Scientific Computations in R

STAT2005

Chapter 6

Basic definition of vector and matrix

A vector is just a column or a row of numbers. For example,

$$x = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}$$

is an $n \times 1$ column vector and

$$y = [y_1, \cdots, y_p]$$

is an $1 \times p$ row vector.

Matrix is a collection of numbers arranged in rectangular shape. For example,

$$X = \begin{bmatrix} x_{11} & \cdots & x_{1p} \\ \vdots & & \vdots \\ x_{n1} & \cdots & x_{np} \end{bmatrix}$$

is an $n \times p$ matrix and sometimes denoted by

$$X = (x_{ij})$$

where x_{ij} is the (i, j)-th element of X.

Vectors can be considered as a matrix with one column or one row.

Transpose of vector and matrix

The transpose of an $n \times 1$ column vector is an $1 \times n$ row vector and the transpose of an $1 \times p$ row vector is a $p \times 1$ column vector. For example,

$$x' = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}' = [x_1, \ldots, x_n]$$

The transpose of an $n \times p$ matrix is a $p \times n$ matrix with the original rows become columns and vice versa. For example,

$$X' = \begin{bmatrix} x_{11} & \cdots & x_{n1} \\ \vdots & & \vdots \\ x_{1p} & \cdots & x_{np} \end{bmatrix}$$

Matrix addition and multiplication

Let

$$A=\begin{bmatrix} a_{11} & \cdots & a_{1p} \\ \vdots & & \vdots \\ a_{n1} & \cdots & a_{np} \end{bmatrix}, B=\begin{bmatrix} b_{11} & \cdots & b_{1q} \\ \vdots & & \vdots \\ b_{p1} & \cdots & b_{pq} \end{bmatrix} \text{ and } X=\begin{bmatrix} x_{11} & \cdots & x_{1p} \\ \vdots & & \vdots \\ x_{n1} & \cdots & x_{np} \end{bmatrix}$$

be $n \times p$, $p \times q$ and $n \times p$ matrices respectively. Then

$$A+X=\begin{bmatrix} a_{11}+x_{11} & \cdots & a_{1p}+x_{1p} \\ \vdots & & \vdots \\ a_{n1}+x_{n1} & \cdots & a_{np}+x_{np} \end{bmatrix}$$

is an $n \times p$ matrix formed by simply adding up the corresponding elements in A and X.

Let C be the product of A and B, i.e., $C = AB$.

C is an $n \times q$ matrix whose elements are:

$$C = \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} + \ldots + a_{1p}b_{p1} & \cdots & a_{11}b_{1q} + a_{12}b_{2q} + \ldots + a_{1p}b_{pq} \\ \vdots & & \vdots \\ a_{n1}b_{11} + a_{n2}b_{21} + \ldots + a_{np}b_{p1} & \cdots & a_{n1}b_{1q} + a_{n2}b_{2q} + \ldots + a_{np}b_{pq} \end{bmatrix}$$

In general, the (i, j)-th element of C is

$$c_{ij} = \sum_{k=1}^{p} a_{ik}b_{kj}$$

for $i = 1, \ldots, n$, $j = 1, \ldots, q$.

For example,

$$X = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix}, \quad y = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}, \text{ then } Xy = \begin{bmatrix} 30 \\ 70 \\ 110 \end{bmatrix}$$

Note that $A + X = X + A$ but $AB \neq BA$ in general.

Let x and y be $p \times 1$ column vectors. The multiplication in vectors is the dot-product or inner-product.

$$x'y = x_1 y_1 + \ldots + x_p y_p = \sum_{i=1}^{p} x_i y_i$$

In particular,

$$y'y = y_1^2 + \ldots + y_p^2 = \sum_{i=1}^{p} y_i^2 \geq 0$$

The norm of y could be defined as $\sqrt{y'y}$, which is sometimes denoted by $\|y\|_2$. Note that $\|y\|_2 = 0$ if and only if $y = 0$.

The outer-product of x and y is a $p \times p$ matrix xy', i.e. the (i, j)-th element of this matrix is $x_i y_j$.

For example

$$y = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}, \quad y'y = (1,2,3)\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} = 14,$$

$$yy' = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}[1,2,3] = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 6 \\ 3 & 6 & 9 \end{bmatrix}.$$

R has built-in function for transpose `t( )`, matrix multiplication %*%, inner product %*% and outer product %o%. Let us illustrate these operations by the following examples.

```
> y<-matrix(1:12,ncol=4) # create a 3x4 matrix y
> y        # or simply use (y<-matrix(1:12,ncol=4))
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12

> t(y)     # transpose of Y, 4x3 matrix
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
[4,]   10   11   12

> t(y)%*%y  # matrix multiplication Y'Y, 4x4 matrix
     [,1] [,2] [,3] [,4]
[1,]   14   32   50   68
[2,]   32   77  122  167
[3,]   50  122  194  266
[4,]   68  167  266  365
```

```
> y%*%t(y)    # YY'3x3 matrix
     [,1] [,2] [,3]
[1,]  166  188  210
[2,]  188  214  240
[3,]  210  240  270

> (x<-1:4)    # create and display a vector x
 [1] 1 2 3 4

> x%*%x       # inter-product = x'x
     [,1]
[1,]   30

> x%o%x       # outer-product = xx', 4x4 matrix
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    2    4    6    8
[3,]    3    6    9   12
[4,]    4    8   12   16
```

Special type of matrices

- A square matrix is matrix with its number of rows equals to its number of columns.
- A symmetric matrix S is a square matrix whose elements are symmetric about its diagonal, i.e., $s_{ij} = s_{ji}$, $\forall\ i, j$, or $S' = S$.
- A diagonal matrix A is a square matrix with all its off-diagonal elements equal zero, i.e., $a_{ij} = 0$ for $i \neq j$.
- An identity matrix of order n, denoted by I_n is an $n \times n$ diagonal matrix whose diagonal elements equal to 1.

An important property of identity matrix is that

$$I_n X = X \text{ and } X I_p = X$$

for any $n \times p$ matrix X. For example,

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix}$$

and

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix}$$

We can form a diagonal matrix from a vector or extract the diagonal elements of a matrix to a vector by using the `diag()` function in R.

```
> x<-1:4      # create a vector
> (y<-diag(x))
# create and display a diagonal matrix
     [,1] [,2] [,3] [,4]
[1,]    1    0    0    0
[2,]    0    2    0    0
[3,]    0    0    3    0
[4,]    0    0    0    4
> diag(y)
# returns a vector if the argument is a matrix
[1] 1 2 3 4
```

This diag() function can still be use even though the matrix is not a square matrix.

```
> (a<-matrix(1:12,3))
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12

> diag(a)
[1] 1 5 9

> (b<-matrix(1:12,4))
     [,1] [,2] [,3]
[1,]    1    5    9
[2,]    2    6   10
[3,]    3    7   11
[4,]    4    8   12

> diag(b)
[1]  1  6 11
```

Exercise

1. Suggest two ways to generate identity matrices in R.
2. Given a matrix u with the following pattern.

```
> u
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    0    1    2    3
[3,]    0    0    1    2
[4,]    0    0    0    1
```

Use a single line command to generate the following matrix.

```
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    2    1    2    3
[3,]    3    2    1    2
[4,]    4    3    2    1
```

Matrix inverse

The inverse of an $n \times n$ square matrix X, denoted by X^{-1}, is an $n \times n$ square matrix such that

$$XX^{-1} = X^{-1}X = I_n.$$

The general formula for the inverse of an $n \times n$ matrix would be complicated, but there is a simple formula of the inverse of a 2×2 matrix worth memorizing. Let

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

Then, the inverse of A is

$$A^{-1} = \frac{1}{ad - bc} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix}$$

For example,

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}, \quad A^{-1} = \frac{1}{-2}\begin{bmatrix} 4 & -2 \\ -3 & 1 \end{bmatrix} = \begin{bmatrix} -2 & 1 \\ 1.5 & -0.5 \end{bmatrix}$$

Check:

$$AA^{-1} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}\begin{bmatrix} -2 & 1 \\ 1.5 & -0.5 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Inverse of a matrix may not always exist. In particular, if $ad - bc = 0$, then the inverse of A does not exist. A matrix is called a singular matrix if its inverse does not exist.

Determinant

The determinant of a square matrix A, denoted as $\det(A)$, is a number that could tell us whether a matrix is invertible.

It can be shown that any square matrix has a unique inverse if and only if its determinant is nonzero.

In particular, if A is a 2×2 matrix

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

then $\det(A) = ad - bc$.

The determinant of a 3×3 matrix

$$A = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$$

is given by

$$\det(A) = a\det\left(\begin{bmatrix} e & f \\ h & i \end{bmatrix}\right) - b\det\left(\begin{bmatrix} d & f \\ g & i \end{bmatrix}\right) + c\det\left(\begin{bmatrix} d & e \\ g & h \end{bmatrix}\right)$$

In R, the function `det()` returns the value of the determinant for any square matrix.

Solving system of linear equations

The inverse of a matrix can be used to solve a system of linear equations. Let us first consider the following system of linear equation:

$$\begin{cases} x_1 + 2x_2 = 5 \\ 3x_1 + 4x_2 = 6 \end{cases}$$

This can be rewritten in matrix form as

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 5 \\ 6 \end{bmatrix}$$

Pre-multiply

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}^{-1} = \begin{bmatrix} -2 & 1 \\ 1.5 & -0.5 \end{bmatrix}$$

to both sides of the equation, we have

$$\begin{bmatrix} -2 & 1 \\ 1.5 & -0.5 \end{bmatrix}\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} -2 & 1 \\ 1.5 & -0.5 \end{bmatrix}\begin{bmatrix} 5 \\ 6 \end{bmatrix}$$

$$\Rightarrow \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} -4 \\ 4.5 \end{bmatrix}$$

Let us try the above example using R.

```
> A<-matrix(c(1,3,2,4),2)
# define the matrix A
> b<-c(5,6)
# define the vector b
> (x<-solve(A,b))
# solve for x and display the answer
[1] -4.0 4.5
> A%*%x
# check Ax to see if equal to b
     [,1]
[1,]    5
[2,]    6
```

In general, if we have a system of n linear equations:

$$\begin{cases} a_{11}x_1 + \ldots + a_{1n}x_n = b_1 \\ \vdots \\ a_{n1}x_1 + \ldots + a_{nn}x_n = b_n \end{cases}$$

which can be rewritten as $Ax = b$, where

$$A = \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & & \vdots \\ a_{n1} & \cdots & a_{nn} \end{bmatrix}, \quad x = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}, \quad b = \begin{bmatrix} b_1 \\ \vdots \\ b_n \end{bmatrix}.$$

This system can be solved by pre-multiplying the inverse of A (given that it exists) to both sides of the equation, i.e.,

$$A^{-1}Ax = A^{-1}b \Rightarrow x = A^{-1}b$$

In terms of linear algebra, finding matrix inverse can be done by first creating an identity matrix on the right hand side, and then use the function `solve()`.

```
> iden<-diag(c(1,1))
# create an identity matrix
> solve(A,iden)
# find the inverse of A or simply use solve(A)
     [,1] [,2]
[1,] -2.0  1.0
[2,]  1.5 -0.5
```

In fact, it could also be done by simply entering `solve(A)`.

Applications of matrices in statistics

Two useful statistics to measure the linear relationship between two random variables are covariance and correlation. Let $x_1 = [x_{11}, \dots, x_{1n}]$ and $x_2 = [x_{21}, \dots, x_{2n}]$ be two vectors of random sample of the random variables X_1 and X_2 respectively. The **sample** covariance between x_1 and x_2 is defined as

$$\mathrm{Cov}(x_1, x_2) = \frac{1}{n-1}\sum_{i=1}^{n}(x_{1i} - \bar{x}_1)(x_{2i} - \bar{x}_2).$$

Note that $Cov(x_1, x_1) = Var(x_1)$.

The correlation between x_1 and x_2 is defined as

$$\mathrm{Corr}(x_1, x_2) = \frac{\mathrm{Cov}(x_1, x_2)}{\sqrt{\mathrm{Var}(x_1)\mathrm{Var}(x_2)}}$$

Note that $Corr(x_1, x_1) = 1$.

In general, if there are p variables $X_1, \ldots, X_p$, then we can form $p(p+1)/2$ pairs of covariance terms. These terms are arranged in a $p \times p$ symmetric matrix called the covariance matrix:

$$C = \begin{bmatrix} c_{11} & \cdots & c_{1p} \\ \vdots & \ddots & \vdots \\ c_{p1} & \cdots & c_{pp} \end{bmatrix} = \begin{bmatrix} \mathrm{Var}(x_1) & \cdots & \mathrm{Cov}(x_1, x_p) \\ \vdots & \ddots & \vdots \\ \mathrm{Cov}(x_p, x_1) & \cdots & \mathrm{Var}(x_p) \end{bmatrix},$$

where x_i is a vector of random sample from X_i, $i = 1, \ldots, p$.

Similarly, we can form a $p \times p$ symmetric matrix of correlation:

$$R = \begin{bmatrix} 1 & \cdots & r_{1p} \\ \vdots & \ddots & \vdots \\ r_{p1} & \cdots & 1 \end{bmatrix} = \begin{bmatrix} 1 & \cdots & \mathrm{Corr}(x_1, x_p) \\ \vdots & \ddots & \vdots \\ \mathrm{Corr}(x_p, x_1) & \cdots & 1 \end{bmatrix}.$$

The relation between covariance and correlation matrix is given by

$$C = SRS \quad \text{or} \quad \mathrm{R} = S^{-1}CS^{-1}$$

where

$$S = \begin{bmatrix} \sqrt{c_{11}} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \sqrt{c_{pp}} \end{bmatrix} \text{ and } S^{-1} = \begin{bmatrix} \frac{1}{\sqrt{c_{11}}} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \frac{1}{\sqrt{c_{pp}}} \end{bmatrix}$$

R has built-in function for computing the covariance and correlation matrix. Let us use the `iris.csv` dataset as an example.

```
> d<-read.csv("iris.csv")
> colnames(d)<-NULL      # remove the column names
> sigma<-var(d[,1:4])    # covariance matrix
> sigma
            [,1]       [,2]       [,3]       [,4]
[1,]  0.6856935 -0.0424340  1.2743154  0.5162707
[2,] -0.0424340  0.1899794 -0.3296564 -0.1216394
[3,]  1.2743154 -0.3296564  3.1162779  1.2956094
[4,]  0.5162707 -0.1216394  1.2956094  0.5810063
> cor(d[,1:4])           # correlation matrix
            [,1]       [,2]       [,3]       [,4]
[1,]  1.0000000 -0.1175698  0.8717538  0.8179411
[2,] -0.1175698  1.0000000 -0.4284401 -0.3661259
[3,]  0.8717538 -0.4284401  1.0000000  0.9628654
[4,]  0.8179411 -0.3661259  0.9628654  1.0000000
```

Markov chain

Define

- $X_t = U$ if the closing price of Hang Seng Index at day t is up by more than 3% compare with yesterday's closing price;
- $X_t = D$ if it is down by more than 3% and
- $X_t = S$ if it is within 3%.

Hence, X_t is a random variable takes three possible states:

$$\{U = \text{Up},\ \ S = \text{Satationary},\ \ D = \text{Down}\}.$$

Suppose we model X_t by the following:

- If $X_{t-1} = U$, then

$$\Pr\left(X_t = M \mid X_{t-1} = U\right) = \begin{cases} 0.3, & M = U, \\ 0.2, & M = S, \\ 0.5, & M = D. \end{cases}$$

- If $X_{t-1} = S$, then

$$\Pr\left(X_t = M \mid X_{t-1} = S\right) = \begin{cases} 0.4, & M = U, \\ 0.3, & M = S, \\ 0.3, & M = D. \end{cases}$$

- If $X_{t-1} = D$, then

$$\Pr\left(X_t = M \mid X_{t-1} = D\right) = \begin{cases} 0.3, & M = U, \\ 0.4, & M = S, \\ 0.3, & M = D. \end{cases}$$

$\{X_t\}_{t \geq 0}$ is called a Markov chain with transition matrix

$$T = \begin{bmatrix} 0.3 & 0.2 & 0.5 \\ 0.4 & 0.3 & 0.3 \\ 0.3 & 0.4 & 0.3 \end{bmatrix}$$

Note that each row in T must sum to 1.

Let π_t be the probability distribution of X_t. Then the probability distribution of X_{t+1} can be represented as

$$\pi_{t+1} = \pi_t T$$

for $t = 1,2, \ldots$.

Suppose today (day 0) is Up, then $\pi_0 = [1,0,0]$

$$\pi_1 = [1,0,0]\begin{bmatrix} 0.3 & 0.2 & 0.5 \\ 0.4 & 0.3 & 0.3 \\ 0.3 & 0.4 & 0.3 \end{bmatrix} = [0.3,0.2,0.5]$$

and

$$\pi_2 = [0.3,0.2,0.5]\begin{bmatrix} 0.3 & 0.2 & 0.5 \\ 0.4 & 0.3 & 0.3 \\ 0.3 & 0.4 & 0.3 \end{bmatrix}$$

$$= [0.32,0.32,0.36]$$

The computations are performed in R as follows.

```
# define transition matrix T
> T<-
matrix(c(0.3,0.2,0.5,0.4,0.3,0.3,0.3,0.4,0.3),
nrow=3, byrow=T)
> p<-c(1,0,0)            # define p(0)
> (p<-p%*%T)             # compute and display p(1)
     [,1] [,2] [,3]
[1,]  0.3  0.2  0.5
> (p<-p%*%T)             # compute and display p(2)
     [,1] [,2] [,3]
[1,] 0.32 0.32 0.36
```

Repeating the last command for more than 9 times (or using a loop instead), the distribution quickly converges to a limiting distribution

$$\pi = [0.3303571, 0.3035714, 0.3660714].$$

This distribution is also called the invariant distribution of T which satisfies $\pi = \pi T$.

Note: π' is the eigenvector of T' with respect to the eigenvalue 1 such that the sum of all entries in π' equals 1.

```
> eig<-eigen(t(T))
> eig$vectors[,1]/sum(eig$vectors[,1])
[1] 0.3303571+0i 0.3035714+0i 0.3660714+0i
```

Operator or Function	Description
A * B	Element-wise multiplication
A %*% B	Matrix multiplication
A %o% B	Outer product. AB'
t(A)	Transpose
diag(x)	Creates diagonal matrix with elements of x in the principal diagonal
diag(A)	Returns a vector containing the elements of the principal diagonal
diag(k)	If k is a scalar, this creates a k×k identity matrix.
solve(A, b)	Returns vector x in the equation Ax = b (i.e., $A^{-1}b$)
solve(A)	Inverse of A where A is a square matrix.
y<-eigen(A)	y$val are the eigenvalues of A y$vec are the eigenvectors of A
cbind(A,B,...)	Combine matrices(vectors) horizontally. Returns a matrix.
rbind(A,B,...)	Combine matrices(vectors) vertically. Returns a matrix.
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 sums.

Finding roots of nonlinear equation

We often encounter the problem of finding roots of nonlinear equation in many applications.

One-dimensional nonlinear equations are relatively simple and easy to solve by using computer.

R has a built-in function `uniroot()` to find roots of one-dimensional function.

Let us illustrate this by solving the following nonlinear equation that calculates bond yield (y) in finance:

$$3e^{-y\times 0.5} + 3e^{-y\times 1.0} + 3e^{-y\times 1.5} + 103e^{-y\times 2.0} = 98.39$$

```
> fx<-function(x) {3*exp(-0.5*x) + 3*exp(-1*x) +
3*exp(-1.5*x) + 103*exp(-2*x) - 98.39}
# define the function fx
> uniroot(fx,c(0.0001,1))
# finding root of fx
$root                # answer
[1] 0.06762566
$f.root        # function value at root
[1] -0.005176046
$iter                # number of iteration
[1] 4
$estim.prec   # precision of solution
[1] 6.103516e-05
```

First we need to define our own function in R to evaluate $f(x)$ in solving $f(x) = 0$.

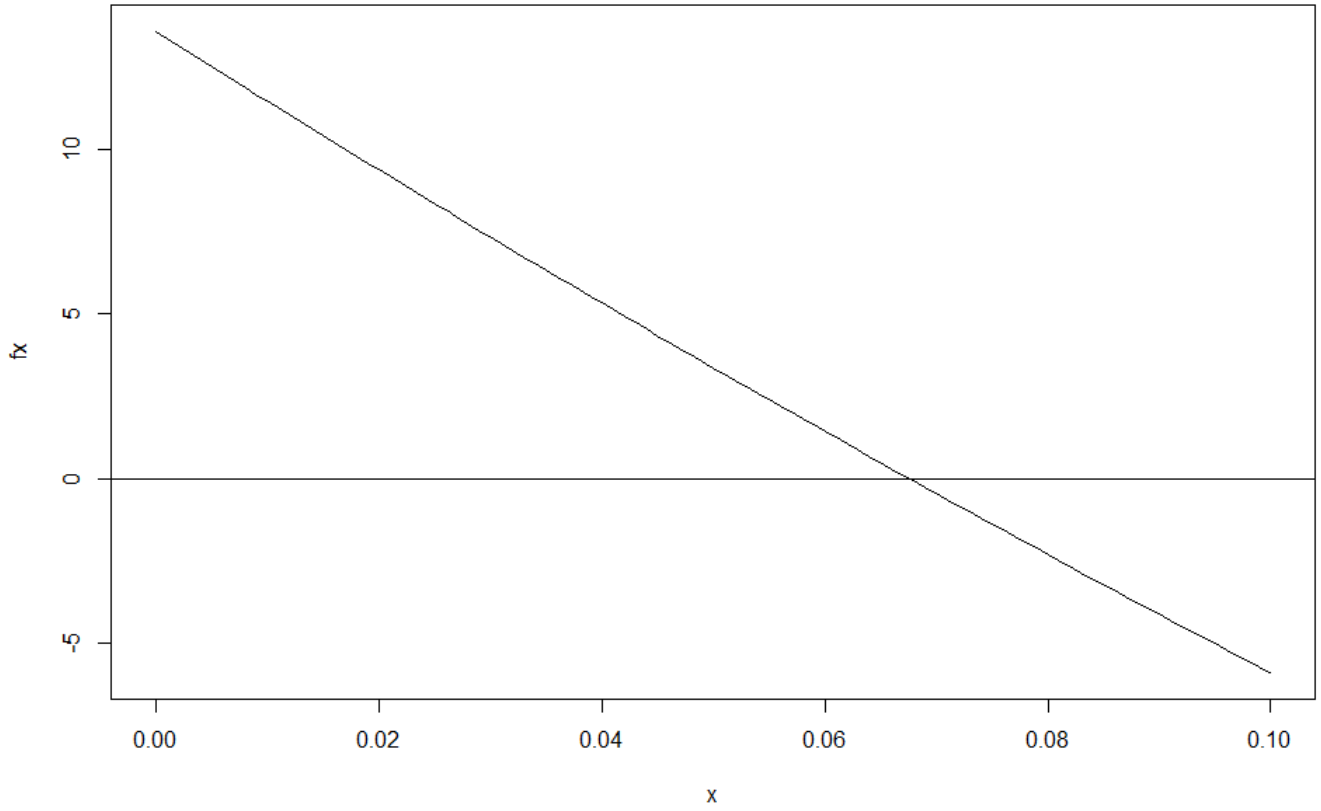
We also need to specify an interval $[0.0001, 1]$ that contains the root.

The value of $f(x)$ at the two end points of this interval must be of different sign, otherwise an error will occur.

The root and other information are given in the output.

We can also plot this function to see the solution graphically.

```
> plot(fx,xlim=c(0,0.1))
# plot fx from 0 to 0.1, or using curve(fx,0,0.1)
> abline(h=0)
# add a horizontal line y=0
```



In fact, the bisection algorithm can be used to find the root of nonlinear equations.

1. Given f, x_1, x_2, compute $f_1 = f(x_1)$, $f_2 = f(x_2)$.
2. If $f_1 f_2 > 0$, stop with error message.
3. Compute $x = \frac{x_1 + x_2}{2}$, $f = f(x)$ and set $n_{\text{it}} = 0$.
4. While $(|f| > \varepsilon)$ and (n_{it} < max. no. of iteration)

   If $f f_2 > 0$, then set $x_2 = x$.

   If $f f_1 > 0$, then set $x_1 = x$.

   Compute $x = \frac{x_1 + x_2}{2}$, $f = f(x)$, increase n_{it} by 1.

1. Output x, f, and n_{it}.

```
bisection<-function(f, x1, x2, n = 1000, err = 1e-05) {
      f1 <-f(x1); f2 <-f(x2)
      if (f1==0) return(x1)
      else if (f2==0) return(x2)
      else if (f1*f2>0) stop("Roots may not exist in range")
      else {
            x <-(x1+x2)/2; fx <-f(x)
            i <-0
            while ((abs(fx)>err)&(i<=n)) {
                  if (fx*f2>0) {
                        x2 <-x
                  } else if (fx*f1>0) {
                        x1 <-x
                  }
                  x <-(x1+x2)/2; fx<-f(x)
                  i <-i+1
            }
      }
      return(x)
}
bisection(fx,0.0001,1,1000,10^-5)
```

Numerical integration

R has built-in function to perform numerical integration.

Let us work with the familiar example to compute the area under the standard normal curve.

```
stdnorm<-function(x) { exp(-x^2/2)/sqrt(2*pi) }
# define standard normal density
integrate(stdnorm, -Inf, 0)
# integrate from –infinity to 0
0.5 with absolute error < 4.7e-05

integrate(stdnorm, -Inf, 1.96)
0.9750021 with absolute error < 1.3e-06
# integrate from –inf to 1.96
```

Differentiation

The build-in function `D()` computes derivative of a simple expression symbolically. Its syntax is

```
D (expr, "x")
```

where `expr` is an expression object in terms of `x`, and `"x"` is a variable name.

To find second order derivative, we may use

```
D (D (expr, "x"), "x")
```

Examples

```
fx <- expression(x^2 + sin(x))      # f(x)
dfdx <- D(fx, "x")                  # f'(x)
x<-5
eval(fx)                            # Compute f(5)
eval(dfdx)                          # Compute f'(5)

gxy <- expression(exp(y*x))
# g(x, y)
dgdx <- D(gxy, "x")
# Partial derivative wrt x
dgdy <- D(gxy, "y")
# Partial derivative wrt y
d2gdxdy <- D(dgdx, "y")
# Cross partial derivative with respect to x and y
```

Univariate optimization

Closely related to the problem of solving nonlinear equation is optimization.

First note that solving the nonlinear equation $f(x) = 0$ is equivalent to minimize the objective function $g(x) = [f(x)]^2$.

R has a built-in function `optimize()` for univariate optimization.

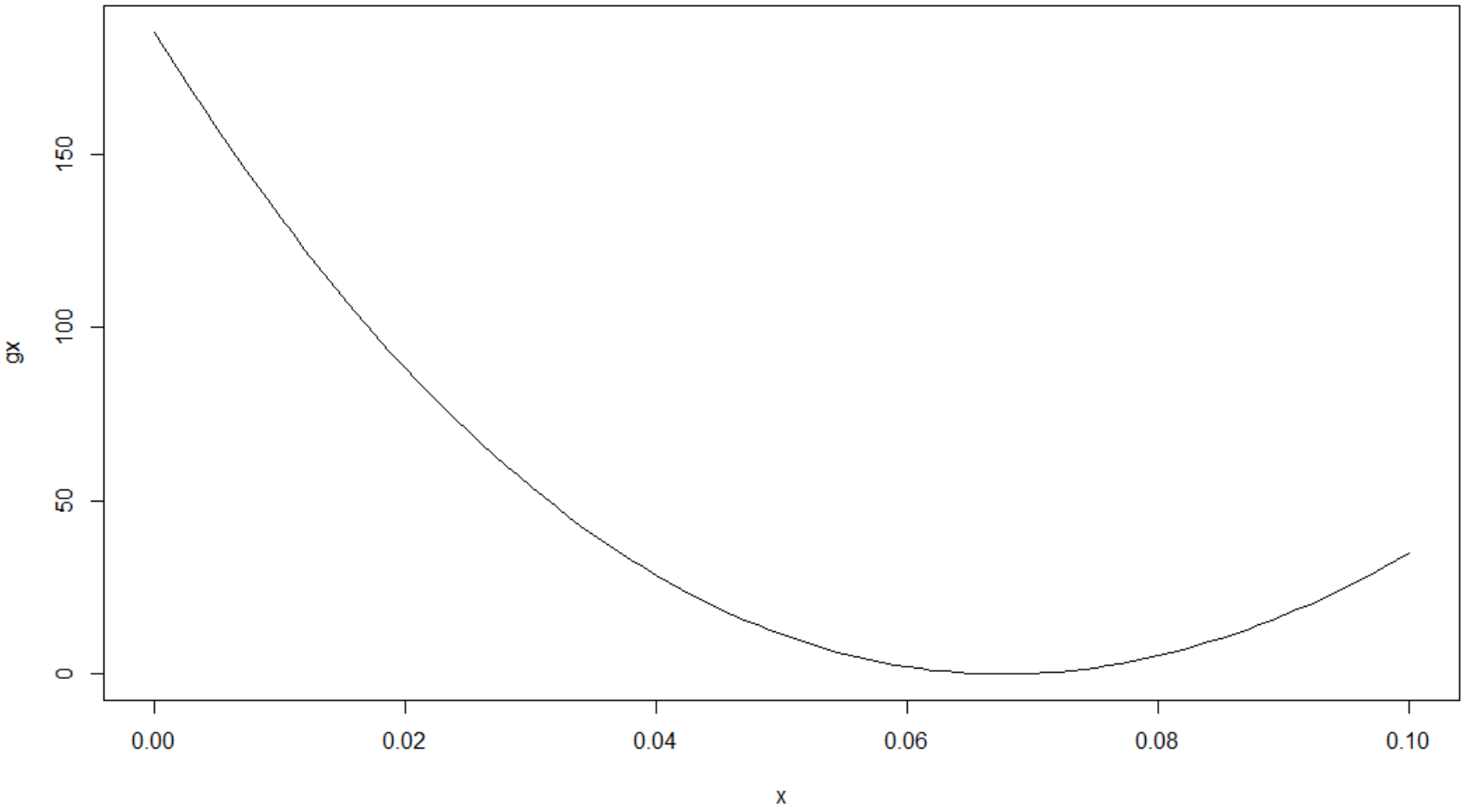
Let us illustrate this by the previous example.

```
fx<-function(x) {3*exp(-0.5*x) + 3*exp(-1*x) +
3*exp(-1.5*x) + 103*exp(-2*x) - 98.39}
gx<-function(x) {fx(x)^2}  # define gx = fx*fx
optimize(gx,c(0.0001,1))   # minimize gx
```

It is of interest to plot the function *g*(x) to compare with *f*(x) and to see the solution.

```
plot(gx,xlim=c(0,0.1)) # or using curve(gx,0,0.1)
```

Note that the default is to minimize the function. If we want to maximize the function, we can add the option `maximum = TRUE` inside the `optimize()` function.


gx
150
100
50
0
0.00
0.02
0.04
0.06
0.08
0.10
x

Let us consider another example. We want to find the value of x such that the following function is minimized:

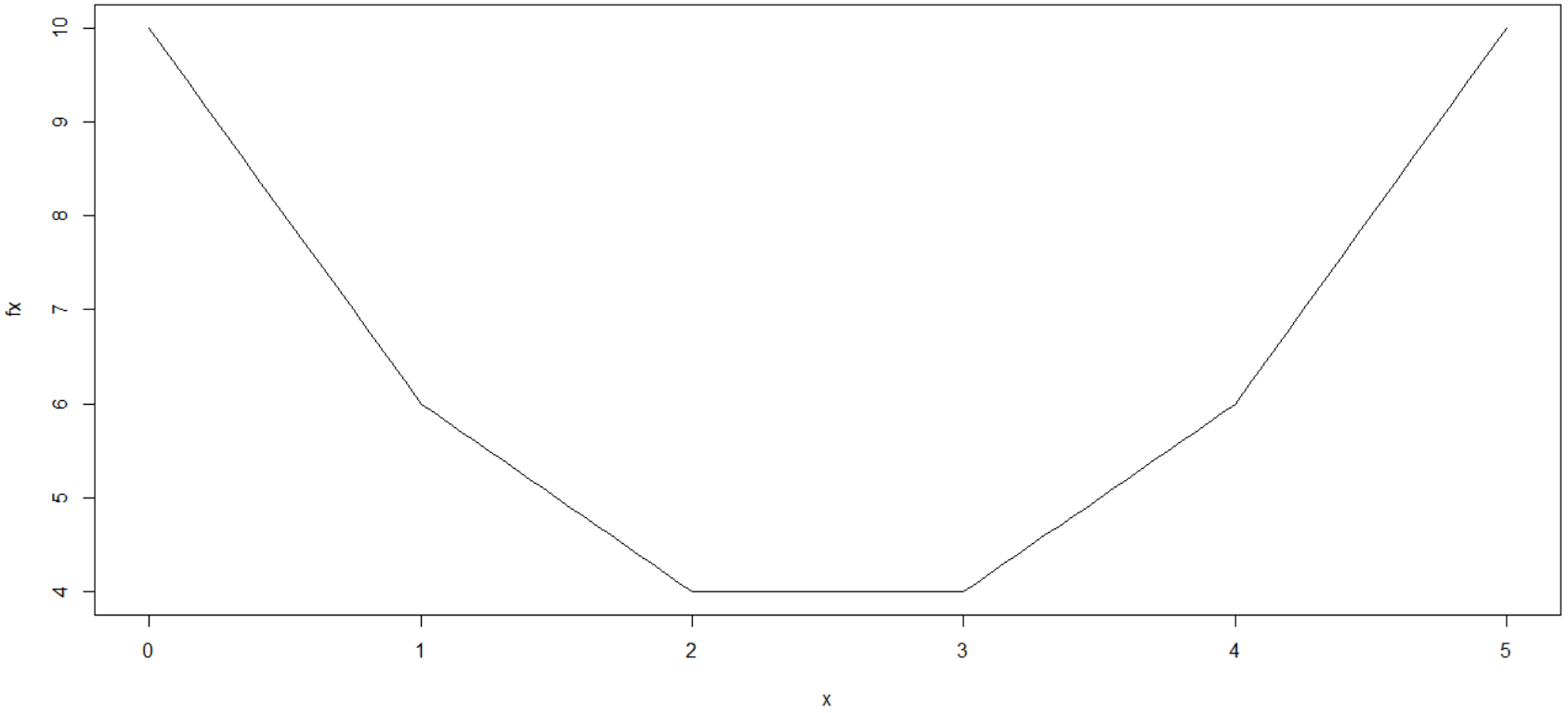
$$f(x) = |x-1| + |x-2| + |x-3| + |x-4|$$

Before we minimize this function using R, can you guess what the answer is and what will this function look like?

```
> fx<-function(x) {abs(x-1)+abs(x-2)+abs(x-
3)+abs(x-4)}  # define fx
> optimize(fx,c(0,10))      # minimize fx
$minimum                    # answer
[1] 2.323395
$objective
[1] 4
```

For one-dimensional function, we can easily plot and look at the function near the optimal value by

```
plot(fx,xlim=c(0,5))
```



In this example, the minimum value of the objective function is 4 but the solution is not unique. In fact, any values between 2 and 3 will be the answer.

Multivariate optimization

Optimizing function of several variables is much more difficult than one-dimensional function.

However, we often encounter multi-dimensional function rather than one-dimensional function in many applications.

R has a built-in function `optim()` to deal with multivariate optimization. This `optim()` function is a general purpose function optimization and the syntax is very similar to the function `optimize()`.

Let us consider the problem of minimizing the following function:

$$f(x_1, x_2) = \frac{-\sin\sqrt{(x_1-5)^2+(x_2-5)^2}}{\sqrt{(x_1-5)^2+(x_2-5)^2}}$$

with initial value $(x_1, x_2) = (2,2)$.

The global minimum of the function value is -1 and the function is minimized at $(x_1, x_2) = (5,5)$.

```
fx<-function(x) {
     # fx takes the vector x as input argument
     p<-sqrt((x[1]-5)^2+(x[2]-5)^2)
     -sin(p)/p
}
optim(c(2,2),fx)
# minimize fx with initial x=(2,2)
$par
[1] 4.999951 4.999967
$value
[1] -1
$counts
function gradient
      73       NA
$convergence
[1] 0
```

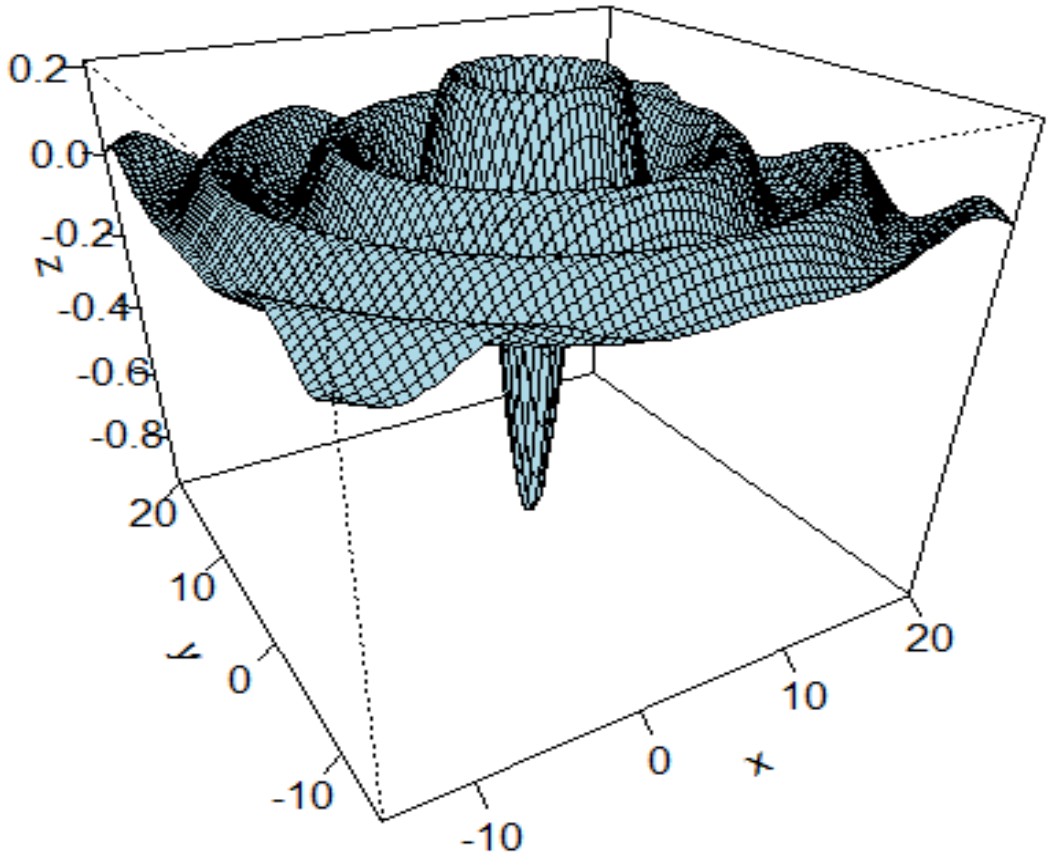
Exercise

Change the initial value $(2, 2)$ to other values and see what will be the solution.

Since the objective function is of two variables, it is possible to produce a 3-dimensional plot of this function near the solution.

R has a built in function `persp()` (stands for Perspective Plots) for a 3D plot.

```
fxy<-function(x,y) {
# re-defnine fx using two arguments x and y
   p<-sqrt((x-5)^2+(y-5)^2)
   -sin(p)/p }
x<-seq(-15,20,by=0.5)
# define a sequence of x and y near the solution
y<-x
z<-outer(x,y,fxy)
# create a 71x71 matrix whose element is fxy(x,y)
persp(x,y,z,theta=-30, phi=30, col="lightblue",
ticktype="detailed")
# persp. plot
```


0.2
0.0
-0.2
-0.4
-0.6
-0.8
z
20
10
0
-10
y
-10
0
10
20
x

- We first create sequence `x` and `y` from -15 to 20 of length 71.
- Then we need to create a 71×71 matrix `z` whose (i,j) element is `fxy(x[i],y[j])`.
- `x`, `y` and `z` are the first three arguments in the function `persp()`.
- The argument `theta` and `phi` in the `persp()` function is the angle of perspective plot.
- Changing these angles will change the orientation of the plot.
- There are many options in `persp()`, see `help(persp)` for details and type `demo(persp)` for demonstration.

Nonlinear least squares using `optim()`

One common optimization problem in statistics is to find parameters in a function to minimize the error sum of squares, usually referred to as the Nonlinear Least Squares (NLS).

Suppose we observe the following points:

x	y
4	0.1957
2	0.1947
1	0.1735
0.5	0.16
0.25	0.0844
0.167	0.0627
0.125	0.0456
0.1	0.0342
0.0833	0.0323
0.0714	0.0235
0.0625	0.0246

Let us define the function:

$$f(x;\theta) = \frac{\theta_1 x(x+\theta_2)}{(x^2+\theta_3 x+\theta_4)}$$

to fit to the data, i.e., we want to find parameters $\theta = (\theta_1, \theta_2, \theta_3, \theta_4)$' such that the error sum of square

$$\sum_{i=1}^{n}[y_i - f(x_i;\theta)]^2$$

is minimized.

First, let us define the following function `fx(x,p)` to fit the data.

```
# define the function to fit, p is a 4x1 parameter vector
fx<-function(x,p){
     (p[1]*x)*(x+p[2])/(x^2+p[3]*x+p[4])
}
```

Next we define the objective function

```
# define objective function, call fx to return error sum
of squares: sum((y-fx)^2)
objF<-function(p,data) {
     # p=parameter vector, data=(xi,yi), two input args
     x<-data[,1]          # save 1st column to x
     y<-data[,2]          # save 2nd column to y
     sum((y-fx(x,p))^2)
     # output error sum of squares
}
```

Finally, we read in the dataset `nls.csv` and use $\theta_0 = (0.25, 0.39, 0.415, 0.39)'$ as the starting value for `optim()`.

```
d<-read.csv("nls.csv")     # read in data to d
p<-c(0.25,0.39,0.415,0.39)# set initial value of parameters
out<-optim(p,objF,data=d)  # call optim with passing d to data
out$par                    # display parameter
[1] 0.1928073 0.1912342 0.1230167 0.1360502
out$value              # display min. objective function value
[1] 0.0003075056
```

Note that we need to include an additional argument `data=d` in `optim()`. This is because the `objF()` takes the parameter `p` as well as `data` as its input arguments.

To plot the fitted curve, we can use

```
# plot fitted curve
x<-seq(0,4,by=0.01)
fit<-fx(x,out$par)
plot(x,fit,type="l",ylab="f(x)",col="blue")
points(d,col="red")
```