An Introduction to Matlab

Version 2.3

David F. Griffiths

Department of Mathematics

The University
Dundee DD1 4HN

With additional material by Ulf Carlsson
Department of Vehicle Engineering
KTH, Stockholm, Sweden

Contents

1   MATLAB

- Matlab is an interactive system for doing numerical computations.

- A numerical analyst called Cleve Moler wrote the first version of Matlab in the 1970s. It has since evolved into a successful commercial software package.

- Matlab relieves you of a lot of the mundane tasks associated with solving problems numerically. This allows you to spend more time thinking, and encourages you to experiment.

- Matlab makes use of highly respected algorithms and hence you can be confident about your results.

- Powerful operations can be performed using just one or two commands.

- You can build up your own set of functions for a particular application.

- Excellent graphics facilities are available, and the pictures can be inserted into LaTeX and Word documents.

These notes provide only a brief glimpse of the power and flexibility of the Matlab system. For a more comprehensive view we recommend the book

Matlab Guide
D.J. Higham & N.J. Higham
SIAM Philadelphia, 2000, ISBN: 0-89871-469-9.

2   Starting Up

2.1   Windows Systems

On Windows systems MATLAB is started by double-clicking the MATLAB icon on the desktop or by selecting MATLAB from the start menu.
The starting procedure takes the user to the Command window where the Command line is indicated with '>>'. Used in the calculator mode all Matlab commands are entered to the command line from the keyboard.
Matlab can be used in a number of different ways or modes; as an advanced calculator in the calculator mode, in a high level programming language mode and as a subroutine called from a C-program. More information on the first two of these modes is given below.
Help and information on Matlab commands can be found in several ways,

- from the command line by using the 'help topic' command (see below),

- from the separate Help window found under the Help menu or

- from the Matlab helpdesk stored on disk or on a CD-ROM.

Another useful facility is to use the 'lookfor keyword' command, which searches the help files for the keyword. See Exercise 16.1 (page 17) for an example of its use.

2.2   Unix Systems

- You should have a directory reserved for saving files associated with Matlab. Create such a directory (`mkdir`) if you do not have one. Change into this directory (`cd`).

- Start up a new `xterm` window (do `xterm &` in the existing `xterm` window).

- Launch Matlab in one of the `xterm` windows with the command

    ```
    matlab
    ```

  After a short pause, the logo will be shown followed by a window containing the Matlab interface. Should you wish to run Matlab in an `xterm` window, use the command

    ```
    matlab -nojvm
    ```

  and, following dislpay of the logo, the Matlab prompt `>>` will appear.

  Type `quit` at any time **to exit from Matlab**.

2.3   Command Line Help

Help is available from the command line prompt. Type `help help` for "help" (which gives a brief synopsis of the help system), `help` for a list of topics. The first few lines of this read

```
HELP topics:

matlab/general -  General purpose commands.
matlab/ops     -  Operators and special char...
matlab/lang    -  Programming language const...
matlab/elmat   -  Elementary matrices and ma...
matlab/elfun   -  Elementary math functions.
matlab/specfun -  Specialized math functions.
```

(truncated lines are shown with . . .). Then to obtain help on "Elementary math functions", for instance, type

```
>> help elfun
```

This gives rather a lot of information so, in order to see the information one screenful at a time, first issue the command `more on`, i.e.,

```
>> more on
>> help elfun
```

Hit any key to progress to the next page of information.

2.4 Demos

Demonstrations are invaluable since they give an indication of Matlabs capabilities. A comprehensive set are available by typing the command

```
>> demo
```

(Warning: this will clear the values of all current variables.)

3 Matlab as a Calculator

The basic arithmetic operators are `+ - * / ^` and these are used in conjunction with brackets: `( )`. The symbol `^` is used to get exponents (powers): `2^4=16`.
You should type in commands shown following the prompt: >>.

```
>> 2 + 3/4*5
ans =
    5.7500
>>
```

Is this calculation `2 + 3/(4*5)` or `2 + (3/4)*5`? Matlab works according to the priorities:

1. quantities in brackets,
2. powers `2 + 3^2` ⇒`2 + 9 = 11`,
3. `* /`, working left to right (`3*4/5=12/5`),
4. `+ -`, working left to right (`3+4-5=7-5`),

Thus, the earlier calculation was for `2 + (3/4)*5` by priority 3.

4 Numbers & Formats

Matlab recognizes several different kinds of numbers

Type	Examples
Integer	$1362, -217897$
Real	$1.234, -10.76$
Complex	$3.21 - 4.3i \quad (i = \sqrt{-1})$
Inf	Infinity (result of dividing by 0)
NaN	**N**ot **a N**umber, 0/0

The "`e`" notation is used for very large or very small numbers:
`-1.3412e+03` $= -1.3412 \times 10^3 = -1341.2$
`-1.3412e-01` $= -1.3412 \times 10^{-1} = -0.13412$
All computations in MATLAB are done in double precision, which means about 15 significant figures. The

Command	Example of Output
`>>format short`	31.4162(4–decimal places)
`>>format short e`	3.1416e+01
`>>format long e`	3.141592653589793e+01
`>>format short`	31.4162(4–decimal places)
`>>format bank`	31.42(2–decimal places)

format—how Matlab prints numbers—is controlled by the "format" command. Type `help format` for full list. Should you wish to switch back to the default format then `format` will suffice.
The command

```
format compact
```

is also useful in that it suppresses blank lines in the output thus allowing more information to be displayed.

5 Variables

```
>> 3-2^4
ans =
    -13
>> ans*5
ans =
    -65
```

The result of the first calculation is labelled "`ans`" by Matlab and is used in the second calculation where its value is changed.
We can use our own names to store numbers:

```
>> x = 3-2^4
x =
    -13
>> y = x*5
y =
    -65
```

so that `x` has the value -13 and $y = -65$. These can be used in subsequent calculations. These are examples of **assignment statements**: values are assigned to variables. Each variable must be assigned a value before it may be used on the right of an assignment statement.

5.1 Variable Names

Legal names consist of any combination of letters and digits, starting with a letter. These are allowable:

```
NetCost, Left2Pay, x3, X3, z25c5
```

These are **not** allowable:

```
Net-Cost, 2pay, %x, @sign
```

Use names that reflect the values they represent.
Special names: you should avoid using
`eps = 2.2204e-16` $= 2^{-54}$ (The largest number such that `1 + eps` is indistinguishable from 1) and
`pi = 3.14159... =` π.
If you wish to do arithmetic with complex numbers,both `i` and `j` have the value $\sqrt{-1}$ unless you change them

```
>> i,j, i=3
ans = 0 + 1.0000i
ans = 0 + 1.0000i
i   = 3
```

6   Suppressing output

One often does not want to see the result of intermediate calculations—terminate the assignment statement or expression with semi–colon

```
>> x=-13; y = 5*x, z = x^2+y
y =
   -65
z =
   104
>>
```

the value of x is hidden. Note also we can place several statements on one line, separated by commas or semi–colons.

Exercise 6.1 *In each case find the value of the expression in Matlab and explain precisely the order in which the calculation was performed.*

i)	-2^3+9	ii)	2/3*3
iii)	3*2/3	iv)	3*4-5^2*2-3
v)	(2/3^2*5)*(3-4^3)^2	vi)	3*(3*4-2*5^2-3)

7   Built–In Functions

7.1   Trigonometric Functions

Those known to Matlab are
sin, cos, tan
and their arguments should be in radians.
e.g. to work out the coordinates of a point on a circle of radius 5 centred at the origin and having an elevation $30^o = \pi/6$ radians:

```
>> x = 5*cos(pi/6), y = 5*sin(pi/6)
x =
    4.3301
y =
    2.5000
```

The inverse trig functions are called asin, acos, atan (as opposed to the usual arcsin or $\sin^{-1}$ etc.). The result is in radians.

```
>> acos(x/5), asin(y/5)
ans = 0.5236
ans = 0.5236
>> pi/6
ans = 0.5236
```

7.2   Other Elementary Functions

These include sqrt, exp, log, log10

```
>> x = 9;
>> sqrt(x),exp(x),log(sqrt(x)),log10(x^2+6)
ans =
     3
ans =
   8.1031e+03
ans =
    1.0986
ans =
    1.9395
```

exp(x) denotes the exponential function $\exp(x) = e^x$ and the inverse function is log:

```
>> format long e, exp(log(9)), log(exp(9))
ans = 9.000000000000002e+00
ans = 9
>> format short
```

and we see a tiny rounding error in the first calculation. log10 gives logs to the base 10. A more complete list of elementary functions is given in Table 2 on page 32.

8   Vectors

These come in two flavours and we shall first describe **row vectors**: they are lists of numbers separated by either commas or spaces. The number of entries is known as the "length" of the vector and the entries are often referred to as "elements" or "components" of the vector.The entries must be enclosed in square brackets.

```
>> v = [ 1 3, sqrt(5)]
v =
    1.0000    3.0000    2.2361
>> length(v)
ans =
     3
```

Spaces can be vitally important:

```
>> v2 = [3+ 4 5]
v2 =
     7     5
>> v3 = [3 +4 5]
v3 =
     3     4     5
```

We can do certain arithmetic operations with vectors of the same length, such as v and v3 in the previous section.

```
>> v + v3
ans =
    4.0000    7.0000    7.2361
>> v4 = 3*v
v4 =
    3.0000    9.0000    6.7082
>> v5 = 2*v -3*v3
v5 =
   -7.0000   -6.0000  -10.5279
>> v + v2
??? Error using ==> +
Matrix dimensions must agree.
```

4

i.e. the error is due to v and v2 having different lengths. A vector may be multiplied by a scalar (a number—see v4 above), or added/subtracted to another vector of the **same** length. The operations are carried out elementwise.

We can build row vectors from existing ones:

```
>> w  = [1 2 3],  z = [8 9]
>> cd = [2*z,-w], sort(cd)
w =
     1     2     3
z =
     8     9
cd =
    16    18    -1    -2    -3
ans =
    -3    -2    -1    16    18
```

Notice the last command **sort**'ed the elements of **cd** into ascending order.

We can also change or look at the value of particular entries

```
>> w(2) = -2, w(3)
w =
     1    -2     3
ans =
     3
```

8.1   The Colon Notation

This is a shortcut for producing row vectors:

```
>>  1:4
ans =
     1     2     3     4
>> 3:7
ans =
     3     4     5     6     7
>> 1:-1
ans =
     []
```

More generally $a : b : c$ produces a vector of entries starting with the value a, incrementing by the value b until it gets to c (it will not produce a value beyond c). This is why 1:-1 produced the empty vector [].

```
>> 0.32:0.1:0.6
ans =
    0.3200    0.4200    0.5200
>> -1.4:-0.3:-2
ans =
   -1.4000   -1.7000   -2.0000
```

8.2   Extracting Bits of a Vector

```
>>  r5 = [1:2:6, -1:-2:-7]
r5 =
     1     3     5    -1    -3    -5    -7
```

To get the 3rd to 6th entries:

```
>> r5(3:6)
ans =
     5    -1    -3    -5
```

To get alternate entries:

```
>> r5(1:2:7)
ans =
     1     5    -3    -7
```

What does r5(6:-2:1) give?
See **help colon** for a fuller description.

8.3   Column Vectors

These have similar constructs to row vectors. When defining them, entries are separated by ; or "newlines"

```
>> c = [ 1; 3; sqrt(5)]
c =
    1.0000
    3.0000
    2.2361
>> c2 = [3
4
5]
c2 =
     3
     4
     5
>> c3 = 2*c - 3*c2
c3 =
   -7.0000
   -6.0000
  -10.5279
```

so column vectors may be added or subtracted **provided that they have the same length**.

8.4   Transposing

We can convert a row vector into a column vector (and vice versa) by a process called *transposing*—denoted by '.

```
>> w, w', c, c'
w =
     1    -2     3
ans =
     1
    -2
     3
c =
    1.0000
    3.0000
    2.2361
ans =
    1.0000    3.0000    2.2361
>> t = w + 2*c'
t =
    3.0000    4.0000    7.4721
>> T = 5*w'-2*c
T =
    3.0000
```

```
   -16.0000
    10.5279
```

If `x` is a *complex vector*, then `x'` gives the *complex conjugate transpose* of `x`:

```
>> x = [1+3i, 2-2i]
ans =
   1.0000 + 3.0000i   2.0000 - 2.0000i
>> x'
ans =
   1.0000 - 3.0000i
   2.0000 + 2.0000i
```

Note that the components of `x` were defined without a `*` operator; this means of defining complex numbers works even when the variable `i` already has a numeric value. To obtain the plain transpose of a complex number use `.'` as in

```
>> x.'
ans =
   1.0000 + 3.0000i
   2.0000 - 2.0000i
```

9 Keeping a record

Issuing the command

```
>> diary mysession
```

will cause all subsequent text that appears on the screen to be saved to the file `mysession` located in the directory in which Matlab was invoked. You may use any legal filename *except* the names `on` and `off`. The record may be terminated by

```
>> diary off
```

The file `mysession` may be edited with your favourite editor (the Matlab editor, emacs, or even Word) to remove any mistakes.
If you wish to quit Matlab midway through a calculation so as to continue at a later stage:

```
>> save thissession
```

will save the current values of all variables to a file called `thissession.mat`. **This file cannot be edited**. When you next startup Matlab, type

```
>> load thissession
```

and the computation can be resumed where you left off. A list of variables used in the current session may be seen with

```
>> whos
```

See `help whos` and `help save`.

```
>> whos
Name  Size Elements  Bytes  Density Complex
ans  1 by 1  1         8       Full     No
  v  1 by 3  3        24       Full     No
 v1  1 by 2  2        16       Full     No
```

Name	Size	Elements	Bytes	Density	Complex
v2	1 by 2	2	16	Full	No
v3	1 by 3	3	24	Full	No
v4	1 by 3	3	24	Full	No
x	1 by 1	1	8	Full	No
y	1 by 1	1	8	Full	No

```
Grand total is 16 elements using 128 bytes
```

10 Plotting Elementary Functions

Suppose we wish to plot a graph of $y = \sin 3\pi x$ for $0 \le x \le 1$. We do this by sampling the function at a sufficiently large number of points and then joining up the points (x, y) by straight lines. Suppose we take $N + 1$ points equally spaced a distance h apart:

```
>> N = 10; h = 1/N; x = 0:h:1;
```

defines the set of points $x = 0, h, 2h, \ldots, 1 - h, 1$. Alternately, we may use the command `linspace`: The general form of the command is `linspace (a,b,n)` which generates $n + 1$ equispaced points between a and b, inclusive. So, in this case we would use the command

```
>> x = linspace (0,1,11);
```

The corresponding y values are computed by

```
>> y = sin(3*pi*x);
```

and finally, we can plot the points with

```
>> plot(x,y)
```

The result is shown in Figure 1, where it is clear that the value of N is too small.
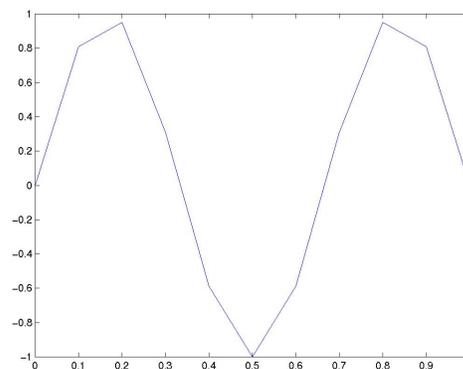

Figure 1: Graph of $y = \sin 3\pi x$ for $0 \le x \le 1$ using $h = 0.1$.

On changing the value of N to 100:

```
>> N = 100; h = 1/N; x = 0:h:1;
>> y = sin(3*pi*x);  plot(x,y)
```

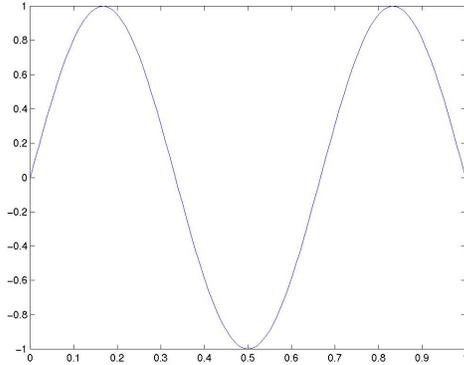we get the picture shown in Figure 2.

Figure 2: Graph of $y = \sin 3\pi x$ for $0 \le x \le 1$ using $h = 0.01$.

10.1 Plotting—Titles & Labels

To put a title and label the axes, we use

```
>> title('Graph of y = sin(3pi x)')
>> xlabel('x axis')
>> ylabel('y-axis')
```

The *strings* enclosed in single quotes, can be anything of our choosing. Some simple LaTeX commands are available for formatting mathematical expressions and Greek characters—see Section 10.9.
See also `ezplot` the "Easy to use function plotter".

10.2 Grids

A dotted grid may be added by

```
>> grid
```

This can be removed using either `grid` again, or `grid off`.

10.3 Line Styles & Colours

The default is to plot solid lines. A solid white line is produced by

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

The third argument is a string whose first character specifies the colour(optional) and the second the line style. The options for colours and styles are:

Colours		Line Styles	
y	yellow	.	point
m	magenta	o	circle
c	cyan	x	x-mark
r	red	+	plus
g	green	–	solid
b	blue	*	star
w	white	:	dotted
k	black	-.	dashdot
		--	dashed

The number of available plot symbols is wider than shown in this table. Use `help plot` to obtain a full list. See also `help shapes`.

10.4 Multi–plots

Several graphs may be drawn on the same figure as in

```
>> plot(x,y,'w-',x,cos(3*pi*x),'g--')
```

A descriptive legend may be included with

```
>> legend('Sin curve','Cos curve')
```

which will give a list of line–styles, as they appeared in the plot command, followed by a brief description. Matlab fits the legend in a suitable position, so as not to conceal the graphs whenever possible.
For further information do `help plot` etc.
The result of the commands

```
>> plot(x,y,'w-',x,cos(3*pi*x),'g--')
>> legend('Sin curve','Cos curve')
>> title('Multi-plot ')
>> xlabel('x axis'),  ylabel('y axis')
>> grid
```

is shown in Figure 3. The legend may be moved manually by dragging it with the mouse.
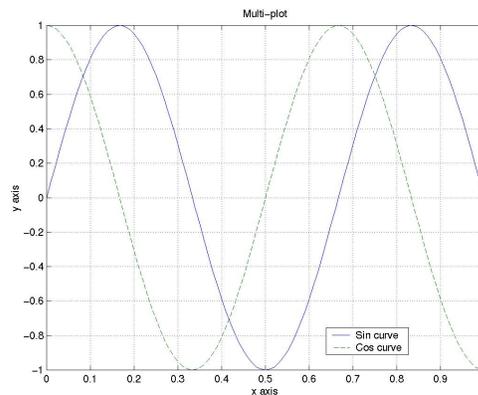
Figure 3: Graph of $y = \sin 3\pi x$ and $y = \cos 3\pi x$ for $0 \le x \le 1$ using $h = 0.01$.

10.5 Hold

A call to `plot` clears the graphics window before plotting the current graph. This is not convenient if we wish to add further graphics to the figure at some later stage. To stop the window being cleared:

```
>> plot(x,y,'w-'), hold on
>> plot(x,y,'gx'), hold off
```

"`hold on`" holds the current picture; "`hold off`" releases it (but does not clear the window, which can be done with `clf`). "`hold`" on its own toggles the hold state.

10.6 Hard Copy

To obtain a printed copy select $\boxed{\text{Print}}$ from the $\boxed{\text{File}}$ menu on the Figure toolbar.

Alternatively one can save a figure to a file for later printing (or editing). A number of formats is available (use `help print` to obtain a list). To save a file in "Encapsulated PostScript" format, issue the Matlab command

```
print -deps fig1
```

which will save a copy of the image in a file called `fig1.eps`.

10.7 Subplot

The graphics window may be split into an $m \times n$ array of smaller windows into which we may plot one or more graphs. The windows are counted 1 to mn row–wise, starting from the top left. Both `hold` and `grid` work on the current subplot.

```
>> subplot(221), plot(x,y)
>>   xlabel('x'),ylabel('sin 3 pi x')
>> subplot(222), plot(x,cos(3*pi*x))
>>   xlabel('x'),ylabel('cos 3 pi x')
>> subplot(223), plot(x,sin(6*pi*x))
>>   xlabel('x'),ylabel('sin 6 pi x')
>> subplot(224), plot(x,cos(6*pi*x))
>>   xlabel('x'),ylabel('cos 6 pi x')
```

`subplot(221)` (or `subplot(2,2,1)`) specifies that the window should be split into a 2×2 array and we select the first subwindow.

10.8 Zooming

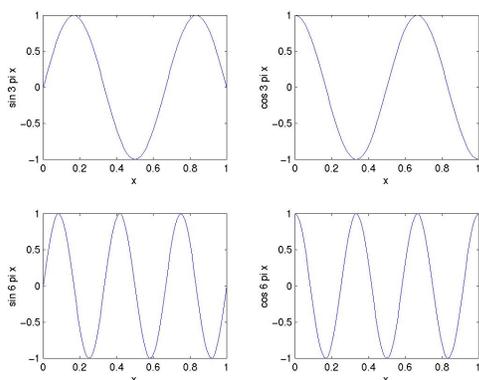We often need to "zoom in" on some portion of a plot in order to see more detail. Clicking on the "Zoom in" or "Zoom out" button on the Figure window is simplest but one can also use the command

```
>> zoom
```

Pointing the mouse to the relevant position on the plot and clicking the left mouse button will zoom in by a factor of two. This may be repeated to any desired level.

Clicking the right mouse button will zoom out by a factor of two.

Holding down the left mouse button and dragging the mouse will cause a rectangle to be outlined. Releasing the button causes the contents of the rectangle to fill the window.

`zoom off` turns off the zoom capability.

Exercise 10.1 *Draw graphs of the functions*

$$
\begin{aligned}
y &= \cos x \\
y &= x
\end{aligned}
$$

for $0 \le x \le 2$ on the same window. Use the zoom facility to determine the point of intersection of the two curves (and, hence, the root of $x = \cos x$) to two significant figures.

The command `clf` clears the current figure while `close 1` will close the window labelled "Figure 1". To open a new figure window type `figure` or, to get a window labelled "Figure 9", for instance, type `figure (9)`. If "Figure 9" already exists, this command will bring this window to the foreground and the result subsequent plotting commands will be drawn on it.

10.9 Formatted text on Plots

It is possible to change to format of text on plots so as to increase or decrease its size and also to typeset simple mathematical expressions (in LaTeX form).

We shall give two illustrations.

First we plot the first 100 terms in the sequence $\{x_n\}$ given by $x_n = \left(1 + \frac{1}{n}\right)^n$ and then graph the function $\phi(x) = x^3 \sin^2(3\pi x)$ on the interval $-1 \le x \le 1$. The commands

```
>> set(0,'Defaultaxesfontsize',16);
>> n = 1:100;   x = (1+1./n).^n;
>> subplot (211)
>> plot(n,x,'.',[0 max(n)],exp(1)*[1 1],...
       '--','markersize',8)
>> title('x_n = (1+1/n)^n','fontsize',12)
>> xlabel('n'), ylabel('x_n')
>> legend('x_n','y = e^1 = 2.71828...',4)
>> %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
>> subplot (212)
>> x = -2:.02:2; y = x.^3.*sin(3*pi*x).^2;
>> plot(x,y,'linewidth',2)
>> legend('y = x^3sin^2 3\pi x',4)
>> xlabel('x')
```

produce the graph shown below. The salient features of these commands are

1. The first line increases the size of the default font size used for the axis labels, legends and titles.

2. The size of the plot symbol "." is changed from the default (6) to size 8 by the additional string followed by value "`'markersize',8`".

3. The strings `x_n` are formatted as x_n to give subscripts while `x^3` leads to superscripts x^3.

   Note also that $sin^2 3\pi x$ translates into the Matlab command `sin(3*pi*x).^2`—the position of the exponent is different.

4. Greek characters $\alpha, \beta, \ldots, \omega, \Omega$ are produced by the strings `'\alpha'`, `'\beta'`, ...,`'\omega'`, `'\Omega'`. the integral symbol: $\int$ is produced by `'\int'`.

5. The thickness of the line used in the lower graph is changed from its default value (0.5) to 2.

6. Use `help legend` to determine the meaning of the last argument in the `legend` commands.

One can determine the current value of any plot property by first obtaining its "handle number" and then using the `get` command such as

```
>> handle = plot (x,y,'.')
>> get (handle,'markersize')
ans =
     6
```

Experiment also with `set (handle)` (which will list possible values for each property) and
`set(handle,'markersize',12)`
which will increase the size of the marker (a dot in this case) to 12. Also, all plot properties can be edited from the Figure window by selecting the Tools menu from the toolbar. For instance, to change the `linewidth` of a graph, first select the curve by double clicking (it should then change its appearance) and then select Line Properties... from the Tools . This will pop up a dialogue window from which the width, colour, style,... of the curve may be changed.
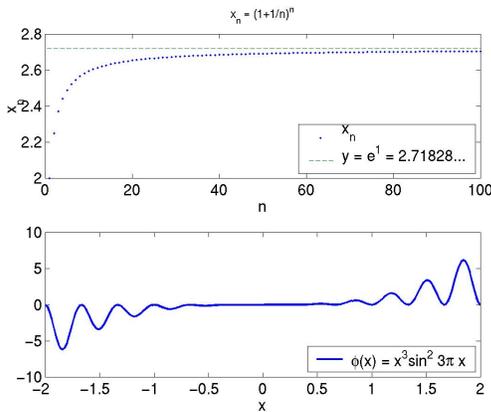
10.10   Controlling Axes

Once a plot has been created in the graphics window you may wish to change the range of x and y values shown on the picture.

```
>> clf, N = 100; h = 1/N; x = 0:h:1;
>> y = sin(3*pi*x); plot(x,y)
>> axis([-0.5 1.5 -1.2 1.2]),  grid
```

The `axis` command has four parameters, the first two are the minimum and maximum values of x to use on the axis and the last two are the minimum and maximum values of y. Note the square brackets. The result of these commands is shown in Figure 4. Look at `help axis` and experiment with the commands `axis equal`, `axis verb`, `axis square`, `axis normal`, `axis tight` in any order.
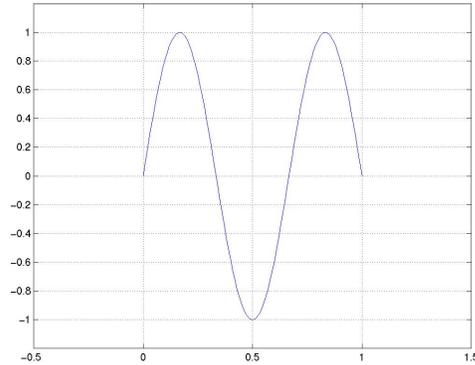
Figure 4: The effect of changing the axes of a plot.

11   Keyboard Accelerators

One can recall previous Matlab commands by using the ↑ and ↓ cursor keys. Repeatedly pressing ↑ will review the previous commands (most recent first) and, if you want to re-execute the command, simply press the return key.

To recall the most recent command starting with `p`, say, type `p` at the prompt followed by ↑. Similarly, typing `pr` followed by ↑ will recall the most recent command starting with `pr`.

Once a command has been recalled, it may be edited (changed). You can use ← and → to move backwards and forwards through the line, characters may be inserted by typing at the current cursor position or deleted using the `Del` key. This is most commonly used when long command lines have been mistyped or when you want to re–execute a command that is very similar to one used previously.

The following emacs–like commands may also be used:

`cntrl a`	move to start of line
`cntrl e`	move to end of line
`cntrl f`	move forwards one character
`cntrl b`	move backwards one character
`cntrl d`	delete character under the cursor

Once you have the command in the required form, press return.

Exercise 11.1 *Type in the commands*

```
>> x = -1:0.1:1;
>> plot(x,sin(pi*x),'w-')
>> hold on
>> plot(x,cos(pi*x),'r-')
```

Now use the cursor keys with suitable editing to execute:

```
>> x = -1:0.05:1;
>> plot(x,sin(2*pi*x),'w-')
>> plot(x,cos(2*pi*x),'r-.'), hold off
```

12 Copying to and from Word and other applications

There are many situations where one wants to copy the output resulting from a Matlab command (or commands) into a Windows application such as Word or into a Unix file editor such as "emacs" or "vi".

12.1 Window Systems

Copying material is made possible on the Windows operating system by using the Windows clipboard.

Also, pictures can be exported to files in a number of alternative formats such as encapsulated postscript format or in jpeg format. Matlab is so frequently used as an analysis tool that many manufacturers of measurement systems and software find it convenient to provide interfaces to Matlab which make it possible, for instance, to import measured data directly into a `*.mat` Matlab file (see `load` and `save` in Section 9).

Example 12.1 *Copying a figure into Word.*

Diagrams prepared in Matlab are easily exported to other Windows applications such as Word. Suppose a plot of the functions $\sin(2\pi ft)$ and $\sin(2\pi ft + \pi/4)$, with $f = 100$, is needed in a report written in Word. We create a time vector, t, with 500 points distributed over 5 periods and then evaluate and plot the two function vectors.

```
>> t = [1:1:500]/500/20;
>> f = 100;
>> y1 = sin(2*pi*f*t);
>> y2 = sin(2*pi*f*t+pi/4);
>> plot(t,y1,'-',t,y2,'--');
>> axis([0 0.05 -1.5 1.5]);
>> grid
```

In order to copy the plot into a Word document

- Select "Copy Figure" under the Edit menu on the figure windows toolbar.

- Switch to the Word application if it is already running, otherwise open a Word document.

- Place the cursor in the desired position in the document and select "Paste" under the "Edit" menu in the Word tool bar.

12.2 Unix Systems

In order to carry out the following exercise, you should have Matlab running in one window and either Emacs or Vi running in another.

To copy material from one window to another, (here L means click Left Mouse Button, etc)

First select the material to copy by L on the start of the material you want and then either dragging the mouse (with the buttom down) to highlight the text, or R at the end of the material. Next move the mouse into the other window and L at the location you want the text to appear. Finally, click the M .

When copying from another application into Matlab you can only copy material to the prompt line. On Unix systems figures are normally saved in files (see Section 10.6) which are then imported into other documents.

13 Script Files

Script files are normal ASCII (text) files that contain Matlab commands. It is essential that such files have names having an extension `.m` (e.g., `Exercise4.m`) and, for this reason, they are commonly known as *m-files*. The commands in this file may then be executed using
```
>> Exercise4
```
Note: the command does not include the file name extension `.m`.

It is only the output from the commands (and not the commands themselves) that are displayed on the screen. Script files are created with your favourite editor under Unix while, under Windows, click on the "New Document" icon at the top left of the main Matlab window to pop up a new window showing the "M-file Editor". Type in your commands and then save (to a file with a `.m` extension).

To see the commands in the command window prior to their execution:
```
>> echo on
```
and `echo off` will turn echoing off.

Any text that follows `%` on a line is ignored. The main purpose of this facility is to enable comments to be included in the file to describe its purpose.

To see what m-files you have in your current directory, use
```
>> what
```

Exercise 13.1   *1. Type in the commands from §10.7 into a file called* `exsub.m`.

   2. Use `what` *to check that the file is in the correct area.*

   3. Use the command `type exsub` *to see the contents of the file.*

   4. Execute these commands.

See §21 for the related topic of function files.

14 Products, Division & Powers of Vectors

14.1 Scalar Product (∗)

We shall describe two ways in which a meaning may be attributed to the product of two vectors. In both cases the vectors concerned must have the same length.

The first product is the standard scalar product. Suppose that $\underline{u}$ and $\underline{v}$ are two vectors of length n, $\underline{u}$ being a **row** vector and $\underline{v}$ a **column** vector:

$$\underline{u} = [u_1, u_2, \ldots, u_n], \qquad \underline{v} = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix}.$$

The scalar product is defined by multiplying the corresponding elements together and adding the results to give a single number (scalar).

$$\underline{u}\,\underline{v} = \sum_{i=1}^{n} u_i v_i.$$

For example, if $\underline{u} = [10, -11, 12]$, and $\underline{v} = \begin{bmatrix} 20 \\ -21 \\ -22 \end{bmatrix}$

then $n = 3$ and

$$\underline{u}\,\underline{v} = 10 \times 20 + (-11) \times (-21) + 12 \times (-22) = 167.$$

We can perform this product in Matlab by

```
>> u = [ 10, -11, 12], v = [20; -21; -22]
>> prod = u*v      % row times column vector
```

Suppose we also define a row vector `w` and a column vector `z` by

```
>> w = [2, 1, 3], z = [7; 6; 5]
w =
    2    1    3
z =
    7
    6
    5
```

and we wish to form the scalar products of `u` with `w` and `v` with `z`.

```
>> u*w
??? Error using ==> *
Inner matrix dimensions must agree.
```

an error results because `w` is not a column vector. Recall from page 5 that transposing (with ') turns column vectors into row vectors and vice versa.

So, to form the scalar product of two row vectors or two column vectors,

```
>> u*w'      % u & w are row vectors
ans =
    45
>> u*u'      % u is a row vector
```

```
ans =
    365
>> v'*z      % v & z are column vectors
ans =
    -96
```

We shall refer to the Euclidean length of a vector as the **norm** of a vector; it is denoted by the symbol $\|\underline{u}\|$ and defined by

$$\|\underline{u}\| = \sqrt{\sum_{i=1}^{n} |u_i|^2},$$

where n is its dimension. This can be computed in Matlab in one of two ways:

```
>> [ sqrt(u*u'), norm(u)]
ans =
    19.1050    19.1050
```

where `norm` is a built–in Matlab function that accepts a vector as input and delivers a scalar as output. It can also be used to compute other norms: `help norm`.

Exercise 14.1 *The angle, θ, between two column vectors $\underline{x}$ and $\underline{y}$ is defined by*

$$\cos\theta = \frac{\underline{x}'\underline{y}}{\|\underline{x}\|\,\|\underline{y}\|}.$$

Use this formula to determine the cosine of the angle between

$$\underline{x} = [1, 2, 3]' \quad and \quad \underline{y} = [3, 2, 1]'.$$

Hence find the angle in degrees.

14.2 Dot Product (.∗)

The second way of forming the product of two vectors of the same length is known as the Hadamard product. It is not often used in Mathematics but is an invaluable Matlab feature. It involves vectors of the same type. If $\underline{u}$ and $\underline{v}$ are two vectors of the same type (both row vectors or both column vectors), the mathematical definition of this product, which we shall call the **dot product**, is the **vector** having the components

$$\underline{u} \cdot \underline{v} = [u_1 v_1, u_2 v_2, \ldots, u_n v_n].$$

The result is a vector of the same length and type as $\underline{u}$ and $\underline{v}$. Thus, we simply multiply the corresponding elements of two vectors.

In Matlab, the product is computed with the operator `.*` and, using the vectors `u, v, w, z` defined on page 11,

```
>> u.*w
ans =
    20 -11 36
>> u.*v'
ans =
    200 231 -264
>> v.*z, u'.*v
ans =
    140 -126 -110
ans =
    200 231 -264
```

Example 14.1 *Tabulate the function* $y = x \sin \pi x$ *for* $x = 0, 0.25, \ldots, 1$.

It is easier to deal with column vectors so we first define a vector of x-values: (see Transposing: §8.4)
```
>> x = (0:0.25:1)';
```
To evaluate y we have to multiply each element of the vector x by the corresponding element of the vector $\sin \pi x$:

x	$\times$	$\sin \pi x$	$=$	$x \sin \pi x$
0	$\times$	0	$=$	0
0.2500	$\times$	0.7071	$=$	0.1768
0.5000	$\times$	1.0000	$=$	0.5000
0.7500	$\times$	0.7071	$=$	0.5303
1.0000	$\times$	0.0000	$=$	0.0000

To carry this out in Matlab:
```
>>  y = x.*sin(pi*x)
y =
          0
     0.1768
     0.5000
     0.5303
     0.0000
```

Note: a) the use of `pi`, b) `x` and `sin(pi*x)` are both column vectors (the `sin` function is applied to each element of the vector). Thus, the dot product of these is also a column vector.

14.3   Dot Division of Arrays (./)

There is no mathematical definition for the division of one vector by another. However, in Matlab, the operator `./` is defined to give element by element division—it is therefore only defined for vectors of the same size and type.

```
>> a = 1:5, b = 6:10, a./b
a =
     1     2     3     4     5
b =
     6     7     8     9     10
ans =
 0.1667  0.2857  0.3750  0.4444  0.5000
>> a./a
ans =
     1     1     1     1     1
>> c = -2:2, a./c
c =
    -2    -1     0     1     2
Warning: Divide by zero
ans =
 -0.5000  -2.0000  Inf   4.0000   2.5000
```

The previous calculation required division by 0—notice the `Inf`, denoting infinity, in the answer.

```
>> a.*b -24, ans./c
ans =
   -18   -10     0    12    26
```

```
Warning: Divide by zero
ans =
     9    10   NaN    12    13
```

Here we are warned about 0/0—giving a `NaN` (**N**ot **a** **N**umber).

Example 14.2 *Estimate the limit*

$$\lim_{x \to 0} \frac{\sin \pi x}{x}.$$

The idea is to observe the behaviour of the ratio $\frac{\sin \pi x}{x}$ for a sequence of values of x that approach zero. Suppose that we choose the sequence defined by the column vector
```
>> x = [0.1; 0.01; 0.001; 0.0001]
```
then
```
>> sin(pi*x)./x
ans =
     3.0902
     3.1411
     3.1416
     3.1416
```

which suggests that the values approach π. To get a better impression, we subtract the value of π from each entry in the output and, to display more decimal places, we change the format
```
>> format long
>> ans -pi
ans =
  -0.05142270984032
  -0.00051674577696
  -0.00000516771023
  -0.00000005167713
```

Can you explain the pattern revealed in these numbers? We also need to use `./` to compute a scalar divided by a vector:
```
>> 1/x
??? Error using ==> /
Matrix dimensions must agree.
>> 1./x
ans =
    10    100   1000   10000
```

so `1./x` works, but `1/x` does not.

14.4   Dot Power of Arrays (.^)

To square each of the elements of a vector we could, for example, do `u.*u`. However, a neater way is to use the `.^` operator:
```
>> u.^2
ans =
   100   121   144
>> u.*u
ans =
   100   121   144
```

```
>> u.^4
ans =
        10000        14641        20736
>> v.^2
ans =
   400
   441
   484
>> u.*w.^(-2)
ans =
    2.5000   -11.0000     1.3333
```

Recall that powers (`.^` in this case) are done first, before any other arithmetic operation.

15   Examples in Plotting

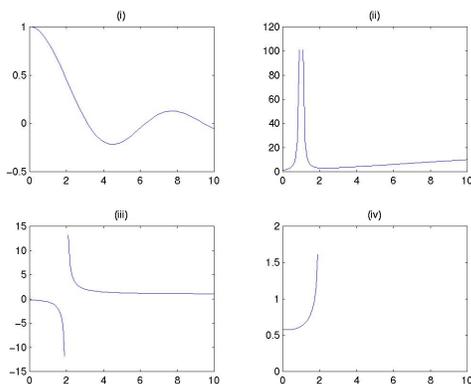**Example 15.1** *Draw graphs of the functions*

$$
\begin{array}{llll}
i) & y = \frac{\sin x}{x} & ii) & u = \frac{1}{(x-1)^2} + x \\
iii) & v = \frac{x^2+1}{x^2-4} & iv) & w = \frac{(10-x)^{1/3}-2}{(4-x^2)^{1/2}}
\end{array}
$$

for $0 \le x \le 10$.

```
>> x = 0:0.1:10;
>> y = sin(x)./x;
>>     subplot(221), plot(x,y), title('(i)')
Warning: Divide by zero
>> u = 1./(x-1).^2 + x;
>>     subplot(222),plot(x,u), title('(ii)')
Warning: Divide by zero
>> v = (x.^2+1)./(x.^2-4);
>>     subplot(223),plot(x,v),title('(iii)')
Warning: Divide by zero
>> w = ((10-x).^(1/3)-1)./sqrt(4-x.^2);
Warning: Divide by zero
>>     subplot(224),plot(x,w),title('(iv)')
```


Note the repeated use of the "dot" operators.
Experiment by changing the axes (page 9), grids (page 7) and hold(page 7).

```
>>    subplot(222),axis([0 10 0 10])
>>    grid
>>    grid
>>    hold on
>>    plot(x,v,'--'), hold off, plot(x,y,':')
```

Exercise 15.1 *Enter the vectors*

$$
\underline{U} = [6, 2, 4], \quad \underline{V} = [3, -2, 3, 0],
$$

$$
\underline{W} = \begin{bmatrix} 3 \\ -4 \\ 2 \\ -6 \end{bmatrix}, \quad \underline{Z} = \begin{bmatrix} 3 \\ 2 \\ 2 \\ 7 \end{bmatrix}
$$

into Matlab.

1. *Which of the products*
    ```
    U*V,  V*W,  U*V', V*W', W*Z', U.*V
    U'*V, V'*W, W'*Z, U.*W, W.*Z, V.*W
    ```
    is legal? State whether the legal products are row or column vectors and give the values of the legal results.

2. *Tabulate the functions*

    $$
    y = (x^2 + 3)\sin \pi x^2
    $$

    and

    $$
    z = \sin^2 \pi x/(x^{-2} + 3)
    $$

    for $x = 0, 0.2, \ldots, 10$. *Hence, tabulate the function*

    $$
    w = \frac{(x^2 + 3)\sin \pi x^2 \sin^2 \pi x}{(x^{-2} + 3)}.
    $$

    Plot a graph of w over the range $0 \le x \le 10$.

16   Matrices—Two–Dimensional Arrays

Row and Column vectors are special cases of **matrices**. An $m \times n$ matrix is a rectangular array of numbers having m rows and n columns. It is usual in a mathematical setting to include the matrix in either round or square brackets—we shall use square ones. For example, when $m = 2, n = 3$ we have a 2×3 matrix such as

$$
A = \begin{bmatrix} 5 & 7 & 9 \\ 1 & -3 & -7 \end{bmatrix}
$$

To enter such an matrix into Matlab we type it in row by row using the same syntax as for vectors:

```
>> A = [5 7 9
        1 -3 -7]
A =
     5       7       9
     1      -3      -7
```

Rows may be separated by semi-colons rather than a new line:

```
>> B = [-1 2 5; 9 0 5]
B =
    -1       2       5
     9       0       5
>> C = [0, 1; 3, -2; 4, 2]
C =
     0       1
     3      -2
     4       2
```

13

```
>> D = [1:5; 6:10; 11:2:20]
D =
     1     2     3     4     5
     6     7     8     9    10
    11    13    15    17    19
```

So A and B are 2×3 matrices, C is 3×2 and D is 3×5. In this context, a row vector is a $1 \times n$ matrix and a column vector a $m \times 1$ matrix.

16.1 Size of a matrix

We can get the size (dimensions) of a matrix with the command `size`

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

So A is 2×3 and x is 3×1 (a column vector). The last command `size(ans)` shows that the *value* returned by `size` is itself a 1×2 matrix (a row vector). We can save the results for use in subsequent calculations.

```
>> [r c] = size(A'), S = size(A')
r =
     3
c =
     2
S =
     3     2
```

16.2 Transpose of a matrix

Transposing a vector changes it from a row to a column vector and vice versa (see §8.4). The extension of this idea to matrices is that transposing interchanges rows with the corresponding columns: the 1st row becomes the 1st column, and so on.

```
>> D, D'
D =
     1     2     3     4     5
     6     7     8     9    10
    11    13    15    17    19
ans =
     1     6    11
     2     7    13
     3     8    15
     4     9    17
     5    10    19
>> size(D), size(D')
ans =
     3     5
ans =
     5     3
```

16.3 Special Matrices

Matlab provides a number of useful built–in matrices of any desired size.
`ones(m,n)` gives an $m \times n$ matrix of 1's,

```
>> P = ones(2,3)
P =
     1     1     1
     1     1     1
```

`zeros(m,n)` gives an $m \times n$ matrix of 0's,

```
>> Z = zeros(2,3), zeros(size(P'))
Z =
     0     0     0
     0     0     0
ans =
     0     0
     0     0
     0     0
```

The second command illustrates how we can construct a matrix based on the size of an existing one. Try `ones(size(D))`.
An $n \times n$ matrix that has the same number of rows and columns and is called a **square** matrix.
A matrix is said to be **symmetric** if it is equal to its transpose (i.e. it is unchanged by transposition):

```
>> S = [2 -1 0; -1 2 -1; 0 -1 2],
S =
     2    -1     0
    -1     2    -1
     0    -1     2
>> St = S'
St =
     2    -1     0
    -1     2    -1
     0    -1     2
>> S-St
ans =
     0     0     0
     0     0     0
     0     0     0
```

16.4 The Identity Matrix

The $n \times n$ **identity** matrix is a matrix of zeros except for having ones along its leading diagonal (top left to bottom right). This is called `eye(n)` in Matlab (since mathematically it is usually denoted by I).

```
>> I = eye(3), x = [8; -4; 1], I*x
I =
     1     0     0
     0     1     0
     0     0     1
x =
     8
    -4
     1
ans =
     8
    -4
     1
```

Notice that multiplying the 3×1 vector `x` by the 3×3 identity `I` has no effect (it is like multiplying a number by 1).

16.5 Diagonal Matrices

A diagonal matrix is similar to the identity matrix except that its diagonal entries are not necessarily equal to 1.

$$D = \begin{bmatrix} -3 & 0 & 0 \\ 0 & 4 & 0 \\ 0 & 0 & 2 \end{bmatrix}$$

is a 3×3 diagonal matrix. To construct this in Matlab, we could either type it in directly

```
>> D = [-3 0 0; 0 4 0; 0 0 2]
D =
    -3     0     0
     0     4     0
     0     0     2
```

but this becomes impractical when the dimension is large (e.g. a 100×100 diagonal matrix). We then use the `diag` function. We first define a vector `d`, say, containing the values of the diagonal entries (in order) then `diag(d)` gives the required matrix.

```
>>  d = [-3 4 2], D = diag(d)
d =
    -3     4     2
D =
    -3     0     0
     0     4     0
     0     0     2
```

On the other hand, if `A` is any matrix, the command `diag(A)` extracts its diagonal entries:

```
>> F = [0 1 8 7; 3 -2 -4 2; 4 2 1 1]
F =
     0     1     8     7
     3    -2    -4     2
     4     2     1     1
>> diag(F)
ans =
     0
    -2
     1
```

Notice that the matrix does not have to be square.

16.6 Building Matrices

It is often convenient to build large matrices from smaller ones:

```
>> C=[0 1; 3 -2; 4 2]; x=[8;-4;1];
>> G = [C x]
G =
     0     1     8
     3    -2    -4
     4     2     1
>> A, B, H = [A; B]
A =
```

```
     5     7     9
     1    -3    -7
B =
    -1     2     5
     9     0     5
ans =
     5     7     9
     1    -3    -7
    -1     2     5
     9     0     5
```

so we have added an extra column (`x`) to `C` in order to form `G` and have stacked `A` and `B` on top of each other to form `H`.

```
>> J = [1:4; 5:8; 9:12; 20 0 5 4]
J =
     1     2     3     4
     5     6     7     8
     9    10    11    12
    20     0     5     4
>> K = [ diag(1:4) J; J' zeros(4,4)]
K =
    1    0    0    0    1    2    3    4
    0    2    0    0    5    6    7    8
    0    0    3    0    9   10   11   12
    0    0    0    4   20    0    5    4
    1    5    9   20    0    0    0    0
    2    6   10    0    0    0    0    0
    3    7   11    5    0    0    0    0
    4    8   12    4    0    0    0    0
```

The command `spy(K)` will produce a graphical display of the location of the nonzero entries in `K` (it will also give a value for `nz`—the number of nonzero entries):

```
>> spy(K), grid
```

16.7 Tabulating Functions

This has been addressed in earlier sections but we are now in a position to produce a more suitable table format.

Example 16.1 *Tabulate the functions $y = 4\sin 3x$ and $u = 3\sin 4x$ for $x = 0, 0.1, 0.2, \ldots, 0.5$.*

```
>> x = 0:0.1:0.5;
>> y = 4*sin(3*x); u = 3*sin(4*x);
>> [ x' y' u']
ans =
         0         0         0
    0.1000    1.1821    1.1683
    0.2000    2.2586    2.1521
    0.3000    3.1333    2.7961
    0.4000    3.7282    2.9987
    0.5000    3.9900    2.7279
```

Note the use of transpose (`'`) to get column vectors. (we could replace the last command by `[x; y; u;]'`) We could also have done this more directly:

```
>>  x = (0:0.1:0.5)';
>> [x 4*sin(3*x) 3*sin(4*x)]
```

16.8 Extracting Bits of Matrices

We may extract sections from a matrix in much the same way as for a vector (page 5).

Each element of a matrix is indexed according to which row and column it belongs to. The entry in the ith row and jth column is denoted mathematically by $A_{i,j}$ and, in Matlab, by `A(i,j)`. So

```
>> J
J =
     1     2     3     4
     5     6     7     8
     9    10    11    12
    20     0     5     4
>> J(1,1)
ans =
     1
>> J(2,3)
ans =
     7
>> J(4,3)
ans =
     5
>> J(4,5)
???  Index exceeds matrix dimensions.
>> J(4,1) = J(1,1) + 6
J =
     1     2     3     4
     5     6     7     8
     9    10    11    12
     7     0     5     4
>> J(1,1) = J(1,1) - 3*J(1,2)
J =
    -5     2     3     4
     5     6     7     8
     9    10    11    12
     7     0     5     4
```

In the following examples we extract i) the 3rd column, ii) the 2nd and 3rd columns, iii) the 4th row, and iv) the "central" 2×2 matrix. See §8.1.

```
>> J(:,3)          % 3rd column
ans =
     3
     7
    11
     5
>> J(:,2:3)        % columns 2 to 3
ans =
     2     3
     6     7
    10    11
     0     5
>> J(4,:)          % 4th row
ans =
     7     0     5     4
>> J(2:3,2:3)      % rows 2 to 3 & cols 2 to 3
ans =
     6     7
    10    11
```

Thus, : on its own refers to the entire column or row depending on whether it is the first or the second index.

16.9 Dot product of matrices (.*)

The dot product works as for vectors: corresponding elements are multiplied together—so the matrices involved must have the same size.

```
>> A, B
A =
     5     7     9
     1    -3    -7
B =
    -1     2     5
     9     0     5
>> A.*B
ans =
    -5    14    45
     9     0   -35
>> A.*C
??? Error using ==> .*
Matrix dimensions must agree.
>> A.*C'
ans =
     0    21    36
     1     6   -14
```

16.10 Matrix–vector products

We turn next to the definition of the product of a matrix with a vector. This product is only defined for **column vectors** that have the same number of entries as the matrix has columns. So, if A is an $m \times n$ matrix and $\underline{x}$ is a column vector of length n, then the matrix–vector $A\underline{x}$ is legal.

An $m \times n$ matrix times an $n \times 1$ matrix $\Rightarrow$ a $m \times 1$ matrix.

We visualise A as being made up of m row vectors stacked on top of each other, then the product corresponds to taking the **scalar** product (See §14.1) of each row of A with the vector $\underline{x}$: The result is a column vector with m entries.

$$
\begin{aligned}
A\underline{x} &= \begin{bmatrix} \boxed{\begin{matrix} 5 & 7 & 9 \end{matrix}} \\ \boxed{\begin{matrix} 1 & -3 & -7 \end{matrix}} \end{bmatrix} \begin{bmatrix} 8 \\ -4 \\ 1 \end{bmatrix} \\
&= \begin{bmatrix} 5 \times 8 + 7 \times (-4) + 9 \times 1 \\ 1 \times 8 + (-3) \times (-4) + (-7) \times 1 \end{bmatrix} \\
&= \begin{bmatrix} 21 \\ 13 \end{bmatrix}
\end{aligned}
$$

It is somewhat easier in Matlab:

```
>> A = [5 7 9; 1 -3 -7]
A =
     5     7     9
     1    -3    -7
>> x = [8; -4; 1]
x =
     8
    -4
     1
>> A*x
ans =
```

```
      21
      13
```

$$(m \times \boxed{n}) \text{ times } (\boxed{n} \times 1) \Rightarrow (m \times 1).$$

```
>> x*A
??? Error using ==> *
Inner matrix dimensions must agree.
```

Unlike multiplication in arithmetic, **A*x is not the same as x*A**.

16.11 Matrix–Matrix Products

To form the product of an $m \times n$ matrix A and a $n \times p$ matrix B, written as AB, we visualise the first matrix (A) as being composed of m row vectors of length n stacked on top of each other while the second (B) is visualised as being made up of p column vectors of length n:

The entry in the ith row and jth column of the product is then the **scalar**product of the ith row of A with the jth column of B. The product is an $m \times p$ matrix:

$$(m \times \boxed{n}) \text{ times } (\boxed{n} \times p) \Rightarrow (m \times p).$$

Check that you understand what is meant by working out the following examples by hand and comparing with the Matlab answers.

```
>> A = [5  7  9;  1 -3 -7]
A =
     5     7     9
     1    -3    -7
>> B = [0, 1; 3, -2; 4, 2]
B =
     0     1
     3    -2
     4     2
>> C = A*B
C =
    57     9
   -37    -7
>> D = B*A
D =
     1    -3    -7
    13    27    41
    22    22    22
>> E = B'*A'
E =
    57   -37
     9    -7
```

We see that E = C' suggesting that

(A*B)' = B'*A'
Why is $B * A$ a 3×3 matrix while $A * B$ is 2×2?

Exercise 16.1 *It is often necessary to factorize a matrix, e.g., $A = BC$ or $A = S^T X S$ where the factors are required to have specific properties. Use the 'lookfor keyword' command to make a list of factorizations commands in Matlab.*

16.12 Sparse Matrices

Matlab has powerful techniques for handling sparse matrices — these are generally large matrices (to make the extra work involved worthwhile) that have only a very small proportion of non–zero entries.

Example 16.2 *Create a sparse 5×4 matrix S having only 3 non–zero values: $S_{1,2} = 10$, $S_{3,3} = 11$ and $S_{5,4} = 12$.*

We first create 3 vectors containing the i–index, the j–index and the corresponding values of each term and we then use the sparse command.

```
>> i = [1, 3, 5]; j = [2,3,4];
>> v = [10 11 12];
>> S = sparse (i,j,v)
S =
   (1,2)        10
   (3,3)        11
   (5,4)        12
>> T = full(S)
T =
     0    10     0     0
     0     0     0     0
     0     0    11     0
     0     0     0     0
     0     0     0    12
```

The matrix T is a "full" version of the sparse matrix S.

Example 16.3 *Develop Matlab code to create, for any given value of n, the sparse (tridiagonal) matrix*

$$B = \begin{bmatrix} 1 & n & & & \\ -2 & 2 & n-1 & & \\ & -3 & 3 & n-2 & \\ & & \ddots & \ddots & \ddots \\ & & & -n+1 & n-1 & 1 \\ & & & & -n & n \end{bmatrix}$$

We define three COLUMN vectors, one for each "diagonal" of non–zeros and then assemble the matrix using spdiags (short for sparse diagonals). The vectors are named l, d and u. They must all have the same length and only the **first** $n-1$ terms of l are used while the **last** $n-1$ terms of u are used. spdiags places these vectors in the diagonals labelled -1, 0 and 1 (0 defers to the leading diagonal, negatively numbered diagonals lie below the leading diagonal, etc.)

```
>> n = 5;
>> l = -(2:n+1)'; d = (1:n )'; u = ((n+1):-1:2)';
>> B = spdiags([l' d' u'],-1:1,n,n);
>> full(B)
ans =
```

```
   1     5     0     0     0
  -2     2     4     0     0
   0    -3     3     3     0
   0     0    -4     4     2
   0     0     0    -5     5
```

17 Systems of Linear Equations

Mathematical formulations of engineering problems often lead to sets of simultaneous linear equations. Such is the case, for instance, when using the finite element method (FEM).

A general system of linear equations can be expressed in terms of a coefficient matrix A, a right-hand-side (column) vector $\mathbf{b}$ and an unknown (column) vector $\mathbf{x}$ as

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

or, componentwise, as

$$
\begin{aligned}
a_{1,1}x_1 + a_{1,2}x_2 + \cdots a_{1,n}x_n &= b_1 \\
a_{2,1}x_1 + a_{2,2}x_2 + \cdots a_{2,n}x_n &= b_2 \\
&\vdots \\
a_{n,1}x_1 + a_{n,2}x_2 + \cdots a_{n,n}x_n &= b_n
\end{aligned}
$$

When A is non-singular and square ($n \times n$), meaning that the number of *independent* equations is equal to the number of unknowns, the system has a unique solution given by

$$\mathbf{x} = A^{-1}\mathbf{b}$$

where A^{-1} is the inverse of A. Thus, the solution vector $\mathbf{x}$ can, in principle, be calculated by taking the inverse of the coefficient matrix A and multiplying it on the right with the right-hand-side vector $\mathbf{b}$.

This approach based on the matrix inverse, though formally correct, is at best inefficient for practical applications (where the number of equations may be extremely large) but may also give rise to large numerical errors unless appropriate techniques are used. These issues are discussed in most courses and texts on numerical methods. Various stable and efficient solution techniques have been developed for solving linear equations and the most appropriate in any situation will depend on the properties of the coefficient matrix A. For instance, on whether or not it is symmetric, or positive definite or if it has a particular structure (sparse or full). Matlab is equipped with many of these special techniques in its routine library and they are invoked automatically.

The standard Matlab routine for solving systems of linear equations is invoked by calling the matrix left-division routine,

```
>> x = A \ b
```

where "\" is the matrix left-division operator known as "backslash" (see `help backslash`).

Exercise 17.1 *Enter the symmetric coefficient matrix and right–hand–side vector* $\mathbf{b}$ *given by*

$$
A = \left[\begin{array}{ccc} 2 & -1 & 0 \\ 1 & -2 & 1 \\ 0 & -1 & 2 \end{array} \right], \qquad \mathbf{b} = \left[\begin{array}{c} 1 \\ 0 \\ 1 \end{array} \right]
$$

and solve the system of equations $A\mathbf{x} = \mathbf{b}$ *using the three alternative methods:*

  i) $\mathbf{x} = A^{-1}\mathbf{b}$, *(the inverse* A^{-1} *may be computed in Matlab using* `inv(A)`.)

 ii) `x = A \ b`,

iii) $\mathbf{x}^T = \mathbf{b}^t A^T$ *leading to* `xT = b' / A` *which makes use of the "slash" or "right division" operator "/". The required solution is then the transpose of the row vector* `xT`.

Exercise 17.2 *Use the backslash operator to solve the complex system of equations for which*

$$
A = \left[\begin{array}{ccc} 2+2i & -1 & 0 \\ -1 & 2-2i & -1 \\ 0 & -1 & 2 \end{array} \right], \qquad \mathbf{b} = \left[\begin{array}{c} 1+i \\ 0 \\ 1-i \end{array} \right]
$$

Exercise 17.3 *Find information on the matrix inversion command* `'inv'` *using each of the methods listed in Section 2 for obtaining help.*
What kind of matrices are the `'inv'` *command applicable to?*
Obviously problems may occur if the inverted matrix is nearly singular. Suggest a command that can be used to give an indication on whether the matrix is nearly singular or not. [Hint: see the topics referred to by `'help inv'`.]

17.1 Overdetermined system of linear equations

An overdetermined system of linear equations is a one with more equations (m) than unknowns (n), i.e., the coefficient matrix has more rows than columns ($m > n$). Overdetermined systems frequently appear in mathematical modelling when the parameters of a model are determined by fitting to experimental data. Formally the system looks the same as for square systems but the coefficient matrix is rectangular and so it is not possible to compute an inverse. In these cases a solution can be found by requiring that the magnitude of the residual vector $\mathbf{r}$, defined by

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

be minimized. The simplest and most frequently used measure of the magnitude of $\mathbf{r}$ is require the Euclidean length (or norm—see Section 14.1) which corresponds to the sum of squares of the components of the residual. This approach leads to the least squares solution of the overdetermined system. Hence the least squares solution is defined as the vector $\mathbf{x}$ that minimizes

$$\mathbf{r}^T\mathbf{r}.$$

It may be shown that the required solution satisfies the so–called "normal equations"

$$C\mathbf{x} = \mathbf{d}, \text{ where } C = A^T A \text{ and } \mathbf{d} = A^T\mathbf{b}.$$

This system is well–known that the solution of this system can be overwhelmed by numerical rounding error in practice unless great care is taken in its solution (a

large part of the difficulty is inherent in computing the matrix–matrix product $A^T A$. As in the solution of square systems of linear equations, special techniques have been developed to address these issues and they have been incorporated into the Matlab routine library. This means that a direct solution to the problem of overdetermined equations is available in Matlab through its left division operator "\". When the matrix A is not square, the operation

$$x = A\backslash b$$

automatically gives the least squares solution to $A\mathbf{x} = \mathbf{b}$. This is illustrated in the next example.

Example 17.1 *A spring is a mechanical element which, for the simplest model, is characterized by a linear force-deformation relationship*

$$F = kx,$$

F being the force loading the spring, k the spring constant or stiffness and x the spring deformation. In reality the linear force–deformation relationship is only an approximation, valid for small forces and deformations. A more accurate relationship, valid for larger deformations, is obtained if non–linear terms are taken into account. Suppose a spring model with a quadratic relationship

$$F = k_1 x + k_2 x^2$$

is to be used and that the model parameters, k_1 and k_2, are to be determined from experimental data. Five independent measurements of the force and the corresponding spring deformations are measured and these are presented in Table 1.

Force F [N]	Deformation x [cm]
5	0.001
50	0.011
500	0.013
1000	0.30
2000	0.75

Table 1: Measured force-deformation data for spring.

Using the quadratic force-deformation relationship together with the experimental data yields an overdetermined system of linear equations and the components of the residual are given by

$$
\begin{aligned}
r_1 &= x_1 k_1 + x_1^2 k_2 - F_1 \\
r_2 &= x_2 k_1 + x_2^2 k_2 - F_2 \\
r_3 &= x_3 k_1 + x_3^2 k_2 - F_3 \\
r_4 &= x_4 k_1 + x_4^2 k_2 - F_4 \\
r_5 &= x_5 k_1 + x_5^2 k_2 - F_5.
\end{aligned}
$$

These lead to the matrix and vector definitions

$$
A = \begin{bmatrix} x_1 & x_1^2 \\ x_2 & x_2^2 \\ x_3 & x_3^2 \\ x_4 & x_4^2 \\ x_5 & x_5^2 \end{bmatrix} \text{ and } \mathbf{b} = \begin{bmatrix} F_1 \\ F_2 \\ F_3 \\ F_4 \\ F_5. \end{bmatrix}
$$

The appropriate Matlab commands give (the components of x are all multiplied by `1e-2`, i.e., 10^{-2}, in order to change from cm to m)

```
>> x = [.001 .011 .13 .3 .75]*1e-2;
>> A = [x' (x').^2]
A =
     0.0000      0.0000
     0.0001      0.0000
     0.0013      0.0000
     0.0030      0.0000
     0.0075      0.0001
>> b = [5 50 500 1000 2000];
```

and the least squares solution to this system is given by

```
>> k = A\b'
k =
  1.0e+07 *
     0.0386
    -1.5993
```

Thus, $\mathbf{k} \approx \begin{bmatrix} 0.39 \\ -16.0 \end{bmatrix} \times 10^6$ and the quadratic spring force-deformation relationship that optimally fits experimental data in the least squares sense is

$$F \approx 38.6 \times 10^4 x - 16.0 \times 10^6 x^2.$$

The data and solution may be plotted with the following commands

```
>> plot(x,f,'o'), hold on      % plot data points
>> X = (0:.01:1)*max(x);
>> plot(X,[X' (X.^2)']*k,'-')  % best fit curve
>> xlabel('x[m]'), ylabel('F[N]')
```

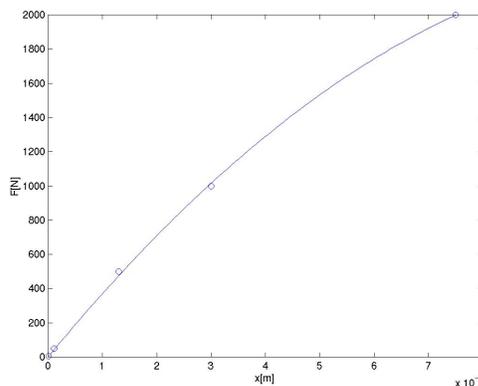and the results are shown in Figure 5.

Figure 5: Data for Example 17.1 (circles) and best least squares fit by a quadratic model (solid line).

Matlab has a routine `polyfit` for data fitting by polynomials: see "`help polyfit`". It is not applicable in this example because we require that the force – deformation law passes through the origin (so there is no constant term in the quadratic model that we used).

18 Characters, Strings and Text

The ability to process text in numerical processing is useful for the input and output of data to the screen or to disk-files. In order to manage text, a new datatype of "character" is introduced. A piece of text is then simply a string (vector) or array of characters.

Example 18.1 *The assignment,*

```
>> t1 = 'A'
```

assigns the value `A` *to the* `1-by-1` *character array* `t1`. *The assignment,*

```
>> t2 = 'BCDE'
```

assigns the value `BCDE` *to the* `1-by-4` *character array* `t2`.

Strings can be combined by using the operations for array manipulations.
The assignment,

```
>> t3 = [t1,t2]
```

assigns a value `ABCDE` to the `1-by-5` character array `t3`. The assignment,

```
>> t4 = [t3,' are the first 5        ';...
'characters in the alphabet.']
```

assigns the value
`'ABCDE are the first 5 '`
`'characters in the alphabet.'`
to the `2-by-27` character array `t4`. It is essential that the number of characters in both rows of the array `t4` is the same, otherwise an error will result. The three dots `...` signify that the command is continued on the following line
Sometimes it is necessary to convert a character to the corresponding number, or vice versa. These conversions are accomplished by the commands 'str2num'—which converts a string to the corresponding number, and two functions, 'int2str' and 'num2str', which convert, respectively, an integer and a real number to the corresponding character string. These commands are useful for producing titles and strings, such as 'The value of pi is 3.1416'. This can be generated by the command ['The value of pi is ',num2str(pi)].

```
>> N = 5; h = 1/N;
>> ['The value of N is ',int2str(N),...
', h = ',num2str(h)]
ans =
The value of N is 5, h = 0.2
```

19 Loops

There are occasions that we want to repeat a segment of code a number of different times (such occasions are less frequent than other programming languages because of the : notation).

Example 19.1 *Draw graphs of* $\sin(n\pi x)$ *on the interval* $-1 \le x \le 1$ *for* $n = 1, 2, \ldots, 8$.

We could do this by giving 8 separate `plot` commands but it is much easier to use a loop. The simplest form would be

```
>> x = -1:.05:1;
>> for n = 1:8
    subplot(4,2,n), plot(x,sin(n*pi*x))
  end
```

All the commands between the lines starting "`for`" and "`end`" are repeated with `n` being given the value `1` the first time through, `2` the second time, and so forth, until $n = 8$. The `subplot` constructs a 4×2 array of subwindows and, on the nth time through the loop, a picture is drawn in the nth subwindow.

The commands

```
>> x = -1:.05:1;
>> for n = 1:2:8
     subplot(4,2,n), plot(x,sin(n*pi*x))
     subplot(4,2,n+1), plot(x,cos(n*pi*x))
   end
```

draw $\sin n\pi x$ and $\cos n\pi x$ for $n = 1, 3, 5, 7$ alongside each other.

We may use any legal variable name as the "loop counter" (`n` in the above examples) and it can be made to run through all of the values in a given vector (`1:8` and `1:2:8` in the examples).

We may also use `for` loops of the type

```
>> for counter = [23  11  19  5.4  6]
      .......
   end
```

which repeats the code as far as the `end` with `counter=23` the first time, `counter=11` the second time, and so forth.

Example 19.2 *The Fibonnaci sequence starts off with the numbers* 0 *and* 1, *then succeeding terms are the sum of its two immediate predecessors. Mathematically,* $f_1 = 0$, $f_2 = 1$ *and*

$$f_n = f_{n-1} + f_{n-2}, \qquad n = 3, 4, 5, \ldots.$$

Test the assertion that the ratio f_{n-1}/f_n *of two successive values approaches the golden ratio* $(\sqrt{5} - 1)/2$ = 0.6180\ldots.

```
>> F(1) = 0; F(2) = 1;
>> for i = 3:20
     F(i) = F(i-1) + F(i-2);
   end
>> plot(1:19, F(1:19)./F(2:20),'o' )
>> hold on,  xlabel('n')
>> plot(1:19, F(1:19)./F(2:20),'-' )
>> legend('Ratio of terms f_{n-1}/f_n')
>> plot([0 20], (sqrt(5)-1)/2*[1,1],'--')
```

The last of these commands produces the dashed horizontal line.
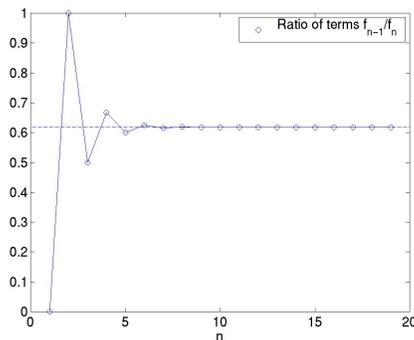
Example 19.3 *Produce a list of the values of the sums*

$$
\begin{aligned}
S_{20} &= 1 + \tfrac{1}{2^2} + \tfrac{1}{3^2} + \cdots + \tfrac{1}{20^2} \\
S_{21} &= 1 + \tfrac{1}{2^2} + \tfrac{1}{3^2} + \cdots + \tfrac{1}{20^2} + \tfrac{1}{21^2} \\
&\vdots \\
S_{100} &= 1 + \tfrac{1}{2^2} + \tfrac{1}{3^2} + \cdots + \tfrac{1}{20^2} + \tfrac{1}{21^2} + \cdots + \tfrac{1}{100^2}
\end{aligned}
$$

There are a total of 81 sums. The first can be computed using `sum(1./(1:20).^2)` (The function `sum` with a vector argument sums its components. See §22.2].) A suitable piece of Matlab code might be

```
>> S = zeros(100,1);
>> S(20) = sum(1./(1:20).^2);
>> for n = 21:100
>>   S(n) = S(n-1) + 1/n^2;
>> end
>> clf; plot(S,'.',[20 100],[1,1]*pi^2/6,'-')
>> axis([20 100 1.5 1.7])
>> [ (98:100)'  S(98:100)]
ans =
    98.0000    1.6364
    99.0000    1.6365
   100.0000    1.6366
```

where a column vector `S` was created to hold the answers. The first sum was computed directly using the `sum` command then each succeeding sum was found by adding $1/n^2$ to its predecessor. The little table at the end shows the values of the last three sums—it appears that they are approaching a limit (the value of the limit is $\pi^2/6 = 1.64493\ldots$).

Exercise 19.1 *Repeat Example 19.3 to include 181 sums (i.e., the final sum should include the term* $1/200^2$.)

20 Logicals

Matlab represents `true` and `false` by means of the integers 0 and 1.

$$\text{true} = 1, \qquad \text{false} = 0$$

If at some point in a calculation a scalar `x`, say, has been assigned a value, we may make certain logical tests on it:

`x == 2`	is `x` equal to 2?
`x ~= 2`	is `x` **not** equal to 2?
`x > 2`	is `x` greater than 2?
`x < 2`	is `x` less than 2?
`x >= 2`	is `x` greater than or equal to 2?
`x <= 2`	is `x` less than or equal to 2?

Pay particular attention to the fact that the test for equality involves two equal signs `==`.

```
>> x = pi
x =
    3.1416
>> x ~= 3,   x ~= pi
ans =
    1
ans =
    0
```

When `x` is a vector or a matrix, these tests are performed elementwise:

```
x =
   -2.0000    3.1416    5.0000
   -1.0000         0    1.0000
>> x == 0
ans =
     0     0     0
     0     1     0
>> x > 1,   x >=-1
ans =
     0     1     1
     0     0     0
ans =
     0     1     1
     1     1     1
>> y = x>=-1, x > y
y =
     0     1     1
     1     1     1
ans =
     0     1     1
     0     0     0
```

We may combine logical tests, as in

```
>> x
x =
   -2.0000    3.1416    5.0000
   -5.0000   -3.0000   -1.0000
>> x > 3 & x < 4
ans =
     0     1     0
     0     0     0
>> x > 3 | x == -3
ans =
     0     1     1
     0     1     0
```

As one might expect, `&` represents **and** and (not so clearly) the vertical bar `|` means **or**; also `~` means **not** as in `~=` (not equal), `~(x>0)`, etc.

```
>> x > 3 | x == -3 | x <= -5
ans =
     0     1     1
     1     1     0
```

One of the uses of logical tests is to "mask out" certain elements of a matrix.

```
>> x, L =  x >= 0
x =
   -2.0000    3.1416    5.0000
   -5.0000   -3.0000   -1.0000
L =
     0     1     1
     0     1     1
>> pos = x.*L
pos =
        0    3.1416    5.0000
        0         0         0
```

so the matrix `pos` contains just those elements of `x` that are non–negative.

```
>> x = 0:0.05:6; y = sin(pi*x); Y = (y>=0).*y;
>> plot(x,y,':',x,Y,'-' )
```
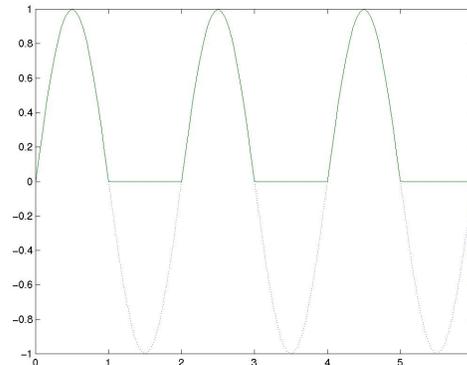

20.1   While Loops

There are some occasions when we want to repeat a section of Matlab code until some logical condition is satisfied, but we cannot tell in advance how many times we have to go around the loop. This we can do with a `while...end` construct.

Example 20.1 *What is the greatest value of n that can be used in the sum*

$$1^2 + 2^2 + \cdots + n^2$$

and get a value of less than 100?

```
>> S = 1; n = 1;
>> while S+ (n+1)^2 < 100
     n = n+1;    S = S + n^2;
   end
>> [n, S]
ans =
     6     91
```

The lines of code between `while` and `end` will only be executed if the condition `S+ (n+1)^2 < 100` is true.

Exercise 20.1 *Replace 100 in the previous example by 10 and work through the lines of code by hand. You should get the answers $n = 2$ and $S = 5$.*

Exercise 20.2 *Type the code from Example20.1 into a script–file named `WhileSum.m` (See §13.)*

A more typical example is

Example 20.2 *Find the approximate value of the root of the equation $x = \cos x$. (See Example 10.1.)*

We may do this by making a guess $x_1 = \pi/4$, say, then computing the sequence of values

$$x_n = \cos x_{n-1}, \qquad n = 2, 3, 4, \ldots$$

and continuing until the difference between two successive values $|x_n - x_{n-1}|$ is small enough.

Method 1:

```
>> x = zeros(1,20); x(1) = pi/4;
>> n = 1;  d = 1;
>> while d > 0.001
     n = n+1; x(n) = cos(x(n-1));
     d = abs( x(n) - x(n-1) );
   end
   n,x
n =
    14
x =
Columns 1 through 7
0.7854 0.7071 0.7602 0.7247 0.7487 0.7326 0.7435
Columns 8 through 14
0.7361 0.7411 0.7377 0.7400 0.7385 0.7395 0.7388
Columns 15 through 20
   0    0    0    0    0    0
```

There are a number of deficiencies with this program. The vector x stores the results of each iteration but we don't know in advance how many there may be. In any event, we are rarely interested in the intermediate values of x, only the last one. Another problem is that we may never satisfy the condition $d \leq 0.001$, in which case the program will run forever—we should place a limit on the maximum number of iterations. Incorporating these improvements leads to

Method 2:

```
>> xold = pi/4; n = 1;  d = 1;
>> while d > 0.001 &  n < 20
     n = n+1;  xnew = cos(xold);
     d = abs( xnew - xold );
     xold = xnew;
   end
>> [n, xnew, d]
ans =
   14.0000    0.7388    0.0007
```

We continue around the loop so long as $d > 0.001$ **and** $n < 20$. For greater precision we could use the condition $d > 0.0001$, and this gives

```
>> [n, xnew, d]
ans =
   19.0000    0.7391    0.0001
```

from which we may judge that the root required is $x = 0.739$ to 3 decimal places.
The general form of `while` statement is

```
while a logical test
            Commands to be executed
            when the condition is true
end
```

20.2 if...then...else...end

This allows us to execute different commands depending on the truth or falsity of some logical tests. To test whether or not π^e is greater than, or equal to, e^π:

```
>> a = pi^exp(1); c = exp(pi);
>> if a >= c
     b = sqrt(a^2 - c^2)
   end
```

so that b is assigned a value only if $a \geq c$. There is no output so we deduce that $a = \pi^e < c = e^\pi$. A more common situation is

```
>> if a >= c
     b = sqrt(a^2 - c^2)
   else
     b = 0
   end
b =
    0
```

which ensures that b is always assigned a value and confirming that $a < c$.
A more extended form is

```
>> if a >= c
     b = sqrt(a^2 - c^2)
   elseif a^c > c^a
     b = c^a/a^c
   else
     b = a^c/c^a
   end
b =
    0.2347
```

Exercise 20.3 *Which of the above statements assigned a value to* b*?*

The general form of the `if` statement is

```
if logical test 1
     Commands to be executed if test 1 is
     true
elseif logical test 2
     Commands to be executed if test 2 is
     true but test 1 is false
     ⋮

end
```

21 Function m–files

These are a combination of the ideas of script m–files (§7) and mathematical functions.

Example 21.1 *The area, A, of a triangle with sides of length a, b and c is given by*

$$A = \sqrt{s(s-a)(s-b)(s-c)},$$

where $s = (a+b+c)/2$. Write a Matlab function that will accept the values a, b and c as inputs and return the value of A as output.

The main steps to follow when defining a Matlab function are:

1. Decide on a name for the function, making sure that it does not conflict with a name that is already used by Matlab. In this example the name of the function is to be `area`, so its definition will be saved in a file called `area.m`

2. The first line of the file must have the format:

   `function` [*list of outputs*]
   $= function\_name(list\ of\ inputs)$

   For our example, the output (A) is a function of the three variables (inputs) a, b and c so the first line should read

   ```
   function [A] = area(a,b,c)
   ```

3. Document the function. That is, describe briefly the purpose of the function and how it can be used. These lines should be preceded by `%` which signify that they are comment lines that will be ignored when the function is evaluated.

4. Finally include the code that defines the function. This should be interspersed with sufficient comments to enable another user to understand the processes involved.

The complete file might look like:

```
function  [A] = area(a,b,c)
% Compute the area of a triangle whose
% sides have length a, b and c.
% Inputs:
%    a,b,c:  Lengths of sides
% Output:
%        A: area of triangle
% Usage:
%      Area = area(2,3,4);
% Written by dfg, Oct 14, 1996.
s = (a+b+c)/2;
A = sqrt(s*(s-a)*(s-b)*(s-c));
%%%%%%%% end of area %%%%%%%%%%
```

The command
```
>> help area
```
will produce the leading comments from the file:

```
  Compute the area of a triangle whose
  sides have length a, b and c.
  Inputs:
     a,b,c:  Lengths of sides
  Output:
         A: area of triangle
  Usage:
       Area = area(2,3,4);
  Written by dfg, Oct 14, 1996.
```

To evaluate the area of a triangle with side of length 10, 15, 20:

```
>> Area = area(10,15,20)
Area =
   72.6184
```

where the result of the computation is assigned to the variable `Area`. The variable `s` used in the definition of the function above is a "local variable": its value is local to the function and cannot be used outside:

```
>> s
??? Undefined function or variable s.
```

If we were to be interested in the value of s as well as A, then the first line of the file should be changed to

```
function [A,s] = area(a,b,c)
```

where there are two output variables.
This function can be called in several different ways:

1. No outputs assigned

   ```
   >> area(10,15,20)
   ans =
      72.6184
   ```

   gives only the area (first of the output variables from the file) assigned to `ans`; the second output is ignored.

2. One output assigned

   ```
   >> Area = area(10,15,20)
   Area =
      72.6184
   ```

   again the second output is ignored.

3. Two outputs assigned

   ```
   >> [Area, hlen] = area(10,15,20)
   Area =
      72.6184
   hlen =
      22.5000
   ```

Exercise 21.1 *In any triangle the sum of the lengths of any two sides cannot exceed the length of the third side. The function **area** does not check to see if this condition is fulfilled (try **area(1,2,4)**). Modify the file so that it computes the area only if the sides satisfy this condition.*

21.1   Examples of functions

We revisit the problem of computing the Fibonnaci sequence defined by $f_1 = 0, f_2 = 1$ and

$$f_n = f_{n-1} + f_{n-2}, \qquad n = 3, 4, 5, \ldots.$$

We want to construct a function that will return the nth number in the Fibonnaci sequence f_n.

- **Input**: Integer n
- **Output**: f_n

We shall describe four possible functions and try to assess which provides the best solution.

Method 1:   File `Fib1.m`

```
function f = Fib1(n)
% Returns the nth number in the
% Fibonacci sequence.
F=zeros(1,n+1);
F(2) = 1;
   for i = 3:n+1
     F(i) = F(i-1) + F(i-2);
   end
f = F(n);
```

This code resembles that given in Example 19.2. We have simply enclosed it in a function m–file and given it the appropriate header,

Method 2: File `Fib2.m`
The first version was rather wasteful of memory—it saved all the entries in the sequence even though we only required the last one for output. The second version removes the need to use a vector.

```
function f = Fib2(n)
% Returns the nth number in the
% Fibonacci sequence.
if n==1
   f = 0;
elseif n==2
   f = 1;
else
   f1 = 0; f2 = 1;
   for i = 2:n-1
     f = f1 + f2;
     f1=f2; f2 = f;
   end
end
```

Method 3: File: `Fib3.m`
This version makes use of an idea called "recursive programming"— the function makes calls to itself.

```
function f = Fib3(n)
% Returns the nth number in the
% Fibonacci sequence.
if n==1
   f = 0;
elseif n==2
   f = 1;
else
   f = Fib3(n-1) + Fib3(n-2);
end
```

Method 4: File `Fib4.m`
The final version uses matrix powers. The vector `y` has two components, $\underline{y} = \begin{bmatrix} f_n \\ f_{n+1} \end{bmatrix}$.

```
function f = Fib4(n)
% Returns the nth number in the
% Fibonacci sequence.
A = [0 1;1 1];
y = A^n*[1;0];
f=y(1);
```

Assessment: One may think that, on grounds of style, the 3rd is best (it avoids the use of loops) followed by the second (it avoids the use of a vector). The situation is much different when it cames to speed of execution. When $n = 20$ the time taken by each of the methods is (in seconds)

Method	Time
1	0.0118
2	0.0157
3	36.5937
4	0.0078

It is impractical to use Method 3 for any value of n much larger than 10 since the time taken by method 3 almost doubles whenever n is increased by just 1. When $n = 150$

Method	Time
1	0.0540
2	0.0891
3	—
4	0.0106

Clearly the 4th method is much the fastest.

22   Further Built–in Functions

22.1   Rounding Numbers

There are a variety of ways of rounding and chopping real numbers to give integers. Use the definitions given in the table in §28 on page 32 in order to understand the output given below:

```
>> x = pi*(-1:3),  round(x)
x =
  -3.1416  0    3.1416  6.2832   9.4248
ans =
    -3     0     3      6       9
>> fix(x)
ans =
    -3     0     3      6       9
>> floor(x)
ans =
    -4     0     3      6       9
>> ceil(x)
ans =
    -3     0     4      7      10
>> sign(x),  rem(x,3)
ans =
    -1     0     1      1       1
ans =
  -0.1416  0   0.1416  0.2832   0.4248
```

Do "`help round`" for help information.

22.2   The sum Function

The "`sum`" applied to a vector adds up its components (as in `sum(1:10)`) while, for a matrix, it adds up the components in **each column** and returns a row vector. `sum(sum(A))` then sums all the entries of `A`.

```
>>  A = [1:3; 4:6; 7:9]
A =
     1     2     3
     4     5     6
     7     8     9
>> s = sum(A), ss = sum(sum(A))
```

```
s =
    12    15    18
ss =
    45

>> x = pi/4*(1:3)';
>> A = [sin(x), sin(2*x), sin(3*x)]/sqrt(2)
>> A =
    0.5000    0.7071    0.5000
    0.7071    0.0000   -0.7071
    0.5000   -0.7071    0.5000

>> s1 = sum(A.^2), s2 = sum(sum(A.^2))
s1 =
    1.0000    1.0000    1.0000
s2 =
    3.0000
```

The sums of squares of the entries in each column of `A` are equal to 1 and the sum of squares of all the entries is equal to 3.

```
>> A*A'
ans =
    1.0000         0         0
         0    1.0000    0.0000
         0    0.0000    1.0000
>> A'*A
ans =
    1.0000         0         0
         0    1.0000    0.0000
         0    0.0000    1.0000
```

It appears that the products AA' and $A'A$ are both equal to the identity:

```
>> A*A' - eye(3)
ans =
    1.0e-15 *
   -0.2220         0         0
         0   -0.2220    0.0555
         0    0.0555   -0.2220
>> A'*A - eye(3)
ans =
    1.0e-15 *
   -0.2220         0         0
         0   -0.2220    0.0555
         0    0.0555   -0.2220
```

This is confirmed since the differences are at round–off error levels (less than 10^{-15}). A matrix with this property is called an *orthogonal* matrix.

22.3   max & min

These functions act in a similar way to `sum`. If `x` is a vector, then `max(x)` returns the largest element in `x`

```
>> x = [1.3 -2.4 0 2.3], max(x), max(abs(x))
x =
    1.3000   -2.4000         0    2.3000
ans =
    2.3000
ans =
```

```
    2.4000
>> [m, j] = max(x)
m =
    2.3000
j =
    4
```

When we ask for two outputs, the first gives us the maximum entry and the second the index of the maximum element.

For a matrix, `A`, `max(A)` returns a row vector containing the maximum element from each column. Thus to find the largest element in `A` we have to use `max(max(A))`.

22.4   Random Numbers

The function `rand(m,n)` produces an $m \times n$ matrix of random numbers, each of which is in the range 0 to 1. `rand` on its own produces a single random number.

```
>> y =  rand, Y = rand(2,3)
y =
    0.9191
Y =
    0.6262    0.1575    0.2520
    0.7446    0.7764    0.6121
```

Repeating these commands will lead to different answers.

Example: *Write a function–file that will simulate n throws of a pair of dice.*

This requires random numbers that are integers in the range 1 to 6. Multiplying each random number by 6 will give a real number in the range 0 to 6; rounding these to whole numbers will not be correct since it will then be possible to get 0 as an answer. We need to use

```
floor(1 + 6*rand)
```

Recall that `floor` takes the largest integer that is smaller than a given real number (see Table 2, page 32).

File: dice.m

```
function  [d] = dice(n)
% simulates "n" throws of a pair of dice
% Input:    n, the number of throws
% Output:   an n times 2 matrix, each row
%           referring to one throw.
%
% Useage:  T = dice(3)
   d = floor(1 + 6*rand(n,2));
%% end of dice
```

```
>> dice(3)
ans =
     6     1
     2     3
     4     1
>> sum(dice(100))/100
ans =
    3.8500    3.4300
```

The last command gives the average value over 100 throws (it should have the value 3.5).

22.5 `find` for vectors

The function "`find`" returns a list of the positions (indices) of the elements of a vector satisfying a given condition. For example,

```
>> x = -1:.05:1;
>> y = sin(3*pi*x).*exp(-x.^2); plot(x,y,':')
>> k = find(y > 0.2)
 k =
  Columns 1 through 12
   9  10  11  12  13  22  23  24  25  26  27  36
  Columns 13 through 15
  37  38  39
>> hold on, plot(x(k),y(k),'o')
>> km = find( x>0.5 & y<0)
km =
    32    33    34
>> plot(x(km),y(km),'-')
```
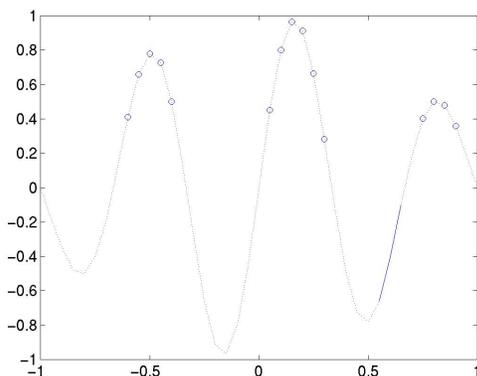

22.6 `find` for matrices

The `find`–function operates in much the same way for matrices:

```
>> A = [ -2 3 4 4; 0 5 -1 6; 6 8 0 1]
A =
    -2     3     4     4
     0     5    -1     6
     6     8     0     1
>> k = find(A==0)
k =
     2
     9
```

Thus, we find that `A` has elements equal to 0 in positions 2 and 9. To interpret this result we have to recognize that "`find`" first reshapes `A` into a column vector—this is equivalent to numbering the elements of `A` by columns as in

$$\begin{array}{cccc} 1 & 4 & 7 & 10 \\ 2 & 5 & 8 & 11 \\ 3 & 6 & 9 & 12 \end{array}$$

```
>> n  = find(A <= 0)
n =
```

```
     1
     2
     8
     9
>> A(n)
ans =
    -2
     0
    -1
     0
```

Thus, `n` gives a list of the locations of the entries in `A` that are ≤ 0 and then `A(n)` gives us the values of the elements selected.

```
>> m = find( A' == 0)
m =
     5
    11
```

Since we are dealing with `A'`, the entries are numbered by rows.

23 Plotting Surfaces

A surface is defined mathematically by a function $f(x, y)$—corresponding to each value of (x, y) we compute the height of the function by

$$z = f(x, y).$$

In order to plot this we have to decide on the ranges of x and y—suppose $2 \leq x \leq 4$ and $1 \leq y \leq 3$. This gives us a square in the (x, y)–plane. Next, we need to choose a grid on this domain; Figure 6 shows the grid with intervals 0.5 in each direction. Finally, we have
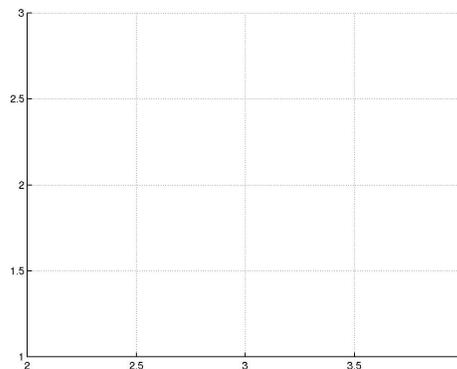
Figure 6: An example of a 2D grid

to evaluate the function at each point of the grid and "plot" it.

Suppose we choose a grid with intervals 0.5 in each direction for illustration. The x– and y–coordinates of the grid lines are

```
x = 2:0.5:4;  y = 1:0.5:3;
```

in Matlab notation. We construct the grid with `meshgrid`:

```
>> [X,Y] = meshgrid(2:.5:4, 1:.5:3);
>> X
X =
    2.0000    2.5000    3.0000    3.5000    4.0000
    2.0000    2.5000    3.0000    3.5000    4.0000
    2.0000    2.5000    3.0000    3.5000    4.0000
    2.0000    2.5000    3.0000    3.5000    4.0000
    2.0000    2.5000    3.0000    3.5000    4.0000
>> Y
Y =
    1.0000    1.0000    1.0000    1.0000    1.0000
    1.5000    1.5000    1.5000    1.5000    1.5000
    2.0000    2.0000    2.0000    2.0000    2.0000
    2.5000    2.5000    2.5000    2.5000    2.5000
    3.0000    3.0000    3.0000    3.0000    3.0000
```

If we think of the ith point along from the left and the jth point up from the bottom of the grid) as corresponding to the (i, j)th entry in a matrix, then ($X(i,j)$, $Y(i,j)$) are the coordinates of the point. We then need to evaluate the function f using X and Y in place of x and y, respectively.

Example 23.1 *Plot the surface defined by the function*

$$f(x,y) = (x-3)^2 - (y-2)^2$$

for $2 \leq x \leq 4$ and $1 \leq y \leq 3$.

```
>> [X,Y] = meshgrid(2:.2:4, 1:.2:3);
>> Z = (X-3).^2-(Y-2).^2;
>> mesh(X,Y,Z)
>> title('Saddle'), xlabel('x'),ylabel('y')
```
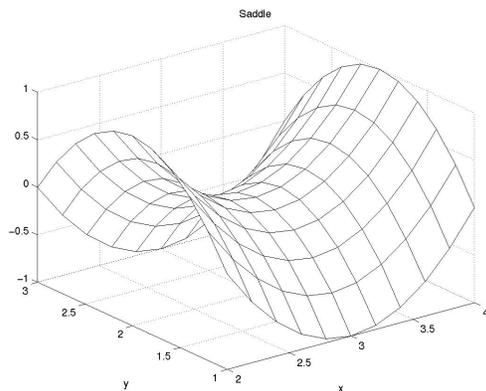

Figure 7: Plot of Saddle function.

Exercise 23.1 *Repeat the previous example replacing* mesh *by* surf *and then by* surfl. *Consult the help pages to find out more about these functions.*

Example 23.2 *Plot the surface defined by the function*

$$f = -xye^{-2(x^2+y^2)}$$

on the domain $-2 \leq x \leq 2, -2 \leq y \leq 2$. Find the values and locations of the maxima and minima of the function.

```
>> [X,Y] = meshgrid(-2:.1:2,-2:.2:2);
>> f =  -X.*Y.*exp(-2*(X.^2+Y.^2));
>> figure (1)
>> mesh(X,Y,f), xlabel('x'), ylabel('y'), grid
>> figure (2),  contour(X,Y,f)
>> xlabel('x'), ylabel('y'), grid, hold on
```
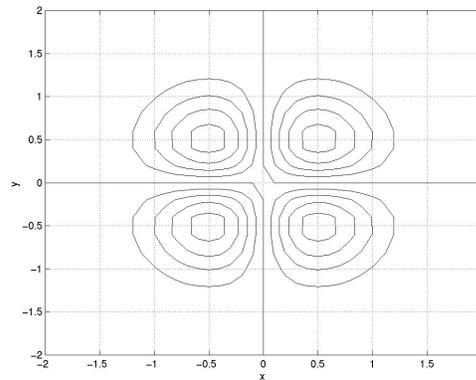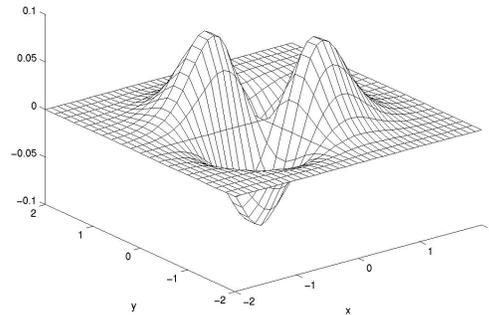

Figure 8: "`mesh`" and "`contour`" plots.

To locate the maxima of the "`f`" values on the grid:

```
>> fmax = max(max(f))
fmax =
    0.0886
>> kmax = find(f==fmax)
kmax =
    323
    539
>> Pos = [X(kmax), Y(kmax)]
Pos =
   -0.5000    0.6000
    0.5000   -0.6000
>> plot(X(kmax),Y(kmax),'*')
>> text(X(kmax),Y(kmax),' Maximum')
```

24   Timing

Matlab allows the timing of sections of code by providing the functions `tic` and `toc`. `tic` switches on a stopwatch while `toc` stops it and returns the CPU time
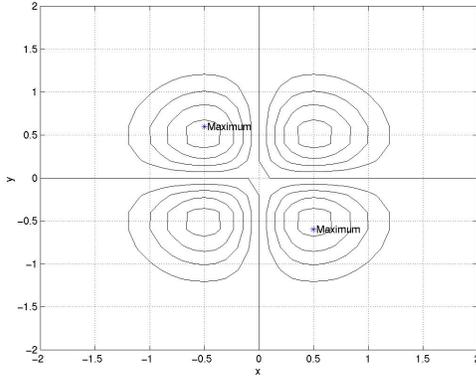
28

Figure 9: `contour` plot showing maxima.

(Central Processor Unit) in seconds. The timings will vary depending on the model of computer being used and its current load.

```
>> tic,for j=1:1000,x = pi*R(3);end,toc
elapsed_time =      0.5110
>> tic,for j=1:1000,x=pi*R(3);end,toc
elapsed_time =      0.5017
>> tic,for j=1:1000,x=R(3)/pi;end,toc
elapsed_time =      0.5203
>> tic,for j=1:1000,x=pi+R(3);end,toc
elapsed_time =      0.5221
>> tic,for j=1:1000,x=pi-R(3);end,toc
elapsed_time =      0.5154
>> tic,for j=1:1000,x=pi^R(3);end,toc
elapsed_time =      0.6236
```

25   On–line Documentation

In addition to the on–line help facility, there is a hypertext browsing system giving details of (most) commands and some examples. This is accessed by

```
>> doc
```

which brings up the *Netscape* document previewer (and allows for "surfing the internet superhighway"—the World Wide Web (WWW). It is connected to a worldwide system which, given the appropriate addresses, will provide information on almost any topic).

Words that are underlined in the browser may be clicked on with LB and lead to either a further subindex or a help page.

Scroll down the page shown and click on general which will take you to "General Purpose Commands"; click on clear. This will describe how you can clear a variable's value from memory.

You may then either click the "Table of Contents" which takes you back to the start, "Index" or the Back button at the lower left corner of the window which will take you back to the previous screen.

To access other "home pages", click on Open at the bottom of the window and, in the "box" that will open up, type

```
http://www.maths.dundee.ac.uk
```

or

```
http://www.maths.dundee.ac.uk/software/
```

and select Matlab from the array of choices.

26   Reading and Writing Data Files

Direct input of data from keyboard becomes impractical when

- the amount of data is large and
- the same data is analysed repeatedly.

In these situations input and output is preferably accomplished via data files. We have already described in Section 9 the use of the commands `save` and `load` that, respectively, write and read the values of variables to disk files.

When data are written to or read from a file it is crucially important that a correct data format is used. The data format is the key to interpreting the contents of a file and must be known in order to correctly interpret the data in an input file. There a two types of data files: formatted and unformatted. Formatted data files uses format strings to define exactly how and in what positions of a record the data is stored. Unformatted storage, on the other hand, only specifies the number format.

The files used in this section are available from the web site

```
http://www.maths.dundee.ac.uk/software/#matlab
```

Those that are unformatted are in a satisfactory form for the Windows version on Matlab (version 6.1) but not on Version 5.3 under Unix.

Exercise 26.1 *Suppose the numeric data is stored in a file 'table.dat' in the form of a table, as shown below.*

```
100   2256
200   4564
300   3653
400   6798
500   6432
```

The three commands,

```
>> fid = fopen('table.dat','r');
>> a = fscanf(fid,'%3d%4d');
>> fclose(fid);
```

respectively

1. open a file for reading (this is designated by the string 'r'). The variable `fid` is assigned a unique integer which identifies the file used (a file identifier). We use this number in all subsequent references to the file.

2. read pairs of numbers from the file with file identifier `fid`, one with 3 digits and one with 4 digits, and

3. close the file with file identifier `fid`.

This produces a **column vector a** with elements, 100 2256 200 4564 ...500 6432. This vector can be converted to 5×2 matrix by the command
`A = reshape(2,2,5)';`.

26.1   Formatted Files

Some computer codes and measurement instruments produce results in formatted data files. In order to read these results into Matlab for further analysis the data format of the files must be known. Formatted files in ASCII format are written to and read from with the commands `fprintf` and `fscanf`.

> `fprintf(fid, 'format', variables)` writes variables an a format specified in string 'format' to the file with identifier fid

> `a = fscanf(fid, 'format',size)` assigns to variable `a` data read from file with identifier `fid` under format 'format'.

Exercise 26.2 *Study the available information and help on fscanf and fprintf commands. What is the meaning of the format string, '%3d\n'?*

Example 26.1 *Suppose a sound pressure measurement system produces a record with 512 time – pressure readings stored on a file 'sound.dat'. Each reading is listed on a separate line according to a data format specified by the string, '%8.6f %8.6f'.*

A set of commands reading time – sound pressure data from 'sound.dat' is,

Step 1: Assign a namestring to a file identifier.

> `>> fid1 = fopen('sound.dat','r');`

The string 'r' indicates that data is to be read (not written) from the file.

Step 2: Read the data to a vector named 'data' and close the file,

> `>> data = fscanf(fid1, '%f %f');`
> `>> fclose(fid1);`

Step 3: Partition the data in separate time and sound pressure vectors,

> `>> t = data(1:2:length(data));`
> `>> press = data(2:2:length(data));`

The pressure signal can be plotted in a `lin-lin` diagram,

> `>> plot(t, press);`

The result is shown in Figure 10.
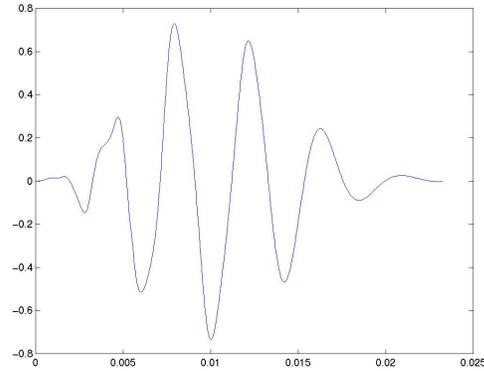
Figure 10: Graph of "sound data" from Example 26.1

26.2   Unformatted Files

Unformatted or binary data files are used when small-sized files are required. In order to interpret an unformatted data file the data precision must be specified. The precision is specified as a string, e.g., `'float32'`, controlling the number of bits read for each value and the interpretation of those bits as character, integer or floating point values. Precision `'float32'`, for instance, specifies each value in the data to be stored as a floating point number in 32 memory bits.

Example 26.2 *Suppose a system for vibration measurement stores measured acceleration values as floating point numbers using 32 memory bits. The data is stored on file 'vib.dat. The following commands illustrate how the data may be read into Matlab for analysis.*

Step 1: Assign a file identifier, `fid`, to the string specifying the file name.

> `>> fid = fopen('vib.dat','rb');`

The string `'rb'` specifies that binary numbers are to be read from the file.

Step 2 Read all data stored on file 'vib.dat' into a vector vib.

> `>> vib = fread(fid, 'float32');`
> `>> fclose(fid);`
> `>> size(vib)`
> `ans =`
> `       131072`

The `size(vib)` command determines the size, i.e., the number of rows and columns of the vibration data vector.

In order to plot the vibration signal with a correct time scale, the sampling frequency (the number of instrument readings taken per second) used by the measurement system must be known. In this case it is known to be 24000 Hz so that there is a time interval of 1/24000 seconds between two samples.

30

Step 3: Create a column vector containing the correct time scale.

```
>> dt = 1/24000;
>> t = dt*(1:length(vib))';
```

Step 4: Plot the vibration signal in a `lin-lin` diagram

```
>> plot(t,vib);
>> title('Vibration signal');
>> xlabel('Time,[s]');
>> ylabel('Acceleration, [m/s^2]');
```

27   Graphic User Interfaces

The efficiency of programs that are used often and by several different people can be improved by simplifying the input and output data management. The use of Graphic User Interfaces (GUI), which provides facilities such as menus, pushbuttons, sliders etc, allow programs to be used without any knowledge of Matlab. They also provides means for efficient data management.
A graphic user interface is a Matlab script file customized for repeated analysis of a specific type of problem. There are two ways to design a graphic user interface. The simplest method is to use a tool especially designed for the purpose. Matlab provides such a tool and it is invoked by typing 'guide' at the Matlab prompt. Maximum flexibility and control over the programming is, however, obtained by using the basic user interface commands. The following text demonstrates the use of some basic commands.

Example 27.1 *Suppose a sound pressure spectrum is to be plotted in a graph. There are four alternative plot formats; lin-lin, lin-log, log-lin and log-log. The graphic user interface below reads the pressure data stored on a binary file selected by the user, plots it in a lin-lin format as a function of frequency and lets the user switch between the four plot formats.*

We use two m–files. The first (`specplot.m`) is the main driver file which builds the graphics window. It calls the second file (`firstplot.m`) which allows the user to select among the possible `*.bin` files in the current directory.

```
%  File: specplot.m
%
% GUI for plotting a user selected frequency spectrum
% in four alternative plot formats, lin-lin,
% lin-log, log-lin and log-log.
%
% Author: U Carlsson, 2001-08-22

      % Create figure window for graphs
figWindow = figure('Name','Plot alternatives');
      % Create file input selection button
fileinpBtn = uicontrol('Style','pushbutton',...
   'string','File','position',[5,395,40,20],...
   'callback','[fdat,pdat] = firstplot;');
      % Press 'File' calls function 'firstplot'
```

```
% Create pushbuttons for switching between four
% different plot formats. Set up the axis stings.
X = 'Frequency, [Hz]';
Y = 'Pressure amplitude, [Pa]';
linlinBtn = uicontrol('style','pushbutton',...
   'string','lin-lin',...
   'position',[200,395,40,20],'callback',...
   'plot(fdat,pdat);xlabel(X);ylabel(Y);');
linlogBtn = uicontrol('style','pushbutton',...
   'string','lin-log',...
   'position',[240,395,40,20],...
   'callback',...
   'semilogy(fdat,pdat);xlabel(X);ylabel(Y);');
loglinBtn = uicontrol('style','pushbutton',...
   'string','log-lin',...
   'position',[280,395,40,20],...
   'callback',...
   'semilogx(fdat,pdat);xlabel(X);ylabel(Y);');
loglogBtn = uicontrol('style','pushbutton',...
   'string','log-log',...
   'position',[320,395,40,20],...
   'callback',...
   'loglog(fdat,pdat);xlabel(X);   ylabel(Y);');

% Create exit pushbutton with red text.

exitBtn = uicontrol('Style','pushbutton',...
   'string','EXIT','position',[510,395,40,20],...
   'foregroundcolor',[1 0 0],'callback','close;');
```

```
% Script file: firstplot.m
% Brings template for file selection. Reads
% selected filename and path and plots
% spectrum in a  lin-lin diagram.
% Output data are frequency and pressure
% amplitude vectors: 'fdat' and  'pdat'.
% Author: U Carlsson, 2001-08-22


function [fdat,pdat] = firstplot

% Call Matlab function 'uigetfile' that
% brings file selction template.

[filename,pathname] = uigetfile('*.bin',...
   'Select binary data-file:');
% Change directory
cd(pathname);
% Open file for reading binary floating
% point numbers.
fid = fopen(filename,'rb');
data = fread(fid,'float32');
% Close file
fclose(fid);
% Partition data vector in frequency and
% pressure vectors
pdat = data(2:2:length(data));
fdat = data(1:2:length(data));
% Plot pressure signal in a lin-lin diagram
plot(fdat,pdat);
% Define suitable axis labels
xlabel('Frequency, [Hz]');
ylabel('Pressure amplitude, [Pa]');
```

Executing this GUI from the command line
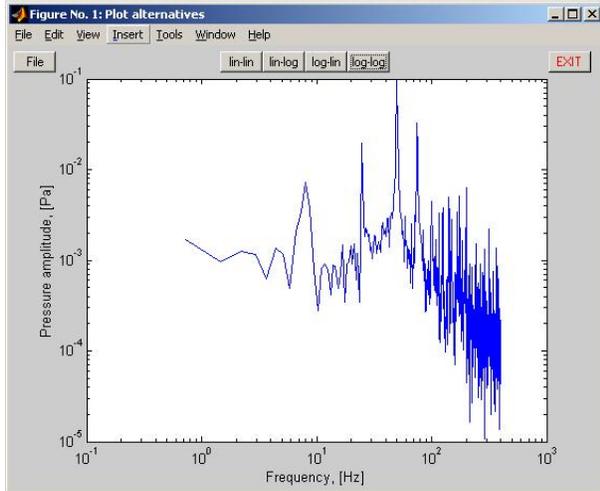(`>> specplot`) brings the following screen.

Figure 11: Graph of "vibration data" from Example 27.1

Example 27.1 illustrates how the '`callback`' property allows the programmer to define what actions should result when buttons are pushed etc. These actions may consist of single Matlab commands or complicated sequences of operations defined in various subroutines.

Exercise 27.1 *Five different sound recordings are stored on binary data files, `sound1.bin`, `sound2.bin`, . . ., `sound5.bin`. The storage precision is '`float32`' and the sounds are recorded with sample frequency 12000 Hz. Write a graphic user interface that, opens an interface window and*

- *lets the user select one of the five sounds,*

- *plots the selected sound pressure signal as a function of time in a lin-lin diagram,*

- *lets the user listen to the sound by pushing a 'SOUND' button and finally*

- *closes the session by pressing a 'CLOSE' button.*

28 Command Summary

The command
`>> help`
will give a list of categories for which help is available (e.g. `matlab/general` covers the topics listed in Table 3. Further information regarding the commands listed in this section may then be obtained by using:
`>> help` *topic*
try, for example,
`>> help help`

`abs`	Absolute value
`sqrt`	Square root function
`sign`	Signum function
`conj`	Conjugate of a complex number
`imag`	Imaginary part of a complex number
`real`	Real part of a complex number
`angle`	Phase angle of a complex number
`cos`	Cosine function
`sin`	Sine function
`tan`	Tangent function
`exp`	Exponential function
`log`	Natural logarithm
`log10`	Logarithm base 10
`cosh`	Hyperbolic cosine function
`sinh`	Hyperbolic sine function
`tanh`	Hyperbolic tangent function
`acos`	Inverse cosine
`acosh`	Inverse hyperbolic cosine
`asin`	Inverse sine
`asinh`	Inverse hyperbolic sine
`atan`	Inverse tan
`atan2`	Two–argument form of inverse tan
`atanh`	Inverse hyperbolic tan
`round`	Round to nearest integer
`floor`	Round towards minus infinity
`fix`	Round towards zero
`ceil`	Round towards plus infinity
`rem`	Remainder after division

Table 2: Elementary Functions

Managing commands and functions.	
`help`	On-line documentation.
`doc`	Load hypertext documentation.
`what`	Directory listing of M-, MAT- and MEX-files.
`type`	List M-file.
`lookfor`	Keyword search through the HELP entries.
`which`	Locate functions and files.
`demo`	Run demos.
Managing variables and the workspace.	
`who`	List current variables.
`whos`	List current variables, long form.
`load`	Retrieve variables from disk.
`save`	Save workspace variables to disk.
`clear`	Clear variables and functions from memory.
`size`	Size of matrix.
`length`	Length of vector.
`disp`	Display matrix or text.
Working with files and the operating system.	
`cd`	Change current working directory.
`dir`	Directory listing.
`delete`	Delete file.
`!`	Execute operating system command.
`unix`	Execute operating system command & return result.
`diary`	Save text of MATLAB session.
Controlling the command window.	
`cedit`	Set command line edit/recall facility parameters.
`clc`	Clear command window.
`home`	Send cursor home.
`format`	Set output format.
`echo`	Echo commands inside script files.
`more`	Control paged output in command window.
Quitting from MATLAB.	
`quit`	Terminate MATLAB.

Table 3: General purpose commands.

Matrix analysis.	
`cond`	Matrix condition number.
`norm`	Matrix or vector norm.
`rcond`	LINPACK reciprocal condition estimator.
`rank`	Number of linearly independent rows or columns.
`det`	Determinant.
`trace`	Sum of diagonal elements.
`null`	Null space.
`orth`	Orthogonalization.
`rref`	Reduced row echelon form.
Linear equations.	
`\ and /`	Linear equation solution; use "help slash".
`chol`	Cholesky factorization.
`lu`	Factors from Gaussian elimination.
`inv`	Matrix inverse.
`qr`	Orthogonal- triangular decomposition.
`qrdelete`	Delete a column from the QR factorization.
`qrinsert`	Insert a column in the QR factorization.
`nnls`	Non–negative least- squares.
`pinv`	Pseudoinverse.
`lscov`	Least squares in the presence of known covariance.
Eigenvalues and singular values.	
`eig`	Eigenvalues and eigenvectors.
`poly`	Characteristic polynomial.
`polyeig`	Polynomial eigenvalue problem.
`hess`	Hessenberg form.
`qz`	Generalized eigenvalues.
`rsf2csf`	Real block diagonal form to complex diagonal form.
`cdf2rdf`	Complex diagonal form to real block diagonal form.
`schur`	Schur decomposition.
`balance`	Diagonal scaling to improve eigenvalue accuracy.
`svd`	Singular value decomposition.
Matrix functions.	
`expm`	Matrix exponential.
`expm1`	M- file implementation of expm.
`expm2`	Matrix exponential via Taylor series.
`expm3`	Matrix exponential via eigenvalues and eigenvectors.
`logm`	Matrix logarithm.
`sqrtm`	Matrix square root.
`funm`	Evaluate general matrix function.

Table 4: Matrix functions—numerical linear algebra.

Graphics & plotting.	
`figure`	Create Figure (graph window).
`clf`	Clear current figure.
`close`	Close figure.
`subplot`	Create axes in tiled positions.
`axis`	Control axis scaling and appearance.
`hold`	Hold current graph.
`figure`	Create figure window.
`text`	Create text.
`print`	Save graph to file.
`plot`	Linear plot.
`loglog`	Log-log scale plot.
`semilogx`	Semi-log scale plot.
`semilogy`	Semi-log scale plot.
Specialized X-Y graphs.	
`polar`	Polar coordinate plot.
`bar`	Bar graph.
`stem`	Discrete sequence or "stem" plot.
`stairs`	Stairstep plot.
`errorbar`	Error bar plot.
`hist`	Histogram plot.
`rose`	Angle histogram plot.
`compass`	Compass plot.
`feather`	Feather plot.
`fplot`	Plot function.
`comet`	Comet-like trajectory.
Graph annotation.	
`title`	Graph title.
`xlabel`	X-axis label.
`ylabel`	Y-axis label.
`text`	Text annotation.
`gtext`	Mouse placement of text.
`grid`	Grid lines.
`contour`	Contour plot.
`mesh`	3-D mesh surface.
`surf`	3-D shaded surface.
`waterfall`	Waterfall plot.
`view`	3-D graph viewpoint specification.
`zlabel`	Z-axis label for 3-D plots.
`gtext`	Mouse placement of text.
`grid`	Grid lines.

Table 5: Graphics & plot commands.

Index