A

A Mathematical Introduction to Matrices

Matrices are objects which have special properties and there are a number of rules which must be adhered to in order to manipulate them in a consistent (and correct) manner. A matrix can most readily be defined as an $n \times m$ array of numbers which is comprised of n rows and m columns. For example, a two-by-one matrix (two rows and one column) has the general form

$$\begin{pmatrix} a \\ b \end{pmatrix}$$

whereas a three-by-two matrix (three rows and two columns) has the general form

$$\begin{pmatrix} a & b \\ c & d \\ e & f \end{pmatrix}.$$

In these examples $a, b, \ldots, f$ may be real or complex numbers. To refer to individual elements of the matrix we use the notation $a_{i,j}$ to denote the element in the i^{th} row and the j^{th} column. Using this notation the three-by-two example could be written in the general form

$$\begin{pmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \\ a_{3,1} & a_{3,2} \end{pmatrix}.$$

Matrices for which $n = m$ (so that the number of rows equal the number of columns) are referred to as *square* matrices. If $m = 1$ then the matrix is simply a *column vector* (as in the first example above). If $n = 1$ (the matrix has only

one row) then we refer to it as a *row vector*. A scalar is simply a matrix in which both n and m are equal to one (that is, a one-by-one matrix). Throughout this text we will adopt the universal convection that both vectors and matrices are denoted by a bold font[1]. In general we shall use upper-case letters to denote matrices and lower-case for vectors.

Example A.1 *We show the rows and columns of a general three-by-three matrix.*

– *First row*

$$\begin{pmatrix} \boxed{a & b & c} \\ d & e & f \\ g & h & i \end{pmatrix}$$

– *Second row*

$$\begin{pmatrix} a & b & c \\ \boxed{d & e & f} \\ g & h & i \end{pmatrix}$$

– *Third row*

$$\begin{pmatrix} a & b & c \\ d & e & f \\ \boxed{g & h & i} \end{pmatrix}$$

– *First column*

$$\begin{pmatrix} \boxed{a} & b & c \\ \boxed{d} & e & f \\ \boxed{g} & h & i \end{pmatrix}$$

– *Second column*

$$\begin{pmatrix} a & \boxed{b} & c \\ d & \boxed{e} & f \\ g & \boxed{h} & i \end{pmatrix}$$

– *Third column*

$$\begin{pmatrix} a & b & \boxed{c} \\ d & e & \boxed{f} \\ g & h & \boxed{i} \end{pmatrix}$$

[1] Of course, other notations do exist. For example, many textbooks employ the notation of a single underbar for vectors $\underline{x}$ and a double underbar for matrices $\underline{\underline{A}}$.

Of all the operations which can be performed on matrices one of the simplest is that of transposition (or taking the transpose of a matrix); this operation is usually denoted by a subscript T or a prime $'$. If $\mathbf{A}$ is n-by-m then $\mathbf{B} = \mathbf{A}^T$ is m-by-n, where the elements of $\mathbf{B}$ are defined by

$$b_{j,i} = a_{i,j} \qquad i = 1, \cdots, n; \quad j = 1, \cdots, m;$$

the transpose is thus obtained by interchanging the rows and columns of matrix $\mathbf{A}$ to give the matrix $\mathbf{B} = \mathbf{A}^T$. If the matrix $\mathbf{A}$ is square then the operation of taking the transpose is equivalent to a reflection in the leading diagonal (which runs from the top left corner to the bottom right). Matrices for which $\mathbf{A}^T = \mathbf{A}$ are referred to as *symmetric* and those for which $\mathbf{A}^T = -\mathbf{A}$ are *anti-symmetric*[2]. In the case of three-by-three matrices the general symmetric and anti-symmetric three-by-three matrices can be written as

$$\mathbf{A}_{\text{symm}} = \begin{pmatrix} a & b & c \\ b & d & e \\ c & e & f \end{pmatrix} \text{ and } \mathbf{A}_{\text{anti}} = \begin{pmatrix} 0 & b & c \\ -b & 0 & e \\ -c & -e & 0 \end{pmatrix}.$$

where $a, b, \ldots, f \in \mathbb{C}$. We remark that if the complex conjugate transpose of a matrix (with elements $a^*_{j,i}$) is equal to the matrix then it is called *Hermitian* and if it is equal to minus its complex conjugate transpose then it is referred to as *skew Hermitian*.

We pause here to state that $\mathbf{A} = \mathbf{B}$ implies that $a_{i,j} = b_{i,j}$ for all the elements of the matrices, whereas $\mathbf{A} \neq \mathbf{B}$ only requires that $a_{i,j} \neq b_{i,j}$ for one pair (i, j).

Example A.2 *Determine the transposes of the following matrices:*

$$\begin{pmatrix} 1 & 4 & 7 \\ -4 & -3 & 4 \end{pmatrix}, \qquad \begin{pmatrix} 4 & 2 & 3 \end{pmatrix}, \qquad \begin{pmatrix} 1 & 2 & 3 \\ 2 & 0 & 2 \\ 3 & 2 & 3 \end{pmatrix}.$$

The solutions are

$$\begin{pmatrix} 1 & -4 \\ 4 & -3 \\ 7 & 4 \end{pmatrix}, \qquad \begin{pmatrix} 4 \\ 2 \\ 3 \end{pmatrix}, \qquad \begin{pmatrix} 1 & 2 & 3 \\ 2 & 0 & 2 \\ 3 & 2 & 3 \end{pmatrix}.$$

Notice elements on the leading diagonal, that is elements of the form $a_{i,i}$, remain unchanged by transposition. In the final example the transpose is equal to the original matrix and therefore the matrix is symmetric.

[2] Note that the diagonal elements of anti-symmetric matrices must be equal to zero. This can be seen by setting $i = j$ in the relation $a_{i,j} = -a_{j,i}$, which is only true if $a_{i,i} = 0$).

The usual arithmetic operations of addition, subtraction and multiplication also apply to matrices. However, there are now several additional rules (or constraints) under which these operations can be performed on two (or more) matrices. These are outlined below.

Addition and subtraction: two matrices can only be added together if they are the same size (that is, they have the same number of rows **and** the same number of columns). In this case the operation of addition is performed element by element. For example, if $\mathbf{A}, \mathbf{B}$ are both n-by-m matrices then $\mathbf{C} = \mathbf{A} + \mathbf{B}$ is defined as the matrix with elements $c_{i,j} = a_{i,j} + b_{i,j}$. A similar rule holds for subtraction.

Scalar multiplication: matrices of any size can be multiplied by scalars. The multiplication is performed element by element so that $\mathbf{C} = \lambda\mathbf{A}$ where $c_{i,j} = \lambda a_{i,j}$.

Matrix multiplication: in order to multiply two matrices $\mathbf{A}$ and $\mathbf{B}$ together the number of columns of $\mathbf{A}$ must equal the number of rows of $\mathbf{B}$. To perform the multiplication we "multiply" the first row of $\mathbf{A}$ by the first column of $\mathbf{B}$, multiplying the first element of each together and then the second ones, etc, and finally adding up all the results. This gives the element in the top left hand corner. We then proceed to multiply the first row by the second column in the same manner (and put the result in the first row, second column). Mathematically this can be written as

$$c_{i,j} = \sum_{k=1}^{m} a_{i,k} b_{k,j} \qquad i = 1, \cdots, n; \qquad j = 1, \cdots, p,$$

where $\mathbf{A}$ is n-by-m and $\mathbf{B}$ is m-by-p. Then the answer $\mathbf{C}$ is n-by-p.

This rule for matrix multiplication highlights one of their important properties, namely that the order of multiplication is important. In this example, with $\mathbf{A}$ an n-by-m matrix and $\mathbf{B}$ an m-by-p matrix, the operation "$\mathbf{A}$ *times* $\mathbf{B}$" is defined. However, the operation "$\mathbf{B}$ *times* $\mathbf{A}$" (that is, pre-multiplying matrix $\mathbf{A}$ by matrix $\mathbf{B}$) is not defined **unless** $p = n$. Even if $p = n$, in general $\mathbf{AB} \neq \mathbf{BA}$. This is equivalent to saying that, unlike scalar multiplication, matrix multiplication is not commutative.

Example A.3 *We demonstrate these concepts by an example involving two two-by-two matrices, namely*

$$\mathbf{A} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \text{ and } \mathbf{B} = \begin{pmatrix} \alpha & \beta \\ \gamma & \delta \end{pmatrix}.$$

The sum $\mathbf{C} = \mathbf{A} + \mathbf{B}$ *is determined as follows. Firstly the element* $c_{1,1}$ *is obtained by adding the corresponding elements in* $\mathbf{A}$ *and* $\mathbf{B}$*, so that*

$$\begin{pmatrix} \boxed{a} & b \\ c & d \end{pmatrix} + \begin{pmatrix} \boxed{\alpha} & \beta \\ \gamma & \delta \end{pmatrix} = \begin{pmatrix} \boxed{a+\alpha} & \\ & \end{pmatrix}.$$

Now for the $c_{1,2}$ *entry (the top right element):*

$$\begin{pmatrix} a & \boxed{b} \\ c & d \end{pmatrix} + \begin{pmatrix} \alpha & \boxed{\beta} \\ \gamma & \delta \end{pmatrix} = \begin{pmatrix} a+\alpha & \boxed{b+\beta} \\ & \end{pmatrix}.$$

Similarly for $c_{2,1}$ *(the bottom left element)*

$$\begin{pmatrix} a & b \\ \boxed{c} & d \end{pmatrix} + \begin{pmatrix} \alpha & \beta \\ \boxed{\gamma} & \delta \end{pmatrix} = \begin{pmatrix} a+\alpha & b+\beta \\ \boxed{c+\gamma} & \end{pmatrix}$$

and finally for $c_{2,2}$ *(the bottom right element) we have*

$$\begin{pmatrix} a & b \\ c & \boxed{d} \end{pmatrix} + \begin{pmatrix} \alpha & \beta \\ \gamma & \boxed{\delta} \end{pmatrix} = \begin{pmatrix} a+\alpha & b+\beta \\ c+\gamma & \boxed{d+\delta} \end{pmatrix}.$$

Example A.4 *An example of multiplication of two two-by-two matrices will serve to highlight the differences between addition and multiplication of matrices. Consider the matrices*

$$\mathbf{A} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \text{ and } \mathbf{B} = \begin{pmatrix} \alpha & \beta \\ \gamma & \delta \end{pmatrix}.$$

By our earlier rule the product $\mathbf{C} = \mathbf{AB}$ *is defined and is determined as follows. We start with the top left entry, namely* $c_{1,1}$ *(formed by multiplying the first row of* $\mathbf{A}$ *by the first column of* $\mathbf{B}$*)*

$$\begin{pmatrix} \boxed{a & b} \\ c & d \end{pmatrix} \begin{pmatrix} \boxed{\alpha} & \beta \\ \boxed{\gamma} & \delta \end{pmatrix} = \begin{pmatrix} \boxed{a \times \alpha + b \times \gamma} & \\ & \end{pmatrix};$$

now the top right entry, namely $c_{1,2}$ *(which is formed by multiplying the first row of* $\mathbf{A}$ *by the second column of* $\mathbf{B}$*)*

$$\begin{pmatrix} \boxed{a & b} \\ c & d \end{pmatrix} \begin{pmatrix} \alpha & \boxed{\beta} \\ \gamma & \boxed{\delta} \end{pmatrix} = \begin{pmatrix} a \times \alpha + b \times \gamma & \boxed{a \times \beta + b \times \delta} \\ & \end{pmatrix};$$

next the bottom left entry, namely $c_{2,1}$ *(which is formed by multiplying the second row of* $\mathbf{A}$ *by the first column of* $\mathbf{B}$*)*

$$\begin{pmatrix} a & b \\ \boxed{c} & \boxed{d} \end{pmatrix} \begin{pmatrix} \boxed{\alpha} & \beta \\ \boxed{\gamma} & \delta \end{pmatrix} = \begin{pmatrix} a\times\alpha + b\times\gamma & a\times\beta + b\times\delta \\ \boxed{c\times\alpha + d\times\gamma} & \end{pmatrix};$$

and finally the bottom right entry, namely $c_{2,2}$ *(which is formed by multiplying the second row of* $\mathbf{A}$ *by the second column of* $\mathbf{B}$*)*

$$\begin{pmatrix} a & b \\ \boxed{c} & \boxed{d} \end{pmatrix} \begin{pmatrix} \alpha & \boxed{\beta} \\ \gamma & \boxed{\delta} \end{pmatrix} = \begin{pmatrix} a\times\alpha + b\times\gamma & a\times\beta + b\times\delta \\ c\times\alpha + d\times\gamma & \boxed{c\times\beta + d\times\delta} \end{pmatrix}.$$

In general, to calculate the $c_{i,j}$ *entry we multiply the* i^{th} *row of the first matrix* $\mathbf{A}$ *by the* j^{th} *column of the second matrix* $\mathbf{B}$ *term by term.*

Example A.5 *Calculate the product* $\mathbf{AB}$ *of the matrices*

$$\mathbf{A} = \begin{pmatrix} 1 & -1 \\ 0 & 3 \end{pmatrix} \text{ and } \mathbf{B} = \begin{pmatrix} -2 & 1 \\ 4 & -2 \end{pmatrix}.$$

Using the method given in the previous example

$$\begin{aligned} \mathbf{AB} &= \begin{pmatrix} 1 & -1 \\ 0 & 3 \end{pmatrix} \begin{pmatrix} -2 & 1 \\ 4 & -2 \end{pmatrix} \\ &= \begin{pmatrix} 1\times(-2) + (-1)\times 4 & 1\times 1 + (-1)\times(-2) \\ 0\times(-2) + 3\times(4) & 0\times 1 + 3\times(-2) \end{pmatrix} \\ &= \begin{pmatrix} -6 & 3 \\ 12 & -6 \end{pmatrix}. \end{aligned}$$

(It is worth practising these calculations; try calculating the product $\mathbf{BA}$ *yourself by hand.).*

We now turn our attention to matrix multiplication in which the matrices are not necessarily square.

Example A.6 *Consider the matrices*

$$\mathbf{A} = \begin{pmatrix} 3 & 0 & -1 \\ -4 & 2 & 2 \end{pmatrix}, \qquad \mathbf{B} = \begin{pmatrix} -1 & 7 \\ 3 & 5 \\ -2 & 0 \end{pmatrix} \text{ and } \mathbf{C} = \begin{pmatrix} 2 & 0 \\ -1 & -3 \end{pmatrix}.$$

Calculate the quantities: $\mathbf{AB}$, $\mathbf{BA}$, $\mathbf{A}+\mathbf{B}^T$, $\mathbf{AC}$, $\mathbf{A}^T\mathbf{C}$, $3\mathbf{C}+2(\mathbf{AB})^T$, $(\mathbf{AB})\,\mathbf{C}$ *and finally* $\mathbf{A}\,(\mathbf{BC})$, *where possible (and if not, state the reason why the calculations cannot be performed).*

We shall start (for the first couple of examples) by providing full solutions and thereafter just give answers with a minimum of intermediate steps. So,

$$\begin{aligned}\mathbf{AB} &= \begin{pmatrix} 3 & 0 & -1 \\ -4 & 2 & 2 \end{pmatrix}\begin{pmatrix} -1 & 7 \\ 3 & 5 \\ -2 & 0 \end{pmatrix} \\ &= \begin{pmatrix} 3\times(-1)+0\times 3+(-1)\times(-2) & 3\times 7+0\times 5+(-1)\times(0) \\ (-4)\times(-1)+2\times 3+2\times(-2) & (-4)\times 7+2\times 5+2\times(0) \end{pmatrix} \\ &= \begin{pmatrix} -1 & 21 \\ 6 & -18 \end{pmatrix}.\end{aligned}$$

Similarly

$$\begin{aligned}\mathbf{BA} &= \begin{pmatrix} -1 & 7 \\ 3 & 5 \\ -2 & 0 \end{pmatrix}\begin{pmatrix} 3 & 0 & -1 \\ -4 & 2 & 2 \end{pmatrix} \\ &= \begin{pmatrix} -1\times 3+7\times(-4) & -1\times 0+7\times 2 & -1\times(-1)+7\times 2 \\ 3\times 3+5\times(-4) & 3\times 0+5\times 2 & 3\times(-1)+5\times 2 \\ -2\times 3+0\times(-4) & -2\times 0+0\times 2 & -2\times(-1)+0\times 2 \end{pmatrix} \\ &= \begin{pmatrix} -31 & 14 & 15 \\ -11 & 10 & 7 \\ -6 & 0 & 2 \end{pmatrix}\end{aligned}$$

$$\begin{aligned}\mathbf{A}+\mathbf{B}^T &= \begin{pmatrix} 3 & 0 & -1 \\ -4 & 2 & 2 \end{pmatrix}+\begin{pmatrix} -1 & 3 & -2 \\ 7 & 5 & 0 \end{pmatrix} \\ &= \begin{pmatrix} 2 & 3 & -3 \\ 3 & 7 & 2 \end{pmatrix}.\end{aligned}$$

It is not possible to pre-multiply matrix $\mathbf{A}$ *by* $\mathbf{C}$ *since* $\mathbf{A}$ *is two-by-three and* $\mathbf{C}$ *is two-by-two, so the inner dimensions do not agree (that is, the second dimension of the first matrix and the first dimension of the second matrix are different). For the next example* $\mathbf{A}^T\mathbf{C}$ *we observe that* $\mathbf{A}^T$ *is three-by-two so that the inner dimensions agree and hence the calculation is possible. We obtain*

$$\mathbf{A}^T\mathbf{C} = \begin{pmatrix} 3 & -4 \\ 0 & 2 \\ -1 & 2 \end{pmatrix}\begin{pmatrix} 2 & 0 \\ -1 & -3 \end{pmatrix} = \begin{pmatrix} 10 & 12 \\ -2 & -6 \\ -4 & -6 \end{pmatrix}.$$

The next example requires scalar multiplication and the use of the first of the results in this example; we have

$$\begin{aligned} 3\mathbf{C} + 2\left(\mathbf{AB}\right)^T &= 3\begin{pmatrix} 2 & 0 \\ -1 & -3 \end{pmatrix} + 2\begin{pmatrix} -1 & 6 \\ 21 & -18 \end{pmatrix}, \\ &= \begin{pmatrix} 6 & 0 \\ -3 & -9 \end{pmatrix} + \begin{pmatrix} -2 & 12 \\ 42 & -36 \end{pmatrix}, \\ &= \begin{pmatrix} 4 & 12 \\ 39 & -45 \end{pmatrix}. \end{aligned}$$

The final two calculations serve to demonstrate that matrix multiplication is associative, that is for three matrices $\mathbf{A}, \mathbf{B}$ *and* $\mathbf{C}$, $\mathbf{A}(\mathbf{BC}) = (\mathbf{AB})\mathbf{C}$. *(Notice that this does not constitute a formal proof.)*

$$\left(\mathbf{AB}\right)\mathbf{C} = \begin{pmatrix} -1 & 21 \\ 6 & 18 \end{pmatrix}\begin{pmatrix} 2 & 0 \\ -1 & -3 \end{pmatrix} = \begin{pmatrix} -23 & -63 \\ 26 & 54 \end{pmatrix}.$$

And now the final example

$$\begin{aligned} \mathbf{A}\left(\mathbf{BC}\right) &= \begin{pmatrix} 3 & 0 & -1 \\ -4 & 2 & 2 \end{pmatrix}\left\{\begin{pmatrix} -1 & 7 \\ 3 & 5 \\ -2 & 0 \end{pmatrix}\begin{pmatrix} 2 & 0 \\ -1 & -3 \end{pmatrix}\right\} \\ &= \begin{pmatrix} 3 & 0 & -1 \\ -4 & 2 & 2 \end{pmatrix}\begin{pmatrix} -9 & -21 \\ 1 & -15 \\ -4 & 0 \end{pmatrix} = \begin{pmatrix} -23 & -63 \\ 26 & 54 \end{pmatrix}. \end{aligned}$$

In this example we see that the result of the multiplication $\mathbf{BA}$ *is a three-by-three matrix, further emphasising the fact that* $\mathbf{AB}$ *is not necessarily equal to* $\mathbf{BA}$. *In some cases it may not even be possible to perform this second multiplication. This example serves to demonstrate that matrix multiplication is associative (that is* $\mathbf{A}(\mathbf{BC}) \equiv (\mathbf{AB})\mathbf{C}$*), but it is not, in general, commutative. It is also a simple matter to show that matrix multiplication is distributive, that is* $\mathbf{A}(\mathbf{B} + \mathbf{C}) = \mathbf{AB} + \mathbf{AC}$ *for any three matrices* $\mathbf{A}, \mathbf{B}$ *and* $\mathbf{C}$ *for which the matrix multiplications are permitted.*

A.1 Special Matrices

There are two special matrices that we will make use of often within the text and we introduce them here. The first is the zero matrix, which we will denote by $\mathbf{0}$.

This is simply a matrix whose elements are all equal to zero[3]. Not surprisingly, the zero matrix has no effect when it is added to another matrix (of the same size). So $\mathbf{A} + \mathbf{0} = \mathbf{A} = \mathbf{0} + \mathbf{A}$. We will make use of this matrix to initialise matrices in preparation to assigning answers to a matrix.

The second important matrix we will have call to use often with the text is the *identity* (or *unit*) matrix, denoted by $\mathbf{I}$. The identity matrix $\mathbf{I}$ is a n-by-n matrix whose elements consist of 1s (ones) along the main diagonal and are zero everywhere else. For example, the three-by-three identity matrix is given by

$$\mathbf{I} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

Multiplying a square n-by-n matrix $\mathbf{A}$ by $\mathbf{I}$ has no effect:

$$\mathbf{AI} = \mathbf{A} = \mathbf{IA}.$$

A.2 Inverses of Matrices

The inverse of a matrix, written as $\mathbf{A}^{-1}$, is defined as the matrix which when pre- and post-multiplied by the matrix $\mathbf{A}$ produces the identity matrix $\mathbf{I}$:

$$\mathbf{A}^{-1}\mathbf{A} = \mathbf{A}\mathbf{A}^{-1} = \mathbf{I}.$$

Only square matrices can have an inverse but it is only a subset of all square matrices for which the inverse exists. The existence of the inverse of a matrix (that is, whether the matrix is invertible or not) is intimately linked with the determinant of the matrix. We introduce this, and many other properties of matrices, in Chapter 6.

The utility of the inverse of a matrix is best seen when solving systems of equations. An example will serve by way of illustration.

Example A.7 *Consider the system of simultaneous equations:*

$$x_1 + x_2 = 3, \tag{A.1a}$$

$$x_1 + 2x_2 = 5. \tag{A.1b}$$

[3] Of course, we could include the dimensions of the matrix by writing $\mathbf{0}_{nm}$ to denote that it is an n-by-m matrix. We will adopt the convention in the text that when we refer to the zero matrix we are taking a matrix of the appropriate size required for the operation.

These equations can be solved using conventional means. To do this we first subtract (A.1a) from (A.1b) to give

$$x_1 + 2x_2 - (x_1 + x_2) = 5 - 3$$

or

$$x_2 = 2,$$

and now substituting back into either equation (let us use (A.1a)) we have

$$x_1 + 2 = 3$$

which gives

$$x_1 = 1.$$

We can just as easily write the system (A.1) as a matrix equation

$$\begin{pmatrix} 1 & 1 \\ 1 & 2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 3 \\ 5 \end{pmatrix}$$

or as

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

Try multiplying out the matrix equation to check you get (A.1a) and (A.1b). Elementary linear algebra shows that the solution is given by $\mathbf{A}^{-1}\mathbf{b}$*, which can be written in MATLAB as* `inv(A)*b` *or* `A\b`*. The operator \ determines the effect of multiplying by the inverse of the first argument on the second, without ever constructing the inverse. The code for this example would be:*

```
>> A = [1 1; 1 2]; % Initialise the matrix A
>> b = [3; 5];     % Initialise the vector b
>> x = inv(A)*b    % Determine the solution vector x
```

Before we proceed we note in the previous three lines of MATLAB code everything after the percent sign % is taken by MATLAB to be a comment. Comments are a useful way of making your MATLAB code readable by both you and others.

We can check the answer from our matrix computation by typing `A*x`*, which should be equal to* `b`*.*

Example A.8 *Determine the vector* $\mathbf{x}$ *which satisfies the equation* $\mathbf{Ax} = \mathbf{b}$ *where*

$$\mathbf{A} = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 4 & 3 & 2 & 1 \\ 1 & 0 & -1 & 0 \\ -1 & 1 & -1 & 1 \end{pmatrix} \text{ and } \mathbf{b} = \begin{pmatrix} 5 \\ 10 \\ 15 \\ 20 \end{pmatrix}.$$

We enter the matrix $\mathbf{A}$ *and vector* $\mathbf{b}$ *directly using*

```
>> A = [1 2 3 4; ...
   4 3 2 1; ...
   1 0 -1 0; ...
   -1 1 -1 1];
>> b = [5; 10; 15; 20];
```

(Note we have used the three dots ... (or ellipsis) to indicate to MATLAB that the input line continues. It is good practise to have a space before the dots at the end of the line.) These results can then be used to form $\mathbf{x}$*:*

```
>> inv(A)*b

ans =

    3.2500
    4.5000
  -11.7500
    7.0000
```

This solution was obtained by considering the equation $\mathbf{Ax} = \mathbf{b}$ *and pre-multiplying each side by the inverse of the matrix* $\mathbf{A}$ *(we have deliberately chosen* $\mathbf{A}$ *so that its inverse exists). This gives* $\mathbf{A}^{-1}\mathbf{Ax} = \mathbf{A}^{-1}\mathbf{b}$ *but we recall from the definition of the inverse that* $\mathbf{A}^{-1}\mathbf{A} = \mathbf{I}$ *and that* $\mathbf{Ix} = \mathbf{x}$*. Hence we have the solution* $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$*.*

B
Glossary of Useful Terms

This appendix is provided purely as a guide. MATLAB has a very informative help feature `help command` which is supplemented with several other features `lookfor maths`. You can also access the help files on the web `helpdesk`.

This appendix is broken down into:

- arithmetic and logical operators
- symbols
- plotting commands
- general MATLAB commands

B.1 Arithmetic and Logical Operators

`+, -` Used to add or subtract variables of the same size together, whether they are matrices, vectors or scalars.

```
A = [1 2; 3 4];
B = ones(2);
A + B
A - B
5 + 0.5
7 - 4
```

It is also worth noting that these operations will add (or subtract) scalar quantities from matrices. For instance:

```
A = ones(3);
B = A + 2;
C = 3 - A;
```

This produces a three-by-three matrix full of threes in `B` and a three-by-three matrix full of twos in `C`. MATLAB will complain if these operations are not viable.

`*, /` Used to multiply or divide variables as long as the operation is mathematically viable:

```
4 * 3.2
4 / 2.3
A = [1 2; 3 4];
B = ones(2);
A * B
A / B
```

The last two operations give $\mathbf{AB}$ and $\mathbf{AB}^{-1}$. Notice that the multiplication operation is only viable if the inner dimensions agree: the number of columns of the first matrix must equal the number of rows of the second.

It can also multiply (or divide) matrices by scalars:

```
A = ones(3);
B = A/3;
C = A*4;
```

These commands give `B` as a three-by-three matrix full of 1/3's and `C` as a three-by-three matrix full of fours.

`.*` This binary operator allows one to perform multiplication calculations element by element on arrays of the same size. If we consider the vectors $\mathbf{x} = (x_1, x_2, x_3, x_4)$ and $\mathbf{y} = (y_1, y_2, y_3, y_4)$. Then the calculation $\mathbf{x}$`.*`$\mathbf{y}$ gives $(x_1y_1, x_2y_2, x_3y_3, x_4y_4)$. Notice that both $\mathbf{x}$ and $\mathbf{y}$ were row vectors of length 4 and so is the answer. Let us consider the example:

```
A = [4 3; 2 1];
B = [1 2; 3 4];
C = A.*B;
```

This does the calculation element-wise and gives the result:

$$\mathtt{C} = \begin{pmatrix} 4\times 1 & 3\times 2 \\ 2\times 3 & 1\times 4 \end{pmatrix} = \begin{pmatrix} 4 & 6 \\ 6 & 4 \end{pmatrix}.$$

This command can also be used on values which are scalars, so for instance `A = ones(2); B = A.*2;` gives `B` as a matrix full of twos.

The most common use of this operator is again in the construction of functions.

```
x = 1:5;
f = x.*sin(x);
g = (3*x+4).*(x+2);
```

This gives us `x = [1 2 3 4 5]` and then: $f = x\sin x$ evaluated at those points, i.e. `[sin(1) 2*sin(2) 3*sin(3) 4*sin(4) 5*sin(5)]`; and $g = (3x+4)(x+2)$ at the points. Notice it is not necessary to use the operator `.*` when calculating `3*x` since 3 is a scalar.

`./` This binary operator allows one to perform division calculations element by element on arrays of the same size. If we consider the vectors $\mathbf{x} = (x_1, x_2, x_3, x_4)$ and $\mathbf{y} = (y_1, y_2, y_3, y_4)$, then the calculation $\mathbf{x}$`./`$\mathbf{y}$ gives $(x_1/y_1, x_2/y_2, x_3/y_3, x_4/y_4)$. Notice that both $\mathbf{x}$ and $\mathbf{y}$ were row vectors of length 4 and so is the answer. Let us consider the example:

```
A = [4 3; 2 1];
B = [1 2; 3 4];
C = A./B;
```

This does the calculation element-wise and gives the result:

$$\mathtt{C} = \begin{pmatrix} 4/1 & 3/2 \\ 2/3 & 1/4 \end{pmatrix} = \begin{pmatrix} 4 & \frac{3}{2} \\ \frac{2}{3} & \frac{1}{4} \end{pmatrix}.$$

This command can also be used on values which are scalars, so for instance `A = ones(2); B = A./2;` gives `B` as a matrix full of halves.

The most common use of this operator is in the construction of functions.

```
x = 1:5;
f = x./sin(x);
g = (3*x+4)./(x+2);
```

This gives us `x = [1 2 3 4 5]` and then: $f = x/\sin x$ evaluated at those points, i.e. `[1/sin(1) 2/sin(2) 3/sin(3) 4/sin(4) 5/sin(5)]`; and $g = (3x+4)/(x+2)$ at the points.

We can use this command where either of its arguments are scalars (in fact, it is necessary for this example).

```
x = [1 2 3 4 5];
y = 2./x
```

This gives `[2/1 2/2 2/3 2/4 2/5]`. This construction is useful when working out functions of the form $f(x) = 2/x$.

`.^` This binary operator allows one to perform exponentiation (raising to a power) calculations element by element on arrays of the same size. If we consider the vectors $\mathbf{x} = (x_1, x_2, x_3, x_4)$ and $\mathbf{y} = (y_1, y_2, y_3, y_4)$, then the calculation $\mathbf{x}$`.^`$\mathbf{y}$ gives $(x_1^{y_1}, x_2^{y_2}, x_3^{y_3}, x_4^{y_4})$. Notice that both $\mathbf{x}$ and $\mathbf{y}$ were row vectors of length 4 and so is the answer. Let us consider the example:

```
A = [4 3; 2 1];
B = [1 2; 3 4];
C = A.^B;
```

This does the calculation element-wise and gives the result:

$$\mathtt{C} = \begin{pmatrix} 4^1 & 3^2 \\ 2^3 & 1^4 \end{pmatrix} = \begin{pmatrix} 4 & 9 \\ 8 & 1 \end{pmatrix}.$$

This command can also be used on values which are scalars, so for instance `A = ones(2); B = A.^2;` gives `B` as a matrix full of ones (that is one squared).

The most common use of this operator is in the construction of functions.

```
x = 1:5;
f = x.^(x+1);
g = (3*x+4).^x;
```

This gives us `x = [1 2 3 4 5]` and then: $f = x^{x+1}$ evaluated at those points, i.e. `[1^2 2^3 3^4 4^5 5^6]`; and $g = (3x + 4)^x$ at the points.

As with the operator './' we can use this command when either of its arguments are scalars. For instance:

```
x = 1:5;
y = 2.^x;
z = x.^2;
```

This gives `y=[2^1 2^2 2^3 2^4 2^5]` and `z=[1^2 2^2 3^2 4^2 5^2]`.

\ This works out the effect of pre-multiplying by the inverse of the first argument on the second argument. So if we want to solve the set of linear equations represented in the matrix equation $\mathbf{Ax} = \mathbf{b}$, we need to construct $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$.

```
A = [1 2; 3 4];
b = [2; 3];
x = A\b;
```

This solves the set of equations:

$$x_1 + 2x_2 = 2$$
$$3x_1 + 4x_2 = 3.$$

by defining

$$\mathbf{A} = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \text{ and } \mathbf{b} = \begin{pmatrix} 2 \\ 3 \end{pmatrix},$$

with $\mathbf{x} = (x_1, x_2)^T$.

== Checks equality, rather than sets equal. This is usually exploited within logical statements with scalars:

```
if i==7
   disp(' i is seven ')
else
   disp(' i is not seven ')
end
```

This can be used in other forms: `x = 1:12; mod(x,3)==0`. This gives the output:

```
0     0     1     0     0     1     0     0     1     0     0     1
```

that is it is true provided the corresponding element of `x` is divisible by 3.

`~=` Checks not equal to. Again this is mainly used in logical statements with scalars:

```
if i~=7
   disp(' i is not seven ')
else
   disp(' i is equal to seven ')
end
```

This can be used inline `x = 1:12; mod(x,3)~=0`. This gives the output:

```
1     1     0     1     1     0     1     1     0     1     1     0
```

that is it is true provided the corresponding element of `x` is not divisible by 3. **Note**: great care is needed with this command since the simple order change to `y=~x` means set `y` equal to `not(x)`.

`>, >=` Checks greater than and greater than or equal to. Mainly used with scalars within logical statements:

```
if i>7
   disp(' i is greater than seven ')
end

if i>=7
   disp(' i is greater than or equal to seven ')
end
```

Can be used for an array `x = 1:12; x>7` which gives

```
0     0     0     0     0     0     0     1     1     1     1     1
```

whereas `x = 1:12; x>=7` gives

```
0     0     0     0     0     0     1     1     1     1     1     1
```

that is the second one is true for $x = 7$.

`<, <=` Checks less than and less than or equal to. Mainly used with scalars within logical statements:

```
if i<7
   disp(' i is less than seven ')
end

if i<=7
   disp(' i is less than or equal to seven ')
end
```

Can be used for an array `x = 1:12; x<7` gives

```
1     1     1     1     1     1     0     0     0     0     0     0
```

whereas `x = 1:12; x<=7`

```
1     1     1     1     1     1     1     0     0     0     0     0
```

that is the second one is true for $x = 7$.

`all` This returns a value of true if all of the arguments are true: `all(x.^2>0)` would be true (provided all the values of `x` are real); `x = 1:12; all(x>0)` would give true since all the elements of `x` are positive.

`and` Boolean operator for *and* can also use &. This is true only if both its arguments are true.

```
if and(x>7,x<9)
   disp(' x is between 7 and 9')
end
```

This can also be written as `x>7 & x<9` and can be applied to arrays, so that `x = 1:12; and(x>7,x<9)` gives:

```
0     0     0     0     0     0     0     1     0     0     0     0
```

`any` This returns a value of true if any of the arguments are true: `any(x<0)` would be true if any of the values in the vector `x` are strictly negative; `x = 1:12; any(x>10)` would be true since some elements of `x` are greater than 10.

`find` Provides a list of integers when the condition is true, for instance `x = 1:10; [i]=find(x.^2>26);` gives the locations `i=[6 7 8 9 10]`.

`not` Negates logical variables, can also use $\sim$. This can be used to turn true into false and vice versa, so that

```
if not(x>=7)
   disp(' x is not greater than or equal to 7')
end
```

This could also be written as $\sim$`(x>=7)`. This can be applied to arrays, so that `x = 1:12; not(x>=7)` gives:

```
1     1     1     1     1     1     0     0     0     0     0     0
```

`or` Boolean operator for *or* can also use |. This is true provided one of its arguments is true.

```
if or(x<7,x>9)
   disp(' x is less than 7 or greater than 9')
end
```

This can also be written as `x<7 | x>9` and can be applied to arrays, so that `x = 1:12; or(x<7,x>9)` gives:

```
1     1     1     1     1     1     0     0     0     1     1     1
```

`xor` Exclusive or. This gives true if one of the values is true and false if they are both false or both true.

```
x = [0 0 1 1];
y = [0 1 0 1];
xor(x,y)
```

gives `[0 1 1 0]`.

We note that this can also be done for `or` and `and`.

B.2 Symbols

`...` Used to link lines together (but cannot be used within a string):

```
x = 1:10;
f = x.^2+sqrt(x.^2+1) ...
     + cos(x)./x;
```

This gives the function $f = x^2 + \sqrt{x^2+1} + \cos x/x$ at the values from x equals 1 to 10.

`%` Means the rest of the line is a comment. It also has special meaning at the start of a code. It can be very useful for eliminating the execution of certain lines of a code.

```
% This code calculates the value
% of x times sin(x)
f = x.*sin(x);

% and the value of cos(x^2)

g = cos(x.^2); % where x can be a vector.
```

If this code was saved as `mycode.m` then typing `help mycode` would produce:

```
This code calculates the value
of x times sin(x)
```

; Used at the end of a phrase to suppress output. It can occur at the physical end of a line or at the end of a set of commands.

```
a = ones(2);
b = ones(2); c = ones(2);
d1 = ones(2); d2 = ones(2)
```

This code will set up the two-by-two matrices `a`, `b`, `c`, `d1` and `d2` full of ones. It will only report on the initiation of `d2` since this phrase is not concluded with a semicolon.

It can also be used to end lines within matrices (enclosed within pairs of square brackets)

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

This gives the matrix

$$\mathbf{A} = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}.$$

(Note that the semicolon is used in both its incarnations here.) We note that both uses of the semicolon correspond to the end of the phrase: it is merely that the square brackets are unbalanced in the latter case so MATLAB knows that it is going to read the next line of the matrix.

: Used to delimit values when setting up a vector and also to refer to entire rows or columns of matrices. When setting a row vector there are two syntaxes:

```
x1 = 1:12
x2 = 1:2:13
```

The first of these commands sets up an array from 1 to 12 in steps of unity (and is equivalent to `1:1:12`) whereas the second array runs from 1 to 13 in steps of 2, thus it yields `[1 3 5 7 9 11 13]`. The syntaxes are `a:b` (an array from `a` to a value not exceeding `b` in steps of unity) and `a:h:b` (an array from `a` to a value not exceeding `b` in steps of `h`) (see also `linspace`). Note that `h` can be negative `11:(-2):1` gives `[11 9 7 5 3 1]`. We note that the length of an array set up using `a:h:b` is not $(b-a)/h$ (presuming that this is an integer), but $(b-a)/h+1$. For example `0:0.1:1` has eleven points `[0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0]` rather than ten. This dimension can be determined using `length`.

The second use of this symbol allows reference to all viable values of a row or column.

```
A = [11 12 13; 21 22 23; 31 32 33];

A(1,:)      % First row of A
A(:,2)      % Second column of A
A(:,:)      % Whole of A.
A(:,1:2)    % First and second columns of A
A(1:2:3,:)  % First and third rows of A
A(1:2,1:2)  % Top left two-by-two corner of A
```

, Can be used to delimit sets of commands where feedback is required:

```
a = 1, b = 2
```

and to separate elements of matrices on a particular line

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

It is also used to separate arguments of functions

```
x = linspace(0,1,200);
```

' ' Used to surround strings and for passing arguments to various functions:

```
a = 'Do robots dream of electronic sheep?'
x = 1:12;
y = x.^2;
plot(x,y,'LineWidth',2)
```

see the comments on `plot` below.

' Gives the complex conjugate transpose of a matrix. If the n-by-m matrix `A` has elements $a_{i,j}$, then `B = A'` is m-by-n and has elements $b_{i,j} = a^*_{j,i}$ (where $i \in \{1, \cdots, m\}$ and $j \in \{1, \cdots, n\}$).

```
A = [1+2*i 2-i; 3 4+i];
B = A';
```

This gives

$$\mathbf{B} = \begin{pmatrix} 1-2i & 3 \\ 2+i & 4-i \end{pmatrix}.$$

`.'` Gives the transpose of a matrix. If the n-by-m matrix `A` has elements $a_{i,j}$, then `B = A.'` is m-by-n and has elements $b_{i,j} = a_{j,i}$ (where $i \in \{1, \cdots, m\}$ and $j \in \{1, \cdots, n\}$).

```
A = [1+2*i 2-i; 3 4+i];
B = A.';
```

This gives

$$\mathbf{B} = \begin{pmatrix} 1+2i & 3 \\ 2-i & 4+i \end{pmatrix}.$$

It is better to use this form rather than `A'` unless you are concerned with issues of complex matrices.

`(space)` Can be used to separate elements of matrices, as for the comma; but cannot be used to separate commands. Hence we can use:

```
A = [1 2 3; 4 5 6];
B = [ 1:3 ; 3 2 1];
```

The reason that `(space)` cannot be used to delimit commands is that it can occur naturally within a command, for instance `a = 2` which is used rather than `a=2` merely to improve the readability of the code.

`.` decimal point – This can be used firstly as the mathematical decimal point: `3.145` or `567.3245`. It is also used to punctuate file names into two parts: an identifier (descriptive of the contents of the code, or its function) and then the file's type (`.m`, `.mat`, `.dat` or `.fig`). For this reason you should only have one '.' in each number or filename.

`[ ]` used to enclose elements of a vector or matrix:

```
A = [1 2 3 5 6 7];
B = [1 2 3;
     3 4 5];
ms = ['Vector array' ' of strings'];
```

These brackets should balance.

( ) used to surround mathematical expressions and lists of arguments for functions:

```
x= 1:12
y = 1./(x.^2+1);
z = x.*sin(y);
```

Again these brackets should balance.

B.3 Plotting Commands

Before we start this section we need to discuss the idea of a handle. This is essentially a variable which allows us to access properties of an object.

```
>> x = 1:12; y = x.^2;
>> h = plot(x,y)
>> get(h)
    Color = [0 0 1]
    EraseMode = normal
    LineStyle = -
    LineWidth = [0.5]
    Marker = none
    MarkerSize = [6]
    MarkerEdgeColor = auto
    MarkerFaceColor = none
    XData = [ (1 by 12) double array]
    YData = [ (1 by 12) double array]
    ZData = []

    ButtonDownFcn =
    Children = []
    Clipping = on
    CreateFcn =
    DeleteFcn =
    BusyAction = queue
    HandleVisibility = on
    Interruptible = on
    Parent = [3.00012]
```

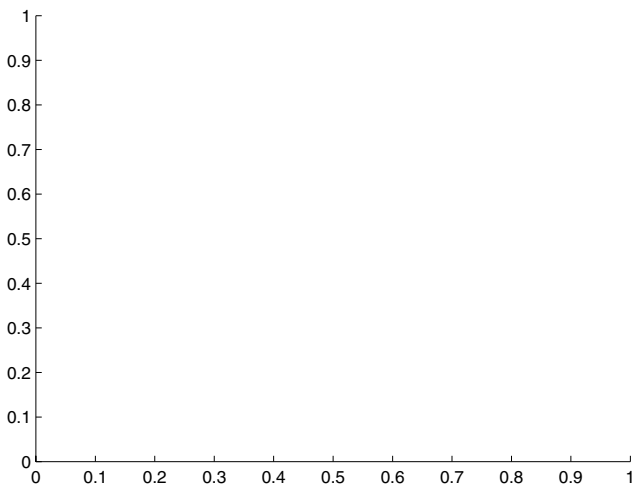
```
Selected = off
SelectionHighlight = on
Tag =
Type = line
UserData = []
Visible = on
```

The variable `h` allows us to access all of these commands, which can be changed using the command `set`, for instance `set(h,'MarkerSize',10)`.

`axes` Used to initiate a set of axes

```
figure(1)
h = axes
figure(2)
h = axes('position',[0.2 0.2 0.6 0.6])
```

The first command yields a full window set of axes with default ranges. The second command gives a reduced set of axes:

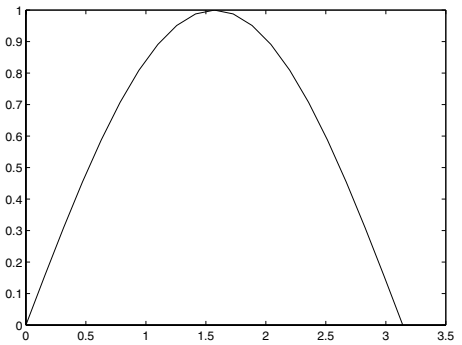


in the centre of the window of dimension 0.4 times the window width and height. The attributes of the handle can then be used by `get` and `set`.

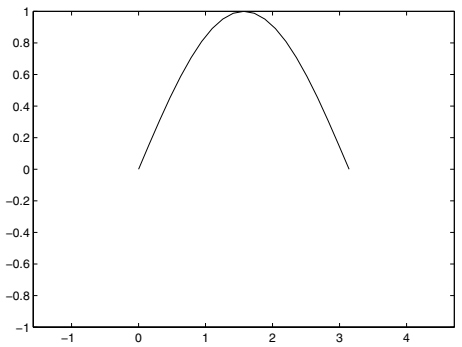
`axis` In plots used to set the range of the plot, the argument is a row vector of length 4 (for two-dimensional plots) or of length 6 (for three-dimensional plots).

```
x = 0:pi/20:pi; y = sin(x);
plot(x,y)
axis([-pi/2 pi/2 -1 1])
```

The initial ranges for the axes are given by `[min(x) max(x) min(y) max(y)]` which gives:



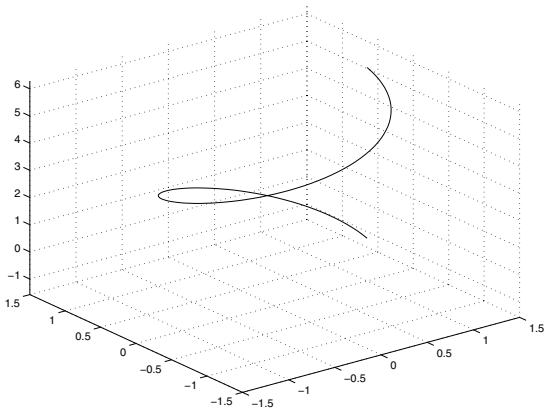
whereas typing the `axis` command above gives



For three dimensions a similar structure applies with an extra pair of variables representing the minimum and maximum of the third variable.

```
t = linspace(0,2*pi);
x = cos(t);
y = sin(t);
z = t;
plot3(x,y,z)
grid on
axis([-1.5 1.5 -1.5 1.5 -pi/2 2*pi])
```

This gives the plot:



We have used `grid on` to add the dotted lines.

The command can also be used as:

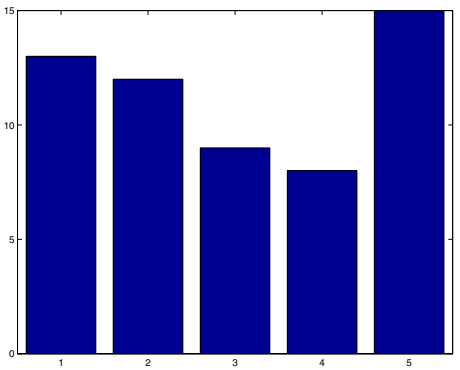
```
axis off    % Removes the axis from the current figure

axis equal  % Sets the axis scaling to be equal.

axis square % Sets the axis to be square.

axis tight  % Uses the max and min of the
            % data for axis limits
```

for more variants see `help axis`.

`bar` Produces a bar chart of the data.

```
x = [1 2 3 4 5];
y = [13 12 9 8 15];
bar(x,y)
```

This gives:

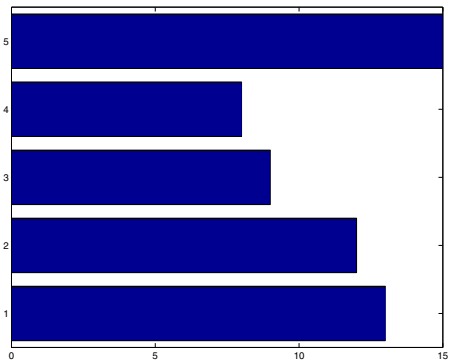


If only one argument is supplied it uses `x = 1:m` where `m` is the number of values of `y`.

`barh` Produces a horizontal bar chart of the data.

```
x = [1 2 3 4 5];
y = [13 12 9 8 15];
barh(x,y)
```

This gives:



If only one argument is supplied it uses `x = 1:m` where `m` is the number of values of `y`.

`clf` This clears the current figure, in fact it removes all children with visible handles. It removes any axes from the current figure. It is useful to use this before we start plotting.

`close` This is used to close figures; `close all` closes all figures. There are basically three syntaxes:

```
close       % Closes the current figure
close(4)    % closes figure number 4
close all   % closes all figures.
```

This can be used for other windows by using their handle: `close(h)` closes the window with handle `h`.

`figure` This brings the requested figure to the fore, or creates it if it doesn't exist.

```
figure      % Creates a figure
figure(3)   % Ensures that Figure No. 3 is the current
            % figure and is at the fore.
figure(h)   % Ensures that the figure with the handle
            % h is the current figure
```

`gca` Returns the handle to the current axis; this allows various properties to be displayed (`get`) and modified (`set`).

There are many variables involved:

```
>> x = 1:12; y=x.^2;
>> plot(x,y)
>> h = gca
>> get(h)
        AmbientLightColor = [1 1 1]
        Box = on
        CameraPosition = [6 75 17.3205]
        CameraPositionMode = auto
        CameraTarget = [6 75 0]
        CameraTargetMode = auto
        CameraUpVector = [0 1 0]
        CameraUpVectorMode = auto
        CameraViewAngle = [6.60861]
```

```
CameraViewAngleMode = auto
CLim = [0 1]
CLimMode = auto
Color = [1 1 1]
CurrentPoint = [ (2 by 3) double array]
ColorOrder = [ (7 by 3) double array]
DataAspectRatio = [6 75 1]
DataAspectRatioMode = auto
DrawMode = normal
FontAngle = normal
FontName = Helvetica
FontSize = [10]
FontUnits = points
FontWeight = normal
GridLineStyle = :
Layer = bottom
LineStyleOrder = -
LineWidth = [0.5]
NextPlot = replace
PlotBoxAspectRatio = [1 1 1]
PlotBoxAspectRatioMode = auto
Projection = orthographic
Position = [0.13 0.11 0.775 0.815]
TickLength = [0.01 0.025]
TickDir = in
TickDirMode = auto
Title = [287.001]
Units = normalized
View = [0 90]
XColor = [0 0 0]
XDir = normal
XGrid = off
XLabel = [288]
XAxisLocation = bottom
XLim = [0 12]
XLimMode = auto
XScale = linear
XTick = [ (1 by 7) double array]
XTickLabel =
        0
        2
        4
        6
        8
        10
        12
XTickLabelMode = auto
XTickMode = auto
YColor = [0 0 0]
YDir = normal
YGrid = off
YLabel = [289]
YAxisLocation = left
```

```
YLim = [0 150]
YLimMode = auto
YScale = linear
YTick = [0 50 100 150]
YTickLabel =
        0
        50
        100
        150
YTickLabelMode = auto
YTickMode = auto
ZColor = [0 0 0]
ZDir = normal
ZGrid = off
ZLabel = [290]
ZLim = [-1 1]
ZLimMode = auto
ZScale = linear
ZTick = [-1 0 1]
ZTickLabel =
ZTickLabelMode = auto
ZTickMode = auto

ButtonDownFcn =
Children = [285]
Clipping = on
CreateFcn =
DeleteFcn =
BusyAction = queue
HandleVisibility = on
HitTest = on
Interruptible = on
Parent = [1]
Selected = off
SelectionHighlight = on
Tag =
Type = axes
UIContextMenu = []
UserData = []
Visible = on
```

`gcf` Returns the handle of the current figure. This allows various properties to be displayed (`get`) and modified (`set`).

```
x = 1:12; y = x.^2;
plot(x,y)
h = gcf
get(h)
```

returns the list printed on page 348.

`get` Extracts a particular attribute from a list, retrieved for example by `gca` or `gcf`.

```
>> x = 1:12; y = x.^2;
>> plot(x,y)
>> h = gcf
>> a = gca
>> get(h,'Color')

ans =

    0.8000    0.8000    0.8000

>> get(a,'YTick')

ans =

     0    50   100   150
```

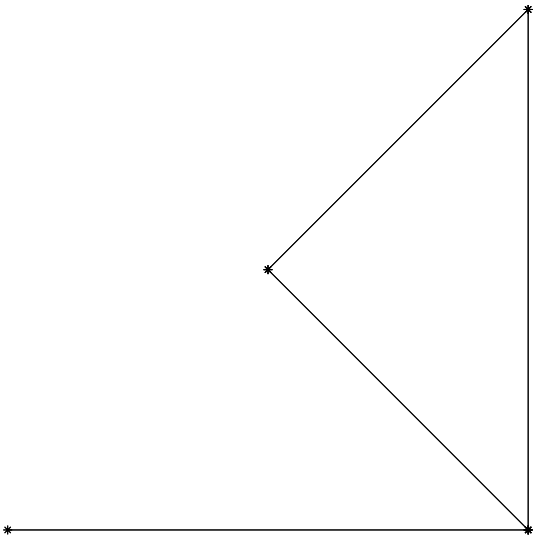
The colour variable is returned as a triple (a one-by-three vector giving the RGB (Red-Green-Blue) value). The values associated with the axis and the figure can be changed using the `set` command.

`ginput` Returns coordinates of mouse clicks on the current figure in terms of axis units; very useful for obtaining initial estimates for roots (see also `zoom`).

```
x = 0:pi/20:4*pi;
y = sin(x);
plot(x,y)
[xp,yp]= ginput
```

Notice that control is passed over to the figure and the points are stored in the arrays `xp` and `yp` (until the return key is hit). The command can also be used as `n = 10; [xp,yp] = ginput(n)` to only get 10 points.

`gplot` Plot a graph using the vertices and the adjacency matrix. To plot the graph



we use the code:

```
xy = [0 0;1 0;
      1 1; 0.5 0.5];
A = [0 1 0 0;
     1 0 1 1;
     0 1 0 1;
     0 1 1 0];
gplot(A,xy,'-*')
axis equal
axis off
```

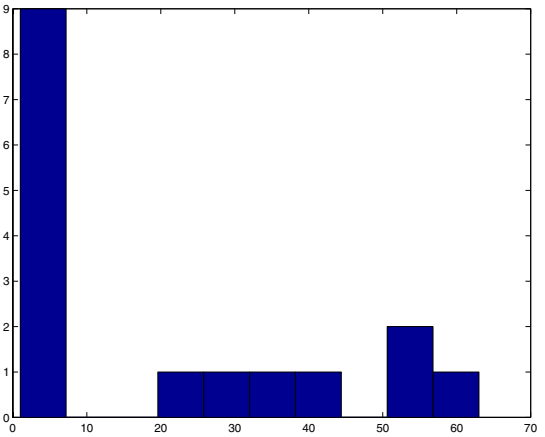
`grid` Add a grid to a plot (or turn it off).

```
grid on   % Turns the grid on

grid off  % Turns the grid off

grid      % Toggles the grid state
```

`hist` Produces a histogram of the data.

```
>> y = [1 2 3 43 32 54 33 2 4 53 5 63 21 1 5 2];
>> hist(y)
```

gives



We can use a second argument for `hist` to define the number of bins.

`hold` Stops overwriting of current figure.

```
hold on   % Turns the hold on

hold off  % Turns the hold off

hold      % Toggles the hold state
```

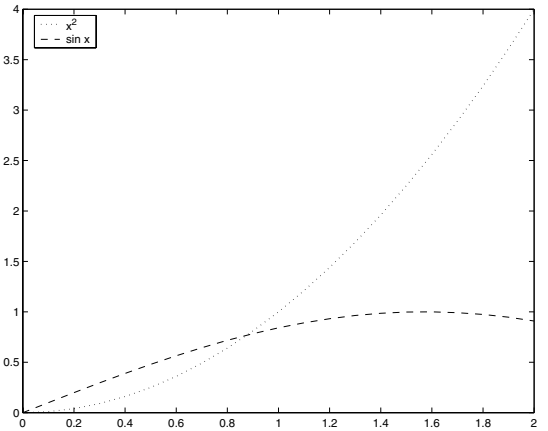
This is useful for putting multiple lines on a plot.

```
x = 0:pi/20:pi;
y = sin(x);
z = cos(2*x);
clf
plot(x,y)
hold on
plot(x,z)
hold off
```

`legend` Used to add a "legend" to a figure, to describe the lines.

```
x = 1:10;
y = x.^2;
z = sin(x);
plot(x,y,x,z)
legend('x^2','sin x', 0)
```

This gives



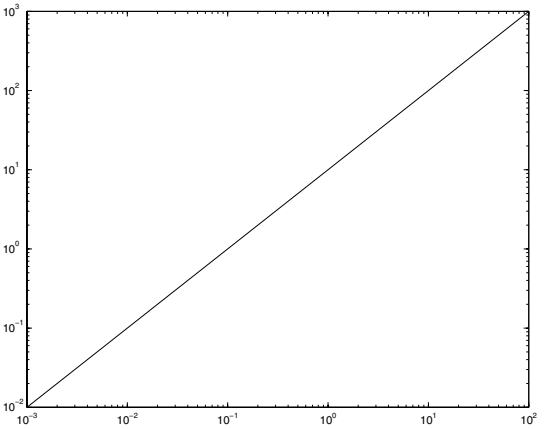
The arguments for `legend` consist of the labelling for the lines and a number which is chosen from:

0	=	Automatic "best" placement (least conflict with data)
1	=	Upper right hand corner (default)
2	=	Upper left hand corner
3	=	Lower left hand corner
4	=	Lower right hand corner
-1	=	To the right of the plot

`loglog` Produces a graph of natural log against natural log of the data.

```
x = [1e-3 1e-2 1e-1 1 1e1 1e2];
y = [10e-3 10e-2 10e-1 10 10e1 10e2];
loglog(x,y)
```

This gives:



see also `semilogx` and `semilogy`.

`plot` Used to set up figures and plotting of data. The simplest form would be `plot(x,y)`. This command is extremely powerful and has many variants (typing `help plot` helps tremendously).

```
x = -1:0.1:1;
y = x.^3;
plot(x,y)      % Produces a simple plot of y=x^3
               % using the default line colour

plot(x,y,'go') % Produces a simple plot of y=x^3
               % using green circles.
```

The arguments in `plot` either occur: in pairs in which they are pairs of coordinates for plotting or in triples in which case they are pairs of coordinates and the line style with which the data is to be plotted. You can also define variables associated with the plot in the statement:

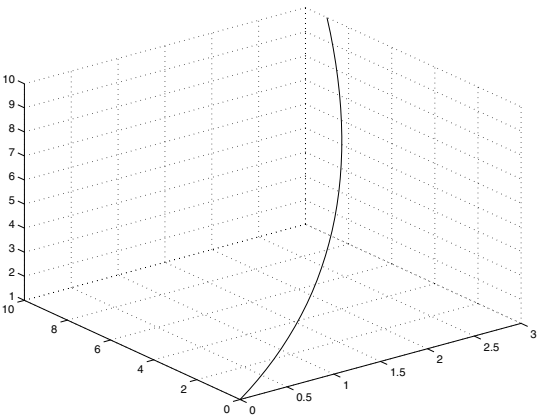
```
plot(x,y,'LineWidth',2)
```

plots the line but with a thicker line.

`plot3` This is similar to `plot` but gives a line in three dimensions.

```
x = 0.:0.1:3.0;
y = x.^2;
z = 3*x+1;
plot3(x,y,z)
grid on
```

This gives

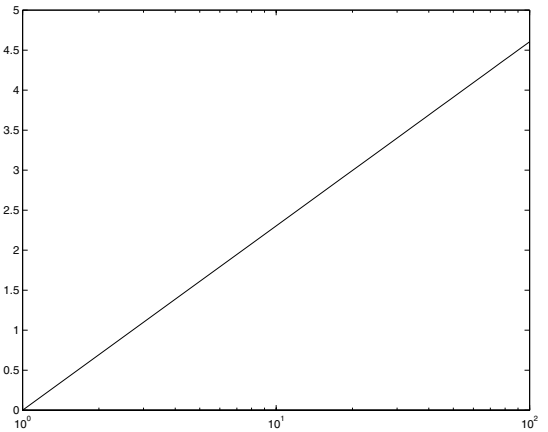


`print` Used to output the contents of figures to files, for Postscript use `print -dps2 output.ps`. For colour Postscript use `print -dpsc2 output.cps` or for JPEG format `printf -djpeg90 output.jpg`.

`semilogx` Produce a plot of y versus $\ln x$.

```
x = linspace(1,100,30);
semilogx(x,log(x))
```

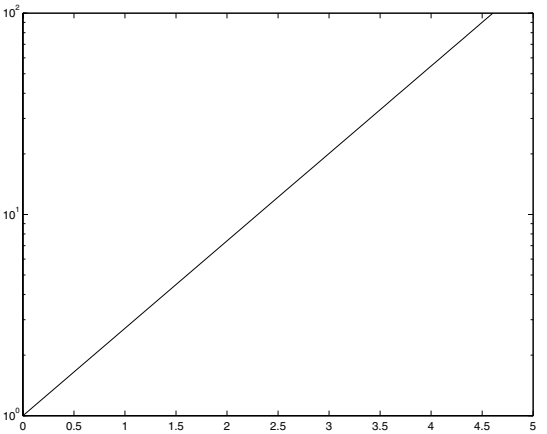
gives:



`semilogy` Produce a plot of $\ln y$ versus x.

```
x = linspace(1,100,30);
semilogy(log(x),x)
```

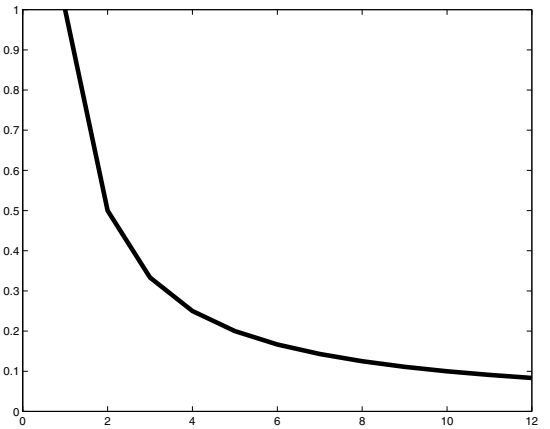
gives:



`set` Allows the definition of a particular attribute from a list, retrieved for example by `gca` and `gcf`, or directly from a `plot` command.

```
x = 1:12;
y = 1./x;
h = plot(x,y)
set(h,'LineWidth',4)
```

This gives



For a list of the attributes of an object (and their values) use `get(h)`, where `h` is defined directly using `gca` or `gcf`.

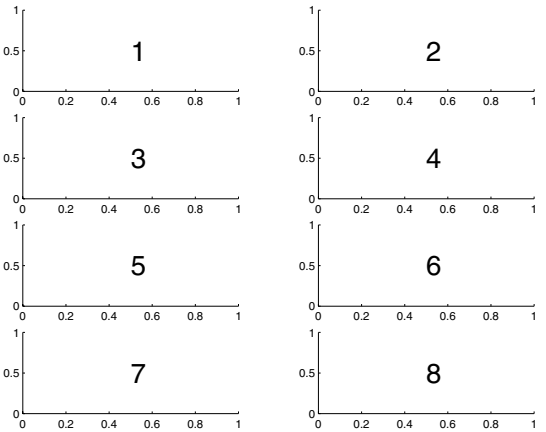
`subplot` Sets up sub-elements of a plot and points to which one is current. It takes three arguments:

```
subplot(4,2,3)  % Gives an array of figures 4-by-2
```

and sets the current axis to be the third figure, that is the left hand figure on the second row.

```
for j = 1:8
   subplot(4,2,j)
   text(0.5,0.5,int2str(j),'FontSize',24)
end
```

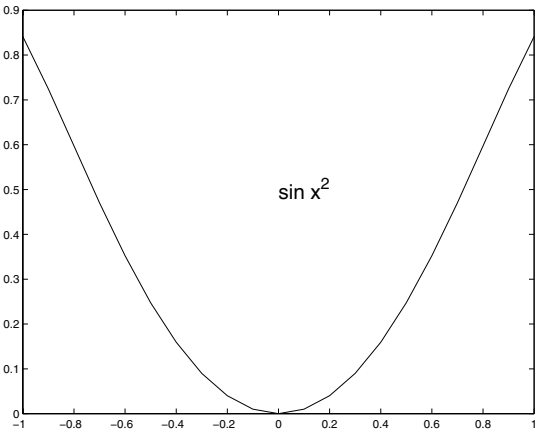
This gives eight subplots (which have been labelled with their corresponding numbers).



`text` Used to add useful labels to figures.

```
x = -1:0.1:1;
y = sin(x.^2);
plot(x,y)
h = text(0,0.5,'sin x^2')
set(h,'FontSize',18)
```

This gives:

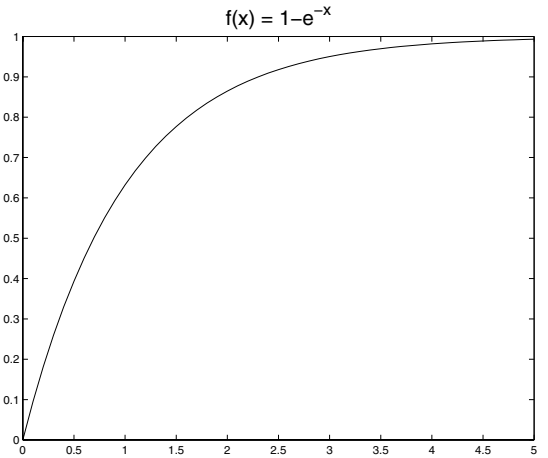


The syntax is `text(a,b,string)` where `(a,b)` is the coordinate in terms of the data and `string` is enclosed in single quotes. Notice here that `sin x^2` actually appears as `sin` x^2. These commands also recognise underscore for subscript and Greek letters in the form \omega, for instance.

`title` Used to set the title of a plot or subplot. This has quite simple syntax and attaches it to the current set of axes.

```
x = 0.0:0.1:5.0;
y = 1-exp(-x);
plot(x,y)
title('f(x) = 1-e^{-x}','FontSize',18)
```

which gives:

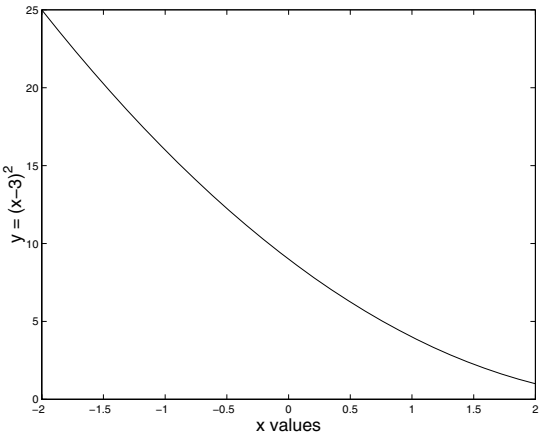


Notice we have used curly brackets to group the power of e for the title.

`xlabel, ylabel` Sets the text for the x and y axis.

```
x = linspace(-2,2,30);
y = (x - 3).^2;
plot(x,y)
hx = xlabel('x values')
hy = ylabel('y = (x-3)^2')
set(hx,'FontSize',16)
set(hy,'FontSize',16)
```

which gives:



`zoom` Permits zooming into figures: right click enlarge, left click to reduce. This has the syntax:

```
zoom on   % Turns the zoom on

zoom off  % Turns the zoom off

zoom      % Toggles the zoom state
```

B.4 General MATLAB Commands

`abs` Returns the absolute value of a real number or the modulus of a complex one (actually both are the same thing).

```
abs(-1)      % Gives 1
abs(1+i)     % Gives sqrt(2)

x = [1 -2 3+3i];
abs(x)
```

This also works for vectors and matrices so that the final example gives `[1.0000 2.0000 4.2426]` (where the last value is $3\sqrt{2}$).

`angle` Returns the argument of a complex number.

```
angle(sqrt(-1))     % This gives pi/2
angle(2)            % This gives 0
angle(-2)           % This gives pi
```

This can be used for an array of values and returns a vector of the same size full of the corresponding arguments.

`atan2` Gives the arctangent with values between $(-\pi, \pi]$. (see also `tan`,`atan`). Rather than calculating y/x and then taking the arctangent this function takes account of which quadrant the value is in.

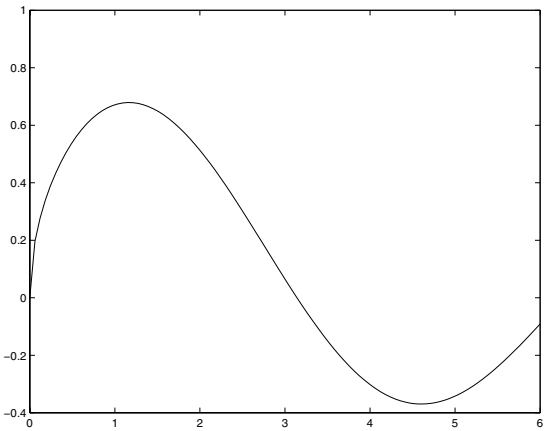
```
atan2(1,1)       % Gives pi/4
atan2(-1,-1)     % Gives -3pi/4
atan2(1,0)       % Gives pi/2
```

Note that the y value is given first. This command can be used to determine the argument of a complex number as `atan2(imag(z),real(z))` (which can be compared with `angle(z)`.

`besselj` Gives the solution $J_\nu(x)$ to Bessel's equation $x^2y'' + xy' + (x^2 - \nu^2)y = 0$.

```
x = linspace(0,6);
y = besselj(0.5,x);
plot(x,y)
```

This gives the graph:



The parameter ν is set to be $1/2$ here and in fact $J_{\frac{1}{2}}(x) = \sin x/\sqrt{x}$. There are other Bessel functions: `bessely(nu,x)`, `besseli(nu,x)` and `besselk(nu,x)`.

`break` Stop current level of execution and go back to the previous level, for instance exit a function.

```
function [sx] = takesqrt(x)
if x<0
   disp(' x is negative ')
   sx = NaN;
   break
end
sx = sqrt(x);
```

This routine finds the square root of positive quantities and `break`s if x is negative.

`case` Elements of a `switch` list, plausible values which the argument can take (see `switch` entry for example).

`ceil` Rounds up to the integer above, has the syntax `ceil(x)` where `x` can be a matrix, vector or scalar.

```
>> x = [0.3 0.9; 1.01 -2.3];
>> ceil(x)
ans =

     1     1
     2    -2
```

`clear` Used to reset objects; `clear variables` removes all variables.

```
clear all         % Clears variables, globals, fns etc
clear variables   % Clear all variables
clear global      % Clear global variables
clear             % Same as clear variable

clear x           % Clear local variable x
clear x*          % Clear local variables starting with x.
```

`cond` Gives the condition number of a matrix, that is the ratio of its largest and smallest eigenvalues. This reflects the ease with which the matrix can be inverted, amongst other things.

```
A = [100 0; 0 0.1];
cond(A)
```

This matrix has eigenvalues of 100 and 0.1 and `cond(A)` returns 1000, that is 100/0.1. In particular the Hilbert matrix is particularly badly conditioned (ill-conditioned), see `hilb`.

`conj` Gives the conjugate of a complex number or an array of them.

```
x = [1 1+i -2-i 4+3i];
conj(x)
```

`corrcoef` Gives the correlation coefficient between two sets of data.

```
>> x = [ 1 2 3 4 5 6];
>> y = [ 3 4 2 1 4 5];
>> corrcoef(x,y)
ans =

    1.0000    0.3268
    0.3268    1.0000
```

This means that `x` and `y` are totally correlated with themselves and that the correlation coefficient between the vectors is `0.3268`. That is a slight positive correlation. The correlation coefficient between two random variables X and Y is given by

$$r = \frac{\mathrm{cov}(X,Y)}{\sqrt{\mathrm{var}(X)\mathrm{var}(Y)}},$$

where the variances are defined in Equation (B.2) and the covariance in Equation (B.1).

`cos, acos` Cosine and arccosine. These functions need to be used with brackets `cos(x)` and `acos(x)` (without, it produces a bizarre result, for instance `cos pi` gives a one-by-two row vector with the elements cosine of the ASCII code for "p" followed by the cosine of the ASCII code for "i"). These functions can also be used for vectors and matrices.

```
x = 0:pi/20:pi;
y = cos(x)
z = acos(y)
```

`cosh` Hyperbolic cosine, equal to $(e^x + e^{-x})/2$.

`cov` Gives the covariance of two sets of data.

```
>> x = [ 1 2 3 4 5 6];
>> y = [ 3 4 2 1 4 5];
>> cov(x,y)
ans =

    3.5000    0.9000
    0.9000    2.1667
```

The top left and bottom right elements are the variances of `x` and `y` and the other elements are the covariances. The covariance is defined as

$$\sigma_{XY} = \frac{1}{N-1}\sum_{i=1}^{N}(x_i - \bar{x})(y_i - \bar{y}). \tag{B.1}$$

For the definition of the variance see Equation (B.2). This is normalised using $N-1$ (rather than N) since this gives the best unbiased estimate of the covariance.

`cputime` Gives the current value of the CPU time. This can be used to time how long parts of the code take:

```
t = cputime;
A = rand(100);
B = inv(A);
t2 = cputime - t;
disp(['Took ' num2str(t2) ' seconds'])
```

`dec2hex` Converts a decimal number to a hexadecimal number. The output will be a string; `a=23456; dec2hex(a)`. See also `hex2dec`.

`demo` Demonstrates the features and capabilities of MATLAB.

`det` This gives the determinant of a matrix. `A = ones(10); det(A)`.

`diag` Sets one of the diagonals of a matrix. The diagonals are referred to as: 0 the leading diagonal, which runs from top left to bottom right. The super-diagonals 1, 2, etc are above the leading diagonal and the sub-diagonals -1, -2, etc are below the leading diagonal. Note that the n^{th} diagonal is $|n|$ units longer than the leading one. If no diagonal is specified then the leading one is used.

```
x = 1:4;
A = diag(x);
B = diag(x,2) + diag(x,-2);
```

The matrix `A` is a four-by-four matrix with the elements of `x` (i.e. 1, 2, 3 and 4 down the leading diagonal), whereas `B` is the matrix

```
B =

     0     0     1     0     0     0
     0     0     0     2     0     0
     1     0     0     0     3     0
     0     2     0     0     0     4
     0     0     3     0     0     0
     0     0     0     4     0     0
```

The command can also be used in reverse, for instance `diag(A)` gives `[1 2 3 4]` and `diag(B,1)` gives `[0 0 0 0 0]`: here we have extracted diagonals.

As a further example we run the code

```
A = zeros(10);
for i = -9:9
    A = A+diag(ones(10-abs(i),1),i)*(i);
end
```

which gives

```
>> A

A =

     0     1     2     3     4     5     6     7     8     9
    -1     0     1     2     3     4     5     6     7     8
    -2    -1     0     1     2     3     4     5     6     7
```

```
-3    -2    -1     0     1     2     3     4     5     6
-4    -3    -2    -1     0     1     2     3     4     5
-5    -4    -3    -2    -1     0     1     2     3     4
-6    -5    -4    -3    -2    -1     0     1     2     3
-7    -6    -5    -4    -3    -2    -1     0     1     2
-8    -7    -6    -5    -4    -3    -2    -1     0     1
-9    -8    -7    -6    -5    -4    -3    -2    -1     0
```

`diary` Records the user commands and output. This is useful to see which commands have been used. One can specify the file in which the output is stored:

```
diary('list.diary')
x = 1:10;
y = x.^2;
diary off
```

`diff` Gives the difference between successive elements of a vector (is one unit shorter than the original vector): `x=(1:10).^2; diff(x)`, where $d(j) = x(j+1) - x(j)$.

`disp` Displays its argument, which is usually a string: `a=10; disp(['The value of a is ' int2str(a)])`.

`edit` Invokes the MATLAB editor, which is very useful since it allows us to see the current values of variables and provides a very user friendly environment for developing MATLAB programs.

`eig` Returns all the eigenvalues and eigenvectors of a matrix.

```
A = [1 2; -1 2];
[V, D] = eig(A)
```

`V` is a two-by-two matrix with the eigenvectors and columns and `D` is a diagonal matrix containing the eigenvalues on the leading diagonal.

`eigs` Returns certain eigenvalues and eigenvectors of a matrix, specified by certain criteria. `[V,D] = eigs(A,3,'SM')` gives the three eigenvalues of $\mathbf{A}$ with the smallest magnitude (and the corresponding eigenvectors). There are many options for this command, see `help eigs`.

`else` If the argument for the preceding `if` statement is false execute these statements.

`elseif` Same as `else` but imposing an alternative constraint.

`end` This command ends all of the loop structures and for each starting argument there must be a corresponding `end` (used with `for`, `if`, `switch` and `while`).

`eps` The distance from 1.0 to the next largest floating point number. So MATLAB cannot tell the difference between `1` and `1+eps/2`, for instance.

`error` Causes the code to stop execution; `error('Broken!')` usually used within a conditional statement.

`exist` Checks to see whether an object exists:

```
if ~exist('a')
   disp(['The variable a ' ...
        'does not exist'])
```

This can be used beyond variables: see the help lines for the command.

`exp` Evaluate the expression e^x, can be used with vectors and matrices.

`expm` Evaluate the expression $e^{\mathbf{A}}$, where $\mathbf{A}$ is a matrix. This differs from `exp(A)`, which evaluates e^x for all the elements of `A` rather than $e^{\mathbf{A}}$ which is given by:

$$e^{\mathbf{A}} \equiv \mathbf{I} + \sum_{j=1}^{\infty} \frac{\mathbf{A}^j}{j!}.$$

`eye` Sets up the identity matrix: `eye(n)` gives $\mathbf{I}_n$.

`factor` Gives the prime factors of an integer.

`factorial` This calculates the factorial of an integer n: `factorial(n)` gives $n!$.

`feval` Evaluates a function, either user-defined or intrinsic. `feval('sin',pi)`.

`fix` Rounds to the nearest integer (closest to zero), also works for matrices.

`fliplr` Flips an object left to right. This has no effect on column vectors.

```
x = 1:6;
y = fliplr(x)
```

sets `y` to be `[6 5 4 3 2 1]`. This also works with matrices.

`flipud` Flips an object upside down. This has no effect on row vectors.

```
x = transpose(1:6);
y = flipud(x)
```

sets y to be `[6; 5; 4; 3; 2; 1]`. This also works with matrices.

`floor` Rounds down to the integer below, also works for matrices.

`fmins` This uses the Nelder–Mead simplex (direct search) method to find the minimum of a function. For instance to find the minimum of the function $f(x_1, x_2) = (x_1 + 2x_2 - 1)^2/(x_2^2 + 1)$ we use

```
function [f]=func(x)
f = (x(1)+2*x(2)-1)^2/(x(2)^2+1);
```

and the command `[x]=fmins('func',[0 0])`.

`for` Defines the start of a loop which runs over a list of objects.

```
N = 10
for j = 2:10
    disp(j)
end
```

displays the numbers from 2 to 10.

`format` Used to specify how MATLAB displays variables. The options can be retrieved using `help format`.

`full` Tells the programme to treat the matrix as `full`, rather than `sparse`.

```
A = [1 0 2; 0 -2 0; -1 0 0];
B = sparse(A);
C = full(B);
```

```
>> A

A =

     1     0     2
     0    -2     0
    -1     0     0
```

```
>> B

B =

   (1,1)        1
   (3,1)       -1
   (2,2)       -2
   (1,3)        2
```

and `C` is back to `A` again.

`function` Occurs at the start of a function definition;
`function [v1,v2]=testfn(in1,in2,in3)`.

`fzero` Determines one zero of the function passed as the first argument to `fzero`. This function takes many different arguments and these can be displayed using `help fzero`.

```
f = inline('sin(3*x)');
x = fzero(f,2);
```

finds a zero of the function $f(x) = \sin 3x$ near x equals two.

```
Zero found in the interval: [1.8869, 2.1131].
>> x

x =

    2.0944
```

`gcd` Gives the greatest common divisor of two integers. This is unity if they are coprime: used as `gcd(x,y)`.

`global` This enables variables to be accessed from other places in the code without being passed directly as an argument. There needs to be a `global` statement in the context in which the variable is defined and also one where it is used.

`help` Gives help on MATLAB commands and can be used to expand the material given in this glossary.

`helpbrowser` Launches a web browser help facility (MATLAB 6).

`helpdesk` Provides access to the web-based help facility.

`hex2dec` Converts a hexadecimal number to a decimal: the input needs to be a string; `a='FF0'; hex2dec(a)`. See also `dec2hex`.

`hilb(n)` This sets up the n-by-n Hilbert matrix with entries $(\mathbf{A})_{i,j} = 1/(i+j)$.

`i,j` Initially set to be the square root of minus 1. These can be used to set up complex numbers:

```
a = 3 + 2*i;
b = -4 + j;
```

(Note that once either of these variables has been used in another context they will not necessarily be equal to $\sqrt{-1}$.)

`if` Start of a conditional block: `if` the statement is true then execute its contents.

```
if x>1
   disp('x is greater than 1')
end
```

This statement uses the logical structures described in section B.1.

`imag` Gives the imaginary part of a complex number; `imag(1+i)` gives 1.

`Inf` This represents answers which are infinite, for instance 1/0.

`inline` Used to define functions which will be evaluated `inline`, see the help function; `g = inline('t^2')` gives $g = t^2$ and then it can be used in `feval` as `feval(g,5)`.

`input` Used for user entry of data `a=input('Enter a: ');`; can also be used to enter strings `name = input('Enter name ',s);`.

`int2str` This takes an integer and returns a string: `int2str(10)` gives '10'.

`inv` Works out the inverse of a matrix (if it exists).

```
a = [1 3; 2 -1];
b = inv(a);
a * b
```

This gives the two-by-two identity (which could be constructed using the command `eye(2)`).

`isempty` Checks whether a variable *is empty*.

```
if isempty(x)
   disp('x is empty')
end
```

This means that the array has either zero rows or zero columns but still exists; to check the existence of a variable use the `exist` command.

`isreal` Checks whether a variable *is real*.

```
isreal(exp(0))
isreal(exp(i))
```

Care is needed since this command checks to see if the imaginary part of the complex number is exactly zero and does not allow for computation errors: for instance $e^{i\pi} = -1$, but the command `isreal(exp(i*pi))` suggests that the quantity is complex.

`isprime` Checks whether a variable *is prime*: `isprime(24)` gives false (that is zero) whereas `isprime(3571)` gives true (that is one).

`length` Gives the length of a vector or alternatively the larger dimension of a matrix:

```
a = [1 2 3; 4 5 6];
b = 1:16
length(a)
length(b)
```

gives the values 3 and 16 respectively.

`linspace` Sets up a grid of one hundred points from the first argument to the second. If there are three arguments use this as the number of points in the grid.

```
x = linspace(0,1,6);
z = linspace(1,10);
```

This gives `x = [0 0.2 0.4 0.6 0.8 1]` and `z` being the array running from 1 to 10 in steps of $(10-1)/99$; since the endpoints are included the step length is not $(10-1)/100$ as you might initially expect.

`load` Reads in data, either directly into variable `load data` (which loads data.mat) or `load 'data.dat'` which returns a matrix `data`.

`log` Natural logarithms.

```
x = [1 exp(1) exp(2)]
y = log(x)
```

This would be written mathematically as $y = \ln x$.

`log10` Logarithm to base ten.

```
x = [1 10 10^2]
y = log10(x)
```

This would be written mathematically as $y = \log_{10} x$. We note that $\log_{10} x = \ln x / \ln 10$.

`lookfor` Allows one to search help files for any command which has a string in its specification `lookfor bessel`.

`lower` Converts the characters in a string to lower case:

```
name = 'Bob Roberts';
lower(name)
```

This gives `bob roberts` (the opposite command is `upper`).

`lu` This produces the LU decomposition of a matrix and can provide information concerning pivoting.

```
A = [1 2 3; -1 3 2; -1 0 1];
[L,U] = lu(A)
```

Gives

```
L =

    1.0000         0         0
   -1.0000    1.0000         0
   -1.0000    0.4000    1.0000
```

```
U =

     1     2     3
     0     5     5
     0     0     2
```

To obtain the pivoting information we would have used `[L,U,P] = lu(A)`. In this case `P` is the three-by-three identity.

`magic` Returns a magic square:

```
>> magic(4)

ans =

    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1
```

Notice that not only do the rows, columns and diagonals add up to 34, but so do the four numbers in each corner, the numbers in each two-by-two block in the corner and the central two-by-two block.

`max` This returns the maximum of a vector: if a matrix is supplied it returns a vector providing the maxima along the rows.

```
x = 0:pi/4:2*pi
max(sin(x))
```

gives 1, and

```
A = [1 2 3; 4 5 6];
max(A)
max(transpose(A))
```

gives `[4 5 6]` (that is the maxima of the columns) and `[3 6]` (the maxima of the rows), respectively. Instead of using the transpose we can use the syntax `max(A,[],2)`, which determines the maximum along the second dimension. To find the maximum of a two-dimensional array we use `max(max(A))`.

`mean` Calculates the mean of a set of data, $1/n \sum_{i=1}^{n} x_i$.

```
mean([1 2 3 4 5 6 7])
```

This can also be used on matrices:

```
a = [1 2 3; 4 5 6];
mean(a,1)
mean(a,2)
```

the former giving the averages of the columns and the latter the averages of the rows.

`median` Gives the median of a set of data, that is the one in the middle when the data is ordered. This works in the same way as `mean` on matrices.

`min` Similar to `max` but giving the minimum.

`mod` This gives the remainder when the first argument is divided by the second. If used in the context `mod(x,1)` this gives the fractional part of `x`. It is similar to `rem`.

`NaN` Not-a-Number, used to return quantities which are not assigned, for instance 0/0.

`norm` Gives the mathematical norm of a quantity, particularly useful for working out the length of vectors. For vectors we have

```
norm(v,p)
```

gives

$$\left(\sum_{i=1}^{n} |v_i|^p \right)^{1/p}.$$

If `p` equals two this is a "conventional" norm and the commands

```
norm(v,inf)
norm(v,-inf)
```

give `max(abs(V))` and `min(abs(V))`, respectively.

`num2str` Converts a number to a string with a specified number of digits; `num2str(pi,4)`. This can also be used without specifying the number of digits (which uses the default corresponding to four places after the decimal point).

`ode23,ode45` , Hybrid Runge–Kutta routines to integrate functions, the former being a combination of second- and third-order schemes and the latter fourth and fifth.

We consider the solution of the differential equation

$$\frac{\mathrm{d}y}{\mathrm{d}t} = t^2 - y^2$$

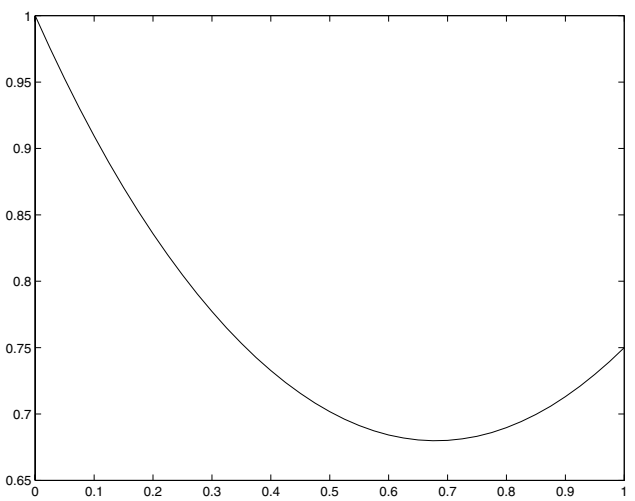
subject to the initial conditions $y(0) = 1$. Firstly we need to set up a function to give y':

```
function [out] = func(t,y)
out = t.^2-y.^2;
```

(which we shall presume has been saved as `func.m`). This can now be called using:

```
trange = [0 1];
yinit = 1;
[t,y] = ode45('func',trange,yinit);
```

This gives the solution:



We can set tolerances, amongst other options. Consider the system of differential equations

$$\frac{\mathrm{d}x}{\mathrm{d}t} = t - y$$
$$\frac{\mathrm{d}y}{\mathrm{d}t} = x,$$

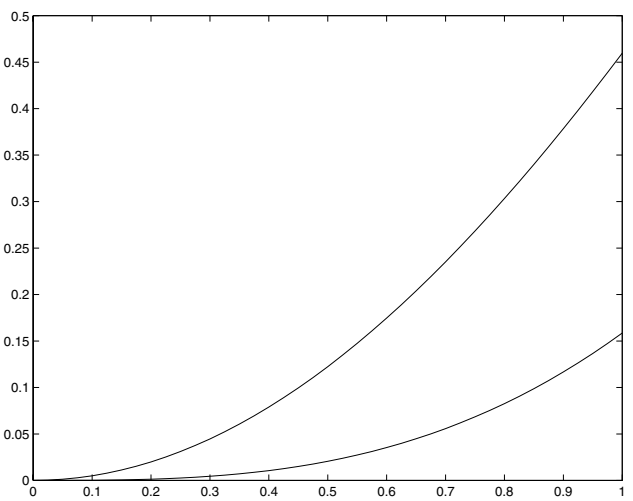
subject to the initial conditions $x(0) = y(0) = 0$. We introduce the vector $\mathbf{y} = (x(t), y(t))^T$, and as such we modify the code `func.m` to be:

```
function [out] = func(t,in)
% in(1) is x(t) and in(2) is y(t).
out = zeros(2,1);
out(1) = t-in(2);
out(2) = in(1);
```

This is called using the code:

```
trange = [0 1];
yinit = [0; 0];
options = odeset('RelTol',1e-4,'AbsTol',[1e-4 1e-4]);
ode45('func',trange,yinit,options);
```

This is for the above version of `func.m` for the coupled first-order systems, which sets the relative tolerance to be `1e-4` and the absolute tolerances to be `[1e-4 1e-5]`. This gives:



The upper line is $x(t)$ and the lower one is $y(t)$.

`ones` Sets up a matrix full of ones. `ones(n,m)` gives $\mathbf{A}$ which is an n-by-m matrix for which $a_{i,j} = 1$. `ones(n)` gives a square matrix (n-by-n).

`otherwise` If none of the `case`s correspond to the argument of the `switch` command then these statements are executed.

`path` A list of places MATLAB looks for files, also serves as a command to alter this variable. This command varies on different platforms and as such you should look at `help path`.

`pause` Causes the programme to wait a specified time, or can be used to wait until the user touches a key.

```
x = linspace(0,10);
for its = 1:5
   y = besselj(its/2,x);
   clf
   plot(x,y)
   pause
end
```

This programme runs through the functions $J_{n/2}(x)$ for $n = 1$, 2, 3, 4 and 5 as the user presses a key.

`pi` π - this is `4*atan(1)` or `imag(log(-1))`.

`poly` This returns the characteristic polynomial of a matrix.

```
a = [1 2 3;
     -1 2 0;
     -1 1 1];
poly(a)
```

This gives `[1.0000 -4.0000 10.0000 -7.0000]`, which represents $|\mathbf{A} - \lambda\mathbf{I}| = \lambda^3 - 4\lambda^2 + 10\lambda - 7$. The eigenvalues of the matrix can then be found using `roots`.

`polyfit` Tries to fit a polynomial of best fit, using a least squares idea. If there are n points in `x` (with no repeats) and `y`, and the user requests a polynomial of degree $n - 1$ then the fit is "exact":

```
x = [1 2 3];
y = [4 5 -2];
p = polyfit(x,y,2);
```

Note that the coefficients are returned with the one corresponding to the largest power first. This gives the quadratic $-4x^2 + 13x - 5$: however `polyfit(x,y,1)` on the same data gives the straight line $-3x+25/3$ (which is the line of best fit). It is possible to get information about the level of the fit using the form `[p,s] = polyfit(x,y,1)`. The object `s` contains information, for instance the covariance `s.R`.

`polyval` Evaluates a polynomial specified by its coefficients at a set of data points `y=polyval(c,x)`.

```
x = [1 2 3];
c = [-4 13 -5];
y = polyval(c,x);
```

This reconstructs the data used in the example for `polyfit`. Notice that the coefficients are given with the one corresponding to the largest power first.

`primes` Lists the primes up to and including the argument. The syntax is simply `n=20; primes(n)`.

`prod` Similar to `sum` and gives the product of the elements of the vector `x`, so that `prod(x)` returns $\prod_{i=1}^{n} x_i$.

```
x = 1:10;
prod(x) % Gives 10!
z = [1 4 5 6 -2];
prod(z)
```

The factorial could also be found using `factorial`.

`rand` Generates random numbers between zero and one. This can be used to form a matrix of random numbers as well. There are many versions of the argument for this command, see `help rand`.

`randn` Generates normally distributed real numbers: again can be used to generate matrices `randn(n,m)`.

`randperm` Generates a random permutation of a list of objects: To rearrange the letters of our names:

```
s = 'otto denier';
for its = 1:20
   l = randperm(11);
   s(l)
   pause
end
```

`rank` This yields the rank of a set of linear equations and can be used to see whether the system has no solutions, a unique solution or an infinite number of solutions (yielding information about the degrees of freedom).

```
A = [-1 1 1 2;
     3 -1 1 1;
     0 0 1 2];
rank(A)
```

`real` Gives the real part of a complex number `real(z)`, where `z` can be a scalar, vector or matrix.

`realmax` This is the largest floating point number representable on the computer.

`realmin` This is the smallest floating point number representable on the computer.

`rem` This is the remainder attained by dividing the first argument by the second one: `rem(3.32,1.1)` gives `0.02`.

```
rem([3 4 5],2) % Gives 1 0 1
rem(5,[1 2 3]) % Gives 0 1 2
rem([3 4 5],[1 2 3])
               % Gives 0 0 2
```

`reshape` This simply reshapes a matrix into a new shape. This is used most commonly to make a vector into a matrix, but can be used to `reshape` matrices.

```
s = rand(100,1);
a = reshape(s,10,10);
b = [1 3 4; 2 3 4];
d = reshape(b,1,6);
```

This gives `d = [1 2 3 4 4]`; the elements of `b` are read column-wise.

`roots` This gives the roots of the polynomial which is passed to the routine. For instance to find the roots of the cubic $x^3 + 4x^2 + 7x + 2$ we use

```
co = [1 4 7 2];
roots(co)
```

Notice again that the coefficients are listed with the one corresponding to the largest power first (similarly for `polyfit` and `polyval`).

`round` Rounds to the nearest integer.

`save` Saves values of variables to a .mat file.

`sin, asin` Sine and arcsine. These functions need to be used with brackets `sin(x)` and `asin(x)` (without, it produces bizarre results, for instance `sin pi` gives a one-by-two row vector with the elements sine of the ASCII code for "p" followed by the sine of the ASCII code for "i"). These functions can also be used for vectors and matrices.

```
x = -pi/2:pi/20:pi/2;
y = sin(x)
z = asin(y)
```

`sinh` Hyperbolic sine.

`size` Returns the dimensions of a matrix: `[rows,cols]=size(A)`.

`sort` This returns a list of numbers sorted into ascending order, together with a map from their original position to that in the revised list.

`sparse` Defines the matrix as `sparse` so that the computer only operates on non-zero entries: this can dramatically reduce the time spent doing computations (see `full`).

`spline` Fits cubic splines through a set of data points (x, f) and evaluates them at a further set of points z; `y=spline(x,f,z)`.

`sqrt` This finds the square root of a matrix element by element. If necessary the answer may be returned as a complex number.

`std` Calculates the standard deviation of a vector.

`str2mat` As the name suggests, this takes a string and returns a matrix.

`sum` This sums the contents of a vector, or the rows of a matrix. It can also be called with a second argument which defines which dimension needs to be summed.

`switch` This defines the start of a group of statements, the argument for which is a variable, the likely values of which are listed in the `case`s.

`tan,atan` Tangent and arctangent. These functions need to be used with brackets `tan(x)` and `atan(x)` (without, it produces bizarre results, for instance `tan pi` gives a one-by-two row vector with the elements tangent of the ASCII code for "p" followed by the tangent of the ASCII code for "i"). These functions can also be used for vectors and matrices.

```
x = 0:pi/20:pi/4;
y = tan(x)
z = atan(y)
```

(see also `atan2`).

`tour` Gives a tour of the facilities of MATLAB.

`transpose` Returns the transpose of a matrix: can also be done using `A.'`. (Note that `A'` transposes and also takes the conjugate.)

`type` Prints out the contents of a MATLAB script.

`upper` Converts the characters in a string to upper case:

```
name = 'Bob Roberts';
upper(name)
```

This gives `BOB ROBERTS` (the opposite command is `lower`).

`var` Gives the variance of a set of data. The covariance is defined as

$$\sigma_X = \frac{1}{N-1}\sum_{i=1}^{N}(x_i - \bar{x})^2. \tag{B.2}$$

`warning` Allows codes to issue warnings when there may be problems. Is also used to affect how the system issues warnings.

`which` Tells a user where a MATLAB file can be found, and which version is going to be run.

`while` Defines the start of a loop which is continued while a certain condition is fulfilled.

`whos` List of all variables (with details): this can be restricted using things like `whos a*`.

`zeros` Sets up a matrix full of zeros: `zeros(n,m)` gives $\mathbf{A}$ which is an n-by-m matrix for which $a_{i,j} = 0$. `zeros(n)` gives $\mathbf{0}_n$.

C
Solutions to Tasks

Please note that these solutions are given for guidance only and are by no means unique. At the outset we shall give MATLAB output: however subsequently we shall merely give the commands which can be used to solve the problems.

C.1 Solutions for Tasks from Chapter 1

Solution 1.1 *The MATLAB code to solve these problems is:*

```
x = 1.3;
p = x^2+3*x+1

x = 30/180*pi;
y = sin(x);

x = 1;
f = atan(x);

x = sqrt(3)/2;
h = acos(x);
g = sin(h)
```

The values this returns are: `6.5900`, `0.5000`, `0.7854` *(which is* $\pi/4$*) and* `0.5000`.

Solution 1.2 *To calculate the function* $y(x) = |x| \sin x^2$ *we use the code:*

```
x = pi/3;
y = abs(x)*sin(x^2);
```

and similarly for $x = \pi/6$. *Notice care is needed with the brackets and the syntax.*

Solution 1.3 *The MATLAB commands to determine these quantities are:*

`sin(pi/2);`

`cos(pi/3);`

`tan(60/180*pi);`

`x=0.5; log(x+sqrt(x^2+1))` *(and with* `x=1`*);*

`x=0; x/((x^2+1)*sin(x))`

and finally `x=pi/4; x/((x^2+1)*sin(x))`.

Notice the penultimate part of this task generates `NaN`*: this is because MATLAB does not know how to evaluate zero divided by zero.*

Solution 1.4 *This is a matter of either typing out all the values or exploiting the fact that MATLAB can operate on vectors. We can use the code:*

```
x = [0.3 1/3 0.5 1/2 1.65 -1.34];
round(x)
ceil(x)
floor(x)
fix(x)
```

This example is included to help understand the rôle of various MATLAB commands which can be used to return different roundings to appropriate integers.

Solution 1.5 *The difference between* `rem` *and* `mod` *can be illustrated using the* `y` *value of 4. Thus we have*

```
>> x=[3 4 5]

x =

     3     4     5

>> rem(x,4)

ans =

     3     0     1

>> mod(x,4)

ans =

     3     0     1

>> rem(x,-4)

ans =

     3     0     1

>> mod(x,-4)

ans =

    -1     0    -3

>>
```

So for the second argument being positive we have that `rem` *and* `mod` *are equivalent, whereas for negative values the remainder is signed (thus it shows whether it is positive or negative).*

Solution 1.6 *The MATLAB code is:*

```
x = 1:0.1:2;

% Part 1
y = x.^3+3*x.^2+1;

% Part 2
y = sin(x.^2);

% Part 3
y = (sin(x)).^2;

% Part 4
y = sin(2*x)+x.*cos(4*x);

% Part 5
y = x./(x.^2+1);

% Part 6
y = cos(x)./(1+sin(x));

% Part 7
y = 1./x+x.^3./(x.^4+5*x.*sin(x));
```

Solution 1.7

```
x = 3:0.01:5;
y = x./(x+1./x.^2);
```

Solution 1.8

```
x = -2:0.1:-1;
f = 1./x;
y = f.^3+f.^2+3*f;
```

Solution 1.9 *The code should read*

```
clear all
x = linspace(0,1,200);
g = x.^3+1;
h = x+2;
z = x.^2;
y = cos(x*pi);
f = y.*z./(g.*h);
f(200)
```

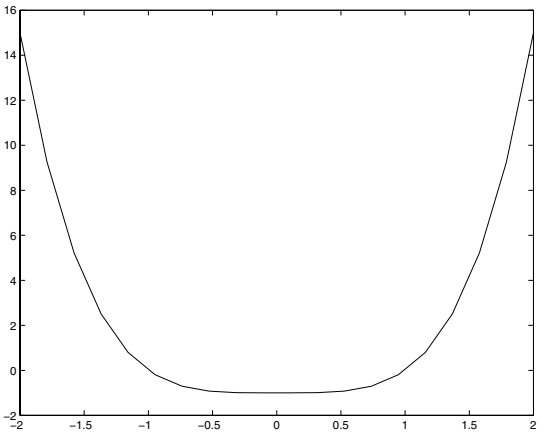
The errors were: the second line should have been first, else this cleared out the contents of `x`*. The default number of points for* `linspace` *is 100, so this needed to be specified as being 200. There were dots missing from the definition of* `g` *and the calculation of* `f`*. MATLAB distinguishes between upper and lower case in variable names so we need to use* `h` *rather than* `H`*. The command to calculate* `y` *needs brackets around the argument of the cosine function and an asterisk between* `x` *and* `pi`*. As mentioned above the dots were missing from* `f` *and the denominator of the fraction needed to be contained within brackets. Finally the answer needs to be printed (which could have been done at the prompt).*

Solution 1.10 *The code should read*

```
x = linspace(-2,2,20);
c = [1 0 0 0 -1];
y = polyval(c,x);
plot(x,y)
```

The errors were: in the first line the vector as defined would contain 21 entries (try typing `length(x)` *after running the original code). A quartic actually has 5 coefficients, so there was a zero missing in the definition of* `c` *and finally the* `x` *and* `y` *needed to be transposed in the plotting command.*

This gives the figure



Solution 1.11 *The corrected code should be*

```
x = 0:0.1:3;
f = x.^3.*cos(x+1);
% x = 2
f(21)
% x = 3
f(end)
```

C.2 Solutions for Tasks from Chapter 2

Solution 2.1 *This is hopefully just a matter of typing the code and saving the answer correctly.*

Solution 2.2 *The revised code could be:*

```
a = input('Enter a : ');
b = input('Enter b : ');
res = a^b;
str1 = 'The answer is ';
str2 = '  when ' ;
str3 = ' is raised to the power ';
disp([str1 num2str(res) str2 ...
  num2str(a) str3 num2str(b)]);
```

Solution 2.3 *The function for this purpose is:*

```
function [out] = twox(x)
out = 2.^x;
```

Note the use of the dot so that it can be called with a vector (or even matrix). Try it with `x=1:8`.

Solution 2.4 *We can do this calculation using only one function*

```
function [out1,out2] = func(x,y)
out1 = x.^2 - y.^2;
out2 = sin(x+y);
```

or using the two codes:

```
function [out] = func1(x,y)
out = x.^2 - y.^2;
```

and

```
function [out] = func2(x,y)
out = sin(x+y);
```

and then using the plot commands in the example. To extend the range to $[0, 2\pi]$ *the first line would need to be changed to* `x=0:pi/10:2*pi;`.

Solution 2.5 *The code should be modified to:*

```
function [out1,out2,out3] = xfuncs(x)
out1 = sin(x);
out2 = cos(x);
out3 = out1.^2 + out2.^2;
```

where we have used the variables `out1` *and* `out2` *to set the value of* `out3` *(which is actually always going to be 1).*

Solution 2.6 *This code takes two inputs and returns two outputs:*

```
function [outx,outy] = mapcode(inx,iny)
outx = mod(inx+iny,1);
outy = mod(inx+2*iny,1);
```

Solution 2.7 *This requires the modification of the second line to read* `y = x.^3+3*x`. *The change in the limit requires the modification of the first line to* `x = -4:1/4:6`. *It is perhaps a good idea to clear out the variables, which is done using* `clear all`.

Solution 2.8 *This question can be solved by noting that this is in fact an equation which is quadratic in* x^2 *and as such can be solved using the formula to have roots:*

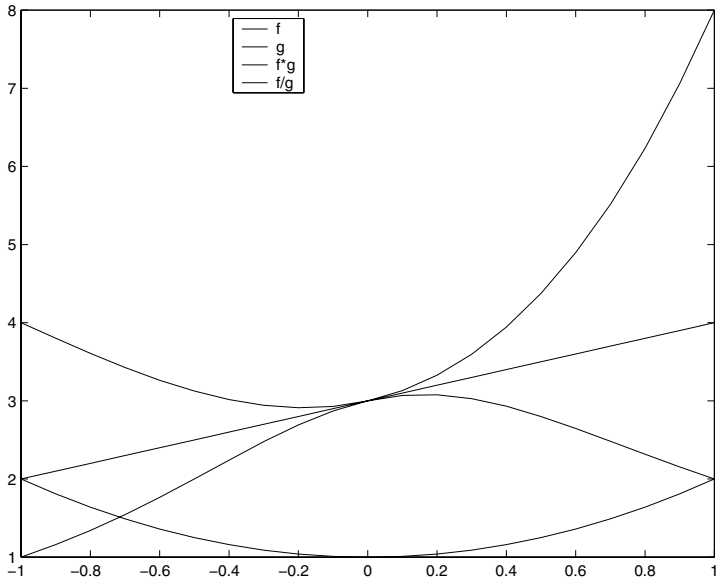
$$x^2 = \frac{-1 \pm \sqrt{1-4a}}{2}.$$

In order to have real roots we require $1 - 4a \geqslant 0$ *so that* $1/4 \geqslant a$ *and the quantity* x^2 *must be positive so that* $\sqrt{1-4a} > 1$, *which means that a has to be less than or equal to zero. This condition is more restrictive than the previous one so consequently we require* $a \leqslant 0$.

Solution 2.9 *In the question the step is not specified: however we shall use* $1/10$ *since this gives relatively smooth functions:*

```
x = -1:0.1:1;
f = x+3;
g = x.^2+1;
fg = f.*g;
f_over_g = f./g;
clf
plot(x,[f; g; fg; f_over_g])
legend('f','g','f*g','f/g',0)
```

This gives



We have added a legend which uses the line styles to show which lines we have plotted. We could have used different style lines. We have used the additional argument at the end of the `legend` *command* `,0` *to tell MATLAB to put the legend in the "best" position.*

Solution 2.10 *Here we shall provide details of how these codes can be improved (or in some cases actually run).*

1. *In the first line we just add a semicolon on the end to suppress output. The second line contains a terrible error, although this is fine as a mathematical equation. In MATLAB you cannot set* `x+2` *equal to* `y`*, we need to set* `y` *equal to* `x+2`*. In the third line: the MATLAB variable for* π *has a lower case "p" and we are missing an asterisk to denote multiplication and brackets around the denominator of the fraction. The corrected code is:*

```
x = 4;
y = x+2;
z = 1/(y^2*pi);
```

2. *In the first line we are missing a single quote and a semicolon. In the definition of the* `for` *loop we have introduced a variable* `n`*, which should be* `N`*. We have the loop variable as* `i` *whereas it is used as* `j` *within the loop. On the next line we have the brackets missing which should surround the*

denominator of the fraction, forcing it to be evaluated first. In the display line the first square bracket is missing and the answer needs to be converted to a string, which should be ***sum*** *not* ***s****. On top of this we have failed to set* ***sum*** *to be zero outside the loop and the command within the loop merely gives the final value rather than calculating the cumulative sum.*

The corrected code is:

```
N = input('Enter N ');
sum = 0;
for j = 1:N
    sum = sum + 1/j+ 1/((j+2)*(j+3));
end

disp(['The answer is ' num2str(sum)])
```

3. *In the first line we have used two equals signs where we should only have one. Two equals signs ask if* ***x*** *is equal to the right-hand side, rather than setting* ***x*** *equal to it. The rest of the errors are in the second line. In the numerator of the fraction we are missing an asterisk and brackets to show that we are taking the cosine of* ***x****. In the denominator we have unbalanced brackets (an extra round bracket needs to be added at the end). We have also used a pair of square brackets which should be round. There is an extra asterisk after the division sign. Since we are operating on a vector all of the operators should be preceded with a dot. The corrected code is:*

```
x = 0.0:0.1:1.0;
f = x.*cos(x)./((x.^2+1).*(x+2))
```

4. *The first line of this code gives us a nine-by-nine matrix rather than a row vector. The third line is fine. In the* ***for*** *loop we have missed out all the asterisks and the* ***end*** *which terminates the loop. The colons at the end of the lines in the loops need to be changed to semicolons.*

   The corrected code is:

```
w = ones(1,9);

w(1) = 1;

for j = 1:4
    w(2*j) = 3;
    w(2*j+1) = 2*j+1;
end
```

Solution 2.11

(a) Here the conversion factor will be worked out by remembering that there are 1760 yards in a mile, a yard is 36 inches and one inch is 2.54cm:

```
s = 'Enter speed in mph ';
sp_mph = input(s);

sp_yards_ph = sp_mph*1760;
sp_inch_ph = sp_yards_ph*36;
sp_cm_ph = sp_inch_ph*2.54;
sp_m_ph = sp_cm_ph/100;
sp_km_ph = sp_m_ph/1000;
disp(['Speed in km/h is ' ...
      num2str(sp_km_ph) ])
```

(b) This is essentially the reverse of the example above:

```
s = 'Enter speed in m/s ';
sp_mps = input(s);
sp_cmps = sp_mps*100;
sp_inch_ps = sp_cmps/2.54;
sp_yard_ps = sp_inch_ps/36;
sp_miles_ps = sp_yard_ps/1760;
sp_mph = sp_miles_ps *3600;
disp(['Speed in mph is ' ...
      num2str(sp_mph)])
```

(c) We now take the solution above and convert it to be a function, so we have

```
function [output] = change(input);
sp_mph = input;
sp_yards_ph = sp_mph*1760;
sp_inch_ph = sp_yards_ph*36;
sp_cm_ph = sp_inch_ph*2.54;
sp_m_ph = sp_cm_ph/100;
sp_km_ph = sp_m_ph/1000;
output = sp_km_ph;
```

(d) This shows that a sprinter will run at 22 mph on average since they do 100 metres in 10 seconds, that is 10 metres per second.

Solution 2.12 *This is accomplished using the code:*

```
x = linspace(-pi/2,pi/2);
f = x./(1+x.^2);
g = tan(x);
fg = g./(1+g.^2);
gf = f./(1+f.^2);
plot(x,fg,x,gf)
```

Solution 2.13 *This is done using the code:*

```
a = input('Coefficient of x squared: );
b = input('Coefficient of x:');
c = input('The constant:');
y = linspace(0,pi);
x = sin(y);
q = a*x.^2+b*x+c;
plot(y,q)
```

C.3 Solutions for Tasks from Chapter 3

Solution 3.1 *The required code is:*

```
s = 0;
for i = 1:100
    s = s+1/i^2;
end
disp(['Required value is ' num2str(s)])
```

Notice that we have changed the command `int2str` *to* `num2str`*: this is because the answer is no longer an integer.*

Solution 3.2 *This only requires modification of the* `for` *line to* `for i=1:2:100`*, which gives a vector which increases in twos.*

Solution 3.3 *The code* `f.m` *needs to be changed to*

```
function [value] = f(i)

value = sin(i*pi/2)/(i^2+1);
```

and then use the same code.

Solution 3.4 *This is accomplished using the code:*

```
x = 0:pi/4:pi;
f = x.^2+1;
```

Solution 3.5 *The code requires very minor modification to:*

```
v = 0.:0.25:0.75;
cosx = zeros(size(v));
N = 10; range = 0:N;
ints = 2*range;
for n = range
    cosx = cosx + ...
    (-1)^n*v.^ints(n)/factorial(ints(n));
end
```

This gives very accurate answers:

```
cos(v)-cosx

ans =

     0     0     0     0
```

This means that the difference between the MATLAB and the series answers are less than `eps`.

Solution 3.6 *We can use code which performs the summations separately for different values of N or note that $S_{N+1} = S_N + 1/(N+1)^2$ where $S_1 = 1$.*

This leads to the simple code:

```
s(1) = 1;
for n = 1:2000
    s(n+1) = s(n)+1/(n+1)^2;
end
```

This gives a value which when divided by π^2 allows one to appreciate that $c \sim 6$. In fact

$$\sum_{i=1}^{\infty} \frac{1}{n^2} = \frac{\pi^2}{6}.$$

Solution 3.7 *We solve this task by using a nested loop structure*

```
for p = 1:4
    sum = 0;
    for j = 1:(p+1)
        sum = sum + j^p;
    end
    disp([' sum for p=' ...
     int2str(p) ' is ' int2str(sum)])
end
```

Solution 3.8 *The codes for this task are:*

```
sumln(1) = -1;
for n = 1:1000
    sumln(n+1) = sumln(n)+(-1)^(n+1)/(n+1);
end

sum2(1) = 1/2;
for n = 1:1000
    sum2(n+1) = sum2(n)+1/((n+1)*(n+2));
end
```

where we have used the terms in the summations evaluated at the $(N+1)^{\text{th}}$ *place.*

Solution 3.9 *The simplest way of doing this would be* `x>2&x<4`*, although there are other ways* $\sim$`xor(x>=2,x=<4)` *for instance. The second example can be done with* `xor`*using* `xor(x>3,x<-1)` *or with* `or` *using* `x>3|x<-1` *(you can use both since the sets are disjoint).*

Solution 3.10 `mod(n,2)==0` *tells us that the remainder when dividing by two is zero (that is* n *is even). In order to ensure that this is only true for values of* $n > 20$ *we need the statement* `mod(n,2)==0 & n>20`*.*

Solution 3.11 *This requires us to work out* $\tan(73\pi/4)$ *but since* tan *is periodic this is equal to* $\tan(\pi/4)$ *(which is one). Hence the first condition is true so* x *is set to 2 and as this is an integer* `floor(x)` *is equal to* `x` *and so* `x` *is set to 10, which is not prime so* `x` *is returned as the string "False". This changes with the initial value of* x*, try for instance* $x = 3$*.*

Solution 3.12 *This uses*

```
start = 1/7;
next = mod(5*start,1)
while floor(next*7) ~= floor(start*7)
      next = mod(5*next,1)
end
```

Here we have used quite a complicated structure to deal with the rounding errors intrinsic to MATLAB (noting that the only possible answers are $n/7$ *where* $n \in \mathbb{N}$*).*

Solution 3.13 *We have*

```
n = 1:50;
f = n.^3-n.^2+40;
ii = find(f > 1000 & mod(n,3) ~= 0);
n(ii)
```

Solution 3.14 *The key here is to start with the string of the first ten letters, namely "abcdefghij". We ask the user to enter the first value of n outside the* `while` *loop: this avoids the need for the* `first` *flag. The required code is then:*

```
str = 'abcdefghij';
msg = 'Enter an integer from 1 to 10: ';
n = input(msg);

while (round(n)~=n) | (n<1 | n>10)
    warning(' Not valid ')
    n = input(msg);
end
str(1:n)
```

Solution 3.15 *Although this problem can be solved in one code it is preferable to use a couple of functions. The first one checks whether a character is a letter (lower or upper case):*

```
function [val] = isletter(charac)

lchar = lower(charac(1));
if lchar>='a' & lchar <='z'
   val = 1;
else
   val = 0;
end
```

Firstly the first character is extracted and converted to lower case. Then a check is made to see if it is a character in the lower case alphabet: if it is the variable `val` *is set to be true (that is 1). A similar code checks for whether a character is a digit:*

```
function [val] = isdigit(charac)

lchar = charac(1);
if lchar>='0' & lchar <='9'
   val = 1;
else
   val = 0;
end
```

We can now use the functions:

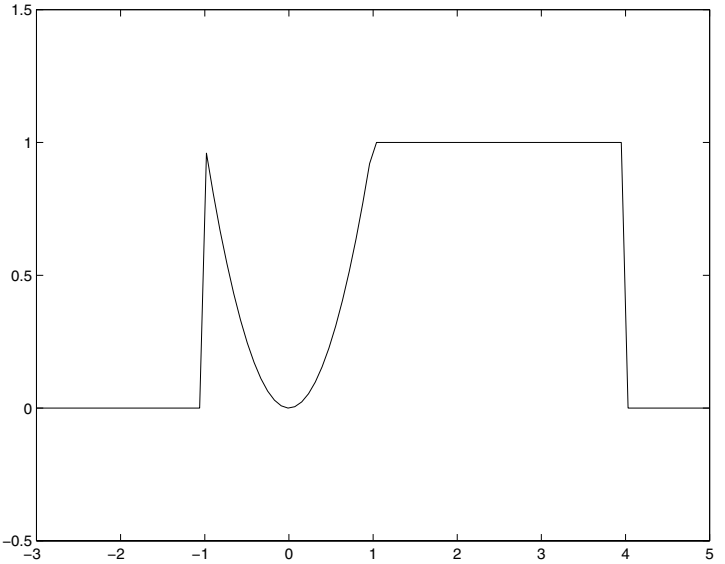
```
msg = 'Please enter a letter and a digit ';

str = input(msg,'s');

while ~isletter(str(1)) | ~isdigit(str(2))
      warning('This is not valid')
      str = input(msg,'s');
end
```

The argument of the `while` *loop checks to see if either of the conditions isn't satisfied (and as such uses* `or`*, that is the vertical line).*

Solution 3.16 *This can be done with the code:*

```
x = linspace(-3,5,100);
for i = 1:length(x)
    if x(i) >= -1 & x(i) <= 1
       f(i) = x(i)^2;
    elseif x(i) > 1 & x(i) < 4
       f(i) = 1;
    else
       f(i) = 0;
    end
end
plot(x,f)
axis([-3 5 -0.5 1.5])
```

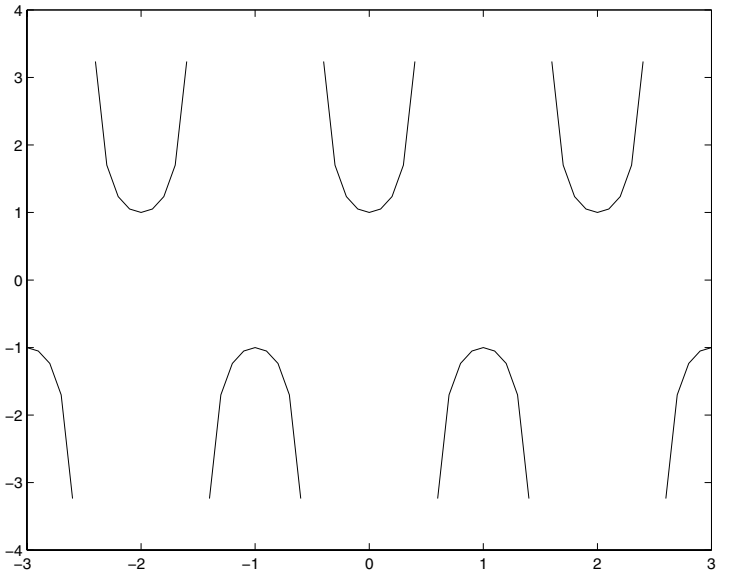
The final command is added purely so that the curve can be distinguished from the axis. This gives



Solution 3.17 *Here we use the code*

```
x = -3:0.1:3;
g = cos(pi*x);
izero = find(abs(g)<=1e-15);
ii = find(abs(g)>=1e-15);
f(izero) = NaN;
f(ii) = 1./g(ii);
plot(x,f)
```

which gives



and the choice of 10^{-15} *is in a sense arbitrary, but reflects how close we come to the singularities.*

Solution 3.18 *The first line contains a spelling mistake: the command should be* `linspace`*. The second line is missing brackets around* `x` *and a semicolon. In the definition of the* `for` *loop we should have used a colon rather than a semicolon. In both the logical expressions on the* `if` *and the* `elseif` *lines the second reference to the array* `x` *uses* `i` *rather than* `j`*. The first logical expression should use an ampersand (&) rather than the word* `and`*; similarly the second one should also have an ampersand rather than the word* `or`*. The first part of the second expression* `x(j)` *should be checked to be greater than one or equal to one (although this change is academic). The following line should refer to* `x(j)` *rather than the entire vector* `x`*, and should be finished off with a semicolon. The variable* `zero` *is used without definition. Finally we are missing an* `end` *statement to balance with the* `for`*; a fact which would be clear if the correct indentation was used.*

The corrected code is:

```
x = linspace(-4,4);
N = length(x);

for j = 1:N
    if x(j) >= 0 & x(j) <= 1
       f(j) = x(j);
    elseif x(j) >= 1 & x(j) < 2
       f(j) = 2 - x(j);
    else
       f(j) = 0;
    end
end
```

C.4 Solutions for Tasks from Chapter 4

Solution 4.1 *The solution to this task involves noting that the zeros of a function* $f(x) = g(x)h(x)$ *will occur at the zeros of the functions* $g(x)$ *and* $h(x)$*, provided neither of the functions are singular. In this case* $g(x) = x$ *is zero at* $x = 0$ *and* $h(x) = \sin x$ *is zero at* 0*,* π*,* 2π *and* 3π *(within the range* $[0, 10]$*).*

Solution 4.2 *We note that* $\cosh x = (\mathrm{e}^x + \mathrm{e}^{-x})/2$ *so in order for* $\cosh x$ *to be zero we require that* $\mathrm{e}^x = -\mathrm{e}^{-x}$ *or multiplying through by* e^x *that* $\mathrm{e}^{2x} = -1$*, but since* $\mathrm{e}^x > 0$ *for all* x*, this can never be so. Consequently* $\cosh x$ *is never zero. Similarly for* $\sinh x$ *we find that* $\mathrm{e}^{2x} = 1$ *which is only true when* $x = 0$*, which is the single isolated zero of* $\sinh x$*. The zeros of the function of* $f(x)$ *occur at the zeros of* $\cosh^m x$ *and* $\sinh^n x$*, which are only at* $x = 0$ *(in which case it is an* n*-fold zero).*

Solution 4.3 *We consider the discriminant of the equation, which is* $b^2 - 4$*. For two real roots we have* $b^2 > 4$*, in which case* $|b| > 2$*, for one real root the discriminant is zero, so that* $b = 2$ *and finally for complex roots the discriminant is negative so that* $|b| < 2$*. This can be verified graphically using:*

```
b = input('Value of b ');
x = -10:0.1:10;
f = x.^2+b*x+1;
plot(x,f)
```

Notice that we have chosen the range $[-10, 10]$ *but we could have used knowledge of the structure of the function to make sure that both roots were on the image (where* $|b| > 2$*).*

Solution 4.4 *We shall try to write the equation in the form*

$$\frac{\cos\theta \sin x + \sin\theta \cos x}{\cos\theta}$$

in which case $\tan\theta = \beta$*, in which case* $\sin\theta = \beta/\sqrt{1+\beta^2}$ *and* $\cos\theta = 1/\sqrt{1+\beta^2}$*. Now the function* $f(x)$ *can be written as*

$$f(x) = \frac{\sin(\theta + x)}{\cos\theta},$$

which has zeros at $\sin(\theta+x) = 0$*, hence* $x = n\pi - \theta$ *where* $\theta = \sin^{-1}(\beta/\sqrt{1+\beta^2})$ *(which is evaluated in MATLAB using* `asin`*). If* $\beta = 0$ *then* $\theta = 0$ *and if* $\beta = 1$ *then* $\theta = \pi/4$*.*

Solution 4.5 *The equation* $f(x) = 0$ *can be rewritten in many forms but we choose* $x = \cos^{-1}(\sin x/2)$ *so that a fixed point scheme would be* $x_{n+1} = \cos^{-1}((\sin x_n)/2)$ *and the corresponding code is:*

```
function g = eqn(x)
g = acos(sin(x)/2);
```

The roots of this function can be determined analytically using a similar method to the previous example and are found to be $n\pi + \theta$ *where* $\theta = \sin^{-1}(2/\sqrt{5})$*.*

Solution 4.6 *The roots of this equation occur at*

$$x_b = \frac{-b \pm \sqrt{b^2 - 4}}{2}.$$

Firstly considering the option $g(x) = -(x^2+1)/b$ *we have that* $g' = -2x/b$ *which at the roots is*

$$g'(x_b) = 1 \mp \sqrt{1 - \frac{4}{b^2}}.$$

Considering $|b| > 2$ the modulus of this function is greater than one for the root corresponding to the negative sign and less than one for the root with the positive sign. For the other option we find that

$$g'(x) = \frac{b}{2\sqrt{-(bx+1)}}.$$

When the roots are substituted in we find that the above situation is reversed. Using the code

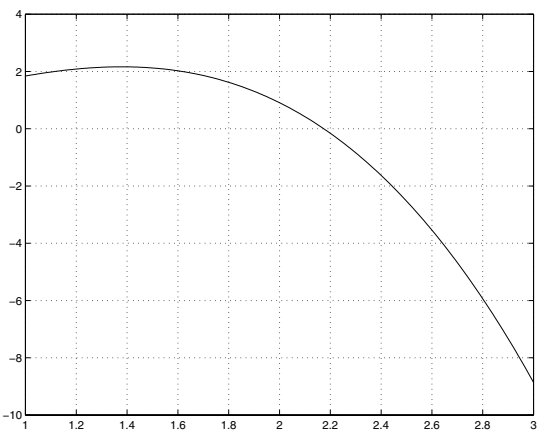
```
function g = eqn(x)
b = 3;
g = -(x^2+1)/b;
```

or with the alternative final line `g = -sqrt(-(b*x+1));` *we find starting with an initial guess of -1 we get different roots depending on which fixed point scheme we use.*

Solution 4.7 *We change the file* `func.m` *to be*

```
function [f] = func(x)
f = 2*x.^2-x.^3+sin(x);
```

Using the routine we produce the plot



Using 2 and 3 as the ends of the range we obtain

```
>> mbisect
 Root = 2.1741 found in 14 iterations
```

with a tolerance of 1×10^{-4}.

Solution 4.8 *The function $f(x) = \cos 3x$ has three zeros in the range 0 to π. Using the full range, we encounter the left root $\pi/6$ (in fact we should check whether $f((b+a)/2)$ is smaller than the tolerance, which it is in this case. For the other two ranges we can select the lower or upper root. We note that the scheme still works for an odd number of roots (since the scheme eliminates them in pairs).*

Solution 4.9 *This is merely a matter of setting up the routines*

```
function [f] = func(x)
f = x.*cos(x)-sin(x);
```

```
function [fp] = func_prime(x)
fp = -x.*sin(x);
```

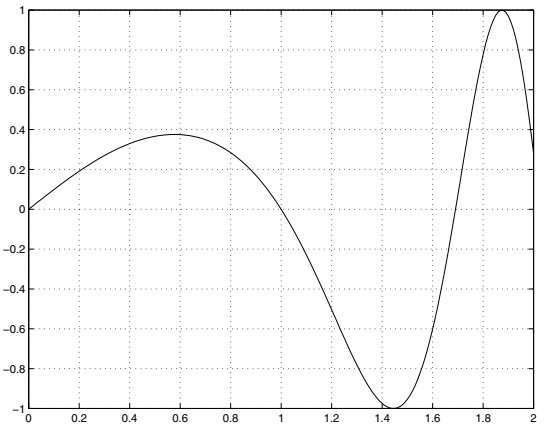
```
function [f] = func(x)
f = (x.^3-x).*sin(x);
```

```
function [fp] = func_prime(x)
fp = (3*x.^2-1).*sin(x) ...
       +(x.^3-x).*cos(x);
```

The roots of the first function are at $x = 0$ and $x \approx \pm 4.41$ and many other roots which tend to the zeros of $\cos x$ (as x increases).

The other function has zeros at $x = 0$ and 1, and then at $n\pi$ where $n \in \mathbb{Z}$.

Solution 4.10 *The zeros of this function occur where $x - x^3 = n\pi$. In order to obtain initial estimates for the range we plot the function*



Now using the code `False_Position.m`*:*

```
>> False_Position
Starting guess point 1 :0.8
Starting guess point 2 :1.2
 Root = 0.9999999982 found in 12 iterations.
>> False_Position
Starting guess point 1 :1.6
Starting guess point 2 :1.8
 Root = 1.690631797 found in 4 iterations.
```

The first root corresponds to $n = 0$ *above. We note that the function gets very oscillatory as* x *increases and may pose problems as more roots are required, in which case the roots of the cubic* $x - x^3 = n\pi$ *can be sought, using for instance* `roots`*.*

Solution 4.11 *We use the code on page 124 which calls:*

```
function [f,fp,fpp] = fun2(x);
f = x.^3-4*x.^2+5*x+2;
fp = 3*x.^2-8*x+5;
fpp = 6*x.^2-8;
```

The roots are at 1 (twice) and 2.

Solution 4.12 *The roots can be calculated using the inline code* `roots([1 1 1 1])` *and* `roots([1 0 0 1 -2 -4])`*.*

Solution 4.13 *This can be done using the code*

```
function [f]=func1(x)
f = x.*sin(x)+cos(x);
```

and then use the code `fzero('ff',3)` *which gives the root* ≈ 2.7984. *There are many others. For the other cases the roots are:* $\sin x = 0$ *or* $\sin x = \pm 1$, *so that* $x = n\pi/2$ *(*$n \in \mathbb{Z}$*); in this case* `fzero` *fails and returns a root at* $x = 1$. *The code has mistaken the fact that the function changes sign for a root.*

Solution 4.14 *The function* $J_{1/2}(x)$ *is actually* $\sin x/\sqrt{x}$ *and consequently the roots are* $n\pi$. *The code needed for* `fzero` *is*

```
function [f] = ourbess(x)
f = besselj(1/2,x);
```

and then `fzero('ourbess',3)`.

Solution 4.15 *The function we want to find the zeros of is* $f(x) = x\sin x - x^2\cos x - 1$ *which has derivative* $f'(x) = \sin x + x\cos x - 2x\cos x + x^2\sin x = (1+x^2)\sin x - x\cos x$. *The correct definitions for* $f(x)$ *and* $f'(x)$ *are:*

f.m

```
function [out] = f(in)
out = in.*sin(in)-in.^2.*cos(in)-1;
```

fp.m

```
function [out] = fp(in)
out = (1+in.^2).*sin(in)-in.*cos(in);
```

and the code to use these would be

```
x = 0;
for j = 1:10
    x = x -f(x)/fp(x);
end
```

(notice that the second term in this expression was the wrong way up in the question). This code could be written far more eloquently.

C.5 Solutions for Tasks from Chapter 5

Solution 5.1 *This code inputs four values which represent the points* (x_1, y_1) *and* (x_2, y_2)*. These are then made into vectors* $\mathbf{x}$ *and* $\mathbf{y}$*, where the former contains the* x *coordinates and the latter the* y *coordinates. The command* `polyfit` *fits a straight line through the points and returns* $y = mx + c$*, where the gradient* m *is the first element of* `p` *and the intercept* c *is the second. Then the final command displays the result. If the user enters both values of* y*, the same equation is just returned as* $m = 0$ *and* c *equals that value. On the other hand if two values of* x *are the same then MATLAB tries to give the line an infinite gradient, since it should be of the form* $x = d$*.*

Solution 5.2 *We write the quadratic as* $y = a + b(x - x_0) + c(x - x_0)(x - x_1)$ *where we choose* x_0 *as the* x *coordinate of one of the points. For convenience we shall take the origin as the first point so that* $x_0 = 0$ *and we see that* $a = 0$*. Using the point* $(2, -1)$ *as the next point we note that the quadratic is* $y = bx + cx(x - 2)$ *and* $-1 = 2b$*. Finally the condition that the curve goes through the final point yields the equation* $5 = -5/2 + 15c$ *so that* $c = 1/2$*. Hence the quadratic is*

$$\begin{aligned} y &= -\frac{1}{2}x + \frac{1}{2}x(x-2) \\ &= \frac{x(x-3)}{2}. \end{aligned}$$

Solution 5.3 *We use the code:*

```
x = 0:10;
co = [1 3 2];
y = x.^2+3*x+2;
for i = 1:3
    xv = i-0.5;
    p = polyfit(x(i:(i+1)),y(i:(i+1)),1);
    err(i) = polyval(p,xv)-polyval(co,xv);
end
```

This gives the same error in each case, namely $1/4$ *(which we would expect from understanding the error associated with this method of interpolation (consider the second derivative)).*

Solution 5.4 *This quadratic will be of the form* $cx(\pi - x)$ *(since it is zero at*

0 and π). The value of c can be determined from the condition that the curve goes through the final point which gives

$$y = \frac{4x}{\pi^2}(\pi - x).$$

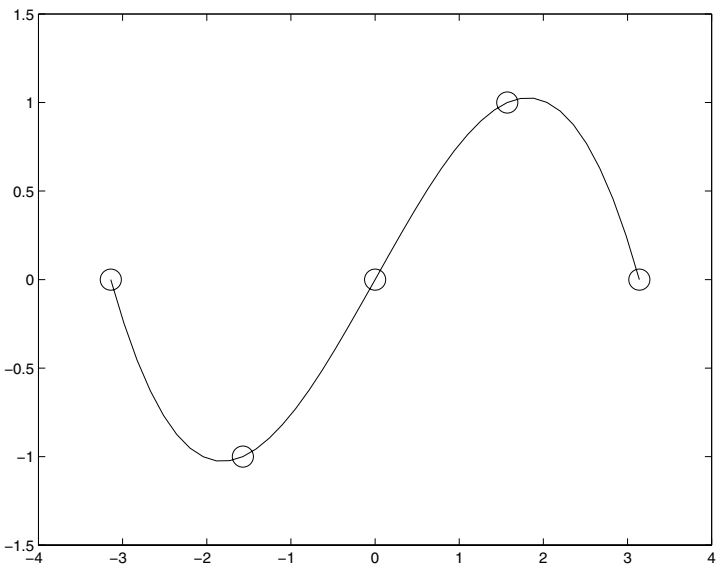
Solution 5.5 *Since the cubic is zero at the points 0 and π we know that it is of the form $x(\pi - x)(a(x - \pi/2) + b)$. The values of a and b can be determined from the other points: $(\pi/2, 1)$ gives $b = 4/\pi^2$; finally $(-\pi/2, -1)$ gives $a = 16/(3\pi^2)$. Hence the cubic is*

$$\begin{aligned} y &= x(\pi - x)\left(\frac{16}{3\pi^3}(x - \pi/2) + \frac{4}{\pi^2}\right) \\ &= x(\pi - x)\left(\frac{16x}{3\pi^3}x + \frac{4}{3\pi^2}\right). \end{aligned}$$

Solution 5.6 *For this we shall use the MATLAB command* `spline` *so that*

```
x = -pi:(pi/2):pi;
y = [0 -1 0 1 0];
z = -pi:(pi/20):pi;
f = spline(x,y,z);
plot(z,f,x,y,'o','MarkerSize',14)
```

This gives:



Solution 5.7 *Here we shall use the method of least squares. As such we shall produce details of the formulation. We calculate the sum of the squares of the errors:*

$$e = \sum_{i=1}^{n} (a \sin x_i + b \cos x_i - f_i)^2$$

and partially differentiating with respect to a and b we have

$$\frac{\partial e}{\partial a} = \sum_{i=1}^{n} \sin x_i (a \sin x_i + b \cos x_i - f_i)$$
$$\frac{\partial e}{\partial b} = \sum_{i=1}^{n} \cos x_i (a \sin x_i + b \cos x_i - f_i).$$

These equations can be rewritten in matrix form as:

$$\begin{pmatrix} \sum_{i=1}^{n} \sin^2 x_i & \sum_{i=1}^{n} \sin x_i \cos x_i \\ \sum_{i=1}^{n} \cos x_i \sin x_i & \sum_{i=1}^{n} \cos^2 x_i \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} \sum_{i=1}^{n} f_i \sin x_i \\ \sum_{i=1}^{n} f_i \cos x_i \end{pmatrix}.$$

This can be solved using the code:

```
x = 0:0.1:1.0;
f = [3.16 3.01 2.73 2.47 2.13 1.82 ...
   1.52 1.21 0.76 0.43 0.03];
A = [sum(sin(x).^2) sum(cos(x).*sin(x)); ...
   sum(cos(x).*sin(x)) sum(cos(x).^2)];
r = [sum(f.*sin(x)); sum(f.*cos(x))];
sol = A\r;
```

This gives

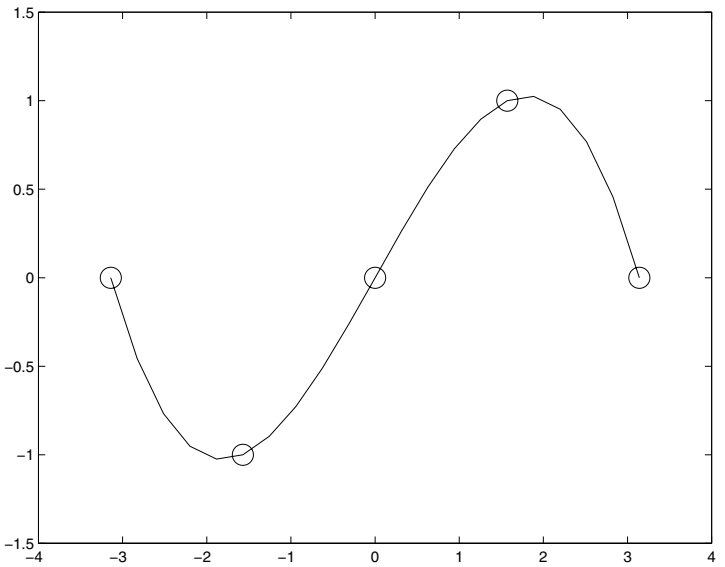
```
sol =

   -1.9941
    3.1892
```

The original data was actually generated with -2 and 3.2, and then noise was added.

Solution 5.8 *This can be done by hand but in fact MATLAB will actually return the required coefficients.*

```
x = -pi:(pi/2):pi;
y = [0 -1 0 1 0];
z = -pi:(pi/10):pi;
pp = spline(x,y);
f = spline(x,y,z);
plot(z,f,x,y,'o','MarkerSize',14)
true = sin(z);
err = sum((true-f).^2);
```

This gives



and

```
>> pp.coefs

ans =

   -0.0860    0.8106   -1.6977         0
   -0.0860    0.4053    0.2122   -1.0000
   -0.0860    0.0000    0.8488         0
   -0.0860   -0.4053    0.2122    1.0000
```

which are the four cubic equations (given with the coefficients of x^3 *first). The total sum of the errors was* 0.1944.

Solution 5.9 *The corrected code is:*

```
x = 2:11;
f = polyval([1 0 0 -1],x) + sin(x);
% x = 4.5
r = 3:4;
c = polyfit(x(r),f(r),1);

yy = polyval(c,4.5)

% x = 15 (extrapolation)
r = length(x)-1:length(x);
c = polyfit(x(r),f(r),1);
yy = polyval(c,15);
```

C.6 Solutions for Tasks from Chapter 6

Solution 6.1 *These calculations can be repeated using the code*

```
A = [3 0 -1; -4 2 2];
B = [-1 7; 3 5; -2 0];
C = [2 0; -1 -3];
A*B
B*A
A+transpose(B)
A*C
A*transpose(C)
3*C+2*transpose(A*B)
(A*B)*C
A*(B*C)
```

Solution 6.2 *Notice that here we set up the matrix* `A` *before changing the elements and this is unnecessary but good practice. Also it is not necessary to set up* `r` *and this can be defined inline. The definition of* `r` *allows for more versatile code.*

```
r = 1:4;
A = zeros(4);
A(1,r) = r;
A(r,4) = flipud(r');
```

Note that the final command could also be replaced by `A(flipud(r),1)=r';` *you should try to understand this command.*

Solution 6.3 *Here we rely on the fact that MATLAB knows that a matrix with a super- or sub-diagonal of length 9 is a ten-by-ten matrix.*

```
a = diag(ones(1,9),1)+diag(-ones(1,9),-1);
```

Solution 6.4 *This is just a matter of typing the commands: however you should be able to decide which ones are viable before doing this.*

Solution 6.5 *The code to solve these problems is given by:*

```
A = [ 1 2; 3 4];
B = [3 4; -1 2];
A*B
C = [3 5; 6 -2];
D = [-1 0; 2 1];
2*C-4*D
E = [1 3 5];
F = [2 -1; -1 0; 7 -2];
E*F
```

This gives the answers

```
ans =

     1     8
     5    20


ans =
```

```
    10    10
     4    -8


ans =

    34   -11
```

Solution 6.6 *Consider a general matrix*

$$\mathbf{A} = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

multiplied by the matrix

$$\mathbf{X} = \begin{pmatrix} \alpha & \beta \\ \beta & \alpha \end{pmatrix}.$$

Now

$$\mathbf{XA} = \begin{pmatrix} \alpha a + \beta c & \alpha b + \beta d \\ \beta a + \alpha c & \beta b + \alpha d \end{pmatrix}$$

and

$$\mathbf{AX} = \begin{pmatrix} \alpha a + b\beta & a\beta + b\alpha \\ c\alpha + d\beta & c\beta + d\alpha \end{pmatrix}.$$

Now comparing $\mathbf{XA}$ *and* $\mathbf{AX}$ *we find that it is necessary for* $a = d$ *and* $b = c$ *(provided* $\beta \neq 0$*). Hence the only matrices which commute with matrices of the form* $\mathbf{X}$ *are those of the same form.*

```
stl = 'Top left element of matrix ';
sbl = 'Bottom left element of matrix ';
for j = 1:2
        a(j) = input([stl num2str(j) ': ']);
        b(j) = input([sbl num2str(j) ': ']);
end
A = [a(1) b(1); b(1) a(1)];
B = [a(2) b(2); b(2) a(2)];
disp(A*B)
disp(B*A)
```

Notice that the answer is also of the form $\mathbf{X}$*.*

Solution 6.7 *The* $(i,j)^{\text{th}}$ *element of* $\mathbf{B}$ *is* $a_{i,j}+a_{j,i}$ *and the* $(j,i)^{\text{th}}$ *element is* $a_{j,i}+a_{i,j}$ *(which are equal so* $\mathbf{B}$ *is symmetric). Similarly the* $(i,j)^{\text{th}}$ *element of* $\mathbf{C}$ *is* $a_{i,j}-a_{j,i}$*, whereas the* $(j,i)^{\text{th}}$ *element is minus this, namely* $a_{j,i}-a_{i,j}$ *so that* $\mathbf{C}$ *is anti-symmetric.*

Solution 6.8 *These matrices can be constructed in MATLAB*

```
theta = 0;
A0 = [cos(theta) sin(theta); -sin(theta) cos(theta)];
theta = pi/2;
A1 = [cos(theta) sin(theta); -sin(theta) cos(theta)];
theta = pi;
A2 = [cos(theta) sin(theta); -sin(theta) cos(theta)];
```

or mathematically and we find

$$\mathbf{A}|_{\theta=0} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix},$$

$$\mathbf{A}|_{\theta=\pi/2} = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix},$$

and finally

$$\mathbf{A}|_{\theta=\pi} = \begin{pmatrix} -1 & 0 \\ 0 & -1 \end{pmatrix}.$$

Working through the cases one at a time

$\theta = 0$ *This gives us the identity matrix, so multiplying leaves all points unchanged.*

$\theta = \pi/2$ *Here if we start with* $\mathbf{x} = (x,y)^T$ $\mathbf{Ax}$ *is* $(y,-x)^T$*. This moves the point round the origin by (unsurprisingly)* $\pi/2$*.*

$\theta = \pi$ *Now the action is to return* $(-x,-y)^T$*, which is a reflection in the origin (or in fact a rotation of* π*).*

In general the action of multiplying by this matrix is to rotate by θ *radians.*

We can work out the inverse of this matrix by noting that its determinant is unity and switching the terms on the leading diagonal and then multiplying the off diagonal terms by minus one. However, we could also exploit the fact that in order to invert the operation of rotating by an angle θ *we merely rotate by* θ *in the other sense (or more specifically by* $-\theta$*). The inverse is given by*

$$\mathbf{A}^{-1} = \begin{pmatrix} \cos(-\theta) & \sin(-\theta) \\ -\sin(-\theta) & \cos(-\theta) \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}.$$

This is easily verified by performing the multiplication of the matrices

$$\begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix}$$
$$= \begin{pmatrix} \cos^2\theta + \sin^2\theta & \cos\theta\sin\theta - \sin\theta\cos\theta \\ \sin\theta\cos\theta - \cos\theta\sin\theta & \sin^2\theta + \cos^2\theta \end{pmatrix}$$
$$= \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = \mathbf{I}.$$

Solution 6.9 *This is solved using the code:*

```
A = [3 4; -1 2];
b = [2 ; 0];
x = A\b;
```

which gives $x = 2/5$ and $y = 1/5$.

Solution 6.10 *This is solved using the code:*

```
A = [1 1 2; 1 -1 -3; ...
   -2 -5 1];
b = [1; 0 ; 4];
x = A\b
```

This gives $x = 4/5$, $y = -1$ and $z = 3/5$.

Solution 6.11 *We are able to add* $\mathbf{A}$ *and* $\mathbf{B}$ *since they are of the same size, namely they both have three rows and two columns.*

$$\begin{pmatrix} 1 & -1 \\ 0 & 2 \\ 3 & 2 \end{pmatrix} + \begin{pmatrix} 2 & -1 \\ -1 & 0 \\ 3 & 2 \end{pmatrix} = \begin{pmatrix} 3 & -2 \\ -1 & 2 \\ 6 & 4 \end{pmatrix}.$$

The matrices $\mathbf{A}$ *and* $\mathbf{C}$ *can be multiplied together since the number of columns of* $\mathbf{A}$ *(two) matches the number of rows of* $\mathbf{C}$*.*

$$\begin{pmatrix} 1 & -1 \\ 0 & 2 \\ 3 & 2 \end{pmatrix} \begin{pmatrix} -1 & 0 \\ 2 & 1 \end{pmatrix} = \begin{pmatrix} 1\times(-1) + (-1)\times 2 & 1\times 0 + (-1)\times 1 \\ 0\times(-1) + 2\times 2 & 0\times 0 + 2\times 1 \\ 3\times(-1) + 2\times 2 & 3\times 0 + 2\times 1 \end{pmatrix}$$
$$= \begin{pmatrix} -3 & -1 \\ 4 & 2 \\ 1 & 2 \end{pmatrix}.$$

The multiplication of $\mathbf{C}$ *times* $\mathbf{B}$ *is not possible since the number of columns of* $\mathbf{C}$ *(two) is not equal to the number of rows of* $\mathbf{B}$ *(three).*

First we calculate $\mathbf{A}-\mathbf{B}$ *(which is possible since both matrices are the same size). This gives another matrix of the same size (again with three rows and two columns), which can now multiply* $\mathbf{C}$ *since this has two rows. The answer is*

$$\begin{pmatrix} 1 & 0 \\ 3 & 2 \\ 0 & 0 \end{pmatrix}$$

The final calculation should give the same answer. The MATLAB code for these calculations is:

```
>> a = [1 -1; 0 2; 3 2];
>> b = [2 -1; -1 0; 3 2];
>> c = [-1 0; 2 1];
>> a+b

ans =

     3    -2
    -1     2
     6     4

>> a*c

ans =

    -3    -1
     4     2
     1     2

>> (a-b)*c

ans =

     1     0
     3     2
     0     0

>> a*c-b*c

ans =
```

```
     1     0
     3     2
     0     0
```

Solution 6.12 *These calculations can both be performed and the solutions are*

$$\begin{pmatrix} 3 \\ 10 \end{pmatrix} \text{ and } \begin{pmatrix} 16 & -2 & 9 & -3 \\ 0 & 2 & -5 & -1 \end{pmatrix}$$

which can be checked using MATLAB

```
>> [1 -1 2; 3 0 1]*[3; 2; 1]

ans =

     3
    10

>> [5 -2;-1 2]*[4 0 1 -1; 2 1 -2 -1]

ans =

    16    -2     9    -3
     0     2    -5    -1
```

Solution 6.13 *The results of both calculations merely returns the matrix unchanged. This is the effect of multiplying by the identity.*

Solution 6.14 *Firstly, we reflect in the leading diagonal to give* $\mathbf{A}^T$ *so that*

$$\mathbf{A}^T = \begin{pmatrix} 3 & 0 \\ 2 & -1 \\ -1 & -2 \end{pmatrix}$$

The results of the multiplications are

$$\begin{pmatrix} 14 & 0 \\ 0 & 5 \end{pmatrix} \text{ and } \begin{pmatrix} 9 & 6 & -3 \\ 6 & 5 & 0 \\ -3 & 0 & 5 \end{pmatrix},$$

which can also be done using MATLAB code:

```
>> A = [3 2 -1; 0 -1 -2];
>> A*transpose(A)

ans =

    14     0
     0     5

>> transpose(A)*A

ans =

     9     6    -3
     6     5     0
    -3     0     5
```

Solution 6.15 *We assume that a general row vector is of the form*

$$(x_1, x_2, \cdots, x_N)$$

and consequently its transpose is the column vector

$$\mathbf{x}^T = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{pmatrix}$$

Hence

$$\mathbf{x}\mathbf{x}^T = \begin{pmatrix} x_1 & x_2 & \cdots & x_N \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{pmatrix} = x_1^2 + x_2^2 + \cdots + x_N^2.$$

This is a scalar which is positive since it is merely the sum of squares.

Solution 6.16 *The matrix equation can be expanded to give*

$$\begin{aligned} x + 4y &= 1 \\ -2x + 3y &= -2 \end{aligned}$$

and the three simultaneous equations can be written as the single matrix equation

$$\begin{pmatrix} 1 & 1 & 1 \\ 1 & -2 & -1 \\ -1 & 3 & -1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 0 \\ 2 \\ -1 \end{pmatrix}.$$

Solution 6.17 *We simply present the code which can be used to determine the character of the systems (this exploits the code* `solns.m` *given on page 188)*

```
a = [3 2; 3 -2]; b=[7; 7];
solns(a,b)
a = ones(6);
for r = 2:6
  a(r,r) = -1;
end
b = ones(6,1);
solns(a,b)
```

This returns the comments:

```
There are 2 equations
with 2 variables
There is a unique solution
```

and for the second case

```
There are 6 equations
with 6 variables
There is a unique solution
```

Solution 6.18 *This can be accomplished using*

```
A = [1 0 0 -1; ...
   -1 2 -1 0; ...
   0 -1 2 -1;
   0 0 0 1];
r = [0 1;0 0; 0 0; 1 0];
sols = A\r;
```

This gives

```
sols =
```

```
1.0000    1.0000
1.0000    0.6667
1.0000    0.3333
1.0000         0
```

where we have solved both systems at once to give $(1,1,1,1)$ *and* $(1,\frac{2}{3},\frac{1}{3},0)$.

Solution 6.19 *We use the code*

```
s = pi:pi/3:(2*pi);
ns = length(s);
for j = 1:ns
    ss = s(j);
    A = [0 1 ss; ...
         ss 0 1; ...
         1 ss 0];
    z(j) = det(A);
end
c = polyfit(s,z,3)
```

which gives $c = (1,0,0,1)$ *so that the determinant of the matrix is* $s^3 + 1$.

Solution 6.20 *We note that*

$$\mathbf{B}^2 = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}\begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} = -\mathbf{I}.$$

As such we find that $\mathbf{B}^3 = \mathbf{B}\mathbf{B}^2 = -\mathbf{B}\mathbf{I} = -\mathbf{B}$ *and that* $\mathbf{B}^4 = \mathbf{B}^2\mathbf{B}^2 = (-\mathbf{I})(-\mathbf{I}) = \mathbf{I}$. *Hence we have the code*

```
n = input('What power :');
b = [0 1; -1 0];
switch mod(n,4)
      case 0
        bn = eye(2);
      case 1
        bn = b;
      case 2
        bn = -eye(2);
      case 3
        bn = -b;
end
```

Solution 6.21 *The eigenvalues can be determined using the code*

```
a = [1 0 0 -1; ...
   0 1 0 0; ...
   0 0 1 0; ...
   -1 0 0 1];
eig(a)
```

which gives 1 *(twice), 0 and 2.*

Solution 6.22 *We start with* $n = 1$ *which is merely the definition, that is* $\mathbf{A} = \mathbf{PDP}^{-1}$. *And we assume that our conjecture is true for* n, *that is* $\mathbf{A}^n = \mathbf{PD}^n\mathbf{P}^{-1}$. *Now premultiply by* $\mathbf{A}$

$$\begin{aligned}\mathbf{AA}^n &= \mathbf{A}(\mathbf{PD}^n\mathbf{P}^{-1})\\ \mathbf{A}^{n+1} &= \mathbf{PDP}^{-1}(\mathbf{PD}^n\mathbf{P}^{-1})\\ &= \mathbf{PD}(\mathbf{P}^{-1}\mathbf{P})\mathbf{D}^n\mathbf{P}^{-1}\\ &= \mathbf{PDD}^n\mathbf{P}^{-1}\\ &= \mathbf{PD}^{n+1}\mathbf{P}^{-1}.\end{aligned}$$

This is merely the statement of our initial conjecture for $n + 1$. *Thus we have shown by induction that* $\mathbf{A}^n = \mathbf{PD}^n\mathbf{P}^{-1}$.

Solution 6.23 *This gives*

```
>> co = charpoly(a);
>> roots(co)

ans =

   2.0000
   1.0000 + 0.0000i
   1.0000 - 0.0000i
   0.0000
```

This confirms the results above.

Solution 6.24 *The eigenvalues of this equation are* $(3 \pm \sqrt{5})/2$. *Consequently using the general form on page 216, we have*

$$\mathbf{x}(t) = \frac{1}{\sqrt{5}} \left\{ \frac{1}{2} \left((3+\sqrt{5})\mathrm{e}^{(3-\sqrt{5})t/2} - (3-\sqrt{5})\mathrm{e}^{(3+\sqrt{5})t/2} \right) \mathbf{I} \right.$$
$$\left. + \left(\mathrm{e}^{(3-\sqrt{5})t/2} - \mathrm{e}^{(3+\sqrt{5})t/2} \right) \mathbf{A} \right\} \begin{pmatrix} 1 \\ -1 \end{pmatrix}.$$

C.7 Solutions for Tasks from Chapter 7

Solution 7.1 *Please try it yourself first but this is the answer (or one of them):*

```
for i = 1:12
    switch mod(i,3)
         case 0
            f(i) = 1;
         case 1
            f(i) = 2;
         case 2
            f(i) = 3;
    end
end
```

There are many alternatives, for instance `f = mod(1:12,3)+1;`.

Solution 7.2 *For the one third case, we can work through the code with* $N = 9$*, so that*

`rodd=1:2:N` *gives* `[1 3 5 7 9]`
`reven=2:2:(N-1)` *gives* `[2 4 6 8]`
`weights(rodd=2)` *gives* `[2 0 2 0 2 0 2 0 2]`
`weights(1)=1` *gives* `[1 0 2 0 2 0 2 0 2]`
`weights(N)=1` *gives* `[1 0 2 0 2 0 2 0 1]`
`weights(reven)=4` *gives* $=$ `[1 4 2 4 2 4 2 4 1]`

and for the three eighths rule with $N = 10$ *we have*

```
m=(N-1)/3 gives 3
rdiff=3*(1:(m-1))+1 gives [3*(1:2)+1] that is [4 7]
weights=3*ones(1,N) gives [3 3 3 3 3 3 3 3 3 3]
weights(1) gives [1 3 3 3 3 3 3 3 3 3]
weights(N) gives [1 3 3 3 3 3 3 3 3 1]
weights(rdiff)=2 gives [1 3 3 2 3 3 2 3 3 1]
```

Solution 7.3 *This is done using the code:*

```
function [val] = fn(x)
val = log(x+sqrt(x.^2+1));
```

Solution 7.4 *We shall use forty points (which should be more than enough) and note that the exact answer is*

$$\int_{x=1}^{3} x^2 - 3x + 2 \, \mathrm{d}x = \left[\frac{x^3}{3} - \frac{3x^2}{2} + 2x \right]_1^3 = \frac{2}{3}.$$

The code is

```
N = 40;
x = linspace(1,3,N);
f = x.^2-3*x+2;
h = x(2)-x(1);
integral = (sum(f)-f(1)/2-f(N)/2)*h;
```

This gives the value 0.6675*, which is within* 8.7×10^{-4} *of the exact answer. Notice that by using either of Simpson's rules we could have retrieved the exact answer, since the original curve is a quadratic.*

Solution 7.5 *We use* $N = 11$ *and modify Simpson's 1/3 rule code on page 233, so that we have*

```
x = linspace(0,1,11);
h = x(2)-x(1);
N = length(x);
rodd = 1:2:N;
reven = 2:2:(N-1);
weights(rodd) = 2; weights(1) = 1;
weights(N) = 1; weights(reven) = 4;
f = x.^3-x+1;
integral = h/3*sum(weights.*f);
disp([integral])
```

This gives an answer of 0.75. *The exact answer is*

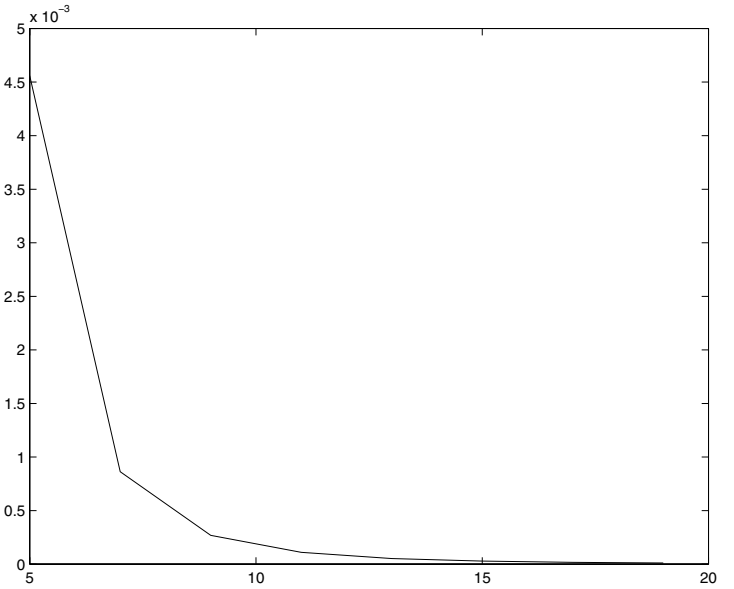
$$\int_0^1 x^3 - x + 1 \,\mathrm{d}x = \left[\frac{x^4}{4} - \frac{x^2}{2} + x\right]_0^1 = \frac{3}{4}.$$

So the scheme does exceedingly well and the error is of the order 10^{-16}. *This is unsurprising since the error is proportional to the fourth derivative, which is identically zero for a cubic.*

Solution 7.6 *In this task we produce a minor modification of the previous solution: the first line needs to read* `x=linspace(0,pi,N);` *and the line defining* $f(x)$ *needs modifying to* `f=sin(x)`. *This now allows us to try different values of* N, *which we do with a loop structure:*

```
Ns = 5:2:19;
for N = Ns
   clear rodd reven weights f x
   x = linspace(0,pi,N);
   h = x(2)-x(1);
   rodd = 1:2:N;
   reven = 2:2:(N-1);
   weights(rodd) = 2;
   weights(1) = 1; weights(N) = 1;
   weights(reven) =  4;
   f = sin(x);
   integral(N) = h/3*sum(weights.*f);
end
plot(Ns,abs(integral-2))
```

This gives



where we have plotted the errors versus the number of points. We have used the exact answer which is

$$\int_0^\pi \sin x \, \mathrm{d}x = [-\cos x]_0^\pi = 2.$$

As we can see the errors tend to zero very rapidly. Of course $\sin x$ *is a very smooth function over this interval and if we had a more oscillatory function more points would be needed.*

Solution 7.7 *We shall use the trapezium rule for simplicity. We also note that the value of this integral over the truncated domain is:*

$$\int_0^a \frac{1}{\sqrt{x^2+1}} \, \mathrm{d}x = \sinh^{-1}(a).$$

We note that $\sinh^{-1}(a) = \ln(a + \sqrt{a^2+1})$ *so in fact the value of the integral diverges, but very slowly.*

We use the code:

```
X = input('Truncate at:');
N = ceil(X)*3;
x = linspace(0,X,N);
h = x(2)-x(1);
f = 1./sqrt(x.^2+1);
int = (sum(f)-f(1)/2-f(N)/2)*h
```

The second line ensures that the step sizes will be smaller than $1/3$*. This gives:*

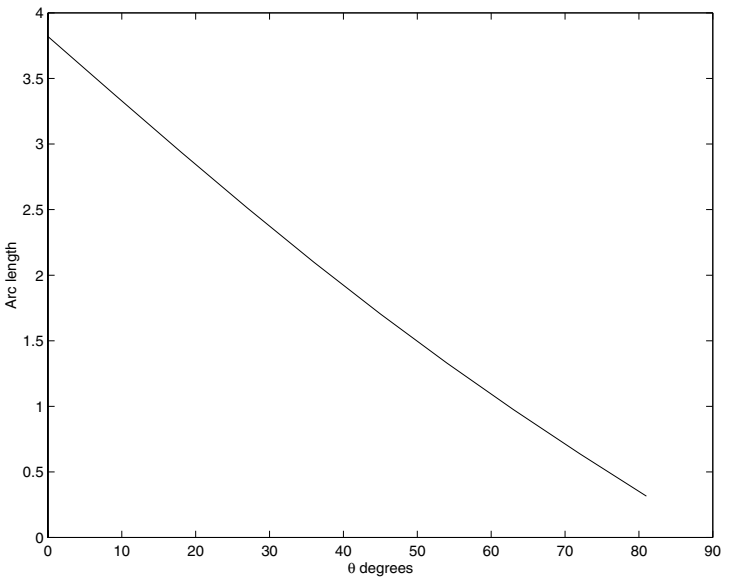
```
>> diverge
Truncate at:10

int =

    2.9981

>> diverge
Truncate at:100

int =

    5.2983

>> diverge
Truncate at:1000

int =

    7.6009
```

which as we see increases as the truncation point increases. (The corresponding values of arcsinh are 2.9982, 5.2983 and 7.6009; so that the integration does a good job).

Solution 7.8 *We can use the code:*

```
theta = 0:pi/20:(pi/2-pi/20);
N = 20;
for it = 1:length(theta);
   theta1 = theta(it);
   clear grid f
   grid = linspace(theta1,pi-theta1,N);
   f = sqrt(1+cos(grid).^2);
   h = grid(2)-grid(1);
   arclen(it) = (sum(f)-f(1)/2-f(N)/2)*h;
end
plot(theta/pi*180,arclen)
xlabel('\theta degrees')
ylabel('Arc length')
```

which gives



Solution 7.9 *Firstly we give details of the analytical solution:*

$$\int_0^{10} \frac{\cos x}{x^{1/2}}\,\mathrm{d}x = \int_0^{\epsilon} \frac{\cos x}{x^{1/2}}\,\mathrm{d}x + \int_{\epsilon}^{10} \frac{\cos x}{x^{1/2}}\,\mathrm{d}x$$

For the first of these integrals we approximate $\cos x$ *by* $1 - x^2/2$ *(that is the*

first two terms in its Taylor series).

$$\int_0^{\epsilon} \frac{1}{x^{1/2}} - \frac{x^{3/2}}{2}\,\mathrm{d}x = \left[2x^{1/2} - \frac{x^{5/2}}{5}\right] = \left[2\epsilon^{1/2} - \frac{\epsilon^{5/2}}{5}\right].$$

We can now use the code

```
clear all
epsil = input('Epsilon :');
int1 = 2*epsil^(0.5)-epsil^(2.5)/5;
N = 100;
x = linspace(epsil,10,N);
h = x(2)-x(1);
f = cos(x)./sqrt(x);
int2 = (sum(f)-f(1)/2-f(N)/2)*h;
int = int1+int2;
```

The first integral gives a significant contribution.

Solution 7.10 *The quadratic through the three points is*

$$y(x) = a_0 + (x - x_0)\Delta a_1 + (x - x_0)(x - x_1)a_2$$

where the constants are

$$\begin{aligned} a_0 &= f_0 \\ a_1 &= \frac{f_1 - f_0}{x_1 - x_0} \\ a_2 &= \frac{(f_2 - f_0)(x_1 - x_0) - (f_1 - f_0)(x_2 - x_0)}{(x_1 - x_0)(x_2 - x_0)(x_2 - x_1)} \end{aligned}$$

Now integrating

$$\int_{x=x_0}^{x_2} y(x)\,\mathrm{d}x \int_{x=x_0}^{x_2} a_0 + (x - x_0)a_1 + (x - x_0)(x - x_1)a_2\,\mathrm{d}x$$

Solution 7.11 *Here we need to use the code*

```
function [f] = fxlnx(x)
f = x.*log(x);
```

and then use the code `quad('fxlnx',1,2)`*. This gives* `0.63629536463993` *(using* `format long`*). The exact value can be calculated using integration by parts*

$$\begin{aligned}\int_{x=1}^{2} q\ln q\,\mathrm{d}q &= \left[\frac{1}{2}q^2\ln q\right]_{x=1}^{2} - \int_{x=1}^{2}\frac{q}{2}\,\mathrm{d}q\\ &= 2\ln 2 - \left[\frac{q^2}{4}\right]_{x=1}^{2}\\ &= 2\ln 2 - \frac{3}{4}.\end{aligned}$$

The value of this expression agrees very well with that above.

C.8 Solutions for Tasks from Chapter 8

Solution 8.1 *Let us firstly find the exact solution. Start by dividing the equation through by y and then integrate with respect to t which gives*

$$\int \frac{1}{y}\frac{\mathrm{d}y}{\mathrm{d}t}\,\mathrm{d}t = -\int \sqrt{t}\,\mathrm{d}t,$$

hence we have

$$\ln y = -\frac{2}{3}t^{3/2} + C.$$

This can be rearranged to give

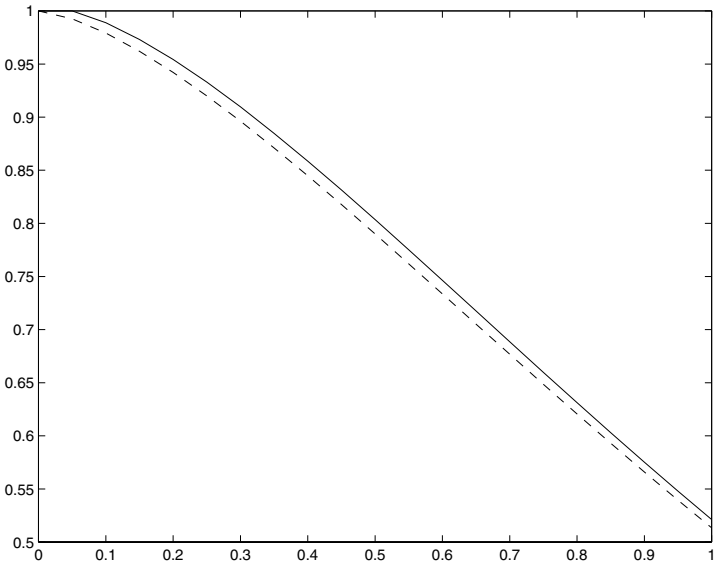
$$y = A\mathrm{e}^{-\frac{2}{3}t^{\frac{3}{2}}}$$

and the particular solution can be found by setting $y(0) = 1$, which gives $A = 1$.

In order to obtain the numerical solution the code should be modified to

```
dt = 0.05;
t = 0.0:dt:1.0;
y = zeros(size(t));
y(1) = 1;
for ii=1:(length(t)-1)
      y(ii+1) = y(ii) + dt * (-y(ii)*sqrt(t(ii)));
end
exact = exp(-2/3*(t).^(3/2));
plot(t,y,t,exact,'--')
```

This produces



Notice that although this solution is "reasonable" it can be improved by reducing the value of `dt`.

Solution 8.2 *Let us start by considering the exact solutions to both differential equations. Again start by dividing through by y and integrating with respect to t, which gives*

$$\int \frac{1}{y}\frac{\mathrm{d}y}{\mathrm{d}t}\,\mathrm{d}t = \int \pm t\,\mathrm{d}t,$$

hence

$$\ln y = \pm\frac{t^2}{2} + C \qquad \Longrightarrow \qquad y = A\mathrm{e}^{\pm t^2/2}.$$

In each calculation the value of the constant is unity, hence we have the solutions

$$y = \mathrm{e}^{t^2/2} \qquad \textit{and} \qquad y = \mathrm{e}^{-t^2/2}.$$

The scheme for the solution of the equations is written as

$$\frac{y_{n+1} - y_n}{\Delta t} = \pm t_n y_n,$$

which can be rearranged to give

$$y_{n+1} = y_n + \pm\Delta t\, t_n y_n = y_n \left(1 + \pm n\Delta t^2\right),$$

where we have used the fact that $t_n = n\Delta t$.

Let us consider the first case. Start with $n = 0$

$$y_1 = 1$$

and now $n = 1$, *etc.*

$$\begin{aligned} y_2 &= 1\left(1 + \frac{1}{16}\right) = \frac{17}{16}, \\ y_3 &= \frac{17}{16}\left(1 + \frac{2}{16}\right) = \frac{153}{128}, \\ y_4 &= \frac{153}{144}\left(1 + \frac{3}{16}\right) = \frac{2907}{2028} \approx 1.419\,. \end{aligned}$$

The exact answer is $\mathrm{e}^{1/2} \approx 1.648$, *hence the absolute error is* $|1.419 - \mathrm{e}^{1/2}| \approx 0.229$ *and the relative error is* $|1.419 - \mathrm{e}^{1/2}|/\mathrm{e}^{1/2} \approx 0.139$ *(or this can be written as* 13.9%*).*

Now we can repeat the calculation for the other case

$$y_1 = 1$$

and now $n = 1$, *etc.*

$$\begin{aligned} y_2 &= 1\left(1 - \frac{1}{16}\right) = \frac{15}{16} \\ y_3 &= \frac{15}{16}\left(1 - \frac{2}{16}\right) = \frac{105}{128} \\ y_4 &= \frac{105}{128}\left(1 - \frac{3}{16}\right) = \frac{1365}{2048}. \end{aligned}$$

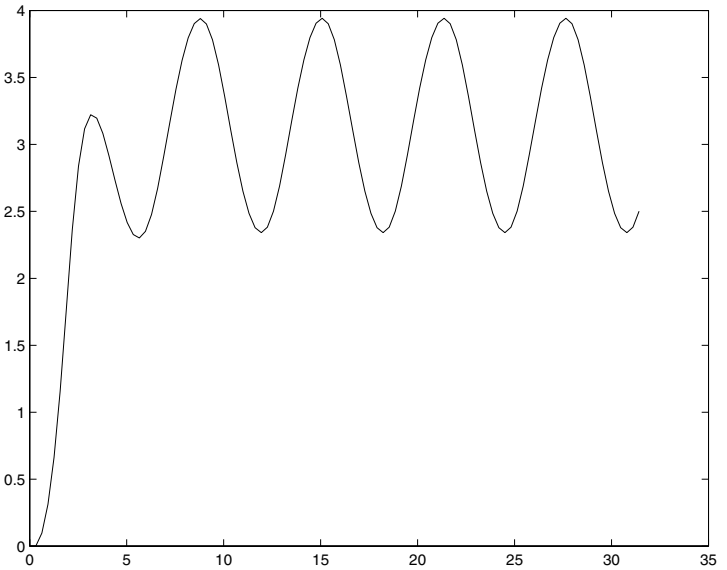
Here the absolute error is ≈ 0.21378 *whereas the relative error is* 0.352 *(or around* 35%*). Despite the absolute errors being comparable the relative errors are different (due to the magnitude of the answers involved).*

It is up to the individual as to which error is best to use and this generally comes with experience.

Solution 8.3 *The code for this task is*

```
dt = pi/10;
t = 0.0:dt:10.0*pi;
y = zeros(size(t));
y(1) = 0;
for ii=1:(length(t)-1)
      y(ii+1) = y(ii) + dt * (sin(t(ii))+sin(y(ii)));
end
```

This produces the result



You can now change the step length, simply by changing the `dt=` *line (it might be a good idea to add a* `clear all` *statement at the top of the code as well).*

Solution 8.4 *First let us construct the exact solution to the equation. We need to multiply through by an integrating factor, namely* $\mathrm{e}^{t/3}$*, which gives*

$$\mathrm{e}^{t/3}\frac{\mathrm{d}y}{\mathrm{d}t} + \frac{1}{3}\mathrm{e}^{t/3}y = -\frac{1}{2}t\mathrm{e}^{t/3}$$
$$\frac{\mathrm{d}}{\mathrm{d}t}\left(y\mathrm{e}^{t/3}\right) = -\frac{1}{2}t\mathrm{e}^{t/3}$$

and now integrating with respect to t *we find that*

$$y\mathrm{e}^{t/3} = A - \int \frac{1}{2}t\mathrm{e}^{t/3}\,\mathrm{d}t.$$

Now integrating the right hand side by parts gives

$$ye^{t/3} = A - \left\{ \left[\frac{3}{2} te^{t/3} \right] - \int \frac{3}{2} e^{t/3} \, dt \right\},$$
$$ye^{t/3} = A - \left\{ \frac{3}{2} te^{t/3} - \frac{9}{2} e^{t/3} \right\}.$$

Hence we have the solution

$$y = Ae^{-t/3} - \left\{ \frac{3}{2} t - \frac{9}{2} \right\}.$$

Now applying the boundary condition gives $A = -9/2$. *The solution is*

$$y = \frac{9}{2} \left(1 - e^{-t/3} \right) - \frac{3t}{2}.$$

Now consider the discretised form of the equation, which is

$$\frac{y_{n+1} - y_n}{\Delta t} = -\frac{t_{n+1}}{2} - \frac{y_{n+1}}{3},$$

which can be rearranged to give

$$y_{n+1} = \frac{1}{1 + \frac{\Delta t}{3}} \left(y_n - \Delta t \frac{t_{n+1}}{2} \right).$$

Now with $\Delta t = 1/3$ *and* $n = 0$ *this gives*

$$y_1 = \frac{1}{1 + \frac{1}{9}} \left(0 - \frac{1}{3} \frac{1}{3 \times 2} \right) = \frac{9}{10} \left(-\frac{1}{18} \right) = -\frac{1}{20},$$

and now with $n = 1$

$$y_2 = \frac{9}{10} \left(-\frac{1}{20} - \frac{1}{3} \frac{2}{3 \times 2} \right) = -\frac{29}{200},$$

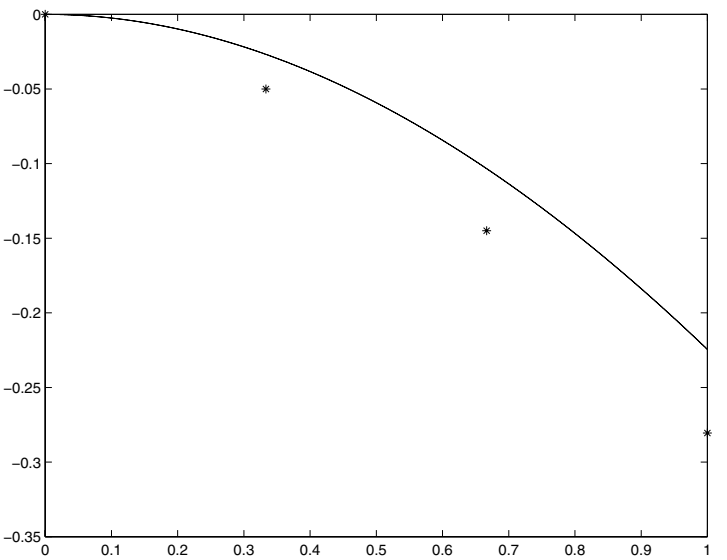
and finally for $n = 2$ *which gives* $y_3 = y(1)$

$$y_3 = \frac{9}{10} \left(-\frac{29}{200} - \frac{1}{3} \frac{3}{3 \times 2} \right) = -\frac{561}{2000}.$$

The code to produce this and the other required solutions is

```
clear all
dt = 1/3;
t = 0.0:dt:1;
y = zeros(size(t));
y(1) = 0;
for ii = 1:(length(t)-1)
      y(ii+1) = 1/(1+dt/3)*(y(ii)-dt*t(ii+1)/2);
end
ts = t; ys = y;
dt = 1/1000;
t = 0.0:dt:1;
y = zeros(size(t));
y(1) = 0;
for ii = 1:(length(t)-1)
      y(ii+1) = 1/(1+dt/3)*(y(ii)-dt*t(ii+1)/2);
end
exact = 9/2*(1-exp(-t/3))-3/2*t;
plot(ts,ys,'*',t,exact,t,y)
```

This gives the picture



where the solution above is shown using stars.

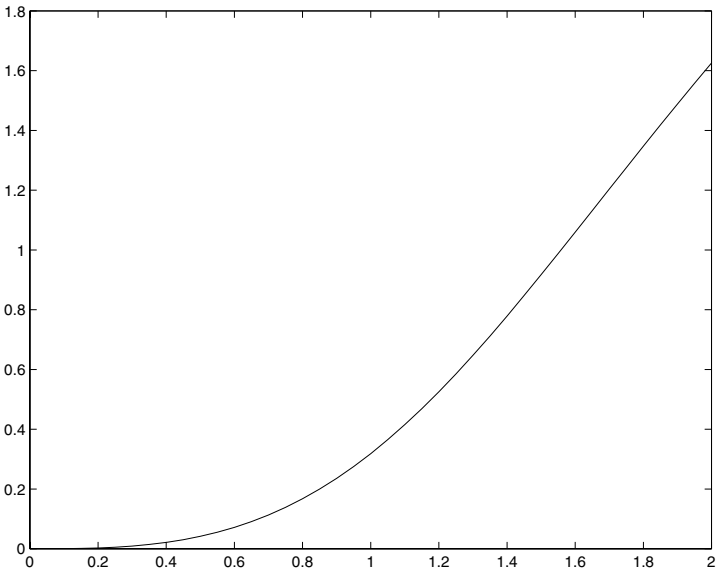
Solution 8.5 *The codes now become*

```
function [value] = odes(t,y)
value = t^2-y^2;
```

and

```
y0 = 0;
tspan = [0 2];
[t,y] = ode45('odes',tspan,y0);
```

These give



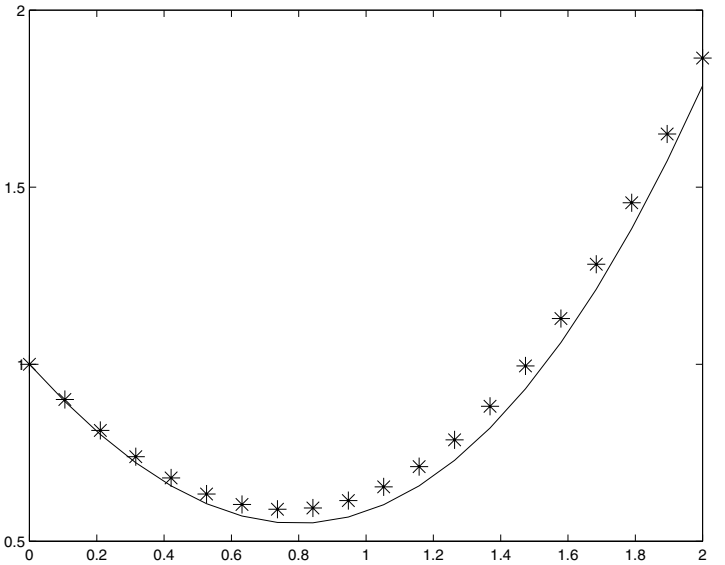
Solution 8.6 *The solution to this equation can be obtained by multiplying by the integrating factor* e^t *and then integrating by parts. After application of the initial condition we find that*

$$y(t) = t^2 - 2t + 2 - \mathrm{e}^{-t}.$$

The numerical solution can be determined using:

```
N = 20;
t = linspace(0,2,N);
dt = t(2)-t(1);
y(1) = 1;
for j = 1:(N-1)
    y(j+1) = y(j)+dt*(-y(j)+t(j)^2);
end
ex = t.^2-2*t+2-exp(-t);
```

This gives:



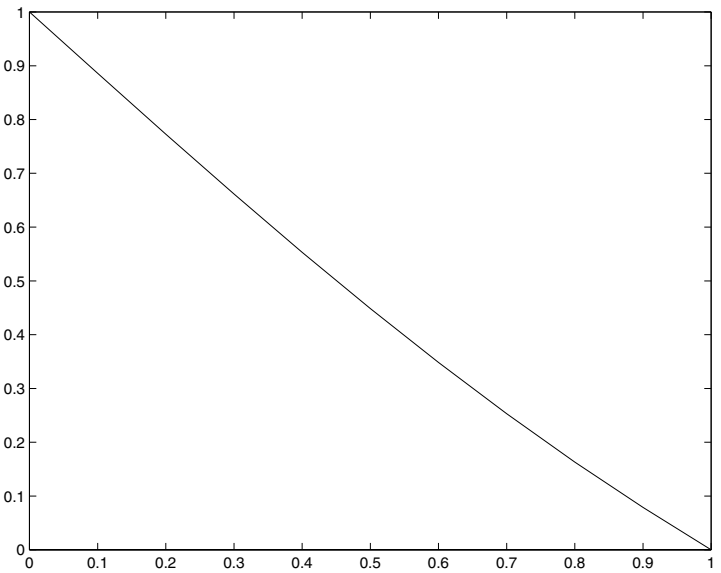
which is a reasonable match (the exact solution is shown with the asterisks).

Solution 8.7 *1. The solution of this problem is:*

$$y(x) = 2\sin x - x\cos x + x(-2\sin 1 + \cos 1 - 1) + 1.$$

```
% Set up system
x = 0.0:0.1:1.0;
N = length(x);
h = x(2)-x(1);
a = 1/h^2*ones(size(x));
b = -2/h^2*ones(size(x));
c = 1/h^2*ones(size(x));
r = x.*cos(x);
a(1) = 0; b(1) = 1; r(1) = 1;
c(N) = 0; b(N) = 1; r(N) = 0;
% Forward sweep
for j = 2:N
    b(j) = b(j)-c(j)*a(j-1)/b(j-1);
    r(j) = r(j)-c(j)*r(j-1)/b(j-1);
end
% Final equation
y(N) = r(N)/b(N);
for j = (N-1):-1:1
    y(j) = r(j)/b(j)-a(j)*y(j+1)/b(j);
end
```

which gives

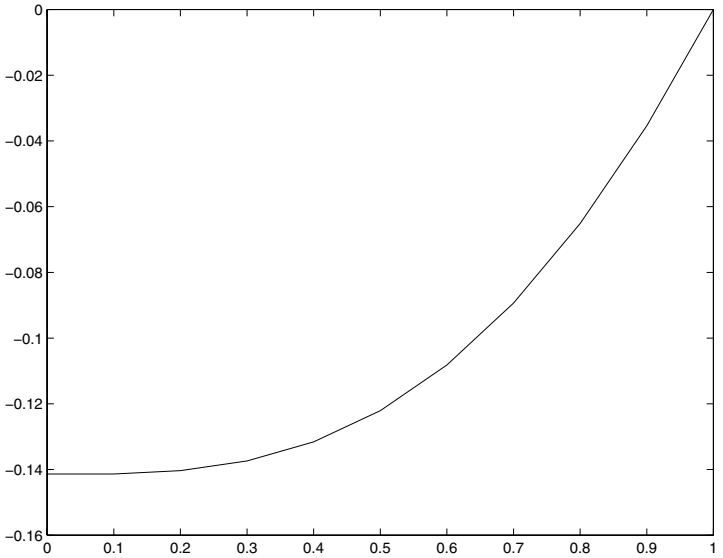


2. The analytic solution here is:

$$y(x) = 2\sin x - x\cos x - x - 2\sin 1 + \cos 1 + 1.$$

```
% Set up system
x = 0.0:0.1:1.0;
N = length(x);
h = x(2)-x(1);
a = 1/h^2*ones(size(x));
b = -2/h^2*ones(size(x));
c = 1/h^2*ones(size(x));
r = x.*cos(x);
a(1) = -1; b(1) = 1; r(1) = 0;
c(N) = 0; b(N) = 1; r(N) = 0;
% Forward sweep
for j = 2:N
    b(j) = b(j)-c(j)*a(j-1)/b(j-1);
    r(j) = r(j)-c(j)*r(j-1)/b(j-1);
end
% Final equation
y(N) = r(N)/b(N);
for j = (N-1):-1:1
    y(j) = r(j)/b(j)-a(j)*y(j+1)/b(j);
end
```

which gives

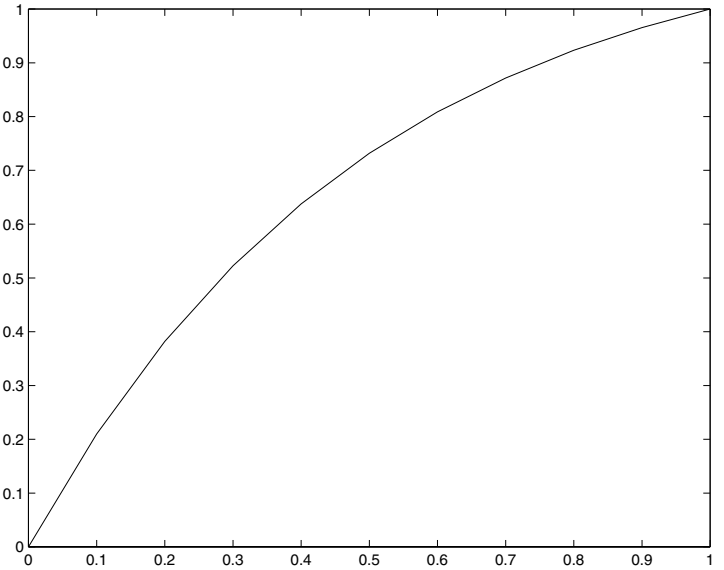


3. The solution here is

$$y(x) = \frac{\mathrm{e}^{-2x} - 1}{\mathrm{e}^{-2} - 1}.$$

```
% Set up system
x = 0.0:0.1:1.0;
N = length(x);
h = x(2)-x(1);
a = (1/h^2+2/(2*h))*ones(size(x));
b = -2/h^2*ones(size(x));
c = (1/h^2-2/(2*h))*ones(size(x));
r = 0;
a(1) = 0; b(1) = 1; r(1) = 0;
c(N) = 0; b(N) = 1; r(N) = 1;
% Forward sweep
for j = 2:N
    b(j) = b(j)-c(j)*a(j-1)/b(j-1);
    r(j) = r(j)-c(j)*r(j-1)/b(j-1);
end
% Final equation
y(N) = r(N)/b(N);
for j = (N-1):-1:1
    y(j) = r(j)/b(j)-a(j)*y(j+1)/b(j);
end
```

which yields



Solution 8.8 *The exact solution here is*

$$y(t) = \frac{1}{3}\left(t - \frac{1}{\sqrt{3}}\sin\sqrt{3}t\right).$$

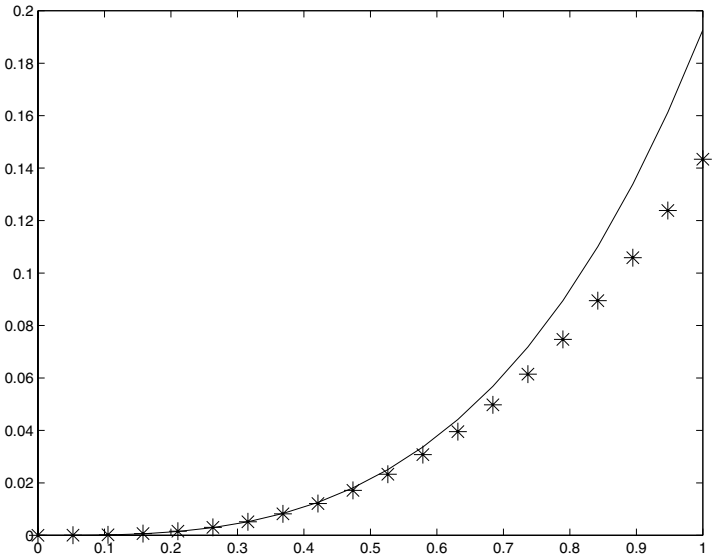
By discretising the equation we have

$$y_{n+1} = 2y_n - y_{n-1} + \Delta t^2\left(-3y_n + t_n\right).$$

The initial conditions can be realised by setting $y_1 = 0$ *and* $y_2 = 0$ *as a result of the fact that* $y'(0) \approx (y_2 - y_1)/\Delta t = 0$.

```
N = 20;
t = linspace(0,1,N);
dt = t(2)-t(1);
y(1) = 0;
y(2) = 0;
for j = 2:N-1
    y(j+1) = 2*y(j)-y(j-1) ...
     +dt^2*(3*y(j)+t(j));
end
ex = (t-sin(sqrt(3)*t)/sqrt(3))/3;
```

This gives

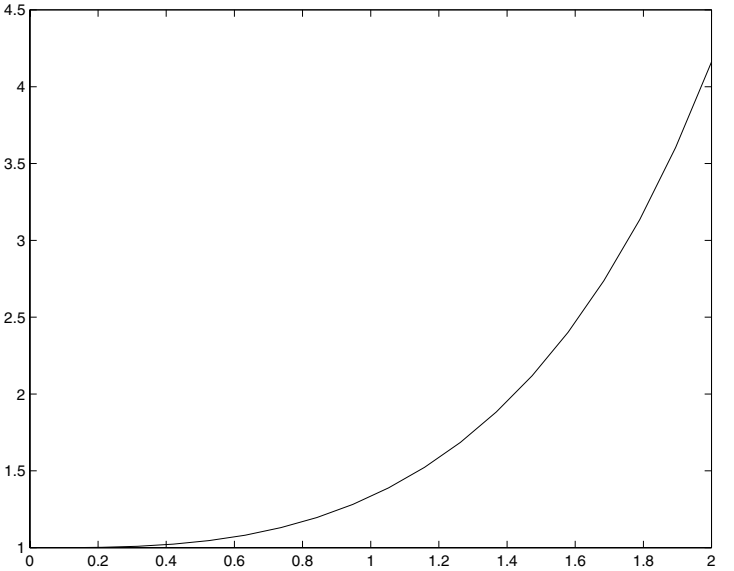


The numerical solution does reasonably initially: however as t increases the solution diverges.

Solution 8.9 *This equation can be solved in a similar manner to the previous task:*

```
N = 20;
t = linspace(0,2,N);
dt = t(2)-t(1);
y(1) = 1;
y(2) = 1;
for j = 2:N-1
    y(j+1) = 2*y(j)-y(j-1) ...
         +dt^2*(t(j)*y(j)+sin(t(j)));
end
```

This gives:



(the solution is expressible in terms of Airy functions, but this does not represent a great advantage to us).

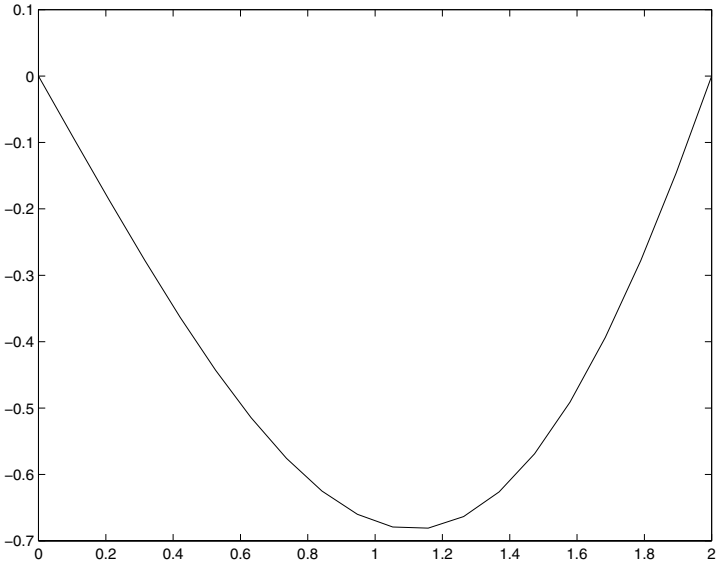
This second problem can be solved by setting up the discretised system

$$\frac{y_{n+1} - 2y_n + y_{n-1}}{\Delta t^2} + t_n y_n = \sin t_n$$

with the conditions that $y_1 = 0$ *(*$y(0) = 0$*)* $y_N = 0$ *(*$y(2) = 0$*).*

```
N = 20;
t = linspace(0,2,N);
dt = t(2)-t(1);
A = zeros(N);
A = diag(-2/dt^2*ones(N,1)+t',0) ...
     +diag(1/dt^2*ones(N-1,1),-1) ...
     +diag(1/dt^2*ones(N-1,1),1);
r = transpose(sin(t));
A(1,:) = 0;
A(1,1) = 1; r(1) = 0;
A(N,:) = 0;
A(N,N) = 1; r(N) = 0;
sol = A\r;
```

This gives:



Solution 8.10 *This equation can be integrated directly by dividing through by* $y(t^2+1)$*. This gives*

$$\frac{y'}{y} = -\frac{2t}{1+t^2}$$

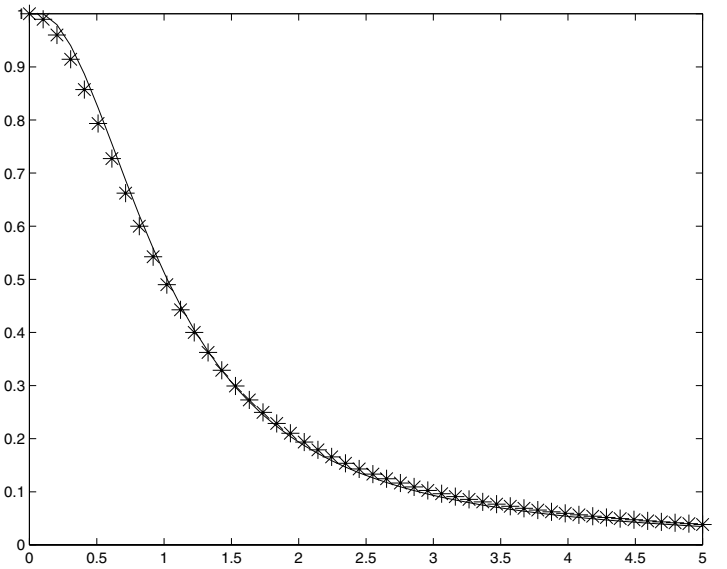
so that

$$y(t) = \frac{1}{1+t^2}.$$

The equation can be solved numerically using

```
N = 50;
t = linspace(0,5,N);
dt = t(2)-t(1);
y(1) = 1;
for j = 1:(N-1)
  y(j+1) = y(j)-dt*2*t(j)*y(j)/(1+t(j)^2);
end
ex = 1./(1+t.^2);
```

This yields



Solution 8.11 *This system can be written as*

$$\mathbf{y}' = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 2 & 1 & -2 \end{pmatrix} \mathbf{y},$$

where $\mathbf{y} = (y, y', y'')$. *Using MATLAB we find that*

```
>> A = [0 1 0; 0 0 1; 2 1 -2];
>> [V,D] = eig(A)

V =

   -0.5774   -0.2182    0.5774
    0.5774    0.4364    0.5774
   -0.5774   -0.8729    0.5774


D =

   -1.0000         0         0
         0   -2.0000         0
         0         0    1.0000
```

And hence we know that

$$\mathrm{e}^{\mathbf{A}t} = \mathbf{V}\mathrm{e}^{\mathbf{D}t}\mathbf{V}^{-1}.$$

For convenience we shall use the non-normalised form of $\mathbf{V}$ *so that*

$$\tilde{\mathbf{V}} = \begin{pmatrix} -1 & 1 & 1 \\ 1 & -2 & 1 \\ -1 & 4 & 1 \end{pmatrix} \text{ and } \tilde{\mathbf{V}}^{-1} = \begin{pmatrix} -1 & \frac{1}{2} & \frac{1}{2} \\ -\frac{1}{3} & 0 & \frac{1}{3} \\ \frac{1}{3} & \frac{1}{2} & \frac{1}{6} \end{pmatrix}$$

Hence

$$\begin{aligned} \mathrm{e}^{\mathbf{A}t} &= \begin{pmatrix} -1 & 1 & 1 \\ 1 & -2 & 1 \\ -1 & 4 & 1 \end{pmatrix} \begin{pmatrix} \mathrm{e}^{-t} & 0 & 0 \\ 0 & \mathrm{e}^{-2t} & 0 \\ 0 & 0 & \mathrm{e}^{t} \end{pmatrix} \begin{pmatrix} -1 & \frac{1}{2} & \frac{1}{2} \\ -\frac{1}{3} & 0 & \frac{1}{3} \\ \frac{1}{3} & \frac{1}{2} & \frac{1}{6} \end{pmatrix} \\ &= \begin{pmatrix} -1 & 1 & 1 \\ 1 & -2 & 1 \\ -1 & 4 & 1 \end{pmatrix} \begin{pmatrix} -\mathrm{e}^{-t} & \frac{1}{2}\mathrm{e}^{-t} & \frac{1}{2}\mathrm{e}^{-t} \\ -\frac{1}{3}\mathrm{e}^{-2t} & 0 & \frac{1}{3}\mathrm{e}^{-2t} \\ \frac{1}{3}\mathrm{e}^{t} & \frac{1}{2}\mathrm{e}t & \frac{1}{6}\mathrm{e}^{t} \end{pmatrix} \\ &= \begin{pmatrix} \mathrm{e}^{-t} - \frac{1}{3}\mathrm{e}^{-2t} + \frac{1}{3}\mathrm{e}^{t} & -\frac{1}{2}\mathrm{e}^{-t} + \frac{1}{2}\mathrm{e}^{t} & -\frac{1}{2}\mathrm{e}^{-t} + \frac{1}{3}\mathrm{e}^{-2t} + \frac{1}{6}\mathrm{e}^{t} \\ -\mathrm{e}^{-t} + \frac{2}{3}\mathrm{e}^{-2t} + \frac{1}{3}\mathrm{e}^{t} & \frac{1}{2}\mathrm{e}^{-t} + \frac{1}{2}\mathrm{e}^{t} & \frac{1}{2}\mathrm{e}^{-t} - \frac{2}{3}\mathrm{e}^{-2t} + \frac{1}{6}\mathrm{e}^{t} \\ \mathrm{e}^{-t} - \frac{4}{3}\mathrm{e}^{-2t} + \frac{1}{3}\mathrm{e}^{t} & -\frac{1}{2}\mathrm{e}^{-t} + \frac{1}{2}\mathrm{e}^{t} & -\frac{1}{2}\mathrm{e}^{-t} + \frac{4}{3}\mathrm{e}^{-2t} + \frac{1}{6}\mathrm{e}^{t} \end{pmatrix}. \end{aligned}$$

The solution is obtained by multiplying $(0, 0, 1)^T$ *(the initial conditions) by this matrix*

$$\mathrm{e}^{\mathbf{A}t} \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} -\frac{1}{2}\mathrm{e}^{-t} + \frac{1}{3}\mathrm{e}^{-2t} + \frac{1}{6}\mathrm{e}^{t} \\ \frac{1}{2}\mathrm{e}^{-t} - \frac{2}{3}\mathrm{e}^{-2t} + \frac{1}{6}\mathrm{e}^{t} \\ -\frac{1}{2}\mathrm{e}^{-t} + \frac{4}{3}\mathrm{e}^{-2t} + \frac{1}{6}\mathrm{e}^{t} \end{pmatrix}.$$

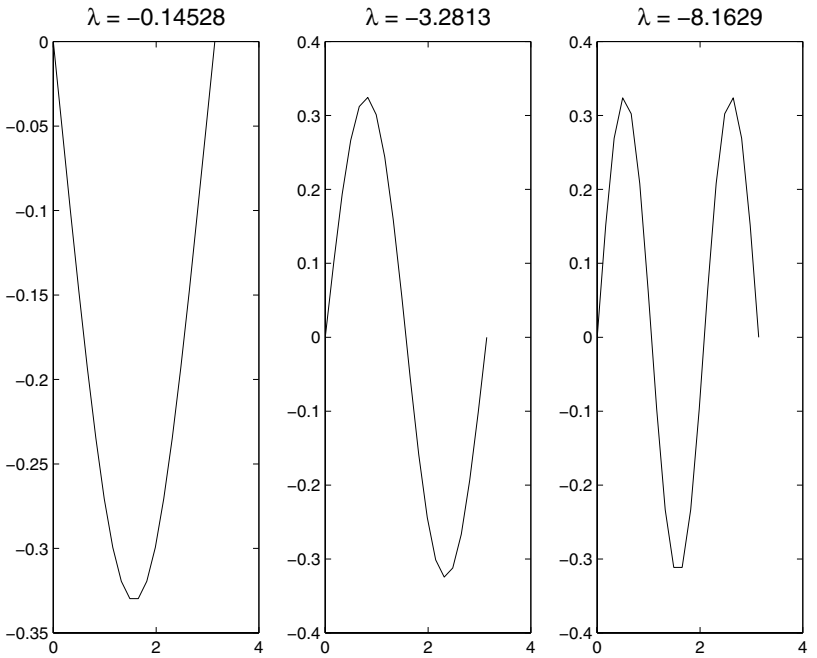
Finally we have the answer

$$y(t) = -\frac{1}{2}\mathrm{e}^{-t} + \frac{1}{3}\mathrm{e}^{-2t} + \frac{1}{6}\mathrm{e}^{t}.$$

Solution 8.12 *This is solved using the finite difference code:*

```
N = 20;
x = linspace(0,pi,N);
h = x(2)-x(1);
% Only the internal points
A = diag(-2/h^2*ones(N-2,1) ...
   +sin(x(2:(N-1)))',0) ...
   +diag(1/h^2*ones(N-3,1),1) ...
   +diag(1/h^2*ones(N-3,1),-1);
[V,D] = eigs(A,3,'SM');
for j = 1:3
    ti = ['\lambda = ' num2str(D(j,j))];
    subplot(1,3,j)
    plot(x,[0; V(:,j); 0])
    title(ti,'FontSize',14)
end
```

This creates



C.9 Solutions for Tasks from Chapter 9

Solution 9.1 *This can be achieved using*

```
x = [3 2 4 5 6 -1 5 6 7 8 2];
y = [2 -6 3 2 0 1 4 5 6 7 8];
mean(x)
mean(y)
median(x)
median(y)
var(x)
var(y)
A = cov(x,y);
A(1,2)
A = corrcoef(x,y);
A(1,2)
```

This gives

```
ans =

    4.2727


ans =

    2.9091


ans =

     5


ans =

     3


ans =

    6.8182


ans =

   15.0909


ans =

    4.2273


ans =

    0.4167
```

respectively.

Solution 9.2 *We note that if $Y = aX + b$ then $\bar{y} = a\bar{x} + b$ which can be seen by substitution into the formula for the mean. Similarly substituting this into the expression for the correlation shows that*

$$\sigma_{XY} = \frac{a}{\sqrt{a^2}} = \frac{a}{|a|} = sign(a).$$

Solution 9.3 *These two can be calculated using*

```
xb=mean(x);
vx=sum((x-xb).^2)/length(x);
std_x=sqrt(vx);
skew=sum(((x-xb)/std_x).^3)/length(x);
kurt=sum(((x-xb)/std_x).^4)/length(x)-3;
```

Solution 9.4 *In order to solve this problem we shall use a basic loop structure*

```
global mu
mu = 0.4;
x(1) = 0.25;
for i = 1:9
     x(i+1) = map(x(i));
end
```

The value this code returns is `4.3420e-05` *(notice if you get MATLAB to write out* `x` *it will appear that this entry is zero, depending on the current format).*

Solution 9.5 *One value which works is $\mu = -2$. Then use*

```
mu = -2;
co = [-mu^3 2*mu^3 -mu^2*(1+mu) (mu^2-1) 0];
[r] = roots(co);
x1 = r(3);
x2 = map(x1);
x3 = map(x2);
disp([x1 x2 x3])
```

Note that `x2` *is also a period 2 point and its image is* `x1`*. These values are actually given by*

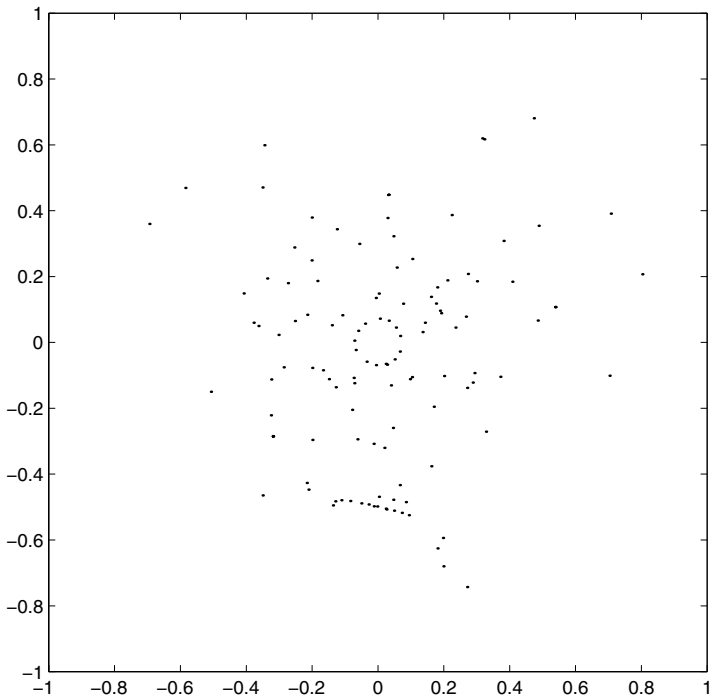
$$\frac{1}{2}\frac{\mu^2 + \mu \pm \sqrt{\mu^4 - 2\mu^3 - 3\mu^2}}{\mu^2}$$

The real values of these roots occur for $\mu < -1$ *and* $\mu > 3$.

Solution 9.6 *The code for this map is*

```
function [xn,yn] = map3(xo,yo)
xn = mod(xo+2*yo,1);
yn = mod(3*xo-2*yo,1);
```

Solution 9.7 *The solution of the Hénon map for* $\cos\alpha = 0.24$ *gives*



Index