An Introduction to Matlab

Version 4.1

David F. Griffiths

Associate Member
Mathematics Division
The University of Dundee
Dundee DD1 4HN
Scotland, UK

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

Preface

These notes have evolved considerably since the originals were used to teach a postgraduate course on Numerical Analysis and Programming at the University of Dundee in around 1991. I recall that the students in those early years who had previous computing experience found Matlab nothing short of revolutionary. Up until that time numerical computation was carried out using languages such as Fortran and Algol and had been a laborious process. The codes had to be compiled and run, and the data transferred to different software if graphics were involved. To quote Hageman from his invited talk to the SIAM National meeting in 1975:

> The problem here is that numerical experimentation is costly and time consuming. It is doubtful if one individual has the time or expertise to consider all numerical alternatives that should be investigated.

(As reported by Fox in The State of the Art in Numerical Analysis, 1976). How times have changed. I, along with most others, now take the immediacy of Matlab for granted.
The computing environment and computer literacy have changed considerably over the years and have been major factors influencing this new edition of these notes. Gone are many features that I believe have become redundant and in their place is, for example, material on management of the Matlab Desktop and accompanying editor (Appendices A–D). I would recommend starting with the short Appendix A before returning to §1. The remaining appendices can be dipped into when further information is required.
The other main additions include a

- section on formatting numeric output,
- case study comparing ten ways of computing the Fibonacci sequence,
- section of extended examples,
- section containing 27 graded exercises,
- greater use of graphics and their properties,
- more extensive index.

This means, of course, an increase in the page-length. However, the basics can still be found in the first 48 pages. This introduction to Matlab (using Release R2018a) is designed for self-study but would be enhanced by a one-off class to orientate the novice.

Thanks to Dr Anil Bharath, Imperial College,
Dr Chris Gordon, University of Christchurch,
Prof. Dr Markus Rottmann, HSR Hochschule Für Technik, Switzerland,
for their contributions to this and earlier versions.

Contents

1 MATLAB

- Matlab is an interactive system for performing numerical computations.
- Cleve Moler wrote the first version of Matlab as a teaching aid in the 1970s. It has since evolved into an invaluable tool in all areas of scientific computation.
- Matlab relieves us of the tedium of arithmetical calculations and so allows more time for thought and experimentation.
- Powerful operations can be performed using just one or two commands.
- The graphics facilities are excellent and the results can readily be inserted into LaTeX or Word documents.

The Matlab interface, where commands are typed and files edited, is described in Appendices A–D. We recommend starting these notes by referring to Appendix A.
These notes provide only a brief glimpse of the power and flexibility of Matlab, for a more comprehensive view we recommend the book by Des & Nick Higham [4].

2 Matlab as a Calculator

The basic arithmetic operators are `+ - * / ^` and these are used in conjunction with parentheses (round) brackets: `( )`. The caret symbol `^` is used to get exponents (powers): `2^4=16`. Square brackets `[ ]` (§7) and curly braces `{ }` have special meanings in Matlab.
Commands are typed at 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 parentheses,
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 `2 + (3/4)*5` by priority 3.

Exercise 2.1 *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) `1*1-1^1+1/1-1`
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)`

3 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-02` $= -1.3412 \times 10^{-2} = -0.013412$
All computations in Matlab are performed in double precision, which means about 15 significant figures (the default is numbers of type `double`). How numbers are printed is controlled by the "`format`" command. Type

```
>> help format
```

for a full list. Typing `format` on its own will switch back to the default format.

Command	Example of Output
`>>format short`	`31.4159` (4 decimal places)
`>>format long`	`31.41592653589793`
`>>format short e`	`3.1416e+01`
`>>format long e`	`3.141592653589793e+01`
`>>format bank`	`31.42` (2 decimal places)
`>>format rat`	`3550/113`

(`rat` is short for rational number, i.e., a fraction.)

The command

```
>> format compact
```

is highly recommended. It suppresses blank lines in the output allowing more information to be displayed in the command window.

4 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` and `y` have the values -13 and -65, respectively, which 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.

4.1 Suppressing output

If the result of intermediate calculation doesn't need to be seen the assignment statement or expression should be terminated with a semi–colon:

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

the value of `x` is hidden. Observe that several statements can be placed on a line, separated by commas or semi–colons.

4.2 Variable Names

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

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

There is a distinction between upper and lower case characters, so `X3` and `x3` refer to different variables. These are **not** allowable:

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

Use names that reflect the values they represent. Among the names to avoid are
`pi = 3.14159...` $= \pi$
and `eps` (which has the value `2.2204e-16`$= 2^{-54}$, the largest number such that `1 + eps` is indistinguishable from 1).
There is potential for conflict if a variable name coincides with that of a Matlab function (see page 41 for an example).
The command `iskeyword` will list Matlab keywords. These cannot be used for variable names.

5 Complex numbers

Arithmetic with complex numbers can be carried out using `i` or `j`, both of which have the value $\sqrt{-1}$ at startup (these startup values are often over-ridden since `i` and `j` are popular names of integer variables that index vectors or matrices).

```
>> i, j, i = 3, z1 = 2+i, z2 = 2+1i
ans =
   0.0000 + 1.0000i
ans =
   0.0000 + 1.0000i
i =
     3
z1 =
     5
z2 =
   2.0000 + 1.0000i
```

Note the use of `2+1i` (no `*`) which ensures correct usage even after a value has been assigned to `i`. The real and imaginary parts of a complex number can be extracted:

```
>> real(z2), imag(z2)
ans =
     2
ans =
     1
```

The command `disp` prints the value of a quantity without displaying its name:

```
>> disp(z)
    2   +    1i
>> disp([2+i 2+1i])
    5   +    0i           2   +  1i
```

both are printed as complex numbers even though the first is real. See Section 7.4 for more on complex numbers.

6 Built–In Functions

We can only give a cursory view of the extensive list of functions available in Matlab. As well as the Help browser (which can be summoned by the button 4 in Fig. 29), help is available from the command line prompt. Type `help help` for a brief synopsis of the help system or `help` for a list of topics. The first few lines of this are

```
HELP topics:       -
                   -
MatlabCode/matlab  - (No table of contents file)
matlab/general     - General purpose commands.
matlab/ops         - Operators and special ...
matlab/lang        - Programming language ...
matlab/elmat       - Elementary matrices ...
matlab/randfun     - Random matrices and ...
matlab/elfun       - Elementary math funct...
matlab/specfun     - Specialized math...
```

(truncated lines are shown with ...). Clicking on a key word, for example `sin` will provide further information together with a link to `doc sin` which provides the most extensive documentation along with examples of its use. Alternatively, type

```
>> help elfun
```

for instance, to obtain help on "Elementary math functions".
The `lookfor` command is useful if you don't know the precise name of a function. For example,

```
>> lookfor integral
```

returns a list of all functions that have "integral" in their first (header) line. See Exercise 15.2 (page 26) for another example of its use.

6.1 Trigonometric Functions

All standard trig functions `sin, cos, tan,...` have been preprogrammed in Matlab—their arguments should be in radians. For example, to calculate the coordinates of a point on a circle of radius 5 centred at the origin and having an elevation $30° = \pi/6$ radians:

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

For angles measured in degrees, use `sind`, `cosd`, `tand`.... 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
```

Use `asind, acosd, atand,...` to obtain the result in degrees.

6.2 Other Elementary Functions

These include `sqrt, exp, log, log10`

```
>> x = 9, sqrt(x), sqrt(x^2+2*x+1)
x =
     9
ans =
     3
ans =
    10
```

For other powers (exponents) use `^`

```
>> pi^2-2^pi
ans =
    1.0446
```

`exp(x)` denotes the exponential function e^x: this seems to give a remarkable result

```
>> A = 20/(exp(pi)-pi)
A =
    1.0000
```

but, inspecting more decimal places,

```
>> format long
>> A
A =
   1.000045003065711
>> format short
```

reveals it to be a near miss. The inverse function of `exp` is `log`:

```
>> exp(log(pi)), log(exp(pi))
ans =
    3.1416
ans =
    3.1416
```

For logs to the base 10 use `log10`. A more complete list of elementary functions is given in Table 1 on page 73.

7 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
```

Linear combinations can be formed from vectors of the same length (the operations are carried out elementwise). With `v` and `v3` defined above:

```
>> 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.
```

the error is due to `v` and `v2` having different lengths.
New row vectors can be built from existing ones:

```
>> w  = [1 2 3];  z = [8 9];
>> cd = [2*z, -w], sort(cd)
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.
The value of particular entries can be inspected or changed

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

There are two exceptions to addition being between vectors of the same length. The first is when adding a scalar to a vector—this adds the scalar to each component:

```
>> 2 + w
ans =
     3     0     5
```

```
>> w - 3*w(1)
ans =
    -2    -5     0
```

The second exception is described in §7.4.

7.1 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 `[]`.

```
>> 7:-2:0
ans =
     7     5     3     1
>> 0.32:0.1:0.6
ans =
    0.3200    0.4200    0.5200
```

See also `linspace` on page 12.

7.2 Extracting Parts of Vectors

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

To extract 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.
The last element in a vector can be found by using the reserved word `end`:

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

7.3 Column Vectors

These have similar constructs to row vectors except that entries are separated by `;`

```
>> c = [ 1; 3; sqrt(5)]
c =
    1.0000
    3.0000
    2.2361
```

or "newlines"

```
>> 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.
The `length` command does not distinguish between row and column vectors:

```
>> length(c)
ans = 3
>> length(r5)
ans = 7
```

Compare with `size` described in §15.1. Adding a scalar to a column vector adds the scalar to each of its components.

7.4 Transposing

A row vector can be converted into a column vector (and vice versa) by a process called *transposing*, which is 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
```

What happens if a row vector is added to a column vector? This is the second exception alluded to on page 6 and is known as "implicit expansion" (Higham [5]):

```
>> w, z
w =
       1    2    3
z =
       4    5
>> w + z'
ans =
       5    6    7
       6    7    8
```

the entry in the ith row and jth column is `w(i)+z(j)`. We shall make use of this later.
When `x` is a complex vector (see page 4), `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
```

To obtain the plain transpose of a complex number use `.'` (as in `x.'`)

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

This might be an opportune time to visit Appendix B in order to get further features of the Desktop.

8 Keyboard Accelerators

Previous Matlab commands can be reviewed in the Command Window by using the ↑ and ↓ cursor keys, the most recent command being displayed first. When the desired command is reached it can be re-executed by pressing 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). The arrow keys ← and → can used to move backwards and forwards through the line, characters may be inserted by typing at the current cursor position or deleted using the `Delete` key. When the command is in the required form, press return.
This process is most commonly used when long command lines have been mis-typed or when it is necessary to execute a command that is very similar to one used previously.
The following (emacs) commands are also available:

`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 right of the cursor
`cntrl k`	delete from cursor to end of line

Exercise 8.1 *Type in the commands*

```
>> t = pi/6; R = 7;
>> x = R*cos(t);,y = R*sin(t)
>> L = sqrt(x^2+y^2)
```

Edit these commands with the cursor keys to execute:

```
>> t = pi/6; R = 7;
>> x = R*cosh(t), y = R*sinh(t)
>> L = sqrt(x^2-y^2)
```

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 folder in which Matlab was invoked. Any legal filename may be used *except* the names `on` and `off`. If the file already exists then the new information will be appended (rather than overwriting the contents).
The record is terminated by

```
>> diary off
```

The file `mysession` will appear in the "Current Folder" pane (on the left hand side in Fig. 30) and may be edited with your favourite editor (the Matlab editor is recommended—see Appendix C) to remove any mistakes or superfluous material.

10 Script Files

Script files are ordinary ASCII (text) files that contain Matlab commands. It is obligatory that such files have names with a `.m` extension (e.g., `sample.m`) and, for this reason, they are commonly known as *m-files*. We first use the command

```
>> which sample
'sample' not found.
```

to confirm that there is no variable in the current session or any Matlab function of that name, thus reducing possible confusion. Note that the command `what` lists all the Matlab files in the current folder. A miscellaneous selection of commands have been typed into the file

`sample.m` (the Matlab editor—see Appendix C—is recommended for creating, running and debugging files).
The contents of `sample.m` are:

```
%SAMPLE Miscellaneous examples.
%    Commented text in the header
%    gives a description of the file
[10^3, exp(7), 2^10]
a = 20/(exp(pi)-pi)
format long, a
format short
%% Second section
% a new vector
x = 10*[cos(pi/3), sind(30)] % xxxxxxx
asind(x/10)
% z = 10*[cosd(30), sind(60) tand(45)]
```

(see also Fig. 32 in Appendix C). The first few lines are comments, each beginning with `%`, whose purpose is to allow descriptive comments to be included in order to assist the human reader. The format of the first line is particularly important:
`%SAMPLE Miscellaneous examples.`
It contains the file name in upper case followed by a brief description of the purpose of the file. The leading comment lines—up to the first executable statement—also contribute to the `help` system. For example,

```
>> help sample
 sample Miscellaneous examples.
    Commented text in the header
    gives a description of the file
```

and the file name is rendered in a lower case bold font.
Lines beginning with exactly two `%%` start a new section and are followed by the section title. These have a special significance in the Matlab editor (Appendix C).
The commands in the file may then be executed using

```
>> sample
```

(without the `.m` extension). Ther commented line
`% z = 10*[cosd(30), sind(60) tand(45)]`
at the end of the file which will not be executed since it starts with a `%`. It is a common strategy to comment out line(s) of code, particularly when testing a script file, in order to locate errors.

Exercise 10.1
Type the following commands into a file `rampi.m`

```
%RAMPI Approximations to pi.
%  Many of the formulae were
%  published by Ramanujan in 1914.
api = [ 22/7; 355/113; sqrt(10);
  19*sqrt(7)/16;
  7*(1+sqrt(3)/5)/3;
  9801*sqrt(2)/4412;
  (9^2+19^2/22)^0.25;
  693/80/(7-3*sqrt(2));
  log(640320^3+744)/sqrt(163)];
format long
[api, pi-api]
```

Now issue the commands

1. `what` *to list the m-files in the current folder,*
2. `help rampi` *to see its effect,*
3. `type rampi` *to view the contents of the file in the "Command Window"*
4. `rampi` *to execute the file*

at the prompt (`>>`). Rank the given formulae according to how well they approximate π.

It is only the output from the commands in a script file (and not the commands themselves) that are displayed in the command window. Typing

```
>> echo on
```

prior to their execution will show the commands as well; `echo off` will turn echoing off. Compare the effect of

```
>> echo on, rampi, echo off
```

with the earlier results. The `echo` commands may also be placed inside a script file.
The related topic of function files will be discussed in §25.

11 Arithmetic with Vectors

11.1 Inner Product (∗)

There are two ways of attributing a meaning to the product of two vectors in Matlab. In both cases the vectors concerned must have the same length.

The first product is the standard inner product: corresponding elements are multiplied together and the results added to give a single number. 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, then, in mathematical notation

$$\underline{u} = [u_1, \ldots, u_n], \ \underline{v} = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix}, \ \underline{u}\,\underline{v} = \sum_{i=1}^{n} u_i v_i.$$

In Matlab

```
>> u = [10, -11,  12]; % row vector
>> v = [20; -21; -22]; % column vector
>> prod = u*v % row times column vector
prod =
   167
```

An error results if both are row (or both column) vectors

```
>> w = [2, 1, 3], u*w
w =
     2     1     3
??? Error using ==> *
Inner matrix dimensions must agree.
```

One way of avoiding this sort of problem is to convert all vectors to column vectors. This is easily achieved:

```
>> u(:)
ans =
    10
   -11
    12
```

In fact, `u(:)` returns a column vector regardless of whether `u` started life as a row or column vector, in contrast with transposing (page 7) which turns column vectors into row vectors and vice versa. Thus, the inner product

```
>> u(:)'*w(:)
ans =
    45
```

is error-free regardless of whether `u` or `w` are row or column vectors.

The Euclidean length of a vector is an example of 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. Two possible ways of computing it are:

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

where `norm` is a built-in Matlab function. It has options to compute other norms: `help norm`.

Exercise 11.1 *The angle, θ, between two row vectors $\underline{x}$ and $\underline{y}$ is defined by*

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

($\underline{y}'$, the transpose of $\underline{y}$ is a column vector). Use this formula to determine the cosine of the angle between $\underline{x} = (1, 2, 3)$ and $\underline{y} = (3, 2, 1)$. Hence show that the angle is 44.4153degrees.

11.2 Elementwise Product (`.*`)

The second way of forming a product of two vectors of the same length is known as the Hadamard product. It is rarely used in the course of normal mathematical calculations but is an invaluable Matlab feature. It involves vectors of the same type. If $\underline{u}$ and $\underline{v}$ are both row vectors or both column vectors, the mathematical definition of this product, called the **Hadamard product**, is the **vector** having the components

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

That is, the product of the corresponding elements of the two vectors resulting in a vector of the same length and type as the originals. Summing the entries in the resulting vector would give their inner product.
In Matlab, the product is computed with the operator `.*`

```
>> u = [10, -11,  12];  w = [2, 1, 3];
>> u.*w
ans =
      20 -11 36
>> u(:).*w(:)
ans =
    20
   -11
    36
```

A common use of the Hadamard product is in the evaluation of mathematical expressions so that they may be tabulated or plotted.

Example 11.1 *Tabulate the function* $y = x \sin \pi x$ *for* $x = 0, 0.25, \ldots, 1$.

We first create a column of of x-values:

```
>> x = (0:0.25:1)';
```

and, to evaluate y we multiply each element of the vector x by the corresponding element of the vector $\sin \pi x$:

```
>>  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 `pi*x`). Thus, the Hadamard product of these is also a column vector.

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

Exercise 11.2 *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. 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.

11.3 Elementwise Division (./)

In Matlab, the operator `./` is defined to give element by element division of one vector by another—it is therefore only defined for vectors of the same size and type.

```
>> a = -2:2, b = 1:5, a./b
a =
   -2    -1     0     1     2
b =
    1     2     3     4     5
ans =
 -2.0000  -0.5000   0  0.2500   0.4000
```

or, changing to `format rat` (short for rational),

```
>> format rat
>> a./b
ans =
 -2    -1/2     0      1/4      2/5
```

and the output is displayed in fractions. The `./` operation is also needed to compute a scalar divided by a vector:

```
>> 5/b
Error using  /
Matrix dimensions must agree.
>> 5./b
ans =
 5   5/2   5/3   5/4  1.0000
```

so `5./b` is legal, but `5/b` is not. Decimal and rational formats deal with division by zero in different ways:

```
>> b./a
ans =
-1/2     -2     1/0     4     5/2
>> a./a
ans =
1     1     0/0     1     1
>> format short % switch formats
```

```
>> b./a
ans =
-0.5000  -2.0000  Inf  4.0000  2.5000
>> a./a
ans =
1     1   NaN     1     1
```

A non-zero divided by zero gives `Inf` (denoting infinity) and `0/0` gives `NaN` (`Not a Number`).

Example 11.2 *Estimate the limit*

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

The idea is to observe the behaviour of the ratio $\sin(\pi x)/x$ 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
```

which reveals an interesting pattern.

11.4 Elementwise Powers (`.^`)

The dot-power operator `.^` applies the same power to each element of a vector:

```
>> u = [10, 11, 12]; u.^2
ans =
   100   121   144
>> u.*u
ans =
   100   121   144
>> ans.^(1/2)
ans =
    10    11    12
>> u.*w.^(-2)
ans =
    2.5000  -11.0000    1.3333
```

Fractional and decimal powers are allowed. Recall that powers are carried out before any other arithmetic operation.
When the base is a scalar and the power is a vector we get:

```
>> n = 0:4
n =
     0     1     2     3     4
>> 2.^n
ans =
     1     2     4     8    16
```

and, when both are vectors of the same size,

```
>> x = 1:3:15
x =
    1    4    7    10    13
>> x.^n
ans =
    1     4    49  1000    28561
```

12 Plotting Functions

In order to plot the graph of a function, $y = \sin 3\pi x$ for $0 \le x \le 1$, say, it is sampled at a sufficiently large number of points and the points (x, y) joined by straight lines. Suppose we take $N + 1$ sampling points equally spaced a distance h apart:

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

defines the set of points $x = 0, h, 2h, \ldots, 1-h, 1$ with $h = 0.1$. Alternately, we may use the command `linspace`. The general form of the command is `linspace (a,b,n)` which generates n equispaced points between a and b, inclusive. So, in this case we would use the command

```
>> x = linspace (0,1,N+1);
```

The corresponding y values are computed by

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

and finally, the points are plotted with

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

The result seen on the left of Fig. 1 clearly has too small a value of N and this is changed to $N = 100$ for the righthand graph.

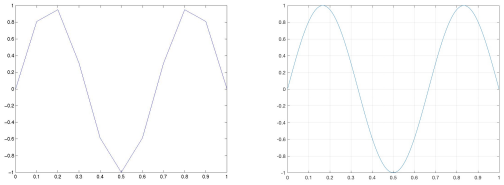


Fig. 1: Graph of $y = \sin 3\pi x$ for $0 \le x \le 1$ using $N = 10$ (left) and $N = 100$ (right) data points.

The command "`grid on`" draws a grid of dotted lines at each of the tick-marks on the axes. It is removed with "`grid off`" and toggled with `grid`.

12.1 Plotting—Titles & Labels

To include a title and to label the axes:

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

The arguments of the commands must be *strings*, i.e., characters enclosed in single quotes (see §18). Some simple LaTeX commands are available for formatting mathematical expressions and Greek characters (see Section 12.10).
See also `ezplot` the "Easy to use function plotter".

12.2 Line Styles & Colours

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

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

The third argument is a string comprising characters that specify the colour (red), the line style (dashed) and the symbol (`x`) to be drawn at each data point. The order in which they appear is unimportant and any, or all, may be omitted. The options for colours, styles and symbols include:

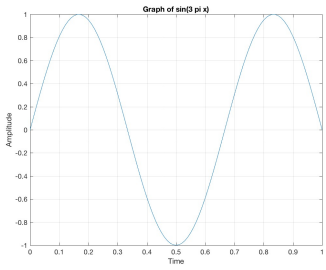


Fig. 2: Graph with title and axes labels.

	Colours		Line Styles/symbols
`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. `help plot` will provide a full list.
The command `clf` clears the current figure while `close(1)` will close the graphics window labelled "Figure 1"; `close all` will close all graphics windows. 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 next plotting commands will be directed to it.

12.3 Multi–plots

Several graphs may be drawn on the same figure as in

```
>> plot(x,y,'k-', 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 appear in the plot command, followed by the brief description provided in the command.
For further information use `help plot` etc.
The result of the commands

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

is shown in Fig. 3. The legend may be moved either manually by dragging it with the mouse or as described in `help legend`.

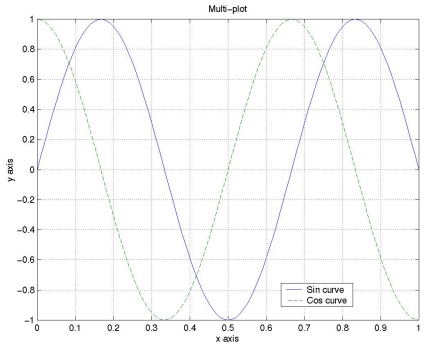


Fig. 3: Graph of $y = \sin 3\pi x$ and $y = \cos 3\pi x$ for $0 \le x \le 1$ using $h = 0.01$.

12.4 Hold

A call to `plot` clears the graphics window before plotting the next 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, 'r-'), hold on
>> plot(x, y.^2, 'g.'), 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.

12.5 Hard Copy

A printed copy may be obtained by selecting Print from the File menu on the Figure toolbar or by issuing the command `print`. The command
`>> print -f2`
will print figure 2 on the default printer.
Alternatively, a figure may be saved to a file for later printing, editing or including in a report or similar document. To do this a format and a filename must by supplied.

```
print -f4 -djpeg figb
```

The characters following the option `-d` specify the format, in this case `jpeg`, and figure 4 will be saved in the file `figb.jpg`. Among the other options are
`-depsc` for "Encapsulated Color PostScript"
`-dpdf` for "Portable Document Format".

12.6 Subplot

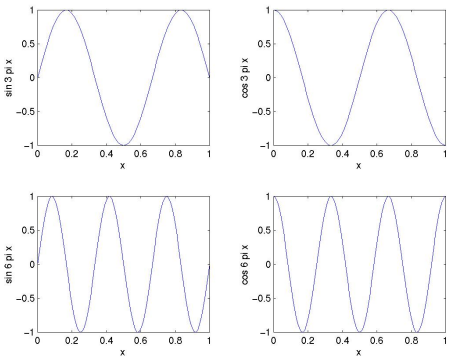
The graphics window may be split into an $m \times n$ array of smaller windows into each of which we may plot one or more graphs. The windows are Numbered 1 to mn row–wise, starting from the top left. Both `hold` and `grid` work on the current subplot.

```
>> 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.

12.7 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.
The coordinates of point(s) on a figure may be be obtained using `ginput`. The command `ginput(3)`, say, will show "crosshairs" on the current figure and return the coordinates of the next three points clicked on with the mouse.

Exercise 12.1 *Draw graphs of the functions*

$$y = \cos x \text{ and } y = x$$

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

12.8 Controlling Axes

It is sometimes necessary to change the axes on a plot in order to get the effect we are looking for—we use the multipurpose command `axis`.
For example, the following code places $N = 100$ points on the circumference in order to draw a circle of radius 6.

```
>> N = 100; t = (1:N)*2*pi/N;
>> x = 6*cos(t); y = 6*sin(t);
>> plot(x,y,'-');
```

The result shown in the left of Fig. 4 is clearly non-circular. This is due to the use of different scales on the horizontal and vertical axes. This is corrected in the image in the centre by using the command `axis` with the `equal` option:

```
>> axis equal
```

This has an alternative form `axis('equal')` which allows more than one option to be specified:

```
>> axis('equal','off')
```

The second option here switches off display of the axes and the result is shown in the rightmost figure. The axes can be reinstated with `>> axis on`.
We recommend looking at `help axis` and experimenting with the commands `axis equal`, `axis off`, `axis square`, `axis normal`, `axis tight` in any order.

12.9 Plot Properties

The properties of a plot can be edited from the Figure window by selecting the Edit menu from the toolbar. For instance, to change the linewidth of a graph, click Edit and choose `Figure Properties...` from the menu. Clicking on the required curve will display its attributes which can be readily modified.
One of the shortcomings of editing the figure window in this way is the difficulty of reproducing the results at a later date. The recommended alternative involves using commands that directly control the graphics properties. Saving these commands in a script file will enable the figure to be reproduced at any later stage.

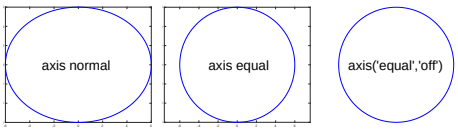


Fig. 4: Options to the `axis` command applied to a circular image

The current setting of any plot property can be determined by first obtaining its "handle", which is saved to a named variable—we choose `ph` here (short for plot handle):

```
>> ph = plot([0 3 3 0],[0 0 4 0],'k-o')
ph =
  Line with properties:
              Color: [0 0 0]
          LineStyle: '-'
          LineWidth: 0.5000
             Marker: 'o'
         MarkerSize: 6
    MarkerFaceColor: 'none'
              XData: [0 3 3 0]
              YData: [0 0 4 0]
              ZData: [1x0 double]
  Show all properties
>> axis equal
```

The result is shown on the left of Fig. 5. The plot has many more attributes that may be seen by clicking on "`all properties`". The colour is described by a rgb triple in which `[0 0 0]` denotes black, `[1 1 1]` denotes white and `[c,c,c]` (with $0 < c < 1$) is used to depict different shades of grey.
There are two ways that the properties can be changed. The simplest is in the plot command itself, for example
`>> plot([2 3],[2 2],'linewidth',2)`
will draw a line from (2,2) to (3,2) with a width of 2pts. The alternative is to use the `set` command, for example
`>> set(ph,'markersize',15)`
will change the size of the marker symbol ('o' in this case). The arguments to `set` are a handle followed by pairs which take the form of a property name in single quotes(here `'markersize'`) followed by its new value. The names of an attribute can be any mixture of upper and lower case and they need not be spelt out in full—only so far as to make their names unique. For example,

```
>> set(ph,'Ydata',[0 1 3 0],...
    'linewi',2,...
    'markerfaceco','r',...
    'color',[0 0 1])
```

where the ellipsis (three periods) `...` signify long commands that continue on the next line.

```
>> ph
ph =
  Line with properties:
              Color: [0 0 1]
          LineStyle: '-'
          LineWidth: 2
             Marker: 'o'
         MarkerSize: 15
    MarkerFaceColor: [1 0 0]
              XData: [0 3 3 0]
              YData: [0 1 3 0]
              ZData: [1x0 double]
  Show all properties
```

Several attributes have been changed including "`Ydata`", the y-coordinates of the data; the results are shown in the right of Fig. 5.
It will be seen that the font size of the numbers along the axes has also been adjusted. This was done via the plot handle of the current axes—this is always `gca`. Typing `gca` will give an abridged list of its properties:

```
>> gca
ans =
  Axes with properties:
          XLim: [-0.4018 3.4018]
          YLim: [0 3]
        XScale: 'linear'
        YScale: 'linear'
GridLineStyle: '-'
Position: [0.1300 0.1100 0.7750 0.8150]
         Units: 'normalized'
  Show all properties
```

Here `Xlim` and `Ylim` give the limits of the plotting area, `XScale` and `YScale` show that the

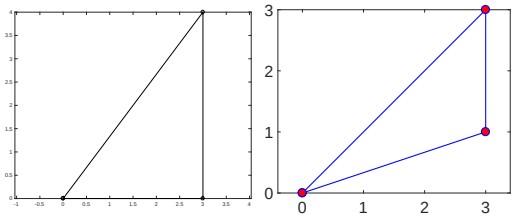


Fig. 5: A plot before (left) and after (right) its properties have been adjusted

scales on both axes are linear (the alternative is 'log' for logarithmic scales). The Position refers to the positioning of the plotting area in the window.
The command get(gca) will give a full list of attributes and set(gca) will also list the options that are available. get can also be used to determine the current values of individual attributes, for example,

```
>> get(gca,'fontsize')
ans =
    10
>> get(gca,'xtick')
ans =
   0  0.50  1.00  1.50  2.00  2.50  3.00
```

showing the current font size (in pts) and the locations of the tick marks on the x-axis. Choosing a larger font size with
`>> set(gca,'fontsize',30)`
automatically adjusts the tick marks to accommodate the larger characters:

```
>> get(gca,'xtick')
ans =
     0     1     2     3
```

12.10 Text on Plots

The command text is available to place text on a plot. For example, the command
`>> text(-3,0,'axis normal','fontsi',40)`
was used to print the text on Fig. 4 (Left). The arguments to the function are the coordinates of where the text should start, the text itself (in single quotes) and this is followed up by changing the font size to be used. The text in Fig. 4 (right) is more demanding to produce since it contains quote symbols. Two single quotes are required in the string to produce one quote character in the output.

```
>> T = text(-5,0,'axis(''equal'',...
''off'')', 'fontsi',40)

T =
Text(axis('equal','off'))with properties:

              String: 'axis('equal','off')'
            FontSize: 40
          FontWeight: 'normal'
            FontName: 'Helvetica'
               Color: [0 0 0]
  HorizontalAlignment: 'left'
            Position: [-5 0 0]
               Units: 'data'

  Show all properties
```

Use of a handle causes selected properties of the text to be listed.
It is possible to typeset simple mathematical expressions (using LaTeX commands) in labels, legends, axes and text. We shall give two illustrations.

Example 12.1 *Plot the first 100 terms in the sequence* $\{y_n\}$ *given by* $y_n = \left(1 + \frac{1}{n}\right)^n$ *and illustrate how the sequence converges to the limit* $\mathrm{e} = \exp(1) = 2.7183....$ *as* $n \to \infty$.

Exercises such as this require a certain amount of experimentation (with font sizes, for example) which is best carried out by saving the commands in a script file:

```
%LATEXPLOT Illustration of LaTeX text
%First set defaults
set(groot,'defaultaxesfontsize',16)
set(groot,'defaulttextfontsize',20)
set(groot,'defaulttextinterpreter','latex')
    N = 100; n = 1:N;
    y = (1+1./n).^n;
subplot(2,1,1)
    plot(n,y,'.','markersize',8)
    hold on
    axis([0 N,2 3])
    plot([0 N],[1, 1]*exp(1),'--')
    text(40,2.4,'$y_n = (1+1/n)^n$')
    text(10,2.8,'$y = e$')
    xlabel('$n$'), ylabel('$y_n$')
```

The results are shown in the upper part of Fig. 6.

The salient features of these commands are

1. The set commands in lines 2–3 increase the size of the default font size for all subsequent axis labels, legends, titles and text. groot returns the graphics root object and contains all the figures that exist.

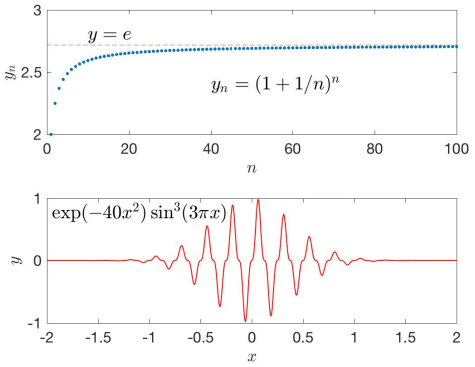


Fig. 6: The output from Example 12.1 (top) and Example 12.2 (bottom).

The fourth line instructs Matlab to interpret any strings contained within `$...$` symbols as LaTeX commands.

2. Defining a variable $N = 100$ makes it easier to experiment with a different number of sampling points.

3. The size of the plot symbol "." is changed from the default (6) to size 8 by the additional string in the `plot` command.

4. The `axis` command changes the dimensions of the plotting area to be $0 \le x \le N$ and $2 \le y \le 3$.

   The argument to `axis` is a vector of length four; the first two elements are the minimum and maximum values of x followed by the minimum and maximum values of y.

5. The `text` command prints text (in this case a mathematical expression enclosed in `$...$`) on the plot as described at the start of this section. The string `y_n` gives subscripts: y_n, while `x^3` gives superscripts: x^3. For x^{n+1} use `x^{n+1}` with curly braces.

For more information on LaTeX we (strongly) recommend Griffiths & Higham [3]. This contains instructions on including graphics files in a LaTeX document.

Example 12.2 *Draw a graph of the function* $y = \mathrm{e}^{-3x^2}\sin^3(3\pi x)$ *on the interval* $-2 \le x \le 2$.

Assuming that the default values from the previous example continue to operate:

```
subplot(2,1,2)
x = -2:.01:2;
y = exp(-3*x.^2).*sin(8*pi*x).^3;
plot(x,y,'r-','linewidth',1)
xlabel('$x$'), ylabel('$y$')
text(-1.95,.75,...
      '$ \exp(-40x^2)\sin^3(8\pi x)$')
print  -djpeg eplot1
```

The results are shown in the lower part of Fig. 6.

1. $\sin^3 8\pi x$ is typeset by the LaTeX string `$\sin^3 8\pi x$` and translates into the Matlab command `sin(8*pi*x).^3`—the position of the exponent is different.

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

3. The thickness of the line used in the `plot` command is changed from its default value (0.5) to 2.

4. The graphics are saved in jpeg format to the file `eplot1`.

Presenting graphical output usually requires considerable experimentation. This is greatly facilitated by the command `unplot`, written by Toby Driscoll [2]. It is (unfortunately) not part of the Matlab distribution but can be downloaded from the MathWorks File Exchange. The effect of `unplot` is to remove the most recent graphics object (line, text, etc.), while `unplot(n)` removes the `n` most recent objects.

13 The `startup` File

When MATLAB starts it executes the file `startup.m` if it can be found on its search path. It is usually saved in a folder "MATLAB" within "Documents". This file can be used to modify the default settings for graphics commands. The following example shows some possibilities.

```
%STARTUP Set defaults on startup
set(0,'DefaultLineLinewidth',1);
```

```
set(0,'DefaultLineMarkerSize',10);
set(0,'DefaultTextFontsize',16);
set(0,'DefaultAxesFontsize',16);

set(groot,...
    'DefaultTextInterpreter','Latex')
set(groot,...
 'DefaultAxesTickLabelInterpreter',...
 'Latex');
set(groot,...
    'DefaultLegendInterpreter','Latex');

format compact
```

14 Further Plot Examples

Example 14.1 *Draw a graph of the function*

$$y = \frac{1}{2-x} + \frac{2}{3x-10} + x^{2/3}\sqrt{x-6}$$

for $0 \le x \le 10$.

Suitable commands (which are saved in a file `plotex.m`) might be:

```
%PLOTEX
figure(1)
x = 0:.1:10;  %101 sample points
y = 1./(x-2) + 2./(3*x-10) ...
     + x.^(2/3).*sqrt(x-6);
plot(x,y,'b-')
grid on
set(gca,'fontsi',20)
Xt = [1.95, 2.15, 3.3, 3.4, 6;
       -10,   10, -20,  11, 0];
T = ['A';'B';'C';'D';'E'];
text(Xt(1,:),Xt(2,:),T)
print -djpeg plotex
```

Note:

1. the repeated use of the "dot" (element-wise) operators in the definition of `y`.
2. the first term in the function has a singularity at $x = 2$. This is a sampling point (`x(21)=2`) and, as a consequence, `y(21)=Inf`. This point is ignored by the plot command and there is a gap between the points A and B in Fig. 7.

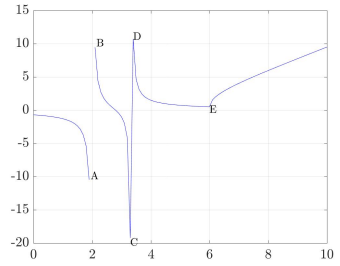


Fig. 7: Output for Example 14.1

3. the singularity in the second term does not occur at a sampling point and so the points C and D are joined together.
4. the function y is complex for $x < 6$ and, when the script is executed we get a warning:

```
>> plotex
Warning: Imaginary parts of complex
      X and/or Y arguments ignored
> In plotex (line 6)
```

5. the curve has a vertical tangent at E which is not well represented.

Example 14.2 *The speed of computers is measured in* flops*—floating point operations/second. The data in the following script contains the flop rates for the fastest supercomputers over the period 1950–2000 taken from an article by Dongarra et al. [1]. The rates vary enormously and a conventional plot of the data would contain little information. However, using a logarithmic scale on the y-axis (the* `semilogy` *command) is very revealing.*

```
%FLOPS Supercomputer speeds 1950-2000
%    Data from SIAM News:
%    Volume 34, No. 9, November 2001
speed = [7e2,9e2,6e4,1e6,6e6,6e6,...
    1e8,1e9,2e9,1e10,1.3e11,...
    1.3e11,1.3e12,3e12];
X = [1950,1951,1960:5:1970 1971,...
    1976,1982,1985,1987,1991,...
    1992,1998,2000];
semilogy(X,speed,'ro',...
    X,10.^(2.85+.194*(X-1950)),'k-')
```

```
title('Supercomputer speeds')
ylabel('Flops')
xlabel('Year'), grid on
set(gca,'fontsi',20)
print -djpeg flops
```

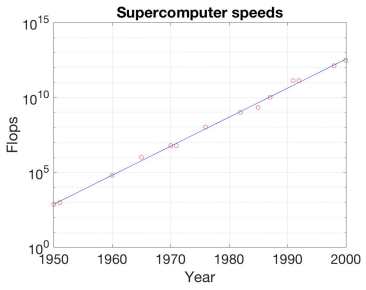


Fig. 8: Flop rates for Example 14.2

The equation of the straight line through the data is approximately (see Example 16.1)

$$\log_{10}(\text{speed}) \approx 2.85 + 0.194x$$

where $x = \text{year} - 1950$. Moore's Law follows from this: Computer speed doubles roughly every 18months ($10^{0.194\times 1.5} \approx 1.95$).

Example 14.3 *The data given in the script* `bolt.m` *shown below contains the split times for Usain Bolt's 2009 World record 100m sprint[8].*

```
%%BOLT Split times for Bolt's 2009 WR
%over 100m taken from SpeedEndurance.com
S = [1.89 .99 .9 .86 .83 .82 .81 ...
      .82 .83 .83];
subplot(3, 2, [1 3])
plot( [0 cumsum(S)], 0:10:100, '.-')
grid on
xlabel( 'Time (s)' )
ylabel( 'Distance (m)')
subplot( 3, 2, [2, 4])
bar( 5:10:95, 10./S)
xlabel( 'Time (s)' )
ylabel( 'Speed (m/s)' )
set(gca, 'xtick', 0:20:100)
print -djpeg bolt
```

The 1×10 array `S` contains the times taken to travel each of the 10m sections of the race. To graph the time taken as a function of distance we use `cumsum` which forms the cumulative sum, i.e., `cumsum(S)` gives
`[S(1), S(1)+S(2), S(1)+S(2)+S(3), ...]`
The subplots create a 3×2 array of plotting areas counted row-wise from the top left. The leftmost plot in Fig. 9 is drawn in the 1st and 3rd of these, the rightmost plot in the 2nd and 4th. This alters the aspect ratio of the plots. The bar chart on the right (whose tick-marks on the x-axis have been changed with the `set` command) shows that Bolt reaches speeds of 12m/s.

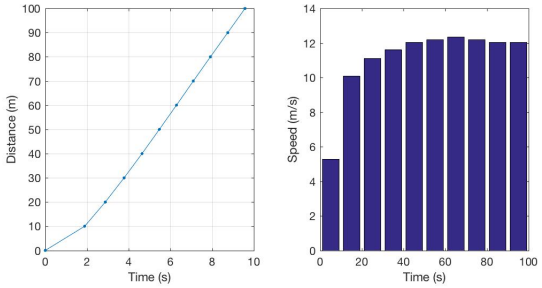


Fig. 9: Output for Example 14.3

15 Two–Dimensional Arrays

A rectangular array of numbers having m rows and n columns is referred to as an $m \times n$ matrix. It is usual in a mathematical setting to enclose such objects in either round or square brackets—Matlab insists on 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
>> 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.
The generic term "array" is used to include both vectors ($m \times 1$ or $1 \times n$) and matrices.

15.1 Size of a matrix

The size (dimensions) of a matrix can be obtained with the command `size`

```
>> size(A), x = [5 3 0]; 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 command `size(ans)` shows that the *value* returned by `size` is itself a 1×2 matrix (a row vector). The results can be saved for subsequent calculations:

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

If only the number of rows (columns) is required the tilde `~` can be used as a place-holder:

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

Arrays can be reshaped. A simple example is:

```
>> A(:)
ans =
     5
     1
     7
    -3
     9
    -7
```

which converts `A` into a column vector by stacking its columns on top of each other. This could also be achieved using `reshape(A,6,1)`. The command

```
>> reshape(A,3,2)
ans =
     5    -3
     1     9
     7    -7
```

also redistributes the elements of `A` columnwise. Linear combination of matrices, for example `2*A+3*B`, can only be carried out if `A` and `B` are of the same size.
The exceptions—further examples of "implicit expansion" (§7.4)—occur when `B` is a scalar, a row vector with the same number of rows as `A` or a column vector with the same number of columns as `A`.

```
>> A
A =
     5     7     9
     1    -3    -7
>> A + 3           % add a scalar
ans =
     8    10    12
     4     0    -4
>> A + (10:12)     % add a row vector
ans =
    15    18    21
    11     8     5
>> A + (10:11)'    % add a column vector
ans =
```

```
    15    17    19
    12     8     4
```

15.2 Transpose of a matrix

Transposing a vector changes it from a row to a column vector and vice versa (see §7.4)—recall that is also performs the conjugate of complex numbers. 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.

```
>> A, A'
A =
     5     7     9
     1    -3    -7
ans =
     5     1
     7    -3
     9    -7
>> size(A), size(A')
ans =
     2     3
ans =
     3     2
```

15.3 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 a matrix may be built based on the size of an existing one.

A square $n \times n$ matrix is said to be **symmetric** if it is equal to its transpose:

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

Alternatively,

```
>> issymmetric(S)
ans =
  logical
   1
```

where a "logical" result `ans = 1` is returned, signifying "true" (see §21).

15.4 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), eye(2,3)
I =
     1     0     0
     0     1     0
     0     0     1
ans =
     1     0     0
     0     1     0
```

so, with two arguments, the `eye` command produces a non-square matrix.

15.5 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 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 size is large. The `diag` function then becomes essential. This command behaves differently depending on whether its argument is a vector or a matrix.
If the diagonal entries are placed (in order) in a vector `d`, say, then `diag(d)` gives:

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

A second argument can be used with `diag`

```
>> e = [1 1]; d = 3*ones(1,3);
>> T = diag(e,-1) + diag(d)-diag(2*e,1)
T =
     3    -2     0
     1     3    -2
     0     1     3
```

and specifies in which "diagonal" the vector entries are to be placed: `-1` is just below the leading diagonal and `1` is just above. Matrices such as T, where the values are constant along diagonals, are called *Toeplitz* matrices and can be constructed with the command `toeplitz`. For example,

```
>> toeplitz([3 -1 0 0], [3 2 0 0])
ans =
     3     2     0     0
    -1     3     2     0
     0    -1     3     2
     0     0    -1     3
```

The arguments supply, respectively, the first column and first row of the matrix. When the leading elements of the two arguments differ, the diagonal entries are supplied by the first argument. When only one argument is supplied the matrix is assumed to be symmetric.
When `A` is a matrix, the command `diag(A)` extracts its diagonal entries. For the (non-square) matrix in §15.2

```
>> diag(A)
ans =
     3
    -3
```

This is a vector so can itself be the argument to `diag`:

```
>> diag(diag(A))
ans =
     3     0
     0    -3
```

15.6 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
H =
     5     7     9
     1    -3    -7
    -1     2     5
     9     0     5
```

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

```
>> J = [1:4; 5:8; 9:12; 2 0 5 4]
J =
     1     2     3     4
```

```
     5     6     7     8
     9    10    11    12
     2     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     2     0     5     4
     1     5     9     2     0     0     0     0
     2     6    10     0     0     0     0     0
     3     7    11     4     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
```

A large matrix composed of tilings of a smaller matrix can be built with `repmat`. Here are two examples:

```
>> repmat( (1:3)', 1, 4 )
ans =
     1     1     1     1
     2     2     2     2
     3     3     3     3
```

a 1×4 replication of a 3×1 vector, and

```
>> A = [ 1:3; 4:6 ]
A =
     1     2     3
     4     5     6
>> B = repmat(A, 3, 2)

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

which is a 3×2 tiling by the matrix A.

Example 15.1 *Build a 4×4 Hilbert matrix H whose entries are $H_{i,j} = 1/(i + j - 1)$.*

This can be done in an elegant way using "implicit expansion" (see page 7):

```
>> I = 1:4; format rat; H = 1./(I'+I-1)
H =
     1       1/2     1/3     1/4
     1/2     1/3     1/4     1/5
     1/3     1/4     1/5     1/6
     1/4     1/5     1/6     1/7
>> format short
```

See also `help hilb`. Some other special matrices can be built with:

`compan`	`hankel`	`spiral`
`givens`	`invhilb`	`vander`
`hadamard`	`pascal`	`wilkinson`

15.7 Tabulating Functions

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

Example 15.2
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
```

A more direct approach would use:

```
>> [ x 4*sin(3*x) 3*sin(4*x) ]
```

15.8 Extracting Parts of Matrices

Sections may be extracted from a matrix in much the same way as for a vector (page 6). 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, using the value given earlier,

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

Individual entries can be changed:

```
>> J(4, 1) = J(4, 1) - 2*J(1, 3)
J =
     1     2     3     4
     5     6     7     8
     9    10    11    12
    -4     0     5     4
```

The colon (§7.1) is crucial in selecting parts or complete rows and columns. In the following examples (i) the 3rd column, (ii) the 2nd and 3rd columns, (iii) the 4th row, and (iv) the "central" 2×2 matrix are extracted.

```
>> 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
>> % To get rows 2 to 3 & cols 2 to 3:
>> J(2:3, 2:3)
ans =
     6     7
    10    11
```

The `:` on its own refers to the entire column or row depending on whether it is the first or the second index. To change an entire row:

```
>> J(2,:) = J(2,:)-5*J(1,:)
J =
     1     2     3     4
     0    -4    -8   -12
     9    10    11    12
    -4     0     5     4
```

The keyword "`end`" can also be used with multidimensional arrays

```
J(1:2, end-1:end )
ans =
    3   4
    7   8
```

Exercise 15.1 *Use suitable row operations to reduce* `J` *to the form*

$$\begin{matrix} 1 & 2 & 3 & 4 \\ 0 & -4 & -8 & -12 \\ 0 & * & * & * \\ 0 & * & * & * \end{matrix}$$

where $*$ denote quantities to be determined.

15.9 Elementwise Products (`.*`)

The elementwise product works as for vectors: corresponding elements are multiplied together—so the matrices involved must have the same size. With `A`, `B` and `C` from §15.6,

```
>> 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
```

Elementwise powers `.^` and division `./` work in an analogous fashion.

15.10 Matrix–Matrix Products

The entry in the ith row and jth column of the product of an $m \times n$ matrix A and a $n \times p$ matrix B is the inner product of the ith row of A with the jth column of B. The result is an $m \times p$ matrix:

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

```
>> 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
>> A*x
??? Error using ==> *
Inner matrix dimensions must agree.
>> 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'`, i.e., `(A*B)' = B'*A'`
Redefining `J` from page 23 and with

```
>> M = eye(4) ...
      - [ [0; J(2:end,1)] zeros(4,3) ]
M =
     1     0     0     0
    -5     1     0     0
    -9     0     1     0
    -2     0     0     1
>> M*J
ans =
     1     2     3     4
     0    -4    -8   -12
     0    -8   -16   -24
     0    -4    -1    -4
```

which reveals the answer to Exercise 15.1.

Exercise 15.2 *It is often necessary to factorize a matrix, e.g., $A = BC$ or $A = S^T XS$ where the factors are required to have specific properties. Use the* `lookfor` *command to make a list of factorization commands in Matlab.*

Exercise 15.3 *If C denotes the Cholesky factorization of the matrix H of Example 15.1 verify that $C^T C = H$.*

15.11 Sparse Matrices

Matlab has powerful techniques for handling sparse matrices — these are generally large matrices (to make the extra book-keeping involved worthwhile) that have only a very small proportion of non–zero entries. Our examples are necessarily small due to space constraints.

Example 15.3 *Create a sparse 5×4 matrix S having only 3 non–zero values: $S_{1,2} = 10$, $S_{3,3} = 11$ and $S_{5,4} = 12$.*

Three vectors are created containing, respectively, the i–indices, the j–indices and the corresponding values of each nonzero term. These are then processed by 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`.

Often the non-zeros in a sparse matrix form a pattern—the most common of which is that of a *banded* matrix, where the non-zeros are confined to a (relatively small) number of diagonals above and below the leading diagonal.

A pattern such as that shown below has non-zeros on two diagonals below and three diagonals above the main diagonal.

The command `spdiags` (short for sparse diagonals) is designed to facilitate the construction of such matrices. Suppose we wish to construct an $m \times n$ banded matrix containing p non-zero diagonals. The four arguments to `spdiags` are

`V:` a matrix with p columns (which will form the diagonals) and at least $\max(m, n)$ rows,

`i:` a vector of length p which identifies which diagonals to fill. The main diagonal is numbered "0", those below the main diagonal have a negative index and those above, a positive index.

`r, c:` the number of rows and columns, respectively, in the final matrix.

For example `V` $= [a, b, c, d, e]$ (where a, b, etc., are column vectors, so `V` is a matrix with five columns), `i` $= -2 : 2$ and `r,c` $= 5$ would describe the matrix in the shaded region:

e_1				
d_1	e_2			
c_1	d_2	e_3		
b_1	c_2	d_3	e_4	
a_1	b_2	c_3	d_4	e_5
	a_2	b_3	c_4	d_5
		a_3	b_4	c_5
			a_4	b_5
				a_5

The six values d_1, e_1, e_2, a_4, a_5 and b_5 outside the shaded region are ignored.

```
>> n = 5;
>> b = (1:n)'; c = flipud(b); d = -b;
>> B = spdiags([ b c d],[-1 0 2], n, n);
>> full(B)
ans =
     5     0    -3     0     0
     1     4     0    -4     0
     0     2     3     0    -5
     0     0     3     2     0
     0     0     0     4     1
```

The command `flipud` (flip up-down) reverses the entries in a column vector. More generally `flipud` reverses the order of rows in a matrix (two dimensional array), while `fliplr` (flip left-right) reverses the order of columns.

The `spdiags` command can also be used to deconstruct a sparse matrix:

```
>> [V, i] = spdiags(B)
V =
     1     5     0
     2     4     0
     3     3    -3
     4     2    -4
     0     1    -5
i =
    -1
     0
     2
```

where the "undefined" values in `V` are returned as zeros. A final use of `spdiags`

```
>> C = ...
   spdiags([b, zeros(n,1)], [1, 2], B);
>> disp( full(C) )
     5     2     0     0     0
     1     4     3     0     0
     0     2     3     4     0
     0     0     3     2     5
     0     0     0     4     1
```

which uses the columns in the first argument to modify the diagonals (specified by the second argument) of the sparse matrix `B`. The result here is a sparse tridiagonal matrix `C`.

Exercise 15.4 *Construct the matrix*

$$U = \begin{bmatrix} 1 & -1 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & 1 & -1 \end{bmatrix}$$

and calculate $S = UU^T$. Extract the non-zero diagonals of S.

16 Solving Linear Equations

The mathematical formulations of many practical problems reduce ultimately to the solution to sets of simultaneous linear equations. The general set of m equations in n unknowns

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

is usually expressed as

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

where A is an $m \times n$ matrix, $\boldsymbol{b}$ a column vector of length m and an (unknown) column vector $\boldsymbol{x}$ of length n .
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). These issues are discussed in most texts on numerical methods, for instance, Trefethen and Bau [14]. Matlab is equipped with many of these special techniques in its routine library and they are usually invoked automatically.
When A is square ($m = n$), the standard Matlab routine uses a version of Gaussian elimination with partial pivoting and is invoked by calling the matrix left-division routine,

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

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

```
>> A = [ 1 -1 0; -1 2 -1; 0 -1 2 ]
A =
     1    -1     0
    -1     2    -1
     0    -1     2
>> b = [0; 1; 0];
>> x = A \ b
x =
    2
    2
    1
```

Changing the (3,3) entry of `A` results in

```
>> B = A; B(3, 3) = 1
B =
     1    -1     0
    -1     2    -1
     0    -1     1
>> x = B\b
Warning: Matrix is singular
            to working precision.
x =
   NaN
   Inf
   Inf
>> sum(B)
ans =
     0     0     0
```

The sum of rows of `B` is zero showing that they are linearly dependent, meaning that the matrix is singular.

Exercise 16.1 *Use the backslash operator to solve the complex system of equations $C\boldsymbol{x} = \boldsymbol{b}$, where*

$$C = \begin{bmatrix} 2+2\mathrm{i} & -1 & 0 \\ -1 & 2-2\mathrm{i} & -1 \\ 0 & -1 & 2 \end{bmatrix}, \ \boldsymbol{b} = \begin{bmatrix} 1+\mathrm{i} \\ 0 \\ 1-\mathrm{i} \end{bmatrix}.$$

It is also possible to solve non-singular linear systems using the inverse of the matrix A: $\boldsymbol{x} = A^{-1}\boldsymbol{b}$. This is easily implemented in Matlab using `inv(A)` to compute the inverse of `A`, so with `A` and `b` as previously defined,

```
>> inv(A)
ans =
     3     2     1
     2     2     1
```

```
     1     1     1
>> b = [0; 1; 0];
>> x = inv(A)*b
x =
     2
     2
     1
```

The approach is frowned upon when dealing with matrices of any appreciable size since it is so inefficient.

16.1 Overdetermined systems

An overdetermined system of linear equations is 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 $\boldsymbol{r}$, defined by

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

be minimized. The simplest and most frequently used measure of the magnitude of $\boldsymbol{r}$ is the Euclidean length (or norm—see Section 11.1) 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 $\boldsymbol{x}$ that minimizes

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

This is achieved in Matlab using again the left division operator "\". Thus, when the matrix A is not square, the operation

```
x = A\b
```

automatically gives the least squares solution to $A\boldsymbol{x} = \boldsymbol{b}$—the process is often known as regression. An illustration is given in the next example.

Example 16.1 *It was observed in Example 14.2 that the logarithm of the computer speed is approximately a linear function of time (the year). Calculate the equation of the straight line shown in Fig. 8.*

We have to determine the coefficients c, d, say, so that

$$\log_{10}(\text{speed}) = c(X - 1950) + d.$$

There are 14 data values given in Example 14.2 leading to 14 linear equations from which to determine c and d. Thus the matrix `A` is 14×2, the data `b` contains the 14 values of $\log_{10}$(speed) and the unknown vector `x = [c; d]`. Assuming that the variables used in Example 14.2 are still available, we need to add the commands

```
>> A = [ X'-1950  ones(size(X')) ];
>> b = log10(speed');
>> x = A\b
>> x =
      0.1941
      2.8490
```

So that $c = 0.1941$ and $d = 2.8490$.
A Matlab routine `polyfit` provides a streamlined solution when the overdetermined equations result from data fitting by polynomials: see "`help polyfit`".

```
>> P = polyfit(X-1950,log10(speed),1)
P =
    0.1941    2.8490
>> x = [1950, 2000];
>> y = polyval(P, x-1950);
>> semilogy(X,speed,'ro', x,10.^y,'b-')
```

`polyfit` requires three arguments: the first two provide the x and y data values and the third the degree of polynomial to fit to the data. If the degree of the polynomial is n, say, and the length of x and y is $(n + 1)$ then a polynomial will be constructed that exactly passes though each of the data points (so long as the components of x are all different). If the length of x and y is $> (n+1)$, as it is in this example, then the polynomial that best fits in a least squares sense is returned. The output `p` from the command `p = polyfit(x,y,n)` is a row vector of

length $(n+1)$ containing the coefficients of the polynomial:

$p(x) = \texttt{p(1)}x^n + \texttt{p(2)}x^{n-1} + \cdots + \texttt{p(n)}x + \texttt{p(n+1)}$.

The companion command `y = polyval(p, t)` evaluates the polynomial `p` at the points `t`—equivalent to $y = p(t)$.

17 Eigenvalue Problems

Eigenvalues and eigenvectors are key concepts in matrix analysis. If A is an $n \times n$ matrix and the equation

$$A\boldsymbol{x} = \lambda \boldsymbol{x}$$

has a non-zero solution $\boldsymbol{x}$ for some value of λ then $\boldsymbol{x}$ and λ are known as an eigenvector and eigenvalue of A, respectively. The Matlab command for determining these is `eig`. It behaves differently according to the number of outputs requested. With one output requested

```
>> B
B =
     1    -1     0
    -1     2    -1
     0    -1     1
 >> d = eig(B)
 d =
    0.0000
    1.0000
    3.0000
```

a vector is returned containing the eigenvalues (in ascending order when they are all real). With two outputs,

```
>> [ V, D ] = eig(B)
V =
   -0.5774   -0.7071    0.4082
   -0.5774    0.0000   -0.8165
   -0.5774    0.7071    0.4082
D =
    0.0000         0         0
         0    1.0000         0
         0         0    3.0000
>> d = diag(D)
d =
    0.0000
    1.0000
    3.0000
```

The matrices are connected via

$$BV = VD$$

where the columns of `V` contain the eigenvectors. When V is nonsingular (there is a full set of eigenvectors) $B = VDV^{-1}$ or $D = V^{-1}BV$—the diagonalization of B .
The eigenvectors (columns of `V`) are normalised to have unit norm (see page 10)

```
>> sqrt(sum(V.^2))
ans =
    1.0000    1.0000    1.0000
```

In general the eigenvalues in `D` appear to no particular order. If they are to be sorted in ascending order, then the columns of `V` need to be reordered accordingly. This can be done with the commands

```
>> [d, i] = sort( diag(D) );
>> V = V(:, i);
```

where `i` is a vector of indices: `d(j) = D(k, k)`, and `k = i(j)`, the `j`th entry of `i`.

Exercise 17.1 *Verify that the eigenvectors of B are mutually orthogonal—i.e., the inner product of any two distinct columns of `V` is zero. This can be accomplished by a single Matlab command.*

The command `eigs` (note the plural) applies to large sparse matrices and, by default, returns just the six largest eigenvalues with their corresponding eigenvectors. See Example 19.2.

18 Characters, Strings and Text

The ability to process text is useful for prompting for input and annotating output of data to the screen or to files. In order to manage text, a new datatype "character" is introduced. A piece of text is then simply a string (vector or array) of characters contained in single quotes. The assignment,

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

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

```
>> t2 = 'BCDE'
t2 =
   BCDE
>> size(t2)
ans =
     1     4
```

assigns the value `BCDE` to the `1-by-4` character array `t2`. Strings can be combined (concatenated):

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

assigns a value `ABCDE` to the `1-by-5` character array `t3` (note the square brackets). Note that `t3(2) = 'B'` and `t3(2:3) = 'BC'`.
It is often 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.
```

The formatting of output will be further examined in §28.

19 `for` Loops

There are occasions when a segment of code has to be repeated a number of times (such occasions are less frequent than other programming languages because of the `:` notation).
A standard `for` loop has the form

```
>> for counter = 1:20
      .......
   end
```

which repeats the code (indicated by ...) as far as the `end` with the variable `counter=1` the first time, `counter=2` the second time, and so forth. Rather more generally

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

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

Example 19.1 *The Fibonnaci sequence starts with the numbers* 0 *and* 1*, and succeeding terms are the sum of its two immediate predecessors. Mathematically,* $f_0 = 0$*,* $f_1 = 1$ *and*

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

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

The commands are saved in a script file `fib.m`:

```
%%FIB Fibonacci sequence example
% F(n+1) corresponds to f(n)
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','fontsize',20)
legend('Ratio of terms f_{n-1}/f_n')
gr = (sqrt(5)-1)/2;
plot([0 20], gr*[1,1], 'r--')
```

The last of these commands produces the dashed horizontal line seen in Fig. 10. A piece of trivia: if f_n represents a distance in miles, f_{n+1} gives (approximately) the same distance in kilometres. For many other ways of computing this sequence see §30.

Example 19.2 *Calculate and plot the eigenvectors corresponding to the six smallest eigenvalues of the sparse* 100×100 *tridiagonal matrix*

$$S = \begin{bmatrix} 2 & -1 & & & & \\ -1 & 2 & -1 & & & \\ & \ddots & \ddots & \ddots & & \\ & & -1 & 2 & -1 & \\ & & & -1 & 2 & \end{bmatrix}.$$

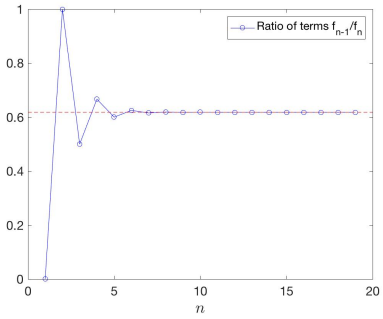


Fig. 10: The ratio of successive terms in the Fibonacci sequence for Example 19.1

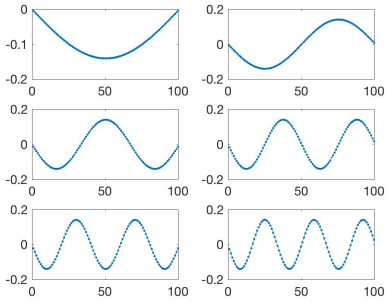


Fig. 11: The eigenvectors for Example 19.2

Note that S has 10,000 entries of which only 298 are nonzero—it is sparse.
The command

```
[ V, D ] = eigs(S, 6);
```

returns the eigenvectors corresponding to the six largest eigenvalues (and eigenvectors) and an extra argument `'sm'` has to be included to calculate the smallest.

```
%%FORLOOP Example of a for loop
n = 100;
S = spdiags(repmat([-1 2 -1],n,1),...
    -1:1,n,n);
% Calculate 6 smallest eigenvalues
[ V, D ] = eigs(S, 6, 'sm');
d = diag(D);
[d, i] = sort(d);
V = V( :, i);
for j = 1:6
    subplot( 3, 2, j)
    plot( V( :, j), '.-')
end
```

The vector `d`, obtained from the diagonal entries of `D`, contains the eigenvalues of `S`. These are sorted into ascending order which means that the columns of `V` must similarly be rearranged (the graphs in Fig. 11 should be familiar to enthusiasts of the *sin* function). The for loop assigns each eigenvector to a different subplot in a 3×2 array. The code within the loop is indented to aid readability.

Exercise 19.1 *Compute and plot the eigenvectors corresponding to the six largest eigenvalues of the matrix S in Example 19.2. Comment on any relationships with those shown in Fig. 11. Verify that the eigenvalues are members of the set* $\left\{4\cos^2(\frac{1}{2}j\pi/(n+1))\right\}_{j=1}^{n}$.

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

```
>> tic, sum((1:10000).^2);toc
Elapsed time is 0.000124 seconds.
>> tic, sum((1:10000).^2);toc
Elapsed time is 0.000047 seconds.
>> tic, s = sum((1:10000).^2);T = toc
T =
   8.2059e-05
```

The first two instances illustrate that there can be considerable variation in successive calls to the same operations (especially for short calculations). The third instance shows that the elapsed time can be assigned to a variable.
An alternative is to use the "Run and Time" button (to the right of ⑤ in Fig. 32) in. the Editor Window. This pops up the "Profiler" window which prompts for the name of a script file and returns a detailed breakdown of the time spent on each component of the script.

21 Logicals

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

`true = 1,  false = 0`

If at some point in a calculation a scalar `x`, say, has been assigned a value, then certain logical tests can be made 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?

Particular attention should be paid to the fact that the test for equality involves two equal signs `==`.

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

Great care should be exercised when comparing two decimal numbers due to the potential influence of rounding errors:

```
>> 4*asin(1/sqrt(2)) == pi
ans =
  logical
   0
>> 4*asin( 1/sqrt(2) ) - pi
ans =
  -4.4409e-16
```

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

```
>> x = [-2 pi 5;-1 0 1]
x =
   -2.0000    3.1416    5.0000
   -1.0000         0    1.0000
>> x == 0
ans =
  2x3 logical array
   0   0   0
   0   1   0
>> y = x>=-1, z = x > y
y =
  2x3 logical array
   0   1   1
   1   1   1
z =
  2x3 logical array
   0   1   1
   0   0   0
```

Logical tests may be combined, as in

```
>> x > 3 & x < 4
ans =
  2x3 logical array
   0   1   0
   0   0   0
>> x > 3 | x < 0
ans =
  2x3 logical array
   1   1   1
   1   0   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 == 0 | x <= -2
ans =
  2x3 logical array
   1   1   1
   0   1   0
```

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

```
>> L = x>=0
L =
  2x3 logical array
   0   1   1
   0   1   1
>> xpos = x.*L
xpos =
         0    3.1416    5.0000
         0         0    1.0000
```

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

Example 21.1 *Plot the half-wave rectified sine curve defined by*

$$y = \begin{cases} \sin \pi x, & \sin \pi x \geq 0 \\ 0, & \sin \pi x < 0. \end{cases}$$

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

The result is shown in Fig. 12.

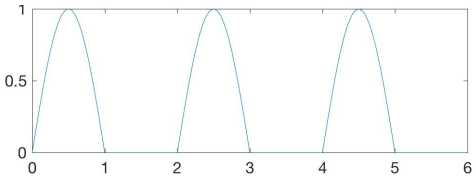


Fig. 12: Half-wave rectified sine curve

22 While Loops

There are occasions when a section of Matlab code needs to be repeated a number of times that cannot be predicted in advance. This is when the `while...end` construct is more appropriate than a "for" loop.

Example 22.1 *What is the greatest value of n that can be used in the sum*

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

and obtain a value that is less than 100?

```
>> S = 1; n = 2;
>> while S + n^2 < 100
       S = S + n^2;  n = n+1;
   end
>> disp( ['n = ', int2str(n-1),...
         ', S = ', int2str(S) ] )
n = 6, S = 91
```

The lines of code between `while` and `end` will only be executed if the condition `S+n^2 < 100` is true.
The next example is a more typical example of `while...end`.

Example 22.2 *Find the approximate value of the root of the equation $x = \cos x$. (See Example 12.1.)*

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, d, between two successive values $|x_n - x_{n-1}|$ is small enough.

First attempt:

```
>> x(1) = pi/4;  n = 0;  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(1:n)
n =
    14
x =
Columns 1 through 6
0.7854 0.7071 0.7602 0.7247 0.7487 0.7326
Columns 7 through 12
0.7435 0.7361 0.7411 0.7377 0.7400 0.7385
Columns 13 through 14
0.7395 0.7388
```

There are a number of deficiencies with this code. The vector `x` stores the results of each iteration but it isn't known in advance how many there may be. This requires the length of `x` to be extended each time around the loop—a notoriously slow process (though it makes little difference here with such a short vector). In any event, we are rarely interested in the intermediate values of `x`, only the last one.
A more serious problem is that we may never satisfy the condition $d \leq 0.001$, in which case the program will run forever unless a limit is placed on the maximum number of iterations. Infinite loops are, unfortunately, a common hazard of computing. Should it occur, the loop can be terminated with the key sequence `ctrl c` (holding the Control key down while typing `c`)[1].
Incorporating these improvements leads to

[1] If an infinite loop is encountered while executing a script file through the Matlab editor, the computation can be stopped via the "Pause" button.

Second attempt:

```
>> xold = pi/4; n = 0;  d = 1;
>> while d > 0.001 &  n < 25
     n = n+1;  xnew = cos(xold);
     d = abs( xnew - xold );
     xold = xnew;
   end
>> [n, xnew, d]
ans =
   13.0000    0.7388    0.0007
```

We continue around the loop so long as $d > 0.001$ **and** $n < 25$. For greater precision the condition $d > 0.0001$ gives

```
>> [n, xnew, d]
ans =
   18.0000    0.7391    0.0001
```

suggesting that the root is $x = 0.739$, to 3 decimal places.
The general form of a `while` statement is

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

Exercise 22.1 *Replace* `100` *in Example 22.1 by* `10` *and work through the lines of code with pencil and paper. (Answers:* $n = 2$, $S = 5$.*)*

Exercise 22.2 *Type the code from Example22.1 into a script–file named* `WhileSum.m` *(See §10) and conduct tests to ensure its correctness.*

22.1 `if...else...end`

The "if" construct allows conditional branching depending on the truth or falsity of some logical tests.
Our examples make use of the function `rem`: `rem(x, n)` gives the remainder when `x` is divided by `n`. When `x` is divisible by `n` the remainder is `0`, the Matlab representation of false. A nonzero number is regarded as a logical true (1). Thus, `rem(x, n)` is true when `x` is not divisible by `n` (see §21). The code

```
X = int2str(x); % convert x to string X
if rem(x, 2)
    disp([ X  ' is an odd number'])
end
```

will display a message only when $\texttt{rem}(x, 2) \neq 0$, i.e. when `x` is odd. (`x` is converted to a string `X` to avoid overly long lines and to aid clarity.) The code is now extended to test whether `x` is also divisible by 4 when it is an even number.

```
if rem(x, 2)
      disp([ X ' is an odd number'])
elseif rem(x, 4)
    disp([X ' is even,' ...
      ' not divisible by 4'])
else
      disp([X ' is divisible by 4'])
end
```

The `elseif` statement is only reached if `x` is even when the remainder on division by 4 is examined. Our final extension is a script `ifthen.m` that illustrates how these conditional tests can be further nested. The `input` statement prints out the string in its argument, reads in a number typed at the keyboard (followed by "Return") and assigns it to `x`.

```
%%IFTHEN Conditional branching.
x = input('Enter a positive integer: ');
X = int2str(x);
%%
if rem(x, 2)
    if ~rem(x, 3)
        disp([X ' is divisible by 3'])
    else
        disp([X ' is odd,' ...
            ' not divisible by 3'])
    end
elseif rem(x, 4)
    disp([X ' is even,' ...
      ' not divisible by 4'])
else
    disp([X ' is divisible by 4'])
end
```

The code examines the case when `x` is odd to see whether or not it is divisible by 3.
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
```

Notice how the indentations of the lines of code in the `ifthen.m` example improve its readability. This can be done in the Matlab editor by highlighting the lines of code with the mouse and using the Indent button (④ in Fig. 32). Conditional branching can also be carried out using `switch`, an example is given in Method 3 on page 56.

Exercise 22.3 *A year is a leap year if it is divisible by 4 and, if it is divisible by 100, it must also be divisible by 400. Write a function file that will input a year and output the strings* `'true'` *or* `'false'`*, as appropriate.*

23 More Built–in Functions

23.1 Rounding Numbers

There are a variety of ways of rounding and chopping real numbers to give integers. The definitions given in the Table 1 on page 73 should be used where necessary in order to understand the output given below:

```
>> x = [-4 -1 1 4]*pi
x =
  -12.5664   -3.1416    3.1416   12.5664
>> round(x)
ans =
   -13    -3     3    13
>> fix(x)
ans =
   -12    -3     3    12
>> floor(x)
ans =
   -13    -4     3    12
>> ceil(x)
ans =
   -12    -3     4    13
>> sign(x)
ans =
    -1     -1     1     1
>> rem(x,3)
ans =
   -0.5664  -0.1416   0.1416   0.5664
```

Do "`help round`" for help information.

23.2 `diff`, `cumsum` & `sum`

The "`diff`" command applied to a vector calculates the difference between consecutive entries:

```
>> x = (0:5).^2
x =
     0     1     4     9    16    25
>> y = diff(x)
y =
     1     3     5     7     9
```

The length of `diff(x)` is one fewer than the length of `x`. `diff(f, k)` is the `k`th difference. It applies `diff` '`k`' times and has `k` fewer entries than `x`:

```
>> diff(x, 2)
ans =
     2     2     2     2
>> diff(x, 3)
ans =
     0     0     0
```

The `cumsum` command applied to a vector returns the cumulative sum of its entries

```
>> cumsum( y )
ans =
     1     4     9    16    25
```

and the length is the same as the input `y`. Prepending a suitable "starting value"

```
>> cumsum([0 y])
ans =
     0     1     4     9    16    25
```

returns the original vector `x`. `sum(x)` simply sums the elements of `x`.
The functions `diff`, `cumsum` and `sum` applied to matrices, operate columnwise. The matrix `C` is constructed by "implicit expansion" (see page 7)

```
>> C = (1:3)' + x(1:4)
C =
     1     2     5    10
     2     3     6    11
     3     4     7    12
>> sum(C)
ans =
     6     9    18    33
```

A second argument can be used to specify which dimension to sum, in this case the 2nd (i.e., sum by rows)

```
>> sum(C, 2)
ans =
    18
    22
    26
```

`sum(C,1)` is the same as `sum(C)`. To sum all the entries:

```
>> [sum(sum(C)), sum( C(:) )]
ans =
    66    66
```

As `diff` operates columnwise:

```
>> diff(C)
ans =
     1     1     1     1
     1     1     1     1
```

and the second argument is used to specify the degree of differencing, a 3rd argument is needed to indicate row-wise (the 2nd dimension) operation

```
>> diff(C, 1, 2)
ans =
     1     3     5
     1     3     5
     1     3     5
```

23.3 `max` & `min`

When `x` is a vector, then `max(x)` returns the largest element in `x`

```
>> x = [1.3 2.3 -5.4 0 2.3]
x =
  1.3000   2.3000  -5.4000   0   2.3000
>> max(x), max( abs(x) )
ans =
    2.3000
ans =
    5.4000
>> [ m, j ] = max( x )
m =
    2.3000
j =
     2
```

With two outputs, the first gives the maximum entry and the second the index of the maximum element. Only the first occurrence of the maximum entry is returned. If only the location of the maximum is required, use

```
>> [ ~, j ] = max(x)
j =
     2
```

For a matrix, `A`, `max(A)` returns a row vector containing the maximum element from each column. Thus, `max(max(A))` finds the largest element in `A` (alternatively, use `max( A(:))`).

```
>> B = [ 8 6 4; 7 5 2; 3 1 9].^2
B =
    64    36    16
    49    25     4
     9     1    81
>> max(B)
ans =
    64    36    81
>> max( B(:) )
ans =
    81
>> min( B )
ans =
     9     1     4
```

23.4 Random Numbers

The functions `rand` and `randn` produce pseudo-random numbers that are uniformly and normally distributed, respectively.
The command `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 while `rand(n)` produces an $n \times n$ matrix.

```
>> 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. In order to produce results that can be replicated on successive calls (a particularly useful feature when testing code), the state of the "random number generator" `rng` needs to be saved and later recalled. This is illustrated:

```
>> s = rng; a = rand, rng(s); b = rand
a =
    0.2417
b =
    0.2417
```

where `a` and `b` have equal values.

Example 23.1 *Write code that will simulate n throws of a pair of dice.*

Random integers in the range 1 to 6 can be produced with `randi(6)`

```
>> randi(6)
ans =
     4
```

`rand(N,m,n)` produces an $m \times n$ array of integers in the range 1 to `N`. To simulate 4 throws of a pair of dice:

```
>> randi(6, 4, 2)
ans =
     1     6
     6     5
     2     1
     2     1
```

To calculate the average value of 1000 throws:

```
>> mean(randi(6,1000,2))
ans =
    3.4460    3.4910
```

the function `mean` applied to a matrix computes the arithmetic mean of its columns (which should theoretically be 3.5 in this case).
The command `randperm(n)` produces a random permutation of the integers `1:n`. This is useful for "shuffling" problems. For instance,

```
>>  S = 'matlab'; n = length(S);
>>  i = randperm( n );  S(i)
ans =
btamal
```

randomises the letters of the string `matlab`.

23.5 `find` for vectors

The function "`find`" returns a list of the positions (indices) of the elements of a vector satisfying a given logical test.
For example, to determine the percentage of throws of a pair of dice where they both show the same number, a $2 \times n$ array of integers in the range `1:6` is created. In the following snippet of code `k` returns a list of the columns containing identical entries:

```
>> n = 1000; dice = randi(6, 2, n);
>> k = find( dice(1,:) == dice(2,:) );
>> per_cent =  100*length(k)/n
per_cent =
    16.9000
```

Note that two equal signs `==` are required to test equality.

Example 23.2 *Estimate the number of zeros of the expression* `y` *defined by Example 12.2 in the interval $(-2, 2)$ by counting the number of sign changes.*

Searching directly for zeros gives:

```
>> x = -2:.01:2;
>> y = exp(-3*x.^2).*sin(8*pi*x).^3;
>> k = find( y==0 )
k =
   201
>> x(k)
ans =
     0
```

Thus y is exactly zero at just one point ($x = 0$). There will be at least one zero of y in the interval `(x(i), x(i+1))` if `y(i)*y(i+1) < 0`. The code

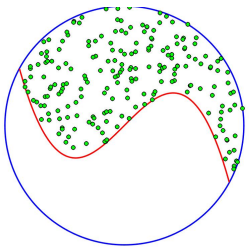
```
>> sum( sign(y(2:end).*y(1:end-1) ) < 0)
ans =
    30
```

counts 30 sign changes giving a total of 31 roots (excluding end-points).

Example 23.3 *The unit disc is divided into two equal halves by the cubic curve* $y = x - 2x^3$. *Draw these regions and populate the upper half with a random set of dots.*

After drawing the circle, points (xc, yc) on the cubic are computed for $-1 \leq xc \leq 1$. `find` is then used to identify those inside the circle.

```
%%FIND_EX Example of find command
x = -1:.002:1;
%% Draw circle
plot(cos(pi*x),sin(pi*x),'b','linewi',2)
axis equal
%% Draw cubic inside circle
yc = x - 2*x.^3;
kc = find(x.^2+yc.^2 < 1);
hold
plot(x(kc), yc(kc), 'r-', 'linewi', 2)
%% Plot random dots in upper half
xd = 2*rand(500,1)-1;
yd = 2*rand(500,1)-1;
kd = find(xd.^2+yd.^2 < 1 & ...
    yd > xd-2*xd.^3);
plot(xd(kd), yd(kd), 'ko',...
    'markerfacecolor','g',...
    'markersize',6)
axis off
hold off
print(mfilename,'-djpeg')
disp(['Fig saved to file: ', mfilename])
```



In the last two lines of this script the function `mfilename` (with no arguments) returns the name of the script file from which it is called. Thus, the plot created by the script `find_ex.m` is saved in the file `find_ex.jpg`. This can be a useful device for associating plots with the file that created them.
To save `figure(4)`, say, we would use
`>> print( 4, [ mfilename '-4'], '-djpeg')`
which appends the string `'-4'` to the file name.

Exercise 23.1 *Can you explain how the following code finds all the prime numbers less than 100?*

```
>> p = 1:100; P = isprime(p);
>> k = find( p.*P ); p = p( k )
```

23.6 `find` for matrices

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

```
>> 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 (see §15.1)—this is equivalent to numbering the elements of `A` by columns as in

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

```
>> 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.
These values can then be used to populate a second matrix:

```
>> B = zeros(size(A)); B(n)  = A(n)
B =
    -2     0     0     0
     0     0    -1     0
     0     0     0     0
```

When the `find` command returns an empty list:

```
>> k = find( A >= 10 )
k =
  0x1 empty double column vector
```

and a branch in a script file may depend on its outcome, we can test for this with `isempty`:

```
if isempty(k)
    disp('No entries match test')
else
    A(k) = A(k) - 10;
end
```

24 Anonymous Functions

These mysteriously named objects are simple one-line functions. For example, in Example 23.3 it would have been useful to create a function

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

since it was used more than once in the script. This can be achieved by defining
`>> f = @(x) x - 2*x.^3;`
The symbol `@` creates a "function handle" and the string `(x)` specifies that `f` is a function of the single variable `x`. The element-wise exponent `.^` is used to allow the function to be called with a vector argument:

```
>> f(1),  f( 0:2 )
ans =
    -1
ans =
     0    -1   -14
```

This new function can also be used as an argument to other functions. For example, to find the root of $f(x)$ close to $x = -1$, we can use the function `fzero`:

```
>> root = fzero( f, -1)
root =
   -0.7071
```

(which may be recognised as $-1/\sqrt{2}$).
Another application would be to compute the area containing the random dots in Example 23.3. That is

$$I = \int_{-1}^{1} \Big(\sqrt{1-x^2} - f(x)\Big)\mathrm{d}x.$$

The integrand is used to create a new anonymous function $g(x)$ which is then input to the built-in function `integral`:

```
>> g = @(x) sqrt(1-x.^2)-f(x);
>> integral(g, -1, 1)/pi
ans =
    0.5000
```

so that $I = \pi/2$ (which could be deduced from symmetry).
The "function handle" `@` can also be used to allow built-in functions to be passed as arguments to other functions. For example to integrate $\sin(x)$ over the interval $(0, \pi)$

```
>> integral(@sin, 0, pi)
ans =
    2.0000
```

25 Function m-files

Function m-files—special cases of m–files (§10)—are appropriate when creating functions that cannot be expressed in a single expression.

Example 25.1 *The area, A, of a triangle with sides of length a, b and c is given by Heron's formula:*

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

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 (see §10). Since the obvious name `area` already exists in the Matlab library:

   ```
   >> which area
   /Applications/MATLAB_R2018a.app/
   toolbox/matlab/specgraph/area.m
   ```

   we shall name our function `heron` and save its definition in a file `heron.m`.

   ```
   >> which heron
   'heron' not found.
   ```

   confirms it is not already in use.

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] = heron(a,b,c)
   ```

3. Document the function. That is, include comments as outlined in §10 that describe the purpose of the function.

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] = heron(a,b,c)
%% HERON Compute area of a triangle
% using Heron's formula
% Inputs:
%    a,b,c:  Lengths of sides
% Output:
%        A: area of triangle
% Usage:
%      Area = heron(2,3,4);

% Written by dfg, Oct 14, 1996.
% Amended by dfg July 25, 2018.
s = (a+b+c)/2;
A = sqrt(s*(s-a)*(s-b)*(s-c));
%%%%%%%%% end of heron %%%%%%%%%%
```

The command
`>> help heron`
will produce the leading comments from the file (up to the first line of code, or blank line):

```
heron Compute area of a triangle
using Heron's formula
Inputs:
   a,b,c:  Lengths of sides
Output:
       A: area of triangle
Usage:
     Area = heron(2,3,4);
```

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

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

where the result of the computation is assigned to the variable `Area`—the capitalised variable name will not clash with the existing function name `area`. If a variable name inadvertently coincides with a function name, as in

```
>> sin = sin(pi/6)
sin =
    0.5000
```

subsequent use of the sin function will cause an error

```
>> sin(pi/2)
??? Subscript indices must either be
    real positive integers or logicals.
```

because the name `sin` now refers to a variable and `pi/2` in the command `sin(pi/2)` is interpreted as an index to a vector. To reclaim the function name the variable `sin` is cleared from memory with

```
>> clear sin
```
If it is unclear how "`sin`", say, is being interpreted we can use the command

```
>> which sin
built-in (/Applications/MATLAB.....)
```

to confirm it is the built-in function.
The variable `s` used in the definition of the `heron` function above is a "local variable": its value is local to the function and is not available outside:

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

If the value of s (as well as A) is required outside the function body, then the first line of the file should be changed to

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

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

1. No outputs assigned

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

   again the second output is ignored.

3. Two outputs assigned

   ```
   >> [Area, hper] = heron(10,15,20)
   Area =
      72.6184
   hper =
      22.5000
   ```

   (`hper`: half perimeter).

4. Two outputs assigned, one a `~`,

   ```
   >> [ ~, hper] = heron(10,15,20)
   hper =
      22.5000
   ```

   Here the first output is ignored.

The previous examples illustrate the fact that a function may have a different number of outputs. It is also possible to write function files that accept a variable number of inputs. For example, in the context of our `heron` function, the area of a right angled triangle may be calculated from the lengths of the two shortest sides, since the third (the hypotenuse) can be calculated by Pythagoras's theorem. So our amended function would operate as before but, when only two input arguments are supplied, it would assume the triangle to be right angled. The reserved variable `nargin` in the body of a function returns the number of input arguments. The revised function, called `heron2`, might then resemble the following code:

```
function  [A] = heron2(a,b,c)
%%HERON2 Compute the area of a triangle
% with sides a, b and c.
% Inputs: either
%    a,b,c:  Lengths of 3 sides
% or
%    a, b: two shortest sides of a
%             right angled triangle
% Output:
%        A: area of triangle
% Usage:
%      Area = area2(2,3,4);
% or
%      Area = area2(3,4);
% Written by dfg, Oct 14, 1996.
% Extended 2012, 2018
if nargin < 2
    error( 'Not enough arguments')
elseif nargin==2
    c = sqrt(a^2+b^2);
end
s = (a+b+c)/2;
A = sqrt(s*(s-a)*(s-b)*(s-c));
%%%%%%%%% end of heron2 %%%%%%%%%%
```

The command `error` issues an error message to the screen and exits the file. It could be usefully deployed in the following exercise.

Exercise 25.1 *Explain the output obtained from the command*

```
heron(4,5,10)
```

Devise a test to warn the user of this type of situation.

Exercise 25.2 *Extend the* `heron2` *function so that it also calculates the area of an equilateral triangle when only one input argument is supplied.*

26 Debugging

There is little doubt that trying to fix a script or function file that fails can be one of the most frustrating aspects of computer programming. There are two main categories of failure:

(a) Syntax errors. These include mis-matched brackets or missing operators (`2a` instead of `2*a`, for example), and are relatively easy to correct—especially when using the Matlab editor (see Appendix D).

Other errors, such as the mis-spelling of function names (`heorn` for `heron`), or the incorrect number of arguments to a function, show up at run-time and their resolution is usually obvious from the diagnostic information given.

```
>> sin(pi,x)
Error using sin
Too many input arguments.
>> heorn(3,4,5)
Undefined function/variable 'heorn'.
Did you mean:
>> heron(3,4,5)
```

Here Matlab was able to suggest the correct function name.

A non-integer index to an array:

```
>> x(d)
Subscript indices must either be
real positive integers or logicals.
```

may be because `d` was typed instead of `id`, say, which is easily fixed. If `d` was expected to be a integer then further investigation is required. The debug facility in the Matlab editor can often help to identify errors.

(b) Mistakes by the programmer. These are the most difficult to detect. Examples include `X^2` instead of `X.^2` (only when `X` is a square matrix, otherwise it is an illegal operation), or `a1` instead of `a+1` when both `a` and `a1` are variables with assigned values. Also if variables are not cleared (by the command `clear`) when switching from one script file to another then, if both files use variables with a common name, the second file might inadvertently inherit its value from the first file rather than be initialised correctly.

Again, using the debug facilities in the Matlab editor can often help to pinpoint mistakes. For further details see Appendix D.

27 Plotting Surfaces

A surface is defined mathematically by a function $f(x, y)$ and, corresponding to each value of (x, y), the height of the function is calculated from

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

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 on which a grid is imposed by lines parallel to the axes. Figure 13 shows the grid with intervals 0.5 in each direction. The x– and y–coordinates of the grid lines in the figure are

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

(in Matlab notation). When these are supplied as arguments to the function `meshgrid` we find:

```
>> [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
```

where the coordinates of the ith point along from the left and the jth point up from the bottom of the grid) are `(X(i,j), Y(i,j))`. The function f is evaluated at each of the points bearing in mind, as in Example 14.1, that element-wise operations (powers, products, etc.) are usually appropriate.

Example 27.1 *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')
```

The separation of grid lines has been reduced to 0.2 for better resolution.
Among the items on the toolbar of the figure window are two important icons:

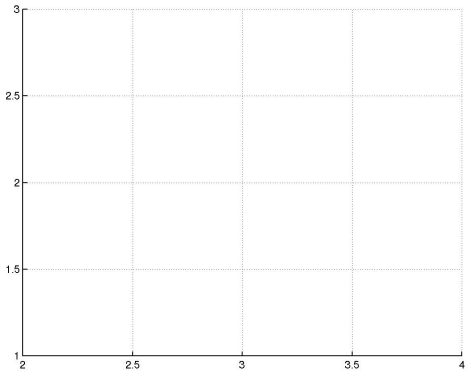


Fig. 13: An example of a 2D grid

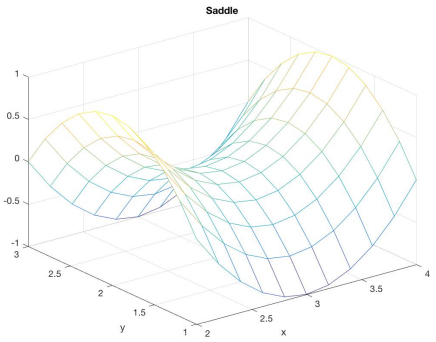


Fig. 14: Plot of a saddle function for Example 27.1

Clicking on the left icon allows the 3D image to be rotated by clicking and dragging with a mouse. During this process the cursor changes to a dotted circle with an arrowhead and, in the lower left corner of the window, are shown AZ, the azimuth (horizontal) rotation, and EL is the vertical elevation (both in degrees). The defaults are
`Az: -37.5, El: 30`
These values can also be set manually via, for instance,
`>> view(30, 45)`
which sets `Az = 30, El = 45`. This feature is useful in a script file when a particular view needs to be recreated.
Clicking on the right-hand icon shown above and then on a point of intersection of the grid lines with the surface shows the coordinates of that point.

Example 27.2 *As an example of an anonymous function of two variables, plot the function* $P(x, y)$ *defined by*

$$P(x, y) = \begin{cases} y, & x \geq y \\ x, & y > x \end{cases}$$

for x, y *in the unit square.*

```
%%PLIN Plot piecewise linear function
P = @(x,y)  min(x, y);
[X, Y] = meshgrid(0 : .1 : 1);
mesh(X, Y, P(X,Y))
colormap([0 0 0])
set(gca, 'fontsi',20)
print(mfilename, '-djpeg')
```

The result is shown in Fig. 15 where the command `colormap([0,0,0])` renders the image in black and white for greater clarity.

A similar effect could have been achieved without having to set up a grid with `meshgrid` by using `fmesh`:

```
>> fmesh(P, [0 1 0 1])
```

or, indeed, without the need to name the anonymous function

```
>> fmesh(@(x,y) min(x, y), [0 1])
```

(this assumes that the x- and y-ranges are the same). Both plots use grids with 35 points in each direction.

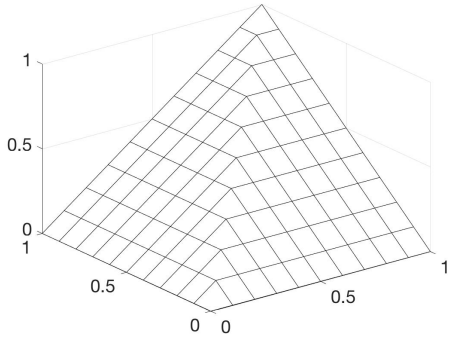


Fig. 15: Plot of the piecewise linear function for Example 27.2

The next example illustrates two other commands, `surf` and `contour`, for visualising 2D functions.

Example 27.3 *Plot the surface defined by the function*

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

on the domain $-2 \le x \le 2, -2 \le y \le 2$.

```
%%TWODPLOT Example of plots in 2 dimensions.
[X,Y] = meshgrid(-2:.1:2);
Z = -X.*Y.*exp(-2*(X.^2+Y.^2));
%%
figure(1)
surf(X, Y, Z)
xlabel('$x$'), ylabel('$y$')
%% Contours & maxima
figure(2)
contour(X, Y, Z)
xlabel('$x$'), ylabel('$y$');
zmax  = max( Z(:) ); % find max element
%k = find(Z == zmax);
disp(['Max. height =  ' num2str(zmax) ])
```

Then

```
>> twodplot
Max. height =  0.09197
```

The results are shown in Fig. 16. See §12.10 for formatting text and labels.

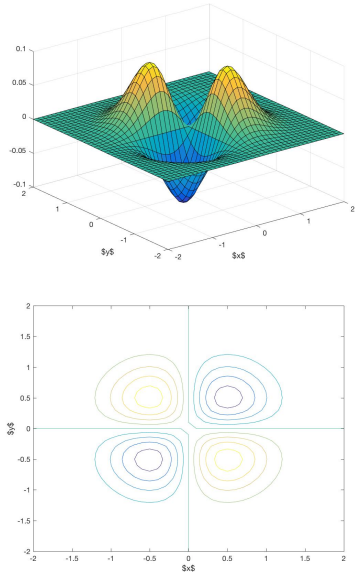


Fig. 16: "`surf`" and "`contour`" plots for Example 27.3

28 Formatted Printing

In this section we build on the rather cursory introduction to formatted output that was presented in §18. This will be done largely through the issues encountered when printing the matrix[2]

$$A = \begin{bmatrix} 10 & 0.73998776 & 0.00224361 \\ 20 & 0.73910250 & 0.00004315 \end{bmatrix}$$

Using `disp` we find

```
>> disp(A)
   10.0000    0.7400    0.0022
   20.0000    0.7391    0.0000
```

It is seen that the default is to use four decimal places, even for the integers in the first column. Also, the last number in the table contains no information (other than it is smaller than 0.00005). The first aim is to print out the three columns in a more natural manner, We get close to this with `num2str`:

[2] The data are the values of `[ n, xnew, d]` from the second example of a while loop in Example 22.2 (page 35) when the termination condition `d > 0.001` is changed to `d > 1e-5`.

```
disp( num2str(A) )
10      0.739988    0.00224361
20      0.739103   4.31502e-05
```

where the third column requires further refinement. For this a second argument is added to `num2str` in the form of a string that contains a number of formatting operators that control the way each column is printed. A formatting operator is a string starting with % and ending with a conversion character. For example,

```
>> disp( num2str(A, '%i %f %e') )
10 0.739988 2.243606e-03
20 0.739103 4.315024e-05
```

and the conversion characters `i`, `f`, `e` signify that the three columns of `A` should be printed as, respectively, an integer[3], a floating point number and a floating point number formatted in e-notation (akin to scientific notation: 2.243606×10^{-3}). The space that separates these formats is replicated in the space between columns in the output. Including a comma and leaving more space between format specifications leads to:

```
>> a = A(1,:);
>> disp( num2str(a,'%i,   %f,   %e') )
10,   0.739988,   2.243606e-03
```

where we now focus on `a`, the first row of `A`.
Further fine tuning is made possible by the introduction of instructions between the `%` and the conversion character (`i`, `f` or `e`). A positive integer specifies the *minimum* number of characters to be output—known as the *field width*. The actual number of characters output will exceed this if the number does not fit in the space allocated. The next example has field widths of 10 (the format specification starts with a `'|'` since leading spaces are ignored by `num2str`):

```
>> disp( num2str(a,'|%10i,%10f,%10e') )
|        10,  0.739988,2.243606e-03
```

where the first element is padded with 8 blanks, the second with two blanks and the third entry is 12 characters wide. Providing a negative field width means that the item should be left-justified:

```
>> disp( num2str(a,'|%-10i,%10f,%10e') )
|10        ,  0.739988,2.243606e-03
```

Further adjustments of the field width result in:

```
>> disp( num2str(a,'|%3i,%10f,%14e') )
| 10,  0.739988,  2.243606e-03
```

If inter-column spaces are inserted manually then the field width is less critical and is often set to zero:

```
>> disp( num2str(a,'|%0i,  %0f,  %0e') )
|10,  0.739988,  2.243606e-03
```

Another important role of the format is to specify the *precision*—the number of digits after the decimal point. This does not apply to the `i` (integer) conversion character. To increase the precision of the second column to 8 digits and reduce that of the third to 2 digits:

```
>> disp( num2str( a,'|%3i,%12.8f,%10.2e') )
| 10,  0.73998776,  2.24e-03
```

or, equivalently, with manually inserted spaces,

```
>> disp( num2str(a,'| %0i, %0.8f, %0.2e') )
```

Suppose that `B` is a 2×14 matrix of integers in the range `1:6` generated by:

```
>> B = randi(6,2,14);
```

This can be printed compactly using

```
>> disp( num2str(B,'  %i') )
3  6  5  4  3  6  3  3  6  3  2  6  4  1
4  6  2  1  5  5  5  6  6  3  5  6  4  6
```

which uses the same format specification for each entry. When there are two (or more) format commands:

```
>> disp( num2str(B,' (%i,%i)') )
(3,6) (5,4) (3,6) (3,3) (6,3) (2,6) (4,1)
(4,6) (2,1) (5,5) (5,6) (6,3) (5,6) (4,6)
```

they are recycled for each pair of integers.
It requires greater elaboration to place commas between each bracketed pair (but not at the end of each line) to produce the string `S`:

```
S =
(3,6),(5,4),(3,6),(3,3),(6,3),(2,6),(4,1)
(4,6),(2,1),(5,5),(5,6),(6,3),(5,6),(4,6)
```

The format used is

```
fmt = [repmat('(%i,%i),', 1, 6) '(%i,%i)'];
```

in which the `repmat` command repeats the format `'(%i,%i),'` six times and is followed by `'(%i,%i)'` without a comma. Then

```
>>  S = num2str(B, fmt)
```

produces the required result.
LaTeX lovers who wish to include a similar structure within a `tabular` environment require "columns" to be separated by `&` and each row, except the last, to end in `\\`.

[3] Integers can also be formatted using `%d`.

```
>>fmt=[repmat('(%i,%i)&',1,6) '(%i,%i)\\\\'];
>>S = num2str(B, fmt)
S =
(3,6)&(5,4)&(3,6)&(3,3)&(6,3)&(2,6)&(4,1)\\
(4,6)&(2,1)&(5,5)&(5,6)&(6,3)&(5,6)&(4,6)\\
```

which, because each row of `B` is treated identically, both end in `\\`. It is important to note that the format contains four `\`. This is because a backslash has special meaning within a format[4] so (just as `''` are needed in a string to display a single quote, see page 17), two backslashes are needed for each one output.

The required output in the previous example (without the last `\\`) can be generated using the commands `sprintf` (which outputs to a string, like `num2str`) and `fprintf` (which outputs directly to the command window or file). These commands use the same formatting specifications as `num2str` but they are sometimes interpreted differently. One reason for this is that these new commands take multiple arguments, of which the first is a format specification and the remainder are variables (scalars, vectors, matrices or strings).

Returning to the row vector `a` used earlier in this section

```
>> disp( sprintf('%5i, %f, %e', a) )
   10, 0.739988, 2.243606e-03
```

we see that the format precedes the items to be printed using `sprintf`. Also, leading spaces are honoured in the first entry and the default for floating point numbers is six decimal places. The same output is produced by the commands

```
>> a1 = a(1); a2 = a(2); a3 = a(3);
>> disp( sprintf('%5i, %f, %e', a1, a2, a3))
   10, 0.739988, 2.243606e-03
```

in which the function `sprintf` has four arguments. The main difference between `num2str` and `sprintf` (also `fprintf`) is in the treatment of column vectors and matrices. `num2str` prints each row on a separate line and the same format (if present) is applied to each row. By contrast, both `sprintf` and `fprintf` convert matrices to column vectors (as in `B(:)`, see §15.1) and then print them as *row vectors*. The three formats specifications in the next command are repeatedly cycled

```
>> disp( sprintf('%i, %f, %e',A) )
10, 20.000000, 7.399878e-017.391025e-01,
0.002244,4.315024e-05
```

[4] Including `\n` or `\t` within a format specification will generate a newline and a tab, respectively.

(the output line is "wrapped") which helps to explain the rather garbled output. To get around this problem we first use the transpose `A'` instead of `A` (so that the entries appear in the correct order when converted to a column vector) then a "newline command" (`\n`) is inserted at the end of each row. So armed, the required command is

```
>> disp( sprintf('%i, %0.5f, %0.2e\n', A' ) )
10, 0.73999, 2.24e-03
20, 0.73910, 4.32e-05
```

The `sprintf` command is now applied to the LaTeX example introduced earlier in this section. The format specification for each row is defined first, for clarity.

```
>>fmt=[repmat('(%i,%i)&', 1, 6) '(%i,%i)'];
>>Sp = sprintf([fmt '\\\\\n' fmt], B')
Sp =
(3,6)&(5,4)&(3,6)&(3,3)&(6,3)&(2,6)&(4,1)\\
(4,6)&(2,1)&(5,5)&(5,6)&(6,3)&(5,6)&(4,6)
```

This can be imported into a document using "Cut & Paste". The command `disp(Sp)` displays the string `Sp` without echoing its name.

By contrast, the command `fprintf` prints directly to the command window:

```
>> fprintf('%i, %0.5f, %0.2e',a)
10, 0.73999, 2.24e-03>>
```

Notice that the command prompt `>>` is appended to the output—"newlines" must be explicitly invoked via `\n`:

```
>> fprintf('%i, %0.5f, %0.2e\n', a)
10, 0.73999, 2.24e-03
>>
```

A particularly useful feature of `fprintf` is its ability to write its output to a file.

This is a three-stage process:

1. Open the file and assign a unique identifier:
   ```
   >> fid = fopen('mydata.dat','w')
   fid =
        7
   ```
   `fid` is the file identifier and the argument `'w'` signifies open for writing.
2. Write the data to the file:
   ```
   >> fprintf(fid, ' %i', B)
   ans =
       56
   ```
   As described earlier, with `B` a 2×14 matrix, this is written to file with this format as a 1×28 row vector. `ans` specifies how many bytes were written.

3. Close the file:

```
>> fclose(fid)
ans =
    0
```

Here `ans = 0` signifies a successful closure, otherwise `-1` is returned.

To later add data to the file it should be opened with

```
>> fid = fopen('mydata.dat','a')
```

where `'a'` signifies `append`.
To read the data back involves a similar process:

1. Open the file and assign a unique identifier:

```
>> fin = fopen('mydata.dat','r')
fin =
     6
```

`fin` is the file identifier and the argument `'r'` signifies open for reading.

2. Read the data from the file: (Note that the format of the data must be known and it should not include any punctuation or text.)

```
>> b = fscanf(fin, ' %i', B);
>> size(b)
ans =
    28     1
```

so `b` is a 28×1 column vector. It can be restored to its original shape using

```
>> reshape(b, 2, 14)
```

3. Close the file:

```
>> fclose(fin)
ans =
    0
```

Once the data has been read the file must be closed and opened again in order to re-read the data. Further aspects are explored in Appendix E.

29 Extended Examples

This section contains rather longer examples that introduce some new functions and techniques. One key idea is the *vectorization* of code—this means avoiding `for` loops (because they can be slow) in favour of writing expressions in terms of vectors and matrices. For example, the average (mean) of pairs of consecutive entries in a (row) vector `x` can be calculated by

```
for j = 1:length(x)-1
   xm(j) = .5*(x(j) + x(j-1));
end
```

or, by the single vectorised statement

```
xm = .5*( x(1:end-1) + x(2:end) );
```

If the entries in `x` are in increasing order and a new row vector `X` is to be formed my merging `x` and `xm` in such a way that the entries in `X` are also in increasing order then, with a loop,

```
X(1) = x(1);
for j = 1:length(x)-1
   X(2*j)   = xm(j);
   X(2*j+1) = x(j+1);
end
```

as opposed to

```
X = sort( [x, xm] );
```

If there is a value `y(j)` associated with each `x(j)` (as in `y = f(x)`, for example) then to merge the vectors `y` and `ym = f(xm)`, the `sort` command also has to return the indexing used:

```
[X, I] = sort( [x, xm] );
Y = [y, ym];
Y = Y(I);
```

so that the values in `Y` are reordered appropriately. These ideas are used in the first example.

Example 29.1 (Fractal Horizon) *A caricature of a "rocky horizon" can be created in an iterative process starting with the coordinates of at least two points held in vectors* `x` *and* `y`*. At each iteration the midpoint of each consecutive pair of points* `(x(i), y(i))`*,* `(x(i+1), y(i+1))` *is moved vertically by a random amount in the range* `(-amp, amp)`*. In our illustration, the amplitude* `amp` *at the* `j`*th iteration is chosen to be* `10/j^1.5` *so that later steps impose finer-scale detail.*

The first file is a function `fractalstep` that carries out a single iteration.

```
function [X,Y] = fractalstep(x,y,amp)
%%FRACTALSTEP Single step in creation
%of a fractal horizon. The midpoint
%of consectutive points (x(i), y(i))
%and (x(i+1), y(i+1))is moved vertically
%by a random amount in the range
%(-amp, amp). If x & y have length N,
%the outputs X & Y  have length 2N-1.

% Find mid-points of x & y coordinates
X = 0.5*(x(1:end-1)+x(2:end));
Y = 0.5*(y(1:end-1)+y(2:end));
```

```
Y = Y + (2*rand(1,length(Y))-1)*amp;
% Merge x & X
[X,i] = sort([x,X]);
Y = [y,Y];
% Reorder entries in Y
Y = Y(i);
```

This function is called by the script file `fractal.m` that carries out seven iterations. Most of its code is devoted to presenting the results.

```
%%FRACTAL Create a fractal landscape
%by repeated calls of the function
%fractalstep.m

%Starting vectors:
x = 0:.5:1;
y = [0 1 0];
for j = 1:7
  [x, y] = fractalstep(x, y, 10/j^1.5);
end
figure(1)
%Fill in the background colour
B =  fill([0 1 1 0], ...
    [(max(y)+2)*[1 1],(min(y)*[1 1])],'b');
%Adjust the background colour
set(B,'facecolor',[.4 .7 1])
hold on
F = fill([x max(x),min(x)], ...
    [y  (min(y)-2)*[1 1] ],'g');
set(F,'facecolor',[0 .65 0])
set(F,'linewi',2,'edgecol',[0 .45 0])
axis off, hold off
% change the aspect ratio
pbaspect([1 .3 1])
print(mfilename,'-djpeg')
```

The coloured areas in Fig. 17 are created by the `fill` command. For example, `fill(X,Y,'b')` fills the polygon defined by the vectors `X` and `Y` with blue (`'b'`). The background is assigned the handle `B` whose colour (the attribute named `facecolor`) is adjusted to have the `rgb` values `[.4 .7 1]`. For the foreground (handle `F`) the shade of green used is adjusted as is the line width and colour of its border. The command `pbaspect` changes the aspect ratio of the plot box—its third argument applies to the `z` coordinate and is irrelevant in this example.

Example 29.2 (Koch Snowflake) *Koch devised his snowflake in 1904 [15]. Starting with an equilateral triangle, it is constructed by recursively building smaller equilateral triangles on the middle third of each edge.*

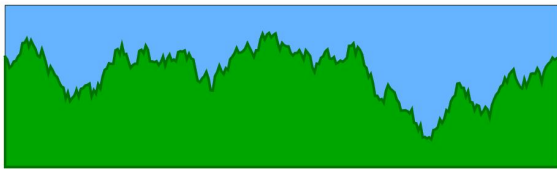


Fig. 17: Output for Example 29.1

To avoid having to carry out the same operations on both the x and y coordinates of vertices (see `fractalstep.m` above) we use complex coordinates $z = x + \mathrm{i}y$. At each stage the figure created is a polygon and the next refinement step loops over all edges to identify the 1/3 and 2/3 points that divide the edges in three equal parts. The line segment joining these points is then rotated counterclockwise through and angle $\pi/3$ about the 1/3 point and this creates the 3rd vertex of a new equilateral triangle. Three additional new vertices are thus created from each "old" vertex. These, along with the coordinates of the "leading" vertex of each edge are then assembled into a $4 \times N$ array before being reshaped into a $1 \times 4N$ vector. Note the use of `.'` to form the transpose of `z(:)`.

```
%KOCH snowlake
% Begin with equilateral triangle
z = exp(-2*pi*i*(0:3)/3);
for j = 1:5
  %Find 1/3 & 2/3 points
  Z = ([2 1;1 2]/3)*[z(1:end-1) ;z(2:end)];
  %Rotate middle 1/3 by pi/3 about 1/3 point
  zr = Z(1,:) + exp(i*pi/3)*diff(Z);
  %Concatenate the pieces: 4xN array
  z = [z(1:end-1);Z(1,:); zr; Z(2,:)];
  %Reorder as a row vector & make last=first
  z = [z(:).' z(1)];
  s = j/4;  % Scale factor
  %Scale by s, translate by (j/3,-j/2), and
  % fill with a shade of grey
  fill(real(z*s)+j/3,imag(z*s)-j/2,...
     [.2 .2 .2]*j)
  hold on
end
hold off
axis('equal','off')
print(mfilename,'-djpeg')
```

The images in Fig. 18 show the first five iterations, which are scaled, filled with different shades of grey

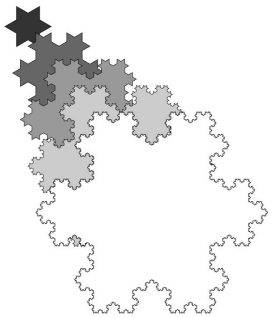


Fig. 18: The first five stages of a Koch snowflake, Example 29.2

(see page 16) and translated to make the result a little more interesting.

Example 29.3 (Curves of constant width) *A closed curve has constant width d if, for every point on the curve, there is a point at the maximum distance d to any other point on the curve. Verify that the perimeter* $= d\pi$ *(Berbier's Theorem).*

We shall, without loss of generality, assume that $d = 2$. The simplest curve of this type is the circle, for which Berbier's Theorem is obviously true. The simplest non-trivial case is that of a Reuleaux triangle, which is made up of three circular arcs, centred at the vertices of an equilateral triangle. Berbier's Theorem is again true since each arc has radius 2 and subtends an angle $\pi/3$ at the corresponding vertex. The first step in the creation of a Reuleaux triangle is illustrated in Fig. 19 (left). With three vertices V_0, V_1, V_2 on the unit circle centred at O, a circular arc of radius $d = 2$ and centre V_0 is drawn through V_1, V_2. The coordinates of n points on this arc are stored in the complex vector $z = x + \mathrm{i}y$ in the function `reuleaux.m` listed below, and then rotated to create the 2nd and 3rd sides.

```
function Y = reuleaux(n)
%%REULEAUX Returns coordinates
%(complex) of 3n points on a
%Reuleaux triangle of diameter 2
%centred on the origin.

if nargin<1, n = 20;end
t = (-n:(n-1))'*pi/(6*n);
z = 2/sqrt(3)  + 2*exp(i*(pi-t));
% Rotator
Z = exp(2*i*pi*(0:2)/3);
y = repmat(z,1,3)*diag(Z');
% Append y(1) to create closed curve
Y = [y(:);y(1)];
```

The curve on the right of Fig. 19 has been drawn with the commands:

```
zr = reuleaux(10);
figure(1)
plot(zr,'b-')
axis('equal','off')
T = 'Reuleaux triangle';
text(0,0,T,...
     'HorizontalAlignment','center',...
     'fontsize',36)
```

Observe that the `plot` command with a complex argument automatically plots the real part versus the imaginary part.

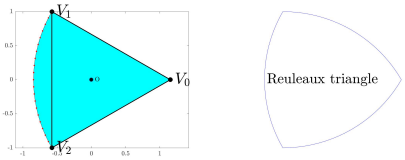


Fig. 19: Reuleaux Triangle (right) and a step in its construction (left)

Kearsley [7] describes a completely different way of constructing curves of constant width and arrives at the parametric representation

$$\begin{aligned} x(t) &= p(t)\cos t - p'(t)\sin t \\ y(t) &= p(t)\sin t + p'(t)\cos t, \end{aligned} \qquad 0 \le t < 2\pi,$$

where $p(t) + p(t+\pi) = 2$ (the diameter). This is implemented in the function file `ccw.m` listed below for $p(t) = 1 + a\cos(kt)$, where a is a parameter in the range $(0, 1/(k^2-1)]$ and k is an odd integer (specifying the number of "sides" in the figure).

```
function z  = ccw(a,n,k)
%%CCW Closed curve of constant width.
% a: parameter <= 1/(k^2-1)
% n: number of sampling points.
%    Default: 60
% k: number of "edges" (odd)
%    Default = 3;
if nargin<2, n = 60; end
if nargin<3, k = 3; end

p  = @(t)  1 + a*cos(k*t);
```

```
dp = @(t)   -k*a*sin(k*t);
t = (0:n)*2*pi/n;
z = (p(t)+1i*dp(t)).*exp(1i*t);
```

Some results are displayed in Fig. 20. Complex numbers are used since the curve can be described succinctly by

$$z = \big(p(t) + \mathrm{i}p'(t)\big)\mathrm{e}^{\mathrm{i}t}, \quad z = x + \mathrm{i}y.$$

It is known as a polynomial curve of constant width since it can be shown that x and y satisfy a polynomial equation of degree eight [11] when the parameter t is eliminated. The consequences of violating the constraint $a \le 1/(k^2-1)$ is illustrated in the middle figure. The solid curve shown in the left of Fig. 20 is drawn with the commands

```
>> z = ccw(0.125,1000);
>> plot(z, 'b-')
>> axis('equal','off')
>> hold on
>> txt = text(0, 0, '$a = 0.125$');
>> set(txt,'HorizontalAlignment','center')
```

The function `ccw` is called with just two arguments, the third taking on its default value. A large number of points are taken on the curve ($n = 1000$) in order to give a sufficiently accurate measure of the perimeter (this is computed by summing the distance between consecutive points on the curve (`sum(abs(diff(z)))`)). The maximum distance has to be calculated for any two points on the curve. Using implicit expansion, (page 7) the `(i, j)` entry in the symmetric matrix `abs(z-z.')` gives the distance between `z(i)` and `z(j)`. The mean diameter and the maximum deviation from the mean can then be calculated using

```
m  = max(abs(z-z.')); % max distance
wd = mean(m);         % mean diameter
per = sum(abs(diff(z))); % perimeter
fprintf(['Mean width = %f\n ' ...
  'Max deviation = %0.1e\n'],...
  wd,max(abs(diff(m))))
fprintf(['Berbier''s Thm:' ...
    'Perimeter/diameter = %f\n'],per/wd)
```

which result in (note the two quotes in Berbier"s, see page 17)

```
Mean width = 2.000000
Max deviation = 8.9e-16
Berbier's Thm:Perimeter/diameter = 3.141587
```

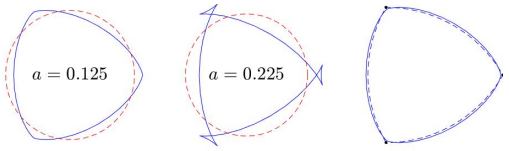


Fig. 20: Left: A polynomial curve of constant width with $a = 1/8$ (solid) along with a circle of the same diameter (dashed). Centre: A curve violating the constraint $a \le 1/(k^2-1)$. Right: a comparison of the case $a = 1/8$ (solid curve) with the Reuleaux Triangle (dashed)

Example 29.4 (Palindromic numbers) *A palindromic number is one which reads the same forwards and backwards, for example, 1441. The question is, if x is a palindromic number, how many others are there up to, and including x?*
Write a Matlab function that will check whether a given number x is palindromic and, if so, return the number of palindromic numbers $\le x$.

If x is the Nth palindromic number and has n digits we first find k, which is defined to be the "left half" of x. For example, if $x = 1221$ then $k = 12$ whereas $k = 123$ when $x = 12321$. The formula for N is

$$N = k + 10^{\lfloor n/2 \rfloor} - 1,$$

where $\lfloor t \rfloor$ (`floor` in Matlab) denotes the integer part of t.
The task requires a given positive integer x to be broken into its individual digits. Perhaps the most direct way of doing this is to note the remainders (using `rem`) when x is repeatedly divided by 10. A simpler solution is to use `num2str` to convert the number to a string (`d` in the function `palin.m` listed below). `fliplr(d)` reverses the characters in `d` and then the function `any` returns "true" (i.e. 1) if any elements in its argument is non-zero (or true).

```
function N = palin(x)
%PALIN Tests whether x is palindromic
% Returns: NaN when result is false
% N = # of palindromic numbers <= x

% Convert x to a string of digits
d = num2str(x);
n = length(d); % # digits in x
% fliplr reverses string d
if any(d-fliplr(d))
    N = nan;   % Not a palindrome
else
```

```
    % get left half of d and
    % convert to a number
    k = str2num(d(1:ceil(n/2)));
    N = k + 10^floor(n/2) - 1;
end
```

Using this we find:

```
>> n = palin(12321)
n =
   222
>> palin(123123)
ans =
  NaN
```

showing that 12321 is the 222nd palindromic number and (unsurprisingly), 123123 is not palindromic.

Example 29.5 *Suppose that*

$$f(x) = \exp(2\cos(x)).$$

Calculate approximate values for (a) $f'(x)$, (b) $f''(x)$, (c) $\int_0^x f(t)\,\mathrm{d}t$ and (d) the arc length of $f(x)$ for $-\pi \le x \le \pi$.

The function f is evaluated at the 201 points specified by the vector `x`. The derivative is then approximated by

$$\frac{f(x_{i+1}) - f(x_i)}{x_{i+1} - x_i} \Rightarrow \texttt{diff(f)./diff(x)}.$$

The result is a vector of 200 components that approximates $f'(x)$ at $\bar{x}_i = \frac{1}{2}(x_i + x_{i+1})$ which also has 200 components. Since the abscissae `x` are equispaced `diff(x)` can be replaced by `dx = x(2)-x(1)`. The `diff` command has to be applied twice for the second derivative and we invoke its second argument: `diff(f,k)` applies `diff` '`k`' times and results in a vector of length `201-k`.

```
>> x = (-1 : 0.01 : 1)*pi;
>> f = exp( 2*cos(x) );
>> plot( x, f), hold on
>> dx = x(2)-x(1);
>> df = diff( f )/dx;
>> xbar = .5*( x(1:end-1) + x(2:end) );
>> d2f = diff( f, 2)/dx^2;
>> plot( xbar,df, '--', x(2:end-1), d2f,'m:')
```

For the integral at $x = x_i$ we use the approximation

$$\int_0^{x_i} f(t)\,\mathrm{d}t \approx h\sum_{j=1}^{i} f(x_j)$$

for $i = 1 : 201$, where $h = x_2 - x_1$. This requires the cumulative sum (see Example 14.3):

```
>> plot( x, dx*cumsum(f), 'k-.')
```

A legend and grid are then added and the results shown in Fig. 21.

```
>> set(groot,...
    'defaulttextinterpreter', 'latex')
>> set(0,'defaultlinelinewidth',2)
>> L=legend('$f(x)$', '$f''(x)$', ...
     '$f''''(x)$',  '$\int_0^x f(t) dt$', ...
     'location', 'northwest');
>> grid on
>> set(L,'interpreter', 'latex')
>> xlabel('$x$', 'fontsize', 24)
```

Note the two quotes for $f'(x)$ and the four quotes for $f''(x)$ in the legend (see page 17). Finally, the arc length is approximated by summing the distances between consecutive points $(x_i, f(x_i))$ and $(x_{i+1}, f(x_{i+1}))$:

```
>> s = sum( sqrt(dx^2 + diff(f).^2) );
>> disp(['Arc length = ', num2str(s)])
Arc length = 17.1011
```

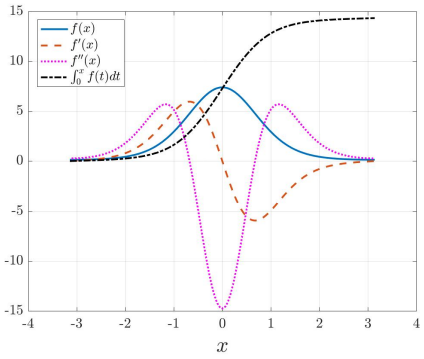


Fig. 21: Graphs of $f(x)$, $f'(x)$, $f''(x)$ and $\int_0^x f(t)\,\mathrm{d}t$ for Example 29.5

A better approximation to the first derivative is described in the next example.

Example 29.6 (Complex step differentiation) *Compare the approximations to the derivative of $f(x) = \sin x$ at x_0 using*

$$\text{(a)} \qquad f'(x_0) \approx \frac{f(x_0 + h) - f(x_0)}{h}$$

$$\text{(b)} \qquad f'(x_0) \approx \frac{f(x_0 + h) - f(x_0 - h)}{2h}$$

$$\text{(c)} \qquad f'(x_0) \approx \Im\frac{f(x_0 + \mathrm{i}h)}{h} \qquad \mathrm{i} = \sqrt{-1},$$

when $h = 10^{-n}$, $(n = -2, -3, \ldots, -10)$.

All three approximations are based on the Taylor series

$$f(x_0+h) = f(x_0) + hf'(x_0) + \tfrac{1}{2!}h^2 f''(x_0) + \tfrac{1}{3!}h^3 f'''(x_0) + \ldots,$$

where $'$ denotes differentiation with respect to x. When this is substituted into (a) and (b) the leading terms in the truncated series are proportional to h and h^2, respectively, which is why they are known as first and second order finite difference approximations to $f'(x)$. When h is replaced by ih alternate terms in the series are real and imaginary and it is readily shown that (c) also provides a second order approximation to $f'(x_0)$. It gives, in fact, identical results to (b) in the absence of rounding error.
This is illustrated by the script file `fdiff.m` listed below which approximates the derivative of $f(x) = \sin(x)$ at $x = x_0$, where $x_0 = 0.4102$ (chosen randomly), with a sequence of step sizes $h = 10^{-n}$, $(n = 2, 3, \ldots, 10)$.

```
%FDIFF Finite difference approximation
% of the derivative of sin(x) at x = x0
% using real and imaginary increments.
h = 10.^(-2:-1:-10);
x0 = 0.4102;
d1 = (sin(x0+h)-sin(x0))./h;
d2 = (sin(x0+h)-sin(x0-h))./(2*h);
di = imag(sin(x0+1i*h))./h;

%%Plot the errors
figure(1)
loglog(h,abs(d1-cos(x0)),'rx-',...
    h,abs(d2-cos(x0)),'bo--',...
    h,abs(di-cos(x0)),'k*-',...
    'markersize',8);
grid on
set(gca,'xminorgrid','off')
xlabel('$h$','fontsize',24)
ylabel('Error','fontsize',24)
L = legend('(a) 1st order',...
    '(b) 2nd order','(c) Complex',...
    'location','southeast');
set(L,'fontsize',24)
print(mfilename,'-djpeg')
```

The results are shown in Fig. 22. The function `loglog` is used in exactly the same way as the `plot` command but uses logarithmic scales on both x and y axes. For method (a) ($\times$) the error in the approximation decreases by a factor of 10 when h is reduced by the same factor until $h = 10^{-8}$. At this

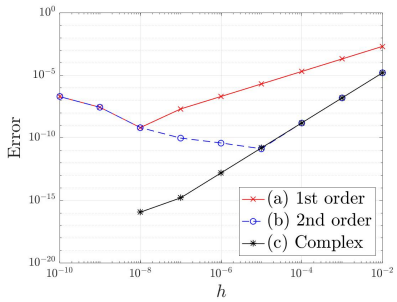


Fig. 22: Error in approximating the derivative of $\sin x$ at $x = x_0$ in Example 29.6

stage, the first seven digits of $\sin(x_0+h)$ and $\sin(x)$ are the same so, since calculations are performed to about 16 digits, only the leading 9 digits in the difference can be correct—thus the rounding error is roughly 10^{-10}, which is the same magnitude as the error in truncating the series. For smaller values of h rounding error dominates and increases as h decreases. For method (b) ($\circ$) the effects of rounding error become apparent for $h \leq 10^{-5}$ and dominate as h decreases. In both (a) and (b) rounding errors are introduced by the cancellation of nearly equal values (called "cancellation errors"). Such errors cannot occur in method (c) ($*$) since no subtraction is involved. There are no data points for $h \leq 10^{-8}$ since the error is zero to working precision!
Method (c) was devised by Lyness and Moler in 1967 (see Cleve Moler's more recent blog [10]) and it is relatively recently that it has gained the prominence it deserves (Higham[6]).

Example 29.7 (Newton's Method I)
Write a Matlab function that uses Newton's method to calculate a root of $f(x) = 0$ from an initial guess x_0.

Newton's method computes the sequence

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

for $n = 0, 1, \ldots$ and will be terminated either when $|f(x_{n+1})| <$ `tol` (where `tol` is a user-specified tolerance) or when the number of iterations exceeds `nlim` (to prevent a non-terminating sequence should the procedure fail to detect a root—see Example 22.2). The function will require inputs:

- the name of the function—we shall use `fun`—which returns the value of the function and its derivative,

- the initial guess x_0, `tol` and `nlim`.

The Matlab function listed below repeatedly updates the initial guess. The outputs from the function will be `x0`, an estimate for the root, `f0` (the value of `fun` at `x0` and a flag that will indicate whether the iteration terminated within `nlim` iterations. The value `0` will indicate failure.

```
function [x0,f0,flag] = newt(fun,x0,tol,nlim)
%%NEWT Newton's Method for solving f(x) = 0.
% Inputs:
% fun : function handle. fun(x) should
%       return a 2x1 vector [f(x); f'(x)].
% x0  : Initial guess (default: x0 = 1)
% tol : Stop when |fun(x0)| < tol
%       Default : 1e-10
% nlim: Max number of steps
%       Default : 20
% Outputs:
% x0  : Estimate of root
% f0  : value of fun(x0) at the root
% flag = 1 means iteration converged within
%           nlim iterations
%     =  0 iteration not converged
% Example:
%  myfun = @(x) [cos(x)-x; -sin(x)-1];
%  [x0,f0,flag] = newt(myfun, 1, 1e-10, 10);

if nargin < 4, nlim = 20; end
if nargin < 3, tol  = 1e-10; end
if nargin < 2, x0   = 1; end

f0 = 1; n = 0;
while abs(f0(1))>tol && n<nlim
    f0 = fun(x0);
    x0 = x0 - f0(1)/f0(2);
    n = n+1;
end
f0 = f0(1);
flag = (n<nlim);
    % +1 if iteration has converged
```

As an example of its use, suppose that $f(x) = \cos(x) - x$. Then using an anonymous function `F`:

```
>> F = @(x) [cos(x)-x; -sin(x)-1];
>> [x, fx, fg] = newt(F, 1)
x =
    0.7391
fx =
    0
fg =
     1
```

indicating that $f(x) = 0$ (to within rounding error) when `x = 0.7931...`. Equivalently, the command could be replaced by

```
>> [x,fx,fg]= newt(@(x)[cos(x)-x; -sin(x)-1])
```

which would produce the same output.
If the function f and its derivative are defined in a separate Matlab function, for example:

```
function f = cosx(x)
%%COSX Example function for Newton's
%method. Returns a 2x1 vector
%[f(x); f'(x)] with f(x) = cos(x)-x.
f =  [cos(x)-x; -sin(x)-1];
```

then the calling sequence requires the function name to be preceded by an `@`:

```
>> [x, fx, fg] = newt(@cosx,1)
```

which again produces the same output.
It is sometimes impractical to apply Newton's method because of the complexity of differentiating the function f. In any event, differentiation can be avoided by using the idea of complex step differentiation introduced in Example 29.6.

Example 29.8 (Newton's Method II) *Adapt the Matlab function in Example 29.7 so as that it generates the derivative $f'(x)$ by complex step differentiation.*

A possible adaptation is shown below. Most of the comment lines have been removed to avoid undue repetition—the only significant change is the first line after the `while` statement.

```
function [x0,f0,flag] = newtc(fun,x0,tol,nlim)
%%NEWTC Newton's Method with complex
%step derivative for solving fun(x) = 0.
% See newt.m for inputs and outputs.

if nargin < 4, nlim = 20; end
if nargin < 3, tol  = 1e-10; end
if nargin < 2, x0   = 1; end
f0 = fun(x0);
n = 0; h = 1e-10;
while abs(f0)>tol && n<nlim
    df = imag(fun(x0+1i*h))/h;
    x0 = x0 - fun(x0)/df;
    f0 = fun(x0);
    n = n+1;
end
flag = n<nlim;
   % +1 (true) if interation converged
```

To determine the root of $\cos(x) = x$:

```
>> f = @(x) cos(x)-x;
>> [x, fx, fg] = newtc(f, 1)
x =
    0.7391
fx =
    0
fg =
     1
```

where the anonymous function now returns a scalar value.

Example 29.9 (van der Pol Oscillator)
Solve the system of two ordinary differential equations defined by $\boldsymbol{u}' = \boldsymbol{f}(\boldsymbol{u}(t))$, *where*

$$\boldsymbol{u}(t) = \begin{bmatrix} x(t) \\ y(t) \end{bmatrix}, \quad \boldsymbol{f}(\boldsymbol{u}(t)) = \begin{bmatrix} \mu(1-y^2)x - y \\ x \end{bmatrix},$$

on the interval $0 \le t \le 20$ *from the starting point* $\boldsymbol{u}(0) = \left[\begin{smallmatrix} 0 \\ 2 \end{smallmatrix}\right]$ *for parameter values* $\mu = 0.1, 5, 10$. *[When* $x(t)$ *is eliminated from the system it is found that* $y(t)$ *satisfies a second order differential equation known as the van der Pol oscillator. It models the current in a certain nonlinear electrical circuit.]*

The code required is listed below and the output shown in Fig. 23. An anonymous function `udot` is defined to represent the right hand side of the differential equation. The main section is a `for` loop that solves the system for each parameter value in turn. It is convenient to hold the parameter values μ and the corresponding line styles for the graphics in arrays. The line style array `ls` is a 3×3 character array so the first row (`'b- '`) has to be padded with a space to fill the three available slots[5]. These are used only in the phase portrait.

```
%VDPOL Integrate the van der Pol Oscillator
udot = @(t,u,mu) ...
    [mu*(1-u(2)^2)*u(1)-u(2); u(1)];
set(groot,'defaulttextinterpreter','latex');
set(groot,'defaultaxesfontsize',20);
% Assign parameter values to a vector M
M = [.1, 5, 10];
% Line styles to be used
ls = ['b- ';'k-.';'r--'];
for j = 1:3
    mu = M(j);
    [t,u] = ode45(udot,[0,20],[0,2],[],mu);
    figure(1)
    subplot(3,1,j)
```

[5]This could be avoided by the use of *cell arrays*, see Higham & Higham [4, Chapter 18.3]

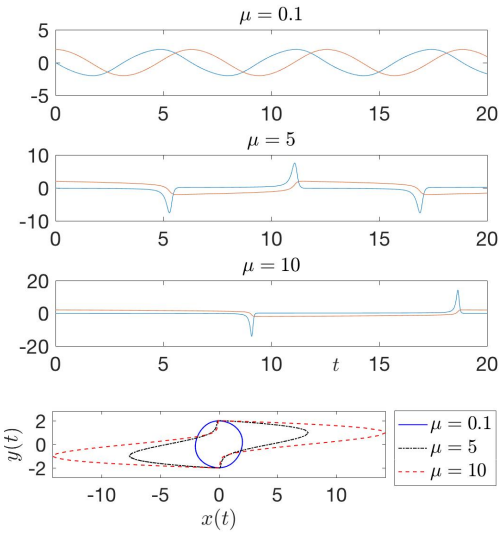


Fig. 23: Top: Solutions of the van der Pol oscillator as functions of time. Bottom: Phase portrait of $y(t)$ versus $x(t)$

```
    plot(t,u)
    title(['$\mu= ' num2str(mu) '$'],...
        'fontsi',20)
    figure(2)
    plot(u(:,1),u(:,2),ls(j,:),'linewi',1)
    hold on
end
axis equal
hold off
pbaspect([1,.2,1])
L=legend('$\mu = 0.1$','$\mu = 5$',...
    '$\mu = 10$');
set(L,'fontsi',20,...
    'Location','southeastoutside',...
    'interpreter','latex')
xlabel('$x(t)$'), ylabel('$y(t)$')
```

The solution of the system is accomplished by one of the more popular ODE solvers `ode45`. There are two outputs, a vector of times covering the specified interval and the corresponding solution (if `t` is an $n \times 1$ array, then `u` is an $n \times 2$ array).
We have called the function with five inputs. These are (i) the name of the function defining the system, (ii) the time interval, (iii) the initial value, (iv) an empty array `[]` (this argument can be used to modify certain options that we do not explore), and (v) the parameter value.

30 Case Study

In this section ten functions are described that perform the same task and are compared for transparency and efficiency. This provides an opportunity to apply some of the techniques discussed up to this point as well as introducing some new ideas.

Example 30.1 (Fibonnaci) *Construct functions that will return the nth Fibonnaci number f_n, where $f_0 = 0, f_1 = 1$ and*

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

(See Example 19.1.) The functions should:

- **Input**: Non-negative integer n
- **Output**: f_n

For functions that use arrays it should be borne in mind in that indexing in Matlab starts at 1, so `F(n+1)` corresponds to f_n.

Method 1: File `fib1.m`

```
function f = fib1(n)
%%FIB1 For loop & array used to return
%nth term in the Fibonacci sequence.
% F(n+1) corresponds to f(n)
F  = zeros(1,n+1);
if n>0
    F(2) = 1;
    for i = 3:n+1
        F(i) = F(i-1) + F(i-2);
    end
else
    error('Argument not a positive integer')
end
f = F(end);
```

This code resembles that given in Example 19.1. It has been enclosed in a function m–file and given an appropriate header. The most significant change is the line `F=zeros(1,n+1)` which serves to both define the value of `F(1)` (i.e., f_0) and to allocate sufficient memory to store a vector to hold the first $n + 1$ Fibonacci numbers. Had this not been done the length of the vector `F` would be extended on each trip around the loop leading to additional costs. The time penalties this would incur in this example would not be significant (since, with modest values of n, it computes in a tiny fraction of a second) but could be important when dealing with large arrays in codes that are run many times over. An error message is issued if the function is called with a non-positive argument (this should also be included in the alternatives below).

Method 2: File `fib2.m`

The first version was rather wasteful of memory—all the entries in the sequence were saved when only the last one is required. This version removes the need for a vector. The incoming and outgoing values to the loop can be visualised as:

Incoming values: `f1` `f2`

f_0 f_1 $\cdots$ f_{i-2} f_{i-1} $f_i = f_{i-2} + f_{i-1}$

Outgoing values: `f1` `f2 = f1 + f2`

```
function f = fib2(n)
%FIB2 Fibonacci without a vector
if n==0
   f = 0;
elseif n==1
   f = 1;
else
   f1 = 0; f2 = 1;
   for i = 2:n
     f  = f1 + f2;
     f1 = f2; f2 = f;
   end
end
```

Method 3: File: `fib3.m`

This version replaces the `if...else...end` construct by a `switch` statement. See `help switch`.

```
function f = fib3(n)
%FIB3 Fibonacci using a switch
switch n
    case 0
        f = 0;
    case 1
        f = 1;
    otherwise
        f1 = 0; f2 = 1;
        for i = 2:n
            f  = f1 + f2;
            f1 = f2; f2 = f;
        end
end
```

Method 4: File `fib4.m`

The solution of the recurrence can be written in terms of a matrix power:

$$\begin{bmatrix} f_{n-1} \\ f_n \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} f_{n-2} \\ f_{n-1} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^{n-1} \begin{bmatrix} f_0 \\ f_1 \end{bmatrix}.$$

```
function f = fib4(n)
%FIB4 Fibonacci-matrix version
A = [0 1;1 1];
f = [0 1]* A^(n-1)*[0;1];
```

Method 5: File `fib5.m`
The matrix `A` in `fib4.m` can be diagonalised: $AV = VD$, where

$$V = \begin{bmatrix} \lambda & 1 \\ -1 & \lambda \end{bmatrix}, \qquad D = \begin{bmatrix} -1/\lambda & 0 \\ 0 & \lambda \end{bmatrix}.$$

$\lambda = \frac{1}{2}(\sqrt{5}+1)$ and $-1/\lambda$ are its eigenvalues. Consequently $A^{n-1} = VD^{n-1}V^{-1}$, leading to `fib5`.

```
function f = fib5(n)
%FIB5 Matrix A in fib4 is diagonalized
L = (sqrt(5)+1)/2;
d = 1 + L^2;
f = ((-1)^(n+1)/L^(n-1) + L^(n+1))/d;
```

Method 6: File `fib6.m`
The recursion may be written as a system of linear algebraic equations

$$\begin{bmatrix} 1 & & & & \\ -1 & 1 & & & \\ -1 & -1 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & -1 & 1 \end{bmatrix} \begin{bmatrix} f_0 \\ f_1 \\ f_2 \\ \vdots \\ f_n \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

in which the coefficient matrix is banded.The system is solved using \.

```
function f = fib6(n)
%%FIB6 Matrix version (non-sparse)
if n==0
   f = 0;
elseif n==1
   f = 1;
else
   e = ones(n+1,1);
   f = (diag(e)-diag(e(1:end-1),-1) ...
       -diag(e(1:end-2),-2))\...
       [0;1;zeros(n-1,1)];
end
f = f(end);
```

Method 7: File `fib7.m`
The matrix in the previous method can be regarded as sparse when n is large. How large n has to be for this to be more efficient will be seen presently.

```
function f = fib7(n)
%%FIB7 Sparse matrix version
if n==0
   f = 0;
elseif n==1
   f = 1;
else
   e = ones(n+1,1);
   f = spdiags([-e -e e],-2:0,n+1,n+1)\...
         sparse(2,1,1,n+1,1);
end
f = full(f(end));
```

Method 8: File `fib8.m`
The built-in function `filter` (available in the Signal Processing Toolbox) solves the recurrence

$$a_1 y_n = b_1 x_n + b_2 x_{n-1} + \cdots + b_{nb+1} x_{n-nb} - a_2 y_{n-1} - \cdots - a_{na+1} y_{n-na}$$

for the sequence $\{y_n\}$ given the sequence $\{x_n\}$ (which is zero in this case).

```
function f = fib8(n)
%%FIB8 Fibonacci by filter
y = filter(1, [1, -1, -1],...
    zeros(1,n+1),[0 1]);
f = y(end);
```

The arguments to `filter` are (i) the '`a`' coefficient vector, (ii) the '`b`' coefficient vector, (iii) the '`x`' sequence, and (iv) the starting values. See `help filter` and its reference page for more details.

Method 9: File `fib9.m`
This version makes use of an idea called "recursion"—the function makes two calls to itself.

```
function f = fib9(n)
%FIB9 Two-recursion version
if n==0
   f = 0;
elseif n==1
   f = 1;
else
   f = fib9(n-1) + fib9(n-2);
end
```

Method 10: File `fib10.m`
This version again features recursion, but is written so there is only one recursive call.

```
function f = fib10(n, f1, f2)
%FIB10 Single-recursion version
if nargin < 3
    f1 = 0; f2 = 1;
end
if n == 0
    f = f1;
else
```

```
    f = fib10(n-1, f2, f1+f2);
end
```

The command `fib10(9)`, with a single argument, computes the 9th Fibonacci number.

Assessment: `fib1` resembles the mathematical formulation but the use of a vector is unnecessary; `fib4` and `fib5` avoid a for loop ; `fib5` does not return an integer so that the results should be rounded; the formulations used by `fib6` and `fib7` can be useful in the theoretical analysis of recurrence relations but are over-elaborate in this context; `fib8` is concise because it exploits a built-in function but the process involved is unknown; `fib9` most resembles the mathematical formulation but we shall see that it is completely impractical for values of n above about 15; `fib10`, the other recursive function, is less transparent but much quicker than `fib9`, suggesting that it is the use of two recursive calls that causes a bottleneck.
It can be deduced from Method 5 that $f_n \approx \lambda^{n-1}$ for large n and the largest integer available in Matlab is given by

```
>> intmax
ans =
  int32
   2147483647
```

from which it follows that the largest Fibonacci number that can be computed exactly (without special treatment) is $f_{44} = 701408733$.

The efficiency of the methods is assessed by the times taken to calculate f_n, $(n = 1 : 44)$, repeatedly five hundred times. The results are shown in Fig. 24 using `semilogy` which uses a logarithmic scale on the y-axis. Rather than show the average time for each calculation (which is sensitive to current activity on the computer) we plot the minimum time for each n—this removes most of the noise and shows each method in its best possible light. The average times for `fib6` are shown by a dashed line for comparison.
When a function file is executed for the first time the code is compiled in order to speed up subsequent calls. The command

```
>> clear functions
```

is issued at the start of each repetition which clears this compiled code and each run starts afresh. The additional "compile time" is the reason for the larger times at $n = 1$.
The most striking feature of Fig. 24 is the explosive growth in the times for Method 9. Methods 10 and 6 have a clear linear growth and the times for the remaining methods remains remarkably constant. Method 9 apart, the range of times varies by around a factor of 60. The linear growth of the times is more evident for larger values[6] of n, as seen in Fig. 25. There is no reason that Methods 4 & 5 should vary with n since they encode formulae for the nth term. The sparse matrix solve (Method 7) overtakes Method 6 at $n \approx 150$. As n

[6]The largest real number, `realmax`, is approximately 1.8×10^{308} from which we deduce that values of f_n for $n > 1476$ are regarded as infinite (`Inf`).

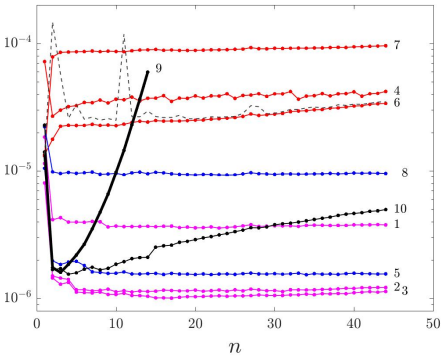


Fig. 24: Minimum time taken for each of the methods in Example 30.1 to compute each of the first 45 Fibonacci numbers. The dashed line shows the average time (over 500 calls) using Method 6

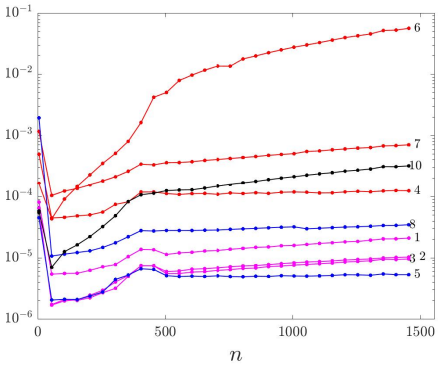


Fig. 25: Minimum time taken for each of the methods in Example 30.1 to compute each of the Fibonacci numbers f_n $(n = 5 : 50 : 1455)$.

increases from 505 to 1455, the times for Methods 4 and 5 remain constant, times for Methods 1–3 and 7 roughly double, Method 10 nearly trebles while the time for Method 6 increases by a factor of 8. Overall Methods 2 and 3 appear to be superior (discounting 5 where the answer is coded directly) and, if there is a moral to be drawn from this example, simplicity prevails!

31 Exercises

These exercises are graded as easy☆ or moderately difficult★, with intermediate ones being unmarked.

1. ☆ The conversion formula from c° Celsius to f° Fahrenheit is $f = \frac{9}{5}c + 32$. Produce a table to give the Fahrenheit readings (to two decimal places) for the Celsius temperatures $-273, 0, 15, 36.6, 100, 1000$.

2. ☆ Rings are made by bending wire of the appropriate length into circular shapes and soldering the ends together. The wire costs 0.67p per mm and the soldering costs 0.78p per ring. Produce a table with two columns showing the cost of producing rings of diameters (in mms) $10, 11, \ldots, 20$.

   What is the cost of the complete set of rings?

3. ☆ Suppose that

$$\boldsymbol{a} = \begin{bmatrix} 1 \\ 3 \\ -4 \end{bmatrix}, \quad \boldsymbol{b} = \begin{bmatrix} 2 \\ 1 \\ 6 \end{bmatrix}.$$

   Evaluate $\boldsymbol{a}^T\boldsymbol{b}$ and $\boldsymbol{a}\boldsymbol{b}^T$ and observe their sizes.

4. ☆ With

$$A = \begin{bmatrix} 2 & 2 & -3 \\ 4 & -5 & 6 \\ 7 & 0 & 3 \end{bmatrix}, \; B = \begin{bmatrix} -2 & 7 & 4 & 6 \\ 3 & 6 & 1 & 0 \\ 2 & 1 & 2 & -1 \end{bmatrix}, \; C = \begin{bmatrix} 2 & -3 \\ 0 & 6 \\ 4 & 2 \\ 3 & 7 \end{bmatrix}$$

   find AB and verify that $(ABC)^T = C^T B^T A^T$.

   Find the sizes and rank of the matrices CC^T and $C^T C$. [Hint: `help rank`.]

5. ☆ With $D = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$, $E = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$, $F = \begin{bmatrix} 1 & -6 \\ -2 & 3 \end{bmatrix}$ verify that $(DE)^{-1} = E^{-1}D^{-1}$ and $(DEF)^{-1} = F^{-1}E^{-1}D^{-1}$.

6. ☆ Express A in Exercise 4 as the sum of a symmetric matrix S and an skew-symmetric matrix U: $S = \frac{1}{2}(A + A^T)$, $U = \frac{1}{2}(A - A^T)$.

   Then $A = U + S$.

   Verify that $Q = (I + U)(I - U)^{-1}$ satisfies the conditions for an orthogonal matrix.

7. Find a basis for the nullspace of C^T in Exercise 4. If the basis vectors form the columns of a matrix Z, verify that $C^T Z = 0$.

   [Hint: `help null`.] Show that

   (a) the 4×4 matrix $[C, Z]$ has full rank so that its columns form a basis for $\mathbb{R}^4$.

   (b) $\text{rank}(C) + \text{nullity}(C^T) = m$, where m is the number of columns of C^T.

   How are these two issues linked?

8. ☆ Let $T = \begin{bmatrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 2 \end{bmatrix}$. Use `C = chol(T)` to find the Cholesky factors of T: $T = C^T C$, where C is upper triangular. Show that C^{-1} is also upper triangular.

9. ☆ Solve the system $A\boldsymbol{x} = \boldsymbol{b}$, where $A = \begin{bmatrix} 1 & -1 & 3 \\ 2 & 1 & 0 \\ 1 & 2 & 2 \end{bmatrix}$ and $\boldsymbol{b} = \begin{bmatrix} 12 \\ 3 \\ 6 \end{bmatrix}$. Check the result by computing the residual $\boldsymbol{r} = \boldsymbol{b} - A\boldsymbol{x}$.

   Find the LU factors of the matrix using `[L,U] = lu(A)` so that $A = LU$. What structure do the matrices L and U have? Why?

   Which of the commands

$$\texttt{p = U\textbackslash L\textbackslash b}, \; \texttt{q = L\textbackslash U\textbackslash b}, \; \texttt{r = L\textbackslash (U\textbackslash b)}, \; \texttt{s = U\textbackslash (L\textbackslash b)}$$

   solves $A\boldsymbol{x} = \boldsymbol{b}$: Give reasons for your choice.

10. ☆ Solve $AX = B$ using the LU factors of the matrix A defined in the previous question and $B = \begin{bmatrix} 12 & -4 \\ 3 & -3 \\ 1 & 1 \end{bmatrix}$.

11. ☆ Find the eigenvalues and eigenvectors of the matrix $M = \begin{bmatrix} 15 & 10 & -8 \\ -10 & -5 & 6 \\ 14 & 10 & -7 \end{bmatrix}$. Repeat for the matrix C of Exercise 8 and for the matrix Q of Exercise 6. Check that the eigenvalues of Q have modulus one ($|x| =$ `abs(x)`).

12. ☆ Find the eigenvalues and eigenvectors of T from Exercise 8: `[X,D]= eig(T)`.

    Verify that $TX = XD$, $X^T TX = D$ and $T = XDX^T$.

    Extract the eigenvalues (as a vector of three numbers) from the matrix D.

13. Describe the matrix constructed by the expression `J = diag(ones(1,5),1)`. Show that J is nilpotent: $J^n = 0$ for some integer n.

    What are the eigenvalues and eigenvectors of J?

    What is the rank of the matrix of eigenvectors V? What does this tell us about J—is it diagonalizable?

14.☆ The perimeter L of the ellipse with equation

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1$$

is given by the integral

$$L = 4a \int_0^{\pi/2} \sqrt{1 - e^2 \sin^2 u}\, du,$$

where e is the *eccentricity* defined by $e = \sqrt{1 - b^2/a^2}$. This is an example of an "elliptic integral" and, although it cannot be evaluated exactly except in the cases $e = 0$ and $e = 1$, the built–in function `ellipke` computes its value for any given e in the range $0 \le e \le 1$ by

```
>> [~, E] = ellipke(e^2)
>> L = 4*a*E;
```

Check that this gives the correct length for both $e = 0$ and $e = 1$.

In some of Kepler's work on planetary motion he used an approximation to L based on the perimeter of a circle whose radius is the average (arithmetic mean) of a and b:

$$L_1 = \pi(a + b).$$

Another idea is to approximate L by the perimeter of a circle whose area is equal to that of the ellipse (πab). This leads to

$$L_2 = 2\pi\sqrt{ab},$$

involving the geometric mean of a and b.

Graph the values of L, L_0, L_1, L_2 for $a = 1$ and $0 \le b \le 1$, where L_0 is the linear function of b that is exact at $b = 0$ and $b = 1$. Provide a legend for the figure.

Comment on the quality of the approximations.

15. Suppose that $a = 1 + x$, $b = 1 + 1/x$, where $x > 1$ is an irrational number (e.g. $\pi, \sqrt{2}$, the golden ratio), and n is a positive integer. The two sequences

$$\begin{array}{llll} \lfloor a \rfloor, & \lfloor 2a \rfloor, & \lfloor 3a \rfloor, \ldots, & \lfloor na \rfloor \\ \lfloor b \rfloor, & \lfloor 2b \rfloor, & \lfloor 3b \rfloor, \ldots, & \lfloor nb \rfloor, \end{array}$$

when merged and sorted, contain a list of the first N positive integers, each integer occurring exactly once, where $N = \lfloor nb \rfloor$ or $N = \lfloor nb \rfloor + 1$.

Check this assertion for several choices of x for $n = 10^p$ ($p = 1 : 6$) and tabulate both N and $\lfloor nb \rfloor$ as functions of n for each choice[7].

[Hint: `diff`, `floor`, `sort`]

$\lim_{n\to\infty} N = \infty$ is a result known as Beatty's Theorem.

16. The roots of the quadratic equation

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

labelled x_+ and x_-, are given by

$$x_\pm = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a},$$

The product of the roots satisfies $x_+ x_- = c/a$, leading to an alternative formula for x_-:

$$x_- = c/(ax_+).$$

Use Matlab to find the roots in the cases

(a) $a = 1, b = 3, c = 2$,

(b) $a = 1, b = -1000, c = 1$,

(c) $a = 1, b = 1000, c = 1$,

and produce a table with three columns: the first containing of x_+ and the second and third giving the two computed values of x_-. Print the results using `format long` and identify any discrepancies.

(a) Vectorise the calculations by storing the coefficients in the three cases as 3×1 column vectors `a`, `b` and `c`. The required roots should then be recalculated by typing in each formula once.

(b) Write a Matlab function file based on your code for part (a) that will accept *vectors* `a`, `b` and `c` as input and produce the corresponding pairs of roots as output. Test the function with suitable data.

Compare your results with those from `roots`.

[7] When $x = \mathrm{e}$ it can be shown (using Maple) that

$$10391023a = 38636751.9999999938...$$
$$28245729b = 38636752.0000000024...$$

to 18 significant digits but in Matlab, which calculates to about 16 digits, both right hand sides evaluate to the integer 38636752 so that it appears that the two sequences share a joint entry. This illustrates the hazards of rounding error when using floating point numbers to address issues concerning integers.

17.☆ Cardboard is available in rectangular sheets measuring 50cm by 60cm. A square of side $x$cm by $x$cm is cut from each corner and the sides folded up to make an open box of volume $V\text{cm}^3$. Show that

$$V = x(50 - 2x)(60 - 2x).$$

Determine (without Calculus!)

(a) the maximum volume possible and the corresponding value of x,

(b) the value of x that produces a box having a volume 9000cm^3.

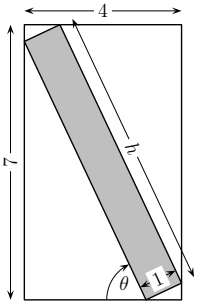
18. A storage tank is in the shape of a right circular cylinder of length 5m and has a circular cross–section of radius 2m. It lies with its axis horizontal and, when filled with liquid to a depth of x m (measured at the vertical diameter), the volume $V\text{m}^3$ of liquid is given by

$$V = 5\left(4\theta - (2 - x)\sqrt{4 - (2 - x)^2}\right)$$

where $\theta = \arccos(1 - x/2)$. Draw a graph of V as a function of x over the maximum allowable domain and give, approximately, the volume when the depth is 3m.

Use the builtin Matlab function `fzero` to determine this volume more accurately.

19.☆



A bookcase h feet high and 1 foot deep is to be moved along a long corridor 4 feet wide and 7 feet high. What is the height of the tallest bookcase that can pass down the corridor?

If the bookcase is inclined at an angle θ to the horizontal then it can be shown that h and θ must satisfy the equations

$$\left.\begin{aligned} h\cos\theta + \sin\theta &= 4 \\ h\sin\theta + \cos\theta &= 7 \end{aligned}\right\} \qquad (1)$$

or, on eliminating h, θ must be a root of the equation $f(\theta) = 0$, where

$$f(\theta) = 4\sin\theta - 7\cos\theta + \cos 2\theta.$$

Write an anonymous Matlab function file that accepts as input the value of θ and outputs the value of $f(\theta)$.

By graphing this function over a suitable range of θ values, determine the root of $f(\theta)$ correct to two decimal places and hence find the corresponding maximum height.

Also use the routine `newtc` developed in Example 29.8 to determine this volume more accurately.

20. Suppose that the sequence of matrices $\{H_k\}_{k=0}^{p}$ is defined recursively by

$$H_k = \begin{bmatrix} H_{k-1} & H_{k-1} \\ H_{k-1} & -H_{k-1} \end{bmatrix}, \qquad H_0 = [1]$$

so that H_k has size $2^k \times 2^k$. These are known as Hadamard matrices.

Show that H_k can be calculated from H_{k-1} with the command `kron`.

Verify your result using the builtin function `hadamard`.

For each $k = 2, 3, 4$ calculate $H_k^T H_k$ and show that it is a scalar multiple of the identity matrix. Conjecture a value for this scalar and hence describe the relationship between H_k^{-1} and H_k.

A Walsh matrix may be calculated by reordering the rows of a Hadamard matrix so that the number of sign changes in each row increases with row number. Calculate the Walsh matrix corresponding to H_4.

21. A cyclic matrix is determined by its first row. Each subsequent row is obtained from its predecessor by rotating one position to the right. For example

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

is defined by the vector `[1, 2, 3, 4]`. Write a Matlab function file that will input a vector and output the corresponding cyclic matrix.

All cyclic matrices of the same size have the same set of eigenvectors. Illustrate this by calculating the eigenvectors of C and showing that they diagonalise a second cyclic matrix created from the vector `[1, -3, 2, -4]`.

22. The sequence of triangles shown in Figure 26 begins with an equilateral triangle with vertices having the complex coordinates $z_n =$

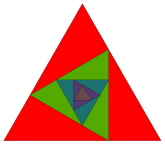


Fig. 26: Inscribed triangles

ie$^{2\pi in/3}$, $(n = 1 : 3)$. An inscribed triangle is then defined by vertices that divide each side in the ratio $r : 1 - r$ for some $0 < r < 1$ (in this case $r = \frac{1}{3}$). If the coordinates of the original triangle form a column vector `z`, show that the coordinates of the inscribed triangle can be calculated from

```
R = cyclic([r, 1-r, 0]);
z = R*z;
```

where `R` is a cyclic matrix as described in Exercise 21. The `rgb` fill–colours are also described cyclically as `c = c*R`, with `c = [1 0 0]` (red) initially.

Can you now reconstruct Fig. 26?

23.★ According to Kearsley [7]: "The existence of curves of constant diameter other than circles has been known for some time. They are mentioned in Liouville's *Journal de Mathématiques* (1860), p. 283, by Barbier, who gives Puiseux' example: the major axis of an ellipse divides the ellipse into two equal parts; if normals of length equal to the major axis are constructed on one half of the ellipse, the locus of their end points, together with the original half ellipse, form a continuous curve of constant diameter equal to the major axis of the ellipse".

Follow these instructions to draw a curve of constant diameter (see Example 29.3) equal to 2 and verify that it has constant width (see Fig. 27).

24.★ Suppose that $g(\boldsymbol{x})$ is a scalar function of $\boldsymbol{x} \in \mathbb{R}^n$. Write a Matlab function file `grad.m` that will input the handle of a function `g` and the coordinates of a point in $\mathbb{R}^n$, defined by a vector `x0`, and return the gradient of g at `x0`, calculated by "complex step differentiation" as described in Example 29.6.

Test your function by replicating the results:

```
>> g = @(x) sin(pi*x(1))*x(2)^2*x(3)^3
g =
  function_handle with value:
    @(x) sin(pi*x(1))*x(2)^2*x(3)^3
>> grad(g, [.5 ; 1; 2])
ans =
    0.0000
   16.0000
   12.0000
```

the dimension n being deduced from the length of the vector `x0`.

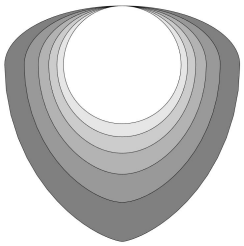


Fig. 27: Curves of constant width $d = 4/b$ $(b = 0.5 : 0.1 : 1)$ for Exercise 23 based on an ellipse with semi-minor axis 2 and major axis d

25.★ Suppose that $\boldsymbol{F}(\boldsymbol{x})$ is a vector function $\boldsymbol{F} : \mathbb{R}^n \mapsto \mathbb{R}^n$. The Jacobian $J(\boldsymbol{x})$ of $\boldsymbol{F}$ at $\boldsymbol{x}_0$ is an $n \times n$ matrix with entries

$$J_{i,j} = \frac{\partial F_i}{\partial x_j}.$$

Write a Matlab function file `jacobian.m` that will input the handle of an anonymous function `F` and the coordinates of a point in $\mathbb{R}^n$, defined by a vector `x0`, and return the Jacobian of $\boldsymbol{F}$ at `x0`, calculated by "complex step differentiation" as described in Example 29.6. Test your function by replicating the results:

```
>> F = @(x) [x(1)*x(2)*x(3); ...
                  x(2)^2; x(1)+x(3)]
F =
 function_handle with value:
  @(x)[x(1)*x(2)*x(3);x(2)^2;x(1)+x(3)]
>> J = jacobian(F, [1 2 3])
J =
     6     3     2
     0     4     0
     1     0     1
```

the dimension n being deduced from the length of the vector `x0`.

26.★ Newton's method for finding a root of the system of nonlinear equations $\boldsymbol{F}(\boldsymbol{x}) = \boldsymbol{0}$ is an iterative process that requires the set of linear equations

$$J(\boldsymbol{x}_k)\boldsymbol{\delta}^k = -\boldsymbol{F}(\boldsymbol{x}^k),$$
$$\boldsymbol{x}^{k+1} = \boldsymbol{x}^k + \boldsymbol{\delta}^k,$$

to be solved at each iteration $k = 0, 1, \ldots$ from a given starting value $\boldsymbol{x}^0$. The Jacobian J of $\boldsymbol{F}$ should be calculated using the function file developed in the previous exercise.

Use this method to determine the roots of the system (1) in Exercise 19. The starting value should be taken as $h = 6$, $\theta = \pi/3$ and the iteration terminated when $\|\boldsymbol{F}(\boldsymbol{x}^k\| < 10^{-4}$.

27.★ Use the integrator `ode45` described in Example 29.9 to solve the coupled pair of odes

$$x' = \mu x + y - x(x^2 + y^2)$$
$$y' = -x + \mu y - y(x^2 + y^2).$$

with $\mu = \pm 0.5$ to reproduce the phase portraits shown in Fig. 28. Experimentation will be needed to determine

(a) the initial conditions (these are indicated by the arrows which also show the direction of flow—replicating these arrows is not necessary),

(b) a suitable time interval $0 \le t \le T$ over which to solve equations.

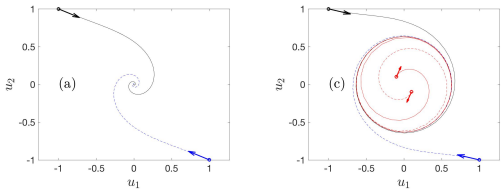
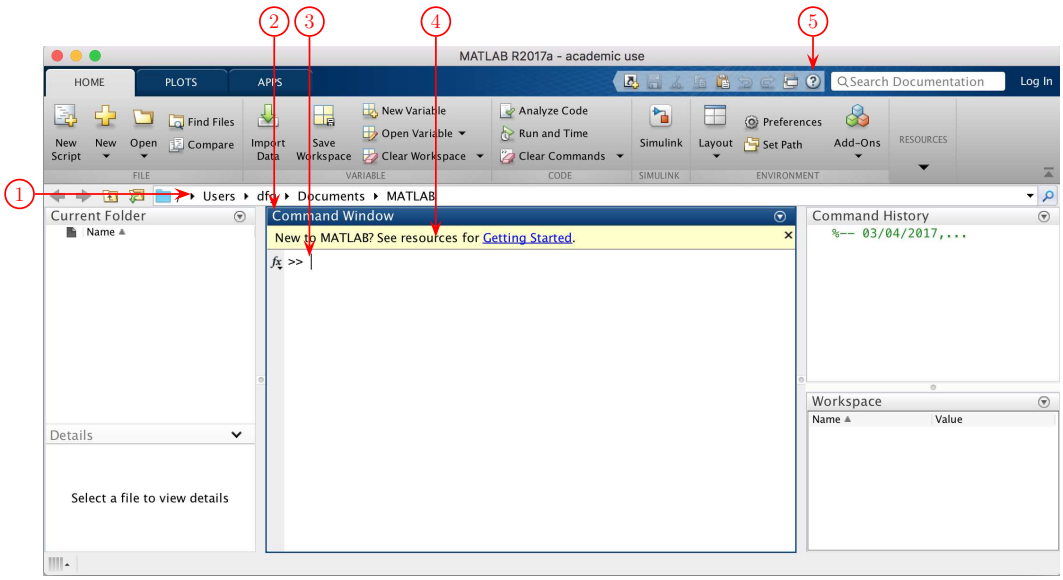


Fig. 28: Phase portrait of $y(t)$ versus $x(t)$ for Exercise 27 with $\mu = -0.5$ (left) and $\mu = 0.5$ (right)

A The Desktop I

Clicking on the MATLAB icon will bring up the MATLAB Desktop shown in Fig. 29. It features a toolstrip consisting of three tabs: HOME (which has been selected), PLOTS and APPS. Below the toolstrip are a number of panes, the one with the dark banner (Command Window) is currently selected.



1: Path to current Folder 2: Command window 3: Command prompt 4: Getting started bar 5: Help Button

Fig. 29: The MATLAB desktop at startup

The marked items are:

① the path to the startup folder. This can be edited

② the main "Command Window"

③ the "command line prompt" `>>`
This is where MATLAB commands are typed. For example, to exit from Matlab type `quit`

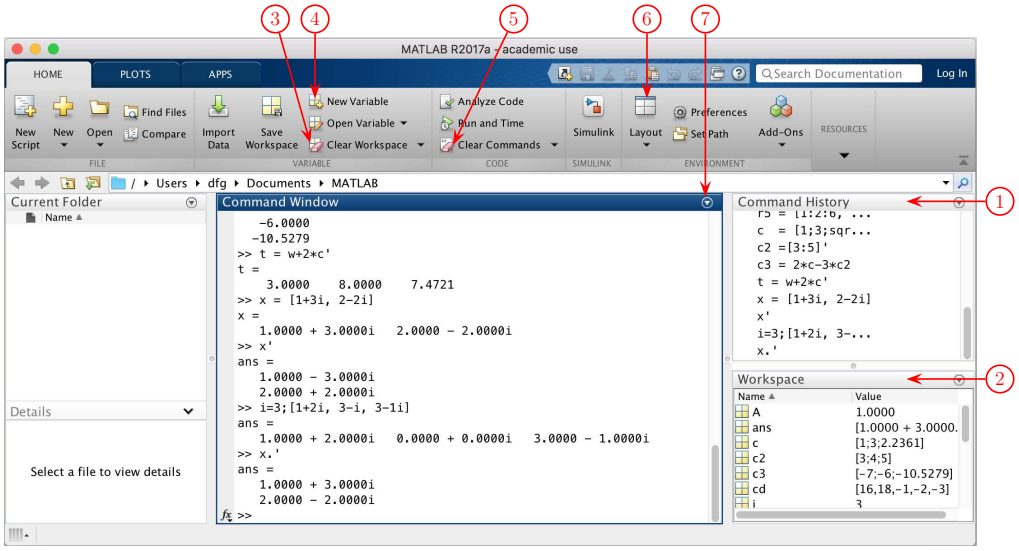
④ the "Getting started banner"

⑤ the Help button which brings up the Help browser.

There will be more about the desktop in the next section. Meantime, we recommend that you begin by clicking on "Getting started" in ④ which links to a page of Tutorials. Click the link to a short video entitled "Getting Started with MATLAB".

B The Desktop II

As a result of the MATLAB commands issued in §1–7, the desktop in Fig. 29 should now resemble that shown in Fig. 30.



1: Command History pane 3: Clear workspace 5: Clear Command Window 7: Action menu
2: Workspace pane 4: New/Open variable 6: Layout button

Fig. 30: The MATLAB desktop: a second look

① Command History: This pane keeps a record of the commands used. Clicking on any previous command will cause it to be executed again.

② Workspace: A list of variables and either their values or their size (length) and type, depending on how much space is available to display the information.

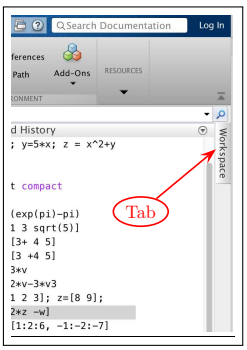
③ Clear Workspace: This clears all variables of their values. Alternatively, type "`clear`" on the command line. The command "`clear A,c`" will clear the values of the two variables `A` and `c`.

④ New/Open variable. These are described below.

⑤ Clear the the command window of both commands and output. The same effect is achieved with the command `clc`. The neighbouring triangular symbol ▼ will bring up a menu that allows the command history ① to be cleared.

⑥ A menu that allows different layouts to be selected.

⑦ ⓥ Brings up an "Actions" menu that varies according to the type of pane. One of the items will be Minimize which causes the pane to collapse to a tab.



An example of this is shown on the left for the Workspace pane. Clicking on the tab will re-open the pane (not necessarily to the same location). Clicking outwith the resulting open pane will cause it to close again. To use the same layout in future sessions choose Save Layout on the Layout menu ⑥. The position of any pane can be adjusted manually by clicking and dragging on its banner. Frames suggesting possible destinations are shown as it is being dragged. To learn more type "desktop layout" into the Help window (⑤ in Fig. 29).

Another item on this menu is Undock, which causes the pane to become detached from the desktop. The layout of the remaining panes adjusts automatically.

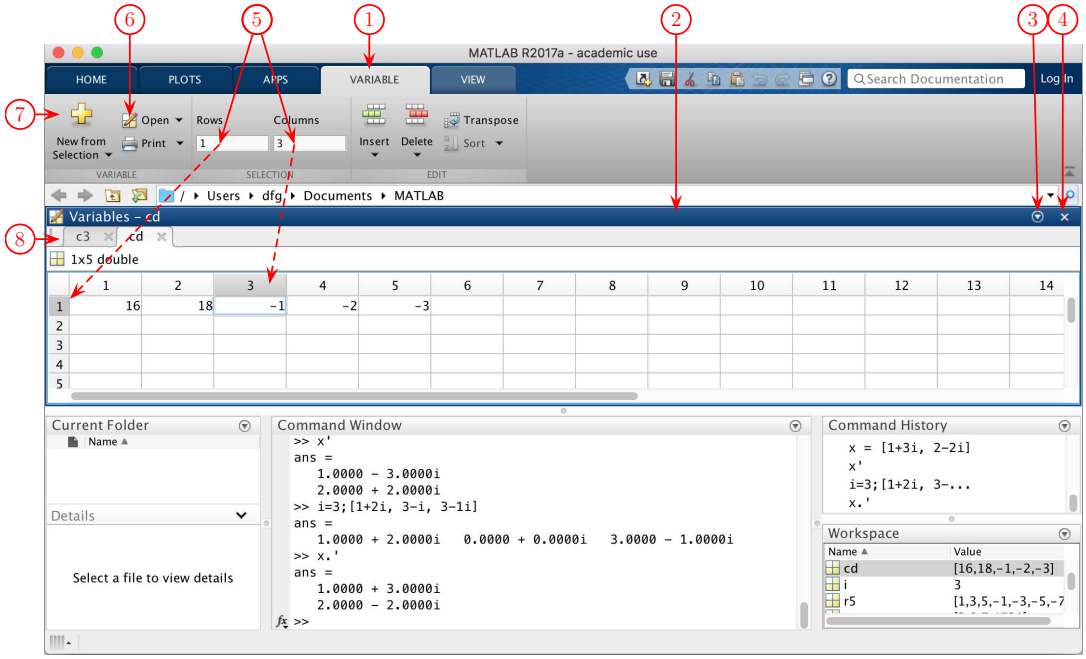


Fig. 31: The MATLAB desktop with an open Variables array editor

Clicking on "New Variable" or "Open Variable" (④ in Fig. 30) or double-clicking on a variable name in the "Workspace" pane will open the "Variables" array editor similar to that shown in Fig. 31. Some of the key features are:

① The "VARIABLE" tab. This shows a toolstrip appropriate for editing variables.

② The "Variables" array. This displays the value of the selected variable (`cd` in this case). Previously assigned values may be changed or new values assigned in any position. For example, if `7` were to be typed into row 1, column 8, then the length of `cd` would automatically be changed to 8 and intermediate cells in columns 6 and 7 would be filled with zeros.

Typing a number into row 3, column 8 would resize the variable `cd` to 3×8, again filling empty cells with zeros.

③ The "Actions" menu for the variables array. Its behaviour is similar to ⑦ in Fig. 30.

④ Close Variables array

⑤ Cells showing the coordinates of the current cursor location in the array.

⑥ Opens a tab in the Variables array to inspect the value of an existing variable.

⑦ If we were to click the ✚ symbol in the window as shown in Fig. 31, a new tab would be opened for a new variable `cd1` with the value `-1` because this is the value in the currently selected cell.

If we now click on the `cd` tab and this time select columns 2:4 (click at the start and shift-click at the end of the selection) and then click ✚, a new tab would be opened for a new variable `cd2` which has the values `[18 -1 -2]`.

⑧ Current variable tabs.

The layouts depicted in Fig. 29–31 have been set up in order to illustrate some of the many features of the Matlab desktop but are too cramped for practical purposes. We recommend minimizing the "Variables", "Current Folder" and "Command History" panes via the Actions menus ▼ on their banners. They can then readily be reopened as required. The chosen layout should be saved using the Layout ⑥ menu in Fig. 30.

If the "Workspace" pane is closed the command `who` can be used to give an alphabetical list of the variables that have currently assigned values.

```
>> who
Your variables are:
ans     v     v1     v2      x
```

The command `whos` provides more information

```
>> whos

Name  Size  Bytes Class   Attributes
ans   1x1     8   double
v     1x3    24   double
v1    1x2    16   double
x     1x2    16   double  complex
```

Among the possible attributes are `double` (the most common), `complex` (see §5), `sparse` (see §15.11), `char` (see §18) and `logical` (see §21).

C The Matlab editor

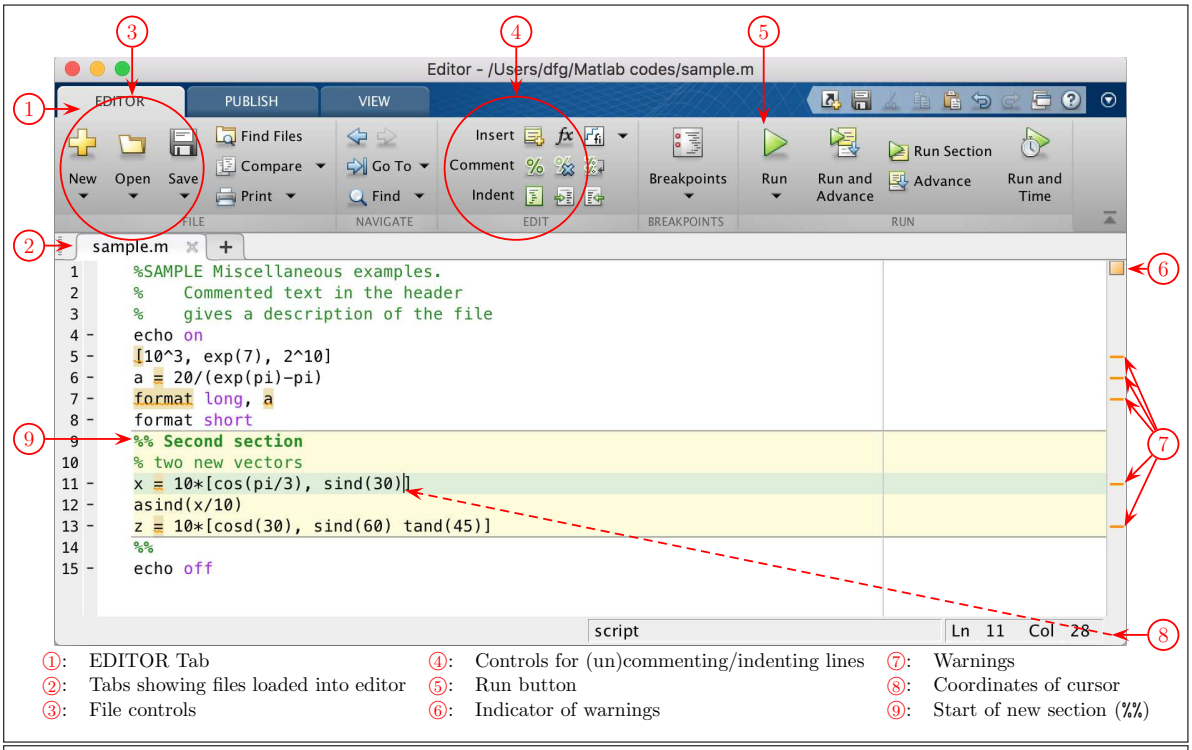
The built-in editor is the recommended way to create and edit script and function files.
Clicking on "New Script" (⑩ in Fig. 32) will bring up the Matlab editor window, similar to that shown in Fig. 32 (top) except that the main pane will be blank and the tab ② will have a temporary file name `untitled.m`. We have typed in 15 lines of code mainly taken from early sections and changed the name of the file (by using the "Save" icon ③) to `sample.m`. The lines are numbered on the left edge of the pane and those containing executable statements have a hyphen following the line number.
Lines beginning with exactly two `%%` start a new section and are followed by the section title. This is also shown on the bottom section of the "Current Folder" pane of the Desktop. The background colour of the section in which the cursor is located has a different colour to the rest of the file—a useful feature when editing larger files.
The symbol ⑥, when coloured orange, is a signal that Matlab has warnings concerning some commands. The commands in question are identified by the orange hyphens ⑦. These change to red when errors are detected. Hovering the cursor over any hyphen will give a brief description of the warning (in this instance most are caused by commands not being terminated with a semi-colon) and an offer for Matlab to remedy the situation.
When an m-file is selected from the "Current Folder" pane ⑪ the corresponding description is shown in the bottom of the pane (⑫). Any text that follows `%` on a line is rendered in a different colour (green) and is ignored by Matlab.
It is common practice when developing code to experiment with possible commands in the "Command Window" until suitable versions are identified. They can then be copied from the "Command History" pane to the editor using either "drag and drop" or "cut and paste".

D Debugging with the Editor

The editor in Matlab has a number of features to assist with debugging a file. For example, the amber symbols such as ⑥ and ⑦ shown in Fig. 32 (top) turn red for lines containing syntax errors. If the cursor is hovered over one of these red lines a pop-up box will appear with a message such as:

> Line 6: Invalid syntax at end of line.
> Possibly a), }, or] is missing.

The main use of the Matlab Editor in the debugging process involves inserting breakpoints into a file so as to temporarily pause the computation and allow the values of variables to be inspected or changed. A breakpoint is installed by clicking the hyphen next to a line number in the left edge of the editor window



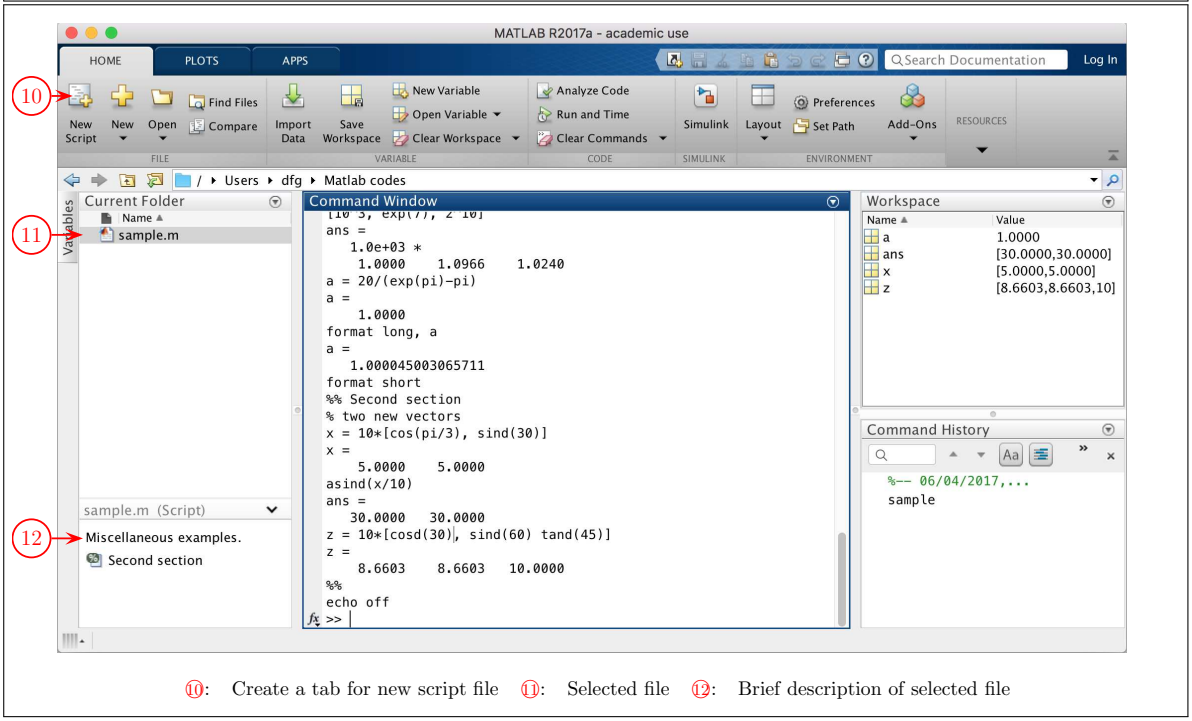


Fig. 32: The Matlab editor (top) and Desktop (bottom)

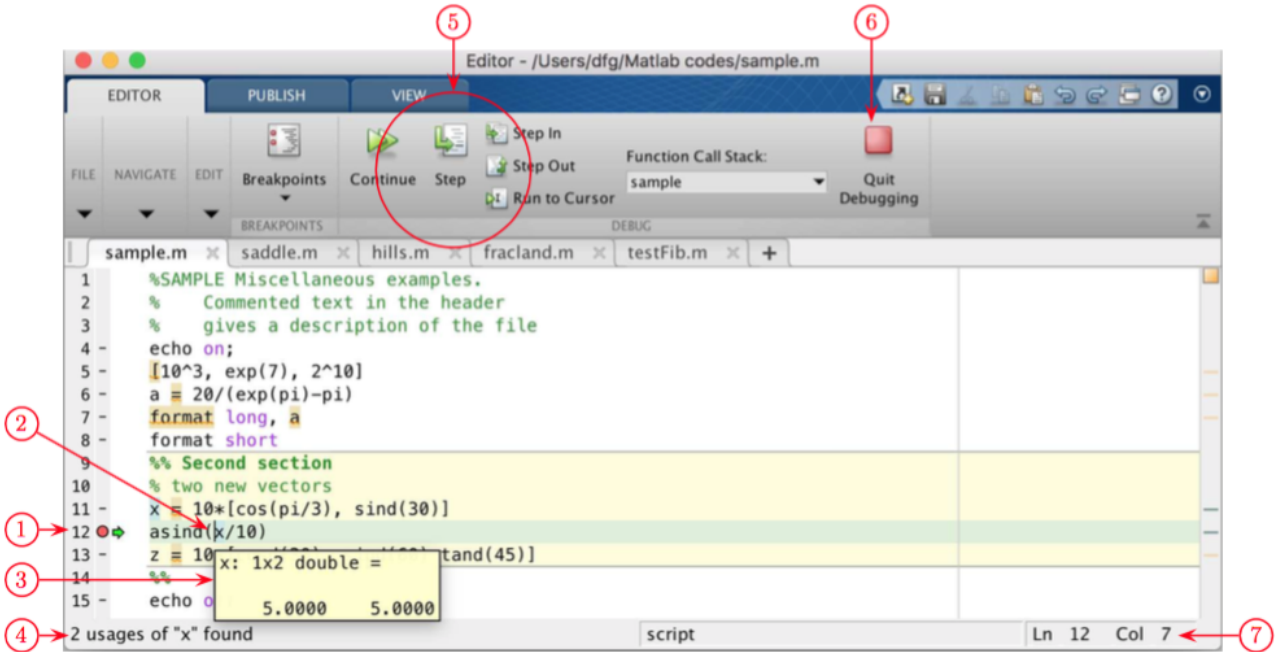


Fig. 33: The Matlab Editor in debug mode. 1: Breakpoint set & computation paused (indicated by the green arrow), 2: Cursor location, 3: Value of `x`, 4: Number of usages of variable at cursor (`x`), 5: Continue/Step controls, 6: Stop debugging, 7: Cursor coordinates

and it is indicated by a red disk (① in Fig. 33). Now when the file is run (a script file using the Run button ⑤ in Fig. 33) a green arrow in the left margin will indicate the line at which the computation has paused (① in Fig. 33). The values of all the variables calculated up to this point are now available for inspection by simply hovering the cursor over a variable name (③ in Fig. 33), by clicking on their names in the Workspace pane, or simply by typing their names in the command window.
The computation can proceed one line at a time using "Step" or advance to the next breakpoint using "Continue" (both ④ in Fig. 33), or exit debugging (⑥ in Fig. 33).
The green arrow in the left margin moves to show the next line of the file to be processed.
Computations in script or function files can also be halted by including the command `keyboard` at any point. When this command is reached control reverts to the keyboard allowing the values of variables to be inspected or changed. Computation can be resumed using `dbcont` or halted with `dbquit`.

E Data Files

Direct input of data from the 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.
When data are written to or read from a file it is crucially important that an appropriate format is used so that the data is interpreted correctly. There are 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 illustrative files used in this section are available from the web site
http://www.maths.dundee.ac.uk/software/matlab.shtml
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.

E.1 Formatted Files

Some computer codes and measurement instruments produce results in formatted data files. The data format of the files must be known in order to read these results into Matlab. Formatted files in ASCII format are written to and read from with the commands `fprintf` and `fscanf` (see §28).

Example E.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 name 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 34.

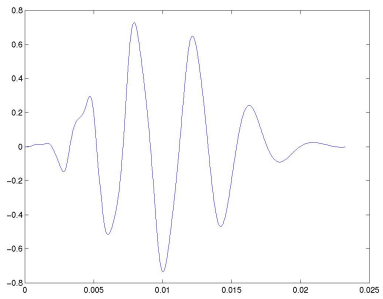


Fig. 34: Graph of "sound data" from Example E.1

E.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 E.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:

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.

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]');
```

F 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 provide 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 F.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.

```
%SPECPLOT 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
figWin = 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;');
```

```
%FIRSTPLOT 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]');
```

Example F.1 illustrates how the 'callback' property allows the programmer to define what actions should result when buttons are pushed. These actions may consist of single Matlab commands or complicated sequences of operations defined in various subroutines.
Typing the command (`>> specplot`) loading the file `bearing.bin` and selecting `log-log` brings the window shown in Fig. 35.

Exercise F.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.*

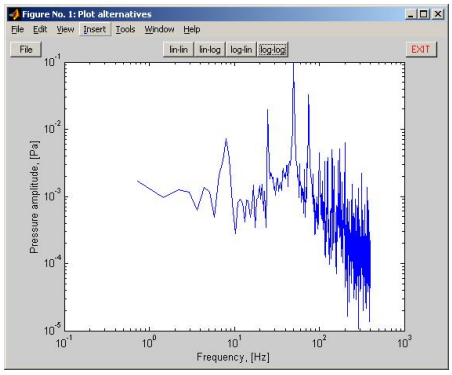


Fig. 35: Graph of "vibration data" from Example F.1

G Command Summary

Some common commands are listed in this section—a full specification of each can be obtained using the help system.

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.	
`clc`	Clear command window.
`home`	Send cursor home.
`format echo`	Echo commands inside script files.
`more`	Control paged output in command window.
Quitting MATLAB.	
`quit`	Terminate MATLAB.

Elementary Functions	
`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, radians
`sin`	Sine function, radians
`tan`	Tangent function, radians
`cosd`	Cosine function, degrees
`sind`	Sine function, degrees
`tand`	Tangent function, degrees
`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, result in radians
`acosd`	Inverse cosine, result in degrees
`acosh`	Inverse hyperbolic cosine
`asin`	Inverse sine, result in radians
`asind`	Inverse sine, result in degrees
`asinh`	Inverse hyperbolic sine
`atan`	Inverse tan, result in radians
`atand`	Inverse tan, result in degrees
`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 1: General purpose commands and Elementary Functions

Linear equations and factorisations.	
\ and /	Linear equation solution; use "help slash".
chol	Cholesky factorization.
lu	Factors from Gaussian elimination.
inv	Matrix inverse.
qr	Orthogonal-triangular decomposition.
pinv	Pseudoinverse.
Eigenvalues and singular values.	
eig	Eigenvalues and eigenvectors.
poly	Characteristic polynomial.
polyeig	Polynomial eigenvalue problem.
hess	Hessenberg form.
qz	Generalized eigenvalues.
schur	Schur decomposition.
balance	Diagonal scaling to improve eigenvalue accuracy.
svd	Singular value decomposition.
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.
Matrix functions.	
expm	Matrix exponential.
logm	Matrix logarithm.
sqrtm	Matrix square root.
funm	Evaluate general matrix function.

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.
text	Write text on figure.
print	Print or save figure to file.
plot	Linear plot.
loglog	Log-log scale plot.
semilogx	Semi-log scale plot.
semilogy	Semi-log scale plot.
contour	Contour plot.
mesh	3-D mesh surface.
surf	3-D shaded surface.
waterfall	Waterfall 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 xlabel	X-axis label.
ylabel	Y-axis label.
text	Text annotation.
gtext	Mouse placement of text.
grid	Grid lines.
view	3-D graph viewpoint specification.
zlabel	Z-axis label for 3-D plots.
gtext	Mouse placement of text.

Table 2: Numerical linear algebra, matrix functions, Graphics & plot commands.

References

[1] J. Dongarra, H. Meuer, H. Simon, and E. Strohmaier, *Biannual Top-500 Computer Lists Track Changing Environments for Scientific Computing*, SIAM News 34(9), November 2001, https://archive.siam.org/news/news.php?id=587

[2] Toby Driscoll, `unplot.m`, https://uk.mathworks.com/matlabcentral/fileexchange/2831-unplot (2003). Online: accessed 22-June-2017.

[3] David F. Griffiths and Desmond J. Higham, *Learning LaTeX, 2nd Edition*, SIAM 2016.

[4] Desmond J. Higham & Nicholas J. Higham, MATLAB *Guide 3rd ed.*, SIAM, 2017.

[5] Nicholas J. Higham, *Implicit Expansion: A Powerful New Feature of* MATLAB *R2016b*, https://nickhigham.wordpress.com/2016/09/20/implicit-expansion-matlab-r2016b/

[6] Nicholas J. Higham, *Differentiation With(out) a Difference*, SIAM News 51(5), June 2018. https://sinews.siam.org/Details-Page/differentiation-without-a-difference

[7] M. J. Kearsley, *Curves of Constant Diameter*, The Mathematical Gazette, 36: 176-179, (1952).

[8] Jimson Lee, *Usain Bolt 10 meter splits 2009*, http://speedendurance.com/2009/08/19/usain-bolt-10-meter-splits-fastest-top -speed-2008-vs-2009/

[9] C. B. Moler. *Numerical Computing with* MATLAB. SIAM, 2004.

[10] Cleve Moler, *Complex Step Differentiation*, http://blogs.mathworks.com/cleve/2013/10/14/complex-step-differentiation/

[11] Stanley Rabinowitz, *A Polynomial Curve of Constant Width.* Missouri Journal of Mathematical Sciences, 9: 23–27 (1997).

[12] L. F. Shampine, I. Gladwell & S. Thompson, *Solving* ODE*s with* MATLAB, Cambridge University Press (2003).

[13] Loren Shure, *Loren on the Art of* MATLAB*: Use nested functions to memoize costly functions*, https://blogs.mathworks.com/loren, http://blogs.mathworks.com/loren/?p=19?s_tid=srchtitle (2006).

[14] Lloyd N Trefethen and David Bau, III, *Numerical Linear Algebra*, SIAM, 1997.

[15] Wikipedia—The Free Encyclopedia, *Koch snowflake*, https://en.wikipedia.org/w/index.php?title=Koch_snowflake&oldid=783283131, Online: accessed 22-June-2017.

Index