Introduction to Matlab

Matthew Roughan
<matthew.roughan@adelaide.edu.au>
Applied Mathematics, School of Mathematical Sciences
The University of Adelaide

April 11th, 2010

Preface

These notes grew out of a set being used for a course called Scientific Computing. In that course, three programming languages were taught: Excel, MATLAB and C, with some emphasis on comparison of the advantages, disadvantages, and commonalities between the three. This set of notes was drawn from the MATLAB component of that course, as we often have a need to teach new students some elements of MATLAB , or to refresh their memory. However, as it was only one component of a larger course, these notes are far from complete, and while they may comprise a suitable set for a student just starting MATLAB , there are plenty of other books and on-line reference materials that are more substantial. Also, a reader may notice comparisons with Excel or C appearing at various places, due to the structure of the original course. Despite, this, these notes should contain a reasonable introduction to programming (specifically programming for scientific or numerical purposes) in MATLAB .

Matthew Roughan

Contents

1 MATLAB Fundamentals **1**

1.1 Reference books for MATLAB 1
1.2 Getting started 2
1.3 How a program works 2
1.4 Variables 3
1.5 Script M-files 5
1.6 Useful features in the MATLAB window 6
1.7 Punctuation! 7
1.8 Programming style 7

2 Vectors and matrices **9**

2.1 Initialising row vectors: explicit lists 9
2.2 Initialising row vectors: the colon operator : 10
2.3 Column vectors 10
2.4 Transposing vectors 10
2.5 Concatenation 11
2.6 Subscripts 11
2.7 Matrices 12
2.8 MATLAB and matrices 13
2.9 Solving linear equations with MATLAB 15
2.10 Strings 16
2.11 Multi-dimensional arrays 17

3 MATLAB as a big calculator **19**

3.1 Numbers 19
3.2 Operators, expressions and statements 20

3.3 Precedence of operators 23
3.4 Vectorisation of formulae 24

4 Input/Output **27**
4.1 `disp` 27
4.2 The `format` statement 28
4.3 `fprintf` 28
4.4 Advanced I/O 32

5 Program flow control **35**
5.1 Making decisions with `if` 35
5.2 Repetition with `for` 41
5.3 Non-deterministic repetition with `while` 48
5.4 Programming style 50
5.5 Other MATLAB statements 51

6 Commonly used functions and variables **53**
6.1 Constants 53
6.2 Elementary Mathematical Functions 54
6.3 Simple Vector/Matrix functions 55
6.4 Set functions 56
6.5 Test functions 56
6.6 String functions 56
6.7 Dates and times 57
6.8 Utility functions 57
6.9 More information 58

7 Graphics **59**
7.1 Basic two-dimensional plots 59
7.2 Decorating the figure 59
7.3 Multiple plots 64
7.4 Printing graphs 67
7.5 Colours 68
7.6 Advanced two-dimensional plots 69
7.7 Three-dimensional plots 74

8 Defining Functions with M-files **79**
8.1 Some examples 79
8.2 The basic rules for function files 81
8.3 Function names as input variables with `feval` 86
8.4 Inline and anonymous functions 87
8.5 Recursion 87

9 0-1 vectors **93**
9.1 Combining logical and numerical vectors 94
9.2 Additional tests 98
9.3 The `find` function 98

10 The Optimization Toolbox **101**
10.1 Linear Programming 101
10.2 Why use MATLAB for optimization 103

11 MATLAB Roundup **105**
11.1 More stuff 105
11.2 Limitations 105
11.3 Summary 106

Chapter 1

MATLAB Fundamentals

We can characterise MATLAB as follows. It is

- **Imperative:** we tell the computer what to do.
- **Procedural:** a program is written as a series of tasks (procedures) to do in a specified order.
- **High-level:** MATLAB is written in a high-level programming language which resembles a mixture of English and mathematics. The exact form has to differ from both because it needs to be more precise than English (computer are fast but stupid), and because a typewriter keyboard (which we will use for entering programs) has limited keys.
- **Interpreted:** MATLAB uses an interpreter to translate our high-level commands into something the machine can do. It does the translation (almost) on the spot when we type a command. A MATLAB interpreter exists for most common computing environments, including Windows, MacOS, and Linux, so MATLAB code is very portable (if it is written carefully).

1.1 Reference books for MATLAB

This course draws from the following text which is available in the Reserve section of the Barr Smith library.

1. Hahn, B.D. *Essential* MATLAB *for Scientists and engineers* (Arnold, London) 1997/2002/2007.

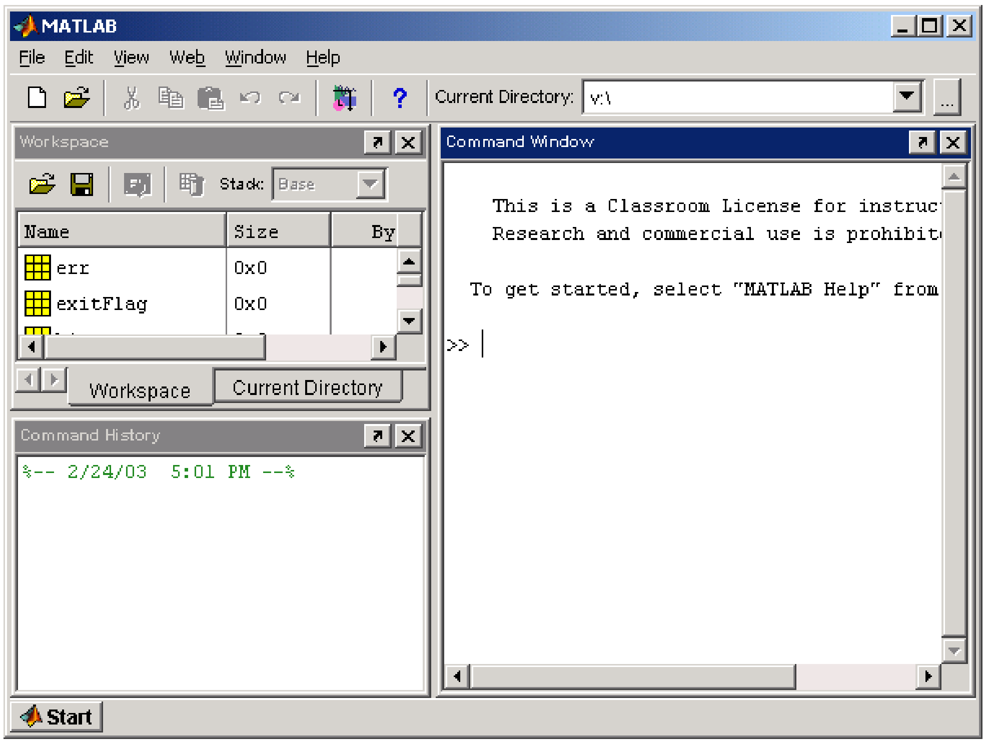
The 1997 version relates to MATLAB 4, 2002 to MATLAB 6.1 and 2007 to MATLAB 7.2. Our Labs now use MATLAB version 7. However, most of this course is not dependent on the version of MATLAB used.

Matlab has extensive built in help, either through typing `help` followed by a topic or function, or through the MATLAB menus. There is also extensive on-line help on the Internet, via the Mathworks web page, or other 3rd party tutorials.

The Uni book shop can order in a student version of MATLAB for less than $200, but you do not need this to complete the course. There is also a free program very similar to MATLAB called `octave`, but it does have some differences particularly in the user interface, and so we do not recommend it for this course, though you may wish to use it in the future.

1.2 Getting started

Invoking `matlab` produces a MATLAB window similar to Figure 1.1.



Current Directory: v:\

Name	Size	By
err	0x0	
exitFlag	0x0	

```
  This is a Classroom License for instruc
  Research and commercial use is prohibit

To get started, select "MATLAB Help" from

>> |
```

```
%-- 2/24/03  5:01 PM --%
```

Figure 1.1: Matlab window

The `>>` is the MATLAB prompt. Initially you will enter commands at this prompt but later we will see how to write and use `.m` files using a text editor.

For an overview of the help facility type `help help`. For a menu-driven graphical user interface of the help facility, type `helpwin`, or use the HELP menu. For help with a specific command type `help command_name` where `command_name` is the name of the command with which you seek help.

To exit MATLAB type `quit` or use the FILE menu.

1.3 How a program works

Consider the following piece of MATLAB code, which we might type at the MATLAB prompt.

```
balance = 1000;
rate = 0.09;
interest = rate * balance;
balance = balance + interest;
disp( 'New balance:' )
disp( balance )
```

Typing this in the MATLAB command window we get the following output:

```
New balance:
     1090
```

The statements in our program are interpreted by MATLAB as

1. assign the value 1000 to the variable `balance`.
2. assign the value 0.09 to the variable `rate`.
3. multiply the value of `rate` by the value of `balance` and assign the answer to `interest`.
4. display (in the command window) the message given in single quotes.
5. display the value of `balance`.

MATLAB processes the statements in *order* from the top down. When the program finishes the variables used will have the values

```
balance: 1090
interest:  90
rate:    0.09
```

1.4 Variables

A variable is a programming structure we define to hold a value. It is called a variable because we can change the value it holds. A variable is created by assigning a value to it. For example

```
a=98
```

Any operations that assign a value to a variable automatically create the variable if needed, or overwrites its current value if it already exists. If the right-hand side of an assignment operation refers to a non-existent variable you will get the error message

```
Undefined function or variable
```

MATLAB allows us to give a variable a value of a string, number, or *array* (a synonym for a vector or matrix). In fact, by default all variables are arrays. Scalars are just stored as 1×1 arrays.

1.4.1 Variable name

A variable name should follow these rules:

1. it may consist of only the letters `A`–`Z` and `a`–`z`, the digits `0`–`9`, and the underscore '_'.
2. it must start with a letter.
3. it must be shorter than 63 characters long (see `namelengthmax`).
4. it must not be a reserved keyword (e.g. `for`, `while`, `function`, or `if`). We can get a list of keywords by calling `iskeyword`.

If in any doubt, we can distinguish valid variable names using the function `isvarname(`*`variable`*`)`.

Examples of valid variables names: `r2d2` and `pay_day`.

Examples of invalid variable names: `pay-day`, `2a`, `name$`, or `_2a`.

It is good programming style to avoid using common functions as variables, for example `sin`, or `cos`. We also prefer to use variables that are meaningful, rather than abstract variables like `x`.

1.4.2 Case sensitivity

MATLAB is *case sensitive* so it distinguishes between upper and lower-case letters. So `BALANCE`, `Balance` and `balance` are three different variables. This is also true of function names.

1.4.3 Class/Type

Each variable has a `Class` (often called a type in other programming languages). The default type in MATLAB is an array (a matrix or a vector) of *double precision floating point* numbers, but typicallyMATLAB assigns the appropriate type to a variable when it is first defined. The type will also automatically change as required throughout a program, so we don't need to explicitly define the type of a variable, but we may need to know the type, or change it.

The command `whos` shows a list of the variables we have defined, along with their size (how big the array is). For example, given the code above, and depending on the precise version of MATLAB you're using you should end up with something like

```
Name            Size            Bytes  Class

balance         1x1                 8  double array
interest        1x1                 8  double array
rate            1x1                 8  double array
```

Each variable occupies 8 *bytes* of storage (64 bits). The variables are scalars, but MATLAB represents scalars as a 1×1 matrix, hence the values in the `Size` column, and the term `array` at the end. The term `double` refers to the fact these are double precision floating point arrays.

The most important variable classes in MATLAB :

- **double precision floating point:** numbers are the default way of representing real numbers. Each `double` uses 8 bytes or memory. MATLAB uses the IEEE Standard 754 for its floating point representation.
- **logical:** or Boolean variables represent the values TRUE and FALSE, using 1 and 0 respectively. We can define a logical variable using logical operators like `==`. In principle a logical variable needs only 1 bit, but MATLAB stores each in 1 byte.
- **char:** represents a ASCII tex character, i.e., the typical typewriter letters. An array of these forms a string (a piece of text). We can define a string using single quotes, e.g.

  ```
  the_string = 'Hello, world!';
  ```

  Quotes may be included in a string by repeating them twice.

  ```
  the_string = 'Hello, ''world''!';
  ```

There are other types in MATLAB, e.g. `single`, `int8`, `uint16`, `function_handle`, etc., but these are less commonly used. Also, one of the pleasures of programming in MATLAB is that one typically doesn't have to worry about the type of a variable as MATLAB handles these for you, unless you have a specific requirement. There are also more advanced data types such as `cell` and `struct` that are outside the scope of this course. MATLAB also allows more object oriented classes such as `<class_name>`, again outside the scope of this course. `help class` can provide more details.

In MATLAB we can often ignore a variable's class and allow MATLAB to work out the details for us, but there are some issues you need to be aware of.

- Double-precision floating point numbers try to represent a real number, but they do NOT do this to arbitrary precision. This allows numerical errors in calculations, and if one does not program carefully these can become a problem, The classic mistake is to test whether two floating points numbers are equal by writing, for instance `x == y`. This may fail because the two numbers are very slightly different. For instance, in MATLAB $\sin(\pi) \neq 0$, because MATLAB only stores an approximate value of π. We will later see how to do this correctly.
- Strings are not a "scalar" variable, but rather are represented as an array of characters. This is sometimes important when operating on them.

1.5 Script M-files

A MATLAB program saved from a text editor with the `.m` extension is called a *script file*. As an example, let's save our earlier interest program in a file with the name `calc_interest.m`.

```
balance = 1000;
rate = 0.09;
interest = rate * balance;
balance = balance + interest;
disp( 'New balance:' )
disp( balance )
```

To run the program `calc_interest` we simply enter the name

```
calc_interest
```

at the MATLAB prompt, and each command in the file will be executed in order. A script file may be listed in the command window with the command `type`, e.g.

```
type calc_interest
```

and MATLAB would output the above `.m` file.

MATLAB has a built in editor that we can use via the MATLAB menus. Go to FILE->NEW to create a new `.m` file, or FILE->OPEN to edit an existing file. The editor has many useful features, e.g. it highlights different parts of the code in different colours to help identify, e.g. comments. It puts line numbers next to the lines of code to aid in debugging, and it has built in debugging tools. Other text editors also support some or all of these features, but for the purposes of this course we will use MATLAB's built in editor.

1.6 Useful features in the MATLAB window

The MATLAB window has some useful features. On the left-hand (by default) side the window has sections allowing us to display the current Workspace, the current directory, and the command history.

1.6.1 The Directory

One of the options we can display is the current directory (sometimes called a folder), showing a list of the `.m` files we have created. We can also manage this directory, or change directories.

1.6.2 The Workspace

A fundamental concept in MATLAB is the *workspace*. If we enter the command `who` we should see a list of variables, for instance, given the previous example `who` would return

```
Your variables are:

balance   interest  rate
```

We can also see the workspace in the top left frame of the MATLAB window.

All variables you create during a session remain in the workspace until you `clear` them, either individually, or if `clear` is called by itself it clears the whole workspace.

The MATLAB window can also display a graphic of the workspace, showing a list of variables their size, and a graphic representation of what type of variable they are.

1.6.3 Command history

The history contains a list of all of the commands we type. It is convenient for us to be able to review this, but more importantly, we can repeat a command easily. The up-arrow on the keyboard allows us to scroll back through these past commands. We can filter the command list by typing a few letters at the command prompt and then using the up-arrow. MATLAB will then scroll through commands that match the letters we typed. This can save us a lot of typing.

1.7 Punctuation!

By default, MATLAB has one command per line. When you hit the enter or return key to start a new line, MATLAB interprets the current command. In a `.m` file, we usually have one command per line of the file. So a single line is like a "sentence" in English, but we don't need to put a full stop at the end.

In MATLAB, various symbols can alter this behaviour.

- **`,`** We can put more than one command on a line with a comma, e.g.
  ```
  x=1, y=x
  ```
  The commands are executed in order from left to right.
- **...** If we have a complicated formula that won't easily fit on one line, we can spread it over two lines using three full stops, e.g.,
  ```
  x = (1 + 2 + 3 + 4 + 5 ...
       + 6 + 7)
  ```
- **`%`** The percentage sign is use to denote a *comment*. Comments are text in the program that has no effect on the program itself. In MATLAB, everything on a line that appears after the % sign is ignored. Comments are very useful for making code easier to understand, e.g.,
  ```
  g = 9.8  % the gravitational constant in m/s^2
  ```
- **`;`** By default, when we enter a MATLAB command, the result of that command will appear in the command window. If we wish to suppress this behaviour we end the line with a semi-colon ";", and the command will execute silently. Omitting the semi-colons can save us typing `disp`, so we only really use `disp` for teaching purposes.

1.8 Programming style

It is extremely important for you to develop the art of writing programs which are laid out well and with all the logic described clearly. Good comments are not fun to write, and are often omitted, or done carelessly. However, good comments make a program more easily maintainable, and reusable. Failing to comment code may seem to save time, but generally costs companies a great deal more than it saves.

In programming MATLAB we expect you to

- Put a comment at the start of all .m files explaining what the file does, who wrote it and when, and some details of any inputs, outputs, or other assumptions. It should also list how it relates to any other programs. Often it is useful to provide a reference to the source of an algorithm, or a set of data.
- Variable names should be meaningful. For example `interest_rate` is preferable to `x`. Where possible, match variable names to the reference text.
- Variable names should not overlap common functions.
- Even where variable names are chosen well, it is useful to accompany a variable definition with a comment. Sometimes this can help understand details of the variable (for instance, we might have two interest rates in our program and wish to help a reader understand which is which). Another use for comments is to specify units, e.g.

  ```
  g = 9.8  % the gravitational constant in m/s^2
  ```
- Comments can be otherwise used to highlight key steps in an algorithm, or otherwise clarify code.
- Spaces can be used in expressions to make them easily readable, e.g. on either side of the equal signs as in `x = [1,2,3]`. We can also use brackets to make complex expressions easier to understand.
- Blank lines can be used to separate different parts of the program. Another convention is to use a row of % signs to separate major segments of code.
- Don't "hardwire" values. Where-ever you have a value that you use more than once in a program, you should assign that value to a variable, and use the variable. This makes maintenance much easier as you will only have to change the value in one place to update the program.

You may want to develop your own style but it is important to pay attention to readability. A good approach is to imagine another person who has to read your code, and modify it. Then apply the principle "Do unto others as you would have done to you." Do the things that you would appreciate when you are reading other peoples' code.

Perhaps a more compelling maxim comes in the form of a quote from Damien Conway (Perl Best Practices)

> Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live.

Chapter 2

Vectors and matrices

A matrix is a rectangular object (e.g. a table) consisting of rows and columns. A vector is a special type of matrix having only one row or column. Vectors are also commonly referred to as *lists* or *arrays*. We'll postpone a discussion of matrices until later. For the moment we'll concentrate on vectors, starting with *row* vectors.

2.1 Initialising row vectors: explicit lists

To get started with entering vectors into MATLAB we'll try the following. We can define a row-vector directly using square brackets.

```
x = [1 3 0 -1 5];
```

We have created a vector (or list) with five *elements*. To see how MATLAB displays a vector we can enter the command `disp(x)`, though omitting the semicolon will have the same effect.

If we enter the command `whos` we'll see that, under the heading `size`, `x` is `1x5` which means that it has 1 row and 5 columns. The function `size` will return the size of a matrix as a `1x5` array. We can also directly obtain the length of our vector using `length(x)`, which will be 5.

We can also put commas instead of spaces between the elements when defining a vector

```
a = [5,6,7]
```

which has exactly the same effect as leaving spaces, but may be substantially easier to read if we, for instance, put more complicated expressions into our array definition, e.g,

```
a = [5+1, 6-2*3, sin(2*pi)]
```

In general the definition of a vector can involve a MATLAB expression perhaps even involving other variables.

We can also define an empty array, e.g.

```
x = [ ]
```

The empty array can be useful in some circumstances, e.g., where we need to have a variable defined, but don't want to put anything in it yet. If we enter and then enter `whos` we find that the size of `x` is given as 0 by 0. This means that `x` is defined but it has no value or size.

2.2 Initialising row vectors: the colon operator :

A vector can also be generated with the *colon operator*. If we enter the following:

```
x = 1:10
```

we obtain a vector with elements that are the integers (1,2,3,4,5,6,7,8,9,10). The command

```
x = 1:0.5:4
```

produces a vector with the elements (1, 1.5, 2, 2.5, 3, 3.5, 4) each of which increases in increments of 0.5. The colons separate three values, and the *middle* value is the increment. Similarly

```
x = 10:-1:1
```

produces a vector with elements (10,9,8,7,6,5,4,3,2,1) since the increment is negative. Another example is

```
x = 1:2:6
```

In this case the elements are 1,3,5. Note that when the increment is positive but not equal to 1 the last element is not allowed to exceed the value after the second colon.

2.3 Column vectors

We can create a column vector by reusing the semi-colon (this is a different use from when we end a line with a semi-colon). We simple define a column vector by

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

which defines the 3x1 column vector

$$x = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$$

2.4 Transposing vectors

We can transpose between row and column vectors using the a single quote, or *apostrophe* `'`, e.g., when we enter

```
y = [1 4 8]'
```

we get the column vector

$$y = \begin{pmatrix} 1 \\ 4 \\ 8 \end{pmatrix}$$

with 3 rows and 1 column. Likewise,

```
y = [4; 5; 1]'
```

Results in

$$y = (4, 5, 1).$$

[Warning: actually this operation gives the conjugate transpose. Replace 1 by i in this example and inspect the output. For simple transpose use `.'` rather than a simple apostrophe.]

2.5 Concatenation

Concatenation basically means sticking one array on the end of another. We can concatenate two vectors by placing them within square brackets, e.g. if we take

```
  a = [1 2 3]
  b = [4 5]
  c = [a -b]
```

Then $c = (1, 2, 3, 4, 5)$. Or for column vectors

```
  a = [1; 2]
  b = [4; 5]
  c = [a; -b]
```

Then

$$c = \begin{pmatrix} 1 \\ 2 \\ 4 \\ 5 \end{pmatrix}$$

2.6 Subscripts

We can refer to particular elements of a vector by means of *subscripts*.

1. Set up the vector
   ```
   r = [2 4 8 16 32 64 128]
   ```
2. The command
   ```
   r(3)
   ```
   gives the third element of the vector `r` (the value is 8). The number 3 is the *subscript*.
3. The command
   ```
   r(2:4)
   ```
   will give the *second*, *third* and *fourth* elements of the vector `r`, i.e., the vector $(4, 8, 16)$.
4. The command
   ```
   r(1:2:7)
   ```
   returns the odd terms $(2, 8, 32, 128)$.
5. We can use an empty vector to *remove* elements from a vector. The command
   ```
   r([1 7 2]) = [ ]
   ```
   will remove the elements 1,7 and 2, so now `r` will look like
   $$r = (8, 16, 32, 64).$$
6. There is a special term `end` we can use to mean the last element of an array, e.g. if
   ```
   r = [2 4 8 16 32 64 128]
   ```
   Then `r(5:end)` would be the array $(32, 64, 128)$.

Warning: MATLAB subscripts start at 1 (the integer 1 means the 1st element of the array). In C, subscripts start at 0. This is a very common source of errors for people who have to write code in both.

2.7 Matrices

A matrix may be thought of as a table consisting of rows and columns. We enter a matrix just as we did for a vector, using a semi-colon to indicate the end of a row. The statement

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

results in

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

A matrix may be transposed in the same way as for a vector. An apostrophe will interchanging rows and columns, e.g., the statement `a′` results in

```
  ans =
      1    4
      2    5
      3    6
```

A matrix can also be constructed from column vectors of the same length by concatenation. The statements

```
  x = 0:30:180
  table = [x′ sin(x*pi/180)′]
```

concatenates the two column vectors `x` and `sin(x*pi/180)′` together into a `7x7` matrix

```
  table =
           0         0
     30.0000    0.5000
     60.0000    0.8860
     90.0000    1.0000
    120.0000    0.8660
    150.0000    0.5000
    180.0000    0.0000
```

Subscripts work as expected. The element in the ith row, and jth column, i.e., the (i, j)th element of matrix `A` can be accessed using `A(i,j)`. As before we can use vector subscripts to extract a portion of the matrix. For instance

```
  table([1 2 3], 2)

  ans =
           0
      0.5000
      0.8660
```

We can replace the whole possible range of an index using either `1:end`, or even more simply just `:`. For instance

```
  table([1 2 3], :)

  ans =
           0         0
     30.0000    0.5000
     60.0000    0.8660
```

2.8 MATLAB and matrices

One of the most powerful features of MATLAB is its ability to operate directly on matrices. For instance, we can multiply all of the elements of a matrix by a scalar simply using the standard multiplication operator $*$. In the above example the function `sin` acts on each element of the column vector, returning a column vector whose elements are sine of `x`. We will discuss this further in the following chapter, but some simple examples are

```
A = [1; 2; 3];
b = 3;
x = b*A;
y = b + A;
```

which will result in

$$x = \begin{pmatrix} 3 \\ 6 \\ 9 \end{pmatrix} \qquad y = \begin{pmatrix} 4 \\ 5 \\ 6 \end{pmatrix}$$

There are also special operators defined in MATLAB for performing matrix operators. A simple example is `x = [1, 2, 3].^2`, where the `.^` operator squares each element of the vector giving $x = (1, 4, 9)$. A more sophisticated example is given below.

Example: If a stone is thrown vertically upward with an initial speed u, its vertical displacement s after time t has elapsed is given by the formula

$$s = ut - \frac{1}{2}gt^2,$$

where g is the acceleration due to gravity. Air resistance has been ignored. We would like to compute the value of s over a period of about 13sec at intervals of 0.1 seconds and to plot the distance-time graph over this period. Our plan for this problem is as follows:

1. Get the data (g, u and t) into MATLAB.
2. Calculate the value of s according to the formula.
3. Plot the graph of s against t.

The resulting program is

```
% Vertical motion under the action of gravity
g = 9.8;                          % acceleration due to gravity
u = 60;                           % initial velocity (metres/sec)
t = 0 : 0.1 : 13 ;                % the time in seconds
s = u * t - g/2 * t.^2 ;          % vertical displacement in metres

plot(t, s)
title( 'Vertical motion under gravity')
xlabel( 'Time' ) , ylabel( 'Vertical displacement' )
grid
```

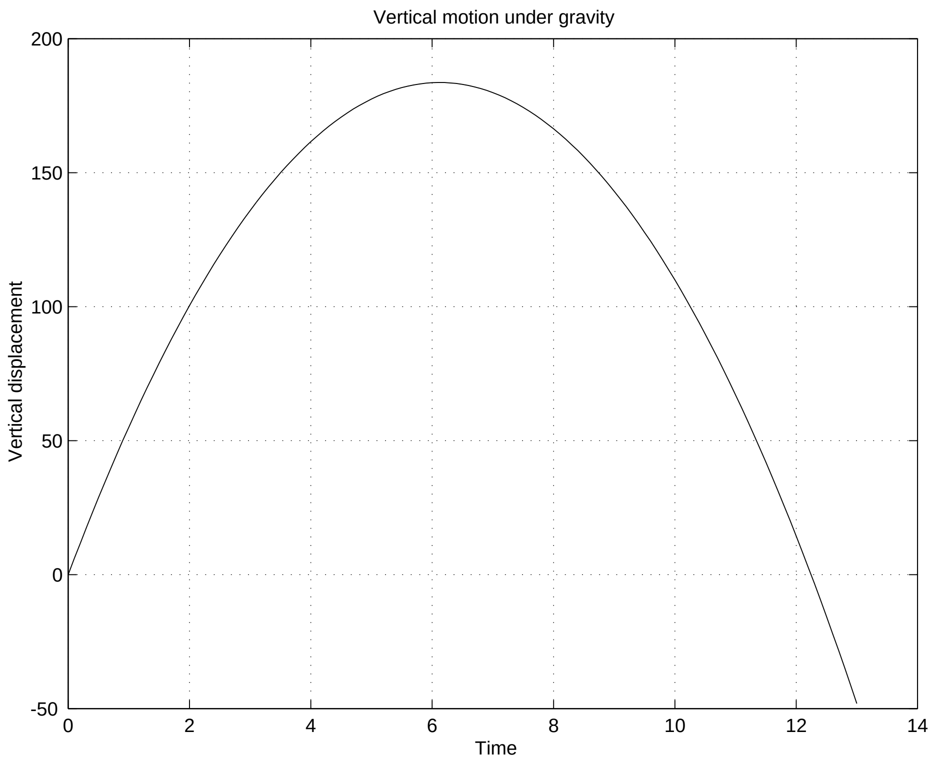


Figure 2.1: Vertical motion under gravity

The graphical output is shown in figure 2.1.

An additional constructor that is often useful when building matrices is the `meshgrid` function. It works as follows: take two vectors `x` and `y`, of lengths N and M respectively, and

```
[X, Y] = meshgrid(x, y);
```

will result in `X` and `Y` that are $N \times M$ matrices, with the rows of `X` the vector `x`, and the columns of `Y` are the vectors `y`. Remember that MATLAB variables are case sensitive so `X` is a different variable to `x`.

A simple example is the construction of a multiplication table much as we did in Excel. Use the following commands

```
x = 1:12;
y = 1:12;
[X, Y] = meshgrid(x, y);
Table = X .* Y;
```

Now the variable `Table` will contain the multiplication table.

2.9 Solving linear equations with MATLAB

One of the most common uses for matrices is in solving a set of linear equations, e.g., we have three variables x_1, x_2, and x_3 and three equations

$$\begin{aligned} 3x_1 + 2x_2 + x_3 &= 2, && (2.1)\\ x_1 + x_2 + 3x_3 &= 2, && (2.2)\\ 2x_1 - x_2 + 2x_3 &= 1. && (2.3) \end{aligned}$$

We can represent a set of such equations by

$$A\mathbf{x} = \mathbf{b},$$

where

$$A = \begin{pmatrix} 3 & 2 & 1 \\ 1 & 1 & 3 \\ 2 & -1 & 2 \end{pmatrix}, \quad \mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \quad \text{and} \quad \mathbf{b} = \begin{pmatrix} 2 \\ 2 \\ 1 \end{pmatrix}$$

In MATLAB we can solve a set of equations such as this simply using

```
x = A \ b
```

When A is invertible, this is equivalent to $\mathbf{x} = A^{-1}\mathbf{b}$ computed using Gaussian elimination. We can obtain the inverse of A directly using

```
inv(A)
```

Note that when A is not invertible, or non-square MATLAB's behaviour is more complex. Also, it is possible to have unexpected results if a matrix is *ill conditioned*, e.g.,

```
A = [2 eps -eps; eps 1 1; -eps 1 1];
b = [2; eps + 2; -eps + 2];
x = A \ b
```

MATLAB will print a warning in this case saying

```
Warning: Matrix is close to singular or badly scaled.
         Results may be inaccurate. RCOND = 2.465190e-32.
```

We can obtain more information using `help mldivide`.

2.10 Strings

Text can be stored in variables in MATLAB, and the result is usually called a *string*.The standard way to create and assign a string to a variable is use single quotes, e.g.,

```
the_string = ’Hello, world!’;
```

Quotes may be included in a string by repeating them twice.

```
the_string = ’Hello, ’’world’’!’;
```

In the latter case, `whos` will tell us that `the_string` is a `1x15` array of characters taking 30 bytes.

Actually, a MATLAB string is an array of numbers, each storing the “Unicode” number for a character in the string. Unicode consists of a repertoire of about 100,000 characters from most world languages. The most commonly used encodings (in English) are ASCII characters. We can write out a table of the printable ASCII characters,

```
ascii = [char(32:79); char(80:127)]
ascii =
 !"#$%&’()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNO
PQRSTUVWXYZ[\]ˆ_‘abcdefghijklmnopqrstuvwxyz{|}˜ˆ
```

For more information about ASCII see, for example, `http://en.wikipedia.org/wiki/ASCII`. MATLAB stores the a numeric code associated with each character, so operations such as `the_string+1` will have unexpected results (it shifts us down the alphabet by one). We can convert between a character array, and an array of double precision numbers using the conversion functions `char()` and `double()`.

A string is an array, and hence they may be concatenated just as other arrays, e.g.,

```
the_string = [’Hello, ’ ’world!’];
```

A typical string is just a row vector of characters, but we can form matrices of characters. There can be problems, however, with such matrices. For instance, it is often appealing to interpret them as a series of lines of text. In contrast to typical text, these are held in an array, and so each row must be the same length. Also operations on these arrays (e.g. transpose) often have unexpected results, so care must be taken. What is often needed is an actual array of strings, which can be formed in MATLAB using a *cell array*, but such arrays fall outside the scope of this course.

2.11 Multi-dimensional arrays

MATLAB allows one to construct multi-dimensional arrays. Often this is simplest using standard constructors of matrices, such as `ones`, `zeros`, and `rand`, which allow for more than 2 input parameters with a resulting multi-dimensional matrix, e.g.,

```
A = ones(3,2,4);
```

will return a $3 \times 2 \times 4$ array. We can access its elements using, for instance, `A(i,j,k)`, and `size(A)` will return the vector `[3 2 4]`. Displaying `A` with the `disp` function will return

```
(:,:,1) =
     1     1
     1     1
     1     1
(:,:,2) =
     1     1
     1     1
     1     1
(:,:,3) =
     1     1
     1     1
     1     1
(:,:,4) =
     1     1
     1     1
     1     1
```

where each group specifies a 3×2 subarray, or which there are four.

Chapter 3

MATLAB as a big calculator

One of the key features of MATLAB is the ability to do complicated calculations. In some ways it resembles a great big calculator, but its capabilities, and even the rules for how calculations work are rather different from your standard calculator.

3.1 Numbers

3.1.1 Writing numbers

Numbers can be represented in MATLAB in the usual decimal form, e.g.

```
1.2345 , -123 , .00001
```

A number may also be represented in *scientific notation*, e.g. $1.2345 \times 10^6 = 1,234,500$ may be represented in MATLAB as

```
1.2345e6
```

3.1.2 Numerical errors

As noted above, numbers are stored as double precision floating point variables, but this means there will be small errors in some numbers. For instance, irrational numbers such as as $1/3$, π, or $\sqrt{2}$ are not possible to represent exactly, but you may find errors even in numbers that are exact, e.g. 1.10101010101010101 will be stored as approximately 1.10101010101009944. Note that when you use the `disp()` function, you only see the value output to a fixed precision.

The function/variable `eps` tells us something about the spacing between floating point numbers. Used as a variable in the current version of MATLAB, `eps = 2.220446049250313e-16`, which gives us difference between `1` and the next largest number that can be represented in MATLAB, but used as a function it can tells us a great deal more. Use `help eps` to find out more information.

3.1.3 Special cases

There are two special cases of number in MATLAB : `NaN` and `Inf`, standing for *Not a Number* and *Infinity*, respectively. MATLAB returns these when certain arithmetic rules (such as never divide by zero) are ignored. For instance

```
 1/0 = Inf
-1/0 = -Inf
 0/0 = NaN
```

We should check if a number falls into these cases using the functions `isinf` and `isnan` because, by definition $NaN \neq NaN$.

3.1.4 Complex numbers

It is very easy to handle complex numbers in MATLAB. The special values of `i` and `j` stand for $\sqrt{-1}$. We must be careful, however, because often programmers reassign these values. They can be set back to $\sqrt{-1}$ using, e.g., `clear i`. We can define a complex number by

```
z = 2 + 3*i
```

The imaginary part of a complex number may also be entered without the asterisk, e.g. `3i`. All the arithmetic operators (and most functions) work with complex number. For instance, `+` adds the real and imaginary components, respectively, while `*` performs standard complex multiplication. The functions `real(z)`, `imag(z)`, `conj(z)` and `abs(z)` all have the obvious meanings. There are also function `isreal` to test if a number is real, or if its imaginary part is non-zero.

Note that imaginary numbers require storage of two double precision floating points numbers, and hence require 16 bytes of storage. Also complex arithmetic involves more computation than real arithmetic, so it is best not to use complex numbers unless needed.

3.2 Operators, expressions and statements

Let us start with some definitions. An *expression* is a formula consisting of variables, numbers, operators and function names. An expression is evaluated when you enter it at the MATLAB prompt, e.g., to evaluate 2π we enter

```
2 * pi
```

MATLAB's response is

```
ans =
      6.2832
```

A *statement* does something. For instance, it might write something in the command window, plot a figure, or assign a value to a variable, e.g.,

```
s = u * t - g / 2 * t.^ 2;
```

This is an example of an *assignment statement*. The value of the *expression* on the right is *assigned* to the variable on the left.

Statements and expressions use operators as short hand for standard mathematical operations. For instance `=` is used to assign a value to a variable. MATLAB has a large number of operators, you can see a list by typing `help ops`. We will discuss some here.

Operation	Algebraic form	MATLAB
Addition	$a + b$	`a+b`
Subtraction	$a - b$	`a-b`
Multiplication	$a \times b$	`a*b`
Right division	a/b	`a/b`
Exponentiation	a^b	`a^b`

Table 3.1: Arithmetic operations between two scalars

Operation	MATLAB	result
comparison	`x == y`	returns TRUE if x and y are equal, and FALSE otherwise
comparison	`x > y`	returns TRUE if $x > y$, and FALSE otherwise
comparison	`x >= y`	returns TRUE if $x \geq y$, and FALSE otherwise
comparison	`x < y`	returns TRUE if $x < y$, and FALSE otherwise
comparison	`x <= y`	returns TRUE if $x \leq y$, and FALSE otherwise
comparison	`x ~= y`	returns TRUE if x is not equal to y, and FALSE otherwise
logical AND	`x & y`	returns TRUE if x AND y are true, and FALSE otherwise
logical OR	`x | y`	returns TRUE if x OR y are true, and FALSE otherwise
logical NOT	`~x`	returns TRUE if x is FALSE, and FALSE otherwise

Table 3.2: Common logical operators.

3.2.1 Arithmetic operators

The evaluation of expressions is often achieved by means of *arithmetic operators* which are similar to those we are familiar with in algebra. The arithmetic operations on two *scalar* constants or variables are shown in Table 3.1.

3.2.2 Logical operators

It is common that we wish to assign a logical, or Boolean value to a variable, or otherwise use it in an expression. The most used logical operators are shown in Table 3.2. There is also an `xor` function where this is needed. There are a number of other logical operators (for instance bitwise operators) that we will not consider here.

A common operation is comparing two numbers to see if they are equal. This is a common source of confusion as there are two similar operators `=` and `==`. The former is an *assignment* operator — it assigns the value of the expression on its right-hand side to the variable on the left-hand side. The latter (the double equals sign) is the comparison operator, which tests whether two values are equal. In preference to the comparison `x == y`, we often use an operation such as `abs(x-y) < epsilon`, where `epsilon` is a small number. This allows for some errors in the floating point representation of the numbers.

Finally, MATLAB also includes a number of *test functions* that return TRUE if their inputs satisfy certain conditions. A more complete table of such functions appears in Section 6.5, but note that we need to use such a function (in particular the function `strcmp`) if we wish to compare strings. The `==` comparison operator only works for numbers, not strings, because a string is really an array not a single value.

3.2.3 Array operators

MATLAB has a large set of array (or matrix) operators. The standard arithmetic operators are translated into their matrix equivalent, e.g., the matrix multiplication AB would be written `A * B` in MATLAB, and addition, subtraction, and logical operators all work on matrices by adding, subtracting, or comparing their individual elements, respectively. Obviously, these operations require matrices of the same size, or an error is returned.

An exception to the same size rule is that MATLAB also combines scalars and matrices in an intuitive fashion. For instance take A to be a matrix, and b a scalar (a 1×1 matrix in MATLAB), and then

- `A + b` means add b to each element of A.
- `A * b` means multiply b by each element of A.
- `A^b` means take the matrix A times itself b times, i.e., $A * A * \cdots * A$. Obviously this can only be done for a square matrix.

MATLAB also introduces a number of operators specific to matrices and vectors. These are based on standard arithmetic operators, but preceded by a dot, e.g.,

- `C = A .* B` means form a matrix C whose (i, j) element c_{ij} is given by $c_{ij} = a_{ij} * b_{ij}$.
- `C = A ./ B` means form a matrix C whose (i, j) element c_{ij} is given by $c_{ij} = a_{ij}/b_{ij}$.
- `C = A.^B` means form a matrix C whose (i, j) element c_{ij} is given by $c_{ij} = a_{ij}^{b_{ij}}$.

The "dot" operators, e.g., `a.*b` are called *element-by-element* operations because they are performed element by element. For `.*` and `./` to work, we need `A` and `B` to be the same size.

Consider the following simple example. Given,

```
a = [2 4 8];
b = [3 2 2];
```

then operator the *array product* denoted by `a .* b` is

```
[a(1)*b(1)   a(2)*b(2)  a(3)*b(3)] = [6 8 16]
```

In a similar way `a./b` gives element-by-element division. The exponential is

```
[2 3 4] .^ [4 3 1] = [16 27 4]
```

we find that the ith element of the first vector is raised to the power of the ith element of the second vector. Note that if we replace one of the matrices with a scalar b, then MATLAB effectively creates a new matrix B whose elements $b_{ij} = b$, e.g.

```
2 .^ [4 3 2 1] = [16 8 4 2]
```

Other matrix operators we have already seen include the vector construction operators `:`, `[` and `]`, the transpose operator `'`, and `;` (for constructing column vectors). There is also a left-division operator, `\`, mentioned earlier.

3.3 Precedence of operators

MATLAB has strict rules about which operations are performed first when several operations are combined in an expression. These are called the precedence rules and are shown in Table 3.3. When operators in an expression have the same precedence the operations are carried out left to right. So `a/b*c` is evaluated as `(a/b)*c` and not as `a/(b*c)`.

Precedence	Operator
1	Parentheses (brackets)
2	Transpose and Exponentiation
3	Unary plus and minus, logical negation
4	Multiplication and division
5	Addition and subtraction
6	colon `:`
7	AND `&`
8	OR `|`

Table 3.3: Precedence of standard operators. Operators of equal precedence are evaluated from left to right.

The precedence rules are similar to those you are familiar with from standard algebra (parentheses, exponentiation, multiplication and division, and then addition). However, in MATLAB , these rules apply to a range of operators, for instance, array multiplication has precedence 5, just as scalar multiplication; element-wise exponentiation `.^` has the same precedence as the scalar, and matrix exponentiation operator `^`. Note that if part-way through evaluating an expression we would end up multiplying two incorrectly sized matrices when following the precedence rules,MATLAB will return an error. No effort is made to interpret the sense of an expression.

In many programming languages there are two types of $\pm$ symbols. The standard operator combines (adds or subtracts) two operands, e.g., $a - b$. The *unary* operator acts on one operand, e.g., $-a$. The two types have different precedence. The other important difference in MATLAB's precedence operators is that MATLAB has some unusual operators related to matrices. For instance, consider the colon operator. The colon operator has a lower precedence than addition as the following shows. In

```
x = 1+1:5
```

the addition is carried out first, and then a vector with elements $x = 2, \ldots, 5$ is initialised. Compare this with

```
x = 1+(1:5)
```

which results in $x = 2, \ldots, 6$, because the brackets reorder the operations.

3.3.1 Parentheses and programming style

Parentheses, or brackets are always first in the precedence order. Hence, we can use brackets to group operators into the order we desire. This is not just for use in cases where the order needs to be changed. Using brackets can often be useful even where the order is already correct, because it can make an expression much easier to read, and debug. As such, using brackets sensibly to make complex expressions more readable is a part of good coding practice.

3.4 Vectorisation of formulae

Array operations can be used to evaluate a formula repeatedly for a large amount of data. Let's consider the following formula for calculating compound interest.

Example: An amount of money A is invested over a period of n years with an annual interest rate r. After n years we have an amount $A(1+r)^n$. Suppose we want to calculate the final balances for investments of $\$750, \$1000, \$3000, \$5000$ and $\$11,999$ over ten years, with an annual interest rate of 9%. The following sequence of commands does the calculation by using an array operations on a vector which contains the initial investments:

```
format bank
A = [750 1000 3000 5000 11999];
r = 0.09;
n = 10;
B = A * (1+r)^n;
disp ([A' B'])
```

The output is

```
    750.00       1775.52
   1000.00       2367.36
   3000.00       7102.09
   5000.00      11836.82
  11999.00      28406.00
```

Notes:

1. `format bank` provides a two-decimal-place fixed format for currency.
2. In the statement `B=A*(1+r)^n`, the expression `(1+r)^n` is evaluated first because exponentiation has a higher precedence than multiplication. This is a scalar operation.
3. After that the array operation between the vector `A` and the scalar `(1+r)^n` is formed.
4. A `*` may be used instead of a `.*` because the array multiplication is between a scalar and a non-scalar (although `.*` would not be wrong).
5. A table is displayed, with columns given by the transposes of `A` and `B`.

The process of writing out a formula such that we can calculate it for a vector of inputs is called *vectorisation* of a formula. Vectorization of MATLAB code is very important. MATLAB has been carefully optimized for vector and matrix operations, and will do these very quickly (almost as quickly as purpose written C-code). Other types of operations are not as fast, as we shall see later.

Example: Often we want to compute much more complicated formulae, for a range of inputs. For instance, let us compute the following formulae for calculating the value of a European call option (using the Black-Scholes model). A European call option on a share gives us the right to buy a share of the stock at price K after T years. The Black-Scholes formula gives its predicted value at

$$C = S\Phi(d_1) - Ke^{-rT}\Phi(d_2),$$

where $\Phi(\cdot)$ is the standard normal cumulative distribution function.

$$\begin{aligned} d_1 &= \frac{\ln(S/K) + (r + \sigma^2/2)T}{\sigma\sqrt{T}} \\ d_2 &= \frac{\ln(S/K) + (r - \sigma^2/2)T}{\sigma\sqrt{T}} \\ S &= \text{current price of the stock} \\ T &= \text{time till option is exercised} \\ K &= \text{exercise price} \\ r &= \text{interest rate} \\ \sigma &= \text{volatility} \end{aligned}$$

Obviously this is rather complicated if we wanted to compute by hand, but we can calculate the value of the option for a variety of exercise prices K using the simple MATLAB code

```
% set the problem parameters
S = 1, r = 0.07, sigma = 1, T = 3;
K = 0:0.1:2;
d1 = (log(S./K) + (r-sigmaˆ2)/2)/(sigma*sqrt(T));
d2 = (log(S./K) + (r-sigmaˆ2)/2)/(sigma*sqrt(T));
C = S * normal_cdf(d1) -exp(-r*T) * K .* normal_cdf(d2)
plot(K, C);
```

where we will describe how to define the function `normal_cdf(x)` in Section 8.2.3. Notice that we put `.*` and `./` operators in the places where we could be operating on vectors. The result is a graph (shown in Figure 3.1) showing us the behaviour of the option values, from which we can assess what we would be willing to pay for such an option.

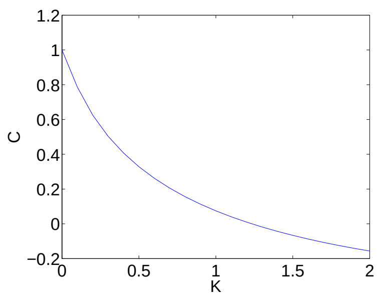


Figure 3.1: Option prices

Chapter 4

Input/Output

Often we need to either obtain input to our program from the user, or from a file, or output information to the user or a file. We have already seen two approaches to sending output to the MATLAB window.

1. With the `disp` function, e.g. `disp(x)`.

2. By entering a variable name, assignment or expression on the command line, without a semi-colon;

In this chapter we will provide more details of these approaches, but also we will introduce other approaches to I/O (Input/Output).

4.1 `disp`

The general form of `disp` for a numeric variable is

`disp(` *variable* `)`

To display a message and a numeric value on the same line we use the following technique:

```
x=2;
disp( ['The answer is ', num2str(x)] )
```

The output should be

```
The answer is 2
```

We want to display a *character string* and a number but the elements of a MATLAB vector must be either all numbers or all strings. To overcome this we convert the number `x` to its *string representation* using the function `num2str`, and the square brackets (`[ ]`) concatenate the two strings to form one string which is displayed.

4.2 The `format` statement

MATLAB has the following two basic rules:

1. It always attempts to display integers (whole number) exactly. If the integer is too large it is displayed in scientific notation with five significant digits, e.g 1 234 567 890 is displayed as `1.2346e+09` (i.e 1.2346×10^9).
2. Numbers with decimal parts are displayed with four significant decimal digits. If the value of x is in the range $0.001 < x \leq 1000$ it is displayed in fixed point form, otherwise scientific (floating point) notation is used e.g. 1000.1 is displayed as `1.0001e+003`.

Note that numbers are not displayed to the precision they are computed to.

This is MATLAB's default format. It is possible to change to format for output. To output numbers displaying more significant digits use `format long`, or `format bank` can be used to output currency to two decimal places. There are many other options, see `help` format for details. However, complete control over the output format requires us to use the `fprintf` function.

4.3 `fprintf`

The `fprintf` statement is much more flexible than `disp`. Consider the example

```
balance = 12345;
rate = 0.09;
interest = balance * rate;
balance = balance + interest;
fprintf( ...
  'Interest rate: %6.3f New balance: %8.2f\n', ...
  rate, balance )
```

If we run this our output should look like

```
Interest rate:  0.090 New balance: 13456.05
```

The `fprintf` function has allowed us to control the format of the output precisely. More generally, we might call `fprintf` using

`fprintf` ('*format string*', *list of variables*)

The *format string* controls how the output appears. It may contain a simple text string, in which case this is printed out. It may also contain one of a series of codes (mixed into the text), and these special codes are replaced (in the output) with either a special character, or the value of one of the variables in the list of input arguments. Table 4.1 gives a list of common codes.

Note the following:

1. In the case of `%e` and `%f` the *field width* and number of decimal places or significant figures may be specified immediately after the `%`. For instance we might write

Code	Action
`%f`	write a numerical variable in decimal notation
`%e`	write a numerical variable in scientific notation
`%g`	write a numerical variable (MATLAB's choice)
`%s`	write a string variable
`%%`	the % sign
`\n`	new line
`\t`	horizontal tab
`\b`	backspace
`\\`	\

Table 4.1: Special codes used by `fprintf`

- `%8f` which means the width of the output will be 8 characters (and extra space will be padded at the left to fill in gaps).
- `%.3f` which means write the number out showing three decimal places. By default `%f` means `%.6f`.
- `%6.1f` which means write the number with width 6, and one decimal place.
- `%12.3e` means scientific notation over 12 columns altogether (including the decimal point, a possible minus sign and four for the exponent) with 3 digits in the mantissa after the decimal point.

Note that numbers are rounded off when outputting with limited precision.

2. The `%g` specifier is mixed and leaves it up to MATLAB to decide exactly what format to use.
3. The `%s` specifier also allows you to specify the width of the output string (padded with spaces if the string is not wide enough), e.g. `%6s`.
4. The *list of variables* contains values that we wish to output.
5. Note that fprintf can take a vector as an input variable, and will recycle the format string until the the elements of the input are all used (they are used columnwise).
6. We often end a format string with `\n` in order to start the next output on a new line.

Table 4.2 shows a series of examples of `fprintf` functions illustrating some of these options.

There are a number of other *conversion characters* (characters following a % sign), and *escape codes* (characters following a backslash `\`), but we will not consider them in detail here. It is noteworthy that the syntax of `fprintf` in MATLAB is similar to that used in `C`, and the two duplicate many escape codes and conversion characters.

4.3.1 Output to a file with `fprintf`

Output may be sent to a file with `fprintf`. To do so we need to *open* the file for writing with the `fopen` function. This will create a *file identifier*, or FID variable. For example:

function call	output
`fprintf('Hello, world!\n')`	`Hello, world!`
`fprintf('pi = %f\n', pi)`	`pi = 3.141593`
`fprintf('pi = %.12f\n', pi)`	`pi = 3.141592653590`
`fprintf('pi = %10.6f\n', pi)`	`pi =   3.141593`
`fprintf('pi = %e\n', 100*pi)`	`pi = 3.141593e+02`
`fprintf('pi = %g\n', 100*pi)`	`pi = 314.159`
`fprintf('x = %.0f\n', 1:3)`	`x = 1`
	`x = 2`
	`x = 3`
`fprintf('x = %3.0f, y = %.3f\n', -1, sqrt(2))`	`x =  -1, y = 1.414`
`fprintf('message = %s\n', 'hello')`	`message = hello`

Table 4.2: Examples of `fprintf`. The first is just printing a string, the second group show different number formats, and the third shows the recycling of the format string when the input variable is a vector.

```
   fid = fopen('exp.txt','w');
```

The first input argument to `fopen` is the name of the file we wish to write to. The second input argument `'w'` specifies that it is to be opened for *writing*. The `fopen` function has lots of other options. Use `help fopen` to find out more.

The `fprintf` command takes an extra input argument, which is the FID variable, in this case `fid`, which tells `fprintf` to output the results to the file. For example,

```
   x = 0:.1:1;
   y = [x; exp(x)];
   fid = fopen('exp.txt','w');
   fprintf(fid,'%6.2f  %12.8f\n',y);
   fclose(fid);
```

After writing the output to the file we need to *close* the file with the `fclose` function. Note that we can give the FID variable any (allowed) variable name, and have more than one open file at a time. We can even have an array of FID variables.

Note that `fopen` can also be used to open a file for reading (or several other options). When reading a file, we might use the `fscanf` function but there is often a better approach.

4.3.2 `sprintf`

Sometimes it is desirable to create a string, including variables. We can use the `sprintf` function to do this. The function is called just as `fprintf`, but it has one output argument, which is the result of the combination of formatted string output. For instance

```
   the_string = sprintf('pi = %f\n', pi)
```

Will result in `the_string` holding the value `'pi = 3.141593'`. This type of construction can often be useful for creating title plots, for instance, consider the following code that creates a plot, with a title that depends on the value of `f`.

```
  f = 3;
  x = 1:0.01:10;
  y = sin(f*x);
  plot(x, y);
  title_str = sprintf('f = %f\n', f);
  title(title_str);
```

4.3.3 The `input` command

Let's rewrite the script file `calc_interest.m` so that it looks like

```
  balance = input( 'Enter bank balance: ' );
  rate = input( 'Enter interest rate: ' );
  interest = rate * balance;
  balance = balance + interest;
  format bank
  disp( 'New Balance:' )
  disp( balance )
```

If we now enter the script file name, which I've called `calc_interest1.m` at the prompt we are interrogated by the computer for the initial values of the balance and interest rate. The command window will contain the following lines:

```
  >> calc_interest1
  Enter bank balance: 1000
  Enter interest rate: 0.15
  New Balance:
         1150.00
```

The input statement provides a more flexible way of getting data into a program. It allows us to enter data *while a script is running*.

The general form of the `input` statement is

variable `= input( '`*prompt*`' )`

1. *prompt* is a message which prompts the user for the values(s) to be entered. It must be enclosed in apostrophes (single quotes).

2. A semi-colon at the end of the `input` statement will prevent the value entered from being immediately echoed on the screen.

3. Vectors and matrices may also be entered with `input`, as long as you remember to enclose the elements in square brackets.

4. Strings may be input if they are enclosed in quotes, e.g.,

   ```
   name = input( 'Enter your name: ' );
   fprintf('Hello %s!\n', name);
   ```

   we would see a prompt and enter our name as follows:

   ```
   Enter your name: 'John'
   Hello John!
   ```

4.4 Advanced I/O

MATLAB has many commands for more advanced Input/Output. A common requirement is to read data from a file. Simple text files are the easiest to read. MATLAB supports this in a variety of ways the most general being the `fscanf` function, which is similar to the function of the same name in `C`.

However, we can use an simple approach when a data file is really a table, i.e., it consists of a series of lines, each of which is in a fixed format. More specifically, row i of the file looks something like

$$data_{i1}, data_{i2}, data_{i3}, ..., data_{iN}$$

where for a given column j, each of the terms $data_{ij}$ is the same type of data (number or string). In this case we can use the `textread` function. We call this function as follows:

```
[data1, data2, ..., dataN] = textread( file, format_str);
```

The file is the file from which we wish to read, and the format string specifies whether the data is a string, number, of some other type of data. Each of the output variables would contain one column of the data from the file, i.e., for a file with m rows `data1` = $(data_{11}, \ldots, data_{m1})'$.

The function `textread` has many optional arguments to, for instance, change the delimiter between data in each row, or to allow for header lines, or comments in the file. For instance, assume we have a file `addresses.dat` as follows:

```
% format:
% last name, first name, address, age
Potter,Harry,Hogworts,17
The Grey,Gandalf,Middle Earth,1023
Christmas,Father,North Pole, 2008
```

We could read this file using the commands

```
file = 'addresses.dat';
format_str = '%s %s %s %f';
[last_name, first_name, address, age] = ...
   textread(file, format_str, ...
            'commentstyle', 'matlab', ...
            'delimiter', ',');
```

The option `commentstyle` allows us to specify that strings beginning with % are comments (just as they are in a MATLAB program), and the `delimiter` option specifies that the file is in CSV format. The data itself would be read into the variables with the corresponding names. For instance, the variable `age` would be a column vector containing `[17; 1023; 2008]`.

Many files can be read using `textread` and various related functions. However, these are all *text* files that basically consist of a series of characters. A *binary* file, consists of a series of 1's and 0's in a format that is (i) often optimized to reduce space requirements, and (ii) depends on the type of data being held. Common examples include Excel's file format, along with common media files such as music and video files. MATLAB has many functions for reading, writing and displaying such data. We will not examine these at length except to note some of the possibilities:

- **Audio:** MATLAB can read several file formats, but the easiest and most common are `.wav` files, which can be read/written and played using the `wavread`, `wavwrite` and `wavplay` functions.
- **Images:** Image in many formats (e.g., `.png`, `.jpg`, `.tiff`, etc.) can be read into MATLAB using `imread`, written using `imwrite` and displayed using the `image` functions.
- **Excel:** Excel files can be read and written using `xlsread` and `xlswrite`.
- **Video:** `.avi` video files can be read and written using `aviread` and `movie2avi`, respectively.

The `help` command can provide further information about all of these functions, and additional I/O functions can be found using `help iofun`.

Chapter 5

Program flow control

We have seen earlier that MATLAB statements are usually executed in the order we type them, either in the command window, or in a `.m` script file. However, sometimes we want the order or execution of statements to change, perhaps depending on the value of a variable. We might want certain statements to only run under some circumstances, or to run multiple times. Program *flow control* refers to the programming constructions used to achieve this type of reordering.

MATLAB is a high-level language, which means that it is intended to look a little like a natural human language — in particular English — combined with standard mathematical formulas. Until now we have mainly concentrated on the mathematical component, but we will now examine some of the "English-like" syntax used in MATLAB that are used to control program flow, in particular the *keywords* `if`, `else`, `end`, `for` and `while` . More information can be found using the `help lang` command.

5.1 Making decisions with `if`

A standard requirement is *conditional execution*, i.e., we only want to execute some piece of code *if* a condition is true. The condition could depend on previous calculations, and so we don't know the result when we are writing the program. We only learn whether the condition holds when the program executes. We typically achieve this type of conditional execution using the `if` statement.

For example, the MATLAB function `rand` generates a random number in the range $0 - 1$. What would we expect if we were to enter the following commands?

```
r = rand;
if r > 0.5
  disp('greater indeed')
end
```

In the second statement we have used `if` and a relational operator `>`. MATLAB will only display the message `greater indeed` if `r` is greater than 0.5.

5.1.1 The `if` statement

The simplest form of the `if` statement is

```
if  condition
    statements
end
```

We note the following points:

1. *condition* is usually a *logical expression*, i.e. an expression containing logical operators such as are found in Table 3.2. Typically it might involve one or more relational (comparison) operators, combined with logical operations like AND and OR.

2. If *condition* is true, *statement* is executed but if *condition* is false, nothing happens.

3. The condition should typically be a scalar. If it is a vector or matrix, it is considered true only if all elements of the matrix are true. A single zero element in a vector or matrix renders it false.

Simple examples of condition are

MATLAB condition	meaning
`b^2<4*a*c`	$b^2 < 4ac$
`x>=0`	$x \geq 0$
`a~=0`	$a \neq 0$
`b^2==4*a*c`	$b^2 = 4ac$
`x >= 0 & x < 5`	$0 \leq x < 5$

5.1.2 The `if-else` statement

Consider the following example:

```
x = 2;
if x < 0
  disp( 'negative' )
else
  disp( 'non-negative' )
end
```

This tests to see if x is negative. If it is it will return the message `negative`, if it is positive or zero it will return the message `non-negative`.

Most banks offer differential interests rates. Suppose that the rate is 9% if the amount of your savings is less than $5000 but 12% otherwise. Given a starting balance, we can calculate our new balance using the following program:

```
% test whether balance is less than 5000
if balance < 5000
  % if it is set the interest rate to be 0.09
```

```
    rate = 0.09
  else
    % balance is greater than or equal to 5000
    % set the interest rate to be 0.12
    rate = 0.12
  end

  % calculate and display the new balance
  new_balance = balance + rate * balance;
  disp('New balance after interest paid is:')
  format bank
  disp( new_balance )
```

The basic form of `if-else` for use in a program file is

```
if  condition
      statements1
else
      statements2
end
```

Note the following:

1. Both *statements1* and *statements2* represent one or more statements.
2. If the condition is true, *statements1* are executed, but if the condition is false, *statements2* are executed. This is how we force MATLAB to choose between two alternatives, and in programming it is often call *branching*.
3. The `else` part is optional. The `if` statement is a special case of the `if-else` statement.

5.1.3 `elseif`

Suppose our bank now offers 9% interest on balances of less than $5000, 12% on balances of $5000 or more but less than $10000 and 15% for balances over $10000. We can calculate the new balance after one year by using the following:

```
  % test whether balance is less than 5000
  if balance < 5000
    % if it is set the interest rate to be 0.09
    rate = 0.09
  % test whether balance is less than 10000
  elseif balance < 10000
    % if it is set the interest rate to be 0.12
    rate = 0.12
  else % balance is greater than or equal to 10000
    % set the interest rate to be 0.15
    rate = 0.15
```

```
end

% calculate and display the new balance
new_balance = balance + rate * balance;
disp('New balance after interest paid is:')
format bank
disp( new_balance )
```

In general the `elseif` clause is used as follows:

```
if  condition1
        statements1
elseif  condition2
        statements2
elseif  condition3
        statements3
...
else
        statementsN
end
```

We sometimes call this an `elseif` ladder. It works as follows:

1. *condition1* is tested. If it is true *statements1* are executed; MATLAB then moves to the next statement after `end`.

2. If *condition1* is false, MATLAB checks *condition2*. If it is true, *statements2* are executed, followed by the statements after `end`.

3. In this way, all the conditions are tested until a true condition is found. As soon as a true condition is found no further `elseif` statements are examined and MATLAB jumps off the ladder.

4. If none of the conditions are true, *statementsN* after `else` are executed.

5. There can be any number of `elseif`'s but at most one `else`.

6. `elseif` must be written as one word.

7. `if` and `if-else` statements are special cases of the `if-elseif-else` statement.

5.1.4 Logical operators

Logical expressions can be constructed using the *three logical operators* `&` (and), `|` (or), `~` (not), that we examined earlier. For example the quadratic equation

$$ax^2 + bx + c = 0,$$

has equal roots, given by $-b/(2a)$, provided that $b^2 - 4ac = 0$ and $a \neq 0$. This can be translated to the following MATLAB statements:

```
if (b^2 - 4*a*c == 0) & ( a ~= 0)
   x = -b / (2*a)
end
```

5.1.5 Nested `if` statements

It is possible, and not uncommon for `if` statements to be *nested*. This means we have one `if` statement inside another, for example

```
if (isreal(x))
  if (x < 0)
    disp('x is real, and negative');
  elseif (x > 0)
    disp('x is real, and positive');
  else
    disp('x is zero');
  end
else
  disp('x is complex');
end
```

The first `if` results in printing out the message `x is complex` unless `x` is real, in which case the second `if` statement is used to discriminate between three cases.

We could have implemented the above nested `if` statements using a single `if-ifelse-else` statement, e.g.,

```
if (isreal(x) & x < 0)
  disp('x is real, and negative');
elseif (isreal(x) & x > 0)
  disp('x is real, and positive');
elseif (isreal(x))
  disp('x is zero');
else
  disp('x is complex');
end
```

Complex conditionals can often be expressed in multiple ways, and the best choice often depends simply on making code as readable as possible. However, an important factor in design of conditional statements is minimizing the number of operations (for instance comparisons) that we need to perform. If we test for common cases first, then we can often eliminate many subsequent comparisons. Likewise, we can sometimes use careful construction of nested `if` statements to reduce the number of comparisons that we need to perform on average, or in the worst case. By doing so we can improve the performance of our code.

For example, imagine that we need to classify a series of inputs into three categories: A, B and C, based on the output of three logical functions `isA`, `isB` and `isC`, and output the results. Imagine also that in our dataset, 1000 cases are in group A, 500 in group B, and only 1 in group C.

We might do this classification in any order, two examples being

```
if (isA(x))
  disp('x is in group A');
elseif (isB(x))
  disp('x is in group B');
elseif (isC(x))
  disp('x is in group C');
else
  disp('Error');
end
```

```
if (isC(x))
  disp('x is in group C');
elseif (isB(x))
  disp('x is in group B');
elseif (isA(x))
  disp('x is in group A');
else
  disp('Error');
end
```

In the example on the left-hand side, we would perform the first comparison for every data point, the second for those in groups B and C, and the third for only those in group C, because the other groups would already have been eliminated by the first two comparisons.

In the example on the right-hand side, we again perform the first comparison for all data points, but the second is performed for all group B and A points, and the third for all of group A. So the total number of comparisons performed for the two approaches is

comparison	left approach	right approach
first	1501	1501
second	501	1500
third	1	1000
total	2002	4001

We can immediately see that the left-hand approach performs nearly half as many comparisons as the right-hand approach, and so would run roughly twice as fast.

5.1.6 The `switch` statement

Where we wish to compare a complex series of conditions, it is often simpler to use a `switch` statement, rather than a long series of `elseif` statements. A switch statement looks like

```
switch  switch_expr
   case  case_expr,
      statements1
   case  case_expr,
      statements2
   case  case_expr,
      statements3
      ...
   otherwise,
      statementsN
end
```

The `switch` statement looks for the first *case_expr* which matches the *switch_expr*, and executes the statements following this case. If none of the case statements match, the statements following the `otherwise` are executed. For example, consider the following code:

```
grade = input('Enter your grade (F,P,C,D,HD):');
switch grade
   case 'F'
       fprintf('Your mark was < 50\n');
   case 'P'
       fprintf('Your mark was between 50 and 65\n');
   case 'C'
       fprintf('Your mark was between 65 and 75\n');
   case 'D'
       fprintf('Your mark was between 65 and 75\n');
   case 'HD'
       fprintf('Your mark was > 85\n');
   otherwise,
       fprintf('Error: %s was not a valid grade\n', grade);
end
```

The response of the program to an input like `'D'` would be `Your mark was between 65 and 75`, and it outputs an error if an invalid grade is input. Note also that the comparison works between strings (we have to input a string in quotes), whereas the comparison operator, e.g. `grade == 'HD'` would treat `'HD'` as a vector and would therefore fail if `grade` was a vector with only one element. You can test multiple conditions in a switch/case statement by putting them in curly brackets, e.g. `{'D','HD'}`.

5.2 Repetition with `for`

Computers are stupid (we have to be quite careful about what we tell them to do) but very fast. Furthermore, many numerical techniques are built around the idea of performing a simple operation many times. As a result, a common requirement in programming is the ability to repeat code more than once (often many times). Obviously we could type the same commands more than once, but this is annoying, and more importantly it is harder to debug (you have to make sure each copy of the commands is exactly the same). There is an easier way. Repeating the same code is called *iteration*.

The most common type of iteration (in MATLAB) is *count controlled* iteration where we create a counter, and iterate over certain values of the counter. In MATLAB we do this using the `for` statement. The following code

```
for i = 1:3
  disp(i)
end
```

creates the *counter* `i` and then iterates the `disp` statement for each value of the counter in the specifier `1:3`. That is, we execute the loop three times, once for each value $i = 1, 2, 3$. The output would be

```
     1
```

```
    2
    3
```

This type of high-level iteration construct was first used in `FORTRAN` in 1956, though it was called a `DO` loop in `FORTRAN`. For many years, `FORTRAN` was the most important language for scientific computation applications, primarily because of these types of high-level constructs which it pioneered.

5.2.1 An example: square roots via Newton's Method

The square root x of any positive number a may be found using only the arithmetic operations of addition, subtraction and division via *Newton's method*. This is an iterative process that refines an initial guess. The following *pseudo-code* describes Newton's method for calculating the square root of a.

```
1. Initialise $x$ to $a/2$
2. Repeat the following steps a number of times ($6$ say)
     • Replace $x$ by $(x + a/x)/2$
3. Stop
```

The MATLAB program to do this (for the case $a = 2$) follows. Note that we print out the value of `x` at each iteration.

```
disp( 'Square roots via Newtons method' )
a = 2;
format long;
format compact;
x = a/2;
for i = 1:6
   x = (x+a/x)/2
end
disp( 'Matlab''s value for sqrt(2) is: ')
disp( sqrt(2) )
```

The output (after making the format long and compact) is

```
x =
   1.500000000000000
x =
   1.416666666666667
x =
   1.414215686274510
x =
   1.414213562374690
x =
     1.414213562373095
  x =
     1.414213562373095
  Matlab’s value for sqrt(2) is:
     1.414213562373095
```

The value of x converges to a limit, which is $\sqrt{a}$. Note that this is identical to the value returned by the MATLAB function `sqrt`.

5.2.2 The basic `for` statement

The simple form of the `for` loop is

```
for index = j:k
    statements
end
```

The loop will be performed exactly once for each value of `index` from $j, j+1, j+2, \ldots, k$, in order. On completion, the variable `index` contains the last value used.

The term *index* can be any valid variable, e.g., `i`, `a_variable`, or `counter`, but cannot take the form `-x`, or any other invalid variable name.

5.2.3 More general `for` statements

More generally, we can perform a `for` loop over any vector, i.e.,

```
for index = vector
    statements
end
```

In this case, the loop is run once for index taking each element of the vector as a value (in order through the vector). Typically the vector is constructed for the `for` loop, e.g.,

```
for index = 1:10:51
  disp( ['index = ' num2str(index)] )
end
```

which would construct the vector `[1 11 21 31 41 51]`, and then apply the loop, outputting

```
index = 1
index = 11
index = 21
index = 31
index = 41
index = 51
```

We can explicitly construct the vector before calling the loop, e.g.,

```
index_values = 10.^[1:3];
for index = index_values
  disp( ['index = ' num2str(index)] )
end
```

which would run the loop once each with the values of `index` being 10, 100 and 1000 and output

```
index = 10
index = 100
index = 1000
```

If the vector is empty, *statements* are not executed and control passes to the statement following the `end` statement. For instance,

```
for i = 5:0
  disp(i)
end
```

will not do anything, because the vector `[5:0]` is empty. The correct form of this would be to use `[5:-1:0]`. It is sometimes worth explicitly testing for an empty index vector using the `isempty` function before entering a loop, and output an error if the vector would be empty.

Example: We can combine `if` and `for` statements. Consider a bottle of wine at temperature $25°C$, which is placed in a refrigerator where the ambient temperature F is $10°C$. We want to find out how the temperature of the wine changes over a period of time. To do this we first need one 'fact', namely **Newton's Law of Cooling** which states:

> *The rate of change of temperature T within a body placed in an environment whose ambient temperature is F is proportional to the difference between the temperature of the body and the ambient temperature, $T - F$.*

Mathematically this translates to (remembering that *rate-of-change* with respect to time is simply the derivative of the function)

$$\frac{\mathrm{d}T}{\mathrm{d}t} = -K(T - F). \tag{5.1}$$

We note

1. K is a constant of proportionality which depends upon the insulating properties of the material, in our case the glass bottle, and also the thermal properties of the wine.

2. The constant K is positive, since if $T > F$, so that the wine is hotter than the ambient temperature, we expect it to cool and hence the rate of change of the temperature must be negative.

A standard way of approaching problems of this type is to break the time period up into a number of small steps, each of length dt. If T_i is the temperature at the *beginning* of step i, we can use a simple **Euler method** to get from T_i to T_{i+1}.

This method relies on approximating the differential equation

$$\frac{\mathrm{d}y}{\mathrm{d}t} = f(y;t),$$

by the approximation

$$\frac{y_{i+1} - y_i}{\Delta t} = f(y_i;t_i).$$

For our cooling problem we have

$$T_{i+1} = T_i - K\Delta t(T_i - F).$$

The following MATLAB script implements this scheme:

```
% Variable initialisations.
K = 0.05;
ambient_temperature = 10;
temperature = 25;
start_time = 0;
end_time = 100;

% Request input values.
delta_t = input('Input the computational interval, delta_t:' );
output_interval = input('Please input the output interval: ' );
if rem(output_interval,delta_t) ~= 0
  error('The output interval must be a multiple of delta_t!')
end

time = start_time;
disp( 'Time Temperature' )
disp( [time temperature] )

for time = start_time+delta_t : delta_t : end_time
  temperature = temperature - K * delta_t * ...
    (temperature-ambient_temperature);
  if rem( time, output_interval ) == 0
    disp( [time temperature] )
  end
end
```

The function `rem` computes a remainder. It is used to check that `output_interval` is an integer multiple of `delta_t`, and to display the results every `output_interval` minutes (when `time` is an integer multiple of `output_interval` the remainder will be zero).

5.2.4 Nested `for` statements

As with `if` statements, it is common in programming to *nest* `for` loops. This means placing one `for` loop inside another. For instance, when we work with an $N \times M$ array of data A, we might use a construction such as

```
N = 12;
M = 12;
for i=1:N
  for j=1:M
```

```
      A(i,j) = i * j;
    end
  end
```

The inner loop is executed once for each outer loop, so we will eventually iterate through all allowed values of i and j.

5.2.5 Avoiding `for` loops by vectorising

Suppose we want to evaluate

$$\sum_{n=1}^{1000000} \frac{1}{n^2}.$$

There are two way to do this. Firstly using the `for` loop

```
  partial_sum = 0;
  for term = 1:1000000
    partial_sum = partial_sum + 1 / term^2;
  end
  disp( partial_sum )
```

We can *vectorise* this command by using the `sum` command

```
  terms = 1:1000000;
  partial_sum = sum(1 ./ terms.^2)
```

In general, MATLAB is highly optimized for vectorized calculation. A vector/matrix computation such as the second approach will be much faster than the first. On my computer, the first approach took 1.03 seconds, and the second approach tool only 0.048 seconds. The second approach was more than 20 times faster!

This type of speedup is most clear in large, nested loops. For instance, if we wished to compute a large multiplication table, we could use the nested loop approach immediately above, or we could use the matrix approach described in Section 2.8. The latter approach will be much faster.

This a key feature/problem with MATLAB. The fact that it can do matrix operations quickly is of great value. However, it is not always easy to vectorise a complex program. Also, there is a cost in memory – the vectorized approach requires us to store a large vector in memory, whereas the nested loop approach can sometimes avoid this.

5.2.6 Preallocating arrays

An additional issue, when we use a loop, is preallocation of arrays. MATLAB's ability to allocate memory for arrays on the fly makes it convenient to simply create an array as we go through a loop, e.g., consider the following code to compute the Fibonacci sequence:

```
  x(1) = 1;
  x(2) = 1;
  N = 10000;
```

```
for i=3:N
  x(i) = x(i-1) + x(i-2);
end
```

Here the vector x is extended in each iteration of the loop. However, when we know it size, it is often more efficient to preallocate the array. For instance:

```
N = 10000;
x = zeros(N,1);
x(1) = 1;
x(2) = 1;
for i=3:N
  x(i) = x(i-1) + x(i-2);
end
```

This creates the array before we use it in the loop, and avoids memory management overhead inside the loop. In my version of MATLAB , the second approach is roughly ten times as fast. Note that the above code could also be vectorized, given even better improvements in speed, but there are some times when vectorization is not an option.

5.2.7 Non-deterministic repetition

The key feature of a for loop is that the loop repetition is deterministic in the sense that it is predetermined at the start of the loop. The number of iterations may depend on earlier defined variables. For example, we may set a variable N which is then used in creating the for loop, e.g.,

```
for i = 1:N
  disp(i)
end
```

but care must be taken with a for loop to avoid tampering with the index variable inside the loop. If we explicitly change the value of the counter variable inside the loop the behaviour of the loop becomes unspecified, and we won't know what will happen in advance. So once the loop is started we (apparently) can't stop it early. Certainly we shouldn't try to stop it by changing the index.

However, there are two statements that allow us the varying the standard looping behaviour. The first allows early breakout from the loop: the break statement will drop us out of a loop early. It is usually used in conjunction with a conditional statement to cause early breakout from a loop in special circumstances, e.g.,

```
for i=1:1000000
  x = some_complex_function(x, i);
  if (isnan(x))
    break;
  end
end
```

This loop repeatedly changes the value of x based on its previous value, and the value of the counter variable i. However, if there is a problem and the function returns NaN, then we break out of the loop prematurely in order to avoid many needless repetitions of the loop.

The `break` statement should be used with considerable caution as it can reduce readability of code significantly.

The other related statement is the `continue` statement, which passes the loop onto its next value, skipping any remaining statements, e.g.

```
for i=1:6
  if (mod(i,2) == 0)
    continue;
  end
  disp(i);
end
```

would output

```
1
3
5
```

Even terms are omitted from the output, even though we loop over all terms, because the `continue` statement is executed in these cases, and hence subsequent statements (e.g. the `disp` statement) are avoided for even values of `i`. Once again, `continue` statements should be used with care because of readability concerns. Frequent use of `break` and `continue` can result in *spaghetti* code.

And there is a better approach, using non-deterministic repetition via a `while` loop.

5.3 Non-deterministic repetition with `while`

Suppose we want to use a loop but we don't know how long it will run *before we start* the loop. It is common that we want to run a numerical algorithm until it has *converged* where we won't know how long this will take until we run the algorithm. We call this a *condition controlled* loop. The number of executions of the loop will be controlled some condition calculated within the loop, rather than by a simple counter defined as part of the loop.

A simple example of this type of loop is the following game. MATLAB 'thinks' of a number between 1 and 10 and we have to guess it. If our guess is too high or too low then the script must say so, and then give us another go. If our guess is correct then the script should congratulate us.

Here's the pseudo-code for the problem

```
1. Generate a random number.
2. Prompt the user for a guess.
3. While the guess is wrong:
     • If the guess is too low
         – Tell the user it is too low
     • Otherwise
```

    - Tell the user it is too high

  - Ask user for a new guess.

4. Congratulations.

5. Stop.

Here is a script to carry out our program

```
matlabs_number = floor( 10 * rand +1);
guess = input( 'Your guess please: ' );

while (guess ~= matlabs_number)
  if guess > matlabs_number
    disp( 'Too high' )
  else
    disp( 'Too low' )
  end
  guess = input( 'Your next guess please: ' );
end

disp('At last you have guessed it! My value was:')
disp( matlabs_number )
```

5.3.1 The `while` statement

In general the `while` statement looks like this:

```
while  condition
   statements
end
```

The `while` statement repeats *statements* WHILE its *condition* remains true. The condition is tested each time BEFORE *statements* are executed. Recall that a *vector* condition is considered to be true only if *all* its elements are non-zero.

We can replicate a `for` loop using a `while` loop by explicitly constructing the counter variable, e.g., if we wanted to construct the following

```
for counter = 1:M:N
   statements
end
```

we would use

```
counter = 1;
while counter <= N
   statements
  counter = counter + M;
end
```

In this example we explicitly create the `counter` variable, and increment it by the required amount M at each loop. The `for` loop construction is more concise, and so often preferred, but the `while` loop form allows more flexibility in combining both count controlled, and condition controlled looping.

5.3.2 Infinite loops

One of the dangers of a condition controlled loop is that the condition will remain true forever. In this case the loop will continue indefinitely. We call this an *infinite loop*, and it will result in a program that doesn't terminate naturally. For instance, the following program will execute indefinitely,

```
while (1)
  disp('Hello, world!');
end
```

The program will repeatedly output "Hello, world!" because the condition `1` is always true (remember that in MATLAB 1 stands for the logical TRUE). In MATLAB we can stop a program by exiting MATLAB , or by typing `Ctrl-C`, which sends an interrupt signal to MATLAB. When MATLAB receives such a signal it should stop what it is doing. It will stop normal execution of a program, or even the infinite loop above.

Occasionally an infinite loop can be useful. Sometimes we want a program to perform some action until the program is halted (by some outside interrupt). This type of programming falls outside the scope of this course.

5.4 Programming style

Whenever we have a conditional, or loop, it is wise to indent the code inside the loops (much as we have done above). This greatly enhances the readability of the code by showing which parts will be executed (conditionally, or iteratively). Where there are multiply nest loops or conditionals, then use multiple levels of indentation. MATLAB's editor will do this for you, but other editors may not do it automatically (for instance when you are coding `C` there is no built in editor).

Sometimes, if the code inside a conditional or loop is long (say longer than one page roughly) then we might also add a comment to the `end` statement, to help indicate which conditional or loop it is associated with. This can make debugging some code easier. For instance, we might write

```
if (A > B)
    statement1
    statement2
      ...
    statementN
end % if (A > B)
```

The comment `% if (A > B)` indicates which conditional the `end` statement is finishing.

On another point, I prefer to place conditionals inside brackets (for `if` and `while` statements). This is not generally necessary, but I find that it separates the logic of the condition from the syntactical components and makes the result more readable.

5.5 Other MATLAB statements

MATLAB has a number of additional programming statements for flow control, mirroring many modern programming languages. For instance

- `try` and `catch`
- `error`
- `assert`
- `return`

These will not be covered in this course, but it is perhaps useful to know they exist. More information can be found using `help lang` or `help` *command*.

Chapter 6

Commonly used functions and variables

MATLAB has a large number of useful constants and functions ranging from mathematical functions to those for manipulating strings. Here we will present a quick summary of a number of the most commonly used. Much more detail on a function can be found by typing `help`, followed by the function's name. If you are trying to find a function, but don't know its name the `lookfor` command can often find it by searching for MATLAB commands that either match the search string (given as an input), or whose first line of description matches the search string.

6.1 Constants

MATLAB has a number of pre-defined constants.

Constant name	Value
`pi`	π
`i`	$\sqrt{-1}$
`j`	$\sqrt{-1}$
`eps`	the distance from 1.0 to the next larger double precision number
`realmax`	largest positive floating point number
`realmin`	smallest positive floating point number
`false`	0 (logical)
`true`	1 (logical)

Note that "constants" are really just variables (in MATLAB) and so can be easily redefined. Also note that irrational numbers can only be approximately represented, so constants such as π are only accurate to machine precision. Some standard constants such as e are not defined, but are easily calculated, for instance $e = \exp(1)$. Some of the above constants are actually functions, e.g., `eps(x)` gives the the positive distance from $|x|$ to the next larger in magnitude floating point number. There are also equivalent function to `realmax` and `realmin` for integers.

6.2 Elementary Mathematical Functions

Here we present some of MATLAB's elementary mathematical functions. Note that if the argument of a function is a vector (or array), the function is applied element by element to the values of the vector, e.g.

```
  sqrt( [1 4 9 16] )
```

returns

```
  1.0000    2.0000   3.0000   4.0000
```

Function list

Function	Effect
`abs(x)`	absolute value of `x`.
`acos(x)`	arc cosine (inverse cosine) of `x` (returns a value between 0 and π).
`acosh(x)`	inverse hyperbolic cosine of `x`, i.e. $\ln(x+\sqrt{x^2-1})$.
`asin(x)`	arc sine (inverse sine) of `x` (returns a value between $-\pi/2$ and $\pi/2$).
`asinh(x)`	inverse hyperbolic sine of `x`.
`atan(x)`	arc tangent of `x` (returns a value between $-\pi/2$ and $\pi/2$).
`atanh(x)`	inverse hyperbolic tangent of `x`.
`ceil(x)`	the smallest integer which is greater than or equal to `x`
`conj(x)`	complex conjugate of `x`.
`cos(x)`	cosine of `x`.
`cosh(x)`	hyperbolic cosine of `x`.
`exp(x)`	value of the exponential function e^x.
`factorial(n)`	$n!$.
`floor(x)`	the largest integer less than or equal to `x`
`imag(x)`	returns the *imaginary* part of `x`.
`log(x)`	natural logarithm of `x`.
`log2(x)`	base 2 logarithm of `x`.
`log10(x)`	base 10 logarithm of `x`.
`mod(x,k)`	`x` mod `k`.
`nchoosek(n,k)`	the number of combinations $\binom{n}{k} = n!/k!(n-k)!$.
`rand`	random number in the interval $[0, 1)$.
`real(x)`	the *real* part of `x`
`rem(x,y)`	remainder when `x` is divided by `y` i.e.`rem(19,5)` returns 4.
`round(x)`	rounds to the nearest integer
`sign(x)`	the value $-1, 0$ or 1 depending upon whether `x` is -ve, zero or +ve.
`sin(x)`	sine of `x`.
`sinh(x)`	hyperbolic sine of `x`.
`sqrt(x)`	square root of `x`. This also handles negative or complex values.
`tan(x)`	tangent of `x`.
`tanh(x)`	hyperbolic tangent of `x`.

Note for trigonometric functions that angles are usually represented in radians, not degrees. There are versions of some trigonometric functions that act in degrees, e.g. `cosd`, however, it is generally better to convert to radians by multiplying by $2\pi/360$.

See `help elfun` and `help specfun` for more mathematical functions, and `help polyfun` for functions related to polynomials.

6.3 Simple Vector/Matrix functions

There are also a number of simple functions designed to be useful specifically with vectors or matrices:

Function	Effect
`det(A)`	the determinant of `A`.
`diag(A)`	get the diagonal entries of a matrix A.
`diag(x)`	construct a diagonal matrix with `x` along its main diagonal.
`diff(x)`	returns a vector containing differences of the input, i.e., $x_{i+1} - x_i$.
`eig(A)`	eigenvalues and vectors of `A`.
`eye(N)`	construct the `N x N` identity matrix.
`find(x)`	find the indices where the logical vector `x=TRUE`
`isempty(x)`	returns TRUE if the matrix/vector is empty.
`length(x)`	number of elements of vector `x`.
`max(x)`	maximum element contained in the vector `x`.
`mean(x)`	mean value of the elements in vector `x`.
`min(x)`	minimum value contained in the vector `x`.
`norm(A)`	the matrix norm of `A`.
`ones(n,m)`	construct an `n x m` matrix of 1s.
`prod(x)`	product of the elements of vector `x`.
`rand(n,m)`	construct an `n x m` matrix of `rand`.
`repmat(A,n,m)`	creates a new array with $n \times m$ blocks formed from matrix A.
`size(A)`	the number of rows and columns of a matrix `A`.
`std(x)`	the standard deviation of the elements of `x`.
`sort(x)`	sorts elements of vector `x` into ascending order.
`sum(x)`	sum of the elements of vector `x`.
`trace(A)`	the trace of `A`.
`zeros(n,m)`	construct an `n x m` matrix of 0s.

Many of the functions above have been described in terms of their vector version (e.g. `sum`, `mean`, `max`) but have a matrix version. Some like `diag` do something completely different with a matrix or a vector, whereas other just generalize the same operation. See the help for these functions for more details.

Additional functions can be found using the above help commands in addition to `help matfun` and `help datafun`. MATLAB also has a series of functions to deal with sparse matrices: more detail can be seen with `help sparfun`.

6.4 Set functions

In MATLAB, we can choose to ignore order in a vector, and treat the vector like a set. There are special functions for operating on sets. A list of common set functions: (assuming `a` and `b` are vectors representing sets) follows.

Function	Effect
`intersect(a,b)`	returns intersections of sets `a` and `b`.
`ismember(a,b)`	returns 0-1 array of the same size as `a` which indicates which values of `a` are members of `b`.
`setdiff(a,b)`	returns elements of `a` that are not in `b`.
`setxor(a,b)`	returns elements of `a` and `b` that are not in the intersection.
`union(a,b)`	returns union of sets `a` and `b`.
`unique(a)`	returns a list of the unique elements of `a`.

6.5 Test functions

MATLAB has a number of functions designed to test certain conditions. The are commonly used in conditionals.

Function	Effect
`exist(name)`	test if a variable, function or file `name` exists.
`isempty(x)`	returns TRUE if the matrix/vector `x` is empty.
`isnan(x)`	returns true if `x=NaN`
`isinf(x)`	returns true if `x=Inf` or `x=-Inf`
`isfinite(x)`	returns true for any case except the above three
`isreal(x)`	returns true if `x` is real (not complex)
`strcmp(s1,s2)`	returns true if strings `s1` and `s2` are identical
`isa(x, ’class_name’)`	tests if variable `x` is of class `’class_name’`
`isnumeric(x)`	tests if variable `x` is a numeric array
`islogical(x)`	tests if variable `x` is a logical array
`ischar(x)`	tests if variable `x` is a character array (a string)
`ismember(a, s)`	tests if variable `a` is a member of set `s`

Typically these functions can operate on an array input, and return a logical array as output.

6.6 String functions

MATLAB has a number of functions specifically for dealing with strings (of characters). A list of common string functions: (assuming `s`, `s1` and `s2` are strings) follows.

Function	Effect
`char(x)`	convert an array of integers `x` into a string of characters.
`double(s)`	convert a string `s` to an array of integers.
`findstr(s1,s2)`	find the shorter of the two strings in the longer.
`lower(s)`	convert `s` to lower case.
`strcat(s1,s2)`	concatenate two strings `s1` and `s2`.
`strcmp(s1,s2)`	returns true if strings `s1` and `s2` are identical.
`strtrim(s)`	remove insignificant whitespace from `s`.
`sprintf(...)`	similar to `fprintf` but returns a string.
`upper(s)`	convert `s` to upper case.

The command `help strfun` will list more possibilities, while `help strings` reveals basic information about strings.

6.7 Dates and times

It is often useful to be able to work with times and dates, and MATLAB has a number of functions for doing so. Some simple time and date related functions are:

Function	Effect
`clock`	returns a six element date vector of the form `[year month day hour minute seconds]`.
`datenum(d)`	converts a date vector `d` into MATLAB's numerical representation of dates.
`datevec(x)`	converts an internal date `x` into a date vector.
`datestr(x)`	converts an internal date `x` into a human readable string.
`tic`	sets a "stopwatch" going.
`toc`	outputs the value of the stopwatch (in seconds).

Commands such as `tic` and `toc` are useful for comparing computation times of alternative algorithms. Commands such as `datenum` and `datestr` have many alternative input and output formats, and can be used quite flexibly to work with dates. When we wish to plot, for instance a timeseries, with dates along one axis, we use the `datetick` command in conjunction with the above.

6.8 Utility functions

We often need to interact with the file system/workspace of the computer we are operating on, and MATLAB has functions for doing so. Simple file system/workspace related functions

Function	Effect
`clear`	remove variables from the workspace.
`cd('d')`	change working directory to `d`.
`dir('d')`	list the files in directory `d`.
`load filename`	load variables from a `.mat` file.
`path('p')`	change the search path MATLAB uses to look for `.m` files.
`pause(n)`	pause a program for `n` seconds.
`pwd`	display the current working directory.
`save filename`	save all the workspace's variables to a `.mat` file.
`what('d')`	list MATLAB specific files in a directory.
`who`	lists variables in the current workspace.
`whos`	lists variables in the current workspace in long form.

6.9 More information

One of the strengths of MATLAB is its many useful functions, we have listed only a fraction here. Use the appropriate `help` and `lookfor` commands to find more, as well as MATLAB's online documentation. The goal of this chapter has been to give you an idea of what is possible using MATLAB's functions, rather than a complete list, or the details. The brief descriptions provided here omit many important details, and you should examine the function in more detail before using it (again using the `help` command, or the online documentation).

In addition, MATLAB has many "Toolboxes", which can be purchased (or sometimes downloaded) to add features to MATLAB, primarily by adding functions. Typing the command `ver` will provide information on what version of MATLAB you have, and what toolboxes have been installed. Additional help about these toolboxes is also available.

Finally, MATLAB also has many functions for displaying data, but we shall examine these in more detail in the following section.

Chapter 7

Graphics

One of the most powerful features of MATLAB is its ability to visualize data using plots. The basic approach is to use one of a set of functions to create a plot. Once created, features (such as axis labels) can be added to the plot either by function calls, or by interacting with the GUI of the plot window. We start this chapter by considering a basic 2D plot of a set of data.

7.1 Basic two-dimensional plots

The most common command to draw a single graph is `plot`, In its simplest form the command takes a single vector argument as in `plot(y)`. In this case the elements of `y` are plotted against their indices, e.g.

```
  y=rand(1,20);
  plot(y);
```

plots 20 random numbers against the integers 1–20 joining successive points with straight lines as in figure 7.1(a).

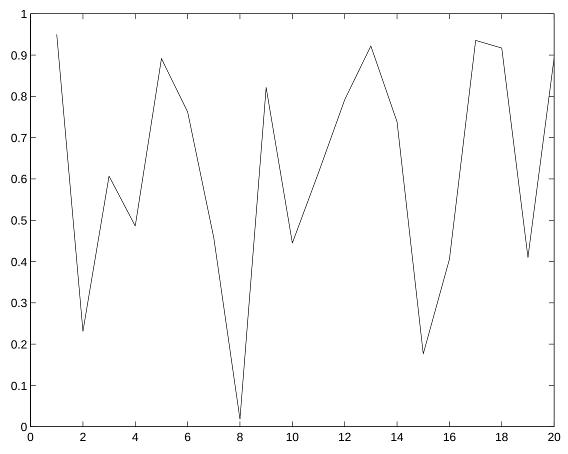
Axes are automatically drawn, and the range of these is scaled to include the minimum and maximum data points, but “nice” values of the extremes are chosen so that the plot will (hopefully) look good.

The most common form of `plot` is `plot(x,y)` where `x` and `y` are vectors of the same length. The plot function uses the first argument as the x locations of data points, and the second as the y locations, and (by default) draws lines between the series of points. For example see Figure 7.1(b), which shows the following example:

```
  x = 0 : pi/40 : 4*pi;
  y = sin(x);
  plot(x, y);
```

7.2 Decorating the figure

The figures above are not satisfactory. For instance, they have no axis labels, the text is small, and the graphs are rather bland. In this section we discuss how to modify and improve our graphs.


(a) `plot(y)`

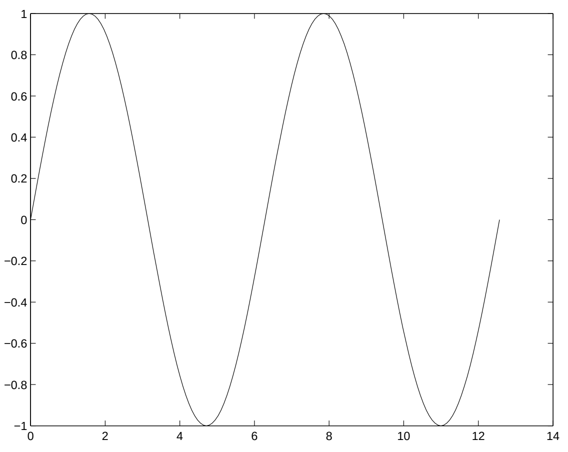

(b) `plot(x,y)`

Figure 7.1: Examples of the `plot` command.

7.2.1 The GUI

When a plot window is created (in recent versions of MATLAB), it has a number of menus, and icons at the top. For instance, see Figure 7.2, which shows a screenshot of a plot window. These allow many features you would expect, for instance, the ability to save the figure into a file (many different formats are supported), or print the figure. The GUI allows us to add labels, legends, change colours and so on. Individual curves can be selected and their properties (such as colour, line thickness, or line type) can be changed. The plot can be changed to a bar plot, a stair plot, or a filled area plot. More importantly, the GUI allows us to view the plot interactively. We can use the magnifying glass icons to zoom in or out of the plot. We can also rotate the plot, though that is only really interesting for 3D plots.

The GUI allows us to create careful, well designed plots suitable for inclusion in any high-quality report or paper. The one problem with the GUI is that it requires manual intervention. It is common for us to want to produce a set of plots using just our program, for instance, if we are analysing a hundred data files, we would not want to manually “fix” each of the resulting plots. So we also need a programatic interface to these features. We discuss a large part of this below.

7.2.2 Labels

The following functions add various labels to graphs. In each case, the input argument `s` is a string, and `x` and `y` are (x, y) co-ordinates on the plot.

Function	Effect
`title(s)`	adds the title `s` above the plot.
`xlabel(s)`	adds the test `s` as a label for the x-axis.
`ylabel(s)`	adds the test `s` as a label for the y-axis.
`text(x, y, s)`	adds the text `s` at co-ordinates (x, y).
`gtext(s)`	puts text `s` where you want it. It puts a cross-hair on the graph window and waits for a mouse button or keyboard key to be pressed.

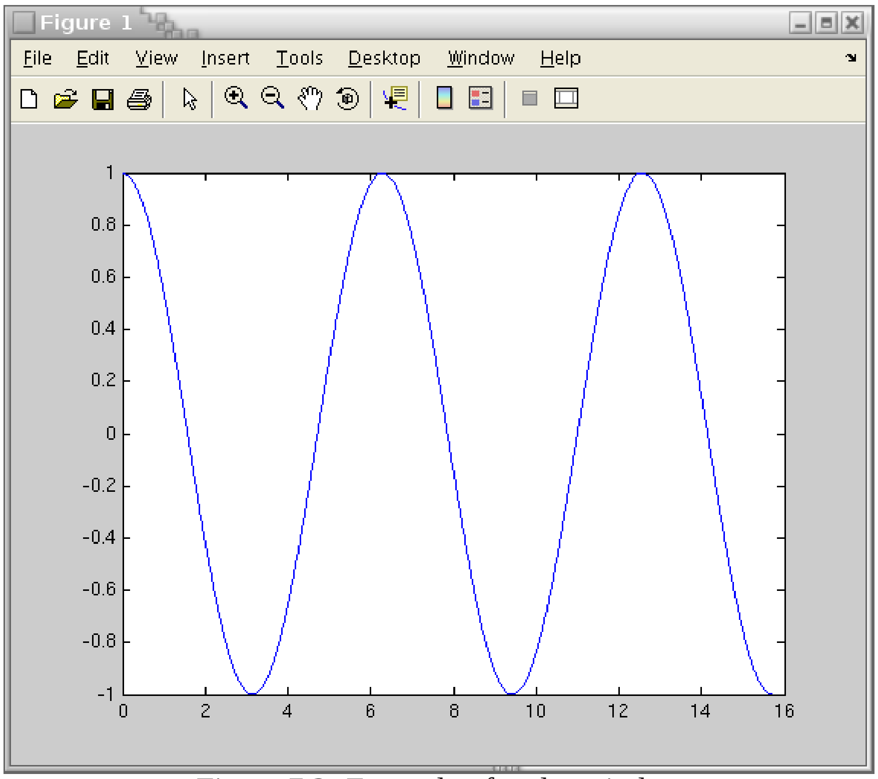


Figure 7.2: Example of a plot window.

The size of the "font" used in labels can be controlled by setting options. For example `text`, `title`, `xlabel`, `ylabel`, and `legend` allow optional arguments of the form `'fontsize'` followed by an integer. The color can be changed using the `'color'` option just as for a plot command (see below). Labels can also include a limited subset of LaTeX commands for mathematical symbols. LaTeX can be used to create greek symbols, superscripts and subscripts and a small set of other mathematical constructions useful in some labels. For instance, the following commands, produce the graph shown in Figure 7.3.

```
x = 0 : pi/40 : 4*pi;
y = sin(x);
plot(x, y);
title( 'Titles appear above the graph', 'fontsize', 20)
xlabel( 'instead of writing phi we write \phi' )
ylabel( 'ylabels are presented vertically' )
text(9.3, 0.2, 'a sin curve')
```

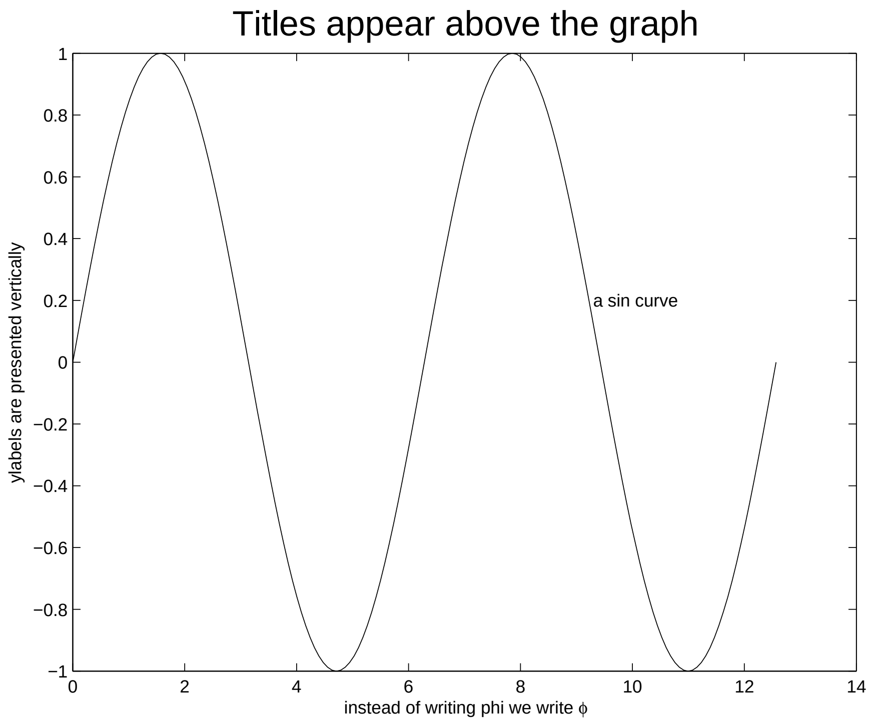


Figure 7.3: Examples of various labelling command.

7.2.3 Other graph features

There are many other ways we can control the general appearance of our plot, for instance:

Function	Effect
`axis([xmin xmax ymin ymax])`	changes the x and y range of the axes
`axis auto`	returns axes to autoscaling
`axis tight`	makes the axes scaling fit the data tightly
`axis ij`	puts the axis in “matrix” mode where the y values are reversed
`axis xy`	puts the axis in the default Cartesian mode
`axis square`	makes the current axes square
`axis normal`	makes the current axes fit the plot window
`axis off`	turns off all axis labelling
`axis on`	turn axis labelling back on
`grid`	adds/removes grid lines to/from the current graph.
`xlim([xmin xmax])`	change the range of the x axis
`ylim([ymin ymax])`	change the range of the y axis

7.2.4 Line types, plot symbols and colour

Line types, and plot symbols, and the line colour may all be set when we create a plot. There are two approaches, firstly, there is an optional input argument to the `plot` function that can contain a series of “flags” which each indicate something about the resulting plot.

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

By combining three (or fewer) of these flags, we construct a particular plot, e.g.,

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

plots `y` against `x` and joins the points with a dashed line whereas

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

draws circles at the data points with no lines joining them. Combinations are valid, e.g.,

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

plots a magenta dashed line, with circles at the data points.

The second approach to setting plot characteristics is to include a series of pairs of optional arguments to the plot command, for instance,

```
plot(x,y, 'color', 'm', 'linestyle', '--', 'marker', 'o')
```

which would draw a plot in the colour magenta, with dashed lines between each point, and the data points themselves shown by ’o’. A brief list of such arguments is shown in the following table:

Option name	Valid values	Default	Effect
`color`	b,g,r,m,c,y,k,w or a $[r, g, b]$-color	b	sets line and marker colour
`linestyle`	’-’, ’–’, ’:’, ’-.’	’-’	sets the line style
`marker`	see table above	none	sets the marker at data points
`linewidth`	positive integers	1	sets the width of lines
`markersize`	positive integers	6	sets the size of markers

Note the American spelling of color. There are many other options for plots (for instance we can change the colour of the inside and edge of a marker independently). See `help plot` for more details.

7.2.5 Fonts

The font refers to the style of writing used. Typically, the main thing we might change about a font would be the size. As noted earlier, several commands (e.g. `text`, `title`, `xlabel`, `ylabel`, `legend`) allow optional arguments of the form `'fontsize'` followed by an integer. For instance we might write:

```
title('This is the BIG title', 'fontsize', 24);
```

which would place a title on the plot in a large font. The color can also be change using the `'color'` option just as for a plot command (see above). A list of common options for fonts is included below

Option name	Valid values	Default	Effect
`color`	b,g,r,m,c,y,k,w or a $[r, g, b]$-color	k	sets font colour
`fontsize`	positive integer	10	text size in “points”
`fontangle`	normal, italic or oblique	normal	style of font
`fontweight`	light, normal, demi or bold	normal	style of font
`rotation`	an angle in degrees	0	rotate the text by the angle

More options can be found in the MATLAB documentation.

However, this will not change the size of text used for numbers on axes. To do so, we need to change the properties of the current set of axes, which we can do using the command:

```
set(gca, 'fontsize', 24);
```

For instance see the results shown in Figure 7.6 where this command has been used to make the log axes more readable. The input argument `gca` is a variable that MATLAB uses to refer to the current set of axes. In fact, `set(gca, ...)` can be used to change many properties of the axes, including where labels appear, and what labels are used. Another common use is to change the thickness of the lines used, e.g. by typing

```
set(gca, 'linewidth', 3);
```

but general use of `set` is beyond the scope of this course, and we will not cover it in more detail here.

One final note: although MATLAB is usually case sensitive, the options used above aren’t, i.e., ’FontSize’ is the same as ’fontsize’.

7.3 Multiple plots

7.3.1 More than one figure

It is common for us to want to see more than one plot at a time. MATLAB allows this through the `figure` function. By itself, `figure` will create a new window, ready for a new plot. If we use `figure(n)`, where it has an integer argument, it will either switch to window `n`, or create a new window `n` if one doesn’t exist. For example, we might type something like

```
x = 0: pi/100: pi;
y = cos(x);
z = sin(x);
figure(1);   % create figure window 1
plot(x,y);   % plot (x,y) in window 1
figure(2);   % create figure window 2
plot(x,z);   % plot (x,z) in window 2

figure(1);   % swap back to window 1
ylabel('y'); % set the y-axis label on window 1
figure(2);   % swap back to window 2
ylabel('z'); % set the y-axis label on window 2
```

This series of commands will result in two figure windows, plotting x versus y and z, with the appropriate y-axis labels.

7.3.2 Multiple subplots

Sometimes we want to do multiple plots, but we wish them to appear as subplots on a single window. We can do this using the `subplot` function. The command takes three input arguments, the first two specify how many rows and columns of subplots should appear. The third specifies which plot to use. For instance, we could take the two plots from the previous example, and place them in the same window using the following statements.

```
x = 0: pi/100: pi;
y = cos(x);
z = sin(x);
figure(1);     % create figure window 1
subplot(2,1,1) % create two subplots, one above the other, and
               %   use the first one for the next plot
plot(x,y);
subplot(2,1,2) % use the second subplot for the next plot
plot(x,z);
```

Figure 7.4(a) shows the result.

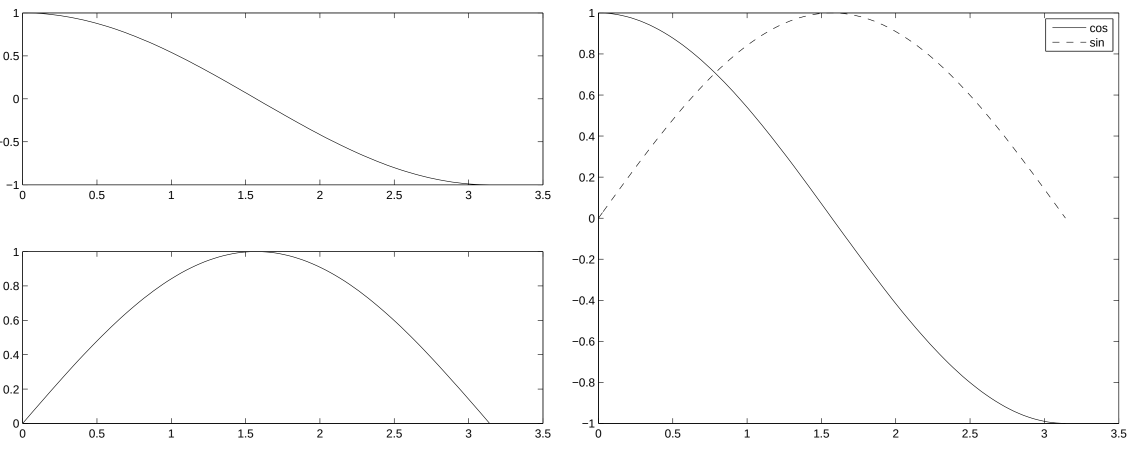
7.3.3 Multiple plots on the same graph

It is sometimes preferable to put two plots onto the same graph so that they can be directly compared. We can do this using the `hold` command. By default, if we call plot twice, the old plot will be removed and replaced by the new one. However, if we call

```
hold on
```

the new plot will be overlaid on the old plot. The command `hold off` reverts back to the default behaviour. For instance, repeating the above example:

```
x = 0: pi/100: pi;
```



(a) Two subplots. (b) Two plots on the same graph.

Figure 7.4: Examples plotting multiple curves.

```
y = cos(x);
z = sin(x);
figure(1);      % create figure window 1
hold off        % remove any old plots on the figure
plot(x,y, '-'); % do the first plot
hold on         % allow us to add an extra plot
plot(x,z, '--');% add a second plot, with a different line style
legend('cos', 'sin');
```

The final command above is for creating a legend for the plots. In its simple form, as illustrated above, we simply call `legend` with a a list of string inputs, and these are assigned to the different plots. Figure 7.4(b) shows the result. Note however, that `legend` has many options.

7.3.4 Plotting a matrix

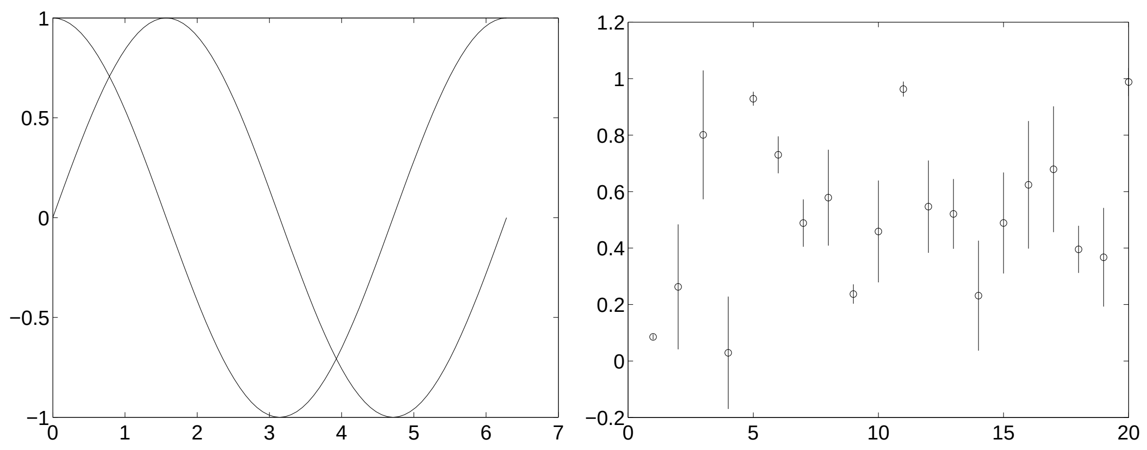
A faster approach to putting multiple plots onto a single graph is to plot a matrix (rather than a vector). MATLAB will then produce (by default) one curve for each column of the matrix using different colours for each line. For instance if we used the following code

```
x = [0: pi/40 : 2*pi]';
Y = [cos(x) sin(x)];
plot(Y);
```

we would plot both the sine and cosine curves on the same graph in different colours. We can use the correct x axis using a command like

```
x = [0: pi/40 : 2*pi]';
Y = [cos(x) sin(x)];
plot(x, Y);
```

and we will get a graph such as shown in Figure 7.5(a) (though in colour). We can make this more general by turning both x and y co-ordinates in the plot into matrices. A simple example of the practicality of this approach is in plotting confidence intervals around data points, e.g., for the purpose of demonstration use random values as follows:



(a) Two subplots. (b) Two plots on the same graph.

Figure 7.5: Examples plotting matrices.

```
  N = 20;
  y = rand(1,N);            % generate some random data
  hold off
  plot(1:N, y, 'o');        % plot the data points
  ci = rand(1,N)/4;         % generate random confidence intervals
  Z = [y-ci; y+ci];         % create the matrix
  hold on
  plot([1:N; 1:N], Z, 'b') % plot vertical bars showing CIs
```

which generates the plot shown in Figure 7.5(b).

7.4 Printing graphs

As noted above, we can save or print a graph directly using the GUI of a plot window. However, we sometimes wish to output a graph from a program. This can be done using the `print` function. The `print` function has a variable list of input arguments which specify

- The output *device* which is typically the format for the output when saving as a file. Common example are as an encapsulated postscript file (a `.eps` file) suitable for inclusion in some reports, or an image such as a JPEG or PNG file.
- The output *file* when saving.
- Other options (not covered here).

Examples appear in the following table:

Print command	result
`print('-deps', 'file.eps')`	print current figure to an EPS file called `file.eps`.
`print('-dpng', 'file.png')`	print current figure to an PNG file called `file.png`.

There are many other options, file formats, and ways of calling `print`. Use `help print` for more details

The size of the output graph can be controlled by setting parameters of the current figure window, which MATLAB refers to as `gcf`. For example

```
set(gcf,'PaperUnits','centimeters','PaperPosition',[0 0 10 12]);
```

would make the figure 10×12 centimeters in size. There are other properties of a figure that can be changed to in turn change the output from the `print` function.

7.5 Colours

MATLAB allows use of colours in plots. There are several ways to do so. The simplest is to use the one character codes given above, with their direct association to a colour. However, some commands do not take such codes, and at other times we wish to use different colours.

There are several approaches, but the most general is to specify a colour as a RGB triple. That is, we give the red, $green$ and $blue$ component of the colour in a 3 element vector. Many colours can be expressed in this way, for instance, the common MATLAB colours are shown below:

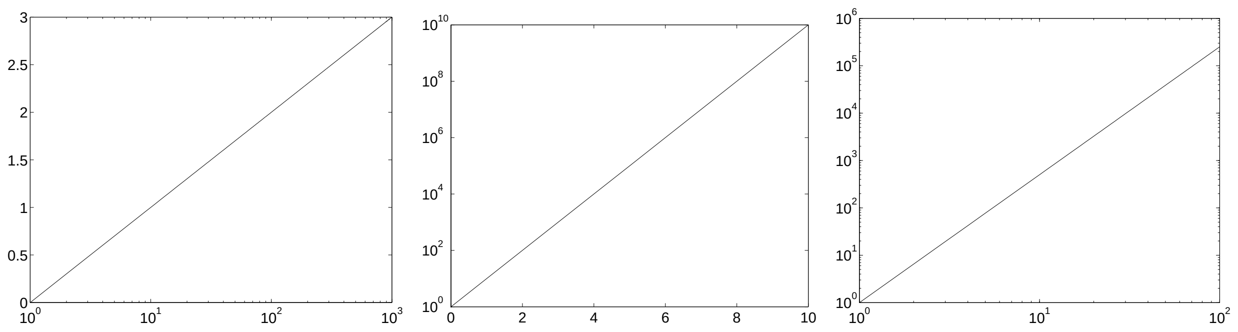
Color code	Color	Red	Green	Blue
b	blue	0	0	1
g	green	0	1	0
r	red	1	0	0
c	cyan	0	1	1
m	magenta	1	0	1
y	yellow	1	1	0
k	black	0	0	0
w	white	1	1	1

A value of 1 (say for the red component) would mean that component is as strong as possible, whereas a value of zero means it is not present at all.

More about RGB colours can be found, for instance, at `http://en.wikipedia.org/wiki/RGB`. Example colours can be found at `http://www.pitt.edu/~nisg/cis/web/cgi/rgb.html`, though note that commonly the RGB values are specified from 0 to 255. To convert to MATLAB values, divide by 255.

An alternative approach to colors sometimes used in MATLAB is to create a `colormap`. The colormap maps a set of integers to a RGB triple. It is represented by a $N \times 3$ matrix, where each row is a RGB triple. An integer n is then mapped to the colour of the nth row of the table. Some plotting commands, such as `surf` and `image` (which we will see later) automatically use the current colormap. MATLAB has a set of predefined color maps that can be used via the `colormap` function. For instance `colormap('default')` uses the standard MATLAB colormap, and `colormap('gray')` uses a grayscale (B&W) colormap. MATLAB has a number of built in examples that demonstrate the power of these color maps, e.g., try

```
load spine
```



(a) `semilogx(x,y)` (b) `semilogy(x,y)` (c) `loglog(x,y)`

Figure 7.6: Examples of log axes graphs.

```
  image(X)
  colormap bone
```

or

```
  load flujet
  image(X)
  colormap(jet)
```

7.6 Advanced two-dimensional plots

MATLAB has many advanced graphing features, and we look at only a few here. For more information and examples use

```
  help graphics
  help graph2d
  help specgraph
```

7.6.1 Base-10 logarithmic plots

It is sometimes beneficial to use plots where one or both axis are presented logarithmically. For logarithmic data of the form $y = k\log_{10}(x)$, the plot of y versus $\log_{10}(x)$ yields a straight line of slope k. It follows that for logarithmic data of the form $y = k\log(x)$, the plot of y versus $\log_{10}(x)$ yields a straight line of slope $k/\log_{10} e$. These can be achieved with the command `semilogx(x,y)`, which is almost the same as the `plot` command, but using a log x axis. For example the plot

```
  x = [1:1000];
  y = log10(x);
  semilogx(x,y)
  set(gca,'fontsize',18)
```

is shown in Figure 7.6(a).

For exponential data of the form $y = 10^{kx}$, the plot of $\log_{10}(y)$ versus x yields a straight line of slope k. (This is since $\log_{10}(y) = kx$ and a plot of kx versus x is a straight line of slope k.) It follows that for exponential data of the form $y = e^{kx}$, the plot of $\log_{10}(y)$ versus x yields a straight line of slope $k \log_{10} e$. These can be achieved with the command `semilogy(x,y)`, e.g.,

```
x = [0:10];
y = 10 .^ x;
semilogy(x,y)
set(gca,'fontsize',18)
```

(results shown in Figure 7.6(b)) and

```
x = [0:10];
y = exp(x);
semilogy(x,y)
set(gca,'fontsize',18)
```

For data of the form $y = x^k$, a plot of $\log_{10}(y)$ versus $\log_{10}(x)$ yields a straight line of slope k. (This is since $\log_{10}(y) = k \log_{10}(x)$ and a plot of $k \log_{10}(x)$ versus $\log_{10}(x)$ is a straight line of slope k.) This can be achieved with the command `loglog(x,y)`. For example Figure 7.6(c) shows the results of

```
x = [0:100];
y = x.^2.7;
loglog(x,y)
set(gca,'fontsize',18)
```

It is useful to note that if we use the `hold on` command to put more than one plot onto a graph, the first plot determines the type of axes. So if we specify `loglog` first, then even if subsequent commands use `plot`, all graphs will appear on with the same log-log axes.

7.6.2 Histograms

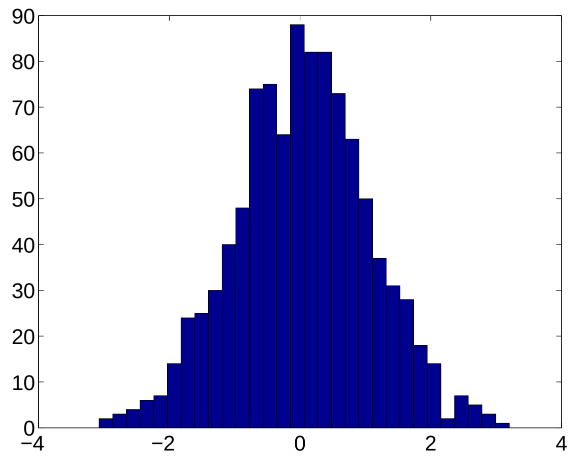
There is a simple approach to producing a histogram in MATLAB. In is simplest form, we simply call `hist(data)`, however, we often want to specify the exact bins being used, or at least their number and we can do so through an optional second argument. For instance:

```
data = randn(1000,3); % generate random "Normal" data
hist(data, 30);       % do a histogram with 30 bins
set(gca,'fontsize',18)
```

Following these commands, MATLAB will automatically scale the bins to include all of the data, and then generate a figure, as shown in Figure 7.7(a).

When the input data is a matrix (rather than a vector) the histogram will show multiple bars for each column of the data.

MATLAB also allows us to put the result of the histogram command into an output vector, rather than a figure, which we can then plot using the various plotting commands we have already seen. If we want to do a histogram plot, though, we can so using the `bar` command, which draws a bar plot of a set of data. For instance,


(a) `hist(data, 30)`

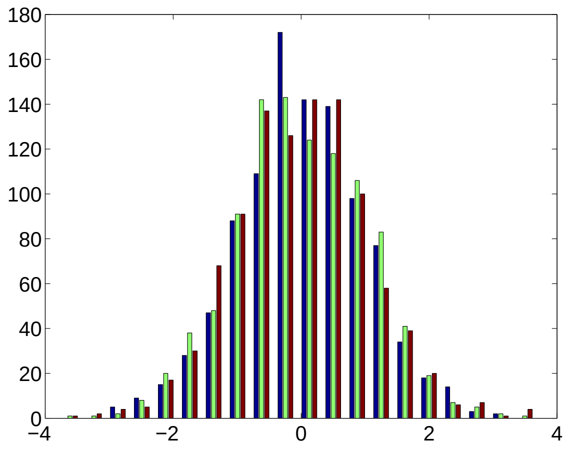

(b) `bar(x, y)`

Figure 7.7: Examples of histograms and bar charts.

```
data = randn(1000,3);   % generate a matrix of data
[y,x] = hist(data, 20); % do a histogram with 30 bins, but don't plot
bar(x, y);
set(gca,'fontsize',18)
```

the results of which are shown in Figure 7.7(b).

7.6.3 The `fill` function

The `fill` function allows us to fill in a polygonal area with a particular colour. The command is executed as follows:

```
fill(x, y, colour)
```

where `x` and `y` vectors specify the x and y co-ordinates of the vertexes of the polygon, and colour specifies the fill colour in one of the formats above (either a letter code or a RGB triple).

An example of the fill command appears below. The code below draws a rainbow. The code constructs an (approximate) elliptical arc for the outside and inside of each "bow" of the rainbow (i.e. for each colour). The code then puts the outer and inner arc together to approximate a circular arc which it then fills with the correct color (specified in the array `colors`).

```
% draw a rainbow
N = 50;
step = 0.07;
r = 1:-step:1-7*step;
theta = 0:pi/N:pi;
alpha = 1.4;
colors = [[1   0   0];   % red
          [1   0.5 0];   % orange
          [1   1   0];   % yellow
          [0   1   0];   % green
          [0   0   1];   % blue
          [0.3 0   0.4]; % indigo
          [0.2 0   0.3]; % violet
         ];
figure(1)
hold off
plot(0,0)
hold on
for i=1:length(r)-1
   x1 = alpha*r(i)*cos(theta);
   y1 = r(i)*sin(theta);
   x2 = alpha*r(i+1)*cos(fliplr(theta));
   y2 = r(i+1)*sin(fliplr(theta));

   x = [x1 x2];
   y = [y1 y2];

   % plot(x, y, '.');
   fill(x, y, colors(i,:));
end
axis equal % make the scales on the two axes equal
axis tight % fit the axes tightly around the plot
```

The output of which is shown in Figure 7.8.

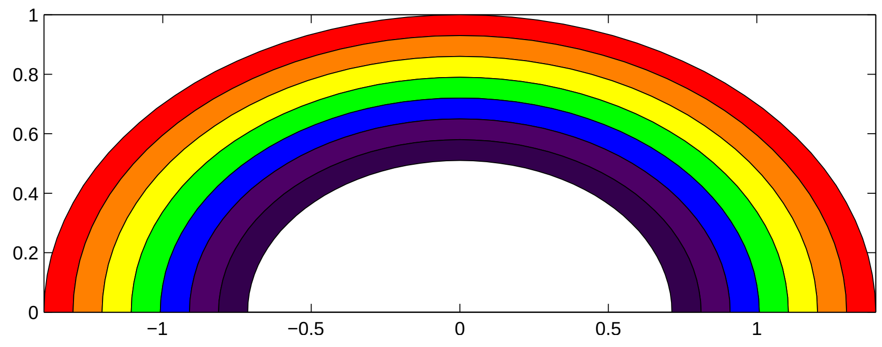


Figure 7.8: Output of rainbow.m.

7.6.4 Images

Displaying images is simple, but there are several details to get right. Firstly, there are two main representations of an image. We can represent it as an array of numbers that are mapped to colors via a colormap, or as an 3D array where two dimensions represent the 2 spatial dimensions of an image, and the third allows us to store the components of the colours (for instance the RGB triple). When creating or reading an image, it is important to understand what format it will be in. See documentation for `imread` to learn more.

The main command for displaying an image is the creatively name `image(C)` function. If `C` is a $M \times N$ matrix, the values will be mapped to colors in the display using the current colormap (see Section 7.5). If the colormap is subsequently changed, then the image will change colours. The function `imagesc` scales the values of `C` to the potential range of the colormap before displaying.

If `C` is a $M \times N \times 3$ array, then it will be displayed interpreting each of the triples as RGB colours, i.e., `C(:,:,1)`, `C(:,:,2)` and `C(:,:,3)` are interpreted as red, green and blue intensities, respectively. The colormap is ignored. The range of values for `C` depends on its type. For a standard `double` array, the range is $[0, 1]$, but if the array uses an integer type such as `uint8` the range is $[0, 255]$. Care must be taken when importing data that the type is correct.

We create a simple example below

```
N = 250; M = 250; alpha = 0.001;
x = 1:N;
y = 1:M;
[X,Y] = meshgrid(x,y);
C = exp( -alpha*((X-N/2).^2 + (Y-M/2).^2) );
colormap('gray'); % change the colormap to B&W
image(255*C);     % show the image using the colormap
axis image;       % make the axes look right

% create a RGB image
C(:,:,1) = exp( -alpha*((X-N/2).^2 + (Y-M/2).^2) );
C(:,:,2) = exp( -alpha*((X-N/2).^2 + (Y-M/2).^2) );
C(:,:,3) = exp( -alpha*((X-N/2).^2 + (Y-M/2).^2) );
image(C);
axis image;
```

7.6.5 Others

There are many other MATLAB functions for drawing plots:

- `plotyy`: Sometimes we wish to plot two curves with different scales on the same plot. `plotyy` does this with one axis on the left, and the other on the right.
- `pie`: allows us to draw a pie chart.
- `polar`: draw a plot of polar co-ordinate data.
- `stem`: does a "stem" plot, showing lines growing from the x axis with a marker at the top. Often used in discrete signal processing, along with stairstep plots, which we can draw with`stairs`.
- `contour`: allows one to draw a contour plot of some set of data. For example, if a matrix `Z` represented heights, then a contour plots would show curves of constant height.
- `quiver`: allows plots of a vector field. There are many other volume and vector visualization functions.

- `movie` in conjunction with `getframe` can be used to create animations that can then be saved as movies.

There are many others: rose plots, waterfall plots, Voronoi diagrams, Pareto charts, comet charts and so on. However, rather than considering each, we shall go on to consider another large group of plots, those in 3D.

7.7 Three-dimensional plots

MATLAB has a number of functions for displaying and visualising data in three dimensions. The function `plot3` is the 3-D version of `plot`, which is called in the following form:

```
plot3( x, y, z )
```

where `x`, `y` and `z` are (typically) vectors specifying the (x, y, z) co-ordinates of a series of points. This command draws a picture of the 3-D curve through the points whose coordinates are the elements of the vectors `x`, `y` and `z`. As an example

```
z = 0 : pi/50 : 10*pi;
x = exp(-0.02*z) .* sin(z);
y = exp(-0.02*z) .* cos(z);
plot3( x, y, z);
xlabel('x-axis');
ylabel('y-axis');
zlabel('z-axis');
```

produces the inwardly-spiralling helix shown in figure 7.9. The `plot3` command admits the same types of options as the 2D `plot` command, so we can set line colour and style, the marker types, and linewidth. We can also control the axes as before, and even place text on the graph using labels, or the text command called with four input parameters, e.g.,

```
text(x,y,z, 'text we wish to write');
```

The figure's GUI also allows the same types of options as before, but there is an additional useful button for 3D plots. The image displayed is really a 2D projection of the 3D curve we wish to examine. The rotate button allows us to use the mouse to rotate the projection that we see so that we can examine the curve from different perspectives. The viewpoint can be changed in the program using the `view` function.

7.7.1 Mesh surfaces

Another standard activity in 3D is to plot a representation of a surface. For instance, imagine we wished to plot the function:

$$f(x, y) = 5y - x^2$$

over the range $-4 \leq x \leq 4$ and $0 \leq y \leq 5$. The simple approach to plotting such a function is to calculate it at a set of sample points, and plot an approximation of the surface at these points using the `mesh` command.

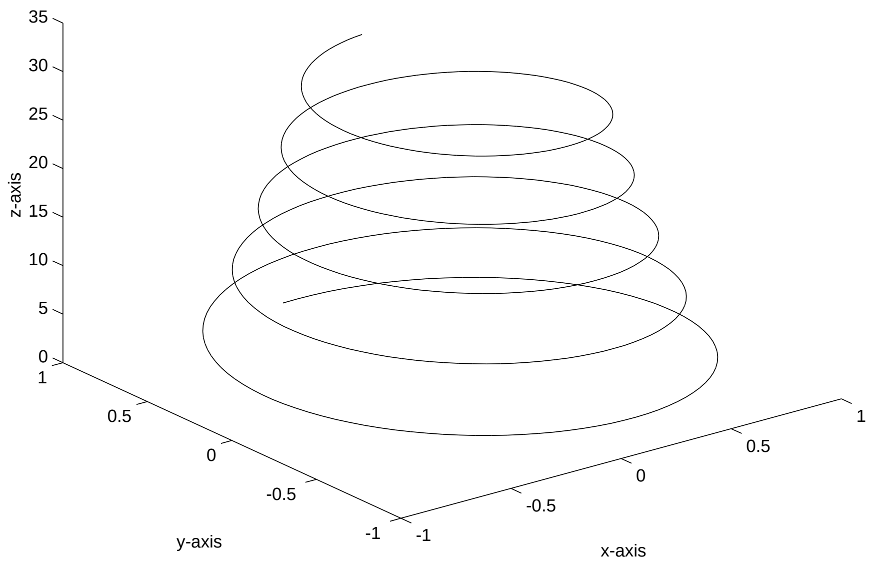


Figure 7.9: An example of `plot3`.

Naively, we can compute the function at a series of points on this range using nested `for` loops:

```
for column = 1:7
  x(column) = column-4;
  for row = 1:6
    y(row) = row-1;
    Z(row, column) = 5*y(row) - x(column).^2;
  end
end
```

This will result in

```
x = [-3    -2    -1     0     1     2     3]

y = [ 0     1     2     3     4     5]

Z =
  -9    -4    -1     0    -1    -4    -9
  -4     1     4     5     4     1    -4
   1     6     9    10     9     6     1
   6    11    14    15    14    11     6
  11    16    19    20    19    16    11
  16    21    24    25    24    21    16
```

However, this approach is rather clumsy and inefficient. A preferable approach is

```
step = 1;
x = -3:step:3;
y = 0:step:5;
[X,Y] = meshgrid(x,y);
Z = 5*Y - X.^2;
```

which produces the same vectors `x` and `y`, and matrix `Z`. It also creates the matrix `X` with the rows equal to the vector `x`, and matrix `Y` with columns equal to the vector `y`. This approach has the advantages that it is more efficient (faster), and also allows us to change the `step` parameter to plot more, or fewer points as needed. The statements

```
mesh(X, Y, Z);
set(gca,'fontsize', 24);
xlabel('x');
ylabel('y');
```

then plots the surface Z as a "wire frame", where the matrices provide the (x, y, z) co-ordinates on the surface we wish to plot. In our case, because we have a regular grid, we might also have called `mesh(x,y,Z)`. In each case, we might imagine the values `Z` as heights on a map of terrain at each (x, y) co-ordinate. The plot produced by these commands in shown in Figure 7.10(a). Note that by default MATLAB will use colours from the current colormap to highlight changes in value (i.e., the range of values of `Z` will be scaled and mapped to the colormap and these colors will be used to change the mesh).

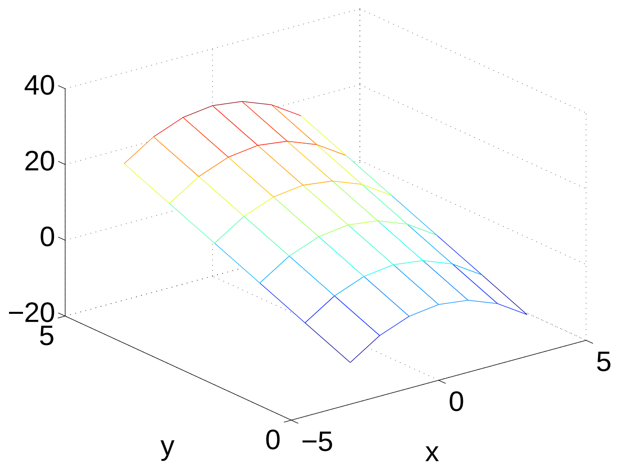
An alternative to a mesh surface is to fill in the rectangles between points using the `surf` function. For instance, the above surface can be displayed as in Figure 7.10(b) using

```
surf(x, y, Z);
set(gca,'fontsize', 24);
xlabel('x');
ylabel('y');
```

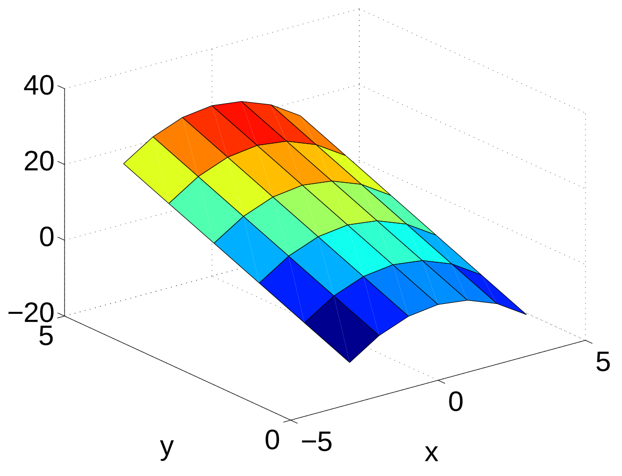
Again MATLAB uses colours from the current colormap to highlight changes in value. The "faceted" view shown by default in `surf` can be smoothed by changing the `shading` mode of the plot. The default is `faceted` interpolation, but we can also use `flat` to get rid of the mesh lines, or `interp` to make the shading color smooth so that we see an apparently smooth surface. Figures 7.10(c) and 7.10(d) show the results of the following commands:

```
surf(x, y, Z); shading flat;
surf(x, y, Z); shading interp;
```

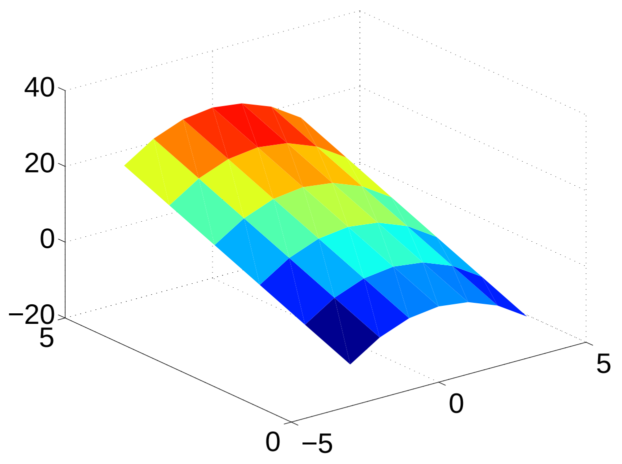
More information on 3D plots can be found from the individual `help` for each function, and from `help graph3d`.



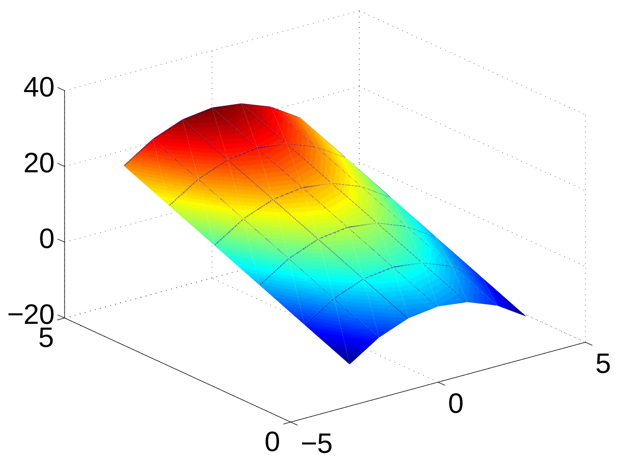
(a) `mesh(x,y,Z)`



(b) `surf(x,y,Z)`



(c) `surf(x,y,Z); shading flat;`



(d) `surf(x,y,Z); shading interp;`

Figure 7.10: Examples of surface and mesh plots.

Chapter 8

Defining Functions with M-files

As we have seen, MATLAB has many built in functions. However, it is common for us to wish to create our own functions. This allows us to encapsulate a group of MATLAB statements. Encapsulation improves code re-usability, by enabling us and others to reuse the same function more than once without retyping the same code. It improves code maintainability by allowing us to fix a bug in one place (in the function) rather than in all of the places we might have used that function.

We call this *modularity* in programming. It separates the internal logic of a function from the *interface* — the inputs and outputs of the function. Modularity is particularly important for large software systems, where more than one programmer may be working on the system. It allows the two programmers to work on separate code without knowing the details, except for the function interfaces. Other programming languages provide modularity in various ways (e.g. subroutines, procedures, etc.), but MATLAB's primary approach is to allow us to define our own functions.

Another reason to use functions in MATLAB is that they will often run faster than the equivalent script file.

8.1 Some examples

We will start with a couple of illustrative examples of function definitions. As when creating script `.m` files, we will use MATLAB's editor to input the commands into a file.

8.1.1 Harmonic oscillators

If two coupled harmonic oscillators are considered as a single system, the output of the system as a function of time t could be something like

$$h(t) = \cos(8t) + \cos(9t).$$

We can create a function `.m` file called `harmonic.m` with the following two lines

```
function h = harmonic(t)
h = cos(8*t) + cos(9*t);
```

Then we can then call the function in another piece of MATLAB code, for instance at the MATLAB prompt, just as we do for other functions, e.g.,

```
x = pi/2;
y = harmonic(x);
```

Note the following:

1. The name of the function — what we type when we want to call it — is defined on the first line of the file by the statement `function y = harmonic(t)`. The filename of the `.m` file must match this name, for instance in this case we save the file as `harmonic.m`.

2. The variable `t` in the function `harmonic.m` is the *input variable*. It implicitly creates a variable called `t` inside the function. The variable will be initialized with the value of the corresponding input variable in the function call. In this example, it will start with the value `pi/2`, obtained from the variable `x`. Note that if we change `t` inside the function, it has no effect on `x`. The input serves only to set the initial value of `t` and thereafter the connection between `x` and `t` is broken.

3. The variable `h` in the function file is the *output variable*. We must assign a value to `h` at some point in our function `.m` file. The value we assign will be passed out to the output variable when we call the function. For instance in the example, the value of `h` inside the function will be assigned to the variable `y`, after the function is finished.

4. It is good coding practice to always use semi-colons at the end of lines in MATLAB functions so that the function has no unintended "side-effects" such as printing out intermediate values. However, we may ignore this guideline during debugging of a function.

5. We don't have to use a variable as input to a function, we could also use an expression, or another function. We can also pass vectors and matrices as inputs, for instance

   ```
   y = harmonic([0:pi/40:6*pi]);
   plot([0:pi/40:6*pi], y);
   ```

We sometimes talk of *input arguments* instead of input variables. They are the same thing (likewise for output arguments).

8.1.2 Statistics

Consider the following more general example which calculates the mean and the standard deviation of the values in the vector `x`. We'll save this function in a file called `stats.m`.

```
function [average, standard_deviation] = stats(x)
% Calculates the mean and standard deviation of
% the data in the vector x.
average = mean(x);
standard_deviation = std(x);
```

We can now test it with some random numbers:

```
r = rand(100,1);
[ave,st_dev] = stats(r)
```

The function will calculate the mean and standard deviation of the input (random) numbers, and will pass these to the output variables `ave` and `st_dev`.

8.2 The basic rules for function files

A function M-file `name.m` has the following general form.

```
function [out1, out2,...  ,outN] = name(inp1, inp2,...  ,inpM)
% comments to be displayed by help
statements
out1 = expression1;
out2 = expression2;
...
```

where `...` is used here to indicate that there may be an indefinite number of input and output variables (you would not type this). The function would typically be called by typing

```
[o1, o2, ... , oN] = name(i1, i2, ... ,iM);
```

The rules that we *must* follow when defining a function are as follows:

1. The function name and the name of the file containing it must be identical except for the `.m` filename extension. Remember MATLAB is case sensitive. It is suggested that you use lower-case for function names as this will increase portability of your code to other operating systems.

2. The function name must follow MATLAB 's rules for variable names, as must the input and output variables.

3. The function file must start with the reserved word `function`, followed by a vector of the outputs, and equals sign, the name of the function, and round brackets with a list of input variables.

4. If there is only one output variable square brackets are not necessary (even if it is a vector). If there is more than one output variable, the output variables must be separated by commas and enclosed in square brackets (as with a vector). This is true both in the function definition, and when we call a function.

5. If an output is unassigned, this may cause an error, so we need to have at least one statement assigning values to each output variable. Often these statements are at the end of a function file. They don't have to be, but it can improve program readability to place them there.

6. Note that if the function changes the value of any of its input variables the change does not affect the corresponding variable used in the call to the function.

8.2.1 Optional inputs and output

We don't need to call a function with all of its inputs and output variables. The input variables that are not assigned will be undefined until they are given a value in the program, as will unassigned outputs. For instance, we could call

```
r = rand(100,1);
ave = stats(r);
```

This code would assign the variable `ave` the value of the internal output variable `average`, but the output variable `standard_deviation` is not assigned to any workspace variable.

Likewise, we can define a function `add_them_up` that adds two numbers, when we input two, or just outputs the original value when we only use one input as follows. It uses the function `exist` to test whether the second variable exists and reacts appropriately.

```
function result = add_them_up(x,y)
if (exist('y','var'))
  result = x+y;
else
  result = x;
end
```

We can now call this function two different ways

```
add_them_up(1, 3)
add_them_up(4)
```

and in both cases it will output the value 4.

This type of construction allows us to have optional inputs to a function, with default arguments when an input is not used. It is somewhat limited, because it is dependent on order. We can't omit input parameter N, but include $N+1$. A better approach might be to use an input variable structure, but this is outside the scope of this course.

The above mechanism is also quite clumsy when the number of inputs is large. To aid in this MATLAB automatically creates two extra variables when a function is called:

Variable	Meaning
`nargin`	The number of input variables for this function call.
`nargout`	The number of output variables for this function call

We could use these in the above example be rewriting the function as follows:

```
function result = add_them_up(x,y)
if (nargin == 2)
  result = x+y;
else
  result = x;
end
```

An even more flexible mechanism is to use `varargin` or `varargout` in place of all or part of the list of input arguments. This allows us to specify explicitly that we don't know the number of input or output arguments. This type of mechanism is used in functions such as `plot` to allow us to specify an arbitrary number of options to the command, or in `fprintf` to allow us to provide an arbitrarily long list of variables to substitute into the format string. We will no go further into this type of construction here, but it is important to realize that such facilities exist.

8.2.2 Scope

The input variables (`inp1`, `inp2`, ...) and output variables (`out1`, `out2`, ...) are the function's means of communicating with the workspace. The input variables allow you to pass in values to the function, and the output variables allow values to be passed out. There can be some other interactions, for instance if the function reads a file, or presents at plot, but typically these do not pass values in or out of the workspace.

Variables defined inside a MATLAB function have *local scope*. This means that they cannot be seen outside of this function. We cannot obtain their values, or interact with these variables in any way. For instance, they will not appear in the workspace. Even the implicitly created input and output variables cannot be accessed outside the function except at the start (for input variables) and the finish (for output variables).

There are various reasons for this. Firstly, it is highly desirable to minimize the side-effects of functions. This makes program behaviour more predictable. Side-effects can have unanticipated consequences!

Secondly, giving function variables local scope improves the modularity of code. It simplifies the interface between the function and other programs. It makes the interface as simple as calling the function, e.g.

```
 [ave,st_dev] = stats(r)
```

The result is that MATLAB functions are easy to use without necessarily examining all of the MATLAB code in the function. We just need to look at the first line of the function to see how to call it. In essence, we can treat a MATLAB function as a black-box that performs an action, and we don't need to understand how it does it.

The third important result of local scope is that we avoid *name collisions* between variables inside and outside the function. Say I want to use a complex function defined by third party. It may define many internal variables. I don't want to have to make sure all of my variables have different names. For instance, they may use the common name `x` inside their code, and I may want to use `x` for something different outside of the code. I don't want to have to check that will be OK. This problem may sound trivial, but imagine I am calling a hundred different functions each with its own variable names. If there was overlap in the namespaces, then not only might my names collide with the functions' variable names, but also the functions might have collisions between each others variables. Local scope of internal variables guarantees that variables can co-exist peacefully within their own functions.

It is possible to create "global" variables that are accessible elsewhere in the MATLAB program. However, this mechanism should very rarely be used for the reasons described above, and so we will not describe how to create such variables here.

8.2.3 Nested functions

One function can call another function! In fact this is common, particularly with respect to MATLAB's built in functions. We will defer discussion of functions that call themselves (*recursive* functions) until Section 8.5. Apart from the examples above, a simple example of a function that calls another function is one we used earlier, `normal_cdf(x)`, which calculates the normal distributions cumulative distribution function in terms of MATLAB's built in function `erf(x)` (which calculates the "error" function).

```
function result = normal_cdf(input)
% compute the CDF of the normal distribution using matlab's
%  built in erf function
result = (1 + erf( input / sqrt(2) ))/2;
```

We can also define a new function inside another function. For instance MATLAB `.m` file may also contain extra sub-functions (extra functions defined inside the same file). Such functions are only visible to the main function defined inside the file. Each starts with its own function definition line. We discourage this, from a programming style perspective, because the scope rules for such functions become complicated, and the behaviour of functions with the same name is also complicated, and it is easy for program to become confusing.

8.2.4 Example: Newton's method re-visited

Newton's method may be used to solve a general equation $f(x) = 0$ by using the iterative process

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)},$$

where $f'(x)$ is the first derivative of the function $f(x)$. We can write a general script to implement Newton's method by first writing M-files for the function $f(x)$ and $f'(x)$. Let's consider the particular example of $f(x) = x^3 + x - 3$. Using an editor we create and save a file `f.m` containing the function we are interested in

```
function y = f(x)
% function of interest: f(x)=x^3+x-3
y = x^3 + x - 3;
```

and another file `f_dashed.m` containing its derivative

```
function y = f_dashed(x)
% function of interest's derivative: f'(x)=3x^2+1
y = 3*x^2 +1;
```

We now need to write our *script* file, `newton2.m`, which will stop when either the absolute value of $f(x)$ is less than 10^{-8} or after 20 steps of the iterative process.

```
% Newton's method example 2
format long
steps = 0; % a counter for the number of steps
x = input( 'Initial Guess: ' );
```

```
  tolerance = 1.0e-8;

  while (abs(f(x)) >= tolerance) & (steps < 20)
    x = x - f(x) / f_dashed(x);
    disp( [x f(x)] )
    steps = steps + 1;
  end

  if abs(f(x)) < tolerance
    disp( 'Zero is at approximately ' )
    disp(x)
  else
    disp( 'Zero not found after 20 steps')
  end
```

Here's an example of the output starting with an initial guess of 1.5

```
  Initial Guess: 1.5

     1.25806451612903   0.24923634654762
     1.21470533274159   0.00701403868629
     1.21341278623619   0.00000608597942
     1.21341166276308   0.00000000000459

  Zero is at approximately
     1.21341166276308
```

8.2.5 Programming style

Apart from the usual programming style guidelines we have already explained, there are some additional style considerations when we write MATLAB functions.

Comment lines up to the first non-comment line in a function file will be displayed if `help` is requested for the function name. This allows us to put interesting information about the function in a place that is easily accessible without reading the file itself. It is common to use these lines of comments to define (i) what the function does, (ii) what its inputs and outputs are, and (iii) to provide meta-information such as the author of the function, its version, and the dates it was created and revised.

The first line of a MATLAB function's comment, and its the function name are also used in `lookfor` command. Hence we should (i) choose a meaningful function name, and (ii) carefully select the first line of comments to make this function easy to find.

It is good coding practice to always use semi-colons at the end of lines in MATLAB functions so that the function has no unintended "side-effects" such as printing out intermediate values.

It is also good practice to avoid redefining common functions. This will avoid simple bugs in code where a function gives an unexpected result.

In any programming language, care should be taken to check input arguments carefully! This makes good sense, as a user of a function may not have read the documentation carefully and may make a mistake in calling the function. Also, failing to check validity of input arguments is a major source of security holes in various pieces of computer code. We generally omit such checks in our examples, to keep them short and simple, but they should not be omitted in practice.

Finally, we should (almost) never use global variables.

8.3 Function names as input variables with `feval`

It is possible that we don't know the function we wish to call when we are writing our program. For instance, we might wish to use Newton's method (above) for a function that a user inputs. We need a way of computing the result of a function, where the function name is held in a variable. To do so we use the `feval` function, for instance:

```
feval( 'sqrt',9)
```

would give the answer 3 (the square root of 9). So `feval('sqrt',x)` is the same as `sqrt(x)`!

The first argument of `feval` is a *string* (i.e. a name enclosed within single quotes) representing the function to be evaluated. The subsequent inputs to `feval` acts as inputs to the function of interest.

In our earlier Newton iteration scheme it would be useful if we could write a general script that accepts the function as an input variable – this would make the script that much more general (and useful). We would therefore like to call `newton` as follows:

```
x0 = 1.5;
[x f conv] = newton( 'f', 'f_dashed', x0)
```

where `'f'` and `'f_dashed'` are the names of the function M-files containing $f(x)$ and $f'(x)$ that we previously defined, and `x0` is the initial guess. The outputs are the approximate location of the zero, the function value at the zero (which should be close to zero) and a variable `converged` to indicate whether or not the output process has converged. The complete, new, M-file `newton.m` is as follows:

```
function [ x, f, converged] = newton(fn, derivative_fn, x0)
% Newton iteration
% Performs Newton iteration to find the root of
% function fn with derivative derivative_fn.
% Initial guess is x0. Returns the final value of
% x and f(x) and the flag converged (1 =
% convergence, 0 = divergence).

steps = 0;
tolerance = 1.0e-8;
x = x0;

while (abs(feval(fn,x)) >= tolerance) & (steps<20)
   x = x - feval(fn,x) / feval(derivative_fn,x);
   disp( [x feval(fn,x)] )
   steps = steps + 1;
```

```
end

f = feval(fn,x);
if (abs(f) < tolerance)
   converged = 1;
else
   converged = 0;
end
```

Now we can call the function as required above, i.e.,

```
x0 = 1.5;
[x f conv] = newton( 'f', 'f_dashed', x0)
```

but we could also call it as

```
x0 = 1.5;
[x f conv] = newton( 'sin', 'cos', x0)
```

and we would now find a zero of the sine function. This gives us a lot more power to create numerical routines that are not tied to particular functions, but can instead take the function of interest as a input. This is a common approach for many optimization routines or DE solvers.

An alternative, slightly more general, method to implement the same process is to use *function handles*. We can obtain a handle to a function using the `@` symbol. The handle is analogous to a pointer in `C`, but it is outside the scope of this course, so we shall simply note that more information can be found using `help function_handle`.

8.4 Inline and anonymous functions

Sometimes, we have a very short function that we want to use several times in a program, but don't think is worth creating a separate `.m` file for. We can use an *inline* function for such a function.

```
cube=inline('x^3');  % define the inline function
y = cube(3);
```

The above code would set $y = 27$, the cube of 3.

The function (and its input arguments) is specified implicitly in the string `'x^3'`. Inline functions don't leave much space for comments, and are not always as clear as a proper `.m` file, so we generally avoid their use except for trivial functions.

Anonymous functions are functions without a name. MATLAB has methods for constructing such functions, and keeping a reference (a handle) to them without a name, but these are beyond the scope of this course.

8.5 Recursion

Many mathematical functions are defined *recursively*, that is, they are defined in terms of themselves. For instance, the factorial function may be defined recursively as

$$n! = n \times (n-1)!$$

provided we define $0! = 1$. MATLAB allows functions to call themselves; a process that is called *recursion*. We can write a M-file for the factorial function, `fact.m`:

```
function y = fact(n)
% Factorial function
% Recursive evaluation of n!

if n == 0
  y = 1;
else
  y = n * fact(n-1);
end
```

The above function simply calculates the value of $0!$ to be 1, but when we call it for $n > 0$, it calculates the value by calling itself again. The program terminates (for non-negative integer inputs) because each time it decrements the input argument to the recursive call of itself.

In general recursion is *not* efficient. An iterative approach to the same calculation, e.g.,

```
function y = fact(n)
% Factorial function
% Iterative evaluation of n!

y = 1;
for i=1:n
  y = y * i;
end
```

would be much faster. In fact, because of the additional memory used by all of those function calls, the recursive approach may fail for large n. In addition, a great deal of care must be taken that the recursion terminates, and we don't fall into an infinite recursion. The recursive implementation of factorial given above would fall into infinite recursion if the input argument is not an integer. Care should be taken to check input arguments carefully!

In point of fact, the above is still not the most efficient approach in MATLAB. We can create a vectorised approach as follows:

```
function y = fact(n)
% Factorial function
% Vectorized evaluation of n!

if n == 0
  y = 1;
else
  y = prod(1:n);
end
```

which would be faster again.

However, there are some programing tasks which are relatively easy using recursion, but quite hard using an iterative approach. The Tower's of Hanoi problem is a classic case. The puzzle was invented by the French mathematician douard Lucas in 1883, and we describe it below.

Towers of Hanoi: There is a legend about an Indian temple which contains a large room with three time-worn posts in it surrounded by 64 golden disks of different sizes. The priests of Brahma, acting out the command of an ancient prophecy, have been moving these disks, in accordance with the rules: they must move all of the disks from one post to another but may never place a larger disk on a smaller.

The trick, in this puzzle, is to realize that to move all 64 disks, we need only move the top 63, then move the bottom disk to the third post and move the top 63 disks onto the third post as well. Likewise, we can move the 63 disk stack by first moving the 62 disk stack, and so on. Therefore this problem has an obvious recursive implementation. The following function implements this, and illustrates how it works.

```
function disks = move_subtower(disks, n, i, j, do_plot);
% move a subtower of size n from post i to j

if (nargin < 5)
    do_plot = 0;
end

if (do_plot)
    plot_disks(disks);
    pause;
end

if (n==1)
    disks = move_disks(disks, i, j);
else
    k = setdiff([1:3], [i j]);
    disks = move_subtower(disks, n-1, i, k);
    if (do_plot)
      plot_disks(disks);
      pause;
    end

    disks = move_disks(disks, i, j);
    if (do_plot)
      plot_disks(disks);
      pause;
    end

    disks = move_subtower(disks, n-1, k, j);
    if (do_plot)
      plot_disks(disks);
      pause;
    end
```

```
end
```

The inputs are a vector giving the pole each disk is on (assuming they are in order of size), the number of disks to move, and the two posts we wish to move a stack between, plus an optional argument that will allow us to plot the results. The function uses two other functions, one that moves individual disks `move_disks`,

```
function disks=move_disks(disks, i, j);
% move the top disk from post i to post j
%    really should also check for errors
k = max(find(disks == i));
disks(k) = j;
```

and one to plot the towers `plot_disks`,

```
function plot_disks(disks);
% plots disks

n = length(disks);

figure(1)
hold off
plot(0,0)
hold on
for i=1:3
    plot([i i], [0 n+1], ...
        'k', 'linewidth', 10);
end
set(gca, 'xlim', [0 4]);
set(gca, 'ylim', [0 n+2]);

n_disks = 0.5*ones(3,1);
for i=1:n
   disk_ra = 0.5*(1 - i/(n+1));
   post = disks(i);
   plot([post-disk_ra post+disk_ra], ...
        [n_disks(post) n_disks(post)], ...
        'linewidth', 20);
   n_disks(post) = n_disks(post) + 1;
end
```

We would call the function as follows:

```
  N = 7; % do a stack of 7 disks
  disks = ones(N,1); % initially all disks are on post 1
  plot_disks(move_subtower(disks, N, 1, 3, 1));
```

Note, however, that this program makes two recursive calls for each level of the tower to be moved, so the total number of recursive calls for a tower of height N would be 2^N, which grows quite quickly. For instance, so a stack of 64 disks, we would require $2^{64} \simeq 2 \times 10^{19}$ function calls. The universe is not in any immediate danger!

Chapter 9

0-1 vectors

As we have seen, MATLAB is more efficient when we can use vector operations instead of `for` loops. Vectorization of matrix of some operations is very natural, but other operations need some thought. In particular, the best approach to vectorise conditional statements (`if-else`'s) is not always obvious. One handy trick is the use of logical vectors (vectors of logical variables). Logical variables are represented in MATLAB as 0's (FALSE) and 1's (TRUE), and so we sometimes call these *0-1 vectors* and *matrices*. To introduce this concept first carry out the following at the MATLAB prompt:

```
r = 0:0.2:1
z = (r <= 0.5)
```

When vectors are involved in logical expressions such as the second step above, the comparison is carried out *term-by-term*. The brackets are not necessary, but make the statement more readable by clarifying that a logical vector is being assigned to `z`. If the comparison is true for a particular element of the vector, the resulting vector has a `1` in the corresponding position, otherwise it has a `0`. Thus the response to these commands will be

```
r =
         0    0.2000    0.4000    0.6000    0.8000    1.0000
z =
     1     1     1     0     0     0
```

We can see that the vector `r` is constructed to take values from 0 to 1 in steps of 0.2. The vector logical `z` takes values of 1, where `1` is less than (or equal to) 0.5 and values of `0` otherwise. The `whos` command returns

```
Name       Size               Bytes  Class      Attributes

r          1x6                   48  double
z          1x6                    6  logical
```

Note that the logical array `z` takes only 6 bytes (one byte for each term in the vector), and so it is much more compact than a double array (though it could be more compact still as it only *needs* 1 byte for each eight elements).

We can easily construct other 0-1 vectors, e.g.,

```
  r = 0:0.2:1;
  z = r == 0.4
```

which returns

```
z =
     0     0     1     0     0     0
```

Note that in the statement the first = sign is an assignment operator (it assigns the value of the right-hand side to the variable `z`). The second "double" `==` is the *comparison* operator that tests where `r` is equal to 0.4.

We can use vector and matrix operations to work on logicals just as we can for other variables, for instance a `.*` product acts like and AND operation, e.g.,

```
  r = 0:0.2:1;
  z = (r <= 0.5)
  y = (r >= 0.2)
  x = z .* y
```

results in

```
  z =
       1     1     1     0     0     0
  y =
       0     1     1     1     1     1
  x =
       0     1     1     0     0     0
```

We could have obtained the same result with

```
  x = z & y
```

where & is MATLAB's AND operator. Other logical and arithmetic operators can be combined to construct more complicated expressions. Combinations of logical and numerical vectors can also be used to great effect as we shall see in the following examples.

9.1 Combining logical and numerical vectors

The key to vectorization of conditional statements is the combination of logical and numerical expressions. We explore this through several examples.

9.1.1 Avoiding division by zero:

Suppose we want to plot the $sinc$ function over the range $x \in [-2\pi, 2\pi]$. Typically we consider the sinc function to be $\sin(x)/x$. We can, of course, set up a vector containing the x-values

```
  x = -2*pi : pi / 20 : 2*pi;
```

and then calculate

```
y = sin(x) ./ x;
```

In this case MATLAB will return the error message `Warning: Divide by zero` and at $x = 0$ the vector y will contain `NaN` (Not a Number).

However, the precise definition of the sinc function is

$$\text{sinc}(x) = \begin{cases} \sin(x)/x, & x \neq 0 \\ 1 & x = 1 \end{cases}$$

The formal definition above explicitly removes the problem at $x = 0$. We can easily check that the resulting function is continuous, and defined everywhere, hence the limit as $x \to 0$ of the sinc function is 1. We exploit this by computing the function at a value very close to zero, rather than at zero. We will replace the element where `x==0` by `x = eps`. This is a special MATLAB value that is defined at the difference between `1` and the next largest number that can be represented in MATLAB; it has the value `eps = 2.220446049250313e-16`. To remove the troublesome `0` from x we can use

```
x = x + ( x == 0)*eps;
```

The expression `x == 0` returns a 0-1 vector with a single `1` in the position corresponding to the zero element; `eps` is only added to this value, so that the new vector is identical to the old, except that it no longer has a zero value anywhere. We can now plot the graph of sinc(x) correctly using

```
x = -2*pi : pi / 20 : 2*pi;
x = x + ( x == 0) * eps;    % change x=0 to x=eps
y = sin(x) ./ x;
plot(x,y)
```

The beauty of the above approach is we don't assume that `x` is equal to zero anywhere. If no term of the vector is ever zero, then the statement `x = x + ( x == 0)*eps;` has no affect. It only effects those places where `x==0`.

9.1.2 Using 0-1 vectors to replace `else-if` ladders

Income tax in Australia depends in a non-linear way on your income. As your income increase, the *marginal tax* — the tax you pay in each extra dollar of income — increases in a series of steps. The rates, as of 2007-2008 are given at `http://www.ato.gov.au/individuals/content.asp?doc=/content/12333.htm&mnu=5464&mfp=001/002` as

Taxable Income	Marginal Tax Rate	Total Payment
<$6,000	0.00	Nil
$6,001-$30,000	0.15	15c for each $1 between $6,000-$30,000
$30,001-$75,000	0.30	+30c for each $1 between $30,000-$75,000
$75,001-$150,000	0.40	+40c for each $1 between $75,000-$150,000
>$150,001	0.45	+45c for each $1 over $150,000

ignoring the medicare levy. The above table expressed income tax rates in the standard form presented to the public, but another way to write these is to look at the rate *differences*, e.g., the difference between the tax bracket from $30,000-75,000 and $6,000-30,000 is $0.3 - 0.15 = 0.15$. For each dollar above the relevant tax bracket we pay this extra *difference* in the income tax. In tabular form:

Taxable Income	**Difference**	**Total Payment**
<$6,000	0.00	Nil
$6,001-$30,000	0.15	15c for each $1 above $6,000
$30,001-$75,000	0.15	+15c for each $1 above $30,000
$75,001-$150,000	0.10	+10c for each $1 above $75,000
>$150,001	0.05	+5c for each $1 above $150,000

The naive approach to this calculation would be to use a complex series of `if` statements in an `if-else` ladder. Note that we use the `diff` function to calculate the tax rate differences.

```
  tax_rate = [0 0.15 0.30 0.40 0.45];
  tax_rate_diff = diff(tax_rate);
  tax_threshold = [6000 30000 75000 150000];
  income_tax = 0;
  if (income > tax_threshold(1))
    income_tax = (income-tax_threshold(1)) * tax_rate_diff(1);
    if (income > tax_threshold(2))
      income_tax = income_tax + (income-tax_threshold(2)) ...
                   * tax_rate_diff(2);
      if (income > tax_threshold(3))
        income_tax = income_tax + (income-tax_threshold(3)) ...
                     * tax_rate_diff(3);
        if (income > tax_threshold(4))
          income_tax = income_tax + (income-tax_threshold(4)) ...
                       * tax_rate_diff(4);
        end
      end
    end
  end
```

Imagine that we were the tax office and wished to calculate the income tax for all 20 million or so Australians. In this case the variable `income` would be a vector. Given the `if` statements above, the only viable approach would be to put a (very) big `for` loop around the above code. As we know, in MATLAB this will be inefficient. It is also inelegant – for instance it assumes that the number of tax brackets will not change. The better option in MATLAB is to use a combination of logical and numerical operations.

We can replace this complex series of statements with a much shorter piece of code, which we given in the form of a function `.m` file below.

```
function tax = income_tax(taxable_income)
% Calculate income tax.
%
% file:       tax.m, (c) Matthew Roughan, Thu Apr 10 2008
```

```
%
% Income tax rates financial year 2007-2008
%    http://www.ato.gov.au/individuals/content.asp?doc=/content/12333.ht
% Taxable income          Tax on this income
%    <$6,000              Nil
%    $6,001-$30,000       15c for each $1 between $6,000-$30,000
%    $30,001-$75,000      +30c for each $1 between $30,000-$75,000
%    $75,001-$150,000     +40c for each $1 between $75,000-$150,000
%    >$150,001            +45c for each $1 over $150,000
% ignoring medicare levy
%
% Inputs:
%   taxable_income = a vector or matrix of incomes
%
% Outputs:
%   tax            = a vector or matrix the same size as the input
%

tax_rate = [0.0 0.15 0.30 0.40 0.45];
tax_rate_diff = diff(tax_rate);
tax_threshold = [6000 30000 75000 150000];

tax = zeros(size(taxable_income));
for i=1:length(tax_rate_d)
  tax = tax + tax_rate_diff(i) * ...
        (taxable_income - tax_threshold(i)) .* ...
        (taxable_income > tax_threshold(i) );
end
```

You will note that the majority of the code is comments; the actual calculation code is only a couple of lines. There is still a `for` loop, but this is a for loop across the number of tax brackets, not the length of the `income` vector. This adds flexibility in the number of brackets, but more importantly, the calculation is done using vector operations, rather than a loop.

9.1.3 Converting marks to grades

In the part of this course on Excel we coxnsidered converting a series of marks into grades according to the following table:

Mark	Grade	Letter code
< 50	Fail	F
50-64	Pass	P
65-74	Credit	C
75-84	Distinction	D
85-100	High Distinction	H

Here we denote each grade by a single letter code, and we seek to construct a vector of such codes from a vector of marks. The following code does so:

```
mark = 100*rand(100,1);  % generate 100 random marks between 0-100
grade = (mark < 50) * 'F' + ...
        (mark >= 50 & mark < 65) * 'P' + ...
        (mark >= 65 & mark < 75) * 'C' + ...
        (mark >= 75 & mark < 85) * 'D' + ...
        (mark >= 85) * 'H';
grade = char(grade)
```

The code relies on the fact that we can treat characters as numbers, and that the logical vectors being constructed will not overlap (i.e., the intersection of values where more than one is true is empty). The final step converts the numbers back into characters.

9.2 Additional tests

We can test if *all* of the elements of a logical vector are true using the `prod` command as follows:

```
check_all = (prod(some_logical_vector) == 1)
```

The logical variable `check_all` will take the value 1, iff all of the elements of the vector `some_logical_vector` are 1, and it will otherwise be 0. We can test if *at least* one of a logical vector is true using the `sum` function, e.g.,

```
check_at_least = (sum(some_logical_vector) > 0)
```

The variable `check_at_least` will take the value 1 if at least one of the elements of the vector `some_logical_vector` is 1, and 0 otherwise.

9.3 The `find` function

An important function, in the context of logical vectors, is the `find` function. The function returns the indices at which a logical vector is true. For example:

```
r = 0:0.2:1;
z = find(r <= 0.5)
```

will return

```
z =
   1     2     3
```

This indicates that the condition `r <= 0.5` is true for the first terms of the vector `r`. Another simple example is

```
r = 0:0.2:1;
z = find(r == 0.4)
```

which returns

```
z =
     3
```

because only the third element of `r == 0.4` is true.

The `find` function returns an empty vector when there are no true elements. We can test this using the `isempty` function, for example:

```
  r = rand(10,1);
  k = find( r < 0.1 );
  if (~isempty(k))
    disp('the following values are < 0.1');
    disp(r(k));
  else
    disp('no results found');
  end
```

The code finds values of the random vector `r` that are less than 0.1, and prints them out, but if none are found, we write `no results found`.

The `find` command is often useful. For instance, in the example above of plotting the `sinc` function, the `NaN` from our computation of by adding `eps` to the zero term does have a limitation. It assumes that our calculation of $\sin(x)/x$ is still accurate for small values of x. In this case, it does not cause a problem, but for some other functions it might. An alternative is to replace the final values of `y` with the correct value. For instance:

```
  x = -2*pi : pi / 20 : 2*pi;
  y = sin(x) ./ x;
  y(find(isnan(y))) = 1;
  plot(x,y)
```

The `find` function looks for indices of `y` where `y=NaN`, and replaces these values of `y` with 1.

For more information on `find` including how to use it for matrices, use `help find`.

Chapter 10

The Optimization Toolbox

Toolboxes add extra functionality to MATLAB. One extremely useful toolbox is the Optimization Toolbox. We can use this to solve a variety of optimization problems, such as those discussed earlier. However, we don't need it for simple problems. As noted earlier, we can program a bisection search (or other) to find zeros and other function properties such as maxima. The place where the optimization toolbox comes into its own is where we have complex optimizations problems, for instance of the linear programming type. We will consider this case below, but note that the Optimization toolbox has a range of functions: see `help optim` for more details.

10.1 Linear Programming

We now consider a problem where we have more than one variable, a linear objective function, and a series of linear constraints. We call such a problem a **linear program**, and there are good reasons to consider such problems, and good techniques for their solution.

This particular example came out of Ragsdale, page 20. The problems originates from a hypothetical company Blue Ridge Hot Tubs that sells two types of hot-tubs (saunas): Aqua-Spa and Hydro-Lux. The manager of BRHT needs to know how many of each tub to produce.

She wishes to maximise her profit each cycle [each month, say] subject to supply constraints [on labour and inputs]. Each hot tub needs one pump and the supply of pumps is at most 200 per month. The Aqua-Spa tub needs 12 feet of copper tubing in its construction, while the Hydro-Lux uses 16 feet and we can only get 2889 feet of tubing per month. Each Aqua-Spa tub needs 9 hours of labour to construct, while the Hydro-Lux needs 6 hours, but there are only 1566 man hours of workers available per month. [BTW Aqua is water in Latin, Hydro is water in Greek (cute!)]. The profit from the sale of an Aqua-Spa tub is $350 and for the Hydro-Lux tub $300.

We now introduce decision x_1 and x_2 which represent the number of Aqua-Spa tubs and the number of Hydro-Lux tubs made in each cycle. We then wish to maximise the profit

$$P = 350x_1 + 300x_2$$

subject to the constraints

$$\begin{aligned} x_1 + x_2 &\leq 200 \\ 12x_1 + 16x_2 &\leq 2889 \\ 9x_1 + 6x_2 &\leq 1566 \end{aligned}$$

and of course $x_1 \geq 0$ and $x_2 \geq 0$. The first constraints says that the number of pumps is limited to 200, the second refers to the available copper tubing, and the third to the total available man-hours. Actually we should also require that x_1 and x_2 be integers (half a hot-tun isn't worth much), but we will won't try to do this here.

In MATLAB, we need to represent this problem in the form *minimize*

$$\mathbf{f}' * \mathbf{x}$$

subject to the constraints

$$\begin{array}{rcl} A * \mathbf{x} & \leq & \mathbf{b} \\ A_{eq} * \mathbf{x} & = & \mathbf{b}_{eq} \end{array}$$

,

and $\mathbf{l} \leq \mathbf{x} \leq \mathbf{u}$. Here, we use $\mathbf{x}$ as a vector of the variables (in this case x_1 and x_2), and $\mathbf{l}$ and $\mathbf{u}$ are lower and upper bounds on the variables. In general, it is not hard to transform a linear program into this form. Some tricks that are useful are

- If the original problem is a maximization, then we simply multiply the objective function by -1 to obtain a minimization problem.
- the vector $\mathbf{f}$ used in the objective function is just given by the co-efficients in the linear objective function, e.g., in the example

$$\mathbf{f} = -(350, 300)'.$$

- If a constraint has a $\geq$ sign, then we can transform it to have a $\leq$ sign by multiplying by -1.
- If a variable doesn't have an upper bound, e.g., in the problem above, then we can use `Inf` in place of a number in the vector $\mathbf{u}$. Similarly with `-Inf` for lower bounds.
- One or the other of the equality, or inequality constraints may be empty. In the example problem there are no equalities so we set `Aeq = [], beq = [];`.
- The rows of A are made up of the co-efficient of the same row of the constraints, e.g., in our example

$$A = \begin{pmatrix} 1 & 1 \\ 12 & 16 \\ 9 & 6 \end{pmatrix}$$

- the vector $\mathbf{b}$ is made up of the right-hand side of the constraints, in the same order they appear, e.g., in our example

$$\mathbf{b} = \begin{pmatrix} 200 \\ 2889 \\ 1566 \end{pmatrix}$$

One the problem has been constructed, we can find the optimal solution using the `linprog` function, which is part of the Optimization Toolbox. It takes a variable number of arguments, but to solve the above problem, we would call it using `x=linprog(f,A,b,Aeq,beq,l,u)`.

So in the example problem, our MATLAB code to construct and solve the linear programming problem would be

```
f = -[350; 300];
A = [[ 1  1];
     [12 16];
     [ 9  6]];
b = [200; 2889; 1566];
Aeq = [];
beq = [];
l = zeros(2,1);
u = [Inf; Inf];
x = linprog(f,A,b,Aeq,beq,l,u)
```

which outputs

```
Optimization terminated.
x =
  122.0000
   78.0000
```

There are a variety of additional input terms, or output terms that one can use to specify initial solutions, numbers of iterations, objective function values, and so on. See `help linprog` for more details.

There are other optimization routines, both in the Optimization toolbox (e.g. `fgoalattain`) and elsewhere that allow us to mimic most of the possibilities in Excel, and some others as well.

10.2 Why use MATLAB for optimization

MATLAB and Excel both allow us to solve optimization problems. When should we use one, or the other. Often it is a matter of person preference – use whichever you are most comfortable with. For many people this will be Excel, because they consider it easier to enter the optimization problem constraints through a spreadsheet. However, MATLAB does have distinct advantages over Excel.

- MATLAB can solve much larger optimization problems than Excel (problems with more variables and more constraints). The exact numbers will depend on versions, and hardware, but typically we might be able to solve problems 10 times larger in MATLAB with roughly equivalent systems. MATLAB is typically a lot faster as well.

- MATLAB enables the solution of many optimization problems. For instance, it is easy, in MATLAB, to write an optimization inside a `for` loop, and thus execute it (with some differences presumably) each time the loop is executed. For instance, each iteration might load a new set of data from a different file. The outputs of each optimization could then be automatically sent to a file.

- MATLAB facilitates using the outputs of previous programming as the inputs to our optimization. For instance, the constraint matrix A might be constructed using other functions. The same can be done in Excel, but if things like the number of constraints change then this is not so easy to cope with whereas in MATLAB it is trivial.

The features make MATLAB more suitable when optimization is to be done on large problems, or frequently, or as part of an automated system.

Finally, there are even better tools for solving optimization problems. Some can handle much larger problems even than MATLAB. However, one thing most other sophisticated tools have in common is that problems are specified (mathematically) in a similar way to that above, so it isn't going to waste your time to learn MATLAB's approach.

Chapter 11

MATLAB **Roundup**

11.1 More stuff

In addition to the many features which we have mentioned but not described in detail, MATLAB has many other features we have not discussed at all:

- MATLAB has a large array of mathematical functions not mentioned here for many tasks. Toolboxes add to this functionality considerably.
- Ability to build new GUIs for specific tasks;
- MATLAB has built in debugging, lint and profiling tools.
- Lots of technicalities,e.g., *function overloading:* it is possible to have two functions with the same name, where the one that is called depends on the input arguments; *persistent variables:* variables with local scope that can keep a value in between function calls; error handling; and so on.
- An interface to `C`, so that C-code can be used directly from MATLAB(see MEX files). Also, MATLAB has interfaces to the system in which it runs (i.e., it can call other programs).
- There are alternative data structures such as *struct* which allows a more object oriented approach in MATLAB, and a *cell array* which allows us to construct arrays of any other datatype, e.g. strings.
- There are many toolboxes which come complete with additional function and features including, for instance, the parallel processing toolbox, and symbolic manipulation toolbox.

11.2 Limitations

MATLAB does have limitations:

- It is much faster than Excel, but still not the fastest possible way to program. Where multiply nested loops are needed, there are faster ways of programming. If vector/matrix operations are used, it can be quite fast, but still better performance may be achieved through carefully optimized low-level code.
- MATLAB is not particularly efficient in its (default) storage of variables. The standard is to always used double precision floating point, which is good much of the time, but may be overkill at 64 bits per variable.
- MATLAB's string handling is not the easiest to program – there are better tools such as Perl.
- Lack of associative arrays (hashes) is sometimes annoying (to me).
- MATLAB will have trouble if you need to access low-level machine dependent components, such as registers.

11.3 Summary

MATLAB also shows us the same set of concepts that we first saw in Excel, i.e.:

- *variables:* a value that we can change or control. In MATLAB, variables can have almost arbitrary names that we choose. MATLAB variables are different from Excel variables in that they have a `type`, which we can access (via `whos`) or change, to effect the amount of memory stored, or the accuracy of floating point numbers. However, MATLAB cannot store a formula into a variable (though a reference to a function can be).
- *vectors, and matrices:* are a standard data structure in most computing environments, and of great use in maths, in particular when we have a group of numbers. MATLAB allows us to store vectors and matrices into variables.
- *functions:* are a useful way of representing something (usually mathematical) that we want to do with some data.
- *reference:* is how we include other data into our functions. In MATLAB, we make a reference to a variable explicitly when calling a function, or using a variable's value in an expression.
- *graphing:* can be used to visualise data in various ways.
- *mapping:* values from say a number to a grade can be easily performed using conditional statements (e.g. `if`).
- *sorting:* data into order, either numerically, or alphabetically.
- *filtering:* to see only values that match a particular criteria. In MATLAB we do this using 0-1 vectors.
- *optimisation:* where we seek to maximise or minimise some quantity (e.g. profit, or revenue).

- *constraints:* limits on our variables.
- *iteration:*, i.e., repeating some set of steps. MATLAB provides explicit constructs for iteration (e.g. `for` and `while` loops).
- *conditionals:* in MATLAB are expressed via an `if` statement allow us to execute code conditional on variables.

In MATLAB these concepts are often more explicit in that they are supported by specific language constructs, rather than implicitly, or by function calls as they are in Excel.

There are additional concepts we have seen here, such as

- *vectorization*: writing formula such that they can be applied to vectors can speed up calculation not just in MATLAB, but also in specific graphics applications, and other areas.
- *modularity*: breaking code into meaningful modules (in MATLAB we use functions to do this).
- *scope*: variables defined inside a MATLAB function cannot be seen outside of this function unless they are explicitly given global scope. This avoids name collisions.
- *recursion*: a function may call itself, and we call this type of function a recursive function.
- *performance*: most tasks can be implemented in many different ways. Part of the job of a programmer is implementing the program efficiently.
- *style*: correctness is not the only task when writing code. We also need to write readable/maintainable code, and coding style can help this dramatically.