number below it:

```
z = x \ y
z =
     1
```

If a statement is too long, an ellipsis consisting of three periods(...) followed by Enter indicates that the statement continues on the next line. This feature is useful only to ensure that all of the commands can be seen in the available screen window. Most implementations will wrap the line automatically.

```
z=5*x +...
2*y
z =
```

```
   210
```

MATLAB remembers past information, once a variable is stored, it is kept in memory until it is reassigned, or cleared (note that variables may be saved on or loaded from disk). This enables us to modify calculations easily:

```
z=10*x + 2*y
z =
   360
```

Indeed we implicitly used this in the last example- we did not have to re-assign `x` and `y` in order to determine a new value of `z`.

Early we said that spaces did not matter in calculations, however variable names may not contain spaces. It is general to use and underscore to join two words to form a single word:

```
long_name=3*z
long_name =
        1080
```

See the reference section for other basic MATLAB maths operations.

1.2.2 The MATLAB working environment

Before proceeding further, there are one or two important details to discuss about the MATLAB environment. During a MATLAB session, MATLAB remembers not only variables, but also a history of previous commands used (just like some UNIX shells). By pressing the arrow up, or down key (↑, or ↓), one can scroll through the list of previous commands. They can be edited using ← or → and delete or backspace. It is not necessary to move to the end of the line to press return to re-evaluate an expression- simply press return when any edits are completed.

MATLAB also allows one to intelligently scroll through the history of commands. If, for example, you made an assignment to `x` and subsequently various other assignments, by typing `x` and then arrow up (↑), MATLAB will only show those lines which start with an `x`. It is possible to type several characters to narrow the search down further. If no command matching the characters you type, MATLAB displays the last command in the history list.

MATLAB 5 incorporates a built-in editor and debugger (Windows 95 and Mac versions only), which highlights and indents Matlab code, which allows code to be stepped through line by line. It is also now possible to profile the performance of MATLAB code.

If you want to recall the names of variables you have defined, use

```
who
Your variables are:

D           ans         d           x
E           b           long_name   y
a           c           pi          z
```

To get a fuller description of variables use:

```
whos
  Name            Size         Bytes  Class

  D               1x4             32  double array
```

```
  E               1x6              48  double array
  a               1x1               8  double array (logical)
  ans             1x1               8  double array
  b               1x1               8  double array (logical)
  c               1x1               8  double array (logical)
  d               1x1               8  double array (logical)
  long_name       1x1               8  double array
  pi              1x1               8  double array
  x               1x1               8  double array
  y               1x1               8  double array
  z               1x1               8  double array

Grand total is 20 elements using 160 bytes
```

Each element stored takes 8 bytes, and the total memory used is shown. The other headings in this table will be explained in later sections, when we come to discuss complex numbers and defining arrays. To recall the value of a variable, type its name at the command prompt

```
long_name
long_name =
        1080
```

1.2.3 Saving and Retrieving Data

As we mentioned earlier, it is possible to save to and load from files. In the Windows version of the program, use the Save Workspace as... menu option from the File menu. This opens a standard dialog box; the default file extension is `.mat`. The Load Workspace as... option works analogously. Under UNIX, it is necessary to type

```
save my_data.mat
```

There are other options for saving (and loading), such as saving (loading) specific variables, and altering the format of the save (load). Using `save my_data.txt -ascii` it is possible to write data out into a form that can be read by a text editor or spreadsheet program. Similarly `load my_data.txt -ascii` allows such data to be read into a single array (`my_data`) in Matlab.

1.2.4 Display Precision

MATLAB always keeps results in memory to full precision (around 16 digits). However it is usually best to display results with fewer digits due to constraints on screen space (later we will be looking at matrices; the screen will rapidly scroll by if you display a 100×100 matrix with each element to 16 digit accuracy).

By default, MATLAB displays to around five digits accuracy, removing any trailing zeros. In the table below we show the effect of various `format` options on the following assignment:

```
format short; a=43+1/3
a =
   43.3333
```

Format Statement	Effect	Result

`format short`	5 digits	`43.3333`
`format long`	16 digits	`43.33333333333334`
`format short e`	5 digits plus exponent	`4.3333e+001`
`format long e`	16 digits plus exponent	`4.333333333333334e+001`
`format hex`	hexadecimal	`4045aaaaaaaaaab`
`format bank`	2 decimal digits	`43.33`
`format +`	displays positive, negative, or zero	`+`
`format rat`	rational approximation	`130/3`

1.2.5 Variable names

Up to now we have not discussed any restrictions, however there are a few rules about valid variable names. Variable names are case sensitive, should not containing spaces, and must start with a letter. They cannot contain punctuation symbols and symbols after the 19th are ignored.

MATLAB also defines some special variables: `ans`, `pi`, `eps`, `inf`, `NaN`, `i`, `j`, `realmin`, `realmax`, which may be used for convenience. Although you may redefine these, they will not be ready 'for convenience' if you do this. We suggest that you avoid these variable names, and also any others that appear as function names in Reference Guide (see the appendix). If you *have* to use one of these names, you could always add a number suffix.

In MATLAB, simply setting a variable to a new value will redefine it:

```
x=3, x=10
x =
     3
x =
    10
x*20
ans =
   200
```

In the above example the variable `x` takes on the last value assigned to it. Should you wish to remove a variable completely from the workspace, it may be cleared

```
clear x
x
??? Undefined function or variable 'x'.
```

If you accidentally define a variable to one of the special names, or the name of a function or command, then `clear` will restore its original value or operation:

```
pi=4.3
pi =
    4.3000
clear pi
pi
ans =
    3.1416
```

To reset all variables use `clear` on its own

```
who
Your variables are:

D           ans         d           z
E           b           long_name
a           c           y
clear
who
```

You may only need to use `clear` to free up memory when handling large amounts of data.

1.3 Mathematical Functions

MATLAB has many built-in common functions used in science and engineering. The reference appendix at the end of this booklet has a list of these, most of them are abbreviated in the standard mathematical way.

```
angle_rad = cos(pi/3)
angle_rad =
    0.5000
acos(angle_rad)
ans =
    1.0472
```

Note that MATLAB uses radians for all angles.

```
sqrt(3)+sqrt(5)
ans =
    3.9681
```

1.3.1 Complex Numbers

To illustrate the use of complex numbers, consider the solutions to the quadratic equation $ax + bx + c = 0$.

```
a=2;b=2;c=-5;
x1=(-b+sqrt(b^2-4*a*c))/(2*a)
x1 =
    1.1583


x2=(-b-sqrt(b^2-4*a*c))/(2*a)
x2 =
   -2.1583
```

(If I had been entering these equations, I would have used the arrow up key (↑) and edited the sign in front of the square root).

This quadratic will have complex roots, if we try to take the square root of a negative number:

```
a=3;b=3;c=3;
x1=(-b+sqrt(b^2-4*a*c))/(2*a)
x1 =
  -0.5000+ 0.8660i
```

```
x2=(-b-sqrt(b^2-4*a*c))/(2*a)
x2 =
  -0.5000- 0.8660i
```

The `i` denotes the square root of -1, and the solution is complex. `j` can also be used, which is more common in engineering. The next example shows the ease of using complex numbers; in MATLAB arithmetic in complex numbers proceeds transparently:

```
z1=4-4i
z1 =
   4.0000- 4.0000i
z2=2-2j
z2 =
   2.0000- 2.0000i
z3=sqrt(3)*i
z3 =
        0+ 1.7321i
z1+z2
ans =
   6.0000- 6.0000i
z1*z2
ans =
        0-16.0000i
z2/z3
ans =
  -1.1547- 1.1547i
```

We need to insert a `*` to a complex part of `sqrt(3)`, since `sqrt(3)i` has no meaning in MATLAB. It is possible to extract the real or imaginary parts from complex numbers:

```
real(z1*z2)
ans =
     0
imag(z1*z2)
ans =
   -16
```

It is possible to convert Cartesian complex numbers to their polar form using

```
z1_length=abs(z1)
z1_length =
    5.6569
z1_ang_rad=angle(z1)
z1_ang_rad =
   -0.7854
```

This angle could also be presented in degrees

```
z1_ang_deg=angle(z1)*180/pi
z1_ang_deg =
   -45
```

1.4 Help

Earlier we mentioned the online help. By now you will probably have many things, which you would like to know whether MATLAB can help you with. In the Windows version of MATLAB there is the usual windows interactive help system driven by clicking on hypertext document links. Help can also be obtained (under, for example, UNIX) by using `help command`.

```
help real
 REAL   Complex real part.
    REAL(X) is the real part of X.
    See I or J to enter complex numbers.

    See also ISREAL, IMAG, CONJ, ANGLE, ABS.

 Overloaded methods
    help sym/real.m
```

At the end of each help summary, there is a list of related commands. This is fine if you know what you are looking for (the reference appendix should help to locate many commands), however this is not always the case. Typing `help` on its own yields a list of directories with MATLAB commands in

```
help
HELP topics:

matlab\general       -  General purpose commands.
matlab\ops           -  Operators and special characters.
matlab\lang          -  Programming language constructs.
matlab\elmat         -  Elementary matrices and matrix manipulation.
matlab\elfun         -  Elementary math functions.
matlab\specfun       -  Specialized math functions.
matlab\matfun        -  Matrix functions - numerical linear algebra.
matlab\datafun       -  Data analysis and Fourier transforms.
matlab\polyfun       -  Interpolation and polynomials.
matlab\funfun        -  Function functions and ODE solvers.
matlab\sparfun       -  Sparse matrices.
matlab\graph2d       -  Two dimensional graphs.
matlab\graph3d       -  Three dimensional graphs.
matlab\specgraph     -  Specialized graphs.
matlab\graphics      -  Handle Graphics.
matlab\uitools       -  Graphical user interface tools.
matlab\strfun        -  Character strings.
matlab\iofun         -  File input/output.
matlab\timefun       -  Time and dates.
matlab\datatypes     -  Data types and structures.
matlab\dde           -  Dynamic data exchange (DDE).
matlab\demos         -  Examples and demonstrations.
```

```
toolbox\symbolic     -  Symbolic Math Toolbox.
toolbox\signal       -  Signal Processing Toolbox.
toolbox\control      -  Control System Toolbox.
control\obsolete     -  (No table of contents file)
toolbox\local        -  Preferences.

For more help on directory/topic, type "help topic".
```

The exact display may differ from the above. Typing help on one of these entries yields more topics

```
help timefun
  Time and dates.

  Current date and time.
    now         - Current date and time as date number.
    date        - Current date as date string.
    clock       - Current date and time as date vector.

  Basic functions.
    datenum     - Serial date number.
    datestr     - String representation of date.
    datevec     - Date components.

  Date functions.
    calendar    - Calendar.
    weekday     - Day of week.
    eomday      - End of month.
    datetick    - Date formatted tick labels.

  Timing functions.
    cputime     - CPU time in seconds.
    tic, toc    - Stopwatch timer.
    etime       - Elapsed time.
    pause       - Wait in seconds.
```

This search displays matches between a given a keyword and words in the first line of the help summaries. It can take some time to complete, if you have a large MATLAB system with many additional files.

```
lookfor random
RAND   Uniformly distributed random numbers.
RANDN  Normally distributed random numbers.
RANDPERM Random permutation.
RJR    Random Jacobi rotation.
SPRAND Sparse uniformly distributed random matrix.
SPRANDN Sparse normally distributed random matrix.
SPRANDSYM Sparse random symmetric matrix.
```

```
DRMODEL Generates random stable discrete nth order test models.
DRSS  Generate random stable discrete-time state-space models.
RMODEL Generates random stable continuous nth order test models.
RSS   Generate random stable continuous-time state-space models.
```

Using the above information, we could determine how to generate random numbers by typing `help rand`.

MATLAB 5 provides over 4000 pages of additional information in html files and .pdf documents (portable document file). These are provided in the Student version. Many features, such as building graphical interfaces and linking to C/ Fortran code are described in great detail in these documents. You will need the latest version of ghostview, or an adobe acrobat reader to view these files.

1.5 Array Operations

So far, all of our examples have used scalar variables. However it is the ability to handle arrays which makes MATLAB particularly powerful. MATLAB 5 permits the use of multidimensional arrays. We will deal initially with one or two dimensional arrays, which are most generally used in linear algebra before returning at the end of the section to multidimensional arrays.

1.5.1 Array Assignments: Rows

Suppose we want to evaluate 5 separate values of `cos(x)`. Naturally, we could use the `cos` function five times, however by assigning an array with the values of interest, this operation can be completed in two steps

```
x=[0 , 0.1*pi, 0.3*pi, 0.5*pi, pi];
y=cos(x)
y =
    1.0000    0.9511    0.5878    0.0000   -1.0000
```

To create the matrix we start with a normal variable assignment, but enclose a list of values in brackets, `[` and `]`, and separate each one with a comma. (Although spaces can be used to separate entries, we strongly advise against this due to possible ambiguities related to embedded space.) We refer to these as elements `x(1)`, `x(2)`, etc.

```
x(2)
ans =
    0.3142
```

MATLAB takes the `cos` of each of the elements of `x` and stores them in the corresponding value of `y`. We now have

```
y(2)
ans =
    0.9511
```

1.5.2 Array Addressing : Colon Notation I

In the above example we have created arrays `x` and `y` with one row and 5 columns, and we accessed individual elements of the array using their subscripts, `x(1)`, `x(2)`, etc. It is also possible to access groups of elements using the colon notation:

```
x(1:5)
```

```
ans =
         0    0.3142    0.9425    1.5708    3.1416
```

This returns the elements of x with subscripts between 1 and 5 inclusive. In general, we can pick out certain columns using

```
x(1:2:5)
ans =
         0    0.9425    3.1416
```

This returns elements 1+0×2, 1+1×2, 1+2×2, and literally means "return column 1 and the next but one column, up to column 5." It is equivalent to

```
[x(1), x(3), x(5)]
ans =
         0    0.9425    3.1416
```

The counting increment can even be negative:

```
x(5:-1:2)
ans =
    3.1416    1.5708    0.9425    0.3142
```

We have reversed the order of the array: literally "return column5, and then every previous column until column 2".

If you want to pick out particular columns in a certain order, which do not conform to the above method, it is also possible to specify an array of subscripts to return.

```
z=y([3 1 4 5])
z =
    0.5878    1.0000    0.0000   -1.0000
```

Now we have assigned `z(1) = y(3)`, `z(2) = y(1)`, `z(3)=y(4)`, `z(4)=y(5)`. If you try to access an array subscript larger than one that exists, an error message is returned:

```
z(13)
??? Index exceeds matrix dimensions.
```

1.5.3 Array Construction: Colon notation II

Suppose we want to sample the `cos` function over one period, from 0 to 2π. We could type in a vector of values to sample, and then take the cosine of this vector. However, we can use the colon notation of the previous section to achieve this much more easily. Let us look at the following example:

```
x=(1:2:10)
x =
     1     3     5     7     9
```

Here we have assigned the odd integers between 1 and 10 to the variable `x`. The colon notation also generalizes to allow non integer steps between the first and last values

```
x=(1:0.1:2)
x =
  Columns 1 through 7
    1.0000    1.1000    1.2000    1.3000    1.4000    1.5000    1.6000
  Columns 8 through 11
```

```
    1.7000    1.8000    1.9000    2.0000
```

In general we have the form `(start: step: end)`, in which any of the three variables can take on any real value. Thus to sample the `cos` function between 0 and 2π we use

```
x = (0:0.4:2*pi)
x =
  Columns 1 through 7
         0    0.4000    0.8000    1.2000    1.6000    2.0000    2.4000
  Columns 8 through 14
    2.8000    3.2000    3.6000    4.0000    4.4000    4.8000    5.2000
  Columns 15 through 16
    5.6000    6.0000


y = cos(x)
y =
  Columns 1 through 7
    1.0000    0.9211    0.6967    0.3624   -0.0292   -0.4161   -0.7374
  Columns 8 through 14
   -0.9422   -0.9983   -0.8968   -0.6536   -0.3073    0.0875    0.4685
  Columns 15 through 16
    0.7756    0.9602
```

We often want to sample a function a certain number of times between two end values. Suppose we want 10 samples between 0 and 2π we could use

```
x = (0: (2*pi-0)/9 : 2*pi)
x =
  Columns 1 through 7
         0    0.6981    1.3963    2.0944    2.7925    3.4907    4.1888
  Columns 8 through 10
    4.8869    5.5851    6.2832
```

However MATLAB gives us a simple way to do this

```
x = linspace(0,2*pi,10)
x =
  Columns 1 through 7
         0    0.6981    1.3963    2.0944    2.7925    3.4907    4.1888
  Columns 8 through 10
    4.8869    5.5851    6.2832
```

In MATLAB, there are often several ways to perform the same task. A similar function exists to produce logarithmically spaced vectors

```
x=logspace(0,2,7)
x =
    1.0000    2.1544    4.6416   10.0000   21.5443   46.4159  100.0000
```

Here we start at 10^0 and produce 7 logarithmically spaced vectors up to 10^2. The general syntax is `logspace(start exponent, end exponent, number of values)`, and non-integer exponents are acceptable.

When you require an array that is not easy to build up using the above methods, you may be forced to type in the values individually. However, a little thought can sometimes save effort. Suppose we want a list of numbers 1, 2, 3, 4, 5, 6, 5, 4, 3, 2, 1. We can combine the methods above:

```
part_1 = (1:1:5)
part_1 =
     1     2     3     4     5
part_2 = (5:-1:1)
part_2 =
     5     4     3     2     1
x = [part_1, 6, part_2]
x =
     1     2     3     4     5     6     5     4     3     2     1
```

Can you do this using only one temporary array assignment? Yes

```
part_3 = (1:5)
part_3 =
     1     2     3     4     5


x = [part_3, 6, part_3(5:-1:1)]
x =
     1     2     3     4     5     6     5     4     3     2     1
```

Although running out of memory is unlikely to be an issue in simple examples, there is no need to be profligate. This last example uses techniques from this section (1.5.3) and the previous one (1.5.2):

- Colon notation to create a linearly spaced array.
- Colon notation to generate a set of subscripts to address an array.

1.5.4 Array Assignments: Columns

Up to now we have considered matrices with only one row, in which elements are separated by commas. To specify row breaks we use a semi-colon:

```
a = [1, 2, 3; 4, 5, 6; 7, 8, 1]
a =
     1     2     3
     4     5     6
     7     8     1
```

This defines a 3×3 matrix. We can also use press `return` at the end of a row, when entering a matrix:

```
a = [1, 2, 3
4, 5, 6
7,8, 1]
```

```
a =
     1     2     3
     4     5     6
     7     8     1
```

To convert a row matrix to a column matrix, we use the transpose operator (`'`). You can create a linearly spaced column vector using

```
a=(1:5)'
a =
     1
     2
     3
     4
     5
```

In summary of 1.5.1 and 1.5.4: we use a **comma** to separate elements of a matrix on the same row, and a **semi-colon** to specify a new column. In a matrix, all rows must be defined with the same number of columns.

1.5.5 Scalar-Array Calculations

Now we can construct arrays, we will look at scalar-array calculations in which a single number acts on the whole of the array. Consider

```
p = (1:2:10)
p =
     1     3     5     7     9
```

To multiply all the elements by, for example, 3, we use:

```
q = 3*p
q =
     3     9    15    21    27
```

We could derive a list of even numbers between 1 and 10 using

```
q = p+1
q =
     2     4     6     8    10
```

If we combine operations, the order of precedence for scalar-array computations is the same as the scalar-scalar case.

```
q = 4 * p - 10 / 2
q =
    -1     7    15    23    31
```

Here we have subtracted `5` (= `10 / 2`) from `4*p`. We can also perform the same operations on matrices with multiple row and columns

```
a = [1, 2, 3; 4, 5, 6; 7, 8, 1]
a =
     1     2     3
     4     5     6
     7     8     1
```

```
a+4
ans =
     5     6     7
     8     9    10
    11    12     5
```

1.5.6 Array-Array Calculations

In this section we consider how array-array computations work. Firstly we consider element-wise calculations, and then consider conventional linear-algebra matrix calculations. To perform a computation **element-wise** between the elements of two matrices, you must ensure that the dimensions of the matrices are the **same**.

```
a=[1, 2; 3, 4] ; b=[3, 5; 2, 1] ;
a+b
ans =
     4     7
     5     5


a-b
ans =
    -2    -3
     1     3
```

We specify an element-wise operation by using the **dot operator** (`.`) in front of the operation to be performed (this is not necessary for addition and subtraction, since the rules of linear algebra always perform these operations element-wise). There is no space between the dot and the operation.

```
a .* b
ans =
     3    10
     6     4
a ./ b
ans =
    0.3333    0.4000
    1.5000    4.0000
a .\ b
ans =
    3.0000    2.5000
    0.6667    0.2500
```

Note that element-wise division (like scalar division) can use either a forward or backward slash, with the elements above the slash being divided by the element below the slash. So `b .\ a` is the same as `a ./ b`. We can also consider raising the elements of an array to a given power

```
a = [1, 3, 5, 7]
a =
```

```
     1     3     5     7
a .^3
ans =
     1    27   125   343
```

Here we have cubed each element in the array. We can also raise a number to the power of each element in an array:

```
2 .^ a
ans =
     2     8    32   128
```

Given two matrices with the same dimensions, we can raise each element of one to the power of the corresponding element in the other

```
a = [1, 6, 3, 2]; b = [10, 3, 2, 8];
a.^b
ans =
     1   216     9   256
```

To perform normal multiplication, we omit the dot. According to the rules of linear algebra, if `A` is a `k*l` matrix, and `B` is `m*n`, the we must now ensure that `l = m`. The result is a matrix of dimension `k*n`.

```
a = [1, 2, 3; 4, 5, 6], b = [2, 3; 5, 6; 5, 9]
a =
     1     2     3
     4     5     6
b =
     2     3
     5     6
     5     9
a*b
ans =
    27    42
    63    96
```

Since taking powers of a matrix is identical to repeatedly multiplying a matrix by itself, you can only take powers of square matrices.

```
c = [3, 4; 2, 10]; c^2
ans =
    17    52
    26   108
```

Conventionally division of matrices is not defined. One always thinks of dividing `A` by `B` as multiplying `A` by the inverse of `B`. This is exactly the case in MATLAB, except it is calculated by a numerically stable method by solving the set of equations `Ax=B` by Gaussian elimination.

```
a = [1, 2; 4, 5], b = [2, 4; 5, 6]
a =
     1     2
     4     5
b =
```

```
     2     4
     5     6
a / b
ans =
    0.5000         0
    0.1250    0.7500
a*inv(b)
ans =
    0.5000         0
    0.1250    0.7500
```

We can also use the left division `\` which divides the matrix above the slash by the matrix below the slash:

```
a \ b
ans =
         0   -2.6667
    1.0000    3.3333
```

In general the results of matrix division do not give results the same size as `A` and `B`. If the set of equations `Ax=B` is over-determined, then the solution given is the least-squares for `x` (see the reference appendix for more details).

1.5.7 The transpose operator

As we mentioned earlier, we take the transpose of a row matrix to give a column matrix. In general we can apply it to matrices, as in standard linear algebra

```
a = [4, 3; 2, 9]
a =
     4     3
     2     9
a'
ans =
     4     2
     3     9
```

If we can use the operation element-wise, the result is the same for real matrices

```
a.'
ans =
     4     2
     3     9
```

If the matrix has complex entries then the transpose is in fact the complex conjugate transpose, so each element `a(i,j)` becomes `real(a(j,i)) - imag(a(j,i))`.

```
z = [1+i, 2-2i; 1+sqrt(3)*i, sqrt(7) + 4*i]
z =
   1.0000+ 1.0000i   2.0000- 2.0000i
   1.0000+ 1.7321i   2.6458+ 4.0000i
z'
ans =
```

```
   1.0000- 1.0000i   1.0000- 1.7321i
   2.0000+ 2.0000i   2.6458- 4.0000i
```

If we use the operation element-wise

```
z.'
ans =
   1.0000+ 1.0000i   1.0000+ 1.7321i
   2.0000- 2.0000i   2.6458+ 4.0000i
```

Then we obtain the transpose of `z`, without conjugation of the elements.

1.5.8 Multidimensional Arrays

Often data takes the form of a multidimensional array. MATLAB 5 extends MATLAB to allow an arbitrary number of dimensions in an array. A three-dimensional array may be formed by stacking a number of two-dimensional arrays:

```
a = [1, 3; 8, 9];
b = [2, 4 ; 6, 8];
c = [3, 5; 7, 0];
d = cat(3, a, b, c)
d(:,:,1) =
     1     3
     8     9
d(:,:,2) =
     2     4
     6     8
d(:,:,3) =
     3     5
     7     0
```

1.6 Handling Arrays

In the last section, we considered how to construct an array, and perform basic matrix arithmetic. In this section, we look at MATLAB's ability to re-arrange matrices and access subsets of the matrix.

1.6.1 Accessing Array Subsets and Re-ordering Arrays: Colon Notation IV.

When we discussed the colon notation in 1.5.2, we rearranged the columns of a matrix. In general, we can build one matrix from another by specifying the rows and columns to take

```
a = [1, 2, 3, 4; 5, 6, 7, 8; 9, 10, 11, 12; 13, 14, 15, 16]
a =
     1     2     3     4
     5     6     7     8
     9    10    11    12
    13    14    15    16
b = a(1:2, 3:4)
b =
```

```
     3     4
     7     8
```

Here we have taken out the upper corner 2×2 submatrix from `a`. To reverse order of the rows of `a`, we could use

```
b = a(4: -1: 1, :)
b =
    13    14    15    16
     9    10    11    12
     5     6     7     8
     1     2     3     4
```

Note that we can use the colon on its own as a shorthand for 1:4, meaning "take all the columns."

We can build up a matrix by repeating sections of another matrix:

```
b = [ a(:,2), a(:,1), a(:,2), a(:,1)]
b =
     2     1     2     1
     6     5     6     5
    10     9    10     9
    14    13    14    13
```

This produces a matrix made up of columns 2 and 1 of matrix `a` repeated twice. It is equivalent to

```
b = a(:, [2, 1, 2, 1])
b =
     2     1     2     1
     6     5     6     5
    10     9    10     9
    14    13    14    13
```

where we have used a vector subscript to address the columns of `a`. MATLAB often allows several ways for a problem to be solved. The colon notation gives much scope for different techniques to be used.

1.6.2 Re-assigning array elements

Suppose we have a matrix already defined, but want to change an individual element of the matrix:

```
a = [1, 2, 3; 4, 5, 6; 7, 8, 1]
a =
     1     2     3
     4     5     6
     7     8     1
```

To change an existing element, we use:

```
a(1,1) = 66
a =
    66     2     3
```

```
     4     5     6
     7     8     1
```

If you try to change a matrix element which is outside the current bounds for this array, MATLAB will augment the matrix with sufficient extra rows and columns to make the assignment. It sets the extra values in the matrix to 0.

```
a(4, 3) = 1
a =
    66     2     3
     4     5     6
     7     8     1
     0     0     1
```

To remove a whole row or column of a matrix, we assign to have a null value: `[]`. It is only possible to remove a whole column or row (if you try to remove an element, MATLAB would not know how to reform the array).

```
a(2, :) = []
a =
    66     2     3
     7     8     1
     0     0     1
```

We can also re-assign whole rows (or columns) of a matrix at once. In this example we replace the second column of `a` with `b`:

```
a = [1 2; 3 1], b = [10; 9]
a =
     1     2
     3     1
b =
    10
     9


a(:,2)=b
a =
     1    10
     3     9
```

1.6.3 Reshaping arrays

If we index a matrix with a single colon, then MATLAB forms a column vector, by stacking the columns of the matrix on top of each other.

```
a = [2, 4, 6; 3, 5, 7;  1, 2, 3]
a =
     2     4     6
     3     5     7
     1     2     3
b=a(:)
b =
```

```
     2
     3
     1
     4
     5
     2
     6
     7
     3
```

It is also possible to reshape an array entirely. `reshape(a, m, n, p, ..)` returns an `m` by `n` by `p` (etc.) array from `a` with the elements taken column-wise. `a` must have m×n×p×… elements:

```
a = [1, 2, 3, 4; 5, 6, 7, 8; 9, 10, 11, 1]
a =
     1     2     3     4
     5     6     7     8
     9    10    11     1
b = reshape(a, 2, 6)
b =
     1     9     6     3    11     8
     5     2    10     7     4     1
```

1.6.4 How Big is a Matrix?

In the examples, we have created matrices, and so we know how large they are. If we load in data from disk into an array, this may not be the case. MATLAB provides the function `size` to enable you to determine the dimensions of an array.

```
a = [1, 2, 3; 4, 5, 6]
a =
     1     2     3
     4     5     6
d = size(a)
d =
     2     3
```

This returns the size of each dimension of `a` (in the order rows, columns, …) as the elements of the matrix `d`. For two-dimensional arrays there are two approaches to return the numbers of rows and columns in separate variables

```
[num_rows, num_cols] = size(a)
num_rows =
     2
num_cols =
     3


num_rows = size(a,1) , num_cols = size(a,2)
num_rows =
```

```
     2
num_cols =
     3
```

This extends to determine the size of multidimensional arrays.

1.6.5 Other Operations for Handling Arrays

- `flipud(a)` flips a matrix upside down.
- `fliplr(a)` flips a matrix left to right.
- `rot90(a)` rotates a matrix anticlockwise by 90º.

1.7 2-D plotting of functions

Easy and efficient visualization of data is one of the most important tools in engineering. Many packages exist which are optimized to view particular forms of data in two and three dimensions. Where available, a specialist package should be used, since it will almost always outperform a general package.

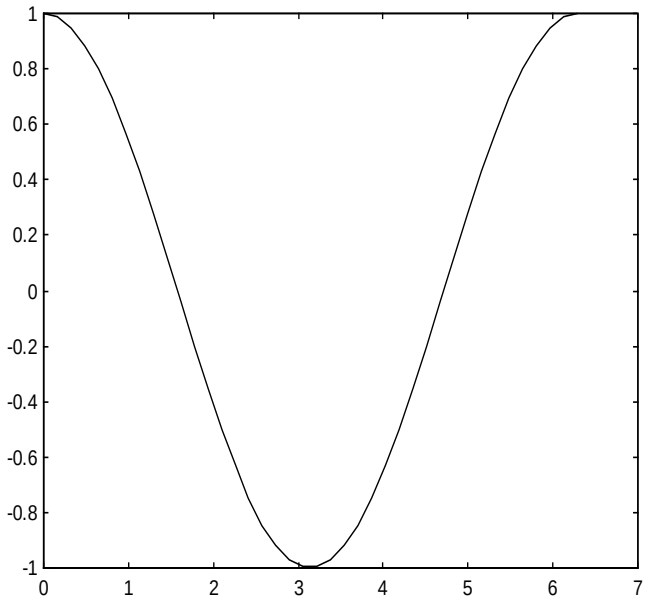
However, such packages often reside on dedicated machines, and more frequently we need a quick plot of some data or equations, to get an idea about how a simulation or experiment is progressing. Spending some time analyzing data *without having to write complex computer code* is often more productive than simply producing more and more results and serves to focus future research. It is in this area that we believe MATLAB is useful.

In this section we will consider how to produce two dimensional plots with a variety of line markers, colours, labels and customized views. We will then consider the different types of plot available, how to manipulate these plots, and finally how to produce hardcopy.

1.7.1 Simple use of the `plot` command

Earlier we produced a linearly spaced array of values of between 0 and 2π. We could use this to look at the cosine function over this range

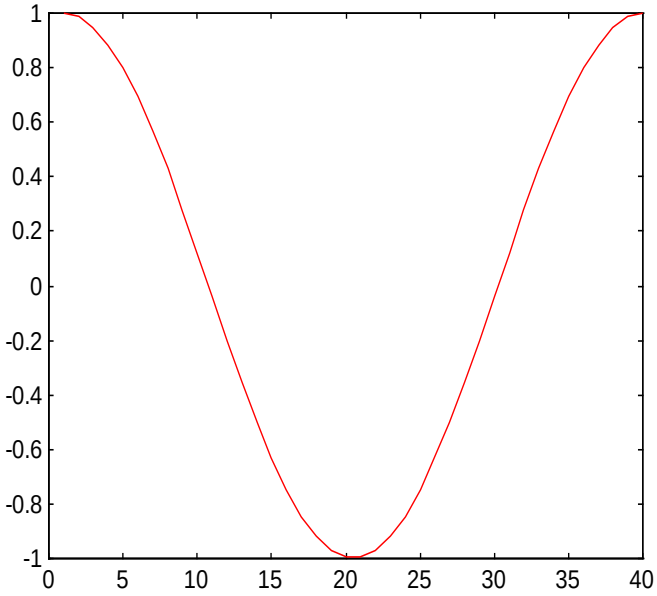
```
x = linspace(0, 2*pi, 40);
y=cos(x);
plot(x,y)
```



We have sampled the function `cos(x)` at 40 evenly spaced points between 0 and 2π, setting the `y` values to the cosine of each of the `x` values. When a plot command is issued, a **figure window** is opened and a plot is produced in which the individual points are connected with straight lines. In this case these were sufficient points to give the illusion of a smooth curve. The axes are scaled and tick marks are added automatically, although later we will see how to customize these. If a figure window is already open, `plot` replaces the old plot with the new one, clearing the window first.

If we omit to specify a set of `x` points, then MATLAB plots the vector value against its subscript:

```
plot(y)
```

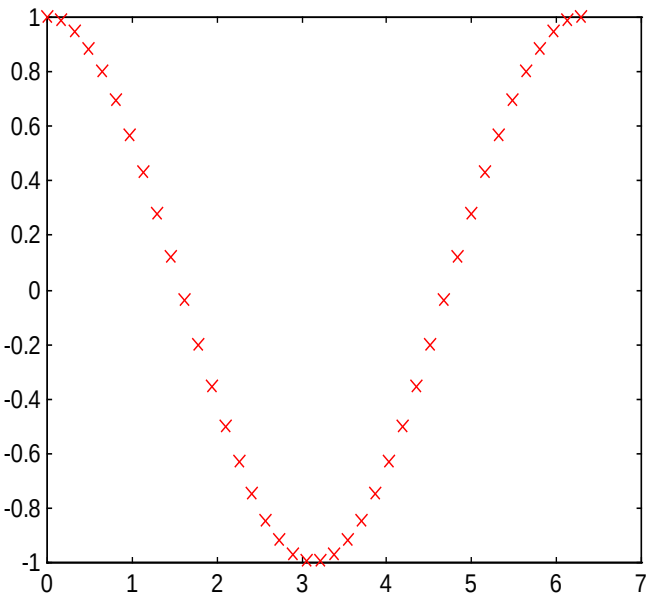


1.7.2 Linestyles, Markers and Colours

Naturally, it is possible to format lines on a plot. This will be particularly useful when we come to consider multiple plots in the next section. `plot(x, y, s)` plots `x` and `y` using a format `s` chosen from the below table. The format consists of a number of characters, which specify a colour, marker, and/ or line style. In MATLAB 5 there is a greater choice of markers for plots.

Symbol	Colour	Symbol	Marker	Symbol	Line Style
y	yellow	.	point	-	solid line
m	magenta	o	circle	:	dotted line
c	cyan	x	x-mark	-.	dash-dot line
r	red	+	plus	--	dashed line
g	green	*	star		
b	blue	s	square		
w	white	d	diamond		
k	black	v	triangle (down)		
		^	triangle (up)		
		<	triangle (left)		
		>	triangle (right)		
		p	pentagram		
		h	hexagram		

```
plot(x,y,'x')
```



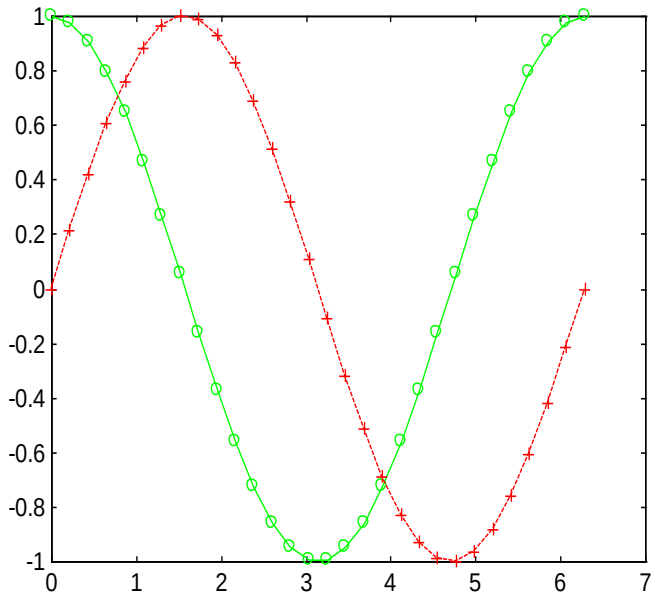
If no colour is specified, MATLAB cycles the colours from yellow to red (depending on other plots which may be present). In the next section we will see how to create multiple plots, and plots with lines and markers.

1.7.3 Multiple plots

In the previous section, we saw that the point styles did not connect up points on the graph. However it is possible to put multiple plots on the same axes, using `plot(x1, y1, s1, x2,`

`y2, s2, x3, y3, s3, ..)`, which enables us to put a plot marking the points, and a plot with a solid line on the same axes. We can also plot other functions, using a comma to separate the triples specifying the x and y vectors and the format statement.

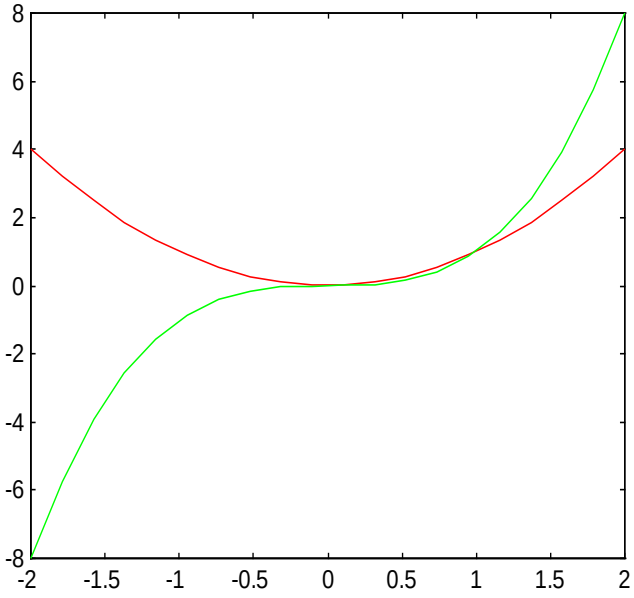
```
x= linspace(0, 2*pi, 30); y1= cos(x); y2= sin(x);
plot(x, y1, 'go', x, y1, 'g-', x, y2, 'r+', x, y2, 'r--')
```



(Note that we may abbreviate the above command to `plot(x, y1, 'go-', x, y2, 'r+--')`.)

If one of the x or y entry vectors is a matrix, then the vector is plotted versus the rows or columns of the matrix (whichever line up).

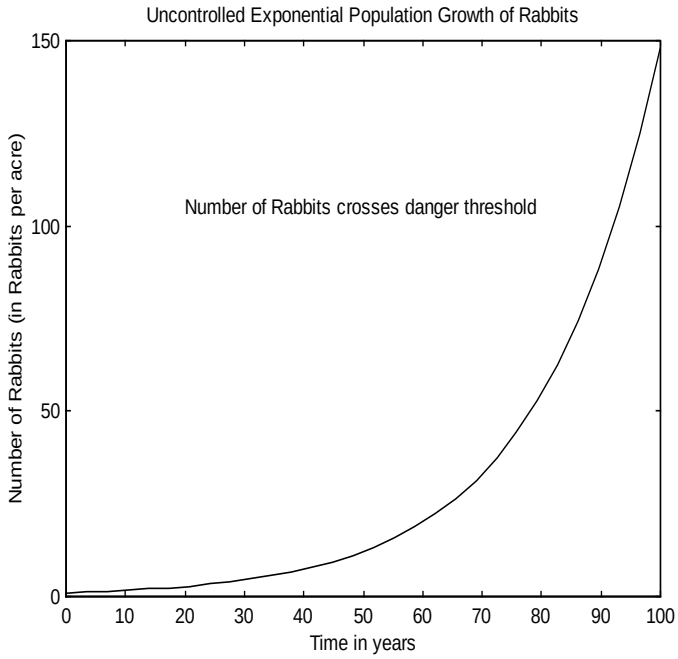
```
x = linspace(-2, 2, 20); a = [x.^2; x.^3];
plot(x,a);
```



1.7.4 Labels and Grids

It is important to label axes on a graph with units and a title. A grid is often added to a graph to ease visualisation. To add a label in a particular place on the graph, use `text(x, y, 'string')`, where `(x, y)` specifies the centre left edge of the text string, using the units from the grid axis.

```
grid on
x = linspace(0, 100, 30); plot(x, exp(0.05*x),'r');
xlabel('Time in years')
ylabel('Number of Rabbits (in Rabbits per acre)')
title('Uncontrolled Exponential Population Growth of Rabbits')
text(20,105,'Number of Rabbits crosses danger threshold')
```



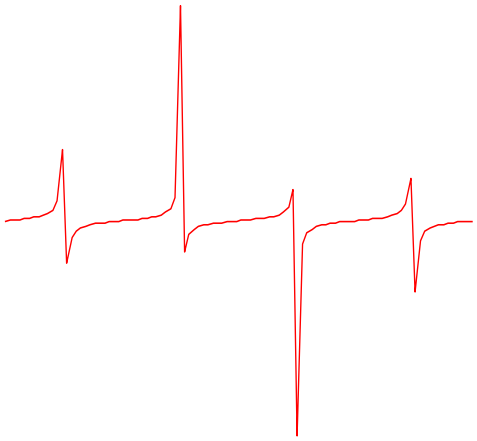
Using the grid we can see that the rabbit population roughly trebles in the last 20 years of the graph (from 80 to 100). The `grid on` command turns the grid on; `grid off` turns it off. `grid` on its own toggles the current state of the grid.

You can place text on the graph using a mouse with `gtext('string')`, when this command is issued, MATLAB switches to the current figure window, and the mouse becomes a cross-hair. When a button on the mouse or keyboard is pressed, the text is placed with the lower left corner of the first character at that location

1.7.5 Customising Axes

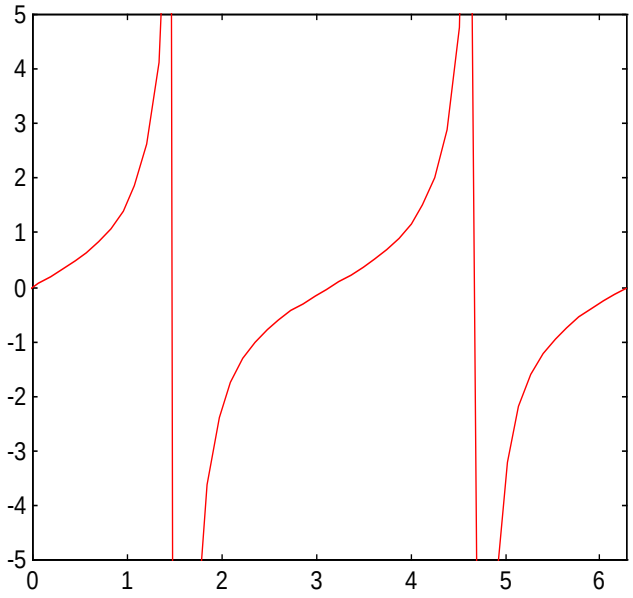
Sometimes it is necessary to override the MATLAB default views and formatting of the axes. The `axis` command allows you to vary the appearance and view of the axes. In its simplest form typing `axis on`, or `axis off` will turn the axes, grid and tick labels on and off respectively. Typing `axis` alone toggles the state of the current axes.

```
x = linspace(-2*pi, 2*pi, 100); y = tan(x);
plot(x,y); axis off;
```



Since the function `tan(x)` gets very large around `pi/2`, it may be best to restrict the view of the axis. To obtain a custom view of the graph use `axis([xmin, xmax, ymin, ymax])`, where the custom `x` axis runs from `xmin` to `xmax`, and the custom `y` axis runs from `ymin` to `ymax`:

```
plot(x,y); axis([0, 2*pi, -5, 5]);
```



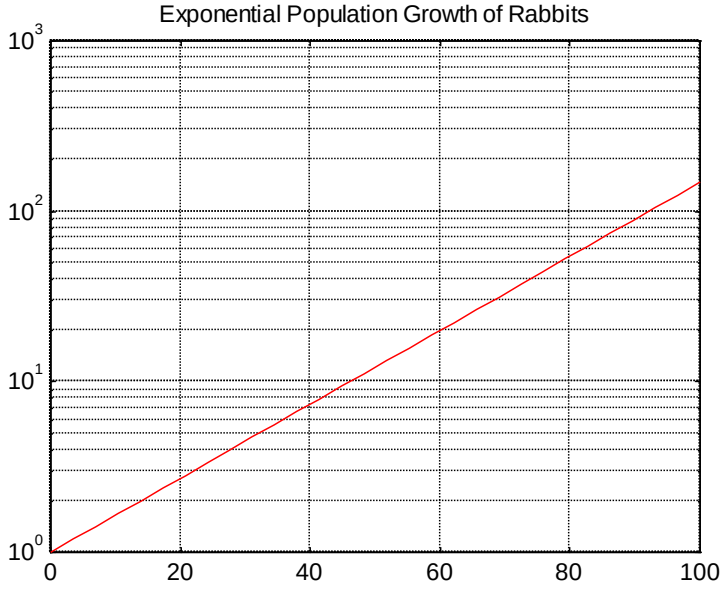
There are numerous options for this function, see `help axis` for further details. Here is a list of some of them

- `axis('auto')` or `axis auto` returns the plot to axis view to its default values.

- `axis('xy')` or `axis xy` uses the default **Cartesian** co-ordinate system, with (0,0) in the bottom left of the plot and `x` and `y` axes increasing to the right and up respectively.
- `axis('ij')` or `axis(ij)` uses **matrix** co-ordinates in which the system origin is at the top left corner, and the `x` and `y` axes increase to the right and bottom respectively.
- `axis('square')` or `axis square` sets the height-width ratio of the plot to be square, rather than rectangular.
- `axis('equal')` or `axis equal` sets both axis scaling factors to equal.
- `axis('normal')` or `axis normal` turns off `axis('square')` or `axis('equal')`.

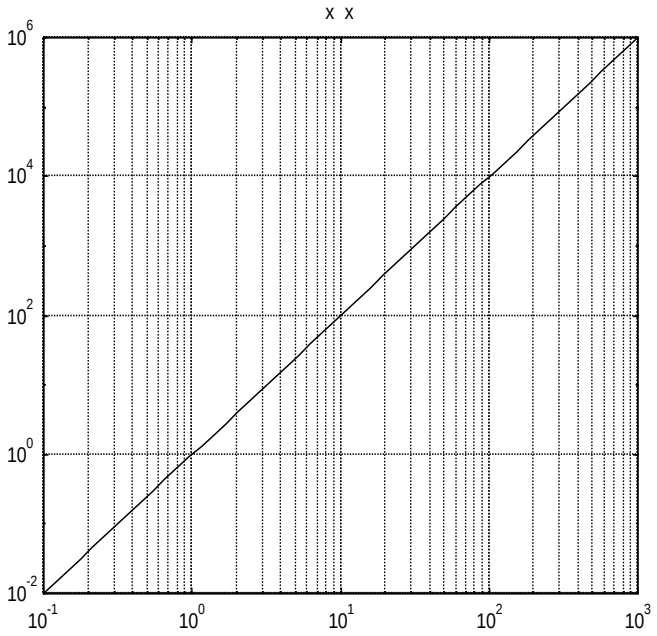
Not only can you change the axis view, but it is possible to specify base 10 logarithmic axes. In the case of our exponential population growth, we could have used:

```
x = linspace(0, 100, 30);
semilogy(x,exp(0.05*x));
title('Exponential Population Growth of Rabbits');
grid on
```



This view allows clearer measurements to be made from the graph in the region between 0 and 40 years. The plotting options for `semilogy` and `semilogx` (which gives a logarithmic scale on the `x` axis) are the same as for `plot`. If you want a logarithmic scale on both axes, then use `loglog(x,y,s)`, which again has the same options as the `plot` command.

```
x = logspace(-1, 3, 50);
loglog(x,x.*x);grid on;
title('x ^ x');
```



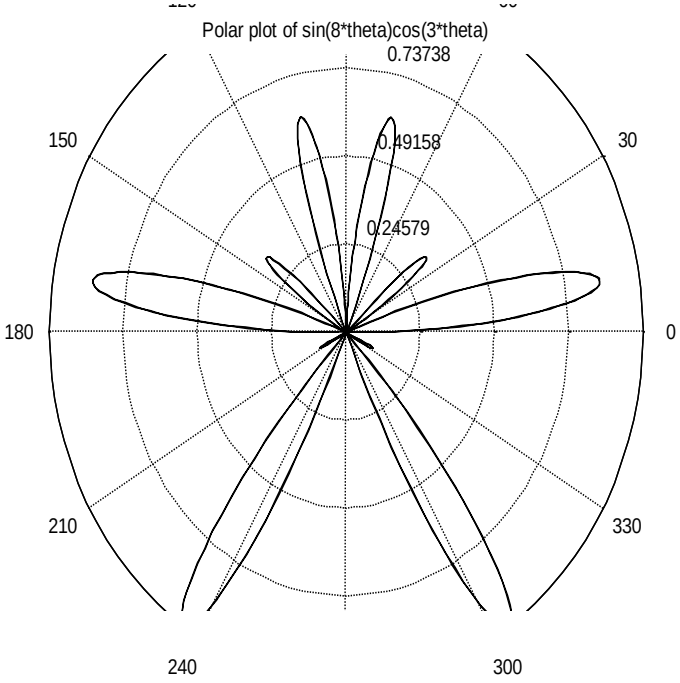
Logarithmic plots are very useful in digital signal processing, where one needs to produce graphs that span many orders of magnitude. Drawing these graphs by hand can be very time consuming.

1.7.6 A Multitude of Graph Types

You can also use MATLAB to produce other types of common graph, such as graphs in polar coordinates, bar graphs, stair plots, histograms, stem plots, graphs with error bars, plots of complex numbers, rose plots. In this section we give examples of each, we have used the examples similar to those given in the help summary for each of these commands. We suggest that you read the appropriate help summary for full details of all the options available for these plots.

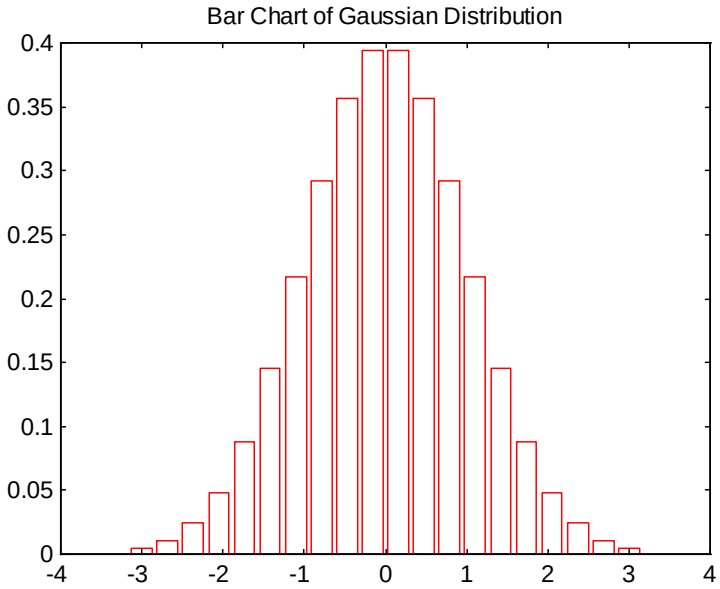
A polar plot parameterises a curve using the distance of the curve from the origin as a function of the angle measured from the `x` axis.

```
theta = 0: 0.01: 2*pi;
r = sin(8*theta) .* cos(3*theta);
polar(theta, r, 'r');
title('Polar plot of sin(8*theta)cos(3*theta)'); axis('equal')
```



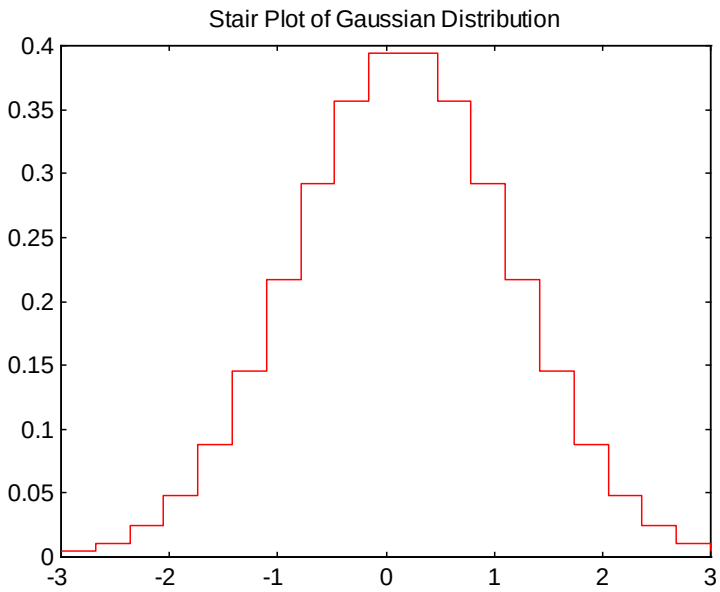
This is a bar chart of the Gaussian (or normal) distribution with mean 0 and standard deviation 1:

```
x = linspace(-3,3,20);
y = exp(-x.*x / 2 ) / sqrt(2*pi);
bar(x, y);
title('Bar Chart of Gaussian Distribution')
```



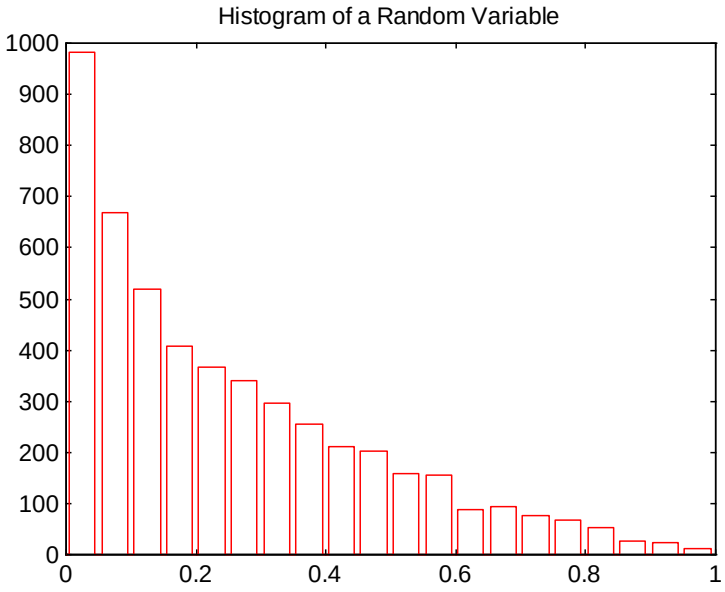
A stair plot traces the outside of a bar chart.

```
stairs(x, y);
title('Stair Plot of Gaussian Distribution')
```



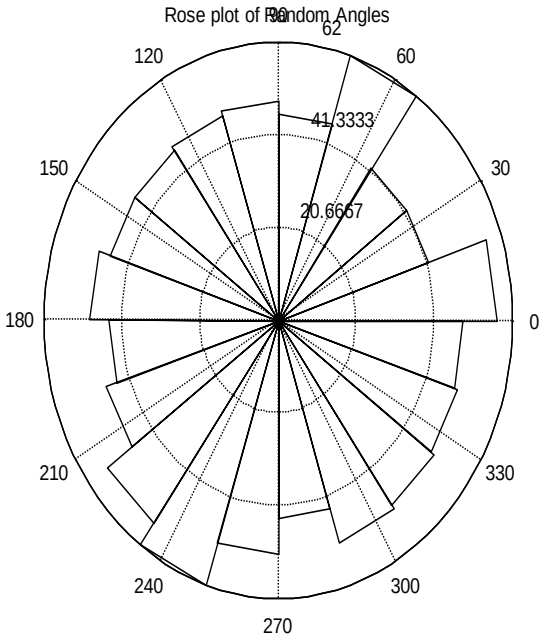
If you have a set of data from an experiment, you may want to produce a histogram of the results. The simplest way to do this is `hist(y)`, which produces a histogram with 10 equally spaced bins along the x axis. `hist(x,y)` produces a histogram of `y` with the bins as specified in `x`, if `x` is a vector. If `x` is a scalar, then `x` equally spaced bins are used. We have generated some random data:

```
r1 = rand(5000,1) .* rand(5000,1);  % A sample from a random variable
hist(r1,20);
title('Histogram of a Random Variable')
```



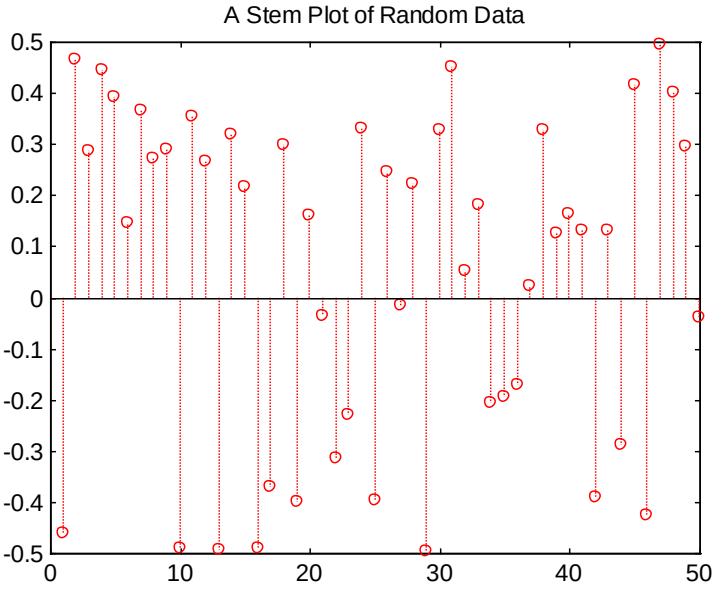
MATLAB gives a specialized histogram for viewing a set of angles, `theta`, between 0 and 2π: `rose(theta, x)` produces a histogram of the angles displaying their distribution between 0 and 2π, with the bins as specified in `x`, if `x` is a vector. If `x` is a scalar, then `x` equally spaced bins are used. If `x` is omitted, then 10 bins are used.

```
r1 = rand(1000,1)*2*pi;
rose(r1,20);
title('Rose plot of Random Angles');
```



A stem plot connects each data value in a vector with the horizontal axis using a line (which can be specified using the same formats as the `plot` function).

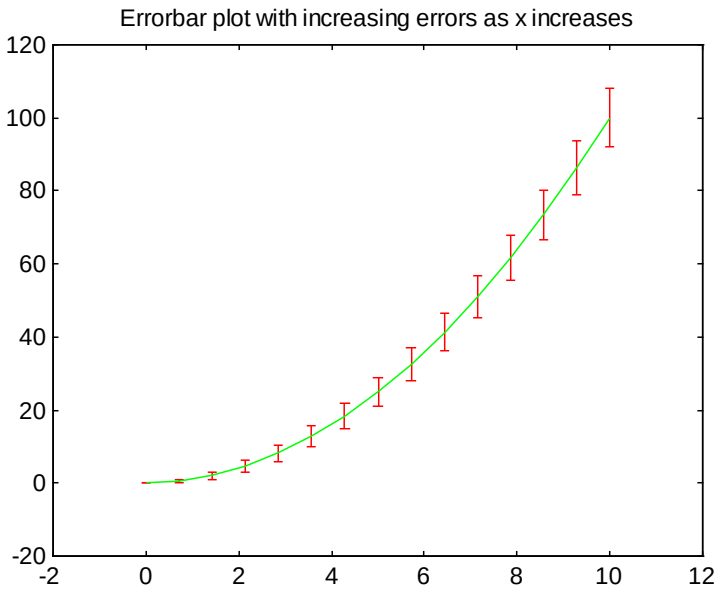
```
r2 = rand(50, 1) - 0.5;
stem(r2, ':');
title('A Stem Plot of Random Data');
```



In an experiment, there are uncertainties in the measured variable. To plot a function with errorbars, MATLAB provides the `errorbar(x, y, e)` function, which plots `(x(i), y(i))`, and draws an errorbar of length `e(i)` above and below the line at each point. Each errorbar is `2*e(i)` long. In this example, the errorbars are proportional to the `x` measurement. These sorts of errors occur in electronic systems, due to thermal (Johnson) noise, where the errors are proportional to the temperature of the system.

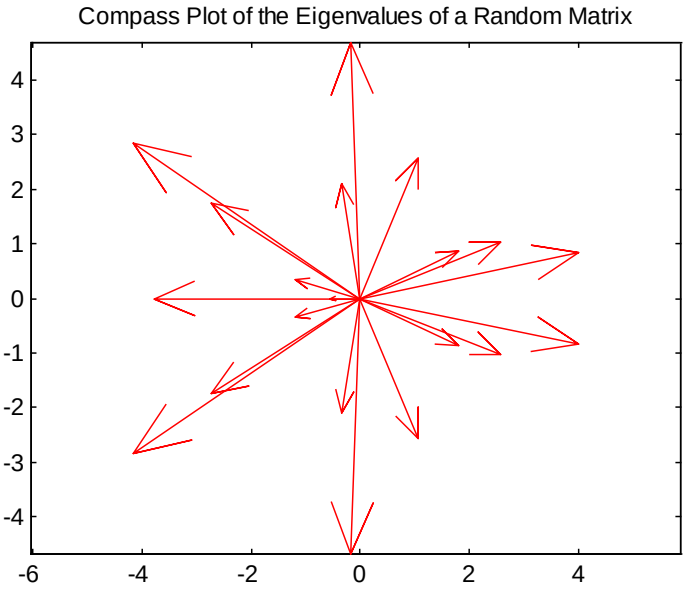
```
x = linspace(0, 10, 15); y = x.^2; err = 0.8*x;
errorbar(x, y, err);
```

```
title('Errorbar plot with increasing errors as x increases')
```



MATLAB gives two ways to visualize complex numbers. They can be useful for examining the distribution of complex eigenvalues. A compass plot displays the magnitude and direction (i.e. the length and polar angle) for a vector of complex numbers as lines which start at the origin.

```
z = eig(randn(20,20));
compass(z);
title('Compass Plot of the Eigenvalues of a Random Matrix')
```



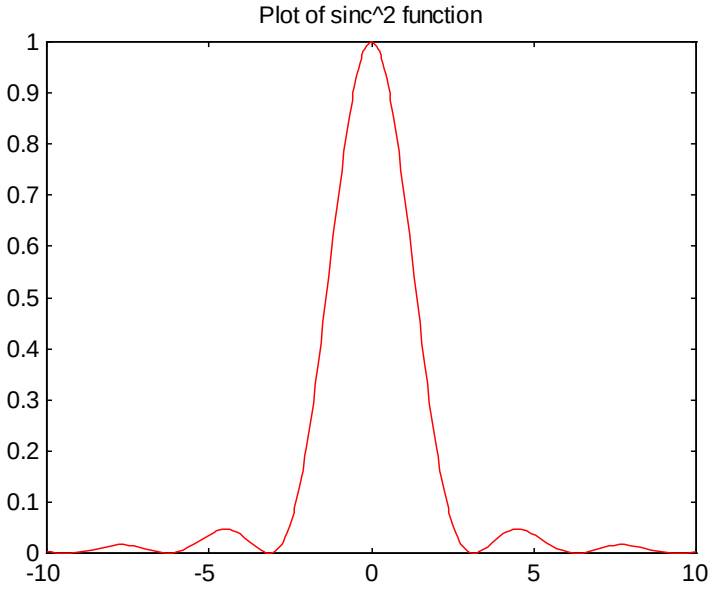
A feather plot displays the same, but spaces the complex numbers along a horizontal line.

```
feather(z);
title('Feather Plot of the Eigenvalues of a Random Matrix')
```

Previously we have plotted functions by defining a vector of points at which to sample, and then calculating the value at each of these points. Using `fplot('function', [xmin, xmax, ymin, ymax])`, we can plot a function of one variable between `xmin` and `xmin`. The `ymin` and `ymax` are optional; if given, they determine the plot view.

The sinc^2 function arises in the brightness patterns of light diffracted through a slit

```
fplot('(sin(x) ./ x).^2 ', [-10, 10, 0, 1] );
title('Plot of sinc^2 function')
```



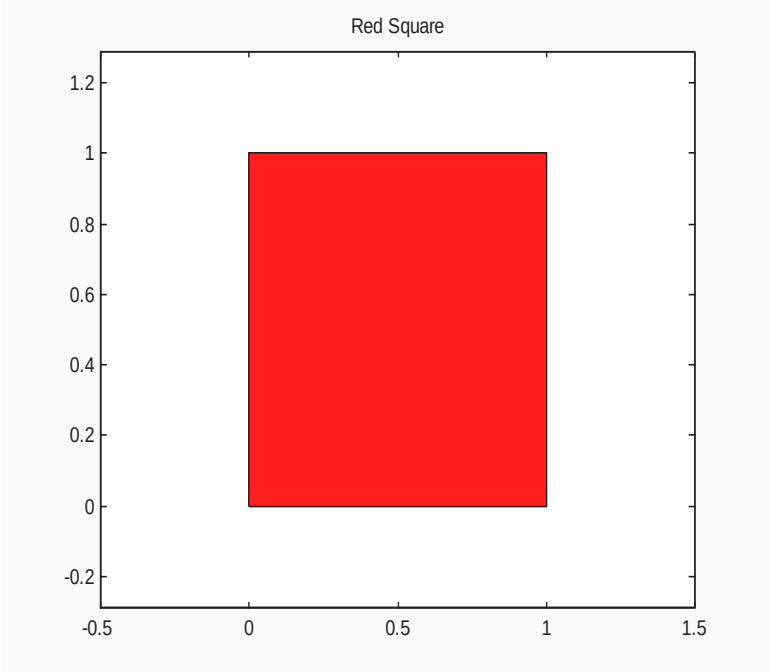
Having produced a plot, it may be necessary to select some points from the plot, for further analysis. `[x, y] = ginput(n)` takes `n` points from the current plots and returns their `x`, `y` co-ordinates in the vectors `x` and `y`. If `n` is not specified, then points are gathered until `return` is pressed. We can use this to gather the maxima of the sinc^2 function, which plotted in the last example by typing

```
[x, y] = ginput(5);
```

after the last graph, and clicking on the maxima of the function. This returns the five values at which you clicked into the vectors `x` and `y`. These maxima give the displacements about the origin at which the diffracted light will be a maximum on a screen behind the slits.

Using the `fill(x, y, c)` command, we can fill the polygon with vertices specified by the vectors `x` and `y` in the colour `c` (using the same specification for colours as the plot command- see 1.7.2). If the polygon is not closed, MATLAB will join the last point of the polygon to the first point.

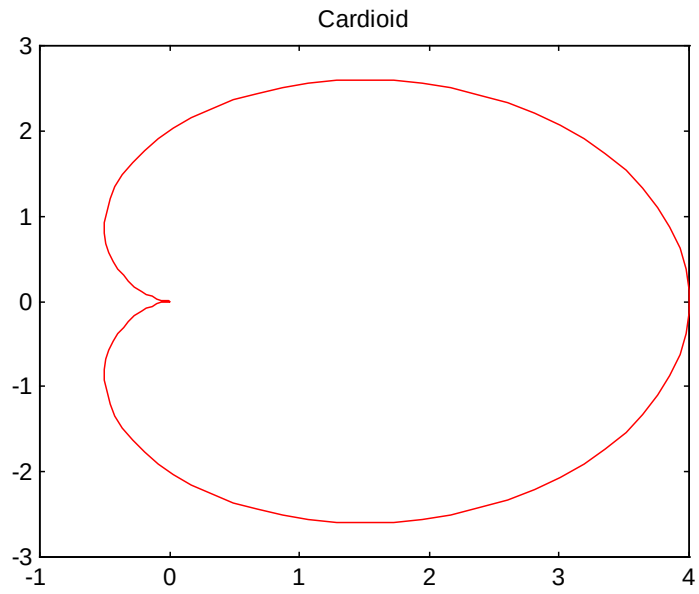
```
x = [0, 1, 1, 0]; y = [0, 0, 1, 1];
fill(x, y, 'r');
title('Red Square');
axis([-0.5 1.5 -0.5 1.5])
axis equal
```



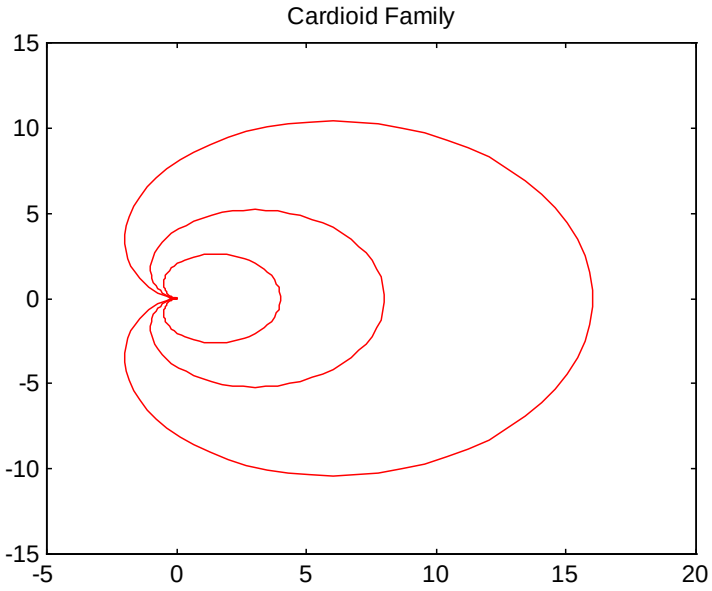
1.7.7 Handling Plots: Hold, Subplot, Figure, and Zoom.

Normally MATLAB clears the axes when a plot command is issued. However, more plots can be added to a graph using `hold on`, which freezes the current plots on the graph. If a new plot will not fit on the axes, they are re-scaled. `hold off` turns this feature off, and MATLAB will clear the axes before producing the next plot. This is particularly useful if the plot commands are complex.

```
theta = linspace(-pi, pi, 100);
plot(2*cos(theta).*(1 + cos(theta)), 2*sin(theta).*(1+cos(theta)))
title('Cardioid');
```



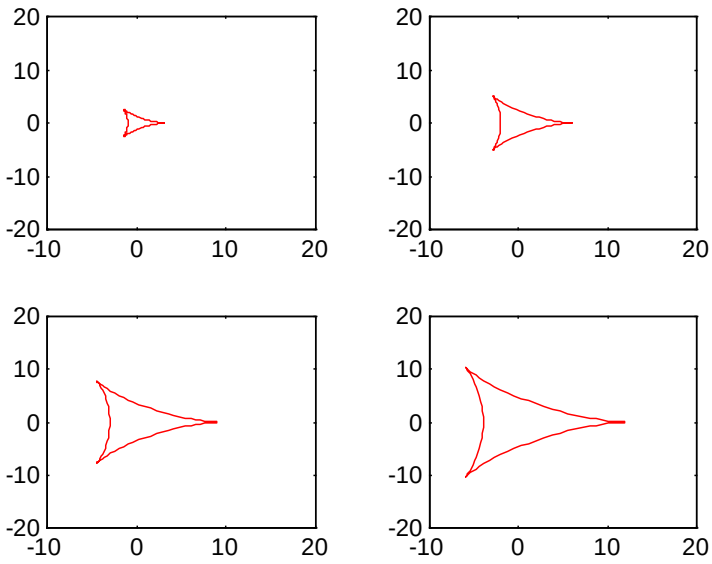

```
hold on
plot(2*2*cos(theta).*(1 + cos(theta)), 2*2*sin(theta).*(1+cos(theta)))
plot(2*4*cos(theta).*(1+ cos(theta)), 2*4*sin(theta).*(1+cos(theta)))
title('Cardioid Family');
hold off
```



To plot several plots in the same figure window, we use `subplot(m, n, p)`, which separates the current plot area into and `m` by `n` region, and sets the `p`'th sub plot to be active. The plots are numbered consecutively along the rows left to right, and then down the columns. In the next example, we display members of the "deltoid" family

```
theta = linspace(-pi, pi, 100);
x = 2*1*cos(theta).*(1+cos(theta)) - 1;
y = 2*1*sin(theta).*(1-cos(theta));
subplot(2, 2, 1);                    % Upper left corner
plot(x, y); axis([-10 20 -20 20]);
subplot(2, 2, 2);                    % Upper right corner
```

```
plot(2*x, 2*y); axis([-10 20 -20 20]);
subplot(2, 2, 3);                    % Lower left corner
plot(3*x, 3*y); axis([-10 20 -20 20]);
subplot(2, 2, 4);                    % Lower right corner
plot(4*x, 4*y); axis([-10 20 -20 20]);
```

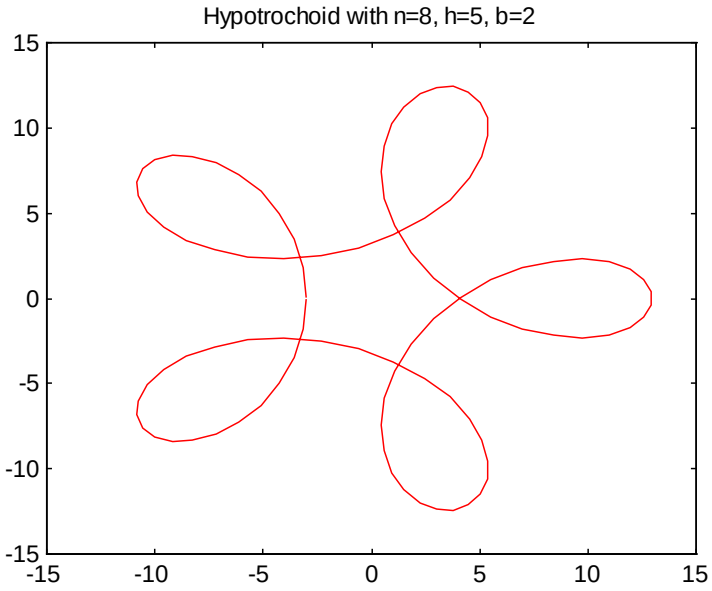


To return to a single plot in the figure window, use `subplot(1, 1, 1)`.

Sometimes it is necessary to produce several full-sized plots in separate figure windows. Using the `figure` command opens a new figure window for the next plot. The figure windows are numbered consecutively from 1, and `figure(n)` sets figure number `n` to be the next active window for the next plot. If there are several windows on the screen, the active one will be brought into the foreground. When using operations such as `hold` and `axis`, they all refer to the active plot. Under Windows, it is possible to select New Figure from the File Menu.

MATLAB allows you to zoom in on a plot. When you type `zoom`, clicking in the current plot window with the left mouse button zooms in by a factor of two centred on the position of the mouse click; clicking with the right mouse button zooms out by a factor of two. By clicking and dragging a bounding box, MATLAB zooms in on the area surrounded. To return the plot to its original state, use `zoom out`. `zoom off` turns off zoom mode. On its own, `zoom` toggles the zoom state of the current figure window. Try zooming in on the following hypotrochoid.

```
subplot(1,1,1);
theta = linspace(-pi, pi, 100);
x = 8*cos(theta) + 5*cos(8*theta/2);
y = 8*sin(theta) - 5*sin(8*theta/2);
plot(x,y);
title('Hypotrochoid with n=8, h=5, b=2')
```



(By replacing the values of 2, 5, and 8, you can generate many of the figures that old spirographs produced !)

1.7.8 Producing Hardcopy: Printing to Paper or File.

After your graph is formatted labelled with all the grids and axes required, it is time to produce hardcopy.

On its own, `print` is used to send a copy of the plot in the active figure window to the default printer. This can also be accessed under Windows from the Print option on the File Menu.

`print` can also write output to files in various formats, using `print -ddevice 'filename'`, which produces a file in the current directory that contains a picture of the plot in the current active figure window as follows:

- `-dps` PostScript for black and white printers.
- `-dpsc` PostScript for color printers.
- `-dps2` Level 2 PostScript for black and white printers.
- `-dpsc2` Level 2 PostScript for color printers.
- `-deps` Encapsulated PostScript (EPSF).
- `-depsc` Encapsulated Color PostScript (EPSF).
- `-deps2` Encapsulated Level 2 PostScript (EPSF).
- `-depsc2` Encapsulated Level 2 Color PostScript (EPSF).
- `-dwin` Send figure to currently installed printer in monochrome (Windows).
- `-dwinc` Send figure to currently installed printer in color (Windows)
- `-dmeta` Send figure to clipboard in Metafile format (Windows)
- `-dbitmap` Send figure to clipboard in bitmap format (Windows)
- `-dsetup` Bring up Print Setup dialog box, but do not print (Windows)

MATLAB will append a suitable extension, such as `.ps` or `.eps`, if the filename does not include one. Since plot outputs the whole of the current figure window, subplots are maintained as they appear on the screen. See 1.12.1 for details about changing directories in MATLAB.

You can also change the orientation of the current plot. The default is `portrait`, and prints the figure in the centre of the page vertically. To find out the current setting use

```
orient
ans =
portrait
```

You can change the current orientation to

- `landscape`, which prints the figure horizontally and fills the page using: `orient landscape`.
- `tall`, which prints the figure vertically, but makes it fill the whole page using: `orient tall`.

Once a figure is in PostScript, or bitmap form, it is easy to include it in a document such as a report, paper or thesis.

1.8 Relational and Logical Operations

Relational and logical operations return true or false values, numerically represented as 1 and 0 respectively. MATLAB uses these operators to perform element-wise operations between two matrices, returning 1 where the condition or operation evaluates to be true, and 0 elsewhere. They can also be used to extract elements (or their indices) from a matrix which obey certain conditions. In a later section, we will see that they can also be used for flow control in pieces of MATLAB code.

On input to a relational or logical expression, MATLAB considers the number 0 to mean false; any other number is true. The output is always 0 or 1.

1.8.1 Relational Operators

Using these we can make comparisons between matrices. As in the case of element-wise arithmetic, If two matrices are to be compared, they must have the same dimensions:

```
a = [1, 2, 3, 4; 5, 6, 7, 8], b = [8, 7, 6, 5; 4, 3, 2, 1]
a =
     1     2     3     4
     5     6     7     8
b =
     8     7     6     5
     4     3     2     1
c = b > a
c =
     1     1     1     1
     0     0     0     0
```

The first four elements of `b` are greater than those in `a`, so `c(1)`, `c(2)`, `c(3)`, `c(4)` are set to 1 (true), the rest evaluate to 0. The other relational operators are

- `<` Less than.
- `<=` Less than or equal to.
- `>` Greater than.
- `>=` Greater than or equal to
- `==` Equal to.
- `~=` Not equal to.

Note that `=` means an assignment, whereas `==` means a test.

```
a = (1:1:5), b = (5:-1:1)
c = a==b
a =
     1     2     3     4     5
b =
     5     4     3     2     1
c =
     0     0     1     0     0
```

`a` and `b` are only equal for the centre number in the list.

We can also compare a matrix with a scalar. The size of the matrix returned is the size of the matrix, and the elements are formed by comparing the scalar with each element of the matrix in turn:

```
 a = (1:1:6), c =  a >= 3
a =
     1     2     3     4     5     6
c =
     0     0     1     1     1     1
```

Since the values returned are 0 and 1, it is possible to perform arithmetic with the values returned. We could, for example, mask out those values of an array that do not fulfil a condition.

```
a = (1:1:6), c =  (a > 3).* a
a =
     1     2     3     4     5     6
c =
     0     0     0     4     5     6
```

This sets the entries in `c` to be the same as that in `a`, or 0 otherwise, depending on whether the entry in `a` was greater than 3 or not, respectively.

1.8.2 Logical Operators

Using logical operators, it is possible to build up more complicated tests. MATLAB provides the following Boolean operators

- `&` And

- `|` Or
- `~` Not

```
a = [2, 3, -10, 2, 3, 6; 3, 6, 4, -3, -4, 8]
a =
     2     3   -10     2     3     6
     3     6     4    -3    -4     8


b = (a > 5) | (a < -6) | (a ==3)
b =
     0     1     1     0     1     1
     1     1     0     0     0     1
```

To return the list of values meeting this set of conditions, we use

```
b = a( (a > 5) | (a < -6) | (a ==3))
b =
     3
     3
     6
   -10
     3
     6
     8
```

1.8.3 Using Relational Operators to Address Arrays

In 1.8.1, we used arithmetic to mask out those values in an array that did not satisfy a condition. However, it is more often the case that we only require those values which meet the criteria, or their indices. MATLAB provides a way to do both of these.

Consider an array with both negative and positive values. If we are measuring some physical quantity, there may be a restriction that the only feasible values are positive. To return only the positive values we use

```
a = [-1, -2, 2, 4, 5, -2, 10, -15]
a =
    -1    -2     2     4     5    -2    10   -15
positive = a( a > 0)
positive =
     2     4     5    10
```

In MATLAB 5, we must indicate explicitly using the function `logical` that we are indexing an array with an array of 0s and 1s and that MATLAB should return only those values which are true:

```
a = [1, 2, 3, 4, 5], a(logical([0, 0, 1, 1, 1]) )
a =
```

```
     1     2     3     4     5


ans =

     3     4     5
```

So our previous example is equivalent to

```
a = [-1, -2, 2, 4, 5, -2, 10, -15]; a(logical([0, 0, 1, 1, 1, 0, 1, 0]))
ans =

     2     4     5    10
```

When we extract a list of values from a matrix, they are returned as a column vector:

```
a = [-1, 2, -3; 4, -5, 6], positive = a( a>0)
a =
    -1     2    -3
     4    -5     6
positive =
     4
     2
     6
```

Sometimes it is more helpful to know the indices of the values matching a given condition. We can use `find` to achieve this.

```
a = [2, 1, 2, 2; 1, 2, 5, 3]
a =
     2     1     2     2
     1     2     5     3


[i, j] = find(a == 2)
i =
     1
     2
     1
     1
j =
     1
     2
     3
     4
```

The matrix `a` has the value 2 at (1,1), (2,2), (1,3), and (1, 4). For a row or column vector, find returns the subscripts of the elements meeting the condition:

```
a = [3; 4; 3; 2;], find(a ~=3)
a =
```

```
     3
     4
     3
     2
ans =
     2
     4
```

Elements 2 and 4 are not equal to three, and so these indices are returned.

If `find` is used without any condition, MATLAB returns those elements that are non-zero. This is often used in sparse matrix calculations.

```
a = [0, 0, 3, 0, 0, 6], find(a)
a =
     0     0     3     0     0     6
ans =
     3     6
```

1.8.4 Other Matrix Operators returning True or False

MATLAB provides a number of useful test operations

The exclusive or operation is used to determine whether one or the other (but not both) elements of two matrices evaluate to true or false.

```
a = [1, 0, 0, 2, 4, -4], b = [1, -1, 0, 0, 2, 2]
a =
     1     0     0     2     4    -4
b =
     1    -1     0     0     2     2
xor(a, b)
ans =
     0     1     0     1     0     0
```

Here `a(2) = 0` (false) and `b(2) = -1` (true), so `xor(0, -1) = 1` (true).

Other operations include:

- `any(a)` Return one if any element in a vector `a` is nonzero. If `a` is a matrix, then one is returned for each column that has a nonzero element.
- `all(a)` Return one if all elements in a vector `a` are nonzero. If `a` is a matrix, then one is returned for each column that has all nonzero elements.
- `isnan(a)` Return ones at NaNs (Not a Number) in `a`.
- `isinf(a)` Return ones at Infs (Infinities) in `a`.
- `finite(a)` Return ones at finite values in `a`.

1.9 Linear Algebra

MATLAB was first written to handle matrices, and simplify linear algebra computations. In this section we assume some knowledge of linear algebra and numerical analysis and examine how to use MATLAB to solve simultaneous equations and find eigenvalues.

1.9.1 Simultaneous Equations

We can write a set of simultaneous equations in the form `Ax = b`. `A` is a matrix of coefficients, `b` is a vector of numbers and `x` is a vector of unknowns, which we wish to find. Consider the following definition for `A`

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

The set of equations `Ax = b` will have a unique solution if the determinant of `A` is non-zero. The determinant is found using the `det` function. In this case we have

```
det(A)
ans =
     0
```

So this set of linear equations has no unique solution. (I think it is a cruel twist of faith that one of the most obvious matrices to write down as a test example has a zero determinant, and cannot be used to demonstrate most linear algebra functions!) If we consider the following matrix, the determinant is non-zero:

```
A = [1, 2, 3; 4, 5, 6; 7, 8, 1], b = [14; 32; 26]
A =
     1     2     3
     4     5     6
     7     8     1
b =
    14
    32
    26
det(A)
ans =
    24
```

We could calculate the solution to `Ax = b` using the inverse of `A`:

```
x = inv(A)*b
x =
    1.0000
    2.0000
    3.0000
```

However, finding the inverse of a matrix is often a numerically unstable problem and there are faster and more stable routines. To solve the set of linear equations, it is much better to use the left division operator (see 1.5.6):

```
x = A \ b
x =
    1.0000
    2.0000
    3.0000
```

which finds the solution to the linear equations using the LU method.

1.9.2 Badly-Conditioned Problems

Unfortunately, at the heart of most interesting problems in engineering and science are 'badly-conditioned' matrices. If we attempt to solve the set of linear equations `Ax = b` when the matrix `A` is badly conditioned, a small change in `b` will result in a large change in the solution `x`. Remember, `b` represents the data we have, `A` represents some physical knowledge about the system, and `x` represents the quantities we are trying to estimate. There can easily be small random errors in `b`. Using the following 3 by 3 matrix, we will show the dangers. This matrix is from the MATLAB gallery of matrices (see 1.10.2).

```
A = gallery(3)
A =
  -149   -50  -154
   537   180   546
   -27    -9   -25
```

Recall that the equation system will only have a solution if `det(A) ~= 0`:

```
det(A)
ans =
     6
```

So this all looks fine. Now let us find the solution for the following right hand sides, using the numerically stable LU decomposition:

```
rh1 = [-711; 2535; -120], x = A \ rh1
rh1 =
        -711
        2535
        -120
x =
    1.0000
    2.0000
    3.0000
rh2 = rh1 + [1; 1; 0.1], x = A \ rh2
rh2 =
  1.0e+003 *
   -0.7100
    2.5360
   -0.1199
x =
   99.6667
 -312.0667
    9.5000
```

What has gone wrong? We perturbed the right hand side by 0.1%, well within the bounds of experimental error, but the solution changed by two orders of magnitude in one case! The matrix is very badly conditioned. We can measure this using:

```
cond(A)
```

```
ans =
  2.7585e+005
```

This number is a measure of how badly conditioned the matrix is. The larger the number, the worse the conditioning for the matrix, which in turn means that even small errors in the data `b` will be amplified and make huge changes in our estimates of `x`. There are other ways to measure, or estimate how badly-conditioned a matrix is, see `help cond` for details. Using MATLAB, it is a simple matter to determine the condition number of a given matrix, before attempting to work with it.

1.9.3 Poor Scaling

A different problem that can arise is 'poor scaling', in which matrix elements vary over several orders of magnitude. Numerical rounding errors cause a loss of accuracy in the final solution. Fortunately MATLAB's built-in functions use techniques such as pivoting to reduce the effects of poor scaling. If you find a difference between MATLAB's results and those from a numerical FORTRAN or C routine, you should check to see whether the coefficients of the problem are poorly scaled. MATLAB will have attempted to control any loss of accuracy.

1.9.4 Finding Eigenvalues and Eigenvectors

The eigenvectors of a matrix are the non-trivial solutions to the equation `Ax = sx`, where `A` is a square matrix, `x` is an eigenvector and `s` the corresponding eigenvalue. In MATLAB we can find the eigenvalues or eigenvectors of a matrix using the `eig` function.

```
A = [1, 2, 3; 4, 5, 6; 7, 8, 1]
A =
     1     2     3
     4     5     6
     7     8     1
d = eig(A)
d =
   12.4542
   -0.3798
   -5.0744
```

In this case, the eigenvalues of A are returned in the column vector d. If we use:

```
[V, D] = eig(A)
V =
   -0.2937   -0.7397   -0.2972
   -0.6901    0.6650   -0.3987
   -0.6615   -0.1031    0.8676
D =
   12.4542         0         0
         0   -0.3798         0
         0         0   -5.0744
```

The eigenvectors are returned as the columns of `V`, and its eigenvalue is the corresponding diagonal entry of the matrix `D`.

1.9.5 Other Linear Algebra Tools

MATLAB provides many other functions for numerical linear algebra. Here are some of them:

- `A.'` is the transpose of the matrix `A`.
- `A'` is the complex conjugate transpose of `A`.
- `[L, U] = lu(A)` returns the lower and upper triangular LU factorisation of the square matrix `A` in `L` and `U` respectively, such that `LU = A`.
- `[Q, R] = qr(A)` returns an upper triangular matrix, `R`, and a unitary matrix, `Q`, such that `QR = A`, for the matrix `A`.
- `[S, V] = svd(A)` returns a diagonal matrix, `S`, of the same dimension as `A`, with non-negative diagonal elements in decreasing order, and unitary matrices `U` and `V` such that `U*S*V' = A`.
- `rank(A)` returns the rank of the matrix `A`.
- `norm(A)` computes the norm of the matrix `A`, which gives some measure of the elements of a matrix using a single number. The l-norm, 2-norm, F-norm, and ∞-norm can be found.
- `poly(A)` finds the characteristic polynomial associated with the square matrix `A`. The roots of this polynomial are the eigenvalues of `A`.

1.10 Special Matrices, and the Rogues' Gallery

MATLAB offers some special functions to enable you to create common matrices easily. It also has a small library of famous test matrices with pathological features, which are useful for testing numerical linear algebra routines.

1.10.1 Special Matrices

Here are some examples of commonly needed matrices, which can be produced using special MATLAB functions. We give examples of matrices of specific size; they all generalize to produce matrices of size `m` by `n` by passing in different parameter values. To produce a matrix filled with the same number, we have a function:

```
ones(2, 3)
ans =
     1     1     1
     1     1     1
```

This provides an easy way to fill a matrix with any number:

```
ones(2, 3)*17
ans =
    17    17    17
    17    17    17
```

If we require a matrix with all zero entries, a separate function we can also use:

```
zeros(3, 2)
ans =
     0     0
```

```
     0     0
     0     0
```

This may not, at first sight be particularly useful. However, it can be deployed to reserve storage space for a matrix of a given size, rather than repeatedly augmenting an existing matrix. This results in a faster execution of the MATLAB code. Consider

```
for i = 1:1000; x(i) = i; end
```

which is identical to `x=1:1000;` (only slower, see 1.13.1 for loop details). On a SPARC-1, this takes about 1.2 seconds, if you precede the loop by

```
x = zeros(1, 1000);
```

the loop takes only 0.2 seconds! In fact, as we shall see later, such a loop should never be used in MATLAB, since there is a more easy (and efficient) vector form for the assignment (see 1.17).

To generate matrices with uniform random entries between 0 and 1 we can use

```
rand(4, 1)
ans =
    0.0435
    0.3317
    0.2292
    0.7966
```

MATLAB also provides a function to generate random numbers with mean 0 and variance 1.

```
randn(3, 4)
ans =
   -0.0715    0.1798    0.8252   -0.5081
    0.2792   -0.5420    0.2308    0.8564
    1.3733    1.6342    0.6716    0.2685
```

You can scale the random numbers from both routines to produce random variables with other characteristics:

```
2*rand(3, 2) - 1
ans =
   -0.0640    0.1643
    0.6692   -0.1343
   -0.0628   -0.9695
```

is a matrix with entries uniformly distributed between -1 and +1.

The identity matrix is accessed using

```
eye(3)
ans =
     1     0     0
     0     1     0
     0     0     1
```

This generalizes to a matrix with ones on the diagonal and zeros elsewhere, if you do not specify a square matrix:

```
eye(3, 4)
ans =
```

```
     1     0     0     0
     0     1     0     0
     0     0     1     0
```

(The purpose in calling this matrix `eye`, seems to be to admit a pun on the word *eye*dentity in the User Guide.)

Using `A = diag(v, k)` MATLAB allows a user to create a square matrix, `A`, or order `n+abs(k)` with the vector `v` on the `k`-th diagonal. If `k = 0` (or is omitted), this is the leading diagonal. If `k > 0`, it is above the main diagonal, and if `k < 0` it is below the main diagonal.

```
v = [1, 2, 3];
A = diag(v)
A =
     1     0     0
     0     2     0
     0     0     3
A = diag(v, -1)
A =
     0     0     0     0
     1     0     0     0
     0     2     0     0
     0     0     3     0
```

Note that if we use `v = diag(A, k)`, where `A` is a matrix, this returns the `k`-th diagonal of `A` in the column vector `v`:

```
A = rand(4)
A =
    0.4875    0.3936    0.8733    0.2025
    0.2652    0.4567    0.1278    0.4905
    0.6827    0.3519    0.8477    0.7910
    0.8088    0.2888    0.3490    0.6320
v = diag(A, 2)
v =
    0.8733
    0.4905
```

You can use the function `size` to return the dimensions of another array (see 1.6.4). We can use it as follows to enable a special matrix to be created with the same dimensions as an existing one:

```
test = [1, 2, 3; 4, 5, 6]
test =
     1     2     3
     4     5     6
same_size = rand( size(test) )
same_size =
    0.2332    0.7386    0.8133
```

```
    0.4215    0.1306    0.8325
```

1.10.2 The Rogues' Gallery

In 1.9.2 we used the first of these as an example of a badly-conditioned matrix:

```
A = gallery(3)
A =
  -149   -50  -154
   537   180   546
   -27    -9   -25
```

The following matrix is one with an interesting eigenvalue structure:

```
A = gallery(5)
A =
          -9          11         -21          63        -252
          70         -69         141        -421        1684
        -575         575       -1149        3451      -13801
        3891       -3891        7782      -23345       93365
        1024       -1024        2048       -6144       24572
```

If we evaluate the eigenvalues numerically, we obtain five distinct eigenvalues: two pairs of complex eigenvalues, and a real eigenvalue:

```
eig(A)
ans =
  -0.0328+ 0.0243i
  -0.0328- 0.0243i
   0.0130+ 0.0379i
   0.0130- 0.0379i
   0.0396
```

However analytically, there is in fact a single eigenvalue with value 0 and algebraic multiplicity of 5. The roots of the characteristic polynomial are the eigenvalues. The characteristic polynomial of the matrix `A` is found using:

```
poly(A)
ans =
    1.0000    0.0000    0.0000    0.0000    0.0000    0.0000
```

This means that the eigenvalues are the solutions of $z^5 = 0$, which are `z = 0` with an algebraic multiplicity of five. Using any number other than 3 or 5 as input to `gallery` yields an empty matrix.

A less serious matrix included in MATLAB is the magic square:

```
magic(4)
ans =
    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1
```

`magic(n)`, for `n` ≥ 3 returns the `n` by `n` matrix constructed from the integers 1 to `n^2`, with equal row and column sums. Other matrices are

- `hadamard(n)` returns the Hadamard matrix of order `n`. They are matrices whose entries are ±1, and whose columns are orthogonal.
- `hilb(n)` returns the Hilbert matrix of order `n`, which is a poorly conditioned matrix.
- `invhilb(n)` returns the inverse of the Hilbert matrix of order `n`. Comparing this `invhilb(n)` with the `inv(hilb(n))` examines the effects of numerical rounding errors in the computation
- `rosser(n)` returns an 8 by 8 matrix test matrix, whose eigenvalues are hard to find.
- `wilkinson(n)` returns the Wilkinson matrix of order `n`. They are test matrices, whose eigenvalues are hard to find.

1.11 Text

MATLAB provides many features to handle numbers. In this section we discuss how MATLAB handles text. Already we have seen, in 1.7.4, how to add text to graphs.

1.11.1 Character strings

A character string is a set of ASCII characters surrounded by single quotes. MATLAB handles character strings like a row vector:

```
old = 'Hello World'
old =
Hello World


old(1:5)
ans =
Hello
```

Since the character string is like a row vector, we can access sections of the text using colon notation. You can also define character string arrays, however, since MATLAB considers the individual strings as the rows of the array, you must add spaces to the strings to make them have the same length:

```
a_string = ['Hello  '
            'How are'
            'You    ']
a_string =
Hello
How are
You
```

1.11.2 Handling Character Strings

It is possible to perform mathematical operations on character strings, using the ASCII number (between 0 and 255) for the character. These numbers are returned as the output

```
a = 'Hello', b = 'World'
a =
Hello
b =
World


a+b
ans =
   159   212   222   216   211
```

MATLAB applies all the normal rules of matrix compatibility when performing these calculations, so in this example, the two character strings must have the same number of characters (or be padded with spaces).

To obtain the string representation of a number (or set of numbers), we use `setstr(x)`, where `x` is a vector of integers in the range `0:255`. If an element of `x` is not an integer in this range, MATLAB determines the character to be printed using `fix(rem(x, 256))`.

This can be used for a simple transposition cipher:

```
code = 'HAL'
code =
HAL
decode = code +1
decode =
    73    66    77


setstr(decode)
ans =
IBM
```

(HAL was the name of the computer in "2001, A Space Odyssey", Arthur C. Clarke got the name by this transposition of the letters of 'IBM')

1.11.3 Cell Arrays

MATLAB 5 introduces a new kind of array, which enables data types of different sizes and types to be handled simply.

```
c = {'Cell arrays ';'do not'; 'need to have the same number of elements in
each row'}
c =


    'Cell arrays '
    'do not'
    'need to have the same number of elements in each row'
```

This cell array has three rows and one column, but each element of the array consists of a character string with a different number of elements. Cell arrays may be addressed and maniupalated just like normal arrays. A cell array may be converted to a character array:

```
char_array = char(c)
char_array =

Cell arrays
do not
need to have the same number of elements in each row
```

The new character array will be padded with spaces:

```
size(char_array)
ans =

     3    52
```

1.12 MATLAB Files and Functions

Up to now, we have used MATLAB by typing in commands at the command line, and receiving the answer back immediately. However, MATLAB provides mechanisms for users to add new functions and write blocks of MATLAB code. In this section we will examine how to handle files in MATLAB, how to bundle together lists of instructions for MATLAB to process, and how to add new functions to the MATLAB language.

1.12.1 File Handling commands

MATLAB allows the user to access the Operating system commands directly using the `!` operator before specifying a command.

MATLAB uses the default extension `.m` on files, if an extension is not specified; script files and functions should be saved with this extension.

The most important MATLAB file handling command is the MATLAB search path. This originates in the operating system environment variable 'MATLABPATH', which is set in the MATLAB startup script, or in `matlabrc.m`. It may be set individually set in `startup.m` (we will discuss using `.m` files in 1.12.2) The `startup.m` file is placed in a user's MATLAB directory, and is processed if it exists.

The path determines how MATLAB determines how a command is interpreted. If you type `hello`, then MATLAB:

- looks for `hello` as a variable.
- checks to see whether `hello` is a built-in function.
- looks in the current directory for the files `hello.mex`, or `hello.m`.
- searches for `hello.mex`, or `hello.m` in the directories specified in the current `path`.

If you want to access your own functions or scripts, they must either be in the current directory, or you must add the directory in which they reside to the MATLAB `path`. You can use `path(p1,p2)` to change the path to the concatenation of `p1` and `p2`.

```
path(path,'c:\')
```

is used to add the root directory of the c drive (under Windows) to your current `path`. On its own, `path` displays the current search path. MATLAB 5 (Windows and Mac versions) includes a graphical path browser.

To display the current working directory, use

```
cd
C:\MATLAB\bin
```

`chdir` or `pwd` will do the same. If you specify a name after `cd`, (or `chdir` or `pwd`), MATLAB will change the current working directory to that directory:

```
cd c:\
```

To access other operating system commands, you can use the MATLAB functions:

- `what` lists all of the MATLAB .m files in the current working directory.
- `dir`, or `ls` lists all files in the current working directory.
- `type hello` displays the MATLAB file `hello.m` in the command window.
- `delete hello` deletes the MATLAB file `hello.m`.
- `which test` returns the directory path to `hello.m`.

1.12.2 Script Files

This is a list of MATLAB commands saved to a file, which execute when you type in the name of the file at the command prompt (they are similar to shell scripts in UNIX). As we mentioned in 1.12.1, the file must reside within MATLAB's search path. In MATLAB these files have the extension `.m`, and are called M-files.

Under Windows, select New from the File menu and select M-file. This will bring up the built in editor and debugging environment and you can type in the list of MATLAB commands. Under UNIX, you should create M-files using your favourite text editor (vi, Emacs, etc.), and save them with the `.m` extension.

If you create the file `startup.m`, in your MATLAB directory, the commands in this will be executed when you start MATLAB. This is commonly used to define the MATLAB search path for your MATLAB extensions, physical constants, and engineering conversion factors.

It is very helpful to add comments to script files. Anything on a line after a `%` sign is ignored.

```
inches = 2.54;      % Conversion factor from inches to centimetres
3*inches            % 3 inches is 3*inches centimetres
ans =
    7.6200
```

MATLAB variables defined in a script file are available to the whole workspace, and the script file has access to any variables defined in the workspace. If you type `echo on` before executing a script, all of its commands are displayed as they are processed (as well as any output). If you use `echo off`, then commands (but not any assignment outputs) are suppressed. The default is `echo off`. Note that with `echo` turned `off`, only those assignments that are not terminated with a semicolon are displayed.

Script files are useful for entering large arrays using a text editor, or setting up a list of data files to load after a simulation run. In this form, they can be saved to disk and easily edited later. Once saved to disk, they can be used repeatedly.

You can use them to list a set of load commands to retrieve important output files from a computational simulation. This saves typing in the list of files each time you want to load and analyze your results. Generally, the names of these files are known to the simulation code.

With a little ingenuity you can write a short routine in C or FORTRAN, so that the list of output files, and the load command is written to a `.m` file when the code is run. By running this M-file, all of your data is loaded into MATLAB, and you are ready to start processing your data.

Script files (and functions) are compiled by MATLAB 5 into an internal representation when they are first called and will run faster on subsequent calls.

1.12.3 Adding New Functions

Up to now, you have used functions to operate on a list of parameters and pass back the results. Although we create functions in a text editor just like script files, they operate in a different manner:

- they can only access workspace variables passed into them.
- they can only alter the workspace variables which the results are passed back in.
- intermediate variables defined in the workspace are local to the function, and do not affect the workspace.

To learn how to write your own functions, let us display the `linspace` function:

```
type linspace
function y = linspace(d1, d2, n)
%LINSPACE Linearly spaced vector.
%   LINSPACE(x1, x2) generates a row vector of 100 linearly
%   equally spaced points between x1 and x2.
%
%   LINSPACE(x1, x2, N) generates N points between x1 and x2.
%
%   See also LOGSPACE, :.

%   Copyright (c) 1984-96 by The MathWorks, Inc.
%   $Revision: 5.2 $  $Date: 1996/03/29 20:24:44 $

if nargin == 2
    n = 100;
end
y = [d1+(0:n-2)*(d2-d1)/(n-1) d2];
```

`linspace` is supplied with MATLAB, and is used to create an array of linearly spaced numbers (see 1.5.3). There are a number of features of the file to note:

- The first line marks the M-file as a function, and gives its name, and the input and output variables expected.
- The function name and file name must be identical. The `linspace` function is saved in the file `linspace.m`.
- Any comment lines (starting with a `%`) up to the first line which does not start with a `%` form the text returned when you type, in this case, `help linspace`.

- The first help line (known as the Hl line) contains a synopsis of the function, and is the line which `lookfor` uses to search for functions matching a given keyword.
- All variables are local to the function, and do not affect the MATLAB workspace. The only workspace variables that the function knows about are those passed in, in this case `(d1, d2, n)`. The only variables that are changed are those on the (left) output side: `y`. Note that these names only define the names within the function- you do not need to call the function with variables called `d1`, `d2`, `n`, or `y`.
- To specify more than one output variable, the function definition is as follows

```
function [in1, in2, in3, etc] = my_function(out1, out2, out3, etc)
```

- When a function is first executed, MATLAB compiles the code into an internal representation, and the function will execute faster on subsequent calls.
- To determine the number of input variables passed to the routine, we use `nargin`. In the case of `linspace`, the `n` is optional. If only two variables are passed in, `n` is set to 100.
- To determine the number of output variables specified for the routine, we use `nargout`. This can be used to allow several possible outputs, but only perform the computations necessary for them if they are requested.

1.12.4 Summary

Mastering the use of functions allows you to add to the functionality of MATLAB. There are many user-contributed M-Files for a wide variety of tasks on the web site, see the bibliography. In section 1.17 we make some comments about how to write efficient M-files.

1.13 MATLAB Programming Structures

In the last section, we discussed how to write MATLAB code to add functions, and gather together commands to avoid re-typing them. MATLAB provides some additional programming structures to allow loops and decision making. This section will only concern you if you are trying to write script, or function files, and we assume basic knowledge of the programming structures discussed.

1.13.1 For Loops

This feature of the MATLAB programming language should never be used, until you have tried every other way to generate the required result. It is often possible to replace loops with vector statements using the colon notation, where possible, such techniques should be employed, since they are considerably more efficient (see 1.5.3).

A `for` loop can be used to repeat a set of commands a fixed number of times:

```
for count = x
   commands
end
```

The commands are repeated once for every column in the array `x`. At each iteration, `k`, the `count` variable takes on the value of the respective column of the array: `x(:,k)`.

```
a = zeros(1,10);
for k = 1:10
   a(1,k) = k^2;
```

```
end
a
a =
     1     4     9    16    25    36    49    64    81   100
```

In this example we set up an array with the squares of the numbers 1 to 10. Note the use of `zeros` before the loop; by predefining the size of the array, the code runs faster. In this example it would have been better to use

```
a = (1:10).^2
a =
     1     4     9    16    25    36    49    64    81   100
```

which is more efficient.

Since the `count` variable takes on the value of the respective column of the array `x`, we can exploit this:

```
a  = [1, 2, 3; -10, -9, -8]
a =
     1     2     3
   -10    -9    -8
for count  = a
    count(1) + count(2)
end
ans =
    -9
ans =
    -7
ans =
    -5
```

MATLAB supports nesting of for loops:

```
a = zeros(3);
for k = 1:3
   for l = 1:3
       a(l,k)=1 / (k+l-1);
    end
end
a
a =
    1.0000    0.5000    0.3333
    0.5000    0.3333    0.2500
    0.3333    0.2500    0.2000
```

This defines the square Hilbert matrix of order 3.

```
hilb(3)
ans =
    1.0000    0.5000    0.3333
```

```
    0.5000    0.3333    0.2500
    0.3333    0.2500    0.2000
```

1.13.2 While Loops

A `while` loop is used to repeat a set of commands while all the elements in an array evaluate to true. This algorithm returns the power of two closest to, but not greater than `n`.

```
n = 12345;
x = 1;
while x<=n
   x = x*2;
end
x/2
ans =
        8192
```

To perform a fast Fourier transform, on a set of data, we need the number of values to be a power of two. If we are to truncate the data, we need to determine the closest power of two, which does not exceed the number of data we have. There are many other ways to implement this, we have used the above method as an example of a `while` loop.

1.13.3 If .. Else Decisions

We have seen an example of making a decision in a function file. In 1.12.3, the function `linspace` tests to determine how many variables have been passed to it:

```
type linspace
function y = linspace(d1, d2, n)
%LINSPACE Linearly spaced vector.
%   LINSPACE(x1, x2) generates a row vector of 100 linearly
%   equally spaced points between x1 and x2.
%
%   LINSPACE(x1, x2, N) generates N points between x1 and x2.
%
%   See also LOGSPACE, :.

%   Copyright (c) 1984-96 by The MathWorks, Inc.
%   $Revision: 5.2 $  $Date: 1996/03/29 20:24:44 $

if nargin == 2
    n = 100;
end
y = [d1+(0:n-2)*(d2-d1)/(n-1) d2];
```

If it has only received 2, then the function returns 100 equally spaced values across the range specified. There are three forms of `if` statement:

1) In the simplest form, a set of statements are executed if an expression evaluates to true:

```
 if x
```

```
   commands
end
```

2) Here there are two sets of commands, one set to execute if `x` is true. The other set is executed if `x` is false.

```
if x
   commands to execute if x is true
else
   commands to execute if x is false
end
```

3) This gives the generalization in which several expressions are tested. The set of commands following the first expression to evaluate to true is executed, and then the rest of the tests and commands are skipped (no more tests are performed). The final `else` is optional.

```
if x1
      commands to execute if x1 is true
   elseif x2
      commands to execute if x2 is true
   elseif x3
      commands to execute if x3 is true
   (etc)
   else
      commands to execute if no other commands are true.
end
```

This example shows the use of the third form of `if`.

```
x1 = [1 0]; x2 = [1, 1]; x3 = [2, 2]; x4=[666];
if x1
    x1
elseif x2
    x2
elseif x3
   x3
else
    x4
end
x2 =
     1     1
```

Note that the conditions can be arrays, which evaluate to true only if all of their components are true. In this case `x2` was the first matrix tested which evaluated to true, and so it was returned.

MATLAB 5 provides `switch .. case` structure, which is more convenient if you wish to test a single argument for equality with one of a number of options, see `help switch` for more details.

1.14 Polynomials

1.14.1 Polynomial Storage

MATLAB handles evaluation of simple polynomials by storing the coefficients of the polynomial in a row vector in descending order of power. MATLAB represents the polynomial $z^5 - 2z^4 + 3z^3 + 4z - 12$ using:

```
z = [1, -2, 3, 0, 4, -12]
z =
     1    -2     3     0     4   -12
```

Note the zero entry for the coefficient of z^2 must be included.

1.14.2 Roots of a Polynomial

To find the roots of a polynomial, we find the values of x that satisfy $z^5 - 2z^4 + 3z^3 + 4z - 12 = 0$.

```
soln = roots(z)
soln =
   1.1667+ 1.7265i
   1.1667- 1.7265i
  -0.8790+ 1.0805i
  -0.8790- 1.0805i
   1.4245
```

In general, some of the roots of a polynomial will be complex, and MATLAB returns a complex column vector with the solutions in. It is important to appreciate that finding the roots of a polynomial can be a badly conditioned problem (see 1.9.2 for an example of a badly conditioned problem). Small changes in the coefficients can result in large changes in the polynomial roots.

MATLAB can also construct a polynomial, given its roots:

```
cubic_roots = [12, 1 + i, 1 - i]
cubic_roots =
          12        1001        -999
cubic_poly = poly(cubic_roots)
cubic_poly =
           1         -14     -999975    11999988
```

So the polynomial is $z^3 - 14z^2 + 26z - 24$. The polynomial coefficients may be complex.

1.14.3 Adding Polynomials

If two polynomials are of the same order, then standard array addition can be used. If not, you must add zero coefficients for the higher powers in the lower order polynomial.

To add $z^3 - 14z^2 + 26z - 24$ to $10z^2 + 3z - 2$, we define

```
z1 = [1, -14, 26, -24],  z2 = [0, 10, 3, -2]
z1 =
     1   -14    26   -24
z2 =
     0    10     3    -2
z1+z2
```

```
ans =
     1    -4    29   -26
```

1.14.4 Multiplying Polynomials

Multiplying two polynomials is the same as taking the Fourier transform of the coefficients, multiplying and then taking the inverse Fourier transform (this comes from the 'convolution theorem'). MATLAB provides a single function, `conv`, to do this. Consider multiplying z^3-$14z^2 + 26z - 24$ and $10z^2 + 3z - 2$:

```
z1 = [1, -14, 26, -24],  z2 = [10, 3, -2]
z1 =
     1   -14    26   -24
z2 =
    10     3    -2
conv(z1,z2)
ans =
    10  -137   216  -134  -124    48
```

The product is $10z^5 - 137z^4 + 216z^3 - 134z^2 - 124z + 48$. Note that we do not need to zero pad the second array. We can check whether this is the same as using Fourier transforms using MATLAB's `fft` function:

```
z1 = [1, -14, 26, -24];  z2 = [10, 3, -2];
x = fft([z1 zeros(1, length(z2) - 1)]);
y = fft([z2 zeros(1, length(z1) - 1)]);
cmplx_poly = ifft( x .* y)
cmplx_poly =
  1.0e+002 *
  Columns 1 through 4
   0.1000- 0.0000i  -1.3700- 0.0000i   2.1600+ 0.0000i  -1.3400+ 0.0000i
  Columns 5 through 6
  -1.2400- 0.0000i   0.4800- 0.0000i
```

Taking the real part of the last array yields

```
real(cmplx_poly)
ans =
   10.0000 -137.0000  216.0000 -134.0000 -124.0000   48.0000
```

which is precisely the product of the polynomials, as determined previously. The actual implementation of `conv` is more subtle (see `type conv` for details).

1.14.5 Dividing Polynomials

Using the `deconv` function, it is possible to divide one polynomial by another. To determine $10z^5 - 137z^4 + 216z^3 - 134z^2 - 124z + 48$ / $10z^2 + 3z - 2$, we use

```
z1 = [10, -137, 216, -134, -124, 48];
z2 = [10, 3, -2];
[q, r] = deconv(z1,z2)
q =
    1.0000  -14.0000   26.0000  -24.0000
```

```
r =
  1.0e-013 *
         0    0.2842         0    0.2842    0.1421   -0.0711
```

This yields a polynomial part, `q`, which is $z^3 - 14z^2 + 26z - 24$, and a remainder, `r`, which is 0 (excluding numerical rounding errors).

```
z1 = [1, 2, 12, 11, 1];
z2 = [1, 0, 4];
[q, r] = deconv(z1,z2)
q =
     1     2     8
r =
     0     0     0     3   -31
```

In this case:

$$\frac{z^4 + 2z^3 + 12z^2 + 11z + 1}{z^2 + 4} = z^2 + 2z + 8 + \frac{3z - 31}{z^2 + 4} \qquad (1.14.1)$$

1.14.6 Polynomial Derivatives

MATLAB provides the `polyder` function to obtain the derivatives of a polynomial.

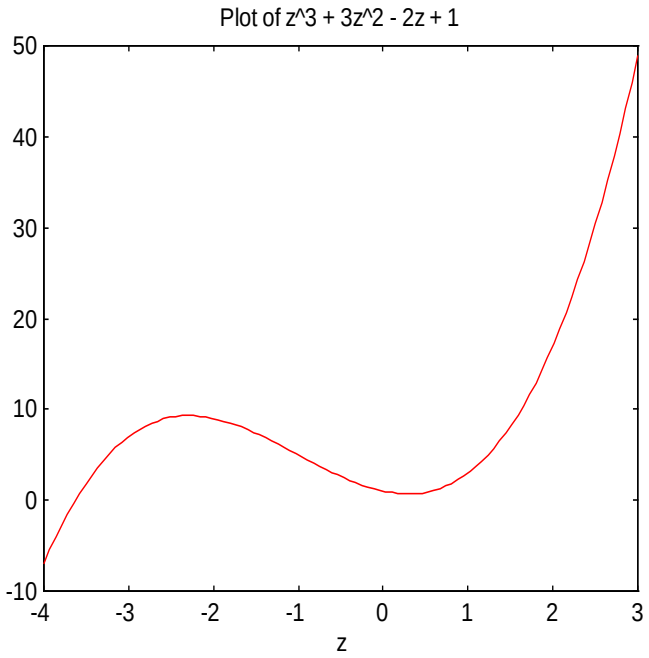
```
z1 = [1, 2, 12, 11, 1];
polyder(z1)
ans =
     4     6    24    11
```

$$\frac{d}{dz}\left(z^4 + 2z^3 + 12z^2 + 11z + 1\right) = 4z^3 + 6z^2 + 24z + 11 \qquad (1.14.2)$$

1.14.7 Polynomial Evaluation

To evaluate a polynomial at a point, you can use the MATLAB `polyval(z, z0)` function, which finds the value of the polynomial defined by the coefficients in `z` at the points in `z0`. An efficient algorithm is used, and it is always best to evaluate a polynomial at a point using this function (see `type polyval` for details). We can plot the polynomial $z^3 + 3z^2 - 2z + 1$ using

```
z = [1, 3, -2, 1];
z0 = linspace(-4,3,100);
value = polyval(z, z0);
plot(z0, value);
title('Plot of z^3 + 3z^2 - 2z + 1')
xlabel('z')
```



1.14.8 Partial Fractions

MATLAB provides a function to expand a polynomial ratio into partial fractions. `[r, p, d] = residue(f, g)` expands:

$$\frac{f}{g} = \frac{r(1)}{z - p(1)} + \frac{r(2)}{z - p(2)} + \ldots \frac{r(n)}{z - p(n)} + d \qquad (1.14.3)$$

where `f` and `g` are polynomials in *z*, the residues are the `r(i)`, the poles are `p(i)`, and the direct term is the polynomial `d`.

Consider:

```
z1 = [1, 0, 1];
z2 = [1, -1, 0];
[r, p, k] = residue(z1, z2)
r =
     2
    -1
p =
     1
     0
k =
     1
```

This is interpreted as

$$\frac{z^2 + 1}{z^2 - z} = \frac{2}{z - 1} - \frac{1}{z} + 1 \qquad (1.14.4)$$

There are other options for using `residue`, see `help residue` for details. For a general polynomial ratio, evaluating a partial fraction expansion is an ill-posed problem: small changes in the input coefficients can make large changes in the partial fraction expansion.

1.15 An Introduction to Numerical Analysis with MATLAB

In this section we introduce some MATLAB tools for performing basic numerical analysis. For more details about the issues raised in this section, see a numerical analysis textbook such as those mentioned in the bibliography.

Throughout this section we will consider MATLAB's built-in `humps` function, which has interesting properties.

1.15.1 Curve Fitting: Least Squares

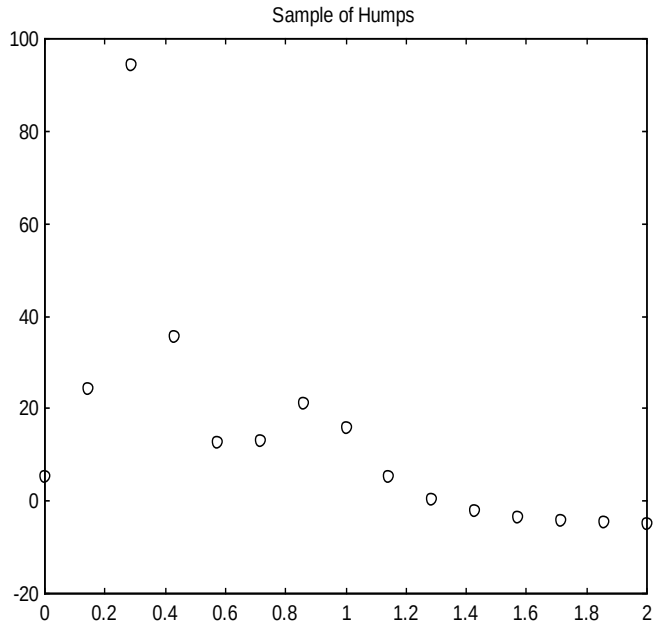
Given a discrete sample of points from some function, what kind of curve could we draw which best represents the trends in the data. Naturally there are lots of possible curves, an infinite number in fact, so which one can we choose? When we fit a curve, we do not expect that the curve will pass through each of the data points.

One way to fit the curve is to minimize the sum of the squares of the discrepancies (or "residuals") between the data and the values predicted by the hypothetical function. We adjust the parameters of the function to minimise this sum. This is known as least squares fitting of data.

MATLAB provides `polyfit(x, y, o)` to perform a least squares fit of a polynomial of order `o` to the data in `x` and `y`. The function `polyval(p, x)`, evaluates the polynomial defined by `p` at the set of points `x`.

Let us take 15 equally spaced points from the function humps, and consider qualitatively the effect of plotting polynomials of various order through these points.

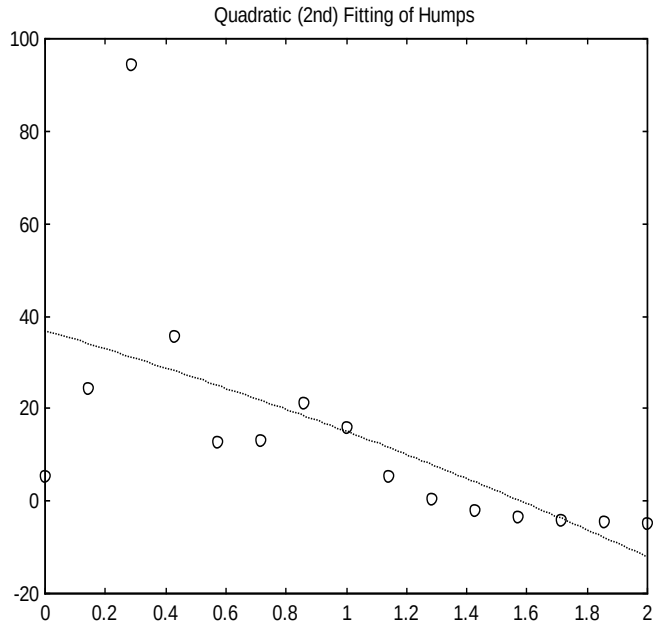
```
x = linspace(0,2,15);
y = humps(x);
plot(x,y,'ro')
title('Sample of Humps')
```



With just this sample of points, it is hard to know what sort of curve to fit. We could try a second order polynomial as follows

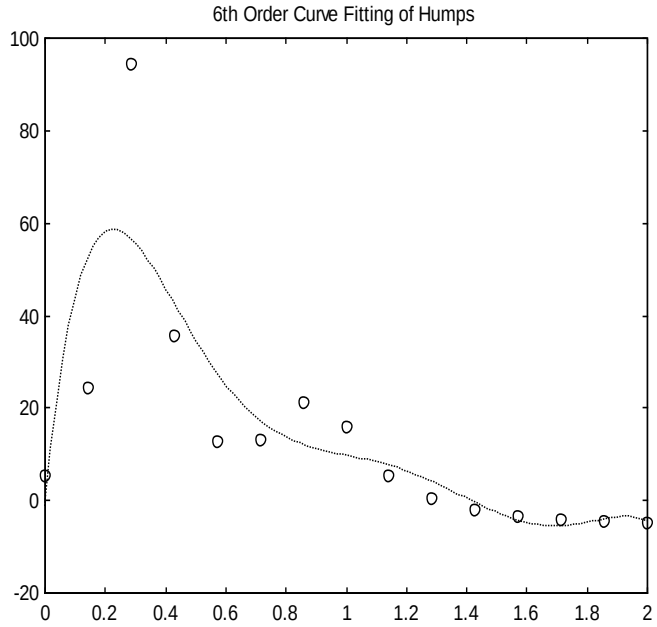
```
order_2 = polyfit(x,y,2);
x_2 = linspace(0,2,100);
y_2 = polyval(order_2, x_2);
```

```
plot(x, y, 'ro', x_2, y_2, 'b:')
title('Quadratic (2nd) Fitting of Humps')
```



The fit does not look very impressive. Perhaps a higher order polynomial will do better:

```
order_6 = polyfit(x,y,6); x_6 = linspace(0,2,100);
y_6 = polyval(order_6, x_6);
plot(x, y, 'ro', x_6, y_6, 'b:')
title('6th Order Curve Fitting of Humps')
```



This fit looks more reasonable. Using `polyval`, we can evaluate the function at each of the sample points, and determine the residual (the difference between the estimate given by our function, and the actual function value).

```
y_s = polyval(order_6, x);
```

```
residual = y_s - y;
results = [x; y; y_s; residual; residual/std(residual)]'
results =
         0    5.1765   -1.3549   -6.5314   -0.4710
    0.1429   24.4541   52.4369   27.9827    2.0177
    0.2857   94.3961   56.5751  -37.8210   -2.7271
    0.4286   35.5055   42.7865    7.2809    0.5250
    0.5714   12.7098   27.5492   14.8393    1.0700
    0.7143   12.9303   17.1772    4.2469    0.3062
    0.8571   21.0235   12.0047   -9.0187   -0.6503
    1.0000   16.0000    9.6716   -6.3284   -0.4563
    1.1429    5.4912    7.5085    2.0173    0.1455
    1.2857    0.3160    4.0234    3.7074    0.2673
    1.4286   -2.0900   -0.5119    1.5781    0.1138
    1.5714   -3.3478   -4.3748   -1.0270   -0.0741
    1.7143   -4.0802   -5.6036   -1.5235   -0.1099
    1.8571   -4.5434   -4.1103    0.4331    0.0312
    2.0000   -4.8552   -4.6910    0.1642    0.0118
```

The table above gives the `x` and `y` values, our estimate of the `y` value using a 6th order polynomial, the residual and the normalized residual. A normalized residual of more than 2 is surprising, so if the humps function were a set of experimental results, we would be suspicious of the y values of 52.4369 and 56.5751, but we would probably not reject them.

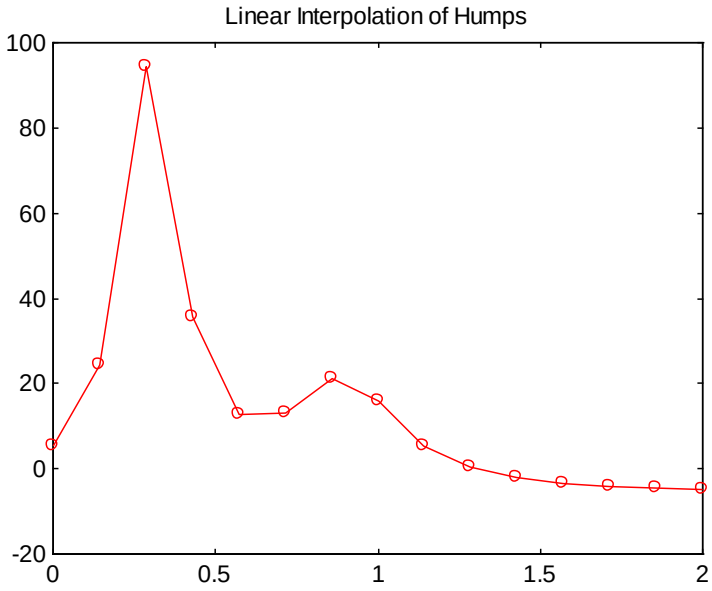
In 1.16, we give a worked example of data-fitting in which the fit is considerably better.

1.15.2 Interpolation I: Linear

Given a discrete sample of points from some function, how can we determine the function value between those points. This is like curve fitting, but the curve must pass through every data point.

The simplest type of interpolation is the linear interpolation used by MATLAB when plotting a curve; the points are joined up by straight lines.

```
x = linspace(0, 2, 15);
y = humps(x);
plot(x, y,'r-', x, y, 'ro');
title('Linear Interpolation of Humps')
```



As the number of sample points increases and the distance between them decreases, linear interpolation becomes more accurate. However, if we only have a fixed number of samples, then we want to do the best job fitting the points we have.

1.15.3 Interpolation II: Polynomials

For `n` data points, a polynomial of order `n-1` will pass through each data point.

```
order_14 = polyfit(x,y,14);
x_14 = linspace(0,2,100); y_14 = polyval(order_14, x_14);
plot(x, y, 'ro', x_14, y_14, 'b:')
title('14th Order Interpolation of Humps')
```

The curve fits through every data points, but it oscillates wildly- especially between the first pair and last pair of points. There is no evidence in the data for this oscillatory behaviour, and yet our attempt to plot a curve through all the points has produced it. If we have more data points, we require an even higher order polynomial. The oscillatory behaviour will be worse, and evaluating the polynomial may become very time-consuming.

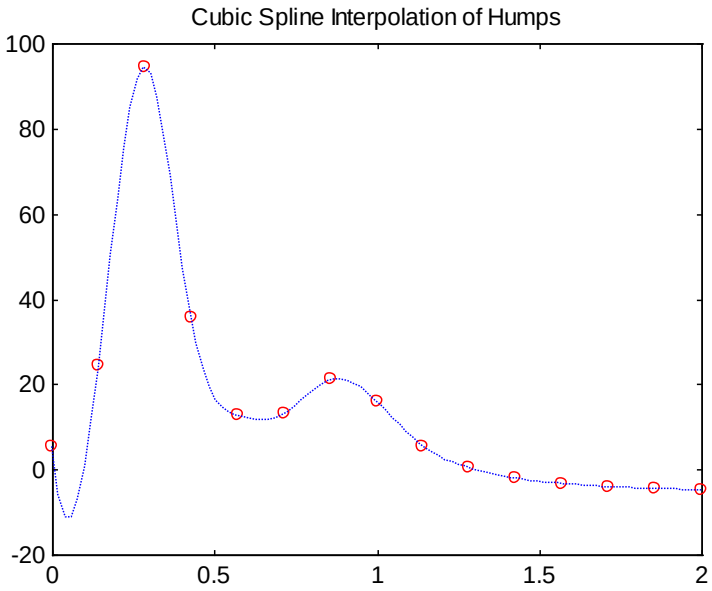
1.15.4 Interpolation III: Splines

Fortunately, there is a rigorous way to draw a smooth curve between a set of points. The basis of this method is to fit sections of curve between sets of points, using the 'spline' method. Many cars are designed using splines to ensure 'smoothness' of the chassis. MATLAB provides the function `interp1(x, y, xi, 'method')` to perform various types of interpolation between points `x` and `y` and return the interpolated value at the points in `xi`:

- `linear` fits a straight line between pairs of points. This is the default if no method is specified, and is the method used by MATLAB when joining adjacent points on a plot.
- `spline` fits cubic splines between adjacent points.
- `cubic` fits cubic polynomials between sets of 4 points. The `x` points must be uniformly spaced

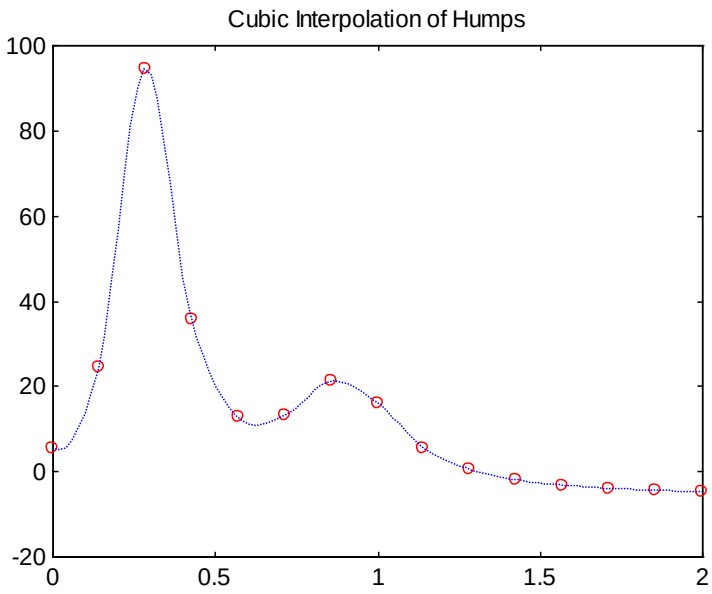
In all cases, `x` must be monotonic (it must either increase or decrease over the range). None of the values in `xi` can lie outside the range of the `x` values supplied- `interp1` will not perform extrapolation.

```
spline_x = linspace(0,2,100);
cubic_spline = interp1(x,y,spline_x,'spline');
plot(x, y, 'ro', spline_x, cubic_spline, 'b:')
title('Cubic Spline Interpolation of Humps')
```



We now have a smooth curve between the points.

```
cubic_x = linspace(0,2,100);
cubic = interp1(x,y,cubic_x,'cubic');
plot(x, y, 'ro', cubic_x, cubic, 'b:')
title('Cubic Interpolation of Humps')
```

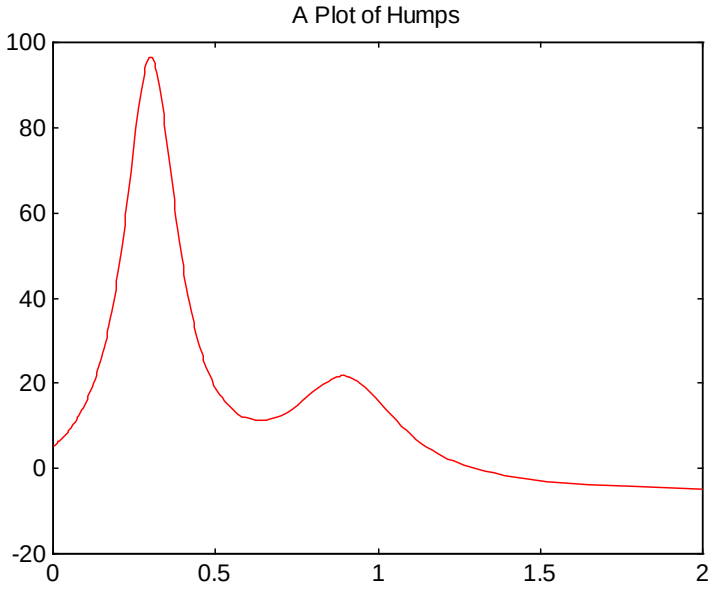


In this case, the '`cubic`' method provides the smoothest curve with the least oscillations between points. In the next section, we reveal what the function looks like.

1.15.5 The Humps function

Here is a plot of the humps function.

```
fplot('humps',[0 2]);
title('A Plot of Humps')
```



The cubic interpolation reproduced a similar plot using only a sample of ten points across the range. For more information about interpolation, see one of the numerical analysis books in the bibliography.

1.15.6 Interpolation III: Surface Splines

(This section includes some 3D plots, so you might like to look briefly at 1.18 first.)

Interpolating over a surface is an extension of interpolating between two data points. In MATLAB we can use `interp2(X, Y, Z, XI, ZI, 'method')` to interpolate at the points `(XI, YI)` the function whose value is `Z` at `X, Y`.

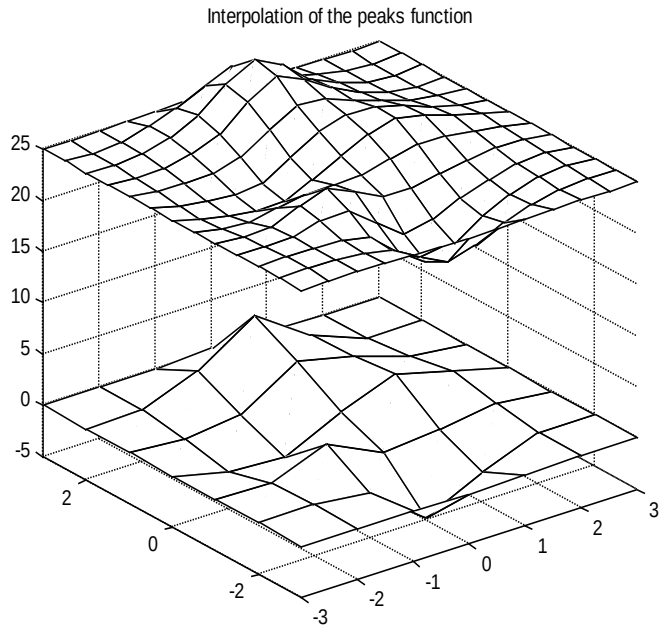
Two methods that can be used

- `linear` linear interpolation. This is the default if no method is specified.
- `cubic` cubic interpolation. The `X` and `Y` points must be uniformly spaced.

In all cases, `X` and `Y` must be monotonic. None of the values in `XI` can lie outside the range of the `X` or `Y` values supplied- `interp2` will not perform extrapolation.

In this example, we use cubic interpolation to produce a smooth mesh from samples of MATLAB's `peaks` function:

```
[X, Y] = meshgrid(-3:1:3);
Z = peaks(X, Y);
[XI, YI]= meshgrid(-3:0.5:3);
ZI_linear = interp2(X, Y, Z, XI, YI, 'linear');
ZI_cubic = interp2(X, Y, Z, XI, YI, 'cubic');
mesh(XI, YI, ZI_cubic+25);
hold on
mesh(X, Y, Z);
colormap([0,0,0]);
hold off
axis([-3, 3, -3, 3, -5, 25]);
title('Interpolation of the peaks function')
```



The bottom part of the figure shows the coarse samples, and the top shows the interpolated grid. We have displace the interpolated grid upwards.

1.15.7 Function Minimization or Maximization

After a function is plotted successfully, either by evaluating the function at many points along the region of interest, or by interpolating between a smaller number of points, we may want to determine where the function has minimum or maximum values. Since maximizing a function `f(x)` is equivalent to minimizing `-f(x)`, we only need a function for minimization.

Analytically, mimima, or maxima are points on the curve where the derivative of the function is zero. In the case of the humps function, we are not given its analytic form, so we cannot use this technique. In engineering one may not have an analytical function, or it may be too hard to find the points at which the derivative is zero. MATLAB provides `fmin('function', xmin, xmax)` to search for the minimum value of a `function` in the range `xmin` to `xmax`.

```
fmin('humps', 0.5, 0.9)
ans =
    0.6370
```

From the plot of the function in 1.15.2, there is a minimum at 0.6370 (4 dp). To find the position of the maximum value of humps, we evaluate

```
fmin('-humps(x)', 0.5, 0.9)
ans =
    0.8927
```

This is a local maximum between 0.5 and 0.9. The global maximum is at

```
fmin('-humps(x)', 0, 1)
ans =
    0.3004
```

Note that we have specified an independent variable `x`, so that MATLAB knows the variable over which to minimize. To minimize (or maximize) a complex function, you can set up an M-file that returns the function value at a given point. For this, and other uses of `fmin`, see `help fmin`.

1.15.8 Finding Zeros of a Function

In an engineering optimization problem, for example a design, you may wants to determine when a function takes on some value `c`. This may be some critical design parameter, or may reflect the cost of the product.

To find the `x` values at which a function `f(x)` takes on a certain value, `c`, we can consider when the function `f(x)-c` takes on the value 0. In MATLAB, we can use `fzero('function', xstart)` to search for a value near to `xstart`, where the function is zero.

```
fzero('humps',0)
ans =
   -0.1316
fzero('humps',2)
ans =
    1.2995
```

`humps` crosses zero at -0.1316 (4 dp) and 1.2995 (4 dp).

To find the x value for which `humps(x) = 60`, we must write a function which returns `'humps(x) -60'`, and then pass the name of this function to `fzero`. To quote the MATLAB (version 4) User's Guide: "it was never given the capability to accept a function described by a character string using x as the independent variable". To translate: this is a bug they are aware of, but have not fixed (yet)!

```
fzero = fzero('humps(x)-60',0.5)
```

```
Warning: Reference to uninitialized variable 'fzero'.
??? Matrix indices must be full double.
```

If this worked, it would return the value of the zero. You could write a function:

```
function y = humps2(x)
% Defines the function humps-60
y = humps(x)-60;
```

and save it to disk as `'humps2.m'`. Using `fzero('humps2',0.5)`, you would find that the function humps takes on the value 60 at 0.3769 (4dp), and using `fzero('humps2',0.2)`, it also takes on the value 60 at 0.2250 (4 dp).

MATLAB returns the first zero it finds near to the start value given, if there are no zeros, then MATLAB will stop and provide a diagnostic message.

1.15.9 Numerical Integration

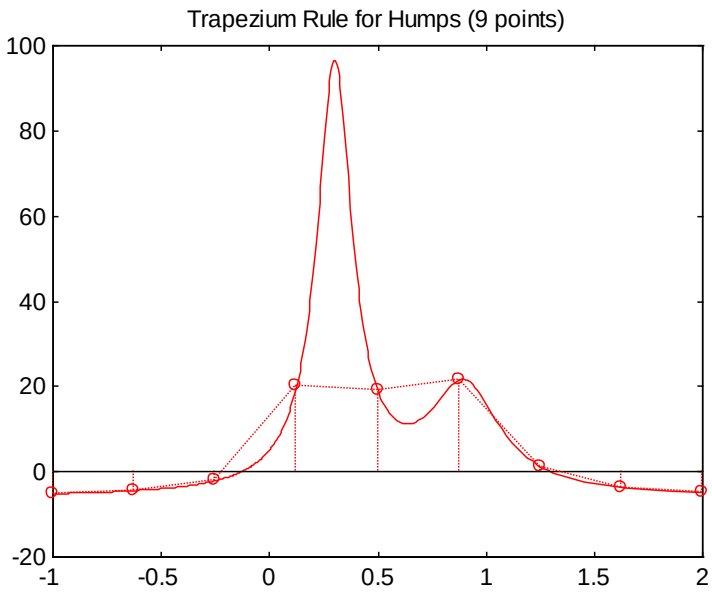
There are some functions (such as that defining the Gaussian distribution) which it is not possible to integrate. MATLAB provides three functions for evaluating a definite integral, by summing up the area under the curve:

- `trapz(x,y)` uses the Trapezium rule.
- `quad('function',a,b,tol)` uses an adaptive recursive Simpson's Rule.
- `quad8('function',a,b,tol)` uses an adaptive recursive Newton-Cotes 8 panel rule.

The Trapezium rule adds up the areas of trapezia under the curve, the value of the function, y, at the points x are passed in and MATLAB sums the area of the resulting trapezia.

Before performing any numerical integration it is imperative to plot the function first, to examine it for singularities or to determine whether the function is negative over some of the range of integration. MATLAB takes no account of whether the function is negative or positive over the range. If you want all the area to be strictly positive, you should work with the absolute value of the function using `abs`. In the next example we return to the `humps` function, and plot the trapezia with the function, and then give the answer in both cases

```
fplot('humps',[-1,2])
x = linspace(-1, 2, 9);
y = humps(x);
hold on
stem(x,y,':')
plot(x,y,':')
title('Trapezium Rule for Humps (9 points)')
hold off
```

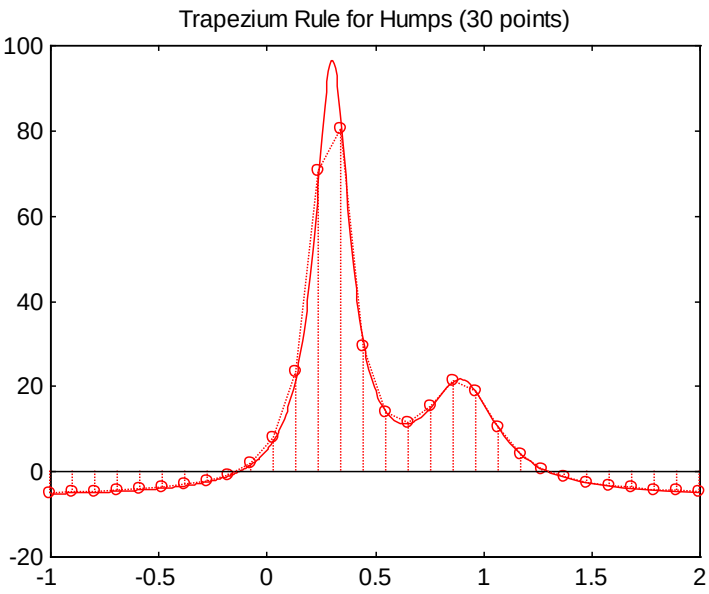


After plotting the function and the trapezium rule approximation, we know already that the answer will be an underestimate:

```
trapz(x,y)
ans =
   17.5601
```

To find a better answer, we can use more points on the curve, but first a plot, to visualize the numerical procedure:

```
fplot('humps',[-1,2])
x = linspace(-1, 2, 30);
y = humps(x);
hold on
stem(x,y,':')
plot(x,y,':')
title('Trapezium Rule for Humps (30 points)')
hold off
```

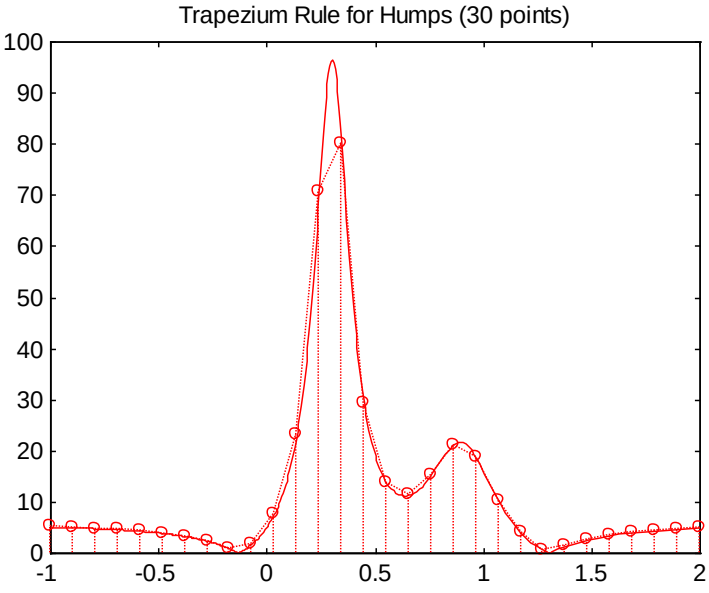


This looks better, although some of the area in the largest peak has been missed:

```
trapz(x,y)
ans =
   26.2102
```

Using MATLAB, we can find the total area between the curve and the `x` axis by working with the absolute value of the function:

```
fplot('abs(humps(x))',[-1,2])
x = linspace(-1, 2, 30);
y = abs(humps(x));
hold on
stem(x,y,':')
plot(x,y,':')
title('Trapezium Rule for Humps (30 points)')
hold off
```



The area is now given by:

```
trapz(x,y)
ans =
   37.5159
```

From the plots, we know in all of these cases that some of the area has been missed. We could add more points along the range of interest, until the approximation was good enough, however this would result in a considerable amount of extra work. There are two solutions to this:

- Add more points around the area of interest. In general there are more subtle and effective ways to do this, than by hand, and it is generally inadvisable to use the trapezium rule on an unequally spaced grid.
- Use a higher order method. This is similar to the method of using the splines to perform linear interpolation (see 1.15.4). This is known as quadrature: by fitting sections of polynomial between groups of points, and calculating the area under these, we can approximate the area more accurately. In many cases, they also require the function to be evaluated at less points than the trapezium rule to achieve a given accuracy.

The functions quad and quad8 are two such methods (with quad 8 using a higher order method). `quad('function',a,b,tol)` finds the definite integral of function from `a` to `b`, accurate to the tolerance `tol`. If `tol` is not specified the value `1e-3` is used. MATLAB warns you if the specified tolerance can't be reached.

```
area = quad('humps',-1,2,1e-3)
area =
   26.3450
area = quad8('humps',-1,2,1e-4)
area =
   26.3450
```

These answers are slightly large than those obtained using the trapezium, which is to be expected from our plots. The exact answer is

```
format long
(10*atan(17) + 5*atan(5.5) - 12) - (10*atan(-13) + 5*atan(-9.5) + 6)
format short
ans =
  26.34496047137833
```

For a more detailed discussion of quadrature and numerical integration, see `help quad`, and the numerical analysis references in the bibliography.

1.15.10 Differential Equations

Finding numerical derivatives and solving differential equations is a complex field crossing the boundaries of mathematics, computer science, and engineering. We do not feel that it is appropriate to attempt to cover this field in a few pages of text. Instead, we suggest that you consult the references in the bibliography (and also look at the references contained within

them) **before attempting a numerical solution of any differential equation**. Many research hours have been wasted by starting the numerics before understanding the theory.

Here are some MATLAB functions which you may find useful for finding numerical derivatives and solving differential equations

- `diff` Differences between elements in columns.
- `del2` Five-point discrete Laplacian.
- `ode23` Solve differential equations, low order method.
- `ode45` Solve differential equations, higher order method.

MATLAB 5 includes a suite of programs to solve differential equations, see `odedemo` for more details.

1.16 Data Analysis

In MATLAB data is stored in the form of matrices, and we can perform statistical analysis on this data. In this section we analyze a data set, which was produced in one of the laboratories in the University, and demonstrate how some of the features of MATLAB can be employed.

1.16.1 A Worked Example

In general, we store data sets in columns: each variable is in a different column, and samples of each variable are in the rows. This mimics the way in which a lab book would be set out.

The following data is a measurement of the resistance (in ohms) of a Nickel wire at various temperatures (in Celsius). We are trying to determine whether there is any relationship between the resistance and the temperature of the wire.

```
wire = [18.0, 1.221; 21.5, 1.233; 25.8, 1.258; 30.5, 1.284; 31.5, 1.290;
33.0, 1.298; 34.6, 1.305; 38.0, 1.322; 46.5, 1.368; 50.0, 1.384; 55.6,
1.424; 60.0, 1.445; 70.0, 1.510; 73.2, 1.530; 74.8, 1.540; 76.4, 1.550;
78.0, 1.560; 81.0, 1.580]
```

```
wire =
   18.0000    1.2210
   21.5000    1.2330
   25.8000    1.2580
   30.5000    1.2840
   31.5000    1.2900
   33.0000    1.2980
   34.6000    1.3050
   38.0000    1.3220
   46.5000    1.3680
   50.0000    1.3840
   55.6000    1.4240
   60.0000    1.4450
   70.0000    1.5100
   73.2000    1.5300
   74.8000    1.5400
   76.4000    1.5500
   78.0000    1.5600
```

```
   81.0000    1.5800
```

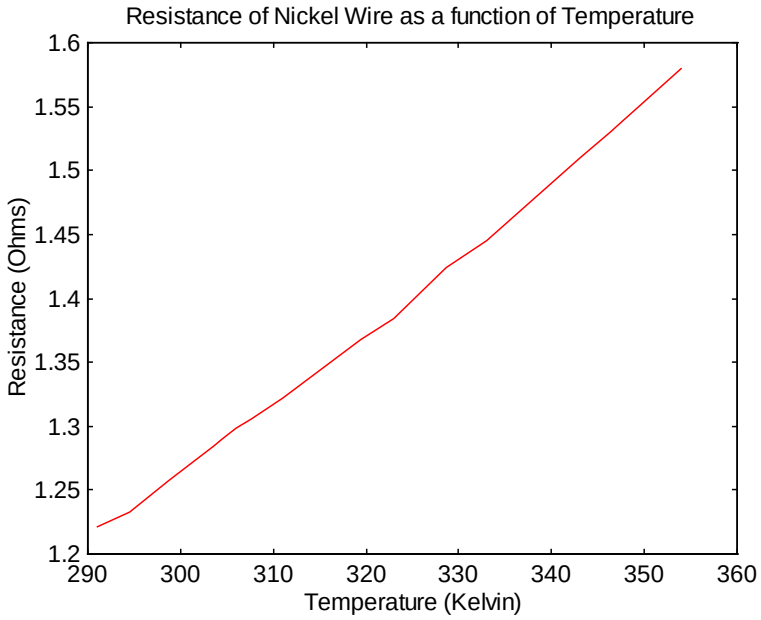
In our calculations, we will require the temperature to be in Kelvin, so we define a new matrix with our data in:

```
wire_K = [wire(:,1) + 273, wire(:,2)]
wire_K =
  291.0000    1.2210
  294.5000    1.2330
  298.8000    1.2580
  303.5000    1.2840
  304.5000    1.2900
  306.0000    1.2980
  307.6000    1.3050
  311.0000    1.3220
  319.5000    1.3680
  323.0000    1.3840
  328.6000    1.4240
  333.0000    1.4450
  343.0000    1.5100
  346.2000    1.5300
  347.8000    1.5400
  349.4000    1.5500
  351.0000    1.5600
  354.0000    1.5800
```

1.16.2 Visualizing the data

In MATLAB, it is easy to visualize the data

```
plot(wire_K(:,1), wire_K(:, 2));
title('Resistance of Nickel Wire as a function of Temperature')
xlabel('Temperature (Kelvin)')
ylabel('Resistance (Ohms)')
```



1.16.3 Determining Statistical Properties of the Data

We can determine the range of temperatures and resistances measured using:

```
range = max(wire_K) - min(wire_K)
range =
   63.0000    0.3590
```

We have used `max` and `min` find the maximum and minimum of the columns of `wire`. We measured temperatures over a range of 63 ºC, and resistances over a range of 0.3590 ohm. MATLAB also allows us to determine the row index of the maximum (or minimum) value in each column.

```
[maximum, index] = max(wire_K)
maximum =
  354.0000    1.5800
index =
    18    18
```

In this case, we have taken eighteen data samples, and the final temperature and resistance yielded the largest readings.

The average resistance of the wire is given by:

```
mean(wire_K(:,2))
ans =
    1.3946
```

where we have specified that we only require the mean of the second column of data. To determine the correlation between the columns in the data, we use the correlation matrix:

```
corrcoef(wire_K)
ans =
    1.0000    0.9992
    0.9992    1.0000
```

The `(i, j)` entry of this matrix gives the correlation between column `i` and `j` of the data. The correlation function quantifies any link we may see when the columns of a matrix are plotted against each other. The entries are between -1 and +1, when -1 means strong negative correlation, and +1 means strong positive correlation. This diagonals are +1, since there is naturally correlation between a number and itself. In this case there seems to be a strong positive correlation between temperature and resistance.

1.16.4 Other Statistical Properties of Data

In other cases, we may wish to use some of the following functions:

- `cov(x)` Covariance matrix.
- `cumprod(x)` Cumulative product of columns.
- `cumsum(X)` Cumulative sum of columns.
- `diff(x)` Differences between elements in columns.
- `median(x)` Median value of columns.
- `prod(x)` Product of elements in columns.
- `sort(x)` Sort columns in ascending order.
- `std(x)` Standard deviation of columns.

1.16.5 A Linear Relationship for the data

From the plot of the data, we could fit a linear relationship, of the form

$$\text{resistance} = m \times \text{temperature} + c \qquad (1.16.1)$$

MATLAB provides the `polyfit` function for this (see 1.15.1):

```
format short e
p = polyfit(wire_K(:,1), wire_K(:,2),1)
format short
p =
  5.7523e-003 -4.6293e-001
```

For `n` points, the error in the slope is given by

$$\text{error in slope} = \frac{\sqrt{\text{Var}\left[y_i - f(x_i)\right]}}{\sqrt{\sum\left(x_i - \bar{x}\right)^2}} = \frac{\sigma_{residual}}{\sqrt{n-1} \times \sigma_x} \qquad (1.16.2)$$

We can evaluate the fitted function, $f(x_i)$, at each of the points x_i using `polyval(p, x)`. `p` is the polynomial row vector, and `x` is a set of points at which to evaluate the polynomial. Hence we can determine the error in the slope:

```
residual = polyval(p,wire_K(:,1)) - wire_K(:,2);
slope_error = std(residual)/(sqrt(size(wire_K,1)-1)*std(wire_K(:,1)))
slope_error =
  5.7351e-005
```

The slope of the line is (5.75 ± 0.06) mΩ K^{-1} (3 sf). The error in the intercept is given by

$$\text{error in intercept} = \frac{\sigma_{residual}\sqrt{\overline{x^2}}}{\sqrt{n-1}\times\sigma_x} \qquad (1.16.3)$$

```
intercept_error = slope_error * sqrt(mean(wire_K(:,1).^2))
intercept_error =
    0.0186
```

The intercept of the line is (-463 ± 19) mΩ (3 sf).

Thus we have a relationship

Resistance of Nickel Wire in mΩ =
(5.75 ± 0.06)× temperature - (463 ± 19)

1.16.6 An Power Law Relationship for the data

If we find the slope of the data from the first 9 points, and compare it to the slope from the last 9 points, we see

```
format short e
p_first9 = polyfit(wire_K(1:9,1), wire_K(1:9,2),1)
p_first9 =
  5.2485e-003 -3.0923e-001
p_last9 = polyfit(wire_K(10:18,1), wire_K(10:18,2),1)
format short
p_last9 =
  6.2535e-003 -6.3473e-001
```

Thus the slope changes from 5.25 mΩ K^{-1} (3 sf) for the first nine points, to 6.25 mΩ K^{-1} (3 sf). So we may try a power relationship between resistance and temperature:

$$\text{resistance} = A \times \text{temperature}^{\alpha} \qquad (1.16.4)$$

Taking logs yields:

$$\log(\text{resistance}) = \alpha \times \log(\text{temperature}) + \log(A) \qquad (1.16.5)$$

In MATLAB we have:

```
log_temp = log10(wire_K(:,1));
log_res = log10(wire_K(:,2));
format short e
p = polyfit(log_temp,log_res,1)
format short
p =
  1.3288e+000 -3.1900e+000
```

We can calculate the errors in the power and intercept using:

```
residual = polyval(p,log_temp) - log_res;
pow_error = std(residual)/(sqrt(size(log_temp,1)-1)*std(log_temp))
pow_error =
    0.0112
intercept_error = pow_error * sqrt(mean( log_temp.^2 ) )
```

```
intercept_error =
    0.0282
```

Thus we have a relationship

$$\text{Resistance of Nickel Wire in m}\Omega = 10^{(-3.19 \pm 0.03)} \times \text{temperature}^{(1.33 \pm 0.01)}$$

With the power law fit, the correlation coefficients for the fit of the data has increased slightly:

```
corrcoef([log_temp,log_res])
ans =
    1.0000    0.9994
    0.9994    1.0000
```

Note that there are more rigorous statistical methods to determine what sort of relationship gives a better fit to a given set of data.

1.17 Some Optimisation Tips

Once you have written a piece of MATLAB code and tested it on some small examples, you may want to scale up the problem size, or analyze more data. We have used MATLAB as a test bed for an algorithm, and then translate to another programming language once we are sure that the method works.

Until recently, the only way to do this was by hand. There are now programs which will turn MATLAB into a variety of other languages (including C, Fortran-77, Fortran-90, and the extensions of these languages designed to run on parallel computers). There are also programs that will compile MATLAB code into an executable. This typically results in a speedup of about a factor of 6, since the code is no longer being interpreted line by line.

However, much work can be done by writing good MATLAB code. We know of a code which was being used extensively in an engineering calculation, which ran over 200 times faster when a few of the techniques used below were applied to it! (Unfortunately, it was being used as a test for a MATLAB to Parallel C translator. Once the MATLAB was optimized, it ran too fast to be used as a test of the performance of the translator).

The methods we describe follow our own experiences after receiving a technical support note from the MathWorks (the authors of MATLAB). Knowing these techniques is only half the story; it is also important to recognize when to use them.

1.17.1 Vectorisation and Built-In Functions

When we introduced the `for` loop, we suggested that it should not be used. The vector MATLAB operations are implemented in highly optimized C, and you are unlikely to do better by implementing your own `for` loop versions of these routines.

Other built-in procedures, for example those for finding eigenvalues, are also optimized, and cover many possible cases to ensure that the methods used is numerically stable. They should be used wherever possible.

An advantage of using these comes if you later translate to another programming language. Most computers have efficient implementations of numerical linear algebra packages (such as LINPACK, BLAS libraries, or ESSL libraries), and the calls to MATLAB routines can easily be replaced with calls to numerical library routines.

Using the professional version of MATLAB you can dynamically link to C or FORTRAN subroutines. We will not discuss this further, see the MATLAB External Interface Guide for details of 'MEX Files'.

1.17.2 Subscripting

MATLAB allows two types of subscripting:

- indexed subscripting
- logical subscripting

In indexed subscripting, the values of the subscript are the indices of the matrix whose elements we require:

```
x = 1:2:10
x =
     1     3     5     7     9
require = x([1, 4])
require =
     1     7
```

In logical subscripting, the matrix used to perform the subscripting has the same dimensions as the matrix to be subscripted. The subscript matrix contains 0's and 1's. The elements returned are those which have a 1 in the corresponding subscripted matrix.

```
x = 1:2:10
x =
     1     3     5     7     9
logic = x(logical([0 1 0 1 0]))
logic =

     3     7
```

We will use logical subscripting later in this section.

1.17.3 Array Operations

In MATLAB operations operate on whole matrices. In most programming languages, it is necessary to set up loops to perform vector calculations, such as a dot product:

```
u = zeros(1,7);
s = rand(1,7); t = rand(1,7);
for k = 1:7
    u(k) = s(k) * t(k);
end
u
u =
    0.1342    0.0083    0.2266    0.0217    0.2711    0.3629    0.2218
```

In MATLAB, this calculation proceeds without explicit reference to the array size:

```
s .* t
ans =
    0.1342    0.0083    0.2266    0.0217    0.2711    0.3629    0.2218
```

Whenever possible, you should use array operations. When this is not possible, it is always best to define the ultimate size of any array to be built-up in the loop. Otherwise, since MATLAB operates by interpreting commands line by line, it will spend a lot of time augmenting the array on each iteration.

1.17.4 Boolean Array operations

We can also perform comparison operations on whole arrays. Suppose you wish to determine which numbers in an array are greater than 3, we can use:

```
x = [1, 5, 2, 6, 2, 9, 10, 2, 0];
x > 3
ans =
     0     1     0     1     0     1     1     0     0
```

Since this is an array of the same length as `x`, we can use the logical subscripting features of MATLAB to return the required values:

```
greater_3 = x(x>3)
greater_3 =
     5     6     9    10
```

MATLAB provides two functions to perform Boolean AND and OR operations across a whole vector: `all` and `any` respectively.

```
if any(x> 9)
   disp('Elements greater than 9 in matrix')
end
Elements greater than 9 in matrix
```

You can also compare two vectors of the same size using the Boolean operators, resulting in expressions such as:

```
x = [2, 4, 6, 8, 10, 12]
x =
     2     4     6     8    10    12


y = [2, 5, 6, 8, 9, 11]
y =
     2     5     6     8     9    11
(x==y) & (y>6)
ans =
     0     0     0     1     0     0
```

Since MATLAB uses IEEE arithmetic, there are special values to denote overflow, underflow, and undefined operations: `Inf`, `-Inf`, and `NaN`, respectively. `Inf` and `-Inf` can be tested for normally:

```
test = Inf == Inf
test =
     1
```

`test` is true, since the two infinities are equal. By the IEEE standard, `NaN` is never equal to anything (even other `NaN`'s). MATLAB provides two special Boolean operators, `isnan` and `isinf`, to test for these values.

1.17.5 Constructing Matrices from Vectors

To select specific elements from a matrix, it is not always necessary to write out a. In this example we take elements [2, 3, 4, 6, 7, 8, 10, 11, 12...] from a matrix using:

```
x = 1:20;
y = reshape(x, 4, length(x)/4);
index = y(2:4,:);
index = index(:)'
index =
  Columns 1 through 12
     2     3     4     6     7     8    10    11    12    14    15    16
  Columns 13 through 15
    18    19    20
```

In 1.10.1, we created a matrix with all entries 17:

```
ones(2, 3)*17
ans =
    17    17    17
    17    17    17
```

To duplicate a vector of size `n` by `1`, `k` times, the first column of the vector is indexed `k` times:

```
x = (1:4)';
a = x(:,ones(4,1))
a =
     1     1     1     1
     2     2     2     2
     3     3     3     3
     4     4     4     4
```

Thus to create a matrix with all the same entry, we can avoid the matrix multiplication:

```
x = 17;
x = x(ones(2,3))
x =
    17    17    17
    17    17    17
```

We can use the same method to duplicate row vectors by swapping the subscripts. It can also be used to duplicate rows or columns of a matrix, provided:

- that you are not selecting the first row or column, or
- that the resulting matrix is not the same size as the original matrix.

If these conditions both hold, then there is an ambiguity. The subscripting vector, which is all ones, could represent logical subscripting where all columns or rows are chosen once, or it could represent indexed subscripting where just the first column or row is chosen several times. If there is ambiguity, MATLAB chooses to view the intent as logical subscripting:

```
x = rand(3)
x =
    0.2855    0.0704    0.9642
```

```
    0.1061    0.3048    0.6130
    0.3135    0.6970    0.4901
```

In the next case, the subscript to the array is viewed as a logical subscript:

```
x(:,ones(1,3))
ans =
    0.2855    0.2855    0.2855
    0.1061    0.1061    0.1061
    0.3135    0.3135    0.3135
```

However, in the next example the subscripting array is not the same size as the array, and so the first column is duplicated:

```
x(:,ones(1,2))
ans =
    0.2855    0.2855
    0.1061    0.1061
    0.3135    0.3135
```

1.17.6 Constructing Special Matrices

The Toeplitz and Hankel matrix functions can be used to create matrices with particular structures.

```
help toeplitz
 TOEPLITZ Toeplitz matrix.
    TOEPLITZ(C,R) is a non-symmetric Toeplitz matrix having C as its
    first column and R as its first row.

    TOEPLITZ(C) is a symmetric (or Hermitian) Toeplitz matrix.

    See also HANKEL.
help hankel
 HANKEL Hankel matrix.
    HANKEL(C) is a square Hankel matrix whose first column is C and
    whose elements are zero below the first anti-diagonal.

    HANKEL(C,R) is a Hankel matrix whose first column is C and whose
    last row is R.

    Hankel matrices are symmetric, constant across the anti-diagonals,
    and have elements H(i,j) = P(i+j-1) where P = [C R(2:END)]
    completely determines the Hankel matrix.

    See also TOEPLITZ.
```

To create a matrix in which each row is the row above shifted cyclically to the right, we specify an initial row and use:

```
x = (1:4)';
```

```
cycle_row = toeplitz(x, x([1, length(x):-1:2]))
cycle_row =
     1     4     3     2
     2     1     4     3
     3     2     1     4
     4     3     2     1
```

The Hankel function allows us to do the same, but with rows (and the cycle shifts elements upwards):

```
x = (1:4)';
cycle_column = hankel(x, x([length(x), 1:length(x)-1]))
cycle_column =
     1     2     3     4
     2     3     4     1
     3     4     1     2
     4     1     2     3
```

By constructing matrices efficiently and in a general way (the examples above will work just as well if `x = (1:10)'`), they can be used in other processing operations.

1.17.7 Functions of two variables

We can often learn a lot by studying the way in which MATLAB implements a function. In this case we examine the way in which MATLAB implements `meshgrid`, which is used to evaluate functions of two variables across the domain specified by two matrices. Here is a version of the meshgrid implementation to evaluate `x*exp(-x^2-y^2)` over the range `-2 < x < 2` and `-2 < y < 2`:

```
xx = (-2:2);
yy = (-1.5:0.5:1.5)';
X = xx(ones(size(yy)), : );
Y = yy(:,ones(size(xx)) );
f = X .* exp(-X.^2-Y.^2)
f =
   -0.0039   -0.0388         0    0.0388    0.0039
   -0.0135   -0.1353         0    0.1353    0.0135
   -0.0285   -0.2865         0    0.2865    0.0285
   -0.0366   -0.3679         0    0.3679    0.0366
   -0.0285   -0.2865         0    0.2865    0.0285
   -0.0135   -0.1353         0    0.1353    0.0135
   -0.0039   -0.0388         0    0.0388    0.0039
```

The trick of repeating the first column, or row has been used to produce a mesh of points over which to evaluate the function and is equivalent to using:

```
[X, Y] = meshgrid(-2:2, -1.5:0.5:1.5);
f = X .* exp(-X.^2-Y.^2)
f =
   -0.0039   -0.0388         0    0.0388    0.0039
   -0.0135   -0.1353         0    0.1353    0.0135
```

```
   -0.0285   -0.2865         0    0.2865    0.0285
   -0.0366   -0.3679         0    0.3679    0.0366
   -0.0285   -0.2865         0    0.2865    0.0285
   -0.0135   -0.1353         0    0.1353    0.0135
   -0.0039   -0.0388         0    0.0388    0.0039
```

Using the matrix multiplication operator, you can sometimes avoid having to define intermediate matrices.

```
x = (-2:2);
y = (-1.5:0.5:1.5);
f = x'*y
f =
    3.0000    2.0000    1.0000         0   -1.0000   -2.0000   -3.0000
    1.5000    1.0000    0.5000         0   -0.5000   -1.0000   -1.5000
         0         0         0         0         0         0         0
   -1.5000   -1.0000   -0.5000         0    0.5000    1.0000    1.5000
   -3.0000   -2.0000   -1.0000         0    1.0000    2.0000    3.0000
```

There are also cases where sparse matrices allow more efficient use of storage space, and also allow very efficient algorithms. We discuss this in more detail in 1.17.9.

1.17.8 Redundancy

If you may have a set of values with duplicates, you may want to know:

- What is the ‘core’ set of values?
- How many times were each of the core values duplicated in the original set?

There are several MATLAB functions which can help with this:

- `diff` finds difference between adjacent pairs of entries in a vector.
- `find` returns the indices of the non-zero, non NaN elements of a matrix.
- `sort` returns a sorted array.
- `max` finds maximum entry of an array.
- `min` finds minimum entry of an array.

Firstly, we will address: “what is the ‘core’ set of values?” If we sort an array of values, any repeated elements will be adjacent, any NaNs will be at the end.

```
x = [1, NaN, 1, 3; 8, 6, 4, 8; 3, 6, 6, 4; 0, 8, 3, 1]
x =
     1   NaN     1     3
     8     6     4     8
     3     6     6     4
     0     8     3     1
y = sort(x(:))'
y =
```

```
  Columns 1 through 12
     0     1     1     1     3     3     3     4     4     6     6     6
  Columns 13 through 16
     8     8     8   NaN
```

The difference between adjacent pairs in this vector will be zero where elements are repeated:

```
z = diff(y)
z =
  Columns 1 through 12
     1     0     0     2     0     0     1     0     2     0     0     2
  Columns 13 through 15
     0     0   NaN
```

We could use the places where this matrix is zero to determine the core elements of `y`. However the vector `z` has one less element than `y`. To remedy this, we actually define `z` as follows.

```
z = diff([y, max(y)+1])
z =
  Columns 1 through 12
     1     0     0     2     0     0     1     0     2     0     0     2
  Columns 13 through 16
     0     0   NaN   NaN
```

We now select the elements we require using

```
core = y(z~=0)
core =
     0     1     3     4     6     8   NaN
```

We did not use the `find` function mentioned above, since it does not return indices for entries which are NaNs, and so elements would have been missed.

Secondly, we will consider: “how many times were each of the core values duplicated in the original set?”

After the vector `x` is sorted, we can use `find` to determine the where the resulting distribution changes. This time we must not append a NaN to the list

```
x = [0, 1, 1, 3; 8, 6, 4, 8; 3, 6, 6, 4; 0, 8, 3, 1]
x =
     0     1     1     3
     8     6     4     8
     3     6     6     4
     0     8     3     1
y = sort(x(:))'
y =
  Columns 1 through 12
     0     0     1     1     1     3     3     3     4     4     6     6
  Columns 13 through 16
     6     8     8     8
```

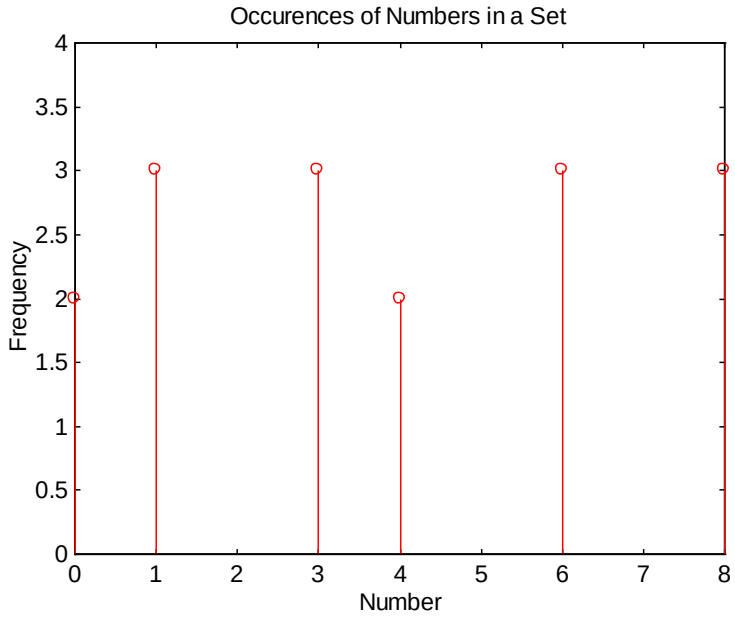
```
z = diff([y, max(y)+1])
z =
  Columns 1 through 12
     0     1     0     0     2     0     0     1     0     2     0     0
  Columns 13 through 16
     2     0     0     1
find(z)
ans =
     2     5     8    10    13    16
```

The difference between adjacent elements will give the number of counts of each element. Note that we must prepend a `0` to the list, so that `diff` returns how many times the first element was repeated

```
repeats = diff([0,find(z)])
repeats =
     2     3     3     2     3     3
core = y(find(z))
core =
     0     1     3     4     6     8
```

A plot shows our results

```
stem(core,repeats)
xlabel('Number');
ylabel('Frequency');
title('Occurences of Numbers in a Set');
axis([0, 8, 0,4]);
```



The above procedure breaks down if there are NaNs or Infs in the input set. These should be removed first, and then totaled separately using

- `repeats_nans = sum(isnan(x(:)))`

- `repeats_infs = sum(isinf(x(:)))`

1.17.9 Sparse Matrices

Sparse matrices contain a large number of zero entries. You can save memory and avoid storing the large number of zeros in the matrix by using MATLAB's extensive sparse matrix functions.

In this section we discuss a way to exploit sparse matrices to solve the problem above. If we know that the elements of the set `x` are integers larger than 1, let us set up a list of all the integers covering the range of `x` values:

```
x2 = fix(rand(1,50)*10)+1;
y = min(x2):max(x2)
y =
     1     2     3     4     5     6     7     8     9    10
```

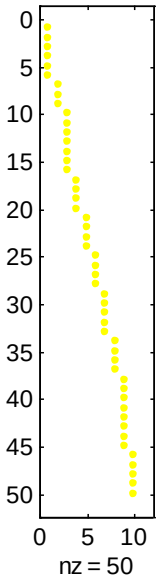
We could now define a matrix `a(i,j)`, in which

`a(i,j) = 1` if `x2(i) = y(j)`, and

`a(i,j) = 0` otherwise.

Most of the entries in this matrix will be zero, the others will be 1. We can set up the following sparse matrix, and visualise it using:

```
x = sort(x2(:));
big = sparse(1:length(x),x(:),1,length(x),max(x)+1);
spy(big)
```



By summing the entries in the columns of `big`, we determine the number of each element present in the original set:

```
sum(big)
ans =
   (1,1)         3
   (1,2)         4
   (1,3)        10
```

```
(1,4)        5
(1,5)        3
(1,6)        4
(1,7)        2
(1,8)        5
(1,9)        7
(1,10)       7
```

MATLAB provides many functions to handle and visualise sparse matrices. If your matrices contain only a relatively small number of non-zero entries, then you should investigate sparse matrix formulations. You can use all the normal matrix handling routines on sparse matrices, but use considerably less memory. We suggest that you read the `help` entry for `sparse` (`help sparse`), and examine the other related functions, which are listed there.

1.17.10 Conclusion

There are many tricks that you can employ to write efficient MATLAB code. You should aim to use MATLAB's built-in functions wherever possible, since these will assist in translating the code to another programming language.

By examining the code for MATLAB's own functions (using `type`), you will pick up many other useful techniques.

1.18 3-D Graphics

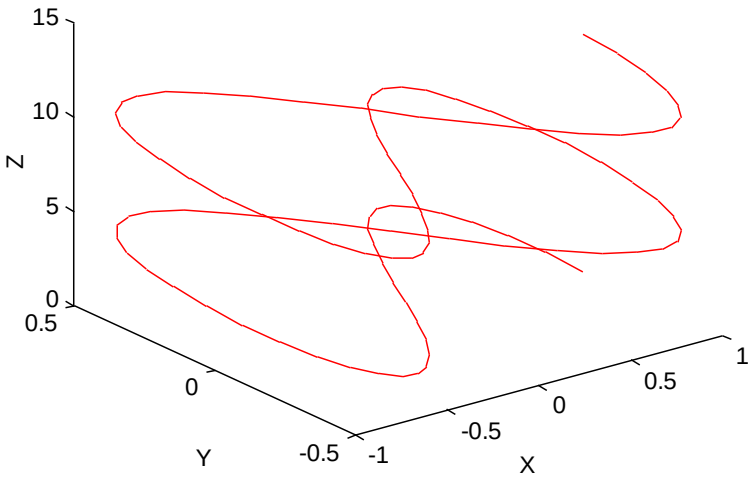
Visualizing three dimensional data can be hard. MATLAB provides many ways to assist in this. In this section, we will briefly introduce some of MATLAB's three dimensional plotting features and discover how to produce the plot on the front cover of this booklet.

Many of the three dimensional plots do not look their best in black and white. For this chapter we particularly recommend that you play with MATLAB, examine the output, experiment with different plots and explore the help system.

1.18.1 An extension of two dimensional plotting

`plot3` is the three dimensional analogue of the `plot` command for two dimensions. `plot3(x1, y1, z1, s1, x2, y2, z2, s2, ..)` plots each of the triples `(x, y, z)` with the formatting style `s`. To give a label to the `z` axis, you use `zlabel('text')`.

```
theta = linspace(0, 4*pi, 100);
plot3( cos(theta), sin(theta).*cos(theta),theta)
title('Eight Curve in three dimensions')
xlabel('X'), ylabel('Y'), zlabel('Z');
```



There are also three dimensional analogues of `fill`, which becomes `fill3`, and `text`, in which you specify a the `(x, y, z)` triple at which you want the text to appear. `axis([xmin, xmax, ymin, ymax, zmin, zmax])` scales the axes of a three dimensional plot. A particularly useful command is `rotate3d`, which enables you to rotate a figure interactively using the mouse to adjust the view point. This also works with 2D figures.

1.18.2 Mesh Plots

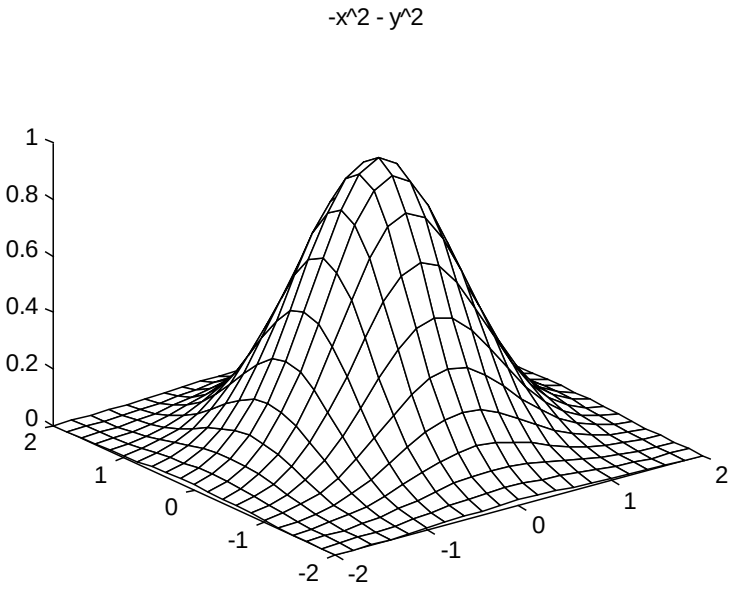
The `mesh(X,Y,Z,C)` function plots a wire grid plot of the function defined by the matrices `X`, `Y`, and `Z`, using the colour defined in `C`. MATLAB provides the `meshgrid` function to set up two matrices of `x` and `y` coordinates at which the function is evaluated. `[X, Y] = meshgrid[x, y]` produces a matrix `X`, with rows which are a copy the vector `x`, and a matrix `Y`, whose columns which are a copy of the vector `y`. If `y` is not specified, then `x` is copied across the columns of `Y`. We then evaluate the function we wish to plot at the array of points `[X, Y]`.

```
[X,Y] = meshgrid(-2 : 1 : 2)
X =
    -2    -1     0     1     2
    -2    -1     0     1     2
    -2    -1     0     1     2
    -2    -1     0     1     2
    -2    -1     0     1     2
Y =
    -2    -2    -2    -2    -2
    -1    -1    -1    -1    -1
     0     0     0     0     0
     1     1     1     1     1
     2     2     2     2     2
Z = exp(-X.^2 - Y.^2)
Z =
    0.0003    0.0067    0.0183    0.0067    0.0003
```

```
    0.0067    0.1353    0.3679    0.1353    0.0067
    0.0183    0.3679    1.0000    0.3679    0.0183
    0.0067    0.1353    0.3679    0.1353    0.0067
    0.0003    0.0067    0.0183    0.0067    0.0003
```

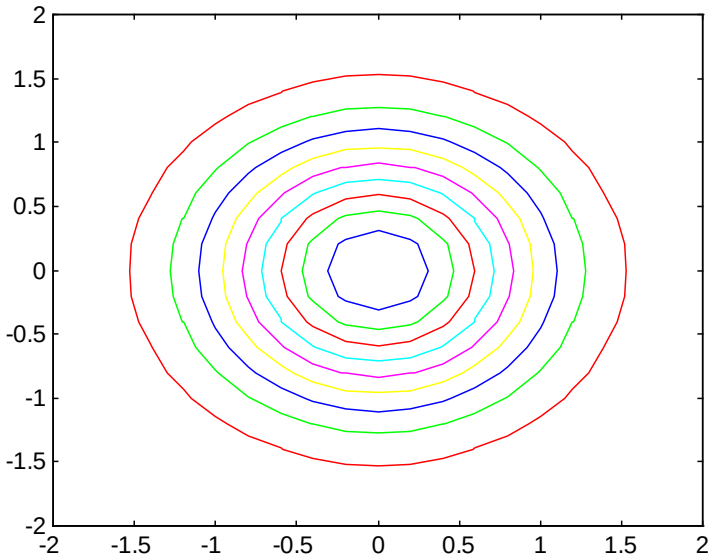
In practice, we use a much finer grid to give a smooth structure, since MATLAB uses linear interpolation between adjacent points.

```
[X,Y] = meshgrid(-2 : 0.2 : 2);
Z = exp(-X.^2 - Y.^2);
mesh(X,Y,Z);
title('-x^2 - y^2')
```



If you do not specify a colour, `C`, MATLAB uses the height to scale the `colormap` (see 1.18.3). You can also plot the contour map of a function analogously using `contour(X, Y, Z, C)`.

```
contour(X, Y, Z)
```



There are many more options for plotting contours and meshes, most of which are best examined ‘on screen’. These include:

- `comet3` An animated plot (!)
- `contour` Contour plot.
- `mesh` Wire meshes.
- `quiver` Gradient plot of function.
- `slice` Volumetric slice plot.
- `surf` Shaded surface plot.

1.18.3 Colour Maps

The `colormap` (note the spelling) scales the active set of colours which are used by the current figure. MATLAB provides various built-in colour maps to colour code data on plots. `colormap(cool(16))` uses a colour map of 16 shades of colour ranging from cyan to magenta to represent values on a plot. Other MATLAB colour maps are:

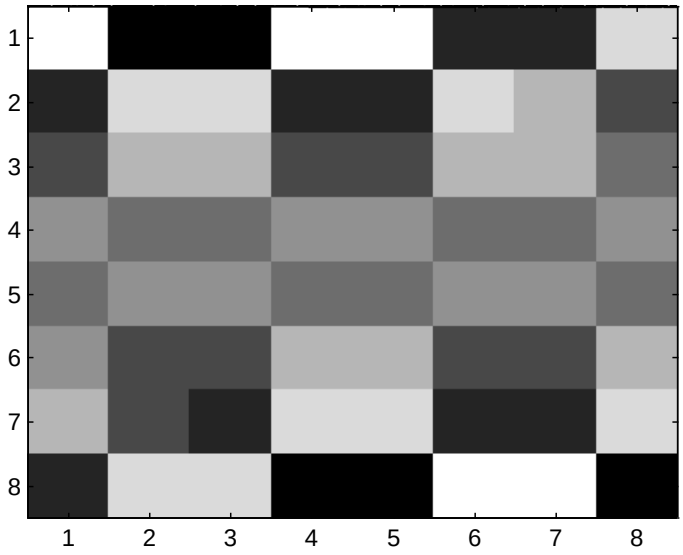
- `hsv` Hue-saturation-value color map.
- `gray` Linear gray-scale color map.
- `hot` Black-red-yellow-white color map.
- `cool` Shades of cyan and magenta color map.
- `bone` Gray-scale with a tinge of blue color map.
- `copper` Linear copper-tone color map.
- `pink` Pastel shades of pink color map.
- `prism` Prism color map.
- `jet` A variant of HSV.
- `flag` Alternating red, white, blue, and black color map.

See `help color` and the other functions listed there for more details about using colour in MATLAB plots.

1.18.4 Back to two dimensions

Sometimes it is desirable to represent three dimensional data using a two dimensional structure: a contour plot is one way of doing this. MATLAB provides `pcolor` and `imagesc` to allow you to colour a two dimensional grid proportional to the value of the function. `imagesc(Z)` colours the cells of `Z` with their values, having scaled them to fill all `colormap`. `pcolor(Z)` specifies the colours of the vertices of `Z`.

```
imagesc(magic(8))
colormap(gray(8))
```



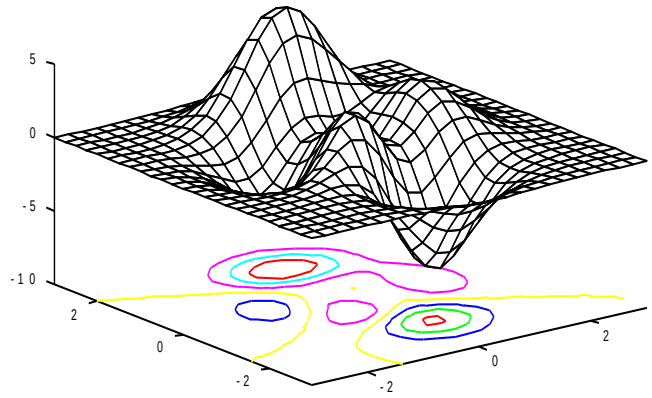
1.18.5 A helpful suggestion

To discover more about MATLAB's three dimensional plotting power, we suggest you type `help plotxyz` (or just `help`, and then ask for `help` on the directory name under which the plotting commands have been installed) and explore.

1.19 About the front cover

On the front cover of this booklet, we have produced a plot which shows both the contours and mesh for the `peaks` function, using `meshc`, which is from the help for `mesh` in the MATLAB User's Guide.

```
[X, Y] = meshgrid(-3:0.25:3);
Z = peaks(X, Y);
meshc(X, Y, Z);
axis([-3, 3, -3, 3, -10, 5])
```



Appendix A MATLAB Summary

A.1 Arithmetic Operators: + - * / \ ^ '

Purpose These perform matrix and array arithmetic.

Synopsis

```
A+B
A-B
A*B   A.*B
A/B   A./B
A^B   A.^B
A'    A.'
```

Description MATLAB has two different types of arithmetic operations. Matrix arithmetic operations are defined by the rules of linear algebra. Array arithmetic operations are carried out element-by-element. The period or decimal point character (`.`) distinguishes the array operations from the matrix operations. However, since the matrix and array operations are the same for addition and subtraction, the character pairs `.+` and `.-` are not used. The matrix arithmetic operations are also used for pure scalars (which can be thought of as 1 by 1 matrices).

`+`	**Addition**. `A+B` adds `A` and `B`. `A` and `B` must have the same dimensions, unless one is a scalar. `A` scalar can be added to a matrix of any dimension.
`-`	**Subtraction**. `A-B` subtracts `B` from `A`. `A` and `B` must have the same dimensions, unless one is a scalar. A scalar can be subtracted from a matrix of any dimension.
`*`	**Matrix multiplication**. `A*B` is the linear algebraic product of the matrices `A` and `B`. The number of columns of `A` must equal the number of rows of `B`, unless one of them is a scalar. A scalar can multiply a matrix of any dimension.
`.*`	**Array multiplication**. `A.*B` is the element-by-element product of the arrays `A` and `B`. `A` and `B` must have the same dimension, unless one of them is a scalar.
`\`	**Backslash** or **matrix left division**. If `A` is a square matrix, `A\B` is roughly the same as `inv(A)*B`, except it is computed in a different way. If `A` is an `n` by `n` matrix and `B` is a column vector with `n` components, or a matrix with several such columns, then `X=A\B` is the solution to the equation `AX = B` computed by Gaussian elimination. A warning message prints if `A` is badly scaled or nearly singular. If `A` is an `m` by `n` matrix with `m ~= n` and `B` is a column vector with `m` components, or a matrix with several such columns, then `X=A\B` is the solution in the least squares sense to the under- or over-determined system of equations `AX = B`. The effective rank, `k`, of `A`, is determined from the QR decomposition with pivoting. A solution `X` is computed which has at most `k` nonzero components per column. If `k < n`, this is usually not be the same solution as `pinv(A)*B`, which is the least squares solution with the smallest residual norm, $\|AX - B\|$.
`.\`	**Array left division**. `A.\B` is the matrix with elements `B(i,j)/A(i,j)`. `A` and `B` must have the same dimensions, unless one of them is a scalar.
`/`	**Slash** or **matrix right division**. `B/A` is roughly the same as `B*inv(A)`. More precisely, `B/A` = `(A'\B')'`. See `\`.

`./`	**Array right division**. `A./B` is the matrix with elements `A(i,j)/B(i,j)`. `A` and `B` must have the same dimensions, unless one of them is a scalar.
`^`	**Matrix power**. `A^p` is `A` to the power `p`, if `p` is a scalar. If `p` is an integer, the power is computed by repeated multiplication. If the integer is negative, `A` is inverted first. For other values of `p`, the calculation involves eigenvalues and eigenvectors, such that if `[V,D] = eig(A)`, then `A^p = V*D.^p/V.` If `a` is a scalar and `P` is a matrix, `x^P` is `x` raised to the matrix power `P` using eigenvalues and eigenvectors. `X^P`, where `X` and `P` are both matrices, is an error.
`.^`	**Array power**. `A.^B` is the matrix with elements `A(i,j)` to the power `B(i,j`. `A` and `B` must have the same dimensions, unless one of them is a scalar.
`'`	**Matrix transpose**. `A'` is the linear algebraic transpose of `A`. For complex matrices, this involves the complex conjugate transpose.
`.'`	**Array transpose**. `A.'` is the array transpose of `A`. For complex matrices, this does not involve conjugation.

A.2 Relational Operators: < ≤ > ≥ == ~=

Purpose Relational operators

Synopsis

```
A < B
A > B
A <= B
A >= B
A == B
A ~= B
```

Description The relational operators are `<`, `≤`, `>`, `≥`, `==`, `~=`, and `~=`. Relational operators perform element-by-element comparisons between two matrices. They return a matrix of the same size, with elements set to 1 where the relation is true, and elements set to 0 where it is not.

The operators `<`, `≤`, `>`, and `≥` use only the real part of their operands for the comparison. The operators `==` and `~=` test real and imaginary parts.

The relational operators have precedence midway between the logical operators (except `~`) and the arithmetic operators.

To test if two strings are equivalent, use `strcmp`, which allows vectors of dissimilar length to be compared.

Examples If one of the operands is a scalar and the other a matrix, the scalar expands to the size of the matrix. For example, the two pairs of statements:

```
X = 5; X >= [1, 2, 3; 4, 5, 6; 7, 8, 10]
ans =
     1     1     1
     1     1     0
     0     0     0
```

```
X = 5*ones(3,3); X >= [1, 2, 3; 4, 5, 6; 7, 8, 10]
ans =
     1     1     1
     1     1     0
     0     0     0
```

produce the same result.

A.3 Logical Operators: & | ~

Purpose Logical operators

Synopsis

```
A & B
A | B
~A
```

Description The symbols `&`, `|`, and `~` are the logical operators AND, OR, and NOT. They work element-wise on matrices, with 0 representing FALSE and anything nonzero representing TRUE. `A & B` does a logical AND, `A | B` does a logical OR, and `~A` complements the elements of `A`. The function `xor(A,B)` implements the exclusive OR operation.

Inputs		AND	OR	XOR
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

The logical operators `&` and `|` have the lowest precedence, with arithmetic operators and relational operators being higher. The logical operator `~` has the same precedence as the arithmetic operators.

The precedence for the logical operators with respect to each other is:

1) NOT has the highest precedence.
2) AND and OR have equal precedence, and are evaluated from left to right.

Examples Here are two scalar expressions that illustrate precedence relationships for arithmetic, relational, and logical operators:

```
a = 1 & 0 + 3, b = 3 > 4 & 1
```

They are equivalent to:

```
a = 1 & (0 + 3), b = (3 > 4) & 1
a =
     1
b =
     0
```

Here are two examples that illustrate the precedence of the logical operators to each other

```
c = 1 | 0 & 0, d = 0 & 0 | 1
```

```
c =
     0
d =
     1
```

A.4 Special Characters: [] () = ' . , ; % !

Purpose Special characters

Synopsis `[] () = ' . , ; % !`

Description

`[]`	**Brackets** are used to form vectors and matrices. `[6.9 9.64 sqrt(-1)]` is a vector with three elements separated by blanks. `[6.9, 9.64, i]` is the same thing. `[1 + j 2-j 3]` and `[1 +j 2 -j 3]` are not the same. The first has three elements, the second has five. `[11 12 13; 21 22 23]` is a 2-by-3 matrix. The semicolon ends the first row. Vectors and matrices can be used inside brackets. `[A B; C]` is allowed if the number of rows of `A` equals the number of rows of `B` and the number of columns of `A` plus the number of columns of `B` equals the number of columns of `C`. This rule generalises to allow fairly complicated constructions. `A = []` stores an empty matrix in `A`.
`()`	**Parentheses** are used to indicate precedence in arithmetic expressions in the usual way. They are used to enclose arguments of functions in the usual way. They are also used to enclose subscripts of vectors and matrices in a manner somewhat more general than usual. If `X` and `V` are vectors, then `X(V)` is `[X(V(1)), X(V(2)), .., X(V(n))]`. The components of `V` are rounded to nearest integers and used as subscripts. An error occurs if any such subscript is less than 1 or greater than the dimension of `X`. Some examples are `X(3)` is the third element of `X`. `X([ 1 2 3 ])` is the first three elements of `X`. `X([ sqrt(2) sqrt(3) 2*atan(1)])` also returns the first three elements of `X`. If `X` has `n` components, `X(n:-1:1)` reverses them. The same indirect subscripting works in matrices. If `V` has `m` components and `W` has `n` components, then `A(V,W)` is the `m` by `n` matrix formed from the elements of `A` whose subscripts are the elements of `V` and `K`. For example: `A([1,5],:) = A([5,1],:)` interchanges rows 1 and 5 of `A`.
`=`	Used in **assignment** statements. `==` is the relational EQUALS operator. See 0.
`'`	**Matrix transpose**. `X'` is the complex conjugate transpose of `X`. `X.'` is the non-conjugate transpose. **Quote**. `'any text'` is a vector whose components are the ASCII codes for the characters. A quote within the text is indicated by two quotes.

.	**Decimal point**. `314/100, 3.14` and `.314e1` are all the same. Element-by-element operations are obtained using `.*`, `.^`, `./`, or `.\`. Three or more points at the end of a line indicate continuation.
,	**Comma**. Used to separate matrix subscripts and function arguments. Used to separate statements in multi-statement lines. For multi-statement lines, the comma can be replaced by a semicolon to suppress printing.
;	**Semicolon**. Used inside brackets to end rows. Used after an expression or statement to suppress printing or separate statements.
%	**Percent**. The percent symbol denotes a comment; it indicates a logical end of line. Any following text is ignored.
!	**Exclamation point**. Indicates that the rest of the input line is a command to the operating system.

A.5 Colon :

Purpose Create vectors, matrix subscripting, and `for` iterations.

Description The colon is one of the most useful operators in MATLAB. It can create vectors, subscript matrices, and specify `for` iterations.

The colon operator uses the following rules to create regularly spaced vectors:

- `j:k` is the same as `[j ,j+1 , . ., k]`
- `j:k` is empty if `j>k`
- `j:i:k` is the same as `[j ,j+i , j+2i, . ., k]`
- `j:i:k` is empty if `i>0` and `j>k` or if `i<0` and `j<k`

Below are the definitions that govern the use of the colon to pick out selected rows, columns, and elements of vectors and matrices:

- `A(:,j)` is the j-th column of `A`
- `A(i,:)` is the i-th row of `A`
- `A(:,:)` is the same as `A`
- `A(:,j:k)` is the `A(j)`, `A(:,j+1)` , . . , `A(k)`
- `A(:)` is all the elements of `A`, regarded as a single column. On the left side of an assignment statement, `A(:)` fills `A`, preserving its shape from before.

Examples Using the colon with integers:

```
D = 1:4
D =
     1     2     3     4
```

Using two colons to create a vector with arbitrary real increments between the elements,

```
E = 0:.1:.5
E =
         0    0.1000    0.2000    0.3000    0.4000    0.5000
```

A.6 Order of Precedence for Operators

The following table gives the order of precedence for arithmetic, logical and relational operators. Those at the top of the table take the highest precedence

^ .^ ' .'
* / \ .* ./ .\
+ - ~ +(number) -(number)
: > < >= <= == ~=
| &