

# M A T L A B

Notes for use with MATLAB version 5.

## 1 Introduction

MATLAB is a program that has been written to make certain numerical calculations easier on a computer. It enables us to use the computer as a high-powered ‘programmable, graphical calculator’. One of the main strengths of MATLAB (and the feature it is named after) is its ability to manipulate matrices.

The **command line** is where we type our commands; it starts with the prompt `>>`. Commands typed in at the command line are not executed before a press of the RETURN key.

In what follows pieces of text that appear in **this font** show what we should see on the computer screen as these notes are followed. Anything following the `>>` prompt represents what is typed in at the MATLAB command line.

### 1.1 Further reading

All the information you require for this course is contained within these notes (either explicitly or *via* on-line help facilities built in to MATLAB).

Further information, including some relatively advanced applications, can be found in “Numerical methods using MATLAB” by John Penny and George Lindfield. This book is in the library under classmark 517.6018-PEN.

### 1.2 Getting help and getting out

MATLAB has built-in facilities for giving help. If, for example, we require more information about the command `trace` we could type

```
>> help trace
```

If the name of a particular command is not known, but we do know (for example) that matrix inverses are of particular interest we can type

```
>> lookfor inverse
```

This will cause MATLAB to search through all<sup>1</sup> of its files (this can take a while!) and report back giving the names of commands that might be what is required. In the example above MATLAB returns over 20 possibilities including things such as ‘inverse hyperbolic cosine’. A quick look through the list tells us that `inv` is the command we wanted. If more information is required then use the `help` feature.

One other fact worth knowing before we begin is that to exit MATLAB type in the command `quit`.

---

<sup>1</sup>There is a technical issue here concerning ‘search paths’, but we can gloss over that point for the purposes of this course

## 2 Vectors and matrices

Most work carried out in MATLAB is done using vectors and matrices. We therefore begin by seeing how these quantities can be defined and manipulated.

### 2.1 Basics

A row-vector can be created by specifying the first entry, an increment and the last entry, for example

```
>> 1:3:10
ans =
     1     4     7    10
```

(The final entry in the resulting vector is arranged to be the largest member of the arithmetic progression which does not exceed the specified last entry, for example

```
>> 1:3:15
ans =
     1     4     7    10    13
```

The next entry in the progression (the value 16) is larger than 15 and is therefore not included.) Vectors can also be defined using square brackets (with entries separated by spaces or commas)

```
>> [11 2 -3 1 0]
ans =
    11     2    -3     1     0
>> [5,6,-9,12,3]
ans =
     5     6    -9    12     3
```

Arithmetic calculations can be carried out on this result

```
>> 2*ans
ans =
    10    12   -18    24     6
>> ans + 5
ans =
    15    17   -13    29    11
>> sum(ans)
ans =
    59
```

We can give names to the quantities we manipulate in MATLAB for easier reference. (If no name is given to the last result then MATLAB will call it `ans` - short for answer.)

```
>> a = [1 3 4 7]
a =
     1     3     4     7
>> b = [2 5 6 2];
```

The semicolon prevents MATLAB from echoing the last result to the screen (this is a useful feature to know about when we manipulate large vectors and matrices).

Matrices can be entered in a similar way. Either of the commands

```
>> A = [1 2 3;4 5 6;7 8 10];
>> A = [1 2 3
4 5 6
7 8 10]
A =
     1     2     3
     4     5     6
     7     8    10
```

will create a  $3 \times 3$  matrix **A**. We can use either a semicolon or a press of the RETURN key to separate rows. It is possible to alter parts of the matrix as follows

```
>> A(3,3) = 9
A =
     1     2     3
     4     5     6
     7     8     9
```

More details of this sort of procedure can be found in §2.8.

The  $2 \times 3$  matrix **B** defined below will appear in some examples to follow.

```
>> B = [1 2 5;3 5 -6]
B =
     1     2     5
     3     5    -6
```

## 2.2 Exercises

1. Define a matrix **C** by

```
>> C = magic(15)
```

Use MATLAB to verify that **C** has equal row, column and diagonal sums.

2. The commands `who` and `whos` are useful. Type them in and see what happens.

## 2.3 Built-in functions

MATLAB has many built-in functions, most with obvious names. Suppose that we want to know the cosine of each entry (in radians) of **B** above

```
>> cos(B)
ans =
    0.5403 -0.4161  0.2837
   -0.9900  0.2837  0.9602
```

More decimal places can be shown by altering the *format* of the output

```
>> format long
>> ans
ans =
    0.54030230586814  -0.41614683654714    0.28366218546323
   -0.98999249660045    0.28366218546323    0.96017028665037
```

The command

```
>> format short
```

returns us to the default output style (which is assumed in the rest of these notes). There are other styles of format available, the `help` command will list these. It should be pointed out that when using the default format MATLAB performs calculations to its maximum accuracy - only the output is truncated to a small number of significant figures.

## 2.4 Exercises

1. Create a vector **x** containing the values 0.1, 0.2, 0.3, ..., 0.6 and hence find the following functions of the entries
  - (a) The hyperbolic sine.
  - (b) The inverse tangent.
  - (c) The natural logarithm.
  - (d) The base-10 logarithm.

2. Evaluate  $\sum_{i=1}^{20} \sin(x_i)$  in which  $x_i = i\pi/20$

Answer:  $\sum_{i=1}^{20} \sin(x_i) =$

3. The `format` command was used above to increase the number of decimal places displayed by MATLAB. It may also be used to give rational number output. Find a rational number approximating  $\cos(\pi/6)$ .

Answer:  $\cos(\pi/6) \approx$  \_\_\_\_\_

## 2.5 Operations

The following matrix operations are available in MATLAB: + (addition), - (subtraction), \* (multiplication), ' (transpose), ^ (power), \ (left division) and / (right division). Multiplication and addition follow the usual rules for matrices. Using the  $3 \times 3$  matrix A and the  $2 \times 3$  matrix B defined above consider the following MATLAB session

```
>> A+B
??? Error using ==> +
Matrix dimensions must agree.
>> A*A
ans =
    30    36    42
    66    81    96
   102   126   150
>> A^2 - A*A
ans =
     0     0     0
     0     0     0
     0     0     0
>> B*A
ans =
    44    52    60
   -19   -17   -15
>> A*B
??? Error using ==> *
Inner matrix dimensions must agree
>> B^2
??? Error using ==> ^
Matrix must be square
```

The three error messages above were caused by us asking MATLAB to perform mathematically invalid operations.

If A is an invertible  $n \times n$  matrix and **b** is a vector of the appropriate size then

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

gives the  $n$ -row solution **x** to  $A*x=b$  and

```
>> x = b/A
```

gives the  $n$ -column solution **x** to  $x*A=b$ .

The command `x = inv(A)*b` has the same effect as `x=A\b`. The command `x = b*inv(A)` has the same effect as `x = b/A`. (In each case we again assume that **b** is of an appropriate shape.) Calculating inverses can be very time-consuming for large matrices, left- or right-division is more efficient.

The command ' gives the transpose of a real matrix. Using the matrix A we defined earlier,

```
>> A
A =
     1     2     3
     4     5     6
     7     8     9
>> A'
ans =
     1     4     7
     2     5     8
     3     6     9
```

as we would expect. What the ' command actually does is give the *complex-conjugate* transpose, so care is required when dealing with complex numbers. Consider

```
>> c = [1+i 2-2i 3-4i]
c =
 1.0000+ 1.0000i  2.0000- 2.0000i  3.0000-4.0000i
>> c'
ans =
 1.0000- 1.0000i
 2.0000+ 2.0000i
 3.0000+ 4.0000i
```

Use the following for non-conjugate transpose

```
>> c.'
ans =
 1.0000+ 1.0000i
 2.0000- 2.0000i
 3.0000- 4.0000i
```

There are other examples of when a full-stop changes the effect of an operation as we see after this exercise.

## 2.6 Exercise

Use MATLAB to find the solution of the system of equations

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 4 & 6 & 6 \\ 1 & 2 & 4 & 3 \\ 3 & 4 & 6 & 2 \end{pmatrix} \begin{pmatrix} w \\ x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 7 \\ 2 \\ 3 \\ 0 \end{pmatrix}.$$

Answers:  $w =$  ,  $x =$  ,  $y =$  ,  $z =$  .

## 2.7 'Pointwise' operations

If we precede the multiplication, power or either division operation with a full stop then the operation is carried out separately on each of the positions in the matrix, for example (using A again)

```
>> A.^2
ans =
     1     4     9
    16    25    36
    49    64    81
>> [1,2,3].*[4,7,4]
ans =
     4    14    12
>> [15 14 17]./[3 2 17]
ans =
     5     7     1
>> [2 3 4].\[10 12 20]
ans =
     5     4     5
>> [1 2 3]./[4 5]
??? Error using ==> ./
Matrix dimensions must agree
```

### Example

Consider the function  $f(x) = \frac{1 + \cos^2(x)}{\cosh(x)}$  for  $x$  taking certain values between 0 and 1. We might define vectors  $\mathbf{x}$  and  $\mathbf{f}$  as follows.

```
>> x = 0:.2:1
x =
     0    0.2000    0.4000    0.6000    0.8000    1.0000
>> f = (1 + cos(x).^2)./cosh(x)
f =
    2.0000    1.9220    1.7097    1.4182    1.1106    0.8372
```

The use of two pointwise operations in the definition of  $\mathbf{f}$  enables us to easily evaluate the function over the given range of  $x$  values.

Pointwise operations are an *exceptionally* useful feature when using MATLAB graphics as we will see later on.

## 2.8 Referencing parts of matrices

Let us define a vector containing the integers  $-5$  to  $5$

```
>> d = -5:5
d =
    -5    -4    -3    -2    -1     0     1     2     3     4     5
```

We can pick out parts of this vector as follows

```
>> d(1)
ans =
    -5
>> d(2:4)
ans =
    -4    -3    -2
>> [d(1) d(10) d(11)]
ans =
    -5     4     5
>> d([1 10 11])
ans =
    -5     4     5
>> d([1:2:9])
ans =
    -5    -3    -1     1     3
```

Returning to our matrix

```
>> A
A =
     1     2     3
     4     5     6
     7     8     9
```

We can isolate elements, rows or columns as shown next.

```
>> A(2,3)
ans =
     6
>> A(:,2)
ans =
     2
     5
     8
```

A colon in this context can be thought of as ‘every’ so that the command above could be read as ‘matrix A, every row, second column’.

We can reference rows in a similar way, the following command could be read as ‘matrix A, third row, every column’

```
>> A(3,:)
ans =
     7     8     9
```

The next command gives entries on the third row in the first and third columns.

```
>> A(3,[1 3])
ans =
     7     9
```

As a final example of this point we now select entries on the first two rows in all columns

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

For further illustration suppose that we want to perform row operations on

```
>> A
A =
     1     2     3
     4     5     6
     7     8     9
```

to find an upper triangular matrix that is row-equivalent to A. The row operations  $r_2 \rightarrow r_2 - 4r_1$  and  $r_3 \rightarrow r_3 - 7r_1$  will create zeros below the diagonal in the first column and may be carried out in MATLAB with the two commands

```
>> A(2,:) = A(2,:) - 4*A(1,:)
A =
     1     2     3
     0    -3    -6
     7     8     9
```

```
>> A(3,:) = A(3,:) - 7*A(1,:)
A =
     1     2     3
     0    -3    -6
     0    -6   -12
```

Finally we eliminate the non-zero below -3 with the row operation  $r_3 \rightarrow r_3 - 2r_2$  implemented by the command

```
>> A(3,:) = A(3,:) - 2*A(2,:)
A =
     1     2     3
     0    -3    -6
     0     0     0
```

which is an upper-triangular matrix.

Reference to matrices can also be carried out with only one input (rather than one each for the row and column), the thing to remember is that MATLAB counts down the columns so that if we define

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

then MATLAB gives the following responses to the inputs shown

```
>> B(3)
ans =
     1
>> B(4)
ans =
    -8
>> B([5:8])
ans =
     6     2     3     9
>> B(1:9)
ans =
     7     4     1    -8     6     2     3     9     5
```

It is possible to get MATLAB to list the entries of a matrix in the order that it has them stored, the command

```
>> B(:)
```

gives, as output, the transpose of the previous output. In this context the colon means ‘list all the entries of B in a column’.

## 2.9 Exercises

1. Using the matrix  $C = \text{magic}(10)$  create a column vector  $z$  which is equal to the 3rd column of  $C$  plus the transpose of the 6th row. What are the minimum and maximum entries in  $z$ ?

Answers: minimum = \_\_\_\_\_, maximum = \_\_\_\_\_.

2. Use the `help` command to tell you about the MATLAB function `eig`. Hence find the eigenvalues and eigenvectors of the matrix

$$A = \begin{pmatrix} 1 & 2 & 8 & 4 \\ 2 & 4 & 6 & 7 \\ 1 & 2 & 5 & 3 \\ 2 & 4 & 5 & 5 \end{pmatrix}.$$

Answers:  $\lambda_1 =$  ,  $\mathbf{x}_1 = \begin{pmatrix} \phantom{0} \\ \phantom{0} \\ \phantom{0} \\ \phantom{0} \end{pmatrix}$  ,  $\lambda_2 =$  ,  $\mathbf{x}_2 = \begin{pmatrix} \phantom{0} \\ \phantom{0} \\ \phantom{0} \\ \phantom{0} \end{pmatrix}$

$\lambda_3 =$  ,  $\mathbf{x}_3 = \begin{pmatrix} \phantom{0} \\ \phantom{0} \\ \phantom{0} \\ \phantom{0} \end{pmatrix}$  ,  $\lambda_4 =$  ,  $\mathbf{x}_4 = \begin{pmatrix} \phantom{0} \\ \phantom{0} \\ \phantom{0} \\ \phantom{0} \end{pmatrix}$

Use MATLAB to verify that  $A\mathbf{x}_k = \lambda_k\mathbf{x}_k$  ( $k = 1, 2, 3, 4$ ).

Is  $(2, -1, 0, 0)^T$  an eigenvector of  $A$ ?      Answer:

Is  $(1, 1, 0, -1)^T$  an eigenvector of  $A$ ?      Answer:

3. What is the rank of the matrix  $A$  that appeared in the previous question?

Answer:  $\text{rank}(A) =$  .

4. Use the `help` feature to familiarise yourself each of the following commands.  
(Space has been left for you to make additional notes.)

- `eye`
  
- `zeros`
  
- `ones`
  
- `size`
  
- `length`
  
- `diag`
  
- `sort`

- diff
- rem
- disp

5. Using some of the commands above create a  $20 \times 20$  matrix  $K$  with the following properties

$$(K)_{ij} = \begin{cases} i + j & \text{if } i = j \\ 1 & \text{otherwise} \end{cases}$$

Hence find the largest eigenvalue of  $K$ .

Answer:

## 3 Graphics

We now describe how MATLAB can be used to create graphical output.

### 3.1 Two-dimensional graphics

Let us start with an example, we create a vector of linearly spaced points between 0 and  $2\pi$

```
>> x = linspace(0,2*pi);  
>> plot(x,cos(x))  
>> plot(x,sin(x))
```

The first graph was replaced by the second, we can hold on to the current graph and cause subsequent plots to appear overlaid as follows

```
>> hold on  
>> y = sin(x).^2;  
>> plot(x,y,'r-.')  
>> title('Plots of sine and sine squared')
```

(Notice that we used `.^` for squaring `sin(x)` to ensure that this operation was performed entrywise.)

Use the `help plot` command to see what other options are available. There is a selection of line styles and colours to choose from.

### 3.2 Exercise

Create a plot of the function  $y(x) = e^{-x^2}$  for  $x$  in the range  $-0.5$  to  $1.5$ . Your plot should have the following features

- The curve should be a black broken line.
- The  $y$ -axis should be labelled 'This is the y axis'.
- There should be a grid of dotted lines in the background corresponding to the  $x$ - and  $y$ -axis ticks.
- The range of  $x$ -values in the graph should be  $-0.5$  to  $1.5$ .
- The range of  $y$ -values should be 0 to 1.
- The aspect ratio should be set so that equal increments on the  $x$  and  $y$  axes appear equal on your graph.

Hint: you may have to resort to the `lookfor` and `help` commands to achieve all of this.

### 3.3 Functions of two variables

Consider the function

$$f(x, y) = x^2 - \left(\frac{y}{4}\right)^2 + \left(\frac{y}{5}\right)^4$$

for  $-1 \leq x \leq 1$  and  $-4 \leq y \leq 4$ . (It can be shown that this function has a saddle point at  $(x, y) = (0, 0)$ .) We can begin by defining vectors of appropriate  $x$  and  $y$  values

```
>> x = linspace(-1,1,25);  
>> y = linspace(-4,4,25);
```

(These commands will define  $x$  and  $y$  as vectors with 25 entries, larger numbers of points will produce smoother graphics, but using too many points can cause printing to be *very* slow.) To obtain a plot of  $f$  we need to evaluate the function at every  $x, y$  combination. This is best done by defining matrices  $X$  and  $Y$  with the `meshgrid` command as follows

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

We can now evaluate the function  $f$

```
>> f = X.^2 - (Y/4).^2 + (Y/5).^4;
```

### 3.4 Exercise

Use the built-in `help` command to explain more about `meshgrid`. Examine the response to the command

```
>> [a,b] = meshgrid([1 2 3],[8 9])
```

### 3.5 Three dimensional graphics

There are many commands that can be used to generate and manipulate plots of  $f(x, y)$ . Try each of the following and note its effect - don't forget the `help` command for extra information.

```
>> surf(X,Y,f)  
>> colorbar  
>> shading interp  
>> view(-40,30)  
>> view(-45,30)  
>> close
```

```
>> surf1(X,Y,f)  
>> shading interp  
>> colormap(copper)  
>> colormap(pink)  
>> close
```

```
>> contour(X,Y,f)  
>> colorbar  
>> C = contour(X,Y,f);  
>> clabel(C)
```

### 3.6 Exercises

1. Create a surface plot of the function

$$f(x, y) = \frac{3y^4 - 4y^3 - 12y^2 + 18}{12(1 + 4x^2)} \quad (-1 \leq x \leq 1, \quad -1.5 \leq y \leq 2.5)$$

adjusting the viewing angle to best effect. Which values of **az** and **e1** in the **view** command did you decide gave the best view?

Answers: **az** = \_\_\_\_\_ , **e1** = \_\_\_\_\_ .

2. Create a contour plot of the above function arranging for there to be 20 contour lines.
3. (a) The function  $f$  has one saddle point, where is this? (You need not find the position of the saddle point analytically, try to 'spot' the location of this point from the graphs you have produced.)  
Answer: saddle point is at  $(x, y) =$  \_\_\_\_\_ .  
(b) What value does  $f$  take at the saddle point?  
Answer: value of  $f$  at the saddle point = \_\_\_\_\_ .  
(c) Produce a contour plot with only one contour line, this corresponding to the value  $f$  takes at its saddle point.

## 4 Logical operations

MATLAB can deal with logical expressions, that is expressions that are either true or false. This is achieved by attaching a numerical value to the words 'true' and 'false'; a value is 'true' if and only if it is non-zero. By way of illustration let us define two vectors

```
>> u = [1 0 2 4 0 2];  
>> v = [5 6 1 0 0 7];
```

- The following command "u and v" gives, as response, the value 1 (true) if and only if the corresponding entries in u and v are both non-zero.

```
>> u & v  
ans =  
     1     0     1     0     0     1
```

- This next command is "u or v"

```
>> u | v  
ans =  
     1     1     1     1     0     1
```

- "not u"

```
>> ~u  
ans =  
     0     1     0     0     1     0
```

- This command asks if the entries of u are equal to 2 (note the double equal sign)

```
>> u == 2  
ans =  
     0     0     1     0     0     1
```

- This next example demonstrates the 'exclusive or' function which is true if one argument is zero and the other argument is non-zero

```
>> xor(u,v)  
ans =  
     0     1     0     1     0     0
```

- Are any of the entries of v non-zero?

```
>> any(v)  
ans =  
     1
```

- Are all of the entries of v non-zero?

```
>> all(v)  
ans =  
     0
```

## 4.1 The find command

The find function identifies places where a logical expression is true, for example

```
>> find(u == 2)
ans =
     3     6
```

which tells us that the 3rd and 6th entries in u are equal to 2. We can use this feature to create index arrays as we show next, consider

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

The following command identifies which entries are greater than 13

```
>> A>13
ans =
     1     0     0     0
     0     0     0     0
     0     0     0     0
     0     1     1     0
```

The array index defined below identifies these positions

```
>> index = find( A>13 )
index =
     1
     8
    12
```

(remember that MATLAB indexes matrices by counting down the columns). We can now use this index array to alter A

```
>> A(index) = 999
A =
   999     2     3    13
     5    11    10     8
     9     7     6    12
     4  999  999     1
```

## 4.2 if, else and elseif

if evaluates a logical expression and executes a group of statements based on the value obtained. In its simplest form the syntax is typified by the following example in which a is an integer

```
>> if rem(a,2) == 0
    disp('the value is even')
end
```

Further details and a more complicated example are available via `help`.

### 4.3 Examples

These two examples should help you with the exercises that follow.

1. The following command produces a row vector found by summing down the columns of the matrix **A** ignoring any negative values.

```
>> sum( A.*(A>0) )
```

2. The command

```
>> all(diff(sort(a)))
```

in which **a** is a vector, returns the value 0 if and only if **a** contains a repeated entry.

### 4.4 Exercises

In the spaces provided write down commands which do the following tasks.

1. Devise a command which returns 0 (false) if a matrix contains a repeated entry and 1 (true) if a matrix contains distinct values.

Command:

2. Find a command which returns 1 if and only if each column of the matrix contains distinct entries.

Command:

3. Use a one-line command to sum down the diagonal of a square matrix where the sum is taken only over values larger than 1 in magnitude (i.e., values in  $(-\infty, -1) \cup (1, \infty)$ .)

Command:

4. Devise a one-line command which overwrites the diagonal entries of a square matrix so that the resulting matrix has every row sum equal to 1.

Command:

Test each of your commands on these two matrices

$$\begin{pmatrix} 1 & 4 & 7 \\ 2 & -3 & 6 \\ 7 & 1 & 4 \end{pmatrix}, \quad \begin{pmatrix} 1 & 8 & 2 & 1 \\ 3 & 2 & -3 & 6 \\ 6 & 7 & 1 & 4 \\ 7 & 2 & 3 & 8 \end{pmatrix}.$$

## 5 M-files

M-files are files containing lines of MATLAB commands. There are two different kinds of m-file.

### 5.1 Script files

These are simply lists of MATLAB commands gathered together in a file. If there is a sequence of commands that will be executed several times then it is often sensible to write these commands together in a script file and then access these commands *via* the file name. For example, create a file called `ma3vex.m` with the following contents

```
x = linspace(0,pi);
plot(x,exp(-0.5*x).*cos(5*x),'g')
hold on
plot(x,exp(-0.5*x).*sin(5*x),'r')
legend('First solution','Second solution')
axis equal
axis([0 pi -1 1])
```

This plots two solutions of a particular differential equation. To make MATLAB run through these commands we type the file name (omitting the `.m` part) at the prompt, in this case

```
>> ma3vex
```

Any sequence of MATLAB commands can be collected together in a file and run in this way. Such m-files are called scripts, or script files. Script-files behave as if we had typed in each line at the `>>` prompt. One important thing to note is that script-files ‘know about’ all of the variables in the workspace (that is, the list of variables shown in response to the command `who`), this is in contrast to *function* m-files as we see next.

### 5.2 Functions

Functions require one or more input arguments, they are self-contained and do not use MATLAB variables in the workspace. The only variables that a function will know about are those passed through the argument list or those defined as *global*. We will not discuss global variables in this course.

The following is a simple example of a function m-file which calculates the sum of the squares of the eigenvalues of a square matrix. This should be saved in the file name `sum2eig.m` in your file space.

```

function s = sum2eig(A)
% SUM2EIG Sum of squares of eigenvalues
% sum2eig(A) returns the sum of the eigenvalues squared of A

% An error message is returned (by eig) if a non-square
% matrix is entered

s = sum( eig(A).^2 );

```

Note that MATLAB ignores any statement on a line after the % character. We use this feature to put comments into m-files.

This function has some elements that are common to all MATLAB functions

- *A function definition line:* `function s = ...`  
This line defines the function name and the input/output variable(s).
- *A H1 line:* `% SUM2EIG Sum of ...`  
The 'help 1' line is the line printed out by the `lookfor` command.
- *Help text:* `% sum2eig(A) returns ...`  
If we type `help sum2eig` at the command line then the output you get will be the first block of comment lines in the file `sum2eig.m`. In the example here we have only one line of help text after the H1 line, but there can be as many as desired. The comment lines beginning `% An error ...` are separated from the help text by the blank line.

The rest of the file contains the function text which leads up to the calculation of the output variable which in this case is called `s`. We access the function in much the same way as we access built in functions, for example

```

>> sum2eig([1 0;0 3])
ans =
    10
>> C = [1 3 4;3 4 5;6 7 8];
>> k = sum2eig(C)
k =
    217.0000

```

### 5.3 Exercise

Write a function called `facsup` which takes a positive value  $x$  as its argument and, as output, gives the smallest natural number  $n$  such that  $n! > x$ .

Add to your m-file so that the user is given a warning if an invalid argument is passed to the function.

### 5.4 More input/output arguments

As hinted above, functions can have several input and output variables. Consider the following function which decomposes a square matrix into the sum of a symmetric

matrix and skew-symmetric matrix.

```
function [X,Y] = ssdecom(A)
% SSDECOM Decomposes matrix into symmetric + skew-symmetric
% [X,Y] = ssdecom(A)
% Given a square matrix A this function returns two square
% matrices X and Y such that A=X+Y, X=X.' and Y=-Y.'

if nargin==1
    warning(' Only one output argument from SSDECOM')
end

X = 0.5*( A + A.' );
Y = 0.5*( A - A.' );
```

We can access this function as follows

```
>> C = [1 7 2;-1 0 3;2 1 4];
>> [P,Q] = ssdecom(C)
P =
     1     3     2
     3     0     2
     2     2     4
Q =
     0     4     0
    -4     0     1
     0    -1     0
```

In the m-file above we have used the variable `nargout` (number of arguments to be output). In the example immediately above we typed `[P,Q] = ...` thus requesting the two outputs P and Q. If we type

```
>> a = ssdecom(C)
```

or simply

```
>> ssdecom(C)
```

then only one output is being requested and `nargout` will be equal to 1. This feature has been used to warn the user that they may have made an error.

Recall the function `eig` which we first saw at the end of §2. If this function is called with two output arguments then the function calculates eigenvalues and eigenvectors. If the function is called with just one output argument then only the eigenvalues are returned. This is an example of how the `nargout` variable can be used to affect output.

## 5.5 Summary

To summarise some of the above we now compare script m-files and function m-files.

<b>Script m-files</b>	<b>Function m-files</b>
Do not accept input arguments or return output arguments	Can accept input arguments and return output arguments
Operate on data in the workspace	Internal variables are local to the function by default
Useful for automating a series of steps you need to perform many times	Useful for extending the MATLAB language for your application

## 6 Loops and when not to use them

Loops are used to repeat a statement (or group of statements) a certain number of times.

### 6.1 ‘for’ loops

These loops repeat statements a fixed number of times, the syntax is

```
for i = start:increment:end
    statement(s)
end
```

(When using loops in an m-file it is good practice to indent statements that are within a loop, this makes the file easier for the user to read.) The *statement(s)* within the loop are executed once for each value of *i* defined by the ‘for’ line. If the increment is omitted MATLAB assumes an increment of 1. For example this loop executes 3 times as shown.

```
>> for i = 1:3
    x(i) = i+1
end
x =
    2
x =
    2    3
x =
    2    3    4
```

Each output of *x* was caused by MATLAB passing through the line *x(i)=i+1*. It is times such as this that use of a semicolon to suppress output is desirable.

An example of how nested for loops can be implemented is available via `help`.

### 6.2 ‘while’ loops

This structure is unlike ‘for’ loops in that we may not know how in advance how many loops will be necessary. The syntax is

```
while logical expression is true
    statement(s)
end
```

For example the commands

```
>> i = 1
>> while i < 4
    x(i) = i+1;
    i = i+1;
end
```

perform three loops and do the same job as the ‘for’ loop in the preceding subsection. However, the commands

```
>> x = 0;
>> change = 1;
>> while abs(change) > 0.00001
    newx = cos(x);
    change = newx-x;
    x = newx;
end
```

(which approximate the solution to  $x = \cos x$  by the iteration  $x_{n+1} = \cos x_n$  with  $x_0 = 0$ ) will run until two successive iterates differ by less than 0.00001. It is not obvious in advance how many iterations (or loops) this will take.

### 6.3 When not to use loops

If it is possible to perform a task using built-in MATLAB commands instead of loops then do so, it is almost certainly more efficient.

Consider the two scripts below. One uses a loop to create a vector with the square roots of 1 to 1000, the other uses vectorised MATLAB commands. (Don’t forget to use the `help` feature to tell you more about any unfamiliar commands you encounter.)

#### Script 1

```
clear
tic
for i = 1:1000
    b(i) = sqrt(i);
end
t=toc;
disp(['Time taken for loop method is ',num2str(t)]);
```

#### Script 2

```
clear
tic
a = 1:1000;
b = sqrt(a);
t = toc;
disp(['Time taken for vector method is ',num2str(t)]);
```

Run both of these and compare the time taken in each case. (Do this a few times to get an average time.)

One way to increase the speed of the first script (although it will still be slower than the second) is to set aside memory for `b`. If the line

```
b = 1:1000;
```

is inserted immediately before the ‘for’ loop then MATLAB will be able to set aside the 1000 ‘lumps’ of memory all in one go, instead of a ‘lump’ at a time. Summarise the results of your experiments by completing the following table.

	Typical run-time
<b>Script 1</b>	
<b>Script 2</b>	
<b>Script 1 (improved)</b>	

## 6.4 Exercise

It is given that the iteration

$$x_{n+1} = \frac{1 + x_n^2(1 - \tanh^2(x_n))}{\tanh(x_n) + x_n(1 - \tanh^2(x_n))} \quad (x_0 = 1.5)$$

converges to the unique positive solution of

$$x \tanh(x) = 1. \tag{1}$$

Use this fact to write an m-file which obtains an approximation to the solution of (1) accurate to 6 decimal places. (This can be assumed to have happened when two successive iterates differ by less than  $\frac{1}{2} \times 10^{-6}$ .)

Answer:  $x \approx$  .