# Programming Language

chiapeilin


- 1. Introduction to the R language
  - 1.1 Calculate
  - 1.2 Vectors
  - 1.3 Factors
  - 1.4 Matrices and arrays
  - 1.5 Data frames
  - 1.6 Lists
- 2.Programming statistical graphics
  - 2.1 Bar charts and dot charts
  - 2.2 Pie charts
  - 2.3 Histograms
  - 2.4 Boxplot
  - 2.5 Scatterplots
  - 2.6 QQ plots
  - 2.7 Adding to plots
- 3.Simulation
  - 3.1 Generation of pseudorandom numbers
  - 3.2 Random Variables of Common Distubution
    - Binomial
    - Poisson
    - Exponential
    - Normal
  - 3.3 Advanced simulation methods
    - Rejection sampling
    - Importance sampling
- 4. Computational linear algebra
  - 4.1 Vectors and matrices in R
  - 4.2 Matrix properties
  - 4.3 Matrix inversion
  - 4.4 Eigenvalues and eigenvectors
- 5. Numerical optimization
  - 5.1 The golden section search method
  - 5.2 Newton–Raphson
  - 5.3 Linear programming


# 1. Introduction to the R language

## 1.1 Calculate

basic operations with R

```
5+49
## [1] 54

2-5
## [1] -3

3*15
## [1] 45

1/0
## [1] Inf

0/0
## [1] NaN

Inf-Inf
## [1] NaN

0*Inf
## [1] NaN

#
31%%7
## [1] 3

#
31%/%7
## [1] 4
```

```
x=10:31
mean(x)
## [1] 20.5

var(x)
## [1] 42.16667
42.16667
## [1] 42.16667

sum((x-mean(x))^2)/21
## [1] 42.16667
```

# 1.2 Vectors

A numeric vector is a list of numbers. The c() function is used to collect things together into a vector.

```
x=c(0,10,20)
x
## [1]  0 10 20
```

Extracting elements from vectors

```
x[-3]
## [1]  0 10
```

Operator for producing simple sequences of integers. Patterned vectors can also be produced using the seq()

function as well as the rep()function.

```
rep(10,5)
## [1] 10 10 10 10 10

rep(seq(11,30,by=3),3)
##  [1] 11 14 17 20 23 26 29 11 14 17 20 23 26 29 11 14 17 20 23 26 29

rep(c(2,3,4),each=3)
## [1] 2 2 2 3 3 3 4 4 4
```

# 1.3 Factors

Factors offer an alternative way of storing character data. For example, a factor with four elements and having the two levels, control and treatment can be created using:

```
grp <- c("control", "treatment", "control", "treatment")
grp
## [1] "control"   "treatment" "control"   "treatment"

grp <- factor(grp)
grp
## [1] control   treatment control   treatment
## Levels: control treatment
```

# 1.4 Matrices and arrays

To arrange values into a matrix, we use the matrix() function:

```
m <- matrix(1:6, nrow=2, ncol=3)
m
```

```
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

We can then access elements using two indices.

```
m[1, 2]
## [1] 3


#slect by row
m[1,]
## [1] 1 3 5


#slect by column
m[, 1]
## [1] 1 2
```

# 1.5 Data frames

These are like matrices, but with the columns having their own names. Columns can be of different types from each other. Use the data.frame()function to construct data frames from vectors:

```
colors <- c("red", "yellow", "blue")
numbers <- c(1, 2, 3)
colors.and.numbers <- data.frame(colors, numbers,more.numbers=c(4, 5, 6))
colors.and.numbers
```

```
  colors numbers more.numbers
1    red       1            4
2 yellow       2            5
3   blue       3            6
```

## 1.6 Lists

Data frames are actually a special kind of list, or structure. Lists in R can contain any other objects. You won't often construct these yourself, but many functions return complicated results as lists.

The list()function is one way of organizing multiple pieces of output from functions. For example,

```
x <- c(3, 2, 3)
y <- c(7, 7)
list(x = x, y = y)
```

```
$x
[1] 3 2 3

$y
[1] 7 7
```

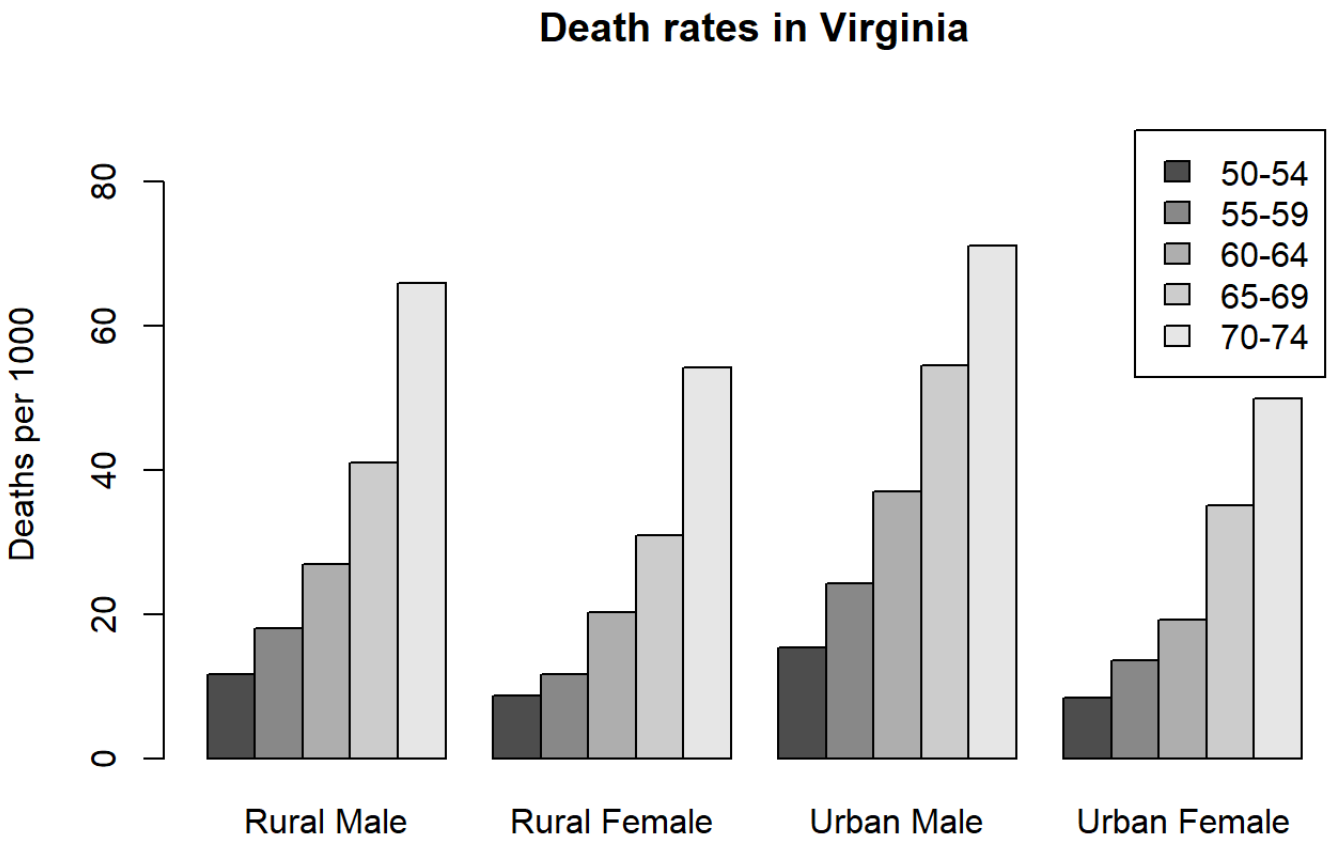
# 2.Programming statistical graphics

## 2.1 Bar charts and dot charts

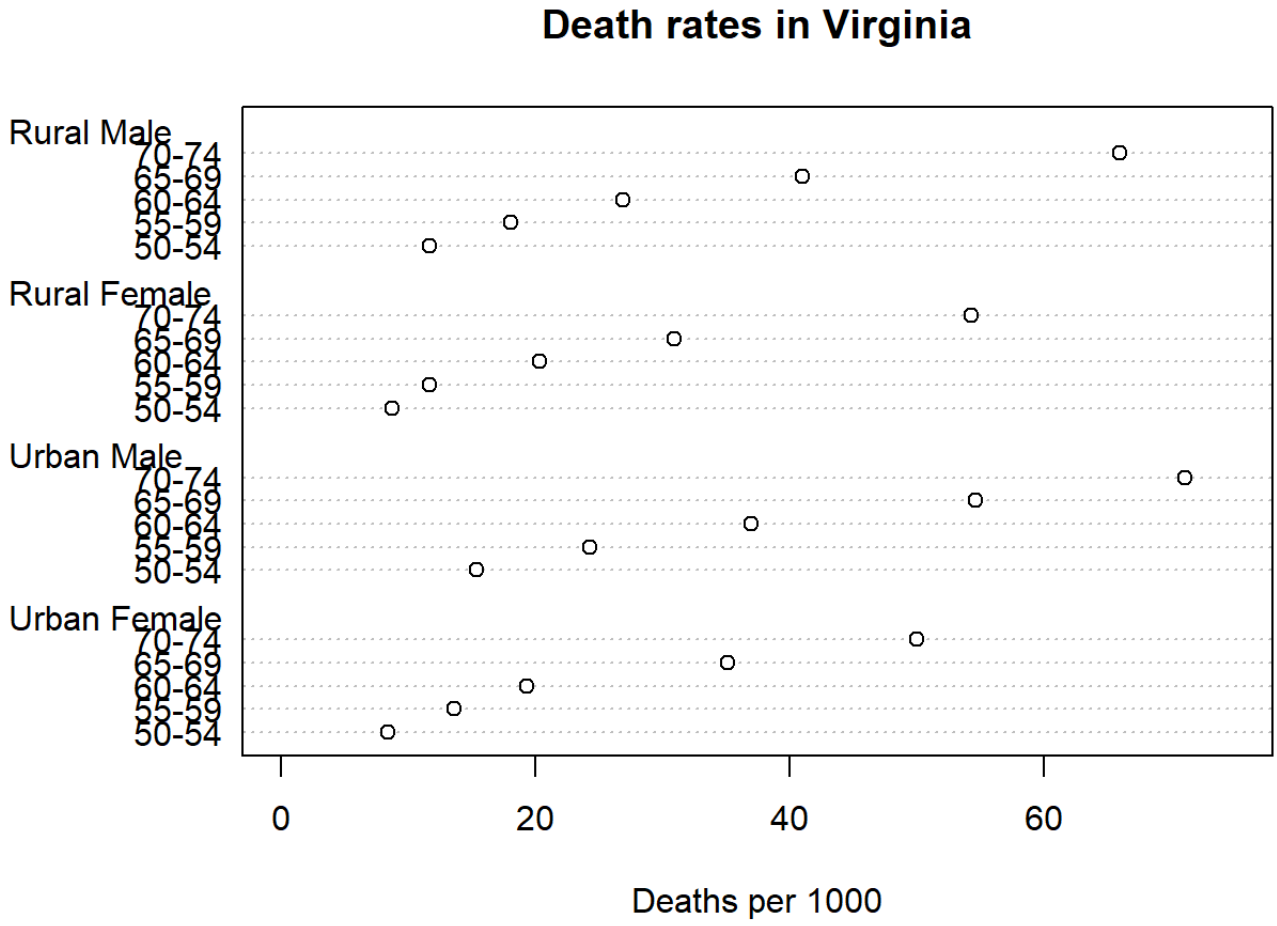
The most basic type of graph is one that displays a single set of numbers.Bar charts and dot charts do this by displaying a bar or dot whose length or position corresponds to the number.

For example, the VADeaths dataset in R contains death rates (number of deaths per 1000 population per year) in various subpopulations within the state of Virginia in 1940.

```
barplot(VADeaths,beside=TRUE,legend=TRUE,ylim=c(0,90),
    ylab="Deaths per 1000",main="Death rates in Virginia")
```



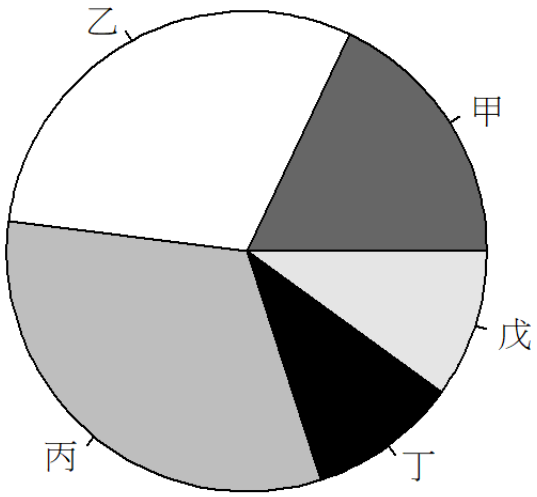

```
dotchart(VADeaths,xlim=c(0,75),xlab="Deaths per 1000",main="Death rates in Virginia")
```



## 2.2 Pie charts

Pie charts display a vector of numbers by breaking up a circular disk into pieces whose angle (and hence area) is proportional to each number.

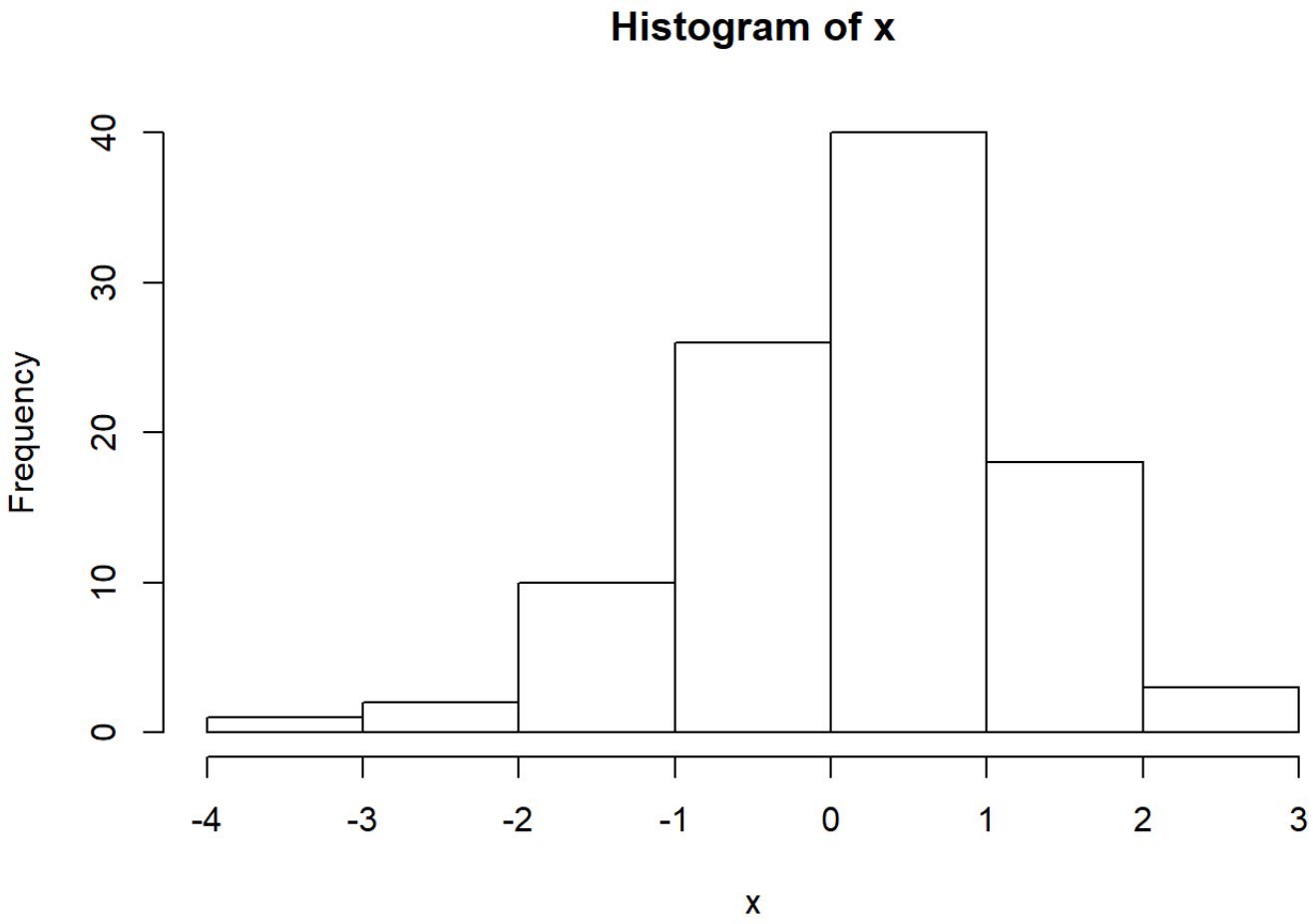
```
groupsizes=c(18,30,32,10,10)
labels=c(" "," "," "," "," ")
pie(groupsizes,labels,col=c("grey40","white","grey","black","grey90"))
```




## 2.3 Histograms

A histogram is a special type of bar chart that is used to showthe frequency distribution of a collection of numbers. Each bar represents the count of x values that fall in the range indicated by the base of the bar.

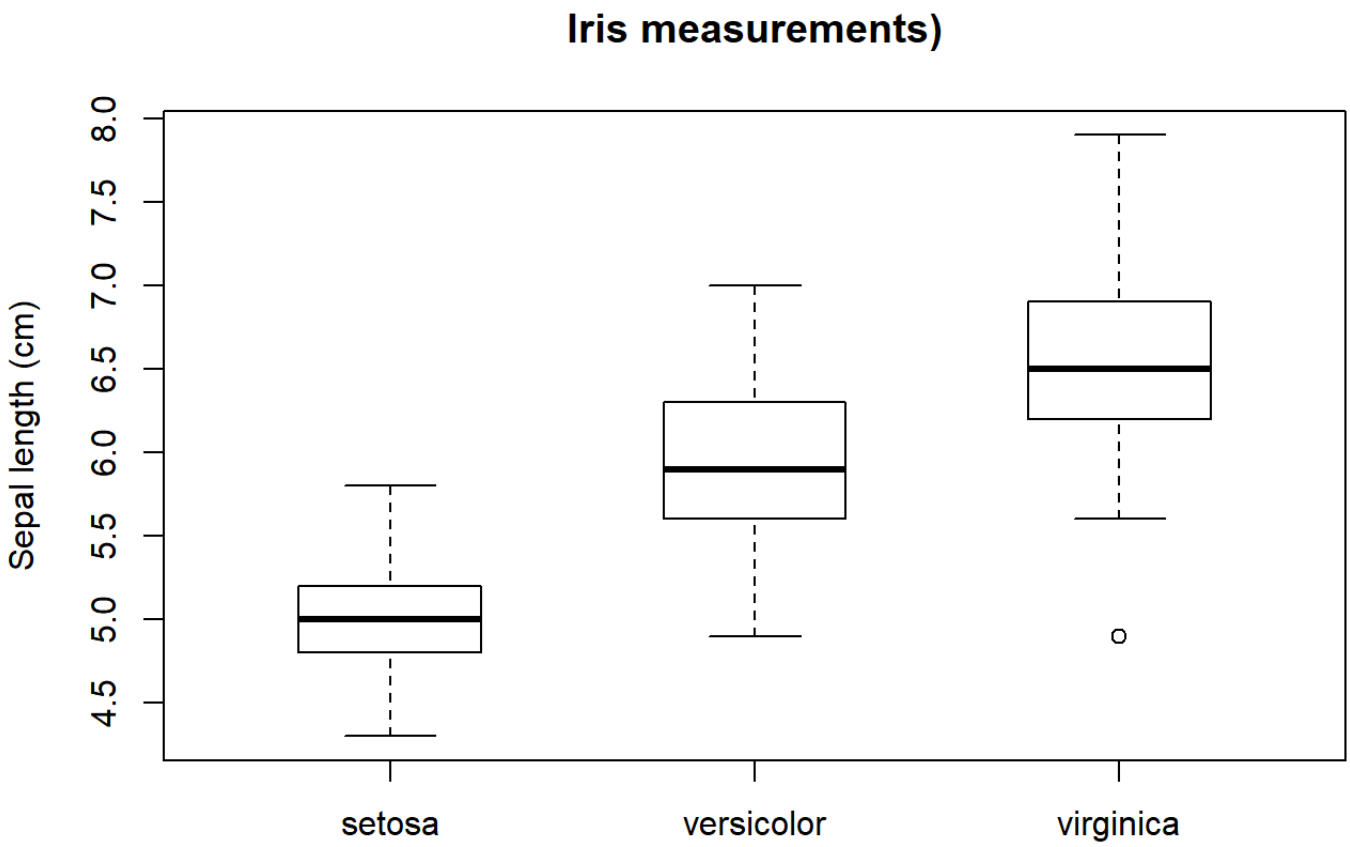
```
x=rnorm(100)
hist(x)
```




## 2.4 Boxplot

A box plot (or “box-and-whisker plot”) is an alternative to a histogram to give a quick visual display of the main features of a set of data.Arectangular box is drawn, together with lines which protrude from two opposing sides.

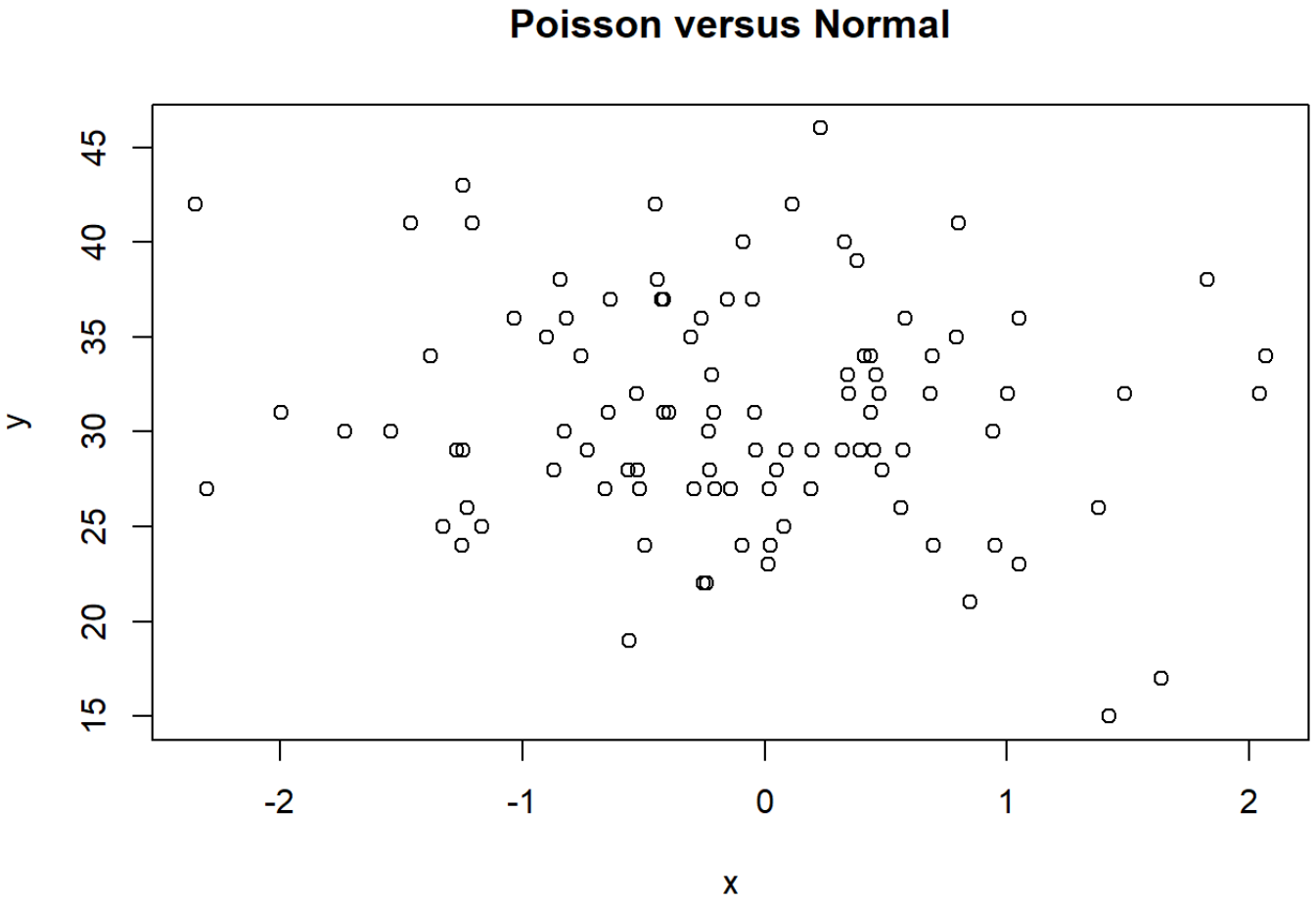
```
boxplot(Sepal.Length~Species,data=iris,ylab="Sepal length (cm)",main="Iris measurements)",boxwex=0.5)
```




## 2.5 Scatterplots

When doing statistics, most of the interesting problems have to do with the relationships between different measurements. To study this, one of the most commonly used plots is the scatterplot.

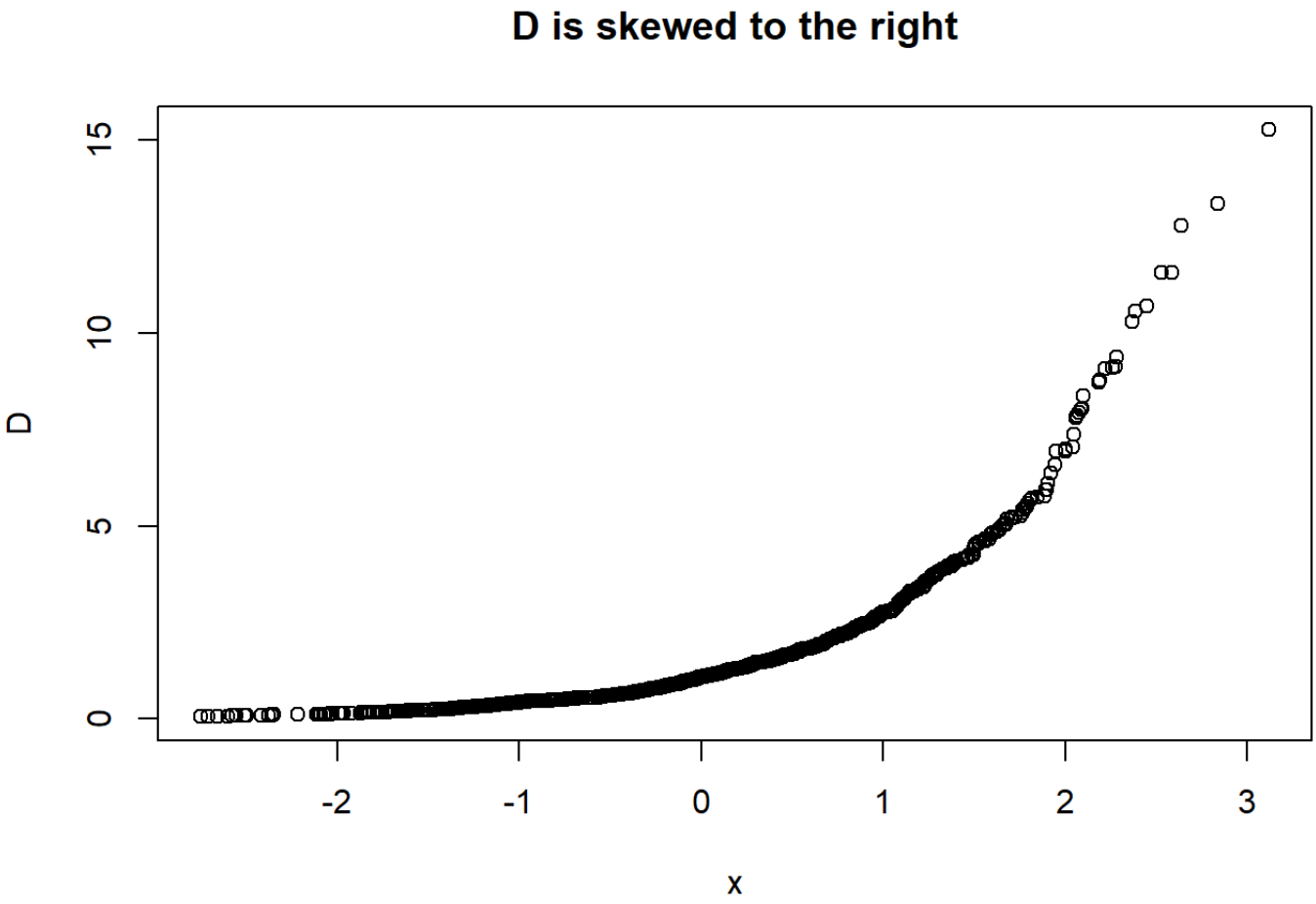
```
x=rnorm(100)
y=rpois(100,30)
plot(x,y,main="Poisson versus Normal")
```




## 2.6 QQ plots

Quantile–quantile plots (otherwise known as QQ plots) are a type of scatterplot **used to compare the distributions of two groups or to compare a sample with a reference distribution.**

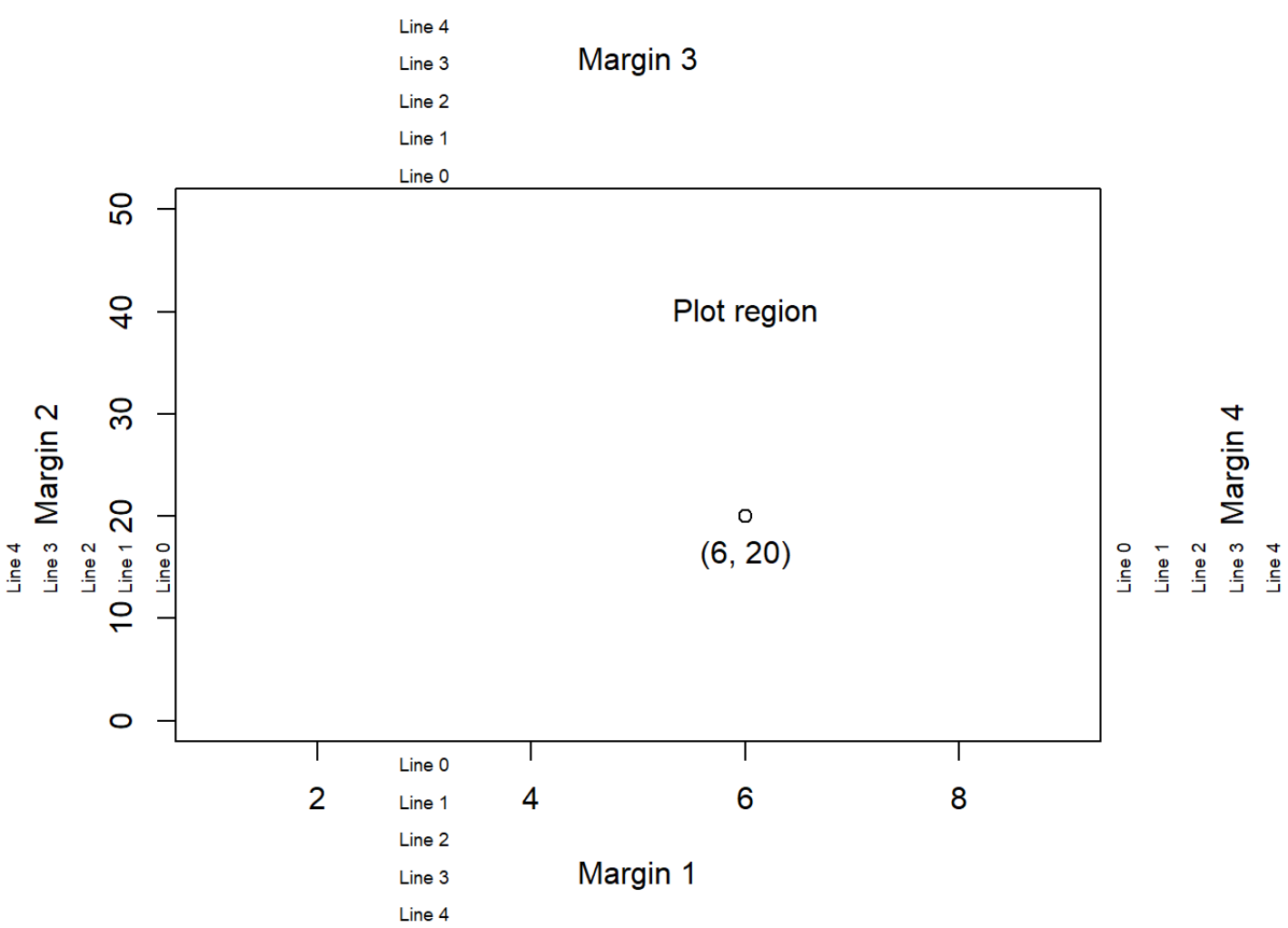
```
x<-rnorm(1000)
A<-rnorm(1000)

D<-exp(rnorm(1000))
qqplot(x,D,main="D is skewed to the right")
```




# 2.7 Adding to plots

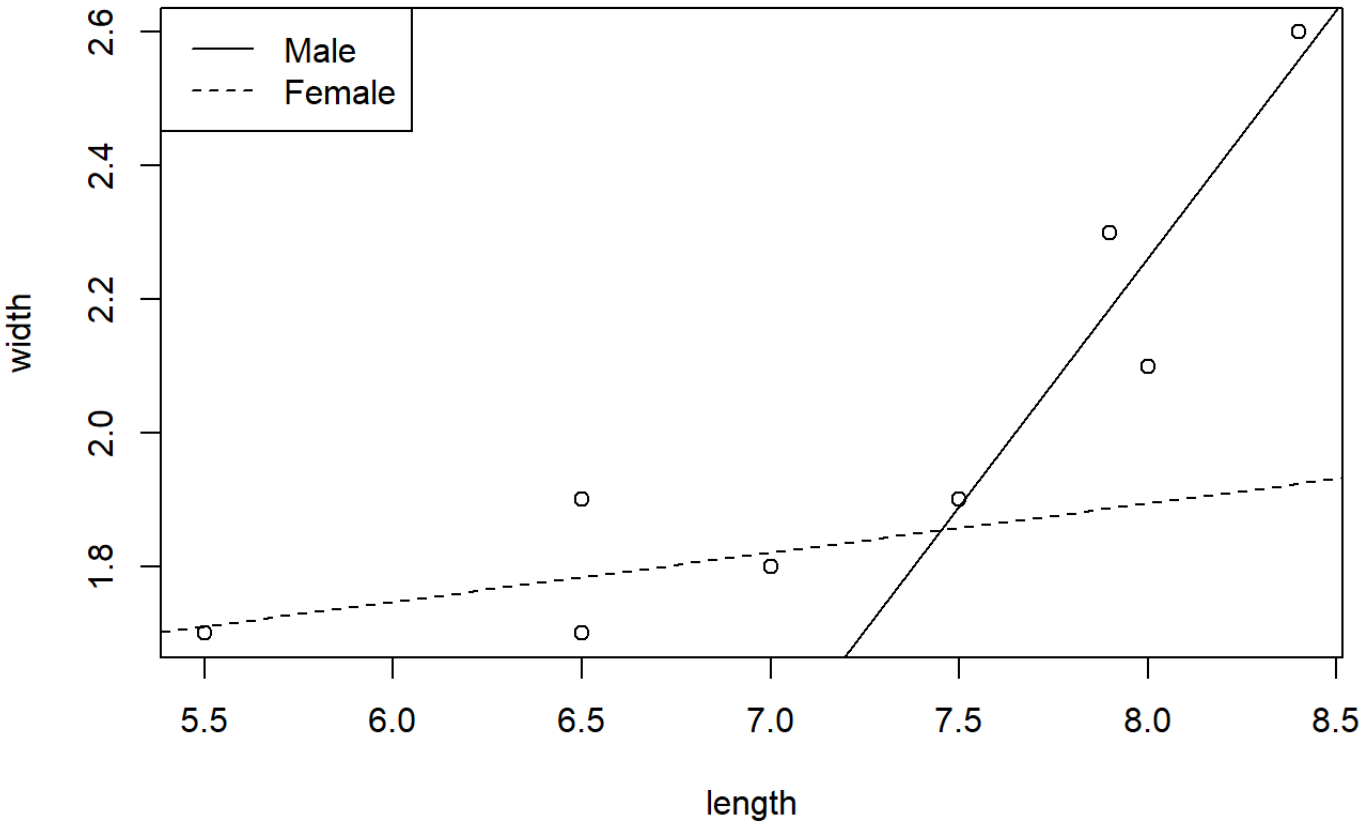
Several functions exist to add components to the plot region of existing graphs.

```
par(mar=c(5, 5, 5, 5) + 0.1)
plot(c(1, 9), c(0, 50), type="n", xlab="", ylab="")
text(6, 40, "Plot region")
points(6, 20)
text(6, 20, "(6, 20)", adj=c(0.5, 2))
mtext(paste("Margin", 1:4), side=1:4, line=3)
mtext(paste("Line", 0:4), side=1, line=0:4, at=3, cex=0.6)
mtext(paste("Line", 0:4), side=2, line=0:4, at=15, cex=0.6)
mtext(paste("Line", 0:4), side=3, line=0:4, at=3, cex=0.6)
mtext(paste("Line", 0:4), side=4, line=0:4, at=15, cex=0.6)
```




The data are stored in a data frame called indexfinger which is displayed below.We can create **a simple scatter plot** of these data **illustrating the relation between length and width**.

```
indexfinger=read.table("D:/temp/indexfinger.txt",header=T)
plot(width~length,data=indexfinger)
abline(lm(width ~length, data=indexfinger, subset=sex=="M"), lty=1)
abline(lm(width ~length, data=indexfinger, subset=sex=="F"), lty=2)
legend("topleft", legend=c("Male", "Female"), lty=1:2)
```




# 3.Simulation

## 3.1 Generation of pseudorandom numbers

We begin our discussion of simulation with a brief exploration of the mechanics of pseudorandom number generation. In particular, we will describe one of the simplest methods for simulating independent uniform random variables on the interval [0,1].

```
# this will store the pseudorandom output
random.number <- numeric(50)
random.seed <- 27218
 for (j in 1:50) {
 random.seed <- (171 * random.seed) %% 30269
 random.number[j] <- random.seed / 30269
 }
random.number
```

```
 [1] 0.76385080 0.61848756 0.76137302 0.19478675 0.30853348 0.75922561
 [7] 0.82757937 0.51607255 0.24840596 0.47741914 0.63867323 0.21312234
[13] 0.44391952 0.91023820 0.65073177 0.27513297 0.04773861 0.16330239
[19] 0.92470845 0.12514454 0.39971588 0.35141564 0.09207440 0.74472232
[25] 0.34751726 0.42545178 0.75225478 0.63556774 0.68208398 0.63636063
[31] 0.81766824 0.82126929 0.43704780 0.73517460 0.71485678 0.24051009
[37] 0.12722587 0.75562457 0.21180085 0.21794575 0.26872378 0.95176583
[43] 0.75195745 0.58472364 0.98774324 0.90409330 0.59995375 0.59209092
[49] 0.24754700 0.33053619
```

Generate five uniform pseudorandom numbers on the interval [0, 1], and 10 uniform such numbers on the interval

[-3,-1].

```
runif(5)
```

```
[1] 0.5572014 0.4849460 0.1118406 0.7478953 0.1610569
```

```
runif(10, min = -3, max = -1)
```

```
 [1] -2.529166 -2.744422 -1.423676 -1.716109 -2.758934 -2.749911 -1.882602
 [8] -2.865225 -1.430868 -2.863913
```

# 3.2 Random Variables of Common Distubution

## Binomial

```
dbinom(x = 4, size = 6, prob = 0.5)
## [1] 0.234375
pbinom(4,6,0.5)
## [1] 0.890625
qbinom(0.89,6,0.5)
## [1] 4
rbinom(24,15,0.1)
##  [1] 0 1 2 0 3 1 1 0 2 4 3 3 2 1 1 0 0 2 2 1 4 1 2 2
```

## Poisson

```
#poisson
dpois(x=3,lambda=0.5)
```

```
[1] 0.01263606
```

```
#poisson process*****
set.seed(1)
N <- rpois(1,1.5*10)
P <- runif(N,max = 10) #    N
sort(P)
```

```
 [1] 0.6178627 1.7655675 2.0168193 2.0597457 3.8410372 6.2911404 6.6079779
 [8] 6.8702285 7.6984142 8.9838968 9.0820779 9.4467527
```

## Exponential

```
pexp(1, rate = 3)
```

```
[1] 0.9502129
```

```
X <- rexp(25, rate = 1.5)
cumsum(X)
```

```
 [1]  0.6377117  0.7357423  1.6628991  2.1709190  2.9959880  5.9452775
 [7]  6.6483063  7.3384689  8.5891590  9.0256568  9.2502791  9.6425989
[13] 11.2189424 11.6468708 11.8429511 12.2201947 12.2909098 12.3305359
[19] 12.7163442 15.3556328 16.1378409 16.8023828 17.7592397 17.7840854
[25] 18.0000922
```

### Normal

```
qnorm(0.95, mean = 2.7, sd = 3.3)
```

```
[1] 8.128017
```

```
rnorm(10, -3, 0.5)
```

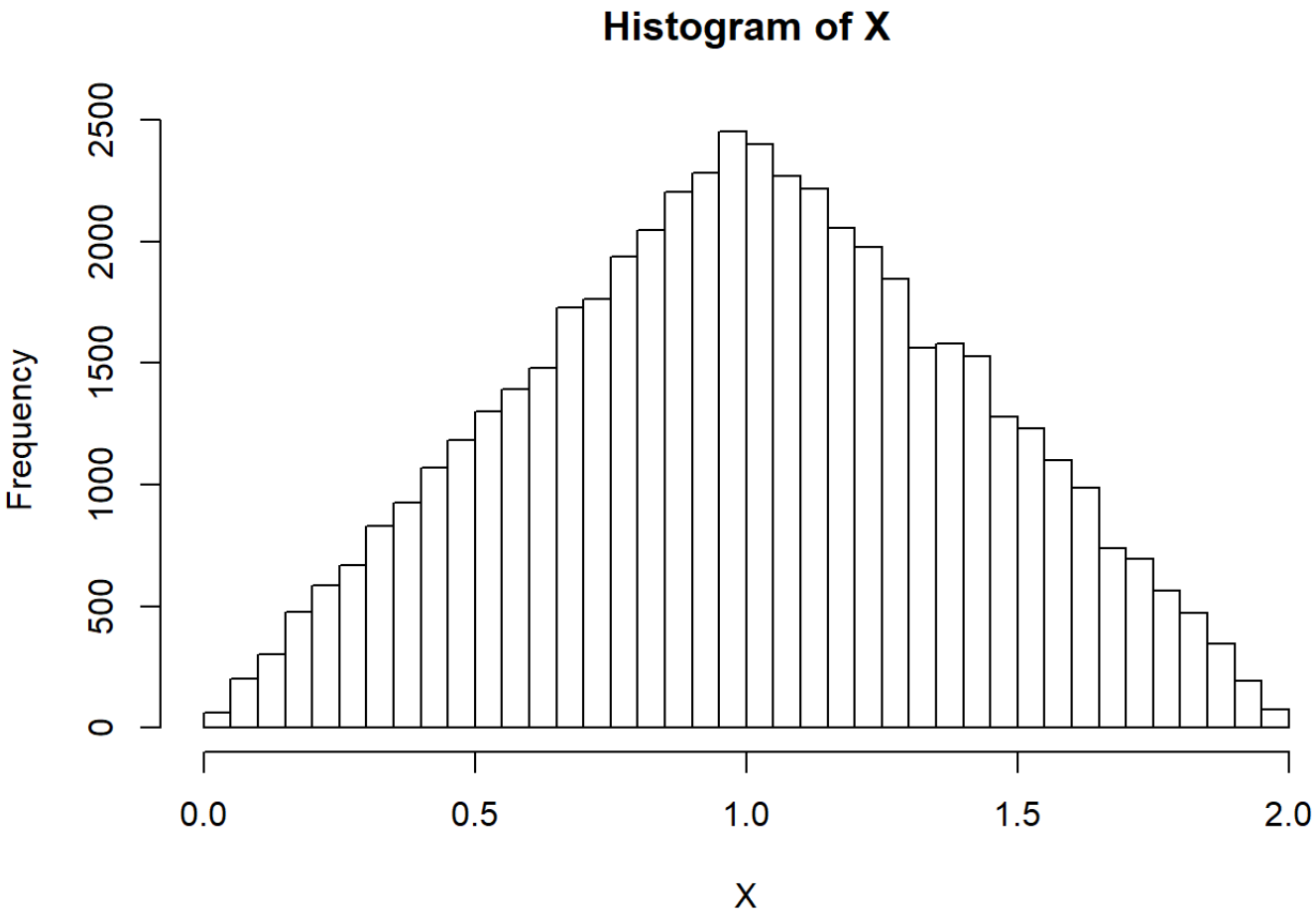
```
 [1] -3.117853 -3.271444 -3.216655 -3.324736 -2.636625 -2.424044 -2.503920
 [8] -3.214757 -2.380848 -3.139673
```

## 3.3 Advanced simulation methods

### Rejection sampling

Sample from a convenient distribution, and select a subsample to achieve the target distribution.

```
set.seed(1)
U1=runif(100000,max=2)
U2=runif(100000)
X=U1[U2<(1-abs(1-U1))]

hist(X,breaks=50)
```




### Importance sampling

Importance sampling is a technique to generate both the sample and the weights randomly, in such a way that weighted averages approximate expected values with respect to a target density function g(x).

```
k=0.45403
g=function(x) 0.5*exp(-(x^1.5))/k

X=rexp(100000)
W=g(X)/dexp(X)

mean0=weighted.mean(X,W)
mean0
## [1] 0.6576605
weighted.mean((X-mean0)^2,W)
## [1] 0.3021293
```

# 4. Computational linear algebra

## 4.1 Vectors and matrices in R

**Numeric matrices** in R are printed as rectangular arrays of numbers, but are **stored internally as vectors** with dimension attributes.

We could also construct the matrix by binding columns together as follows:

```
1/cbind(seq(1, 3), seq(2, 4), seq(3, 5))
```

```
          [,1]      [,2]      [,3]
[1,] 1.0000000 0.5000000 0.3333333
[2,] 0.5000000 0.3333333 0.2500000
[3,] 0.3333333 0.2500000 0.2000000
```

```
matrix(c(1, 2, 3, 1, 4, 9), ncol=2)
```

```
     [,1] [,2]
[1,]    1    1
[2,]    2    4
[3,]    3    9
```

# 4.2 Matrix properties

The **diagonal elements** can be obtained using the**diag()** function, as in:

```
y=cbind(seq(1, 3), seq(2, 4), seq(3, 5))
y
```

```
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    2    3    4
[3,]    3    4    5
```

```
diag(y)
```

```
[1] 1 3 5
```

The diag() function can also be used to turn a vector into a square diagonal matrix whose **diagonal elements correspond to the entries of the given vector**.

```
diag(diag(y))
```

```
     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    3    0
[3,]    0    0    5
```

We can obtain the lower triangular matrix whose nonzero elements match the lower triangular elements of H3 by using

```
Hnew=H3=y
Hnew[upper.tri(H3,diag=TRUE)]=0
Hnew
```

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

The command **X * Y** performs **elementwise multiplication**.

```
x=matrix(1:6,nrow = 3)
y=matrix(c(3,4,5,7,7,7),ncol = 2)

x*y
```

```
     [,1] [,2]
[1,]    3   28
[2,]    8   35
[3,]   15   42
```

In R, the form of **matrix multiplication** can be performed using the operator **%*%**.

```
t(x)%*%y
```

```
     [,1] [,2]
[1,]   26   42
[2,]   62  105
```

# 4.3 Matrix inversion

Matrices are inverted and linear systems of equations are solved using the solve().

```
H3=matrix(c(1,1/2,1/3,1/2,1/3,1/4,1/3,1/4,1/5),nrow = 3)
H3inv <- solve(H3)
H3inv
```

```
     [,1] [,2] [,3]
[1,]    9  -36   30
[2,]  -36  192 -180
[3,]   30 -180  180
```

The function solve(A, b)gives the solution to systems of equations of the form \[ Ax = b \]. For example, let us find x such that H3x = b where H3 is the 3 × 3 Hilbert matrix and b = [1 2 3]T.

```
b=c(1,2,3)
H3=matrix(c(1,1/2,1/3,1/2,1/3,1/4,1/3,1/4,1/5),nrow = 3)
x=solve(H3,b)
x
```

```
[1]   27 -192  210
```

# 4.4 Eigenvalues and eigenvectors

Eigenvalues and eigenvectors can be computed using the function eigen().

```
eigen(H3)
```

```
eigen() decomposition
$values
[1] 1.40831893 0.12232707 0.00268734

$vectors
          [,1]       [,2]       [,3]
[1,] 0.8270449  0.5474484  0.1276593
[2,] 0.4598639 -0.5282902 -0.7137469
[3,] 0.3232984 -0.6490067  0.6886715
```

Extracting eigenvalues and eigenvectors.

```
eigen(H3)$values
```

```
[1] 1.40831893 0.12232707 0.00268734
```

```
eigen(H3)$vectors
```

```
          [,1]       [,2]       [,3]
[1,] 0.8270449  0.5474484  0.1276593
[2,] 0.4598639 -0.5282902 -0.7137469
[3,] 0.3232984 -0.6490067  0.6886715
```

# 5. Numerical optimization

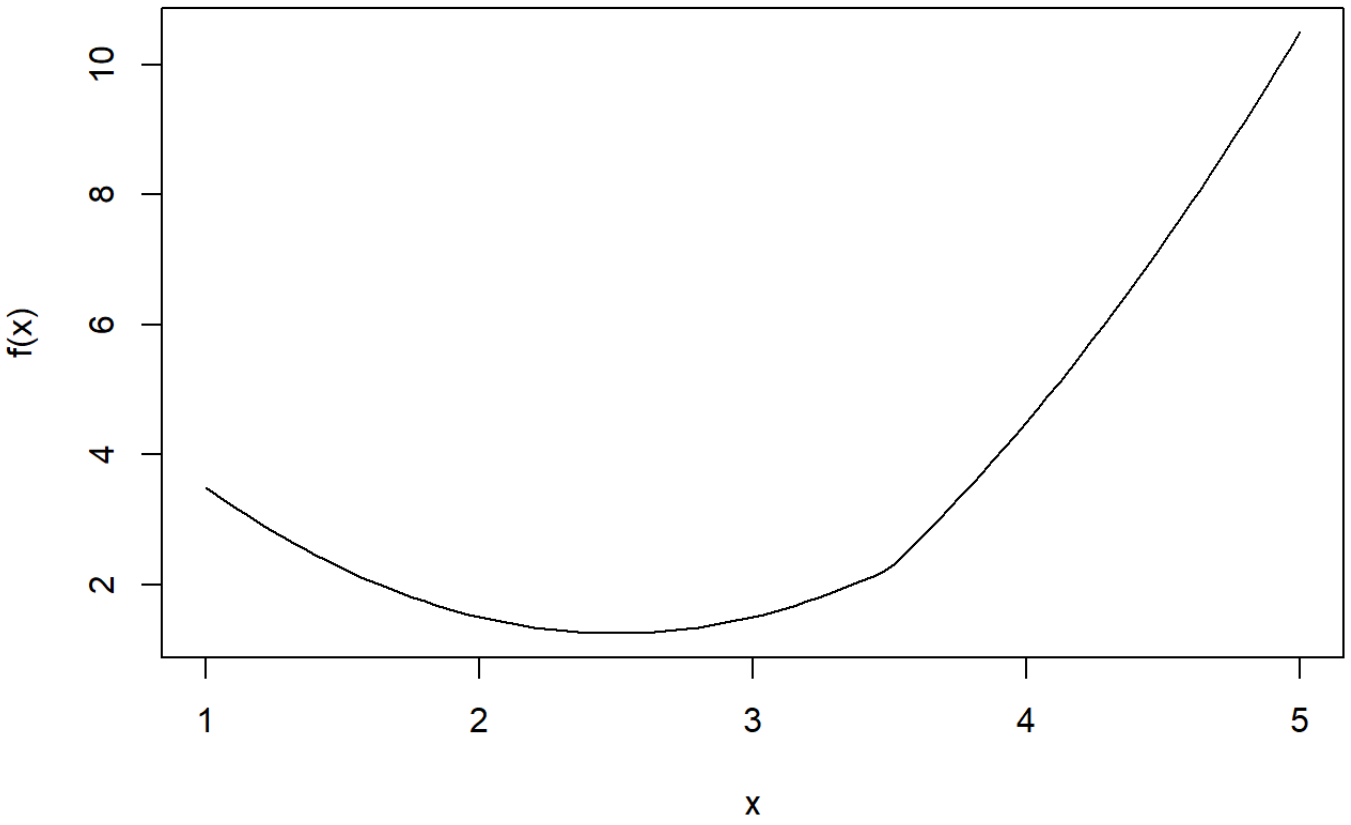
## 5.1 The golden section search method

The golden section search method is a simple way of finding the minimizer of a single-variable function which has a single minimum on the interval [a, b].

```
<Method>
The golden section search method is an iterative method, which may be
outlined as follows:
1. Start with the interval [a, b], known to contain the minimizer.
2. Repeatedly shrink it, finding smaller and smaller intervals [a, b] which still contain the minimizer.
3. Stop when b ??? a is small enough, i.e. when the interval length is less
than a pre-set tolerance.
```

To check that this function has a single minimum in the interval we use the curve() function to plot it:

```
f <- function(x) {
  abs(x - 3.5) + (x - 2)^2
}
curve(f,from=1,to=5)
```




```
golden <- function (f, a, b, tol = 0.0000001)
  {ratio <- 2 / (sqrt(5) + 1)
   x1 <- b - ratio * (b - a)
   x2 <- a + ratio * (b - a)
   f1 <- f(x1)
   f2 <- f(x2)
   while(abs(b - a) > tol) {
     if (f2 > f1) {
         b <- x2
         x2 <- x1
         f2 <- f1
         x1 <- b - ratio * (b - a)
         f1 <- f(x1)
         }
     else {a <- x1
           x1 <- x2
           f1 <- f2
           x2 <- a + ratio * (b - a)
           f2 <- f(x2)
          }
    }
   return((a + b) / 2)
}
golden(f, 1, 5)
```

```
[1] 2.5
```

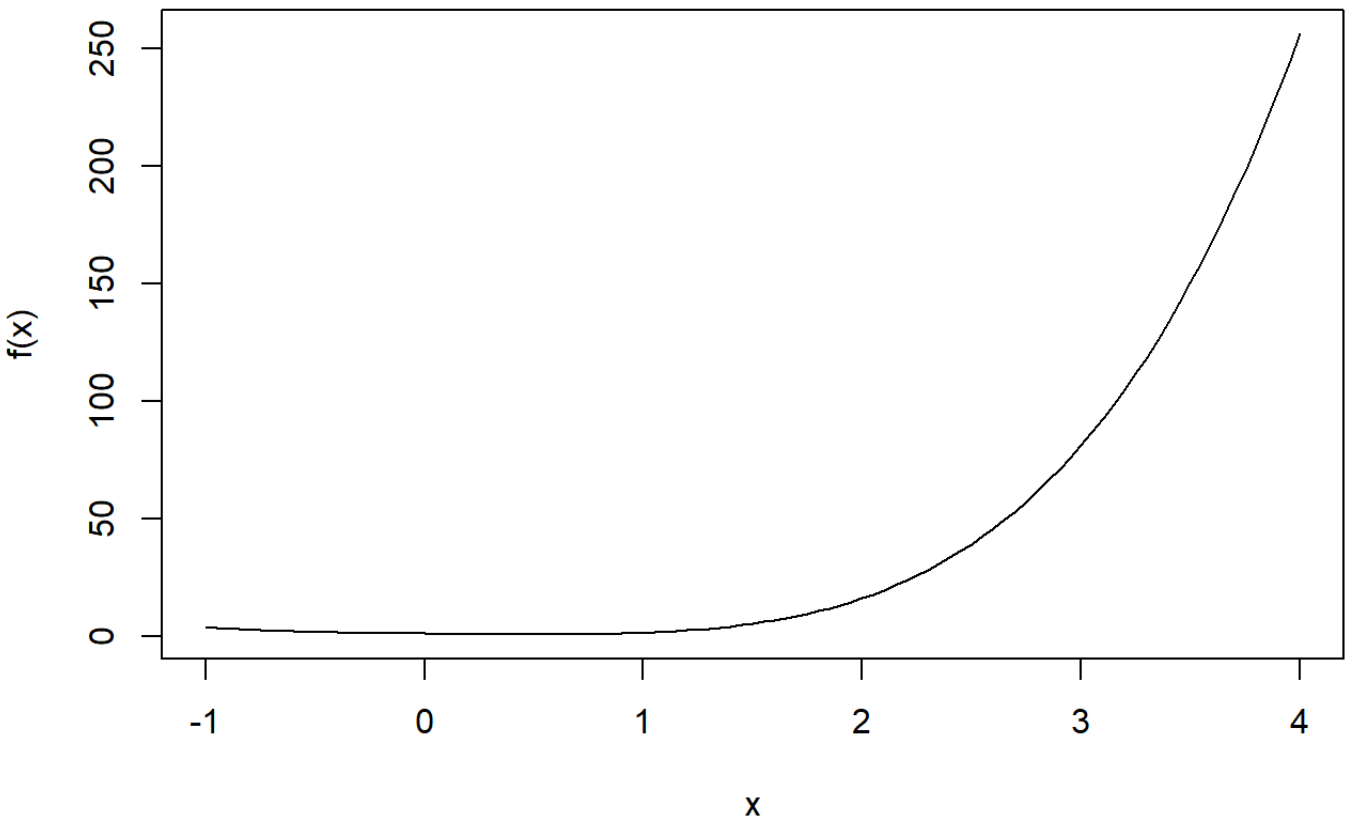
# 5.2 Newton–Raphson

If the function to be minimized has two continuous derivatives and we know how to evaluate them, we can make use of this information to give a faster algorithm than the golden section search.

<Method>
We want to find a minimizer x* of the function f(x) in the interval
[a, b].
1. Provided the minimizer is not at a or b, x??? will satisfy f'(x???) = 0 and f"(x???) > 0.
2. using the Newton–Raphson algorithm to approximate a solution to f'(x???) = 0. Start with an initial guess x0, and co mpute an improved guess using the solution  x1 = x0-f'(x0)/f"(x0).
3. Continue with iterations of the form xn+1 = xn ??? f'(xn)/f"(xn).
This iteration stops when f'(xn) is close enough to 0. Usually, we set a
tolerance ε and stop when |f'(xn)| < ε.

We start by plotting the function to find an initial guess.

```
f <- function(x) exp(-x) + x^4
curve(f, from=-1, to=4)
```



```
f <- function(x) exp(-x) + x^4
fprime <- function(x) -exp(-x) + 4 * x^3
fprimeprime <- function(x) exp(-x) + 12 * x^2
x <- c(0.5, rep(NA, 6))
fval <- rep(NA, 7)
fprimeval <- rep(NA, 7)
fprimeprimeval <- rep(NA, 7)
for (i in 1:6) {
fval[i] <- f(x[i])
fprimeval[i] <- fprime(x[i])
fprimeprimeval[i] <- fprimeprime(x[i])
x[i + 1] <- x[i] - fprimeval[i] / fprimeprimeval[i]
}
data.frame(x, fval, fprimeval, fprimeprimeval)
```

```
          x      fval     fprimeval fprimeprimeval
1 0.5000000 0.6690307 -1.065307e-01       3.606531
2 0.5295383 0.6675070  5.076129e-03       3.953806
3 0.5282544 0.6675038  9.980020e-06       3.938266
4 0.5282519 0.6675038  3.881429e-11       3.938235
5 0.5282519 0.6675038  0.000000e+00       3.938235
6 0.5282519 0.6675038  0.000000e+00       3.938235
7 0.5282519        NA            NA             NA
```

# 5.3 Linear programming

We often need to minimize (or maximize) a function subject to constraints.When the function is linear and the constraints can be expressed as linear equations or inequalities, the problem is called a linear programming problem.

```
<Method>
The idea is to find values of the decision variables x1, x2, . . . , xn
which minimize the objective function C(x), subject to the constraints and nonnegativity conditions.
min C(x) = c1x1 +· · ·+ckxk
```

```
library(lpSolve)
```

```
Warning: package 'lpSolve' was built under R version 3.5.2
```

```
eg.lp <- lp(objective.in=c(5, 8), const.mat=matrix(c(1, 1, 1, 2),
nrow=2), const.rhs=c(2, 3), const.dir=c(">=", ">="))
#the minimum value of the objective function
eg.lp
```

```
Success: the objective function is 13
```

```
#the output tells us that the minimizer
eg.lp$solution
```

```
[1] 1 1
```