"If there is a God, he's a great mathematician."
— Paul Dirac

CHAPTER 2

Arrays, Strings and Data Structures

2.1 WHAT IS A MATRIX OR AN ARRAY?

With the invention of computers the use of matrices and mathematics involving matrices has become very important. Data can be stored in a matrix or an array in a very organized manner. Data stored in a matrix can be processed simultaneously. Each location of data has a unique address. A chess board is an example of an array.

Each cell in the chessboard has a unique address.

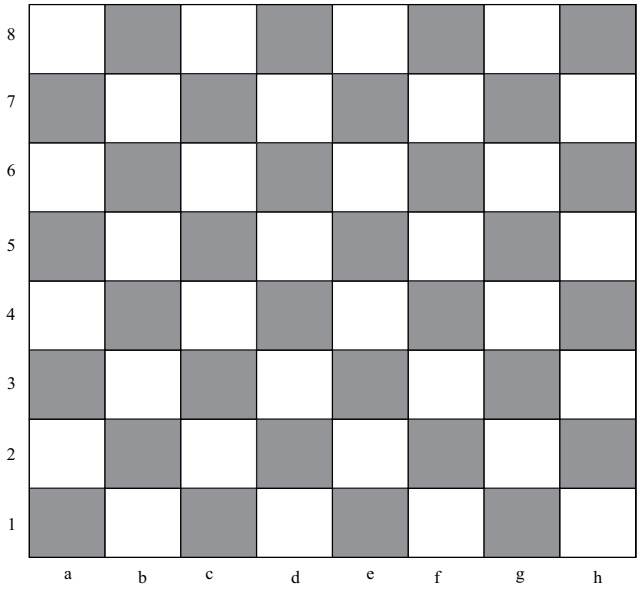

Figure 2.1 A chessboard

would generate a column vector of 100 points whose first point will be 1 and last point will be 4. In fact, it will give the transpose of the array generated by the command

```
x = linspace(1,4)
```

2.2.2 The logspace Function

The logspace function is a similar function for generating a logarithmically spaced vector. It is very similar to the linspcae function used for generating a vector in linear space. It could be considered a corresponding function of linspace in logarithmic space. logspace(a,b,n) generates n values including 10^a and 10^b that are equally spaced in a logarithmic scale.

```
logspace(0,2,10)
ans =
Columns 1 through 8
   1.0000 1.6681 2.7826 4.6416 7.7426 12.9155 21.5443 35.9381
Columns 9 through 10
   59.9484 100.0000
```

We can also assign a name to the vector to be generated.

```
t=logspace(1,2,10)
t =
   Columns 1 through 7
   10.0000 12.9155 16.6810 21.5443 27.8256 35.9381 46.4159
Columns 8 through 10
   59.9484 77.4264 100.0000
```

By default if we do not mention the number of elements to be generated as in the case logspace(0,2), 50 elements in the logarithmically spaced vector will be created. In linspace 100 elements would have been created if the number of elements to be generated is not mentioned.

Here another example has been considered where we have created an array of 10 points in a logarithmic scale. The points as we can see are equally space along the y-axis.

```
y=logspace(0,2,10)
y =
   Columns 1 through 8
1.0000 1.6681 2.7826 4.6416 7.7426 12.9155 21.5443 35.9381
Columns 9 through 10
   59.9484 100.0000
loglog(y,'ro')
```

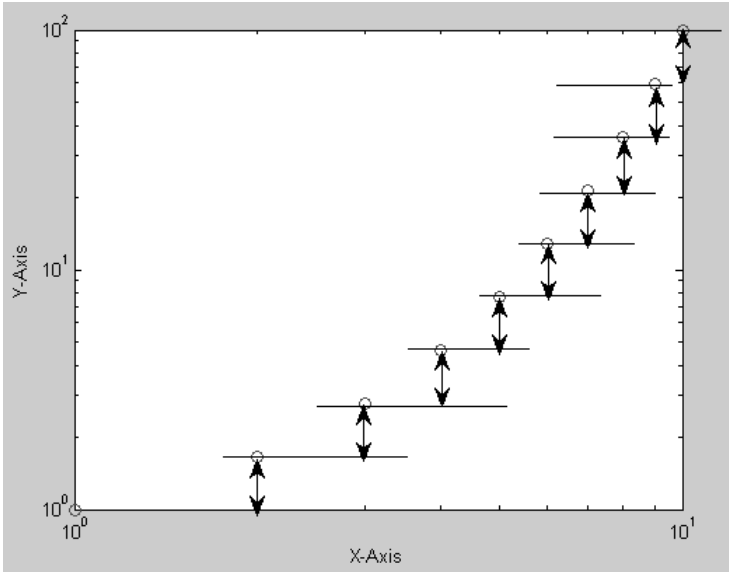

Figure 2.3 Equal spacing of points in a logarithmic scale

```
xlabel('X-Axis')
ylabel('Y-Axis')
```

2.2.3 Using ':' to Create an Array

An array of numbers can also be generated using the '**:**' operator. For example, a command of the type x=1:1:4 will generate an array of numbers starting at 1 and ending at 4 and the numbers being spaced equally by 1. Here the format for creating the array is

initial number : increment : final number or a : i : b

Here we should note that the initial number is the first number of the array that is created. The next number is created by adding i to the initial number and the subsequent numbers are obtained by adding i to the previous number. It may so happen that the final number is not possible to obtain by adding i to the previous number, in that case we have to end the array at a number before we reach the final number. The final number cannot be exceeded. Using a, i and n we can write the series as,

a, a+i, a+2i, a+3i, ……a+ni

where n=fix((b-a)/i).
fix is a MATLAB function which rounds n to the nearest integer towards zero. For example, let us create an array 1:2:12. This is an array starting with 1 and increment 2 but the last number should not exceed 12. So we can find n,

```
n=fix((12-1)/2)

n =
```

```
   5
x=1:2:12
x =
   1 3 5 7 9 11
```

We find that n=5, which means the last number of the array is 1+(5x2)=11.
Let us consider a few other examples,

```
x=1:1:4
x =
   1 2 3 4
```

Another example using this method to create an array is shown below.

```
x=1:4:10
x =
   1 5 9
```

Here also we can use the transpose to generate a vertical array.

```
x=(1:4:10)'
x =
   1
   5
   9
```

A command x=0:4 will generate an array,

```
x=0:4
x =
   0 1 2 3 4
```

A command x = 0:1:4 will also generate an array

```
x = 0:1:4
x =
   0 1 2 3 4
```

Which means that the array starts at 0 ends at 4 and the spacing between points is 1.

x' would lead to a column matrix,

```
x'
```

```
ans =

     0
     1
     2
     3
     4
```

2.3 THE zeros FUNCTION

The function zeros(n) will create an n by n matrix of zeros whereas zeros(m,n) will create an m by n matrix of zeros. A few examples are listed below.

```
zeros

ans =

     0

zeros(3)

ans =

     0     0     0

     0     0     0

     0     0     0

x=zeros(3,4)

x =

     0     0     0     0
     0     0     0     0
     0     0     0     0
```

2.4 THE ones FUNCTION

The function ones(n) will return an n by n matrix of ones. The function ones(m,n) will return an m by n matrix of ones.

```
ones

ans =

     1

ones(3)

ans =
```

```
   1     1     1
   1     1     1
   1     1     1
```

ones(3,4)

ans =

```
   1     1     1     1
   1     1     1     1
   1     1     1     1
```

One can also multiply a constant to this matrix by giving a command,

4*ones(3,4)

ans =

```
   4     4     4     4
   4     4     4     4
   4     4     4     4
```

2.5 THE eye FUNCTION

The eye function returns an identity function. A command link A=eye(n) will generate an n by n matrix having all the elements of the diagonal as 1 and all other elements as 0.

For example, the command eye(3) results in a matrix,

eye(3)

ans =

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

A similar result is obtained when the command eye(3,3) is given.

eye(3,3)

ans =

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

2.6 THE diag FUNCTION

By using the **diag** function, we can find out the elements in the diagonal of a matrix. The comamand X=diag(A,n) returns the elements in the n^{th} diagonal elements of the matrix A in a column vector X. The figure below illustrates how the diagonals of a matrix are numbered.

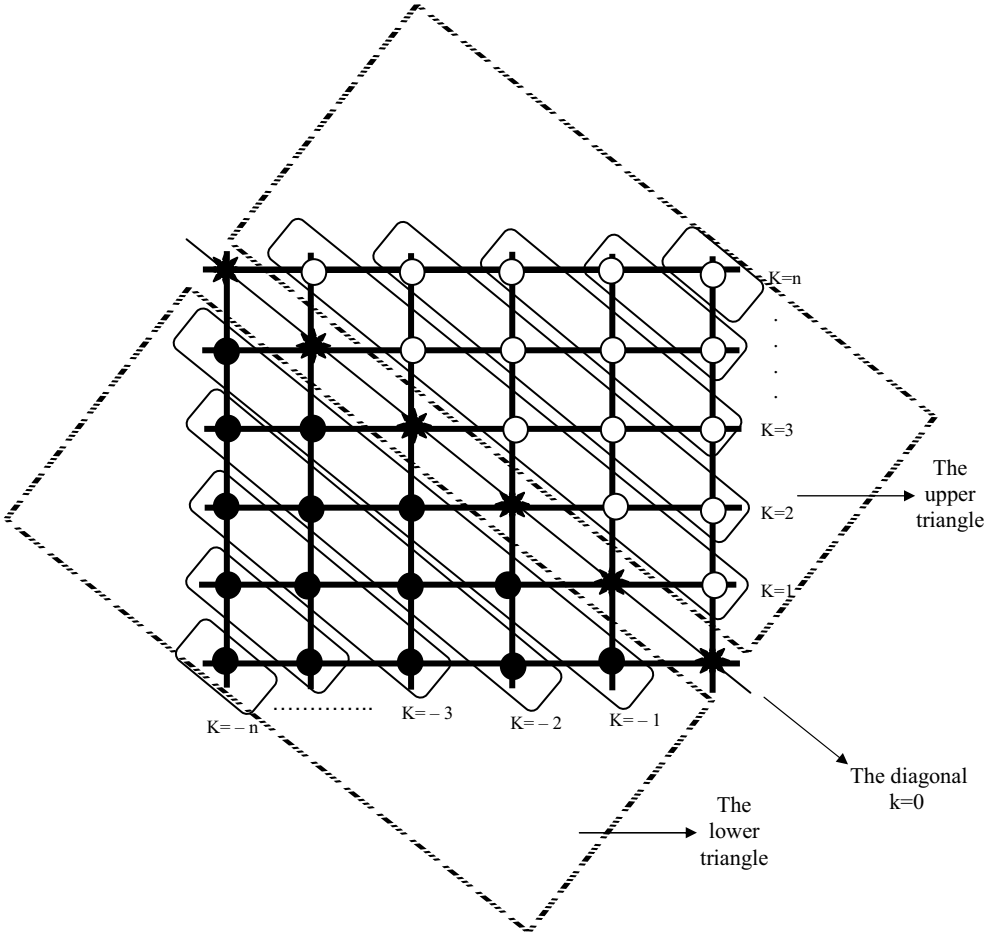

Figure 2.4 Numbering of the diagonals of a matrix

```
x =

   1     2     3
   3     4     5
   5     6     7

diag(x) =

   1
   4
   7

A=[ 1 2 3 4; 5 6 7 8; 9 10 11 12;13 14 15 16]

A =
```

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

```
X=diag(A)

X =

   1
   6
  11
  16
```

diag(A) or diag(A,0) returns the same result.

```
X=diag(A,0)

X =

   1
   6
  11
  16
```

```
X=diag(A,1)

X =

   2
   7
  12
```

```
X=diag(A,2)

X =

   3
   8
```

```
X=diag(A,3)

X =

   4
```

For values of n less than 0, we can get the diagonals that belong to the lower triangle. This has been illustrated below.

```
X=diag(A,–1)

X =
```

```
   5
   10
   15
X=diag(A,–2)
X =
   9
   14
X=diag(A,–3)
X =
   13
```

2.7 THE rand FUNCTION

A command **rand(n,m)** will generate a matrix of n by m with random values drawn from the open interval (0,1). rand(n) will generate an n by n matrix having random values.

```
rand(2) =
   0.1897 0.6822
   0.1934 0.3028
rand (2,2)
ans =
   0.5417 0.6979
   0.1509 0.3784
```

It should be noted here that if a command like zeros(2), ones(2) or rand(2) is used then MATLAB will generate a 2×2 matrix with all the elements of the matrix as zeros, ones or random values respectively.

The rand function can be used for designing games which involve guess work. One such simple game has been shown below in which a couple of random values are generated which are less than one, if the sum is greater than one you win if less than one you lose.

```
k=rand(1,2)
if (k(1)+k(2))>=1
  disp('You Have Won!')
else
  disp('Sorry!Try Again.')
end
```

A few attempts gave mixed results. The program has been saved as tryagain.

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

```
size(A)
```

```
ans =
   4     4
```

```
[i,j]=size(A)
```

```
i =
   4
   j =
   4
```

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

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

```
size(B)
```

```
ans =
   3     4
```

Even a single integer is considered to be an element of an array. For example,

```
C=2
```

```
C =
   2
```

```
size(C)
```

```
ans =
   1     1
```

Any number on its own is the first row and first column element of an array having only one element. This is the same with a complex number.

```
D=3+2i
```

```
D =
   3.0000 + 2.0000i
```

```
size(D)

ans =

   1     1
```

2.9 THE NUMBER OF NON-ZERO ELEMENTS IN A MATRIX

The function nnz gives the number of non-zero elements in a matrix. For example,

```
A=[ 7 0 3 0; 5 0 7 18; 9 0 10 12;13 0 15 16]

A =

   7     0     3     0
   5     0     7    18
   9     0    10    12
  13     0    15    16

c=nnz(A)

c =

  11
```

This tells us that there are 11 non-zero elements in the matrix A. A few other examples using the function nnz are as follows.

```
a=0

a =

   0

nnz(a)

ans =

   0
```

Using nnz for a single complex number, we get

```
D=3+2i

D =

   3.0000 + 2.0000i

nnz(D)

ans =

   1
```

We should note that a complex number will be considered zero if both the real and imaginary parts are zero.

```
D=[3+2i 1]

D =

   3.0000 + 2.0000i 1.0000

nnz(D)

ans =

   2

D=[3+2i 5i]

D =

   3.0000 + 2.0000i 0 + 5.0000i

nnz(D)

ans =

   2

D=[3+2i 0]

D =

   3.0000 + 2.0000i 0

nnz(D)

ans =

   1
```

2.10 SELECTING SPECIFIC ROWS AND COLUMNS OF AN ARRAY

Using MATLAB it is possible to select elements of specific rows and columns of an array. Here both the start and end of the rows and start and end of the columns can be specified to select the elements. A few examples are given below.

```
A=[10 12 4 6 15;12 14 18 9 10;16 12 11 10 7;12 5 7 8 10;12 11 15 16 9]

A =

   10    12     4     6    15
   12    14    18     9    10
   16    12    11    10     7
   12     5     7     8    10
   12    11    15    16     9
```

Here the rows and columns of the above matrix have been shown:

Columns →					Rows ↓
1	2	3	4	5	
10	12	4	6	15	1
12	14	18	9	10	2
16	12	11	10	7	2
12	5	7	8	10	4
12	11	15	16	9	5

The command A(1,5) selects a particular element of the array and in this case it is the first row fifth column element, which is 15.

Suppose we select a section of the array A(3:5,3:5). This suggests that we want to select rows 3 to 5 and columns 3 to 5 and so the selected part of the array is

```
A(3:5,3:5)

ans =

   11    10     7
    7     8    10
   15    16     9
```

If the selection is of the type A(:,3) it means that all rows and 3rd column are selected. So the selected part of the array is

```
A(:,3)

ans =

    4
   18
   11
    7
   15
```

If the selection is of the type A(3,:) it means that the 3rd row and all the columns are selected. So the selected part of the array is,

```
A(3,:)

ans =

   16    12    11    10     7

A(3:5,:)

ans =
```

```
   16    12    11    10     7
   12     5     7     8    10
   12    11    15    16     9
```

```
A(:,3:5)

ans =

    4     6    15
   18     9    10
   11    10     7
    7     8    10
   15    16     9
```

2.11 CREATING A 3-DIMENSIONAL ARRAY

A 3-dimensional array has several layers of arrays of the same m × n dimension stacked one above the other. As shown below it consists of 2-dimensional arrays stacked one after another. Here the 2-dimensional arrays which are spread in the x-y plane are stacked in the z-direction.

```
c(:,:,1)=[4 6;2 8];
c(:,:,2)=[5 7;9 6];
c(:,:,3)=[1 3;5 9];
c

c(:,:,1) =

    4     6
    2     8

c(:,:,2) =

    5     7
    9     6

c(:,:,3) =
    1     3
    5     9
```

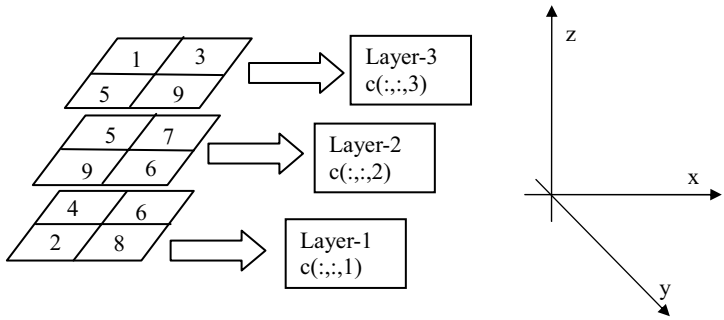

Figure 2.5 Layers in a 3-dimensional array

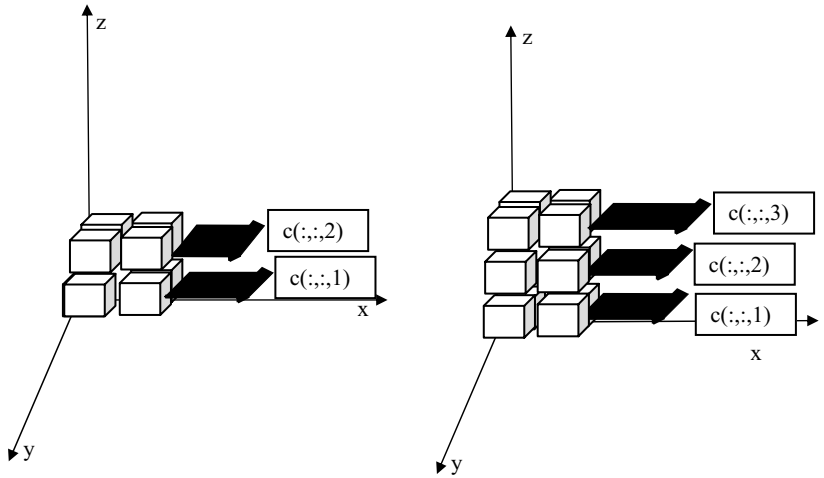

Figure 2.6 Layers in a 3-dimensional array

The command c tells us c(:,:1) , c(:,:,2) and c(:,:,3). c(:,:1) , c(:,:,2) and c(:,:,3) are actually the elements in the various x-y layers that are stacked one above the other in the z direction.

Another example is as follows.

```
c(:,:,1)=[4 4;4 4];
c(:,:,2)=[4 4;4 4];
c(:,:,3)=[4 4;4 4];
c

c(:,:,1) =

   4     4
   4     4

c(:,:,2) =

   4     4
   4     4

c(:,:,3) =

   4     4
   4     4
```

There are also various commands for finding out the size of the matrix.

[l,m,n]=size() gives the exact dimension of the matrix where l is the number of rows, m is the number of columns and n is the number of layers in the z direction.

```
b(:,:,1)=[2 4 9;5 7 11];
b(:,:,2)=[7 2 5;6 7 15];
```

```
b(:,:,3)=[9 12 15;3 7 10];
[l,m,n]=size(b)

l =

   2

m =

   3

n =

   3
```

b is a 2 by 3 by 3 matrix. Another example illustrating the use of the command [l,m,n]=size() is given below.

```
c(:,:,1)=[2 4;5 7];
c(:,:,2)=[1 5;3 9];
c(:,:,3)=[8 2;9 1];
[l,m,n]=size(c)

l =

   2

m =

   2

n =

   3
```

c is a 2 by 2 by 3 matrix. It is also possible to find the size of matrix using size() function. In the examples given above, we used [l,m,n]=size() command but by using just the size command it is also possible to find the size of the matrix. For example,

```
c(:,:,1)=[1 5 9;2 6 8];
c(:,:,2)=[3 7 1;2 5 9];
c(:,:,3)=[4 6 1;3 6 1];
size(c)

ans =

   2     3     3
```

Using the command size(c), we can find the dimension of the matrix c. Here in this case c is a 2 by 3 by 3 matrix. The function ndims also gives us the dimension of the matrix but it does not tell us anything about the number of rows or columns or layers in the matrix. It only tells us the dimension of the matrix. For example,

```
c(:,:,1)=[1 5 8; 9 2 6];
c(:,:,2)=[8 2 7; 5 4 1];
c(:,:,3)=[9 5 3; 2 7 6];
ndims(c)

ans =

   3
```

This tells us that the matrix c is 3-dimensional.

```
   b=[2 5;6 9];
   ndims(b)

ans =

   2
```

This suggests that b is a 2-dimensional matrix.

```
a=2

a =

   2

ndims(a)

ans =

   2
```

The constant a is also a dimensional matrix. It has one row and one column.

2.12 MATRIX MATHEMATICS

2.12.1 A*B and A.*B

Let's now do a few mathematical operations with matrices. Here rand function has been used to generate a 2 × 2 array.

```
A=rand(2)

A =

   0.9501    0.6068
   0.2311    0.4860

B=rand(2)

B =

   0.8913    0.4565
   0.7621    0.0185
```

```
a =

   10     5     3
    4    15    20
    7    16     9

b=inv(a)

b =
    0.1273   –0.0021   –0.0379
   –0.0716   –0.0475    0.1294
    0.0282    0.0860   –0.0895

a*b

ans =

    1.0000         0         0
    0.0000    1.0000         0
    0.0000         0    1.0000
```

So, we find that a*inv(a) is an identity matrix whose diagonal elements are 1 and all other elements are 0.

2.12.3 The trace Function

The trace function can be used to get the sum of the diagonal elements of a matrix.

For example,

```
a=[1 2 3 4;5 6 7 8;9 10 11 12;13 14 15 16]

a =

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

trace(a)

ans =

   34
```

It should be noted that it is possible to find the trace of only square matrices.

2.12.4 The fix Function

A fix command can be used to round off to the closest integer towards 0. Let us use the fix command for an array [4.7 3.4 4.55].

```
fix([4.7 3.4 4.55])
```

```
ans =
   4     3     4
```

When the fix command is used for the matrix having a few negative numbers then in this case also the numbers are rounded off towards 0. This has been shown in the example given below.

```
fix([–4.7 3.4 –4.55])
ans =
  –4     3    –4
```

The fix can also be used for complex numbers, where the real and the imaginary parts of the complex number are rounded off independently. A few examples, using complex numbers are given below.

```
fix(3.2+1.1i)
ans =
   3.0000 + 1.0000i
fix(0.7+0.9i)
ans =
   0
```

2.12.5 The floor Function

The floor command is used to round off to the closest integer towards $-\infty$ as a result the number is rounded off to the closest integer less than or equal to the number. Examples using the floor command are given below.

```
floor([–4.7 3.4 –4.55])
ans =
  –5     3    –5
```

The floor command used on the matrix [4.7 3.4 4.55] will result in the same result as in the case of fix command.

```
floor([4.7 3.4 4.55])
ans =
   4     3     4
ans =
   4     3     4
```

The floor command can also be used for complex numbers where both the real and the imaginary parts are rounded off independently. Examples of the floor command using complex numbers are given below.

```
floor(3.11+2.1i)

ans =

   3.0000 + 2.0000i

floor(–3.11–2.1i)

ans =

   –4.0000 – 3.0000i

floor(0.55+0.77i)

ans =

   0

floor(0.55–1.2i)

ans =

   0 – 2.0000i
```

2.12.6 The ceil Function

The command ceil is used to round off to the closest integer towards $+\infty$. ceil rounds off a number to the closest integer higher than or equal to the number.

So a command like ceil([4.7 3.4 4.55]) would result in,

```
ans =

   5     4     5
```

When the array is [-4.7 3.4 -4.55]

and the ceil command is operated on it,

```
ceil([-4.7 3.4 -4.55])
```

Then the answer is

```
ans =

   –4     4     –4
```

If the ceil command is used for complex numbers then the real and complex parts are rounded off independently. Examples of ceil command using complex numbers are given below.

```
ceil(3.11+2.1i)

ans =
```

```
   4.0000 + 3.0000i
ceil(0.55+0.77i)
ans =
   1.0000 + 1.0000i
```

2.12.7 The tril and triu Function

Using MATLAB it is possible to find out the lower and upper triangular parts of a rectangular matrix. Let us consider a matrix and try to find out the lower and upper triangular parts of the matrix. The function triu(A) returns the upper triangular part of the matrix A, whereas triu(A,n) will return the elements above the n^{th} diagonal of the matrix A. The function tril(A) returns the lower triangular part of the matrix A, whereas tril(A,n) will return the elements below the n^{th} diagonal of the matrix A.

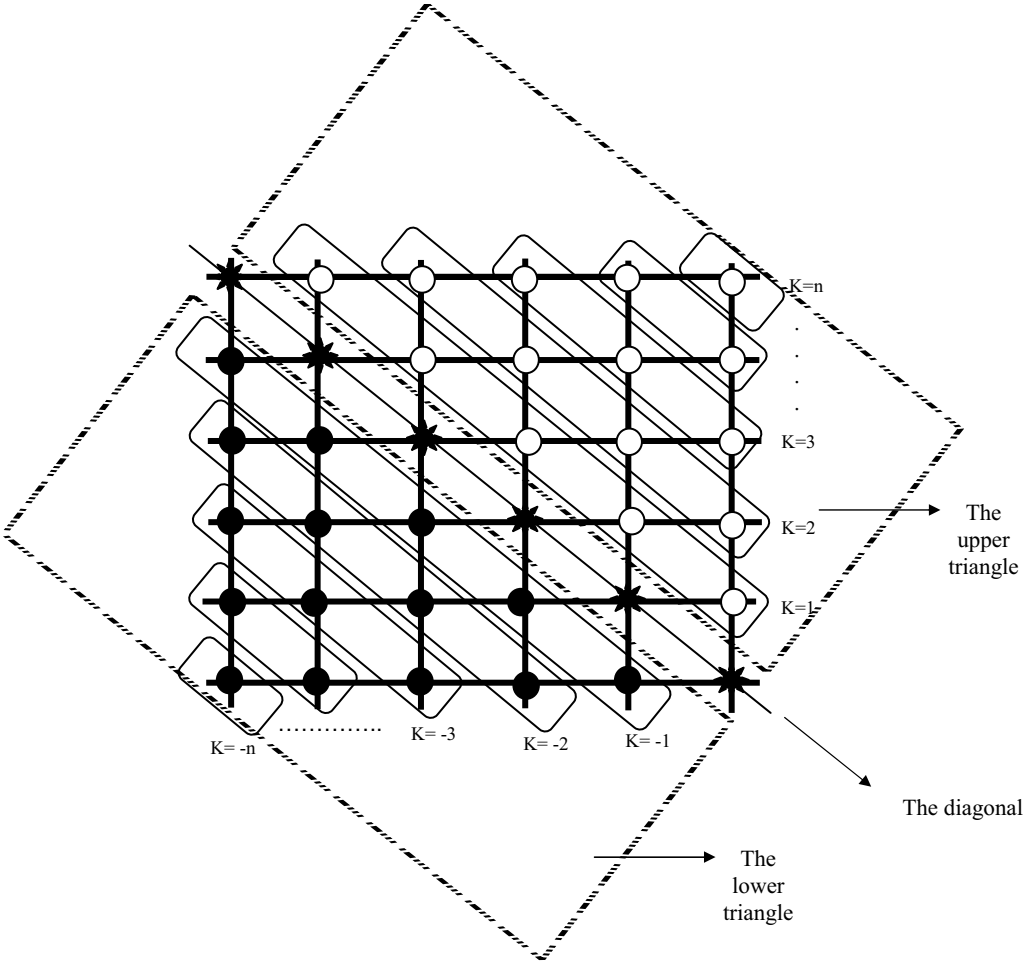

Figure 2.7 The upper and lower triangles in an array

```
tril(A,–3)

ans =

   0   0   0   0
   0   0   0   0
   0   0   0   0
  13   0   0   0
```

Diagonal n=0
Diagonal n=–1
Diagonal n=–2
Diagonal n=–3 is included as part of the lower triangle

```
triu(A,1)

ans =

   0   2   3   4
   0   0   7   8
   0   0   0  12
   0   0   0   0
```

Diagonal n=1 is included as part of the upper triangle

```
triu(A,-3)

ans =

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

triu(A,–2)
```

Diagonal n=–3 is included as part of the upper triangle

```
ans =

   1   2   3   4
   5   6   7   8
   9  10  11  12
   0  14  15  16
```

Diagonal n=–2 is included as part of the upper triangle

2.12.8 The sort Function

The function sort is used to sort the elements of an array. Sorting can be done both in ascending and descending order.

```
A=[2 4 1 5 3]

A =

   2   4   1   5   3
```

This is done for (n–1) elements of the array as the last element cannot be compared with the next. After the (n–1) elements have been sorted the whole process is started once again from the beginning. In other words again A(i) is compared to A(i+1) and if A(i)>A(i+1) the two are swapped. Here this is done again for all the (n–1) elements. This whole process is repeated until a sorted array is achieved. The program below gives a clear picture of the whole process. Here, the user is asked if the array is sorted or not. In case the array is not sorted then the whole process is repeated again.

```
n=input('enter the value of n:')
for i=1:n
  a(i)=input('enter value')
end
i=1;
for j=1:n
for i=1:(n-1)
  if a(i)>a(i+1)
       temp=a(i);
       a(i)=a(i+1);
       a(i+1)=temp;
  end
end
end
disp('The sorted array is:')
a
```

```
m=input('enter the value of m=1 if OK if NOT OK enter m=0:')
for m=0
for j=1:n
for i=1:(n-1)
  if a(i)>a(i+1)
       temp=a(i);
       a(i)=a(i+1);
       a(i+1)=temp;
  end
end
end
end
disp('The sorted array is:')
a
```

```
enter the value of n:8

n =

  8

enter value1
```

```
a =
    1
enter value5
a =
    1     5
enter value4
a =
    1     5     4
enter value8
a =
    1     5     4     8
enter value5
a =
    1     5     4     8     5
enter value3
a =
    1     5     4     8     5     3
enter value9
a =
    1     5     4     8     5     3     9
enter value0
a =
    1     5     4     8     5     3     9     0
The sorted array is:
a =
    0     1     3     4     5     5     8     9
enter the value of m=1 if OK if NOT OK enter m=0:1
m =
    1
```

The sorted array is:

```
a =
   0     1     3     4     5     5     8     9
```

2.13 THE magic FUNCTION

The function magic(n) creates an array of n × n elements whose numbers are from 1 and n^2 and the sum of the elements in a row is equal to the sum of the elements in a column. Here n is a scalar that can take values greater than or equal to 3. It can be noted that the sum of the main diagonal elements is also the same as the sum of the elements in a row and the sum of the elements is a column.

For example, choosing n=3, we get

```
a=magic(3)

a =

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

All the numbers are between 1 and 3^2=9 and the sum of the elements in any row is 15 which is also the sum of the elements in a column. Note that the sum of the main diagonal elements is also 15.

```
a=magic(3)

a =

   8     1     6
   3     5     7
   4     9     2

trace(a)

ans =

   15
```

A few more examples of magic squares are,

```
c=magic(4)

c =

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

sum(c)
```

```
ans =

   34    34    34    34

sum(c')

ans =

   34    34    34    34
```

Let us now find the sum of the elements in the main diagonal,

```
c=magic(4)

c =

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

trace(c)

ans =

   34
```

The sum of the elements in the main diagonal is also 34. The 9 by 9 magic square also gives us the same result.

```
c=magic(9)

c =

   47    58    69    80    1     12    23    34    45
   57    68    79    9     11    22    33    44    46
   67    78    8     10    21    32    43    54    56
   77    7     18    20    31    42    53    55    66
   6     17    19    30    41    52    63    65    76
   16    27    29    40    51    62    64    75    5
   26    28    39    50    61    72    74    4     15
   36    38    49    60    71    73    3     14    25
   37    48    59    70    81    2     13    24    35

sum(c)

ans =

   369   369   369   369   369   369   369   369   369

sum(c')

ans =
```

```
   369   369   369   369   369   369   369   369   369
```

This truly looks like magic as forming such an array having row sum and column sum equal is not very easy. n should be greater than or equal to three.

```
c=magic(9)

c =

   47   58   69   80    1   12   23   34   45
   57   68   79    9   11   22   33   44   46
   67   78    8   10   21   32   43   54   56
   77    7   18   20   31   42   53   55   66
    6   17   19   30   41   52   63   65   76
   16   27   29   40   51   62   64   75    5
   26   28   39   50   61   72   74    4   15
   36   38   49   60   71   73    3   14   25
   37   48   59   70   81    2   13   24   35

trace(c)

ans =

   369
```

The magic squares are indeed marvellous! This is indeed magic with numbers.

2.14 THE reshape FUNCTION

It is possible to reshape a matrix having m rows and n columns keeping $m \times n = p \times q$, where p and q are the number of rows and columns in the new matrix. Here the product of rows and columns in the new matrix is maintained the same as $m \times n$. Here in the example given below a 4×4 matrix is being changed into a 8×2 matrix. The elements are taken column-wise from the given matrix and the reshaped matrix is formed by filling the positions column-wise. As seen in the example below this is done by adding the columns of matrix a one below the other. Here the columns of the new matrix are filled one after another taking the elements of the existing matrix column-wise and then appending one below the other. In the new matrix the first column is filled first followed by the second column, and so on.

```
a=[1 2 3 4;5 6 7 8;9 10 11 12;13 14 15 16]

a =

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

reshape(a,8,2)
```

```
ans =

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

A few other examples are listed below.

```
b=[2 9 5 7;9 10 11 12;1 3 7 9;3 7 9 8]

b =

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

reshape(b,8,2)

ans =

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

reshape(b,4,4)

ans =

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

reshape(b,1,16)

ans =

Columns 1 through 14

   2   9   1   3   9  10   3   7   5  11   7   9   7  12
```

Columns 15 through 16

9 8

As can be seen the elements are taken column-wise from the given matrix and the positions in the reshaped matrix are also filled column-wise.

2.15 THE rot90 FUNCTION

The rot90 function rotates a matrix by 90^o in the counterclockwise direction whereas a command like rot90(A,n) will rotate the matrix, A, in the counterclockwise direction by an angle n x 90^o.

Counterclockwise rotation of the matrix.

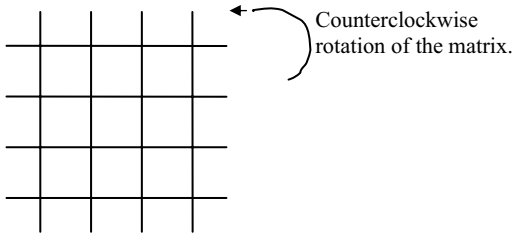

Figure 2.8 Counterclockwise rotation of a matrix.

A=[1 2 3 4;5 6 7 8;9 10 11 12;13 14 15 16;]

A =

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

rot90(A,1) → Rotates matrix A by 1 × 90^o in the counterclockwise direction.

ans =

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

rot90(A,2) → Rotates matrix A by 2 × 90^o in the counterclockwise direction.

ans =

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

rot90(A,–1) → Rotates matrix A by -1 × 90° or 90° in the clockwise direction.

```
ans =

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

rot90(A,–2)

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

2.16 THE fliplr FUNCTION

The fliplr function flips a matrix about a vertical axis. It is like flipping a matrix by placing a mirror in the middle. If there are odd number of columns, then the mirror sits on the middle column and in the case of even number of columns the mirror sits between the middle two rows.

A=[1 2 3;4 5 6;7 8 9;10 11 12]

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

fliplr(A)

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

A=[1 2 3 13;4 5 6 14;7 8 9 15;10 11 12 16]

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

```
B=[4;5;6]

B =

   4
   5
   6

A\B

ans =

   0.0469
   0.6875
   0.1719

inv(A)*B

ans =

   0.0469
   0.6875
   0.1719

A=[1 5 3;4 5 8;7 8 1]

A =

   1     5     3
   4     5     8
   7     8     1

B=[4;5;6]

B =

   4
   5
   6

mldivide(A,B)

ans =

   0.0469
   0.6875
   0.1719
```

So we find that A, B, inv(A)*B and mldivide(A,B) all give the same result.

The function mrdivide(A,B) is the same as A/B and A*inv(B). Here A and B must have the same number of columns.

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

```
A =

   1     5     3
   4     5     8
   7     8     1

B=[1 4 5; 7 9 10;4 8 10]

B =

   1     4     5
   7     9     10
   4     8     10

A/B

ans =

   8.0000    2.6000   -6.3000
  –5.0000   –1.4000    4.7000
  15.0000    7.2000  –14.6000

mrdivide(A,B)

ans =

   8.0000    2.6000   -6.3000
  -5.0000   –1.4000    4.7000
  15.0000    7.2000  –14.6000

A*inv(B)

ans =

   8.0000    2.6000   -6.3000
  –5.0000   -1.4000    4.7000
  15.0000    7.2000   -14.6000
```

So, we find that mrdivide(A,B), A/B and A*inv(B) all give the same result.

2.18 INTEGRATION AND DIFFERENTIATION OF ELEMENTS IN AN ARRAY

It is possible to integrate and differentiate all the elements of an array simultaneously.

```
syms x
a=[x x^2 x^3 x^4];
diff(a)

ans =

   [ 1, 2*x, 3*x^2, 4*x^3]
```

```
int(a)

ans =

   [ 1/2*x^2, 1/3*x^3, 1/4*x^4, 1/5*x^5]
```

2.19 EIGENVALUE AND EIGENVECTOR

Let A be a complex square matrix. Then if ë is a complex number and x a non-zero complex column vector satisfying the equation, Ax = ëx, we call the non-zero vector x the eigenvector of A and ë the eigenvalue of the square matrix A. We can also say that x is an eigenvector corresponding to the eigenvalue ë. The eigenvectors (x) of a square matrix (A) are the non-zero vectors that, after being multiplied by the scalar eigenvalue matrix (l) is proportional to the original vector. The direction of the eigenvector does not change only its magnitude changes by the factor given by the eigenvalue. It should be noted that the eigenvector is a non-zero vector. MATLAB has a built-in function discussed below to find the eigenvector and the eigenvalues of a square matrix A.

Let us try to find the eigenvector and the eigenvalues of a square matrix A.

```
A=[2 3 4;5 7 8;9 10 12]

A =

   2     3     4
   5     7     8
   9    10    12

[x,l]=eig(A)

x =

–0.2491    0.6862   –0.2425
–0.5349    0.2568    0.8191
–0.8073   –0.6806   –0.5199

l =

  21.4030    0         0
   0        –0.8452    0
   0         0         0.4422
```

Here x are the eigenvectors and l the eigenvalues of the matrix A.

```
A=[2 3 4;5 7 8;9 10 12]

A =

   2     3     4
   5     7     8
   9    10    12

eig(A)
```

```
ans =

   21.4030
   –0.8452
    0.4422

[eigenvectors,eigenvalues]=eig(A)

eigenvectors =

   –0.2491    0.6862   –0.2425
   –0.5349    0.2568    0.8191
   –0.8073   –0.6806   –0.5199

eigenvalues =

   21.4030         0         0
         0   –0.8452         0
         0         0    0.4422
```

Note if we give the command eig(A) then it returns the eigenvalues of A only but if we give the command [eigenvectors,eigenvalues]=eig(A) then we get both the eigenvectors and eigenvalues.

2.20 COMPLEX NUMBERS

A complex number is in the form a+ib where a and b are real numbers. i is the imaginary unit multiplied to b. The value of i^2 is –1. i could also be written as $\sqrt{-1}$. In the complex number a+ ib, a is called the real part and b is called the imaginary part. A complex number can be represented in the complex plane as

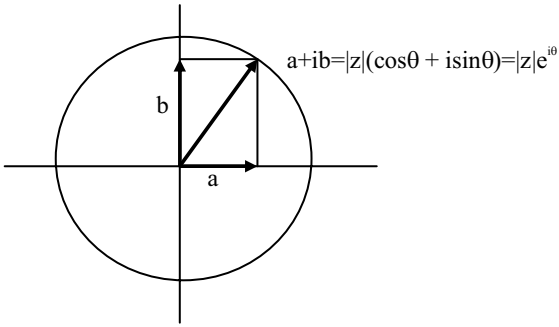

Figure 2.9 A complex number

The complex number a+ib can be written as |z|(cosθ + isinθ) where $|z| = \sqrt{a^2 + b^2}$ is the modulus of the complex number. The complex number a+ib could also be written as $|z|e^{i\theta}$ as cosθ + isinθ = $e^{i\theta}$. MATLAB can be used for processing complex numbers. A few of them have been listed below.

So conj(C1*C2) = conj(C1)*conj(C2).

```
conj(C1/C2)
ans =
   0.4400 – 0.0800i
conj(C1)/conj(C2)
ans =
   0.4400 – 0.0800i
```

So conj(C1/C2) = conj(C1)/conj(C2).

angle

The angle a complex number C, makes with the x-axis can be found using the angle(C) function.

```
C=1+i
C =
   1.0000 + 1.0000i
angle(C)
ans =
   0.7854
```

The angle 0.7854 radians is same as,

(180^o/pi) x 0.7854 = 45.0001^o

The representation of the above complex number would be,

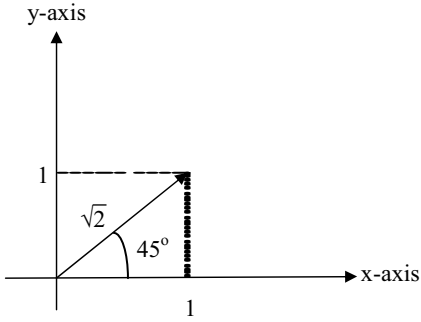

Figure 2.11 Angle of a complex number

Matrix of Complex Numbers

We can also form a matrix of complex numbers.

Let us say C is a row matrix of 3 complex numbers.

```
C= [1+3i 3+2i 4+2i]
C= [1+3i 3+2i 4+2i]
C =
   1.0000 + 3.0000i 3.0000 + 2.0000i 4.0000 + 2.0000i
```

Or

```
C= [1+3*i 3+2*i 4+2*i]
C =
   1.0000 + 3.0000i 3.0000 + 2.0000i 4.0000 + 2.0000i
abs(C)
ans =
   3.1623 3.6056 4.4721
conj(C)
ans =
   1.0000 – 3.0000i 3.0000 – 2.0000i 4.0000 – 2.0000i
imag(C)
ans =
   3      2      2
real(C)
ans =
   1      3      4
C'
ans =
   1.0000 – 3.0000i
   3.0000 – 2.0000i
   4.0000 – 2.0000i
```

Note here that C' leads to conjugate of the complex numbers in a column matrix.

```
abs(C')
```

upper

The **upper** command changes the small letters in the string to capital letters.

```
upper('matlab')
ans =
MATLAB
upper('matlab is all caps')
ans =
MATLAB IS ALL CAPS
```

strcat

The **strcat** command joins one string with another. In other words, it concatenates string a with string b.

```
a = {'MAT'}
a =
   'MAT'
b = {'LAB'}
b =
   'LAB'
ab=strcat(a,b)
ab =
   'MATLAB'
```

strvcat

The command **strvcat** vertically concatenates two or more strings. The command for vertically concatenating $a_1,a_2,a_3,\ldots a_n$ strings is **strvcat(a_1,a_2,a_3,...,a_n).**

```
a=('MAT')
a =
MAT
b=('LAB')
b =
LAB
strvcat(a,b)
```

```
ans =
MAT
LAB
strvcat('M','A','T','L','A','B')
ans =
M
A
T
L
A
B
```

strcmp

The **strcmp** command compares one string with another and returns 1 if the two are same and returns 0 when they are not same.

```
a={'MAT'}
a =
'MAT'
b={'MAT'}
b =
   'MAT'
strcmp(a,b)
ans =
   1
a={'matlab'}
a =
   'matlab'
b={'MATLAB'}
b =
   'MATLAB'
```

```
strcmp(a,b)
ans =
   0
```

isletter

The command **isletter** lets the user know in a string of characters the position of alphabetic and non-alphabetic characters. It returns 1(or true) for alphabetical characters and 0(or false) for non-alphabetic characters.

```
a=['I like this']
a =
I like this
isletter(a)
ans =
   1     0     1     1     1     1     0     1     1     1     1
```

Another example,

```
a=['1 A 2 B 3 C']
a =
1 A 2 B 3 C
isletter(a)
ans =
   0     0     1     0     0     0     1     0     0     0     1
```

Let us try another example,

```
a=['! A & B ? C']
a =
! A & B ? C
isletter(a)
ans =
   0     0     1     0     0     0     1     0     0     0     1
```

So now we know it only returns 1 for alphabets and for number, blanks, special characters like !, & or ? it returns 0.

student(2).roll or student(3).name. The advantage of using structure arrays is that a data of a particular kind associated with an event can be processed together. Here it is possible to find the sum of all the marks of students in a class or directly get all information regarding a student in a large list.

In the example below each data is entered individually.

```
student(1).roll=25;
student(1).name='Akshay';
student(1).marks=88;
student(2).roll=27;
student(2).name='Abhay';
student(2).marks=81;
student(3).roll=33;
student(3).name='Ali';
student(3).marks=80;
student.roll

ans =

   25

ans =

   27

ans =

   33

student.name

ans =

Akshay

ans =

Abhay

ans =

Ali

student.marks

ans =

   88

ans =

   81
```

```
ans =

80
```

Apart from getting individual data all the data in a field can be also obtained. For example, all the marks can be obtained and later the average of the marks can be calculated.

```
marks=[student.marks]

marks =

   88      81      80

avg_marks=mean(marks)

avg_marks =

   83

roll=[student.roll]

roll =

   25      27      33
```

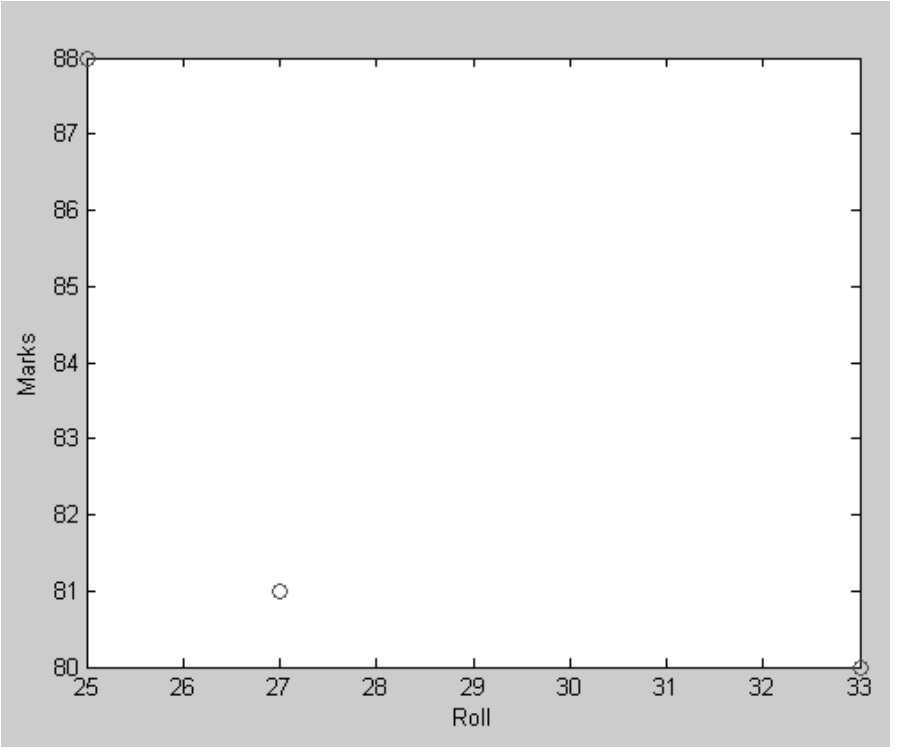

Figure 2.12 Plot of marks obtained by various students

```
plot(roll,marks)
plot(roll,marks,'ro')
plot(roll,marks,'ro')
xlabel('Roll')
ylabel('Marks')
```

As can be seen above it is also possible to plot the data.

Earlier data had been entered individually for all the fields of the structures. We can also enter data of all the fields of the structure together.

```
student(1)=struct('roll',[25],'name',['Akshay'],'marks',[88]);
student(2)=struct('roll',[27],'name',['Abhay'],'marks',[81]);
student(3)=struct('roll',[31],'name',['Ali'],'marks',[80]);
student.roll

ans =

   25

ans =

   27

ans =

   31

student.name

ans =

Akshay

ans =

Abhay

ans =

Ali

student.marks

ans =

   88

ans =

   81

ans =

   80
```

```
x=[student.roll]

x =

   25      27      31

y=[student.marks]

y =

   88      81      80

bar(x,y)

xlabel('Roll')
ylabel('Marks')
```

It is also possible to concatenate structures. For example, if 2 structures str1 and str2 are constructed then a new structure str can be constructed using str1 and str2 by using the command str=[str1 str2]. This has been shown below.

```
str1= struct('field1','abc','field2',123);
str2= struct('field1','def','field2',456);

str=[str1  str2]

str =

1×2 struct array with fields:
```

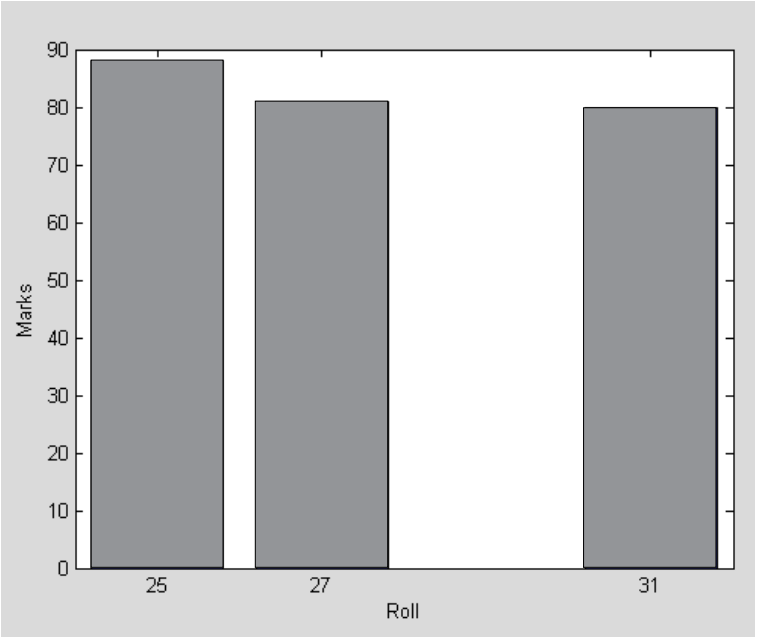

Figure 2.13 Bar plot of marks obtained by various students

Table 2.1 A few commands for structures

Command	Output	Note
namestd=fieldnames(student)	namestd = 'roll' 'name' 'marks'	fieldnames gives all the fields in a structure
ordstd=orderfields(student)	ordstd = 1×3 struct array with fields: marks name roll	orderfields arranges all the fields in the structure in ASCII dictionary order
newstd=rmfield(student,'marks')	newstd = 1×3 struct array with fields: roll name	rmfield removes a field form the structure
fname=isfield(student,'grade')	fname = 0	isfield can be used to test if a field exists in a structure output is 0 if field is not
fname=isfield(student,'marks')	fname = 1	present output is 1 if a the field is present

field1
field2

An example given below illustrates the concatenation of three structures.

```
company(1)=struct('name','ABC','estd',1920,'rank',1)

company =

1×2 struct array with fields:
name
estd
rank

company(2)=struct('name','DEF','estd',1935,'rank',2)

company =

1×2 struct array with fields:
name
estd
rank

company(3)=struct('name','GHI','estd',1940,'rank',3)

company =

1×3 struct array with fields:
```

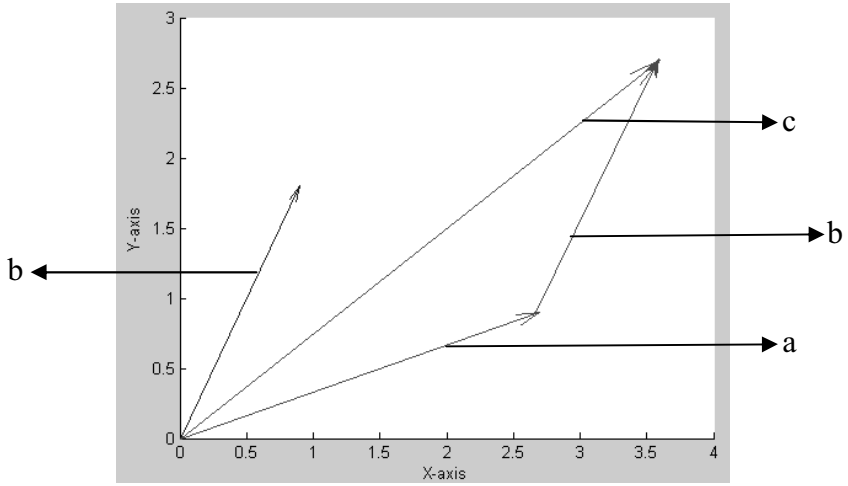

Figure 2.16 Addition of two vectors using MATLAB

Subtraction of Two Vectors

A vector can be subtracted from another vector to give a third vector. For example if **a** and **b** are two vectors then b can be subtracted from a to give to a third vector or the resultant vector **c**. Here vector **b** is translated to meet vector **a** at its end (A) and its direction is completely reversed in order to get vector **–b**. Then the arrow joining the points O and B is the vector **c** = **a** – **b**.

An example of subtraction of two vectors using MATLAB has been shown below.

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

c =

–2          1

quiver(0,0,1,2)
hold on
quiver(0,0,3,1)
quiver(0,0,c(1),c(2))
```

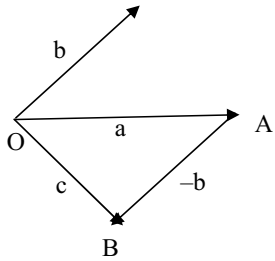

Figure 2.17 Subtraction of two vectors

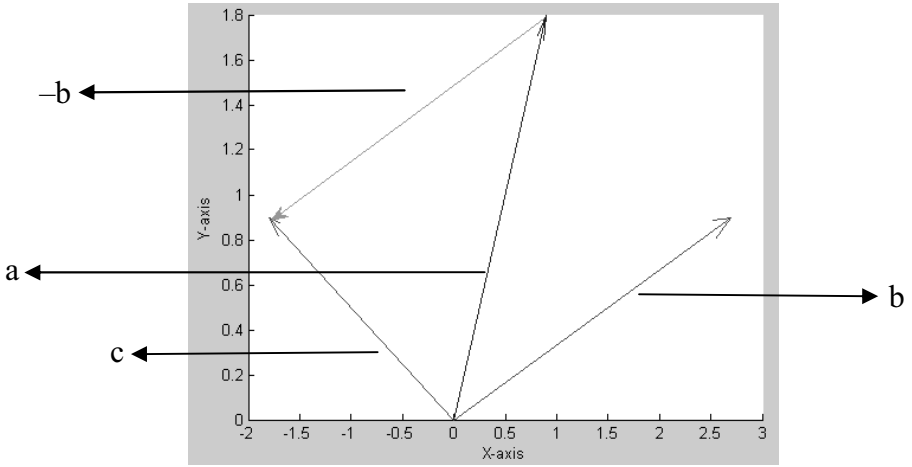

Figure 2.18 Subtraction of two vectors using MATLAB

```
xlabel('X-axis')
ylabel('Y-axis')
```

Dot Product of Two Vectors

The dot product of two vectors V_1 and V_2 is given by,

If $\mathbf{V_1} = a_1\mathbf{i} + b_1\mathbf{j} + c_1\mathbf{k}$ and $\mathbf{V_2} = a_2\mathbf{i} + b_2\mathbf{j} + c_2\mathbf{k}$, then

$\mathbf{V_1}.\mathbf{V_2} = a_1xa_2 + b_1xb_2 + c_1xc_2$

The dot product of two vector quantities is a scalar quantity. The dot product of two vectors perpendicular to each other is 0. In MATLAB the dot product of vectors **a** and **b** can be found using the function dot(a,b).

```
v1=[1 2 3];
v2=[4 5 6];
d=dot(v1,v2)

d =

   32
```

Another example illustrating the dot(a,b) function of MATLAB is given below.

```
a=[2 4 6];
b=[8 10 12];
c=dot(a,b)

c =

   128
```

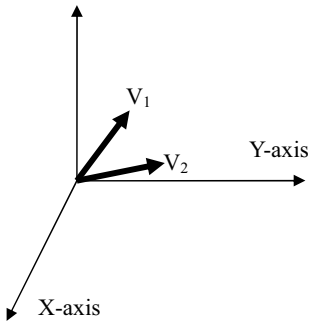

Figure 2.19 Angle between two vectors

```
costheta=((1*2)+(1*2)+(1*2))/(sqrt((1*1)+(1*1)+(1*1))*sqrt((2*2)+(2*2)+(2*2)))

costheta =

   1.0000
```

Which implies that the angle between the vectors is 0^o or the two vectors are parallel to each other.

```
a=[1 0];
b=[0 1];
costheta=((1*0)+(0*1))/(sqrt((1*1)+(0*0))*sqrt((0*0)+(1*1)))

costheta =

   0

plot(a)
hold on
plot(b)
axis([-2 2 -2 2])
xlabel('x-axis')
ylabel('y-axis')
```

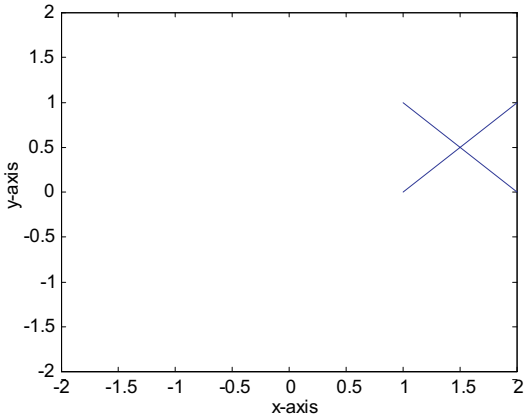

Figure 2.20 Angle between two vectors using MATLAB

```
element(5)
ans =
   name: 'B'
   atwt: 10.8100
      mp: 2076
element(20)
ans =
   name: 'Ca'
   atwt: 40.0780
      mp: 842
```

The names of the various fields can be found using the command fieldnames(structure).

```
name=fieldnames(element)
name =
   'name'
   'atwt'
   'mp'
```

It is possible to order the fields of the structure using the command orderfields(structure).

```
pt
element
element =
1×20 struct array with fields:
   name
   atwt
   mp
ordelement=orderfields(element)
ordelement =
1×20 struct array with fields:
   atwt
   mp
   name
```

It is also possible to remove a field using command rmfield(structure,'fieldname').

```
rmfield(element,'mp')
newelement =
1×20 struct array with fields:
   name
   atwt
newelement
```