## Matrices

### Creating a matrix

Matrices can be created using the ***matrix()*** command.

The arguments to be supplied are

1. vector of values to be entered.
2. dimensions of the matrix, specifying either the numbers of rows or columns.

Additionally you can specify if the values are to be allocated by row or column. By default they are allocated by column.

```
Vec1 = c(1,4,5,6,4,5,5,7,9)   # 9 elements

A = matrix(Vec1,nrow=3)       #3 by 3 matrix, assigned by column.

A

#      [,1] [,2] [,3]
# [1,]    1    6    5
# [2,]    4    4    7
# [3,]    5    5    9
```

Notice how the rows and column are preceded with row and column indexes. To assign by row, we must specify it by setting the appropriate argument accordingly.

```
#3 by 3 matrix. Values assigned by row.
C= matrix(  c(1,6,7,0.6,0.5,0.3,1,2,1), ncol=3 , byrow =TRUE)

C
     [,1] [,2] [,3]
[1,]  1.0  6.0  7.0
[2,]  0.6  0.5  0.3
[3,]  1.0  2.0  1.0
```

## Accessing Rows and Columns

Particular rows and columns of a data object (matrix as well as other objects such as data frames) can be accessed by specifying the row number or column number, leaving the other value blank.

```
A[1,]   # access first row of A
#[1] 1 6 5

C[,2]   # access second column of C
#[1] 6.0 0.5 2.0
```

Naturally - particular elements may be accessed by specifying the row number and column number

```
A[1,1]
# [1] 1

# This is not just for matrices.
# It is for all suitable data objects.

iris[,1]
iris[,1:3]
iris[2,]
iris[2,3]

mtcars[2:5,4:6]
```

## Addition and subtractions

For matrices, addition and subtraction works on an element-wise basis. The first elements of the respective matrices are added, and so on.

```
A+C

#     [,1] [,2] [,3]
#[1,]  2.0 12.0 12.0
#[2,]  4.6  4.5  7.3
#[3,]  6.0  7.0 10.0

A-C

#     [,1] [,2] [,3]
#[1,]  0.0  0.0 -2.0
#[2,]  3.4  3.5  6.7
#[3,]  4.0  3.0  8.0
```

### Matrix Multiplication

To multiply matrices, we require a special operator for matrices; "`%*%`".

If we just used the normal multiplication, we would get an element-wise multiplication.

This type of operation is very useful as a substitute for "for" loops on many occasions.

```
 A %*% C
#     [,1] [,2] [,3]
#[1,]  9.6 19.0 13.8
#[2,] 13.4 40.0 36.2
#[3,] 17.0 50.5 45.5

 A*C
#     [,1] [,2] [,3]
#[1,]  1.0   36 35.0
#[2,]  2.4    2  2.1
#[3,]  5.0   10  9.0
```

## Basic Matrix Calculations

### 1) Inverting a matrix

To invert a matrix we use the command ***`solve()`*** with no additional argument.

Remember - Not all matrices are invertible.  It the determinant of a matrix is zero, then no inverse exists.

```
> solve(C)
             [,1]      [,2]       [,3]
[1,] -0.03333333  2.666667 -0.5666667
[2,] -0.10000000 -2.000000  1.3000000
[3,]  0.23333333  1.333333 -1.0333333
```

We can use this same command to solve a system of linear equations ***Ax=b***. We would specify the vector ***b*** as the additional argument. (We will look at this matter more in the MATLAB component of the course).

### 2) Computing the determinant

To compute the determinant, the command is simply ***det()***

### 3) Determining the dimensions

To find the dimensions of matrix A, we use the ***dim()*** command

### 4) Compute the transpose

To compute the transpose of matrix A, we use the command ***t().***

```
det(C)
#[1] 3

dim(C) # number of rows and columns.

t(C)
     [,1] [,2] [,3]
[1,]    1  0.6    1
[2,]    6  0.5    2
[3,]    7  0.3    1
```

## 5) Cross Products and Kronecker Product

We can compute cross products using the `crossprod()` command. The ***Kronecker product*** ( a very useful command in numerical computation) is also easily implementable using the `kronecker()` command.

```
crossprod(A,C)

kronecker(A,C)

kronecker(C,A)
```

## Diagonals and the Identity Matrix

The `diag()` command is a very versatile function for using matrices.

It can be used to create a diagonal matrix with elements of a vector in the principal diagonal. For an existing matrix, it can be used to return a vector containing the elements of the principal diagonal.

Most importantly, if ***k*** is a scalar (i.e. single number such as 3) , `diag()` will create a k x k ***identity*** matrix.

```
Vec2=c(1,2,3)

diag(Vec2)      #      Constructs a diag. matrix based on Vec2

diag(A)         #     Returns diagonal elements of A as a vector

diag(3)         #     Creates a 3 x 3 identity matrix

diag(diag(A))   #     Creates the diagonal matrix D of matrix A
```

## Linear Algebra Functions

***R*** supports many import linear algebra functions such as cholesky decomposition, trace, rank, eigenvalues etc.

The required results may be determinable from the output of a command that pertains to an overall approach.

The eigenvalues and eigenvectors can be computed using the **eigen()** function.  A data object known as a list is then created.

```
eigen(A)        #eigenvalues and eigenvectors

qr(A)           #returns Rank of a matrix

svd(A)
```

This is a very important type of matrix analysis, and many will encounter it again in future modules.

```
Y = eigen(A)

names(Y)

#   y$val are the eigenvalues of A
#   y$vec are the eigenvectors of A
```

### More on Matrices

Note that the following commands are useful for ***Experimental Design***.

| | |
|---|---|
| ***rowMeans(A)*** | Returns vector of row means. |
| ***rowSums(A)*** | Returns vector of row sums. |
| ***colMeans(A)*** | Returns vector of column means. |
| ***colSums(A)*** | Returns vector of column means. |

## Using rbind() and cbind()

Another methods of creating a matrix is to "bind" a number of vectors together, either by row or by column. The commands are **`rbind()`** and **`cbind()`** respectively.

```
> x1 =c(1,2) ; x2 = c(3,8)

> D= rbind(x1,x2)

> E = cbind(x1,x2)

> det(D)

[1] 2

> det(E)

[1] 2
```

## Solving a System of Linear Equations

To solve a system of linear equations in the form ***Ax=b*** , where A is a square matrix, and b is a column vector of known values, we use the **solve()** command to determine the values of the unknown vector ***x.***

```
b=vec2  # from before

solve(A, b)
```