Chapter 2

Creating and manipulating arrays

The real power of MATLAB arises from its ability to handle arrays: vectors and matrices. These are fundamental to MATLAB and even if you are not intending to use MATLAB explicitly for this purpose, you will need to be familiar with the generation and manipulation of arrays. We start here with a discussion of vectors; matrices will be introduced later.

2.1 Vectors

2.1.1 Basic concepts

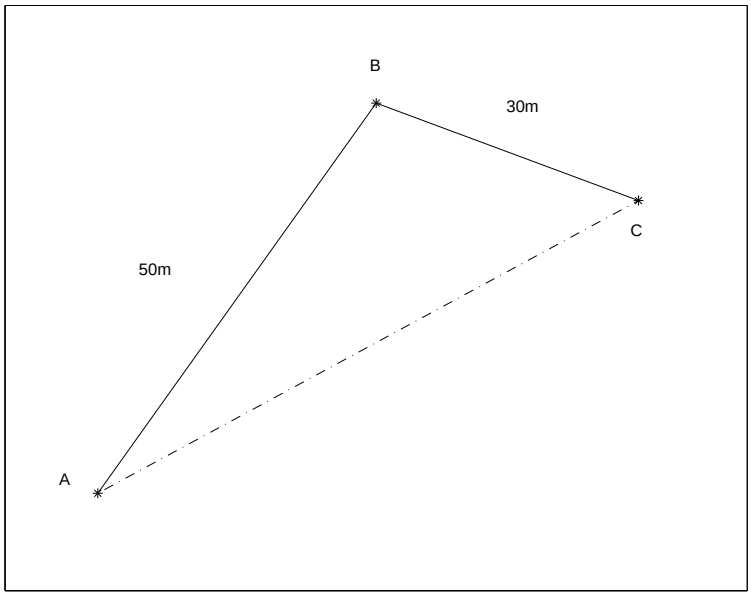
Suppose we walk directly from a point A to a point B, 50m from A, and subsequently to a point C, 30m from B. It is apparent that we have walked 80m altogether, but unless the points A, B and C were all on a straight line, the points A and C will not be 80m apart:

However, it is true to say that whether we walk from point A to point B and then from point B to point C or directly from point A to point C, we will end up at the same point. That is to say, the position of C relative to A is not altered by the route taken. Mathematically this is written as

$$\overrightarrow{AB} + \overrightarrow{BC} = \overrightarrow{AC}$$

where $\overrightarrow{AB}$ indicates that both the length and direction of AB are being considered.

We refer to such quantities involving both magnitude and direction as *vectors*. Those involving just magnitude we call *scalars*. Therefore $\overrightarrow{AB}$ is a vector whose length is equal to that of the line AB and whose direction is that from A to B. The vector $\overrightarrow{BA}$ would have the same length as $\overrightarrow{AB}$ but be in the opposite



direction. The modulus of the vector $\overrightarrow{AB}$ is its magnitude or length, thus

$$|\overrightarrow{AB}| = 50m.$$

2.1.2 Position vectors

In general a vector has no specific location in space. However, there are some quantities that compound in the same way as vectors. For example, the position of any point in a plane can be stated in terms of an ordered pair (x, y), the coordinates of the point. This ordered pair gives the distance from the point to each of the coordinate axes. The point P in the following diagram has coordinates $(2, 3)$ with respect to the origin 0.

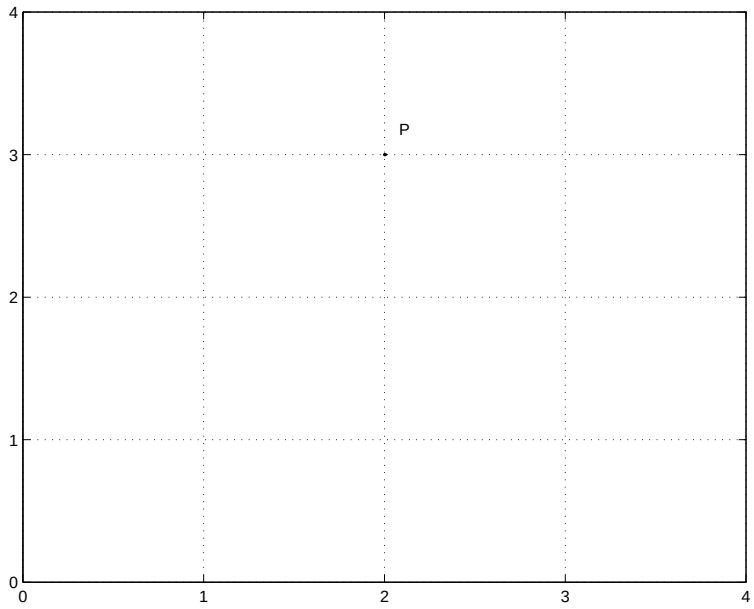
We could also define the point P by giving the distance and direction of P from the origin, by stating the vector $\overrightarrow{0P}$. This vector is called the *position vector* of P. In addition to this, it can also be expressed in terms of its horizontal and vertical *components*:

$$\begin{pmatrix} 2 \\ 3 \end{pmatrix}$$

or

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

This is the type of format that MATLAB employs when dealing with vectors and they are known *column* vector and *row* vectors respectively. MATLAB distiguishes



between these different types, by use of a comma (for separates row entries) or a semicolon (for separate column entries):

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

We can perform simple mathematical operations on vectors, for example addition of two vectors of the same type, and multiplication of a vector by a scalar:

```
>> a+a
ans =
     2     4     6
>> 2*b
ans =
     6
     8
     10
```

2.1.3 The scalar product

The *scalar product* of two vectors $\overrightarrow{a}$ and $\overrightarrow{b}$ is defined to be the product of the magnitude of each vector, multiplied by the cosine of the angle between the two vectors:

$$\overrightarrow{a} \cdot \overrightarrow{b} = |\overrightarrow{a}||\overrightarrow{b}| \cos\theta$$

Since all of the quantites involved, $|\overrightarrow{a}|$, $|\overrightarrow{b}|$ and $\cos\theta$ are scalars, their product will also be a scalar; hence the term scalar product. This product is also known as the *dot product* of two vectors. MATLAB uses the command `dot` for this purpose:

```
>> dot(a,b)
ans =
     26
```

Interestingly, this is equivalent to multiplying the corresponding elements of each vector together and summing the total:

```
>> a*b
ans =
     26
```

Note however, that multiplication of two vectors like this, requires the vectors to be in the correct form. Were we to take the transpose (using ') of the column vector (to give the equivalent row vector) and try multiplying:

```
>> c=b'
c =
     3     4     5
>> a*c
Error using ==> *
Inner matrix dimensions must agree.
```

Note that `dot` is not fussy about the type of vectors involved as long as they are of the same size.

From the definition of the scalar product, it is apparent that two vectors are perpendicular if their scalar product is 0.

```
>> d=[1,1,1];
>> e=[2,-4,2];
>> dot(d,e)
ans =
     0
```

2.1.4 The vector product

The *vector* or *cross* product of two vectors $\overrightarrow{a}$ and $\overrightarrow{b}$ is defined to be a vector of magnitude $|\overrightarrow{a}||\overrightarrow{b}|\sin\theta$, in a direction perpendicular to the plane containing $\overrightarrow{a}$ and $\overrightarrow{b}$ in the sense of a right-handed screw turned from $\overrightarrow{a}$ to $\overrightarrow{b}$.

It is apparent from this definition that

$$\overrightarrow{a} \times \overrightarrow{b} = -\overrightarrow{b} \times \overrightarrow{a}.$$

MATLAB uses the command `cross`, for the cross product:

```
>> c=cross(a,b)
c =
    -2     4    -2
>> dot(a,c)
ans =
     0
```

Why is this necessarily zero?

2.1.5 Other vector commands

MATLAB has many other commands for the mainpulation of vectors. Sometimes, the situation may arise where you wish to take element by element multiplication of two vectors. The command `.*` surfices for this purpose:

```
>> d=a.*c
d =
     3     8     15
```

MATLAB uses certain functions according to the circumstance within which it finds them, for example `sqrt`, for calculating the squareroot of a number or an array:

```
>> sqrt(4)
ans =
     2
>> d=sqrt(a)
d =
    1.0000    1.4142    1.7321
```

Notice that the `sqrt` command is performed on each element of the vector a, without having to specify this to MATLAB. This is because there is no ambiguity using this command.
There are other ways in which MATLAB can create vectors, without us having to specify each element directly. For example:

```
>> a=1:1:10
a =
     1     2     3     4     5     6     7     8     9    10
```

The first number is the first entry of the vector, the second the increment for each entry. The final value is the last value of the vector (or as close as possible to that: try `>> a=1:2:10`).

We can also raise a vector to a power, for example:

```
>> b=a.^2
b =
     1     4     9    16    25    36    49    64    81   100
>> b=a^2
Error using ==> ^
Matrix must be square.
```

Notice that the exponentiation function has to be told to perform the operation on each element of the vector. There are other ways in which this type of vector creation may be utilized:

```
>> a=1:0.25:4
a =
  Columns 1 through 7

    1.0000    1.2500    1.5000    1.7500    2.0000    2.2500    2.5000

  Columns 8 through 13

    2.7500    3.0000    3.2500    3.5000    3.7500    4.0000
>> a=1:-2:-21
a =
     1    -1    -3    -5    -7    -9   -11   -13   -15   -17   -19   -21
>> a=1:-1:2
a =
   Empty matrix: 1-by-0
```

A similar command called `linspace` may also be used to generate row vectors. The usage is slightly different from before; the first value being the initial entry of the vector, the second being the last value, and the third being the number of rows required. As the name suggests, all elements are linearly spaced from each other:

```
>> a=linspace(0,1,11)
a =
  Columns 1 through 7

         0    0.1000    0.2000    0.3000    0.4000    0.5000    0.6000

  Columns 8 through 11

    0.7000    0.8000    0.9000    1.0000
```

Remember if you are in any doubt about any function, the `help` command can assist you:

```
>> help linspace
 LINSPACE Linearly spaced vector.
```

```
LINSPACE(x1, x2) generates a row vector of 100 linearly
equally spaced points between x1 and x2.

LINSPACE(x1, x2, N) generates N points between x1 and x2.

See also LOGSPACE, :.
```

2.2 Matrices

2.2.1 Basic concepts

Matrices are rectangular array of numbers,

$$\begin{pmatrix} 1 & -3 & 4 \\ 2 & 4 & -1 \end{pmatrix}$$

Matrices have a wide variety of applications in problems involving linear relationships between any number of variables.

The size of a matrix is described by the number of rows and columns that it possesses. In the example above, the matrix has 2 rows and three columns and so is described as a 2 by 3 matrix. A more general matrix with m rows and n columns is described as an m by n matrix. You might notice that a matrix with only one row, or only one column is a row or column vector as introduced in the previous section. A *square* matrix, is one in which the number of rows and columns is the same. One particularly special square matrix is called the *identity* matrix, which has a one for each diagonal entry and zeros elsewhere:

$$\begin{pmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \ddots & 0 \\ \vdots & \ddots & 1 & 0 \\ 0 & \cdots & 0 & 1 \end{pmatrix}$$

It is called the identity matrix, because it serves as the identity for multiplication.

Two matrices **A** and **B** are said to be the same size if the number of rows and columns of each is the same. They are said to be equal if each idividual entry is the same.

As with vectors, MATLAB can generate matrices in a number of ways. We may define them manually, by making use of `,` and `;`. For example:

```
>> A=[1,2;5,3]
A =
    1   2
    5   3
```

```
>> B=[3,7;-2,1]
B =
    3   7
    -2  1
```

MATLAB also has a number of standard commands that can generate matrices automatically; for example a matrix full of zeros, full of ones or the identity. These are generated by the commands `zeros`, `ones`, `eye`. For example:

```
>> zeros(3)
ans =
     0    0    0
     0    0    0
     0    0    0
>> ones(2,3)
ans =
     1    1    1
     1    1    1
>> eye(2)
ans =
     1    0
     0    1
```

Notice that `ones(m,n)` produces a matrix with m rows and n columns, whereas a square matrix requires a one number input only. A number of commands exist for determining the dimensions of a matrix. `size(A)` gives two values; namely the number of rows and columns of A, whereas `length(A)` returns the largest of the two values.

```
>> C=ones(2,3);
>> size(C)
ans=
    2  3
>> length(C)
ans=
    3
```

2.2.2 Addition of matrices

If two matrices are of the same size they can be added by summing the corresponding entries:

```
>> A+B
ans =
      4   9
      3   4
```

Note that if the two matrices are not of the same size, addition has no meaning:

```
>> A+zeros(3)
??? Error using ==> +
Matrix dimensions must agree
```

If matrix addition is well defined that it is commutative as with addition of scalars.

2.2.3 Multiplication of matrices

There are two differents sorts of matrix multiplication. The first is multiplying a matrix by a scalar. This is well defined for any scalar and any matrix. For example:

```
>> 2*ones(2,3)
ans =
      2   2   2
      2   2   2
```

It is also possible to multiply two matrices together. However, for this to be well defined, the number of columns of the first matrix must equal the number of rows of the second, *ie.* If $\mathbf{A}$ is an m by n matrix, then $\mathbf{B}$ must be an n by k matrix for $\mathbf{A} \times \mathbf{B}$ to make sense.

```
>> A*ones(2,3)
ans =
     3   3   3
     8   8   8
```

Note that the product of $\mathbf{A}$ and $\mathbf{B}$ will have m rows and k columns.

2.2.4 Matrices of random numbers

Other function for creating matrices are `rand` and `randn`. These generate matrices of pseudo-random numbers. Those generated by the function `rand` being uniformly distributed over the interval $[0, 1]$ and those by `randn` being normally distributed with mean 0 and variance 1. By definition neither process produces genuine random numbers. Instead they use an algorithm whereby an initial number (called a *seed*) is used to generate all the subsequent numbers. MATLAB has a built in initial seed which, if left unchanged, means that the same numbers are produced each time we run MATLAB. The `seed` function enables us to change this:

```
>> randn('seed',222)
>> randn(2)
ans =
  0.4856 -1.7185
  0.1423  1.2027
>> rand(3,2)
```

```
ans =
  0.9501  0.4860
  0.2311  0.8913
  0.6068  0.7621
```

2.2.5 Other useful commands

As well as generating a matrix manually:

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

we may also generate a matrix by combining two or more vectors together:

```
>> a=1:1:5;
>> b=2:2:10;
>> A=[a;b]
A =
     1     2     3     4     5
     2     4     6     8    10
```

MATLAB can also consider a specific part of a previously entered matrix, for example the first two rows and columns:

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

as well as changing a specific entry:

```
>> A(2,2)=7
A =
     1     2
     2     7
```

Another useful command is `inv`, used to find the inverse of a square matrix A. In order for the inverse to be well defined the *determinant* of the matrix must be non-zero. You will find out about *determinants* in the worksheet.

```
>> inv(A)
ans =
  2.3333 -0.6667
 -0.6667  0.3333
```

There are many other commands available for manipulating matrices and vectors. We shall see some of these as the course continues. Don't forget to make use of the *helpdesk* facility, which can point you to many of MATLAB's inbuilt functions. Also the library and bookshop will have many copies of the book MATLAB *Guide* by Desmond and Nicholas Higham, which provides a useful point of reference for this course.

In the next lecture, we will examine some of the programming capabilities of MATLAB as well as creating some of our own functions called *m-files*.