CHAPTER

AN INTRODUCTION TO MATLAB® 1

Abstract

MATLAB® is a software package produced by The MathWorks, Inc. (http://www.mathworks.com) and is available on systems ranging from personal computers to super-computers and including parallel computing. In this chapter we aim to provide a useful introduction to MATLAB, giving sufficient background for the numerical methods we consider. The reader is referred to the MATLAB manual for a full description of the package.

1.1 THE SOFTWARE PACKAGE MATLAB

MATLAB is probably the world's most successful commercial numerical analysis software and the name MATLAB is derived from the phrase "matrix laboratory". It has evolved from some software written by Cleve Moler in the late 1970s to allow his students to access matrix routines in the LINPACK and EISPACK packages without the need to write FORTRAN programs. This first version of MATLAB had only 80 functions, primitive graphics and "matrix" was the only data type. Its use spread to other universities and, after it was reprogrammed in C, MATLAB was launched as a commercial product in 1984. MATLAB provides an interactive development tool for scientific and engineering problems and more generally for those areas where significant numeric computations have to be performed. The package can be used to evaluate single statements directly or a list of statements called a script can be prepared. Once named and saved, a script can be executed as an entity. The package was originally based on software produced by the LINPACK and EISPACK projects but in 2000, MATLAB was rewritten to use the newer BLAS and LAPACK libraries for fast matrix operations and linear algebra, respectively. MATLAB provides the user with:

1. Easy manipulation of matrix structures.
2. A vast number of powerful built-in routines which are constantly growing and developing.
3. Powerful two- and three-dimensional graphing facilities.
4. A scripting system which allows users to develop and modify the software for their own needs.
5. Collections of functions, called toolboxes, which may be added to the facilities of the core MATLAB. These are designed for specific applications: for example neural networks, optimization, digital signal processing, and higher-order spectral analysis.

It is not difficult to use MATLAB, although to use it with maximum efficiency for complex tasks requires experience. Generally MATLAB works with rectangular or square arrays of data, the elements of which may be real or complex. A scalar quantity is thus an array containing a single element. This is an elegant and powerful notion but it can present the user with an initial conceptual difficulty. A user schooled in many traditional computer languages is familiar with a pseudo-statement of the form A = B + C and can immediately interpret it as an instruction that A is assigned the sum of the

Numerical Methods. https://doi.org/10.1016/B978-0-12-812256-3.00010-5

values stored in B and C. In MATLAB the variables B and C may represent arrays so that *each element* of the array A will become the sum of the values of corresponding elements of B and C; that is the addition will follow the laws of matrix algebra.

There are several languages or software packages that have some similarities to MATLAB. These packages include:

Mathematica and Maple. These packages are known for their ability to carry out complicated symbolic mathematical manipulation but they are also able to undertake high precision numerical computation. In contrast MATLAB is known for its powerful numerical computational and matrix manipulation facilities. However, MATLAB also provides an optional symbolic toolbox. This is discussed in Chapter 10.

Other Matlab-style languages. Languages such as Scilab,[1] Octave,[2] and Freemat[3] are somewhat similar to MATLAB in that they implement a wide range of numerical methods, and, in some cases, use similar syntax to MATLAB.

It should noted that the languages do not necessarily have a range of toolboxes like MATLAB.

Julia. Julia[4] is a new high-level, high-performance dynamic programming language. The developers of Julia wanted, amongst other attributes, the speed of C, the general programming easy of Python, and the powerful linear algebra functions and familiar mathematical notation of MATLAB.

General purpose languages. General purpose languages such as Python and C. These languages don't have any significant numerical analysis capability in themselves but can load libraries of routines. For example Python+Numpy, Python+Scipy, C+GSL.

The current MATLAB release, version 9.4 (R2018a), is available on a wide variety of platforms. Generally MathWorks releases an upgraded version of MATLAB every six months.

When MATLAB is invoked it opens a command window. Graphics, editing, and help windows may also be opened if required. Users can design their MATLAB working environment as they see fit. MATLAB scripts and function are generally platform independent and they can be readily ported from one system to another. To install and start MATLAB, readers should consult the manual appropriate to their particular working environment.

The scripts and functions given in this book have been tested under MATLAB release, version 9.3.0.713579 (R2017b). However, most of them will work directly using earlier versions of MATLAB but some may require modification.

The remainder of this chapter is devoted to introducing some of the statements and syntax of MATLAB. The intention is to give the reader a sound but brief introduction to the power of MATLAB. Some details of structure and syntax are omitted and must be obtained from the MATLAB manual. A detailed description of MATLAB is given by Higham and Higham (2017). Other sources of information are the MathWorks website and Wikipedia. Wikipedia should be used with some care.

Before we begin a detailed discussion of the features of MATLAB, the meaning some terminology needs clarification. Consider the terms MATLAB statements, commands, functions, and keywords. If

[1] http://www.scilab.org.
[2] http://www.gnu.org/software/octave.
[3] http://freemat.sourceforge.net.
[4] https://julialang.org/.

we take a very simple MATLAB expression, like `y = sqrt(x)` then, if this is used in the command window for immediate execution, it is a command for MATLAB to determine the square root of the variable `x` and assign it to `y`. If it is used in a script, and is not for immediate execution, then it is usually called a statement. The expression `sqrt` is a MATLAB function, but it can also be called a keyword. The vast majority of MATLAB keywords *are* functions, but a few are not: for example `all`, `long`, and `pi`. The last of these is a reserved keyword to denote the mathematical constant π. Thus, the use of the four word discussed are often interchangeable.

1.2 MATRICES IN MATLAB

A two-dimensional array is effectively a table of data, not restricted to numeric data. If arrays are stacked in the third dimension, then they are three-dimensional arrays. Matrices are two-dimensional arrays that contain only numeric data or mathematical expressions where the variables of the expression have already been assigned numeric values. Thus, 23.2 and x^2 are allowed, *peter* is allowed if it is a numeric constant but not if it is a person's name. Thus a two dimension array of numeric data can legitimately be called an array or a matrix. Matrices can be operated on, using the laws of matrix algebra. Thus if $\mathbf{A}$ is a matrix, then $3\mathbf{A}$ and $\mathbf{A}^{-1}$ have a meaning, whereas, if $\mathbf{A}$ is an alpha-numeric array these statements have no meaning. MATLAB supports matrix algebra, but also allows array operations. For example, an array of data might be a financial statement, and therefore, it might be necessary to sum the 3rd through 5th rows and place the result in the 6th row. This is a legitimate array operation that MATLAB supports.

The matrix is fundamental to MATLAB and we have provided a broad and simple introduction to matrices in Appendix A. In MATLAB the names used for matrices must start with a letter and may be followed by any combination of letters or digits. The letters may be upper or lower case. Note that throughout this text a distinctive font is used to denote MATLAB statements and output, for example `disp`.

In MATLAB the arithmetic operations of addition, subtraction, multiplication, and division can be performed in the usual way on scalar quantities, but they can also be used directly with matrices or arrays of data. To use these arithmetic operators on matrices, the matrices must first be created. There are several ways of doing this in MATLAB and the simplest method, which is suitable for small matrices, is as follows. We assign an array of values to `A` by opening the **command** window and then typing

```
>> A = [1 3 5;1 0 1;5 0 9]
```

after the prompt `>>`. Notice that the elements of the matrix are placed in square brackets, each row element separated by at least one space or comma. A semicolon (`;`) indicates the end of a row and the beginning of another. When the return key is pressed the matrix will be displayed thus:

```
A =
     1     3     5
     1     0     1
     5     0     9
```

All statements are executed by pressing the return or enter key. Thus, for example, by typing `B = [1 3 51;2 6 12;10 7 28]` after the `>>` prompt, and pressing the return key, we assign values to `B`. To add the matrices in the **command** window and assign the result to `C` we type `C = A+B` and similarly if we type `C = A-B` the matrices are subtracted. In both cases the results are displayed row by row in the **command** window. Note that terminating a MATLAB statement with a semicolon suppresses any output.

For simple problems we can use the **command** window. By simple we mean MATLAB statements of limited complexity – even MATLAB statements of limited complexity can provide some powerful numerical computation. However, if we require the execution of an ordered sequence of MATLAB statements (commands) then it is sensible for these statements to be typed in the MATLAB **editor** window to create a script which must be saved under a suitable name for future use as required. There will be no execution or output until the name of this script is typed into the **command** window and the script executed by pressing return.

A matrix which has only one row or column is called a vector. A row vector consists of one row of elements and a column vector consists of one column of elements. Conventionally in mathematics, engineering, and science an emboldened upper case letter is usually used to represent a matrix, for example $\mathbf{A}$. An emboldened lower case letter usually represents a *column* vector, that is $\mathbf{x}$. The transpose operator converts a row to a column and vice versa so that we can represent a row vector as a column vector transposed. Using the superscript T in mathematics to indicate a transpose, we can write a row vector as $\mathbf{x}^\mathsf{T}$. In MATLAB it is often convenient to ignore the convention that the initial form of a vector is a column; the user can define the initial form of a vector as a row or a column.

The implementation of vector and matrix multiplication in MATLAB is straightforward. Beginning with vector multiplication, we assume that row vectors having the same number of elements have been assigned to `d` and `p`. To multiply them together we write `x = d*p'`. Note that the symbol `'` transposes the row `p` into a column so that the multiplication is valid. The result, `x`, is a scalar. Many practitioners use `.'` to indicate a transpose. The reason for this is discussed in Section 1.4.

Assuming the two matrices `A` and `B` have been assigned, for matrix multiplication the user simply types `C = A*B`. This computes `A` post-multiplied by `B`, assigns the result to `C` and displays it, providing the multiplication is valid. Otherwise MATLAB gives an appropriate error indication. The conditions for matrix multiplication to be valid are given in Appendix A. Notice that the symbol `*` must be used for multiplication because in MATLAB multiplication is not implied.

A very useful MATLAB function is `whos` (and the similar function, `who`).

These functions tell us the current content of the work space. For example, provided `A`, `B`, and `C` described above have not been cleared from the memory, then

```
>> whos
  Name       Size                    Bytes  Class
  A          3x3                        72  double array
  B          3x3                        72  double array
  C          3x3                        72  double array
```

This tells us that `A`, `B`, and `C` are all 3×3 matrices. They are stored as double precision arrays. A double precision number requires 8 bytes to store it, so each array of 9 elements requires 72 bytes; a grand total is 27 elements using 216 bytes. Consider now the following operations:

```
>> clear A
>> B = [ ];
>> C = zeros(4,4);
>> whos
  Name      Size                    Bytes  Class
  B         0x0                         0  double array
  C         4x4                       128  double array
```

Here we see that we have cleared (i.e., deleted) `A` from the memory, assigned an empty matrix to `B` and a 4×4 array of zeros to `C`.

Note that the size of matrices can also be determined using the `size` and `length` functions thus:

```
>> A = zeros(4,8);
>> B = ones(7,3);
>> [p q] = size(A)

p =
     4

q =
     8

>> length(A)

ans =
     8

>> L = length(B)

L =
     7
```

It can be seen that `size` gives the size of the matrix whereas `length` gives the number of elements in the largest dimension.

1.3 MANIPULATING THE ELEMENTS OF A MATRIX

In MATLAB, matrix elements can be manipulated individually or in blocks. For example,

```
>> X(1,3) = C(4,5)+V(9,1)
>> A(1) = B(1)+D(1)
>> C(i,j+1) = D(i,j+1)+E(i,j)
```

are valid statements relating elements of matrices. Rows and columns can be manipulated as complete entities. Thus `A(:,3)`, `B(5,:)` refer respectively to the third column of `A` and fifth row of `B`. If `B` has 10 rows and 10 columns, i.e. it is a 10×10 matrix, then `B(:,4:9)` refers to columns 4 through 9 of the matrix. The : by itself indicates all the rows, and hence all elements of columns 4 through 9. Note that in MATLAB, by default, the *lowest matrix index starts at* 1. This can be a source of difficulty when implementing some algorithms.

The following examples illustrate some of the ways subscripts can be used in MATLAB. First we assign a matrix

```
>> A = [2 3 4 5 6;-4 -5 -6 -7 -8;3 5 7 9 1; ...
              4 6 8 10 12;-2 -3 -4 -5 -6]

A =
     2     3     4     5     6
    -4    -5    -6    -7    -8
     3     5     7     9     1
     4     6     8    10    12
    -2    -3    -4    -5    -6
```

Note the use of ... (an ellipsis) to indicate that the MATLAB statement continues on the next line. Executing the following statements

```
>> v = [1 3 5];
>> b = A(v,2)
```

gives

```
b =
     3
     5
    -3
```

Thus `b` is composed of the elements of the first, third, and fifth rows in the second column of `A`. Executing

```
>> C = A(v,:)
```

gives

```
C =
     2     3     4     5     6
     3     5     7     9     1
    -2    -3    -4    -5    -6
```

Thus `C` is composed of the first, third, and fifth rows of `A`. Executing

```
>> D = zeros(3);
>> D(:,1) = A(v,2)
```

gives

```
D =
     3     0     0
     5     0     0
    -3     0     0
```

Here `D` is a 3×3 matrix of zeros with column 1 replaced by the first, third, and fifth elements of column 2 of `A`.

Executing

```
>> E = A(1:2,4:5)
```

gives

```
E =
     5     6
    -7    -8
```

Note that if we index an existing square or rectangular array with a single index, then the elements of the array are identified as follows. Index 1 gives the top left element of the array, and the index is incremented down the columns in sequence, from left to right. For example, with reference to the preceding array `C`

```
C1 = C;
C1(1:4:15) = 10

C1 =
    10     3     4     5    10
     3    10     7     9     1
    -2    -3    10    -5    -6
```

Note that in this example the index is incremented by 4.

When manipulating very large matrices it is easy to become unsure of the size of the matrix. Thus, if we want to find the value of the element in the penultimate row and last column of `A` defined previously we could write

```
>> size(A)

ans =
     5     5

>> A(4,5)

ans =
    12
```

but it is easier to use `end` thus:

```
>> A(end-1,end)

ans =
    12
```

The `reshape` function may be used to manipulate a matrix. As the name implies, the function reshapes a given matrix into a new matrix of any specified size provided it has an identical number of elements. For example a 3 × 4 matrix can be reshaped into a 6 × 2 matrix but a 3 × 3 matrix cannot be reshaped into a 5 × 2 matrix. It is important to note that this function takes each column of the original matrix in turn until the new required column size is achieved and then repeats the process for the next column. For example, consider the matrix `P`.

```
>> P = C(:,1:4)

P =
     2     3     4     5
     3     5     7     9
    -2    -3    -4    -5

>> reshape(P,6,2)

ans =
     2     4
     3     7
    -2    -4
     3     5
     5     9
    -3    -5

>> s = reshape(P,1,12);
>> s(1:10)

ans =
     2     3    -2     3     5    -3     4     7    -4     5
```

1.4 TRANSPOSING MATRICES

A simple operation that may be performed on a matrix is transposition which interchanges rows and columns. Transposition of a vector is briefly discussed in Section 1.2. In MATLAB transposition is denoted by the symbol `'`. For example, consider the matrix `A`, where

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

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

To assign the transpose of `A` to `B` we write

```
>> B = A'

B =
      1      4      7
      2      5      8
      3      6      9
```

Had we used `.'` to obtain the transpose we would have obtained the same result. However, if `A` is complex then the MATLAB operator `'` gives the complex conjugate transpose. For example

```
>> A = [1+2i 3+5i;4+2i 3+4i]

A =
   1.0000 + 2.0000i   3.0000 + 5.0000i
   4.0000 + 2.0000i   3.0000 + 4.0000i

>> B = A'

B =
   1.0000 - 2.0000i   4.0000 - 2.0000i
   3.0000 - 5.0000i   3.0000 - 4.0000i
```

To provide the transpose without conjugation we execute

```
>> C = A.'

C =
   1.0000 + 2.0000i   4.0000 + 2.0000i
   3.0000 + 5.0000i   3.0000 + 4.0000i
```

1.5 SPECIAL MATRICES

Certain matrices occur frequently in matrix manipulations and MATLAB ensures that these are generated easily. Some of the most common are `ones(m,n)`, `zeros(m,n)`, `rand(m,n)`, `randn(m,n)`, and `randi(p,m,n)`. These MATLAB functions generate $m \times n$ matrices composed of ones, zeros, uniformly distributed random numbers, normally distributed random numbers and uniformly distributed random integers, respectively. In the case of `randi(p,m,n)`, `p` is the maximum integer. If only a single scalar parameter is given, then these statements generate a square matrix of the size given by the parameter. The MATLAB function `eye(n)` generates the $n \times n$ unit matrix. The function `eye(m,n)` generates a matrix of m rows and n columns with a diagonal of ones; thus:

```
>> A = eye(3,4), B = eye(4,3)

A =
     1     0     0     0
     0     1     0     0
     0     0     1     0

B =
     1     0     0
     0     1     0
     0     0     1
     0     0     0
```

If we wish to generate a random matrix `C` of the same size as an already existing matrix `A`, then the statement `C = rand(size(A))` can be used. Similarly `D = zeros(size(A))` and `E = ones(size(A))` generates a matrix `D` of zeros and a matrix `E` of ones, both of which are the same size as matrix `A`.

Some special matrices with more complex features are introduced in Chapter 2.

1.6 GENERATING MATRICES AND VECTORS WITH SPECIFIED ELEMENT VALUES

Here we confine ourselves to some relatively simple examples thus:

`x = -8:1:8` (or `x = -8:8`) sets `x` to a vector having elements −8, −7, ..., 7, 8.
`y = -2:.2:2` sets `y` to a vector having elements −2, −1.8, −1.6, ..., 1.8, 2.
`z = [1:3 4:2:8 10:0.5:11]` sets `z` to a vector having the elements

$$[1 \quad 2 \quad 3 \quad 4 \quad 6 \quad 8 \quad 10 \quad 10.5 \quad 11]$$

The MATLAB function `linspace` also generates a vector. However, in this function the user defines the beginning and end values of the vector and the number of elements in the vector. For example

```
>> w = linspace(-2,2,5)

w =
    -2    -1     0     1     2
```

This is simple and could just as well have been created by `w = -2:1:2` or even `w = -2:2`. However, consider

```
>> w = linspace(0.2598,0.3024,5)

w =
    0.2598    0.2704    0.2811    0.2918    0.3024
```

Generating this sequence of values by other means would be more difficult. If we require logarithmic spacing then we can use

```
>> w = logspace(1,2,5)

w =
   10.0000   17.7828   31.6228   56.2341  100.0000
```

Note that the values produced are between 10^1 and 10^2, not 1 and 2. Again, generating these values by any other means would require some thought! The user of `logspace` should be warned that if the second parameter is `pi` the values run to π, not 10^π. Consider the following

```
>> w = logspace(1,pi,5)

w =
   10.0000    7.4866    5.6050    4.1963    3.1416
```

More complicated matrices can be generated by combining other matrices. For example, consider the two statements

```
>> C = [2.3 4.9; 0.9 3.1];
>> D = [C ones(size(C)); eye(size(C)) zeros(size(C))]
```

These two statements generate a new matrix `D` the size of which is double the row and column size of the original `C`; thus

```
D =
    2.3000    4.9000    1.0000    1.0000
    0.9000    3.1000    1.0000    1.0000
    1.0000         0         0         0
         0    1.0000         0         0
```

The MATLAB function `repmat` replicates a given matrix a required number of times. For example, assuming the matrix `C` is defined in the preceding statement, then

```
>> E = repmat(C,2,3)
```

replicates `C` as a block to give a matrix with twice as many rows and three times as many columns. Thus we have a matrix `E` of 4 rows and 6 columns:

```
E =
    2.3000    4.9000    2.3000    4.9000    2.3000    4.9000
    0.9000    3.1000    0.9000    3.1000    0.9000    3.1000
    2.3000    4.9000    2.3000    4.9000    2.3000    4.9000
    0.9000    3.1000    0.9000    3.1000    0.9000    3.1000
```

The MATLAB function `diag` allows us to generate a diagonal matrix from a specified vector of diagonal elements. Thus

```
>> H = diag([2 3 4])
```

generates

```
H =
     2     0     0
     0     3     0
     0     0     4
```

There is a second used of the function `diag` which is to obtain the elements on the leading diagonal of a given matrix. Consider

```
>> P = rand(3,4)

P =
    0.3825    0.9379    0.2935    0.8548
    0.4658    0.8146    0.2502    0.3160
    0.1030    0.0296    0.5830    0.6325
```

then

```
>> diag(P)

ans =
    0.3825
    0.8146
    0.5830
```

A more complicated form of diagonal matrix is the block diagonal matrix. This type of matrix can be generated using the MATLAB function `blkdiag`. We set matrices `A1` and `A2` as follows:

```
>> A1 = [1 2 5;3 4 6;3 4 5];
>> A2 = [1.2 3.5,8;0.6 0.9,56];
```

Then,

```
>> blkdiag(A1,A2,78)

ans =
  1.0000   2.0000   5.0000        0        0        0        0
  3.0000   4.0000   6.0000        0        0        0        0
  3.0000   4.0000   5.0000        0        0        0        0
       0        0        0   1.2000   3.5000   8.0000        0
       0        0        0   0.6000   0.9000  56.0000        0
       0        0        0        0        0        0  78.0000
```

The preceding functions can be very useful in allowing the user to create matrices with complicated structures, without detailed programming.

1.7 MATRIX ALGEBRA IN MATLAB

The matrix is fundamental to MATLAB and we have provided a broad and simple introduction to matrices in Appendix A.

MATLAB allows matrix equations to be simply expressed and evaluated. For example, to illustrate matrix addition, subtraction, multiplication, and scalar multiplication, consider the evaluation of the matrix equation

$$\mathbf{Z} = \mathbf{A}\mathbf{A}^{\mathsf{T}} + s\mathbf{P} - \mathbf{Q}$$

where $s = 0.5$ and

$$\mathbf{A} = \begin{bmatrix} 2 & 3 & 4 & 5 \\ 2 & 4 & 6 & 8 \end{bmatrix} \quad \mathbf{P} = \begin{bmatrix} 1 & 3 \\ 2 & -9 \end{bmatrix} \quad \mathbf{Q} = \begin{bmatrix} -7 & 3 \\ 5 & 1 \end{bmatrix}$$

Assigning `A`, `P`, `Q`, and `s`, and evaluating this equation in MATLAB we have

```
>> A = [2 3 4 5;2 4 6 8];
>> P = [1 3;2 -9];
>> Q = [-7 3;5 1];
>> s = 0.5;
>> Z = A*A'+s*P-Q

Z =
   61.5000   78.5000
   76.0000  114.5000
```

This result can be readily checked by hand!

MATLAB allows a single scalar value to be added to or subtracted from every element of a matrix. This is called explicit expansion. To illustrate this we first generate a fourth-order Riemann matrix using the MATLAB function `gallery`. This function gives the user access to a range of special matrices with useful properties. See Chapter 2 for further discussion. In the following piece of MATLAB code we use it to generate a 4×4 Riemann matrix, and then subtract 0.5 from every element of the matrix.

```
>> R = gallery('riemann',4)

R =
     1    -1     1    -1
    -1     2    -1    -1
    -1    -1     3    -1
    -1    -1    -1     4

>> A = R-0.5

A =
    0.5000   -1.5000    0.5000   -1.5000
   -1.5000    1.5000   -1.5000   -1.5000
   -1.5000   -1.5000    2.5000   -1.5000
   -1.5000   -1.5000   -1.5000    3.5000
```

In the 2016b release of MATLAB this feature has been extended to allow a row or column vector to be added to a matrix. For example

```
>> B = 2*ones(4)-[1 2 3 4]

B =
     1     0    -1    -2
     1     0    -1    -2
     1     0    -1    -2
     1     0    -1    -2

>> C = 2*ones(4)+[2 4 6 8]'

C =
     4     4     4     4
     6     6     6     6
     8     8     8     8
    10    10    10    10
```

Note that `2*ones(4)` produces a 4 × 4 matrix where each element is 2. In computing `B` the results show that 1 is subtracted from each element of the first column, the 2 from each element in the second column, the 3 from each element of the third column and so on. Similarly. in computing `C` the results show that 2 is added to each element of the first row, the 4 to each element in the second row, the 6 to each element of the third row and so on.

1.8 MATRIX FUNCTIONS

Some arithmetic operations are simple to evaluate for single scalar values but involve a great deal of computation for matrices. For large matrices such operations may take a significant amount of time. An example of this is where a matrix is raised to a power. We can write this in MATLAB as `A^p` where `p` is a scalar value and `A` is a square matrix. This produces the power of the matrix for any value of `p`. For the case where the power equals 0.5 it is better to use `sqrtm(A)` which gives the principal square root of the matrix `A`, (see Appendix A, Section A.13). Similarly, for the case where the power equals -1 it is better to use `inv(A)`. Another special operation directly available in MATLAB is `expm(A)` which gives the exponential of the matrix `A`. The MATLAB function `logm(A)` provides the principal logarithm to the base e of `A`. If `B=logm(A)` then the principal logarithm `B` is the unique logarithm for which every eigenvalue has an imaginary part lying strictly between $-\pi$ and π.

For example

```
>> A = [61 45;60 76]

A =
    61    45
    60    76
```

```
>> B = sqrtm(A)

B =
    7.0000    3.0000
    4.0000    8.0000

>> B^2

ans =
   61.0000   45.0000
   60.0000   76.0000
```

1.9 USING THE MATLAB OPERATOR FOR MATRIX DIVISION

As an example of the power of MATLAB we consider the solution of a system of linear equations. It is easy to solve the problem $ax = b$ where a and b are simple scalar constants and x is the unknown. Given a and b then $x = b/a$. However, consider the corresponding matrix equation

$$\mathbf{Ax} = \mathbf{b} \tag{1.1}$$

where **A** is a square matrix and **x** and **b** are column vectors. We wish to find **x**. Computationally this is a much more difficult problem and in MATLAB it is solved by executing the statement

```
x = A\b
```

This statement uses the important MATLAB division operator `\` and solves the linear equation system (1.1).

Solving linear equation systems is an important problem and the computational efficiency and other aspects of this type of problem are discussed in considerable detail in Chapter 2.

1.10 ELEMENT-BY-ELEMENT OPERATIONS

Element-by-element operations differ from the standard matrix operations but they can be very useful. They are achieved by using a period or dot (`.`) to precede the operator. If `X` and `Y` are matrices (or vectors), then `X.^ Y` raises each *element* of `X` to the power of the corresponding element of `Y`. Similarly `X.*Y` and `Y.\X` multiply or divide each element of `X` by the corresponding element in `Y` respectively. The form `X./Y` gives the same result as `Y.\X`. For these operations to be executed the matrices and vectors used must be the same size. Note that a period is *not* used in the operations `+` and `-` because ordinary matrix addition and subtraction *are* element-by-element operations. Examples of element-by-element operations are given as follows:

```
>> A = [1 2;3 4]

A =
     1     2
     3     4

>> B = [5 6;7 8]

B =
     5     6
     7     8
```

First we use normal matrix multiplication thus:

```
>> A*B

ans =
    19    22
    43    50
```

However, using the dot operator (.) we have

```
>> A.*B

ans =
     5    12
    21    32
```

which is element-by-element multiplication. Now consider the statement

```
>> A.^B

ans =
           1          64
        2187       65536
```

In the above, each element of A is raised to the corresponding power in B.

Element-by-element operations have many applications. An important use is in plotting graphs (see Section 1.14). For example

```
>> x = -1:0.1:1;
>> y = x.*cos(x);
>> y1 = x.^3.*(x.^2+3*x+sin(x));
```

Notice here that using the vector x of many values, allows a vector of corresponding values for y and y1 to be computed simultaneously from single statements. Element-by-element operations are in effect operations on scalar quantities performed simultaneously.

1.11 SCALAR OPERATIONS AND FUNCTIONS

In MATLAB we can define and manipulate scalar quantities, as in most other computer languages, but no distinction is made in the naming of matrices and scalars. Thus `A` could represent a scalar or matrix quantity. The process of assignment makes the distinction. For example

```
>> x = 2;
>> y = x^2+3*x-7

y =
     3

>> x = [1 2;3 4]

x =
     1     2
     3     4

>> y = x.^2+3*x-7

y =
    -3     3
    11    21
```

Note that in the preceding examples, when vectors are used the dot must be placed before the operator. This is not required for scalar operations, but does not cause errors if used.

In the case where we multiply a square matrix by itself, for example, in the form `x^2` we get the full matrix multiplication as shown below, rather than element-by-element multiplication as given by `x.^2`.

```
>> y = x^2+3*x-7

y =
     3     9
    17    27
```

A very large number of mathematical functions are directly built into MATLAB. They act on scalar quantities, arrays or vectors on an element-by-element basis. They may be called by using the function name together with the parameters that define the function. These functions may return one or more values. A small selection of MATLAB functions is given in the following table which lists the function name, the function use and an example function call. Note that all function names must be in lower case letters.

All MATLAB functions are not listed in Table 1.1, but MATLAB provides a complete range of trigonometric and inverse trigonometric functions, hyperbolic and inverse hyperbolic functions and

Table 1.1 Selected MATLAB mathematical functions

Function	Function gives	Example
`sqrt(x)`	square root of x	`y = sqrt(x+2.5);`
`abs(x)`	if x is real, is the positive value of x	
	if x is complex, is the scalar measure of x	`d = abs(x)*y;`
`real(x)`	real part of x when x is complex	`d = real(x)*y;`
`imag(x)`	imaginary part of x when x is complex	`d = imag(x)*y;`
`conj(x)`	the complex conjugate of x	`x = conj(y);`
`sin(x)`	sine of x in radians	`t = x+sin(x);`
`asin(x)`	inverse sine of x returned in radians	`t = x+sin(x);`
`sind(x)`	sine of x in degrees	`t = x+sind(x);`
`log(x)`	log to base e of x	`z = log(1+x);`
`log10(x)`	log to base 10 of x	`z = log10(1-2*x);`
`cosh(x)`	hyperbolic cosine of x	`u = cosh(pi*x);`
`exp(x)`	exponential of x, i.e., e^x	`p = .7*exp(x);`
`gamma(x)`	gamma function of x	`f = gamma(y);`
`bessel(n,x)`	nth-order Bessel function of x	`f = bessel(2,y);`

logarithmic functions. The following examples illustrate the use of some of the functions listed before:

```
>> x = [-4 3];
>> abs(x)

ans =
     4     3

>> x = 3+4i;
>> abs(x)

ans =
     5

>> imag(x)

ans =
     4

>> y = sin(pi/4)

y =
    0.7071
```

and

```
>> x = linspace(0,pi,5)

x =
         0    0.7854    1.5708    2.3562    3.1416

>> sin(x)

ans =
         0    0.7071    1.0000    0.7071    0.0000
```

and

```
>> x = [0 pi/2;pi 3*pi/2]

x =
         0    1.5708
    3.1416    4.7124

>> y = sin(x)

y =
         0    1.0000
    0.0000   -1.0000
```

Some functions perform special calculations for important and general mathematical processes. These functions often require more than one input parameter and may provide several outputs. For example, `bessel(n,x)` gives the nth-order Bessel function of x. The statement `y = fzero('fun',x0)` determines the root of the function `fun` near to `x0` where `fun` is a function defined by the user that provides the equation for which we are finding the root. For examples of the use of `fzero`, see Section 3.1. The statement `[Y,I] = sort(X)` is an example of a function that can return two output values. `Y` is the sorted matrix and `I` is a matrix containing the indices of the sort.

In addition to a large number of mathematical functions, MATLAB provides several utility functions that may be used for examining the operation of scripts. These are:

`pause` causes the execution of the script to pause until the user presses a key. Note that the cursor is turned into the symbol P, warning the script is in pause mode. This is often used when the script is operating with `echo on`.

`echo on` displays each line of script in the **command** window before execution. This is useful for demonstrations. To turn it off, use the statement `echo off`.

`who` lists the variables in the current workspace.

`whos` lists all the variables in the current workspace, together with information about their size and class, and so on.

MATLAB also provides functions related to time:

`clock` returns the current date and time in the form: <year month day hour min sec>.

`etime(t2,t1)` calculates elapsed time between `t1` and `t2`. Note that `t1` and `t2` are output from the clock function. When timing the duration of an event `tic ... toc` should be used.

`tic ... toc` times an event. For example, finding the time taken to execute a segment of script. The statement `tic` starts the timing and `toc` gives the elapsed time since the last `tic`.

`cputime` returns the total time in seconds since MATLAB was launched.

`timeit` times the operation of a function. Suppose we carry out a 8192 point Fourier transform using the MATLAB function `fft` (described in Chapter 8) then we run `fft_time = timeit(@()fft(8192))`.

The script `e4s101.m` uses the timing functions described previously to estimate the time taken to solve a 1000 $\times$ 1000 system of linear equations:

```
% e4s101.m  Solves a 5000 x 5000 linear equation system
A = rand(5000); b = rand(5000,1);
T_before = clock;
tic
t0 = cputime;
y = A\b;
tend = toc;
t1 = cputime-t0;
t2 = timeit(@() A\b);
disp('    tic-toc    cputime      timeit')
fprintf('%10.2f %10.2f %10.2f \n\n', tend,t1,t2)
```

Running script `e4s101.m` on a particular computer gave the following results:

```
    tic-toc    cputime      timeit
      2.52       5.09       2.60
```

The output shows that the three alternative methods of timing give essentially the same value. When measuring computing times the displayed times vary from run to run and the shorter the run time, the greater the percentage variation.

1.12 STRING VARIABLES

We have found that MATLAB makes no distinction in naming matrices and scalar quantities. This is also true of string variables or strings. For example, `A = [1 2; 3 4]`, `A = 17.23`, or `A = 'help'` are each valid statements and assign an array, a scalar or a text string respectively to `A`.

Characters and strings of characters can be assigned to variables directly in MATLAB by placing the string in quotes and then assigning it to a variable name. Strings can then be manipulated by specific MATLAB string functions which we list in this section. Some examples showing the manipulation of strings using standard MATLAB assignment are given below.

```
>> s1 = 'Matlab ', s2 = 'is ', s3 = 'useful'

s1 =
Matlab

s2 =
is

s3 =
useful
```

Strings in MATLAB are represented as vectors of the equivalent ASCII code numbers; it is only the way that we assign and access them that makes them strings. For example, the string `'is '` is actually saved as the vector [105 115 32]. Hence, we can see that the ASCII codes for the letters `i`, `s`, and a space are 105, 115, and 32 respectively. This vector structure has important implications when we manipulate strings. For example, we can concatenate strings, because of their vector nature, by using the square brackets as follows

```
>> sc = [s1 s2 s3]

sc =
Matlab is useful
```

Note the spaces are recognized. To identify any item in the string array we can write:

```
>> sc(2)

ans =
a
```

To identify a subset of the elements of this string we can write:

```
>> sc(3:10)

ans =
tlab is
```

we can display a string vertically, by transposing the string vector thus:

```
>> sc(1:3)'

ans =
M
a
t
```

We can also reverse the order of a substring and assign it to another string as follows:

```
>>a = sc(6:-1:1)

a =
baltaM
```

We can define string arrays as well. For example, using the string `sc` as defined previously:

```
>> sd = 'Numerical method'
>> s = [sc; sd]
Matlab is useful
Numerical method
```

To obtain the 12th column of this string we use

```
>> s(:,12)

ans =
s
e
```

Note that the string lengths must be the same in order to form a rectangular array of ASCII code numbers. In this case the array is 2×16. We now show how MATLAB string functions can be used to manipulate strings. To replace one string by another we use `strrep` as follows:

```
>> strrep(sc,'useful','super')

ans =
Matlab is super
```

Notice that this statement causes `useful` in `sc` to be replaced by `super`.

We can determine if a particular character or string is present in another string by using `findstr`. For example

```
>> findstr(sd,'e')

ans =
     4    12
```

This tells us that the 4th and 12th characters in the string are ‘e’. We can also use this function to find the location of a substring of this string as follows

```
>> findstr(sd, 'meth')

ans =
    11
```

The string `'meth'` begins at the 11th character in the string. If the substring or character is not in the original string, we have the result illustrated by the example below:

```
>> findstr(sd,'E')

ans =
     [ ]
```

We can convert a string to its ASCII code equivalent by either using the function `double` or by invoking any arithmetic operation. Thus, operating on the existing string `sd` we have

```
>> p = double(sd(1:9))

p =
    78   117   109   101   114   105    99    97   108

>> q = 1*sd(1:9)

q =
    78   117   109   101   114   105    99    97   108
```

Note that in the case where we are multiplying the string by 1, MATLAB treats the string as a vector of ASCII equivalent numbers and multiplies it by 1. Recalling that `sd(1:9) = 'Numerical '` we can deduce that the ASCII code for N is 78 and for u it is 117, etc.

We convert a vector of ASCII code to a string using the MATLAB `char` function. For example

```
>> char(q)

ans =
Numerical
```

To increase each ASCII code number by 3, and then to convert to the character equivalent we have

```
>> char(q+3)

ans =
Qxphulfdo

>> char((q+3)/2)

ans =
(<84:6327

>> double(ans)

ans =
    40    60    56    52    58    54    51    50    55
```

`char(q)` has converted the ASCII string back to characters. Here we have shown that it is possible to do arithmetic on the ASCII code numbers and, if we wish, convert back to characters. If after manipulation the ASCII code values are non-integer, they are rounded down.

It is important to appreciate that the string '123' and the number 123 are not the same. Thus

```
>> a = 123

a =
   123

>> s1 = '123'

s1 =
123
```

Using `whos` shows the class of the variables `a` and `s1` as follows:

```
>> whos
  Name      Size                    Bytes  Class
  a         1x1                         8  double array
  s1        1x3                         6  char array
```

A total of 4 elements using 14 bytes. Thus, a character requires 2 bytes, a double precision number requires 8 bytes. We can convert strings to their numeric equivalent using the functions `str2num`, `str2double` as follows:

```
>> x=str2num('123.56')

x =
  123.5600
```

Appropriate strings can be converted to complex numbers but the user should take care, as we illustrate below:

```
>> x = str2num('1+2j')

x =
1.0 + 2.0000i
```

but

```
>> x = str2num('1+2 j')

x =
   3.0000                  0 + 1.0000i
```

Note that `str2double` can be used to convert to complex numbers and is more tolerant of spaces.

```
>> x = str2double('1+2 j')

x =
1.0 + 2.0000i
```

There are many MATLAB functions which are available to manipulate strings; see the appropriate MATLAB manual. Here we illustrate the use of some functions.

`bin2dec('111001')` or `bin2dec('111 001')` returns 57.

`dec2bin(57)` returns the string '111001'.

`int2str([3.9 6.2])` returns the string '4 6'.

`num2str([3.9 6.2])` returns the string '3.9 6.2'.

`str2num('3.9 6.2')` returns 3.9000 6.2000.

`strcat('how ','why ','when')` returns the string 'howwhywhen'.

`strcmp('whitehouse','whitepaint')` returns 0 because strings are not identical.

`strncmp('whitehouse','whitepaint',5)` returns 1 because first the 5 characters of strings are identical.

`date` returns the current date, in the form `24-Aug-2011`.

A useful and common application of the function `num2str` is in the `disp` and `title` functions see Sections 1.13 and 1.14 respectively.

1.13 INPUT AND OUTPUT IN MATLAB

To output the names and values of variables, the semicolon can be omitted from assignment statements. However, this does not produce clear scripts or well-organized and tidy output. It is often better practice to use the function `disp` since this leads to clearer scripts. The `disp` function allows the display of text and values on the screen. To output the contents of the matrix `A` on the screen we write `disp(A)`. Text output must be placed in single quotes, for example,

```
>> disp('This will display this test')
This will display this test
```

Combinations of strings can be printed using square brackets `[ ]`, and numerical values can be placed in text strings if they are converted to strings using the `num2str` function. For example,

```
>> x = 2.678;
>> disp(['Value of iterate is ', num2str(x), ' at this stage'])
```

will place on the screen

```
Value of iterate is 2.678 at this stage
```

The more flexible `fprintf` function allows formatted output to the screen or to a file. It takes the form

```
fprintf('filename','format_string',list);
```

Here `list` is a list of variable names separated by commas. The filename parameter is optional; if not present, output is to the screen rather than to the filename. The format string formats the output. The basic elements that may be used in the format string are