Essential R

R. Nazim Khan

July 2021

Contents

1 Introduction

This document as a primer on the `R` computing and statistics environment. It begins with a discussion of data basics in `R` including reading and writing data from and into files. A few important utility functions useful in manipulating data are covered next. A detailed look at graphics in `R` is then followed some basic data analysis. Linear models and model diagnostics are covered next.

This document is an introduction to `R`. You will find more detailed material in online tutorials. Internet searches will provide any required information on `R`, and should be freely and liberally used.

2 Data Basics

In this section we discuss basic data manipulation. We also discuss how to read data into R and write them out to files.

2.1 Data manipulation

The basic data types in R are numerical, characters, strings and nominal/ordinal. Each data type can be stored as constants, vectors, matrices or dataframes. There are several ways to read data into R. Data can be directly entered into R by defining variables.

```
a<-1.2
b<-pi
x<-c(1:5)
y<-c(6:10)
x

## [1] 1 2 3 4 5

y

## [1]  6  7  8  9 10

z<-c(1.1,2.1,3.2,4.5,2.4,3.8,4.3)
z

## [1] 1.1 2.1 3.2 4.5 2.4 3.8 4.3

z[2]

## [1] 2.1

z[1:2]

## [1] 1.1 2.1

z[c(2,4)]

## [1] 2.1 4.5

which(z>4)

## [1] 4 7

z[z<4]

## [1] 1.1 2.1 3.2 2.4 3.8

z[-2]

## [1] 1.1 3.2 4.5 2.4 3.8 4.3

#This lists z without its second element.
```

First the constants $a = 1.2$ and $b = \pi$ have been defined. The vector `x` contains the numbers 1 to 5. Note that the syntax `1:5` indicates integers between 1 to 5 inclusive. The vector z contains real numbers.

```
x*y

## [1]  6 14 24 36 50

x+y

## [1]  7  9 11 13 15

x*z

## Warning in x * z:  longer object length is not a multiple of shorter object
length

## [1]  1.1  4.2  9.6 18.0 12.0  3.8  8.6

y%%2

## [1] 0 1 0 1 0

3%/%2

## [1] 1
```

The symbol + indicated addition, $*$ indicates multiplication while `/` indicated real division, and these are performed term by term. If the objects are not of the same length then the shorter one is re-cycled to match the lengths. The symbol `%%` represents modulo division, while `%/%` represents integer division.

More complicated computations can be performed:

```
(x+y)^2

## [1]  49  81 121 169 225

log(x)

## [1] 0.0000000 0.6931472 1.0986123 1.3862944 1.6094379

log10(x)

## [1] 0.0000000 0.3010300 0.4771213 0.6020600 0.6989700

log2(x)

## [1] 0.000000 1.000000 1.584963 2.000000 2.321928
```

```
logb(x,3)

## [1] 0.0000000 0.6309298 1.0000000 1.2618595 1.4649735

exp(x)

## [1]   2.718282   7.389056  20.085537  54.598150 148.413159

10^x

## [1] 1e+01 1e+02 1e+03 1e+04 1e+05
```

Note that log represents logarithm to the base e.
A large range of mathematical functions is available.

```
sin(pi)

## [1] 1.224606e-16

exp(-1)

## [1] 0.3678794

log(exp(1))

## [1] 1
```

Section 3 contains several function that are used to create and manipulate data.
Note that `R` can also handle complex numbers.

```
x<-(1+2i)
y<-(3+4i)
x+y

## [1] 4+6i

x^2

## [1] -3+4i

x^y

## [1] 0.1290096+0.0339241i

sqrt(x)

## [1] 1.27202+0.786151i

sqrt((-1+0i))

## [1] 0+1i
```

Character types can also be easily handled by R.

```
x<-c("Hummer","Dodge","Bentley")
y<-c("Toyota","Honda","Mitsubishi")
str(x)

##  chr [1:3] "Hummer" "Dodge" "Bentley"
```

Exercises

1. Sales ($millions) for a chain of stores for last year and this year for the month of January, in store correspondence are:

   last years: 1.5, 1.7,2.1,3.4,1.3,2.4,4.5,0.9
   this years: 1.6, 1.8,1.9,3.5,1.1,2.2,4.5, 1.2

   Is there a difference in the mean sales between January last year and this year? Write the R code to compute the appropriate test statistic to test the appropriate hypotheses.

2. Write the R code to compute the variance of the combined sales figures in the previous exercise.

3. Suppose the sales were for a random sample of stores from a franchise. Now compute the appropriate test statistic to determine if the sales have increased from last year.

Matrices are read as follows.

```
x<-matrix(nrow=3,c(1,2,3,4,5,6,7,8,9),byrow=F)
x

##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9

class(x)

## [1] "matrix" "array"

attributes(x)

## $dim
## [1] 3 3

x[1,]
```

```
## [1] 1 4 7

x[,1]

## [1] 1 2 3

summary(x)

##        V1              V2              V3     
##  Min.   :1.0   Min.   :4.0   Min.   :7.0  
##  1st Qu.:1.5   1st Qu.:4.5   1st Qu.:7.5  
##  Median :2.0   Median :5.0   Median :8.0  
##  Mean   :2.0   Mean   :5.0   Mean   :8.0  
##  3rd Qu.:2.5   3rd Qu.:5.5   3rd Qu.:8.5  
##  Max.   :3.0   Max.   :6.0   Max.   :9.0
```

Another very useful function is `gl()`, which creates a factor with the specified number of levels. The basic syntax is

```
Nitrogen<-gl(n= 3, k=10, length = 30, labels = c("Low","Med","High"), ordered = T)
class(Nitrogen)

## [1] "ordered" "factor"

is.factor(Nitrogen)

## [1] TRUE

is.ordered(Nitrogen)

## [1] TRUE
```

where `n` give the number of levels, `k` gives the number of times each is repeated, `length` gives the option of repeating this set, `labels` provides a set of optional levels, and `ordered` specifies whether the factor is ordered.

2.2 Logical comparisons

`R` understands logical comparisons $<, >, <=, >=$, which are applied elementwise. Note that logical equality is $==$ and inequality is $!=$, while & is 'logical and', | is 'logical or'.

```
(1:5) == (5:1)

## [1] FALSE FALSE  TRUE FALSE FALSE

(1:5)>(5:1)
```

```
## [1] FALSE FALSE FALSE  TRUE  TRUE

((1:5)==(5:1))|((1:5)>(5:1))

## [1] FALSE FALSE  TRUE  TRUE  TRUE

((1:5)==(5:1))& ((1:5)<(5:1))

## [1] FALSE FALSE FALSE FALSE FALSE
```

Use `help("!")` to obtain information regarding logical operators. The following functions are also useful.

`any()` Returns `TRUE` if any of the argument satisfies the criteria.

`all()` Returns `TRUE` if all of the argument satisfies the criteria.

`identical()` Returns `TRUE` if the two objects are exactly the same.

`all.equal()` Returns `TRUE` the two objects are (near) equal.

```
identical(sin(pi),0)

## [1] FALSE

all.equal(sin(pi),0)

## [1] TRUE

x<-c(1:4,-1,-3,0)
if(any(x<0)) cat("Some x are negative")

## Some x are negative
```

2.3 Changing the working directory

The working directory can be changed from `File` ⇒ `Change dir...`. This then gives a dialogue box that allows one to browse for the required directory.

2.4 Reading data into `R`

There are various ways to read data into `R` depending on the data format.

1. For files ending in `.R` or `.r` use `source()`.
2. For files ending in `.Rdata` or `.rda` use `load()`.

3. For files ending in `.tab`, `.txt` or `.TXT` use `read.table()`, which produces a dataframe.

4. For files ending in .csv or `.CSV` use `read.table(...,header=TRUE, sep=";")`, which produces a dataframe.

2.4.1 The function `scan`

The function `scan` can be used to read data into `R` from a file. The basic syntax is

```
scan(file = "filename", sep = "",
     skip = 0, nlines = 0, na.strings = "NA")
```

If the filename is “” then the input is from the keyboard (or the `stdin`), and is terminated by a blank line. Only one column of data can be entered this way.

The next example first creates a file of data, then reads it using the `scan` function.

```
cat("1 2 3 4 5","6 7 8 9 10",file="Data/data.txt", sep="\n")
#The file data.txt is saved in the sub-directory or folder Data.
y<-scan("Data/data.txt")
y

##  [1]  1  2  3  4  5  6  7  8  9 10
```

2.4.2 The function `read.table`

This function is used to read a dataframe from a file. The basic syntax is

```
read.table(file, header = FALSE, sep = "", quote = "\",...)
```

Typically the file is a `.txt` or `.csv`. The separator is usually a space `sep=" "` in which case it can be omitted, a tab `sep="\t"`, a comma `sep=","` or a semicolon `sep=";"`. If `header=T` then column names will be read in from the first line of the file. Otherwise column names may be specified by a vector. Similarly, row names may also be specified by a vector.

The examples below illustrate the use of this function.

```
x<-read.table("Data/mydata.txt",sep="\t",header=T, stringsAsFactors = TRUE)
y<-read.table("Data/newdata.txt",header=F,col.names=c("ID", "OH", "HOH", "Charge"))
head(x) #Print the first few lines of x

##   ID  OH   HOH Charge
## 1  1 1.4 120.2      1
## 2  2 2.4  30.5      2
## 3  3 3.4 130.2      3
## 4  3 2.2 110.3      2
## 5  4 2.3 100.2      3
## 6  5 3.2  90.4      3
```

```
head(y)

##   ID  OH   HOH Charge
## 1  1 1.4 120.2      1
## 2  2 2.4  30.5      2
## 3  3 3.4 130.2      3
## 4  3 2.2 110.3      2
## 5  4 2.3 100.2      3
## 6  5 3.2  90.4      3
```

There are related functions `read.csv()`, `read.delim()` and a few variants. Use the `help` function for more details. Excel files can also be read in—see the library `readxl` for details.

2.4.3 Accessing built in data sets

There are several data sets supplied with `R`, and several that come with different packages. To see the list of datasets currently available, use

```
data()
```

All datasets supplied with `R` are directly available by name. To see dataset available within a package, for example `MASS`, use

```
data(package="MASS")
data(Cars93,package="MASS")
```

2.5 Dataframes

Several columns together form a data set, with each row containing a `record`, that is, observations on a single `experimental unit`. In `R` such an object is called a `dataframe`. The matrix class and the dataframe class are very similar in `R` except that some operations can be performed on matrices but not on dataframes. The dataframe will have column names that are used to refer to the variables. In `R` the dataframe may also have row names defined. Row names may be used to refer to each record, and this is meaningful in many contexts. For example, the observations may be demographic information on cities, and the records may be identified by the city names which are stored as row names.

A dataframe is defined by the functions `data.frame()` or `as.data.frame()`. Below we create a simple data frame.

```
Make<-c("Honda","Chevrolet","Ford","Eagle","Volkswagen","Buick","Mitsbusihi","Dodge","Chrys
Model<-c("Civic","Beretta","Escort","Summit","Jetta","Le Sabre","Galant","Grand Caravan","N
Cylinder<-c(rep("V4",5),"V6","V4",rep("V6",3))
Weight<-c(2170,2655,2345,2560,2330,3325,2745,3735,3450,3265)
Mileage<-c(33,26,33,33,26,23,25,18,22,20)
```

```
Type<-c("Sporty","Compact",rep("Small",3),"Large","Compact","Van",rep("Medium",2))
Car<-data.frame(Make,Model,Cylinder,Weight,Mileage,Type)
Car

##          Make         Model Cylinder Weight Mileage    Type
## 1       Honda         Civic       V4   2170      33  Sporty
## 2   Chevrolet       Beretta       V4   2655      26 Compact
## 3        Ford        Escort       V4   2345      33   Small
## 4       Eagle        Summit       V4   2560      33   Small
## 5  Volkswagen         Jetta       V4   2330      26   Small
## 6       Buick      Le Sabre       V6   3325      23   Large
## 7  Mitsbusihi        Galant       V4   2745      25 Compact
## 8       Dodge Grand Caravan       V6   3735      18     Van
## 9    Chrysler    New Yorker       V6   3450      22  Medium
## 10      Acura        Legend       V6   3265      20  Medium

Car[1,]

##    Make Model Cylinder Weight Mileage   Type
## 1 Honda Civic       V4   2170      33 Sporty

Car[,1]

##  [1] "Honda"      "Chevrolet"  "Ford"       "Eagle"      "Volkswagen" "Buick"
##  [7] "Mitsbusihi" "Dodge"      "Chrysler"   "Acura"

Car$Model

##  [1] "Civic"         "Beretta"       "Escort"        "Summit"        "Jetta"
##  [6] "Le Sabre"      "Galant"        "Grand Caravan" "New Yorker"    "Legend"

table(Car$Type)

##
## Compact   Large  Medium   Small  Sporty     Van
##       2       1       2       3       1       1
```

The proportion of cars of each type can be produced by:

```
table(Car$Type)/length(Car$Type)

##
## Compact   Large  Medium   Small  Sporty     Van
##     0.2     0.1     0.2     0.3     0.1     0.1
```

Cross tables can also be produced easily.

```
table(Car$Make, Car$Type)

##
##              Compact Large Medium Small Sporty Van
##   Acura            0     0      1     0      0   0
##   Buick            0     1      0     0      0   0
##   Chevrolet        1     0      0     0      0   0
##   Chrysler         0     0      1     0      0   0
##   Dodge            0     0      0     0      0   1
##   Eagle            0     0      0     1      0   0
##   Ford             0     0      0     1      0   0
##   Honda            0     0      0     0      1   0
##   Mitsbusihi       1     0      0     0      0   0
##   Volkswagen       0     0      0     1      0   0
```

The dataframe can also be sorted by any variable. For example, below the dataframe is sorted by weight.

```
i<-order(Car$Weight);i

##  [1]  1  5  3  4  2  7 10  6  9  8

Car[i,]

##          Make         Model Cylinder Weight Mileage    Type
## 1       Honda         Civic       V4   2170      33  Sporty
## 5  Volkswagen         Jetta       V4   2330      26   Small
## 3        Ford        Escort       V4   2345      33   Small
## 4       Eagle        Summit       V4   2560      33   Small
## 2   Chevrolet       Beretta       V4   2655      26 Compact
## 7  Mitsbusihi        Galant       V4   2745      25 Compact
## 10      Acura        Legend       V6   3265      20  Medium
## 6       Buick      Le Sabre       V6   3325      23   Large
## 9    Chrysler    New Yorker       V6   3450      22  Medium
## 8       Dodge Grand Caravan       V6   3735      18     Van
```

Note that in the Windows platform, dataframes can be accessed directly through `Edit`→Data editor *ldots*, or the command `data1<-edit(data.frame())`. A spreadsheet is produced in which data can be entered directly. However, this requires care to ensure that the data types are defined correctly. This is especially critical for factors.

2.6 Writing data to files

The function `write(x, file = "data",ncolumns = if(is.character(x)) 1 else 5, append = FALSE, sep = " ")` writes `x`, usually a matrix, to a file. If `append=TRUE)` then the data is appended at the end of the file. An example is given below.

```
Height<-c(1.6,1.7,1.8,1.9,1.6,1.8)
Weight<-c(65.4,67.3,70.5,56.4,67.4,73.4)
bmi<-cbind(Height,Weight)
write(bmi,"Data/bmi.txt",sep="\t")
```

A related function is

`write.table(x, file = "", append = FALSE, sep = " ",dec = ".", row.names = TRUE, col.names = TRUE)`

used to write dataframes to file. If the object to be written is not a data frame then it is coerced to be one. Below the dataframe `car` is written to a file, and then read back again.

```
write.table(Car,"Data/Car.txt",sep=" ")
newcar<-read.table("Data/Car.txt",header=T,sep=" ")
head(Car)

##          Make    Model Cylinder Weight Mileage    Type
## 1       Honda    Civic       V4   2170      33  Sporty
## 2   Chevrolet  Beretta       V4   2655      26 Compact
## 3        Ford   Escort       V4   2345      33   Small
## 4       Eagle   Summit       V4   2560      33   Small
## 5  Volkswagen    Jetta       V4   2330      26   Small
## 6       Buick Le Sabre       V6   3325      23   Large

head(newcar)

##          Make    Model Cylinder Weight Mileage    Type
## 1       Honda    Civic       V4   2170      33  Sporty
## 2   Chevrolet  Beretta       V4   2655      26 Compact
## 3        Ford   Escort       V4   2345      33   Small
## 4       Eagle   Summit       V4   2560      33   Small
## 5  Volkswagen    Jetta       V4   2330      26   Small
## 6       Buick Le Sabre       V6   3325      23   Large
```

2.7 `R` script files

When writing a long piece of code it is easy to make mistakes, and the whole code needs to be typed in again. In addition, some code may be used again. One should *always* write the code in a file and then run it in `R`. Simply access the `R Editor` from `File` ⇒ `New File` ⇒ `R Script`and saving it with a `.R` extension by default. Such a file can then be run in `R` , and also used as a template to write new code.

When building a long piece of code, one should enter the code a few lines at a time and test it before continuing.

Exercises

1. The formula for computing the interest paid on a loan of \$1,000 compounded annually if the nominal annual rate is 7.5% is

$$\text{Interest} = 1000\left((1+0.075)^5 - 1\right)$$

   (a) Write the `R` code for the above expression.

   (b) What is the result of your computation above?

   (c) Modify the expression to determine the amount of interest paid if the nominal annual rate is 3.5%.

   (d) What happens if the exponent 5 is replaced by `(1:10)`?

2. Write `R` code that prints out the perfect squares up to and including 100.

3 Utility Functions

R has several useful functions for data manipulation. These can be used to re-organise data, interrogate data structures or obtain summaries. A brief list with descriptions is given below. Further functions will be covered in the description of the key function. More details may be obtained from other references or the R help.

`rbind(x,y,...)` Stands for row bind. This function binds together rows into a single matrix.

```
x<-c(1:5)
y<-c(6:10)
z<-rbind(x,y)
z

##   [,1] [,2] [,3] [,4] [,5]
## x    1    2    3    4    5
## y    6    7    8    9   10
```

`cbind(x,y,...)` Stands for column bind. This function binds together columns into a single matrix.

```
x<-c(1:5)
y<-c(6:10)
z<-cbind(x,y)
z

##      x  y
## [1,] 1  6
## [2,] 2  7
## [3,] 3  8
## [4,] 4  9
## [5,] 5 10
```

`str(x)` Reveals the structure of an object.

```
str(z)

##  int [1:5, 1:2] 1 2 3 4 5 6 7 8 9 10
##  - attr(*, "dimnames")=List of 2
##   ..$ : NULL
##   ..$ : chr [1:2] "x" "y"

colnames(z)

## [1] "x" "y"
```

The object `z` contains 5 rows and 2 columns and contains data of type integer. The data is then listed columnwise. There are no attributes of the data, no row names, and the column names are listed.

`matrix(x,...)` Forms a matrix of the given dimensions from the given data.

```
A<-matrix(1:9,byrow=T,nrow=3)
A

##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
## [3,]    7    8    9

B<-matrix(1:9,byrow=F,nrow=3)
B

##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9

t(A) #Transpose  of A.

##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
```

`data.frame(x,...)` Forms a dataframe from given objects. The objects need to be of the same length.

```
x<-c(1.0,2.0,3.0)
y<-c(4.0,5.0,6.0)
z<-data.frame(cbind(x,y))
z

##   x y
## 1 1 4
## 2 2 5
## 3 3 6

colnames(z)
```

```
## [1] "x" "y"
```

```
attributes(z)
```

```
## $names
## [1] "x" "y"
##
## $class
## [1] "data.frame"
##
## $row.names
## [1] 1 2 3
```

Note that by default the names 1,2,3 have been assigned to the rows of `z`. We can explicitly specify rownames for the dataframe:

```
row.names(z)<-c("length","width","height")
z
```

```
##        x y
## length 1 4
## width  2 5
## height 3 6
```

```
attributes(z)
```

```
## $names
## [1] "x" "y"
##
## $class
## [1] "data.frame"
##
## $row.names
## [1] "length" "width"  "height"
```

`length(x)` Returns the length of a vector.

```
x<-c(1:5)
y<-c(6:10)
z<-data.frame(x,y)
length(x)
```

```
## [1] 5
```

```
length(z)

## [1] 2

length(z[,1])

## [1] 5
```

`mean(x)` Computes the means of the columns of an object.

```
mean(x, na.rm = TRUE)

## [1] 3

mean(y)

## [1] 8

colMeans(z)

## x y
## 3 8
```

`var(x)` Computes the covariance matrix of the columns of an object.

```
var(x)

## [1] 2.5

var(y)

## [1] 2.5

var(z)

##     x   y
## x 2.5 2.5
## y 2.5 2.5
```

`sd(x)` Computes the standard deviation of the columns of an object.

```
sd(x)

## [1] 1.581139

library(matrixStats)
colSds(as.matrix(z))

## [1] 1.581139 1.581139
```

`min(x)` Computes the minimum value of a vector.

`max(x)` Computes the maximum value of a vector.

`range(x)` Computes the range of a vector.

`summary(x)` Computes the six-point summary statistics of an object.

```
summary(x)

##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
##       1       2       3       3       4       5

summary(z)

##        x           y       
##  Min.   :1   Min.   : 6  
##  1st Qu.:2   1st Qu.: 7  
##  Median :3   Median : 8  
##  Mean   :3   Mean   : 8  
##  3rd Qu.:4   3rd Qu.: 9  
##  Max.   :5   Max.   :10
```

`gl()` Generates patterned data, usually for levels of a factor.

Usage: gl(n, k, length = n*k, labels = 1:n, ordered = FALSE) where

n	: an integer giving the number of levels.
k	: an integer giving the number of replications.
length	: an integer giving the length of the result.
labels	: an optional vector of labels for the resulting factor levels.
ordered	: a logical indicating whether the result should be ordered or not.

```
x<-gl(2,2)
x

## [1] 1 1 2 2
## Levels: 1 2

y<-gl(3,1,12)
y

##  [1] 1 2 3 1 2 3 1 2 3 1 2 3
## Levels: 1 2 3

z<-gl(3,2,12)
z

##  [1] 1 1 2 2 3 3 1 1 2 2 3 3
## Levels: 1 2 3

w<-gl(3,2,12,c("low","med","high"),T)
w

##  [1] low  low  med  med  high high low  low  med  med  high high
## Levels: low < med < high

class(w)

## [1] "ordered" "factor"

attributes(w)

## $levels
## [1] "low"  "med"  "high"
##
## $class
## [1] "ordered" "factor"
```

`rep()` Generates patterned data. Can produce the same output at `gl`, except the output cannot be ordered within this function.

Usage : rep(x, times=1, length.out = NA, each=1) where

x : an object indicating the objects to be repeated.

times	: a vector giving the number of times to repeat each element if of length length(x), or to repeat the whole vector if of length 1.
length.out	: non-negative integer. The desired length of the output vector. Ignored if NA or invalid
each	: non-negative integer. Each element of x is repeated each times. Treated as 1 if NA or invalid.

```
options(width=80)
#First set the width so that the output fits on a page.
rep(1:3,2)

## [1] 1 2 3 1 2 3

rep(1:3,each=2)

## [1] 1 1 2 2 3 3

x<-c("low","med","high")
f<-rep(x,times=3,each=2)
factor(f)

##  [1] low  low  med  med  high high low  low  med  med  high high low  low  med
## [16] med  high high
## Levels: high low med

ordered(f)

##  [1] low  low  med  med  high high low  low  med  med  high high low  low  med
## [16] med  high high
## Levels: high < low < med

f

##  [1] "low"  "low"  "med"  "med"  "high" "high" "low"  "low"  "med"  "med"
## [11] "high" "high" "low"  "low"  "med"  "med"  "high" "high"
```

`paste(...,sep=" ", collapse=NULL)` Concatenate vectors after converting to character. Very useful in generating labels or names. The example below generates the names `A1` to `A6`.

```
paste("A", 1:6, sep = "")

## [1] "A1" "A2" "A3" "A4" "A5" "A6"
```

`tapply()` Applies a function to a ragged array to groups defined a set of equal length factors. The `tapply` function is useful when we need to break up a vector into groups defined by some classifying factor, compute a function on the subsets, and return the results in a convenient form. See also `lapply()`, `sapply()`, `apply()`, .

Usage : `tapply(x, by, FUN,...)` where

`x` :	a dataframe containing one or more variables to which FUN is to be applied.
`by` :	a list of equal-length factors by which to apply FUN.
`FUN` :	the function to apply.

```
medical.example <-
    data.frame(patient = 1:100,
               age = rnorm(100, mean = 60, sd = 12),
               treatment = gl(2, 50,
                 labels = c("Treatment", "Control")))
baseball.example <-
    data.frame(team = gl(5, 5,
                 labels = paste("Team", LETTERS[1:5])),
               player = sample(letters, 25),
               batting.average = runif(25, .200, .400))
summary(baseball.example)

##      team      player           batting.average
##  Team A:5   Length:25          Min.   :0.2192
##  Team B:5   Class :character   1st Qu.:0.2668
##  Team C:5   Mode  :character   Median :0.2826
##  Team D:5                      Mean   :0.2955
##  Team E:5                      3rd Qu.:0.3424
##                                Max.   :0.3936

tapply(baseball.example$batting.average, baseball.example$team,mean)

##    Team A    Team B    Team C    Team D    Team E
## 0.3012721 0.2935748 0.2890553 0.2670858 0.3266930

summary(medical.example)
```

```
##     patient              age              treatment
##  Min.   :  1.00   Min.   :32.70   Treatment:50
##  1st Qu.: 25.75   1st Qu.:48.85   Control  :50
##  Median : 50.50   Median :59.40
##  Mean   : 50.50   Mean   :59.11
##  3rd Qu.: 75.25   3rd Qu.:66.68
##  Max.   :100.00   Max.   :97.52

tapply(medical.example$age, medical.example$treatment, mean)

## Treatment   Control
##  60.70302  57.51905
```

You can even specify multiple factors as the grouping variable, for example treatment and sex, or team and handedness.

```
height<-rnorm(100,1.6,0.5)
weight<-rnorm(100,60,5)
sex<-rep(c("M","F"),each=50,times=1)
sex

##   [1] "M" "M" "M" "M" "M" "M" "M" "M" "M" "M" "M" "M" "M" "M" "M" "M" "M" "M"
##  [19] "M" "M" "M" "M" "M" "M" "M" "M" "M" "M" "M" "M" "M" "M" "M" "M" "M" "M"
##  [37] "M" "M" "M" "M" "M" "M" "M" "M" "M" "M" "M" "M" "M" "M" "F" "F" "F" "F"
##  [55] "F" "F" "F" "F" "F" "F" "F" "F" "F" "F" "F" "F" "F" "F" "F" "F" "F" "F"
##  [73] "F" "F" "F" "F" "F" "F" "F" "F" "F" "F" "F" "F" "F" "F" "F" "F" "F" "F"
##  [91] "F" "F" "F" "F" "F" "F" "F" "F" "F" "F"

treatment<-rep(c("A","B"),each=10,times=5)
cohort<-data.frame(height,weight,sex,treatment)
tapply(cohort$weight,list(cohort$sex,cohort$treatment),mean)

##          A        B
## F 61.34534 60.58656
## M 60.17245 61.17765
```

`apply(X, MARGIN, FUN,...)` where

X is an array or matrix

`MARGIN` is a vector giving the subscripts which the function will be applied over. E.g., for a matrix 1 indicates rows, 2 indicates columns, `c(1, 2)` indicates rows and columns. Where `X` has named `dimnames`, it can be a character vector selecting dimension names.

FUN is the function to be applied.

For example, iris3 is a $50 \times 4 \times 3$ array of four measurements on 50 specimens of each of three species of iris flower. Suppose we want the means for each variable by species. Then we need to provide apply() with MARGIN=c(2,3), which refers to the second dimension (measurements) and the third dimension (species).

```
apply(iris3,c(2,3),mean,trim=0.1)

##          Setosa Versicolor Virginica
## Sepal L. 5.0025     5.9375    6.5725
## Sepal W. 3.4150     2.7800    2.9625
## Petal L. 1.4600     4.2925    5.5100
## Petal W. 0.2375     1.3250    2.0325

#Note that trim=0.1 is an argument to mean(). So this shows how to pass arguments to
```

If we want overall means of the four measurements, then we can use

```
apply(iris3,2,mean)

## Sepal L. Sepal W. Petal L. Petal W.
## 5.843333 3.057333 3.758000 1.199333
```

colMeans(), rowMeans(), colSums(), rowSums() Performs the appropriate operation on a given object.

```
colMeans(cbind(height,weight))

##    height    weight
##  1.525252 60.732303
```

rm(x,y,...) Removes or deletes the objects.

3.1 Exercises

1. Generate the sequence 1 2 3 repeated 10 times.
2. Generate a sequence of ten 1's followed by ten 2's, followed by ten 3's, and repeat this whole sequence three times.
3. What is the output of the following code.

```
rep(c(rep(c(1,3,5),2),rep(seq(2,6,2),2)),3)
```

4. An engineer is designing a battery for use in a device that will be subjected to some extreme variations in temperature. He has three possible choices for the plate material. For testing purposes he selects three temperatures. Four batteries are tested at each combination of plate material and temperature and the tests are run in random order. The battery life (hours) under each set of conditions is given in Table 3.1.

	Temperature (°C)					
Material	-10		20		55	
1	130	155	34	40	20	70
	74	180	80	75	82	58
2	150	188	136	122	25	70
	159	126	106	115	58	45
3	138	110	174	120	96	104
	168	160	150	139	82	60

Table 3.1: Life (in hours) data for the battery design example.

   (a) Examine the data and report your observations.
   (b) Write the `R` code to read this data into `R`.
   (c) Find the summary statistics for this data. First think about what sort of statistics you should be interested in.
   (d) Save this code for use in later sections.

5. Consider the following grouped data on seatbelt use and the severity of injury in an accident.

	worn	not worn	unknown
fatal	35	6	15
severe	1142	48	328
minor	7969	76	764
unknown	11404	24	38570

   (a) Enter the data into `R` using the variables `Injury, SeatBelt` and `Frequency`.
   (b) Now we want to create date that contains one record for each case. That is, we need to create 35 entries corresponding to a fatal injury where the seat belt was worn, 6 for when the seat belt was worn, and 15 for unknown. Similarly for the other levels of injuries. Write a short (2 lines!) of `R` code to achieve this, and test your code (for example, by producing a table from your new data).

4 R Programming

In this section we cover some basics of writing R functions and programming. If a particular calculation will be repeated then it is useful to define a function that performs that calculation. Similarly, larger calculations may be written as an R program. R has several built in functions, such as `mean()`, `sd()`, `var()`, `plot()`, ad some of these have been seen earlier.

In this section we first consider some program flow control constructs, followed by writing R functions and finally some simple R programs.

4.1 Flow control structures

Control of flow in a program is determined by some logical statement, which determines the next statement to be executed. The additional logical operators & and || are useful for constructing logical statements. Unlike & and |, which operate componentwise on vectors, these operate on scalar logical expressions.

Below is a list of the control structures inR.

The `if` statement The basic syntax is

```
if(cond) true.branch
```

or

```
if(condition) true.branch else false.branch
```

In the first form, if `condition` is true then `true.branch` is executed; otherwise `true.branch` is ignored. In the second from if `condition` is true then `true.branch` is executed; otherwise `false.branch` is executed.

The `if .. else` construct can be nested.

```
options(width=80)
#Set output width so it fits on the page.
x<-2
y<-6
if(x<1) cat("x is less than 1") else
if(y<x) cat("y is less than x") else
cat("Neither is true\n")

## Neither is true
```

The `ifelse` statement This is a vectorised form of the `if... else` statement. The syntax is

```
ifelse(test, true.value, false.value)
```

where all arguments are vectors and the recycling rule applies if any are short. All arguments are evlauated and `test` is coerced to logical if necessary. In those component positions where the value is true the corresponding component of `true.value` is the result and elsewhere it is that of `false.value`.

```
x<-c(6:-4)
sqrt(x)

## Warning in sqrt(x):  NaNs produced

##  [1] 2.449490 2.236068 2.000000 1.732051 1.414214 1.000000 0.000000      NaN
##  [9]      NaN      NaN      NaN

#This gives warnings for negative numbers.
sqrt(ifelse(x>=0,x,NA))

##  [1] 2.449490 2.236068 2.000000 1.732051 1.414214 1.000000 0.000000       NA
##  [9]       NA       NA       NA

#No warnings here. Note that the following also gives warnings! Can you explain why?
ifelse(x>=0,sqrt(x),NA)

## Warning in sqrt(x):  NaNs produced

##  [1] 2.449490 2.236068 2.000000 1.732051 1.414214 1.000000 0.000000       NA
##  [9]       NA       NA       NA

x.logx<-ifelse(x<=0, 0, x*log(x))

## Warning in log(x):  NaNs produced

#Again gives warnings! Why?
x.logx<-x*log(x+(x==0))

## Warning in log(x + (x == 0)):  NaNs produced

#Alternative ways that do not give errors are:
x.logx<-x*log(pmax(1,y))
x.logx<-x*log(ifelse(x<=0, 1, x))
```

The `while` statement The syntax is

```
while(cond) expr
```

The statement `expr` is executed as long as `cond` is true. Note that here the test is performed first, and if it fails then `expr` is never executed.

```
x<-15
while(x>0){
sqrt(x)
x<-x-1
}
```

The `for` statement The syntax is

`for(variable in sequence) statement`

```
for(i in 1:5) cat(i,i^2,"\n")


## 1 1
## 2 4
## 3 9
## 4 16
## 5 25
```

The `repeat` statement The syntax is

`repeat expr`

The `repeat` statement continues indefinitely unless exited by a `break` statement.

The `next` and `break` statements These two statements are used to modify the control of loops.

- `next` halts the processing of the current iteration and advances the looping index.
- `break` breaks out of the loop; control is transferred to the first statement outside the current loop.

```
i<-0
repeat{
i<-i+1
if(i %%2 ==0) next
cat(i,"\t")
if(i > 10) {cat("\n"); break}
#The ; acts as a separator between two statements on the same line.
}


## 1   3   5   7   9   11
```

4.2 Avoiding Loops

Loops can be much slower than vector operations. If possible vectorise your calculations using `apply()`, `lapply()`, `sapply()`, `mapply()` or `tapply()`.

```
dat<-matrix(rnorm(2000),nrow=100)
apply(dat,2,mean)

##  [1]  0.167757873 -0.087494303  0.038080204  0.001800081 -0.076936056
##  [6] -0.016920653  0.066429918 -0.057703155  0.072113138  0.016883597
## [11] -0.015019455  0.037787780 -0.119728001 -0.212140267  0.038140871
## [16]  0.097249739 -0.144309565 -0.014606328  0.031318786  0.186081700

colMeans(dat)

##  [1]  0.167757873 -0.087494303  0.038080204  0.001800081 -0.076936056
##  [6] -0.016920653  0.066429918 -0.057703155  0.072113138  0.016883597
## [11] -0.015019455  0.037787780 -0.119728001 -0.212140267  0.038140871
## [16]  0.097249739 -0.144309565 -0.014606328  0.031318786  0.186081700

apply(dat,1,sd)

##   [1] 1.1838647 0.7608379 1.1047206 1.1156692 0.6627261 1.1498016 0.9380708
##   [8] 1.0254931 1.0003149 0.9072996 1.1545253 1.0378336 0.8285203 1.0017415
##  [15] 1.1161578 0.9369072 1.1000743 1.0778444 1.1334731 1.4199582 0.9595392
##  [22] 0.7947671 1.0967251 0.9153765 0.9736280 1.0711094 1.4439263 1.0749075
##  [29] 1.0605613 1.1633118 0.9832311 1.0266039 0.7478606 0.9834549 0.9861016
##  [36] 1.3576389 1.0190172 1.1171945 1.0344844 1.2397886 0.8530940 0.8063018
##  [43] 0.8912365 0.8819952 0.9757949 0.8845317 0.9699621 1.0621516 1.1176179
##  [50] 1.0564313 1.0577779 0.7995059 1.2182980 0.8909437 1.2909050 1.1567391
##  [57] 0.8044037 0.9316801 0.9356180 1.0244154 0.9054661 0.5560617 1.1376508
##  [64] 0.9802255 1.0274579 1.1000487 0.9235631 0.7897167 1.1496571 1.0262267
##  [71] 1.2595949 1.0432736 1.0490519 1.2370560 0.9768282 1.0950786 1.1567588
##  [78] 0.8584346 1.1569579 0.9028902 1.0782909 0.9712501 0.7750268 0.7559480
##  [85] 1.0825987 0.8485452 0.9903801 1.1790798 1.1608457 0.5435550 0.9102086
##  [92] 0.8271005 0.9563824 0.9536419 1.0080550 0.9723729 0.9977832 0.6862080
##  [99] 1.1071024 0.8591089
```

4.3 Defining functions

The construct

```
fname<-function(arg1,arg2,...) function.body
```

defines a function, where

`fname` is the name of the function,

`arg1, arg2,...` are the arguments to the function,

`function.body` is the set of statements, usually enclosed within braces,{}, that comprise the function.

If `R` comes across the construct `return(value)` while executing the body of a function then the execution is terminated and `R` returns `value`. If `value` is missing then `NULL` is returned. If `value` is a single expression, the value of the evaluated expression is returned. If the end of a function is reached without calling `return()`, the value of the last evaluated expression is returned on exit.

```
foo<-function(x,i=5) x+i^2
#The default value of i is 5.
foo(3)

## [1] 28

#The default value of i is used here.
foo(3,1)

## [1] 4

#Now the specified value of i is used.
```

4.3.1 Matching Actual and Formal Arguments

The first thing that occurs in a function evaluation is the matching of formal arguments to the actual or supplied arguments. This is done by a three-pass process.

1. *Exact matching on tags*: For each named argument, the list of formal arguments is searched for an item whose name matches *exactly*. If several formal arguments match an actual argument, or vice versa, an error is generated.

2. *Partial matching on tags*: Each remaining named and supplied argument is compared with the remaining formal arguments using partial matching. If the name of the supplied argument matches exactly with the first part of the formal argument then the two arguments are considered matched.

   If the formal arguments contain ... then partial matching is only applied to arguments that precede it.

   ```
   f<-function(fumble,fooey) print("hello world")
   ```

   ```
   f(f=1,fo=2)
   Error in f(f = 1, fo = 2) :  argument 1 matches multiple formal arguments
   #The first supplied argument matches two formal arguments.
   ```

```r
f(f=1,fooey=2)

## [1] "hello world"

#Now there is an exact match of one of the supplied arguments with a formal argument
```

3. *Positional matching*: Any unmatched formal arguments are bound to *u*nnamed supplied arguments, in order. If there is a ... argument, it will take up the remaining arguments, tagged or not.

4.3.2 Lazy evaluation

`R` has a form of *l*azy evaluation of formal arguments.

```r
foo<-function(x,i)
{if(x>5) x^2 else x+i
}
foo(6)

## [1] 36

#i is not needed, so is not checked for.
#foo(2)
foo(2,3)

## [1] 5

bar<-function(y,j){
cat("Argument j is missing:", missing(j),"\n")
foo(x=y,i=j)
}
bar(6)

## Argument j is missing: TRUE
## [1] 36

#Explain the output!
bar(2)

## Argument j is missing: TRUE

## Error in foo(x = y, i = j):  argument "j" is missing, with no default

#Now explain the output.
bar(2,3)

## Argument j is missing: FALSE
## [1] 5
```

4.3.3 The `args()` function

This function give details of the names of the arguments of the function provided, and any default setting.

```
args(read.table)

## function (file, header = FALSE, sep = "", quote = "\"'", dec = ".",
##     numerals = c("allow.loss", "warn.loss", "no.loss"), row.names,
##     col.names, as.is = !stringsAsFactors, na.strings = "NA",
##     colClasses = NA, nrows = -1, skip = 0, check.names = TRUE,
##     fill = !blank.lines.skip, strip.white = FALSE, blank.lines.skip = TRUE,
##     comment.char = "#", allowEscapes = FALSE, flush = FALSE,
##     stringsAsFactors = default.stringsAsFactors(), fileEncoding = "",
##     encoding = "unknown", text, skipNul = FALSE)
## NULL

args(t.test)

## function (x, ...)
## NULL
```

4.4 Example

The following function performs a two-sample *t*-test.

```
my.ttest<-function(y1, y2, test="two-sided", alpha=0.05)
{
n1<-length(y1); n2=length(y2)
ndf<-n1+n2-2
s2<-((n1-1)*var(y1)+(n2-1)*var(y2))/ndf
tstat<- (mean(y1)-mean(y2))/sqrt(s2*(1/n1+1/n2))
tail.area<-switch(test,
"two-sided" = 2*(1-pt(abs(tstat),ndf)),
"lower"=pt(tstat,ndf),
"upper"=1-pt(tstat,ndf),
{warning("test must be `two-sided',`lower' or `upper'")
NULL
}
)
list(tstat=tstat,df=ndf,
reject=if(!is.null(tail.area)) tail.area<alpha, tail.area=tail.area)
}
args(my.ttest)
```

```
## function (y1, y2, test = "two-sided", alpha = 0.05)
## NULL

x<-c(4.2,4.1,5.3,2.1,4.5,5.8,6.3,4.6,3.5)
y<- c(4.4,2.3,5.6,3.4,6.5,7.6,2.3,3.5,5.6,5.7,6.7)
my.ttest(x,y,"two-sided")

## $tstat
## [1] -0.5384912
##
## $df
## [1] 18
##
## $reject
## [1] FALSE
##
## $tail.area
## [1] 0.5968351
```

4.5 Exercises

1. Write a function `my.mean()` that takes as argument a vector and returns its mean and standard deviation. DO NOT USE THE BUILT IN MEAN OR SD FUNCTIONS.

2. Write a function `my.t.test` that takes as argument a vector and returns the test statistic for a t-test. For the moment do not consider the alternative hypothesis, but note that you need a value for the mean under the null hypothesis.

3. Write a function that produces a seven-point summary of a vector; length, minimum, LQ, mean, median, UQ, maximum.

4.6 Reference

Ripley, W. N. and Venables, B. D. (2000) *S* Programming, Springer-Verlag, New York, Inc.

5 Probability Distributions

R has a full suite of probability distributions, listed in table 5.1 below. Note that the Gamma and Beta functions are defined as

$$\Gamma(\alpha) = \int_0^\infty x^{\alpha-1} e^{-x} dx$$
$$B(\alpha,\beta) = \frac{\Gamma(\alpha)\Gamma(\beta)}{\Gamma(\alpha+\beta)}.$$

The prefix to the names in R determines the function. Thus `dnorm, pnorm, qnorm` are respectively the density, CDF and quantile (inverse CDF) of the normal distribution, whereas `rnorm` generates random observations from a normal distribution.

5.1 Exercises

1. Write the R code for the following.
   (a) Simulate 1000 observations from a $N(2,16)$ distribution.
   (b) Simulate 1000 observations each from Poisson distributions with rates $1,5,10$ and store them in a matrix columnwise.
2. In section 2, for the exercise on the sales from the chain store, conduct the test of hypothesis using an appropriate probability distribution and report your results.
3. Repeat the previous question if the stores for each year were a random sample of the chain of stores.

Distribution	Rname	Parameters	Probability mass/density function
beta	`beta`	α, β	$\frac{x^{\alpha-1}(1-x)^{\beta-1}}{B(\alpha,\beta)}, x \geq 0$
binomial	`binom`	n, p	
Cauchy	`cauchy`	θ, η	$\frac{\theta}{\pi\left(\theta^2+(x-\eta)^2\right)}, x \in \mathcal{R}$
chi-squared	`chisq`	df ν	sum of ν independent $N(0,1)$ squared rvs
exponential	`exp`	rate λ	$\lambda e^{-\lambda x}, x \geq 0$
F	`f`	ν_1, ν_2	Ratio of $\chi^2_{\nu_1}$ and $\chi^2_{\nu_2}$ rvs
gamma	`gamma`	shape α, scale λ	$\frac{x^{\alpha-1}e^{-x/\lambda}}{\lambda^\alpha \Gamma(\alpha)}, x \geq 0$
geometric	`geom`	p	
hypergeometric	`hyper`	m, n, k	
log-normal	`lnorm`	μ, σ	$\frac{1}{\sqrt{2\pi}\sigma x} exp\left(\frac{(\log x-\mu)^2}{2\sigma^2}\right)$
logistic	`logis`	`location, scale`	
negative binomial	`nbinom`	n, p	
normal	`norm`	μ, σ	
Poisson	`pois`	μ	
T	`t`	`df`	
uniform	`unif`	`min, max`	
Weibull	`weibull`	`shape, scale`	
Wilcoxon	`wilcox`	m, n	

Table 5.1: Standard probability distributions in R.

6 Graphics

6.1 Types of Graphics Functions

R has three levels of graphics functions. The package `graphics` contains all the functions for the base graphics. The available function can be listed by

```
library(help=graphics)
```

High-level plotting functions are used to produce a basic plot, and *low-level* functions are used for fine control of the plots.

1. High level functions, such as `plot()`, `hist()`, `boxplot()` or `pairs()`, that produce or initialise a plot.

2. Low level function that add to an existing plot that was produced by a high level plot function. Plots can enhanced by adding lines, points, axes and text.

3. Interactive graphics commands: Commands for adding information to, or extracting information from, an existing plot in an interactive manner.

There are also suites of functions in packages such as `Trellis` functions, including `xyplot`, `bwplot` and `histogram` that can produce an entire multipanel display in a single call.

Various parameters can be set for the plot by the `par()` call. Some of these will be illustrated below, but the full set of default values are displayed by the `par()` command. Calls to low level functions can be made to add to a plot that is produces by a high level plotting function. This feature is not available after a trellis function call.

Example 1 The example below illustrates some of the features of high level and low level graphics functions. Observations from the *Exp*(1) distribution are simulated, and histograms of the means of sample sizes $1, 4$ and 16 are plotted in Figure 6.1.

```
rmt <- matrix(rexp(1000 * 16), nrow = 16)
#Generate the exp(1) rvs data and store as a matrix.
#Note the matrix is read in columnwise.
mns <- # Apply the mean function to columns
 cbind(rmt[ 1, ], # means of samples of 1
 apply(rmt[ 1:4, ], 2, mean), # means of samples of 4
 apply(rmt[ 1:16, ], 2, mean) # means of samples of 16
)
#Note the 2 in the function apply indicates column means.
#So we are obtaining 1000
#means of size 1, 4 and 16.
str(mns)

##  num [1:1000, 1:3] 0.00627 0.35873 2.98178 2.77952 2.61053 ...
```

```
meds <- # Apply the median function to columns
cbind(rmt[ 1, ], # medians of samples of 1
 apply(rmt[ 1:4, ], 2, median), # medians of samples of 4
 apply(rmt[ 1:16, ], 2, median) # medians of samples of 16
 )

hist(mns[, 1]) # a histogram of the means of samples of 1
hist(mns[,2], cex=0.6, main = "Means of samples of size 4",
xlab = "Size 4 means", las = 1) # sets the axis label style
hist(mns[,3], cex=0.6,main = "Means of samples of size 16",
 xlab = "Size 16 means", las = 1, col = "darkred", prob = TRUE)
lines(density(mns[,3]), col = "blue")
```

All these histograms have been put together in one figure by appropriately setting the parameter `mfrow=c()` (multifigure row-wise). Note that there are a thousand means of samples of size 1, 4 and 16. Thus the first histogram resembles an exponential distribution, while the third is close to a normal distribution.

6.2 Formula data-specification

Most high level graphics functions in `R` allow formula-data specification for the plot. In the `trellis` package function suite, this is the only way to specify the data for a plot. In `R` a formula is indicated by the character $\sim$. This same symbol is also used to specify statistical models, and so it is usually read as "is modelled as". To use this form of data-specification, first an appropriate data frame needs to be defined. The example below illustrates some of the idea.

Example 2 We will first combine the sample means from the Exponential distribution into a single data frame.

```
expdata<-data.frame(data=c(mns,meds),
szs=gl(3,1000,length=6000,labels = c("1", "4", "16")),
type = gl(2, 3000, labels = c("Mean", "Median")))
str(expdata)

## 'data.frame': 6000 obs. of  3 variables:
##  $ data: num  0.00627 0.35873 2.98178 2.77952 2.61053 ...
##  $ szs : Factor w/ 3 levels "1","4","16": 1 1 1 1 1 1 1 1 1 1 ...
##  $ type: Factor w/ 2 levels "Mean","Median": 1 1 1 1 1 1 1 1 1 1 ...
```

Note that there are 3,000 each of means and medians, making a total of 6,000 points. These are stacked to make one vector, their sizes of their corresponding samples are stored in the variable `szs` and the type (mean or median) is indicated in the variable `type`. Note the use of the function `gl` to generate a vector indicating the sample sizes.

The general form of specifying a plot by a formula is

$$y \sim x \mid g$$

```
par(mfrow=c(2,2))
par(cex=0.5)
hist(mns[, 1],main="Means of samples of size 1",xlab="Size 1 means",las=1)

hist(mns[,2], main = "Means of samples of size 4",
xlab = "Size 4 means", las = 1)

hist(mns[,3], main = "Means of samples of size 16",
 xlab = "Size 16 means", las = 1, col = "darkred", prob = TRUE)
lines(density(mns[,3]), col = "blue")
```

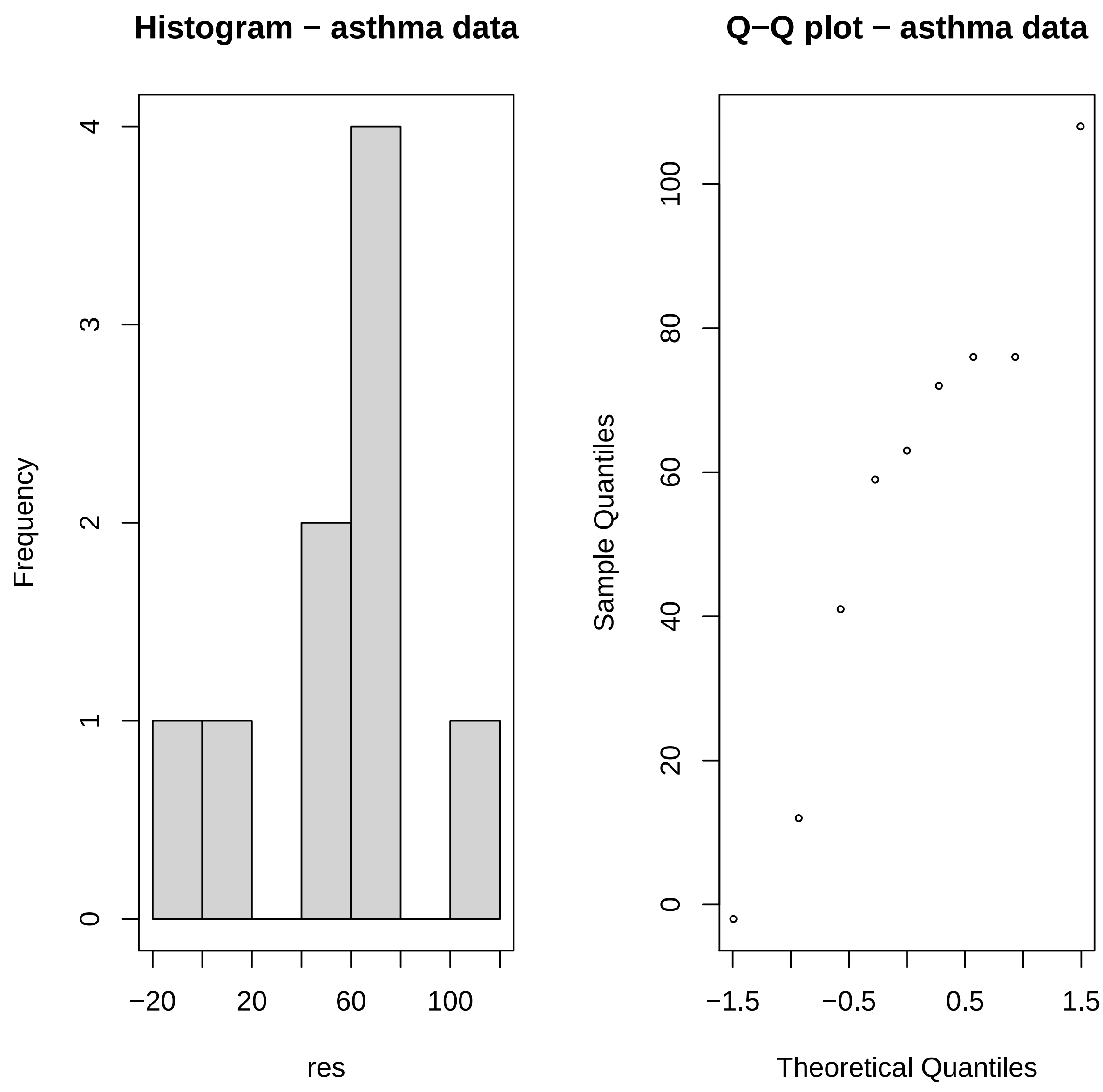


Figure 6.1: Histogram of sample means from exponential data with parameter $\lambda = 1$.

where y is assigned the vertical axis, x is assigned the horizontal axis and g is a grouping factor or expression.

6.3 High-level plotting functions

The commonly used high level plotting functions are described below. Their usage and syntax can be obtained by the `help()` function in `R`.

`plot(x,y)` **or** `plot(y~x)` produces a scatter plot of the variables.

- — If `x` is a dataframe then `plot(X)` produces a matrix plot of the columns of `X` against each other.
- — If `x` is a factor then `plot(y~x)` or `plot(x,y)` produces a boxplot of `y` by the levels of `x`.
- — If `x` is a time series then `plot(x)` produces a time series plot.
- — If `x` is a factor then `plot(X)` produces a bar plot of `x`.

`barplot()` Produces a bar plot.

`boxplot()` Produces a box plot.

`curve()` Produces plots of functions. Useful for plotting mathematical functions.

`contour()` Produces contour plots.

`coplot(a~b|c)` Conditioning plots. Usually `c` is a factor, and a separate plot of the variables is produced for each level of `c`.

`hist(x,...)` Produces a histogram of the variable `x`. Several other arguments can be provided to this function, and several low-level plotting commands can be used to control and modify the output.

`image(x,y,z,...)` Produces a colour image.

`matplot` Plots columns of matrices.

`pairs(X)` Produces a pairwise scatterplot matrix of the variables defined by the columns of X, that is, every column of `X` is plotted against every other column of X and the resulting n(n-1) plots are arranged in a matrix with plot scales constant over the rows and columns of the matrix.

`persp` Perspective plots.

`qqnorm(x)` Plots the numeric vector `x` against the expected Normal scores.

`qqline(x)` To the `qqnorm(x)` plot, adds a straight line through the distribution and data quartiles.

`qqplot(x,y)` Plots quantiles of `x` against the quantiles of `y` to compare their respective distributions.

`stripchart` Produces a one-dimensional dot plot of the data.

The most often used of the above are `plot()`, `hist()` and `boxplot()`, and these will be illustrated with examples later.

There are a number of arguments which may be passed to high-level graphics functions to modify the plot. Some of these are described below.

6.4 Arguments to high-level plotting functions

`add=TRUE` Forces the function to act as a low-level graphics function, superimposing the plot on the current plot (some functions only).

`axes=FALSE` Suppresses the generation of axes, which can be added in later with the `axis()` function. This gives the user flexibility and control over the axes.

`log="x", log="y", log="xy"` Causes the appropriates axes (one or both) to be logarithmic. This will work for many, but not all, types of plots.

`type=` Controls the type of plot produced.

- `type="p"` Plot individual points (the default).
- `type="l"` Plot lines.
- `type="b"` Plot points *connected* by lines (*both*).
- `type="o"` Plot points *overlaid* with lines.
- `type="h"` Plot vertical lines from points to the zero axis (*high-density*).
- `type="s", type="S"` Step-function plots. In the first form the top of the step defines the point; in the second,the bottom.
- `type="n"` No plot is produced, but the axes are drawn and the co-ordinate system is set up according to the data. This is used for creating a plot with subsequent low-level graphics functions, and gives the user more fine control over the plot.

`xlab=`*string*

`ylab=`*string* Axis labels. These are used to override the default labels, usually the names of the objects used in the high-level plotting function.

`main=`*string* Figure title, placed in a large font at the top of the plot.

`sub=`*string* Sub-title, placed just below the x-axis in a smaller font.

6.5 Low-level plotting function

Commonly used low level plotting functions are described below.

`points(x,y)`

`lines(x,y)` Adds points or lines to the current plot. The `type=` argument can also be passed to these functions (and defaults to `"p"` for `points()` and `"l"` for `lines()`).

`text(x,y,labels,...)` Add text to a plot at the points given by `x,y`. Normally `labels` is an integer or character vector in which case `labels[i]` is plotted at `(x[i],y[i])`. This function is often used in the sequence

```
plot(x,y,type="n")
text(x,y,names)
```

The graphic parameter `type="n"` suppresses the point but sets up the axes, and the points are supplied by the function `text()` as special characters specified by the character vector `names`.

`abline(a,b)` Adds the line $y = a + bx$ of slope `b` and intercept `a` to the current plot.

`abline(h=y)` Add the horizontal line $y = h$ to the current plot.

`abline(v=x)` Add the vertical line $x = v$ to the current plot.

`abline(`*lm.obj*`)` Add a line using `lm.onj`, a list of length 2, taken as the intercept and slope of the line in that order.

`polygon(x,y,...)` Draws a polygon defined by the ordered vertices in `(x,y)` and (optionally) shades it with hatched lines or fills it.

`legend(x,y,legend,...)` Adds a legend to the current plot at the specified position. Plotting characters, line styles, colours etc. are identified in the legend with labels in the character vector `legend`, making the reading of graphs easier. At least one other argument v (a vector the same length as legend) with the corresponding values of the plotting unit must also be given, as follows:

`legend(text , fill=v)` Colors for filled boxes

`legend(text , col=v)` Colors in which points or lines will be drawn

`legend(text , lty=v)` Line styles

`legend(text , lwd=v)` Line widths

`legend(text , pch=v)` Plotting characters (character vector)

`title(main, sub)` Adds a title main to the top of the current plot in a large font and (optionally) a sub-title sub at the bottom in a smaller font.

`axis(side, ...)` Adds an axis to the current plot on the side given by the first argument (1 to 4, counting clockwise from the bottom.) Other arguments control the positioning of the axis within or beside the plot, and tick positions and labels. Useful for adding custom axes after calling `plot()` with the `axes=FALSE` argument.

`mtext(text, side = 3, line = 0, outer = FALSE,...)` Writes `text` in one of four margins, counted from bottom (`side=1`) clockwise, on margin line given by `line`, starting at 0 and counting outwards. If `outer=TRUE` then the outer margins are used if available.

Low-level plotting functions usually require some positioning information (e.g., x and y coordinates) to determine where to place the new plot elements. Coordinates are given in terms of user coordinates which are defined by the previous high-level graphics command and are chosen based on the supplied data.

Where x and y arguments are required, it is also sufficient to supply a single argument being a list with elements named x and y. Similarly a matrix with two columns is also valid input. In this way functions such as `locator()` (see below) may be used to specify positions on a plot interactively.

6.6 Mathematical annotation

In some cases, it is useful to add mathematical symbols and formulae to a plot. This can be achieved in `R` by specifying an expression rather than a character string in any one of `text`, `mtext`, `axis` or `title`. For example, the following code draws the formula for the Binomial probability function at the point `(4,5)`:

```
text(4, 5, expression(paste(bgroup("(", atop(n, x), ")"), p^x, q^{n-x})))
```

More information, including a full listing of the features available can obtained from within R using the commands:

```
help(plotmath)
example(plotmath)
demo(plotmath)
```

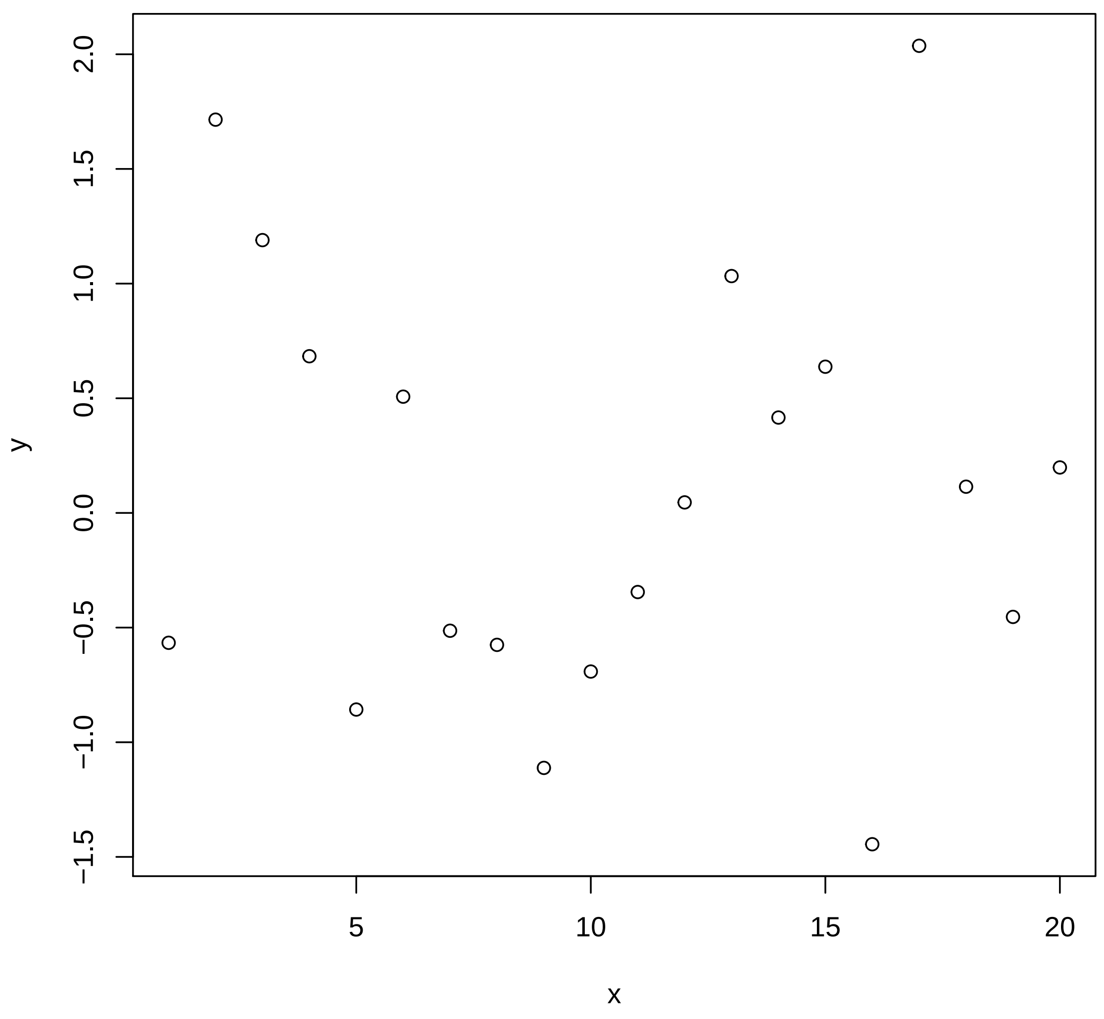
6.7 Interacting with graphics

`R` also provides functions which allow users to extract or add information to a plot using a mouse.

`locator(n, type)` Waits for the user to select locations on the current plot using the left mouse button. This continues until *n* (default 512) points have been selected, or another mouse button is pressed. The type argument allows for plotting at the selected points and has the same effect as for high-level graphics commands; the default is no plotting. `locator()` returns the locations of the points selected as a list with two components x and y.

locator() is usually called with no arguments. It is particularly useful for interactively selecting positions for graphic elements such as legends or labels when it is difficult to calculate in advance where the graphic should be placed. For example, to place some informative text near an outlying point, the command

```
x<-c(1:20)
y<-rnorm(20)
plot(x, y)
text(locator(1), "Outlier", adj=0)

## Error in text.default(locator(1), "Outlier", adj = 0):  no coordinates
were supplied
```



labels the selected point as Outlier. (locator() will be ignored if the current device, such as postscript does not support interactive pointing.)

`identify(x, y, labels)` Allows the user to highlight any of the points defined by x and y (using the left mouse button) by plotting the corresponding component of labels nearby (or the index number of the point if labels is absent). Returns the indices of the selected points when another button is pressed.

Sometimes we want to identify particular points on a plot, rather than their positions. For example, we may wish the user to select some observation of interest from a graphical display and then manipulate that observation in some way. Given a number of (x, y) coordinates in two numeric vectors x and y, we could use the `identify()` function as follows:

```
x<-c(1:20)
y<-rnorm(20)
plot(x, y)
identify(x, y)
```

The `identify()` functions performs no plotting itself, but simply allows the user to move the mouse pointer and click the left mouse button near a point. If there is a point near the mouse pointer it will be marked with its index number (that is, its position in the `x/y` vectors) plotted nearby. Alternatively, you could use some informative string (such as a case name) as a highlight by using the labels argument to `identify()`, or disable marking altogether with the `plot = FALSE` argument. When the process is terminated (see above), `identify()` returns the indices of the selected points; you can use these indices to extract the selected points from the original vectors x and y.

6.8 Using graphic parameters

The `par()` function is used to access and modify the list of graphics parameters for the current graphics device.

`par()` Without arguments, returns a list of all graphics parameters and their values for the current device.

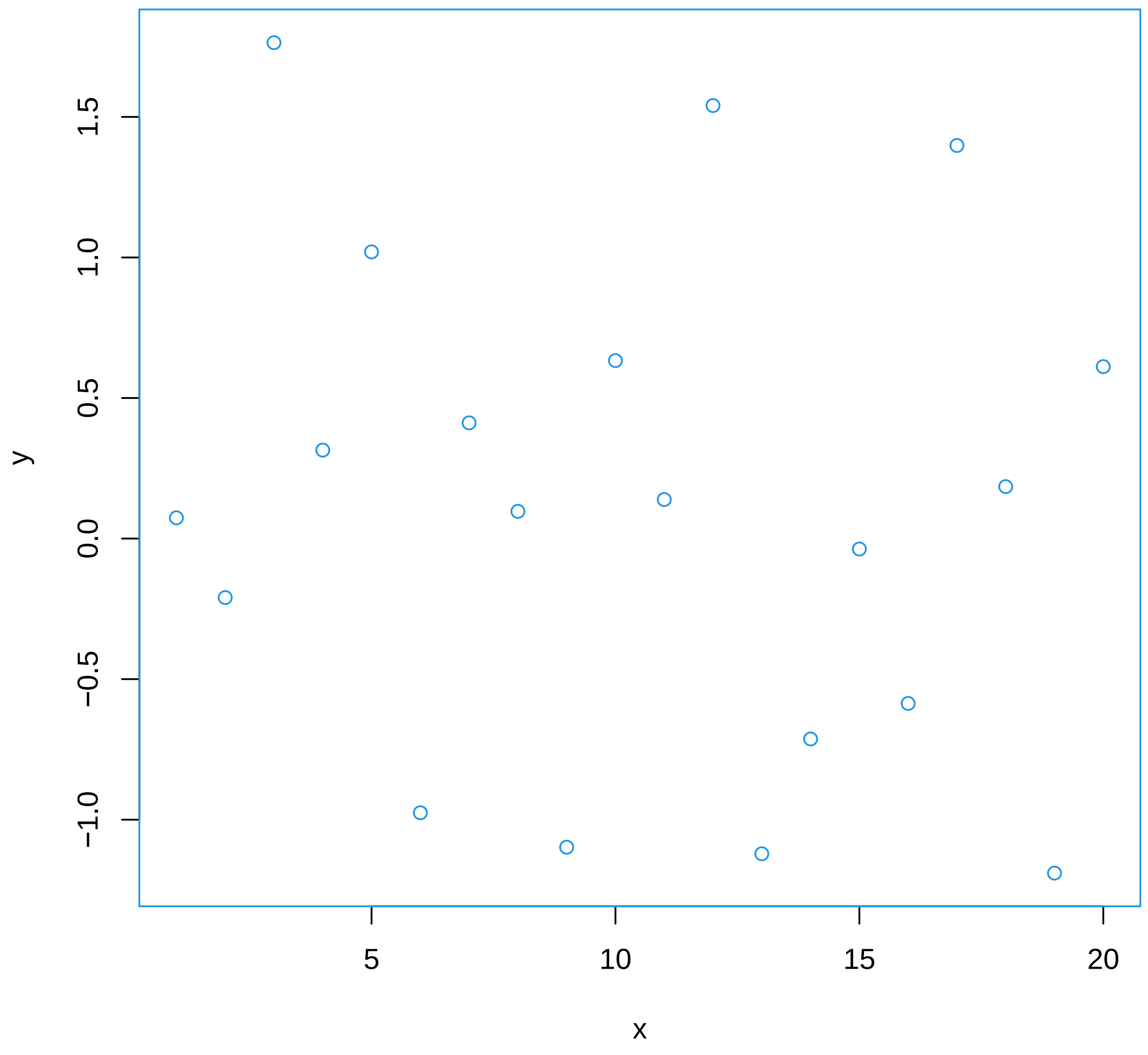
`par(c("col", "lty"))` With a character vector argument, returns only the named graphics parameters (again, as a list.)

`par(col=4, lty=2)` With named arguments (or a single list argument), sets the values of the named graphics parameters, and returns the original values of the parameters as a list.

Setting graphics parameters with the `par()` function changes the value of the parameters permanently, in the sense that all future calls to graphics functions (on the current device) will be affected by the new value. You can think of setting graphics parameters in this way as setting default values for the parameters, which will be used by all graphics functions unless an alternative value is given.

Note that calls to `par()` always affect the global values of graphics parameters, even when `par()` is called from within a function. This is often undesirable behaviourusually we want to set some graphics parameters, do some plotting, and then restore the original values so as not to affect the user's R session. You can restore the initial values by saving the result of `par()` when making changes, and restoring the initial values when plotting is complete.

```
oldpar <- par(col=4, lty=2)
x<-c(1:20)
y<-rnorm(20)
plot(y~x)
```



```
par(oldpar)
```

To save and restore all settable 23 graphical parameters use

```
oldpar <- par(no.readonly=TRUE)
#plotting commands
par(oldpar)
```

6.8.1 Graphical parameter list

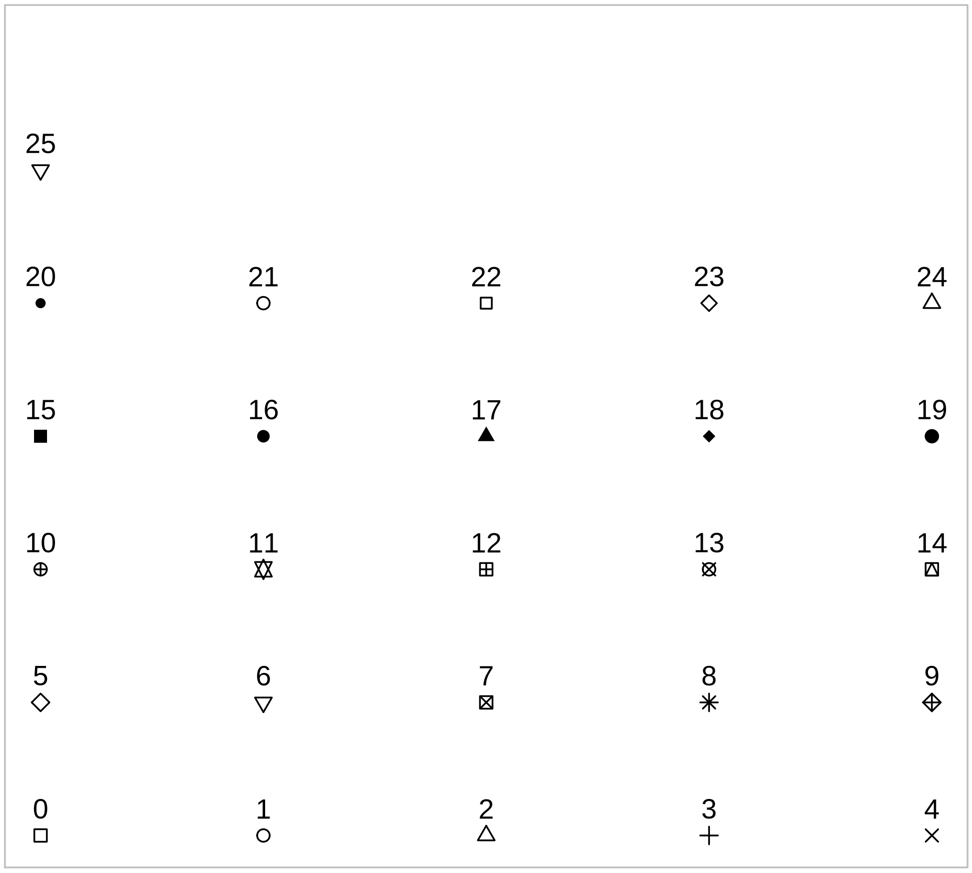
R plots are made up of points, lines, text and polygons (which may be filled). Graphical parameters allow fine control of these graphical elements. Below is a list of the parameters. These parameters may be specified in high level graphical functions (such as `plot()`,low level graphical functions (such as `points`), or using the function `par()`. Note that the first two set these parameters for only the current plot, while the third sets these parameter values for all plots during the R session.

Graphical elements

`pch="+"` Character to be used for plotting points. The default varies with graphics drivers, but it is usually a circle. Plotted points tend to appear slightly above or below the appropriate position unless you use "." as the plotting character, which produces centred points.

`pch=4` When pch is given as an integer between 0 and 25 inclusive, a specialized plotting symbol is produced. The following commands will produce a plot of these symbols.

```
x<-rep(0:4,7)
y<-rep(1:7,each=5)
plot(x,y,type="n",axes=F,xlab="",ylab="")
points(x[1:26],y[1:26],pch=0:25)
text(x[1:26],y[1:26]+0.2,0:25)
box(,col="grey")
```



In addition, pch can be a character or a number in the range 32:255 representing a character in the current font.

`lty=2` Line types. Alternative line styles are not supported on all graphics devices (and vary on those that do) but line type 1 is always a solid line, line type 0 is always invisible, and line types 2 and onwards are dotted or dashed lines, or some combination of both.

`lwd=2` Line widths. Desired width of lines, in multiples of the standard line width. Affects axis lines as well as lines drawn with lines(), etc. Not all devices support this, and some have restrictions on the widths that can be used.

`col=2` **or** `col="red"` Colours to be used for points, lines, text, filled regions and images. A number from the current palette (see `?palette`) or a named colour.

`col.axis, col.lab, col.main, col.sub` The color to be used for axis annotation, x and y labels, main and sub-titles, respectively.

`font=2` An integer which specifies which font to use for text. If possible, device drivers arrange so that 1 corresponds to plain text, 2 to bold face, 3 to italic, 4 to bold italic and 5 to a symbol font (which include Greek letters).

`font.axis, font.lab, font.main, font.sub` The font to be used for axis annotation, x and y labels, main and sub-titles, respectively.

`adj=-0.1` Justification of text relative to the plotting position. 0 means left justify, 1 means right justify and 0.5 means to center horizontally about the plotting position. The actual value is the proportion of text that appears to the left of the plotting position, so a value of -0.1 leaves a gap of 10% of the text width between the text and the plotting position.

`cex=1.5` Character expansion. The value is the desired size of text characters (including plotting characters) relative to the default text size.

`cex.axis, cex.lab, cex.main, cex.sub` The character expansion to be used for axis annotation, x and y labels, main and sub-titles, respectively.

Axes and tick marks Many of `R`'s high-level plots have axes. Axes have three main components: the axis line (line style controlled by the `lty` graphics parameter), the tick marks (which mark off unit divisions along the axis line) and the tick labels (which mark the units.) These components can be customized with the following graphics parameters.

`lab=c(5, 7, 12)` The first two numbers are the desired number of tick intervals on the x and y axes respectively. The third number is the desired length of axis labels, in characters (including the decimal point.) Choosing a too-small value for this parameter may result in all tick labels being rounded to the same number!

`las=1` Orientation of axis labels. 0 means always parallel to axis, 1 means always horizontal, and 2 means always perpendicular to the axis.

`mgp=c(3, 1, 0)` Positions of axis components. The first component is the distance from the axis label to the axis position, in text lines. The second component is the distance to the tick labels, and the final component is the distance from the axis position to the axis line (usually zero). Positive numbers measure outside the plot region, negative numbers inside.

`tck=0.01` Length of tick marks, as a fraction of the size of the plotting region. When `tck` is small (less than 0.5) the tick marks on the x and y axes are forced to be the same size. A value of 1 gives grid lines. Negative values give tick marks outside the plotting region. Use `tck=0.01` and `mgp=c(1,-1.5,0)` for internal tick marks.

`xaxs="r", yaxs="i"` Axis styles for the x and y axes, respectively. With styles `"i"` (internal) and `"r"` (the default) tick marks always fall within the range of the data, however style `"r"` leaves a small amount of space at the edges.

Figure margins

A single plot in `R` is known as a `figure` and comprises a `plot region` surrounded by margins, possibly containing axis labels and titles, and bounded by the axes themselves. The following graphic parameters allow control of the figure layout.

`mai=c(1,0.5,0.5,0)` Widths of the bottom, left, top and right margins respectively, measured in inches.

`mar=c(4,2,2,1)` Similar to `mai`, except the measurement unit is text lines.

`mar` and `mai` are equivalent in the sense that setting one changes the value of the other. The default values chosen for this parameter are often too large; the right-hand margin is rarely needed, and neither is the top margin if no title is being used. The bottom and left margins must be large enough to accommodate the axis and tick labels. Furthermore, the default is chosen without regard to the size of the device surface (see later): for example, using the `postscript()` driver with the `height=4` argument will result in a plot which is about 50% margin unless `mar` or `mai` are set explicitly. When multiple figures are in use (see below) the margins are reduced, however this may not be enough when many figures share the same page.

Multiple figure environments

`R` allows the user to create an *n* by *m* array of figures on a single plot. Each figure has its own margins, and teh array of figures is optionally surrounded by and `outer margin` (see below). The following parameters relate to multiple figures.

`mfcol=c(3,2), mfrow=c(2,4)` Se the size of the multiple figure array. The first value is hte number of rows, and the second value is the number of columns. `mfcol` causes figures to be allocated by columns, while mfrow allocated figures row-wise.

`mfg=c(2,2,3,2)` Position of the figure in a multiple figure environment. The first two numbers indicate the position of the current figure; the last two numbers are the number of rows and columns in the multiple figure environment. This parameter allows the user to jump to a figure in the array. Values different from the *true* values can be used for unequally-sized figures on the same plot.

`fig=c(4,9,1,4)/10` Position of the current figure on the page. Values are the positions of the left, right, bottom and top edges respectively, (or, the horizontal range and vertical range) as a percentage of the page measured from the bottom left corner. The example indicates a figure in the bottom right of the page. This parameter is used for arbitrary positioning of figures within a page. If you want to add a figure to a current page, use new=TRUE as well.

`oma=c(2,0,3,0),omi=c(0,0,0.8,0)` Size of outer margins. As for `mar` and `mai`, the first measure is in text lines and the second in inches, starting from the bottom margins and working around clockwise.

6.9 Examples

Example 1 Below are a scatterplot (Figure 6.2) and barplots (Figure 6.3) of the powerball data.

```
powerball<-read.table("Data/PowerBall.txt",header=T)
head(powerball)

##   BallNumber BarrelA BarrellB
## 1          1      24        6
## 2          2      29        5
## 3          3      32        8
## 4          4      29        9
## 5          5      39        4
## 6          6      26        7

plot(powerball[,2],powerball[,3],xlab="Barrel A",ylab="Barrel B")
```

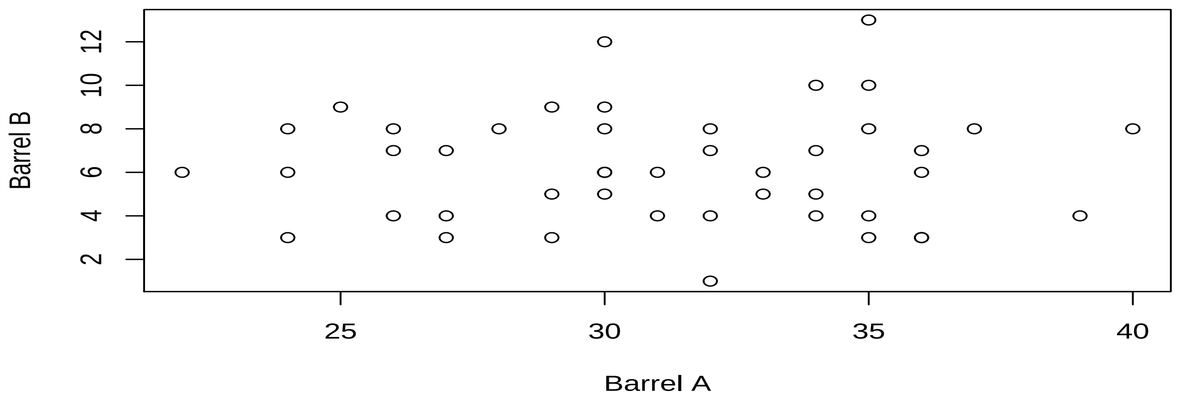


Figure 6.2: Scatter plot of power ball data.

```r
options(width=50)
par(mfrow=c(2,2))
barplot(powerball[,2],main="Barrel A",ylab="Frequency")
barplot(powerball$BarrelA,names.arg=powerball$BallNumber,space=0.5)
barplot(powerball[,"BarrelA"],names.arg=powerball[,"BallNumber"],space=0.9)
barplot(powerball[,"BarrelA"],names.arg=powerball[,"BallNumber"],
space=1.0,xlab="Number",ylab="Frequency",
main="Plot of Powerball Draws: Barrel A")
```

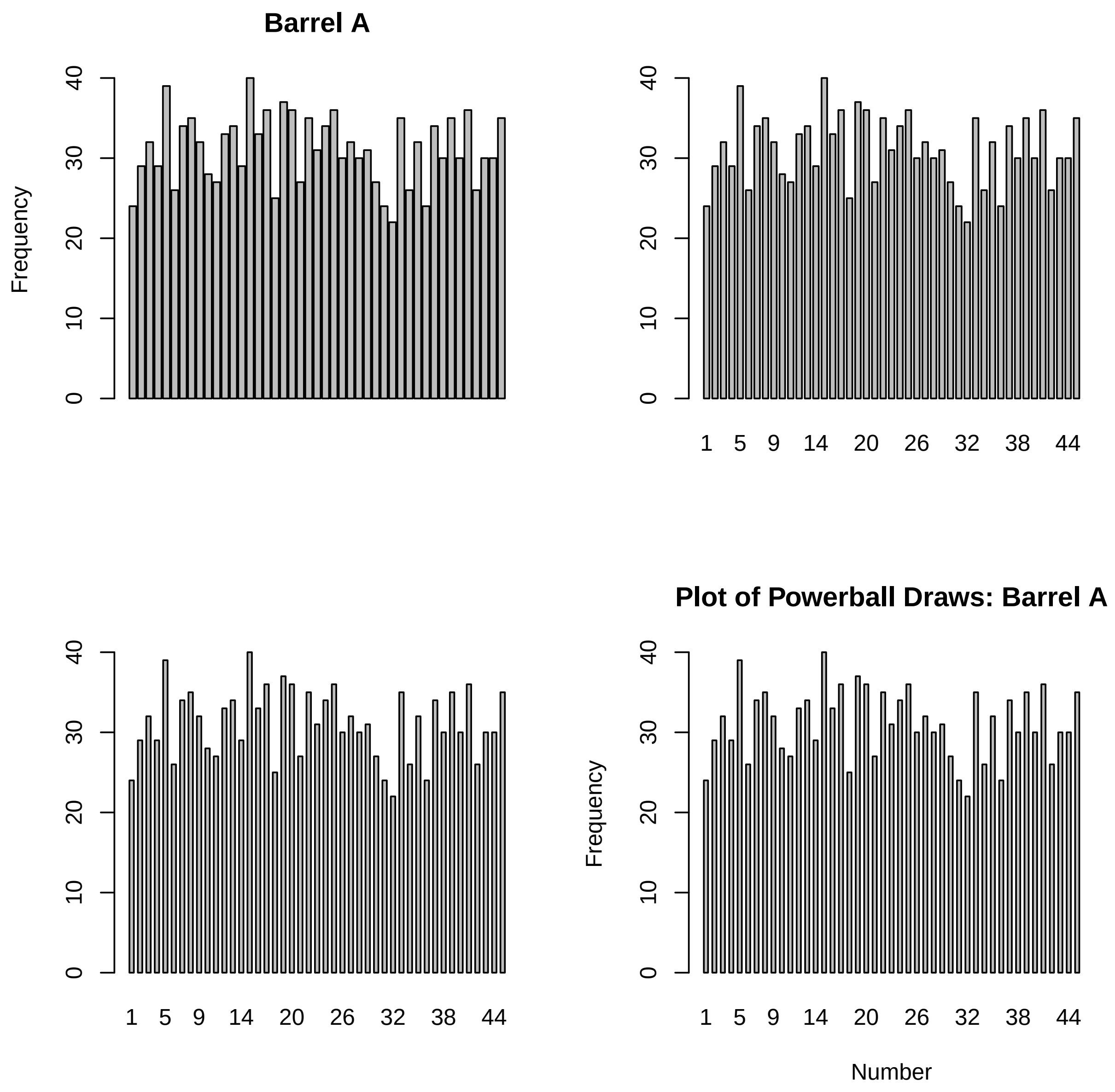


Figure 6.3: Barplot of power ball data.

Example 2 This example illustrates some features of high level and low level plotting functions. Five histograms are plotted on the same plot. Each histogram is positioned on the plot area by using `fig` graphical parameter. While this example uses the high level plotting function `hist`, other high level plotting functions, such as `plot()` or `curve()` may be similarly used.

```
x<-scan("Data/L0.txt")
y<-scan("Data/L1.txt")
z<-scan("Data/L2.txt")
u<-scan("Data/L3.txt")
v<-scan("Data/L4.txt")
#Position the first figure in the top half of the page.
par(fig=c(0.2,0.8,0.5,1),mar=c(3,3,1.5,0.5),cex=0.4)
hist(x,axes=F,nclass=40,xlab="",ylab="",main="")
#Place the time axis title in the margin, on the second line.
mtext("time (ms)",cex=0.6, side=1, line=2, font=3)
#Place the vertical axis title in the margin, on the second line.
mtext("frequency",cex=0.6,side=2, line=2, font=3)
#Main title of the plot.
title("Level 0",cex=0.4)
#Define axes.
axis(1,at=c(0,10,20,30,40,50,60),labels=c(0,10,20,30,40,50,60),cex=0.6)
axis(2,at=c(0,50,100,150,200),labels=c(0,50,100,150,200),cex=0.6)
#Put a nice box around the plot.
box()
par(fig=c(0.05,0.45,0.25,0.5),mar=c(3,3,1.5,0.5),cex=0.4,new=TRUE)
hist(y,axes=F,breaks=c(0,0.04,0.08,0.12,0.16,0.2,0.24,0.28),
xlab="",ylab="",main="",include.lowest=T)
mtext("time (ms)",cex=0.6, side=1, line=2, font=3)
mtext("frequency",cex=0.6, side=2, line=2, font=3)
title("Level 1",cex=0.4)
axis(1,at=c(0,0.04,0.08,0.12,0.16,0.20,0.24,0.28),
labels=c(0,0.04,0.08,0.12,0.16,0.20,0.24,0.28),cex=0.6)
axis(2,at=c(0,50,100,150,200),labels=c(0,50,100,150,200),cex=0.6)
box()
par(fig=c(0.55,0.95,0.25,0.5),new=TRUE)
hist(z,axes=F,breaks=c(0,0.04,0.08,0.12,0.16,0.20,0.24),
xlab="",ylab="",main="")
mtext("time (ms)",cex=0.6, side=1, line=2, font=3)
mtext("frequency",cex=0.6, side=2, line=2, font=3)
title("Level 2",cex=0.4)
axis(1,at=c(0,0.04,0.08,0.12,0.16),
labels=c(0,0.04,0.08,0.12,0.16),cex=0.6)
axis(2,at=c(20,40,60,80,100,120,140),
labels=c(20,40,60,80,100,120,140),cex=0.6)
```

```r
box()
par(fig=c(0.05,0.45,0,0.25),new=TRUE)
hist(u,axes=F,breaks=c(0,0.04,0.08,0.12,0.16,0.20,0.24,0.28,0.32),
xlab="",ylab="",main="")
mtext("time (ms)",cex=0.6, side=1, line=2, font=3)
mtext("frequency", cex=0.6,side=2, line=2, font=3)
title("Level 3",cex=0.4)
axis(1,at=c(0,0.04,0.08,0.12,0.16,0.20),
labels=c(0,0.04,0.08,0.12,0.16,0.20),cex=0.6)
axis(2,at=c(0,20,40,60,80,100,120),
labels=c(0,20,40,60,80,100,120),cex=0.6)
box()
par(fig=c(0.55,0.95,0,0.25),new=TRUE)
hist(v,axes=F,nclass=20,xlab="",ylab="",main="")
mtext("time (ms)",cex=0.6, side=1, line=2, font=3)
mtext("frequency",cex=0.6, side=2, line=2, font=3)
title("Level 4",cex=0.4)
axis(1,at=c(0,0.5,1,1.5,2,2.5,3,3.5,4,4.5),
labels=c(0,0.5,1,1.5,2,2.5,3,3.5,4,4.5),cex=0.6)
axis(2,at=c(0,10,20,30,40,50),labels=c(0,10,20,30,40,50),cex=0.6)
box()
```

6.10 Device drivers

R can generate graphics (of varying levels of quality) on almost any type of display or printing device. Before this can begin, however, R needs to be informed what type of device it is dealing with. This is done by starting a device driver. The purpose of a device driver is to convert graphical instructions from R (draw a line, for example) into a form that the particular device can understand.

Device drivers are started by calling a device driver function. There is one such function for every device driver: type `help(Devices)` for a list of them all. For example, issuing the command

```r
postscript()
```

causes all future graphics output to be sent to the printer in PostScript format. Some commonly-used device drivers are:

`X11()` For use with the X11 window system on Unix-alikes.

`windows()` For use on Windows

`quartz()` For use on Mac OS X

`postscript()` For printing on PostScript printers, or creating PostScript graphics files.

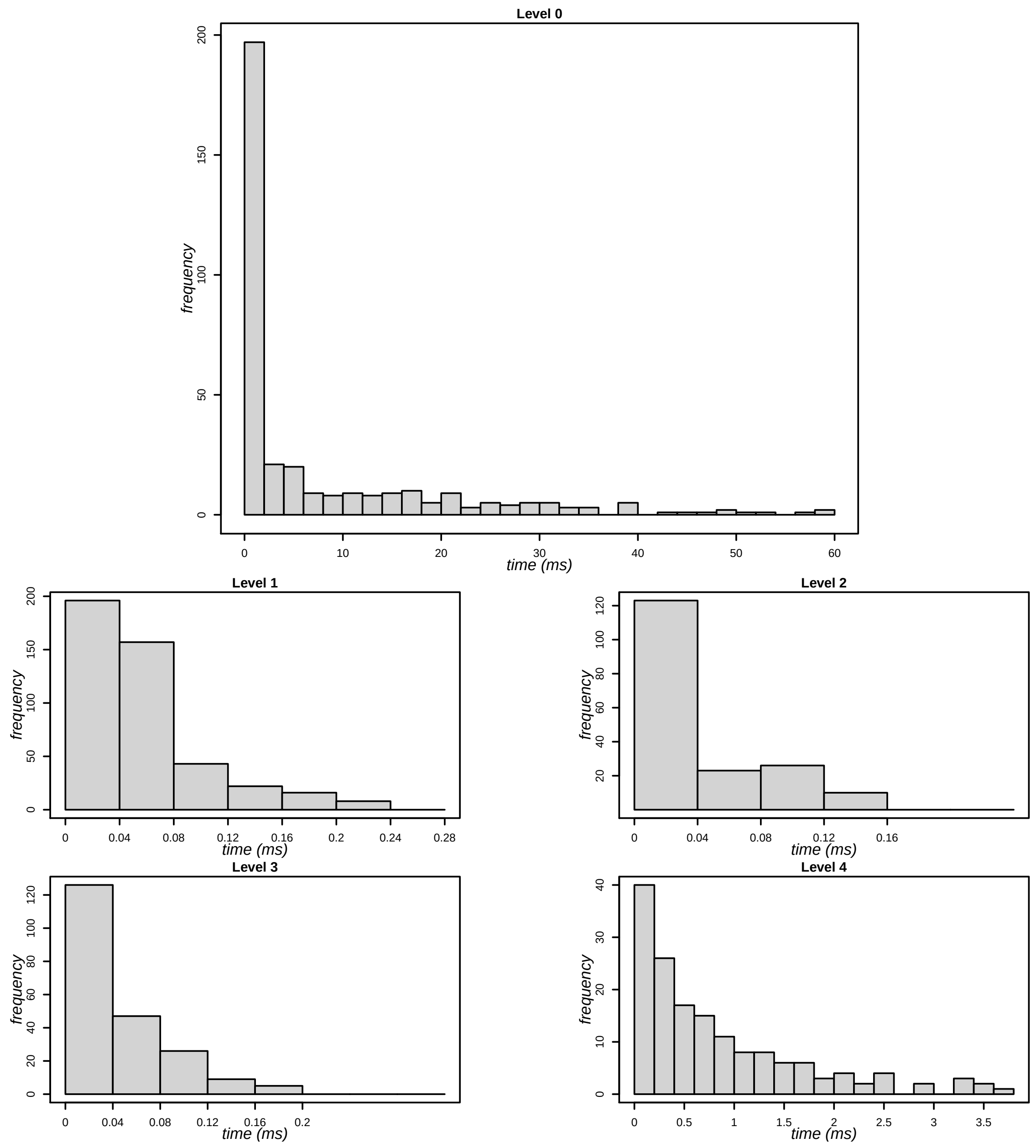


Figure 6.4: Figure produce in Example 2.

`pdf()` Produces a PDF file, which can also be included into PDF files.

`png()` Produces a bitmap PNG file. (Not always available: see its help page.)

`jpeg()` Produces a bitmap JPEG file, best used for image plots. (Not always available: see its help page.)

When you have finished with a device, be sure to terminate the device driver by issuing the command

```
dev.off()

## postscript
##          5
```

This ensures that the device finishes cleanly; for example in the case of hardcopy devices this ensures that every page is completed and has been sent to the printer. (This will happen automatically at the normal end of a session.)

6.11 The package `lattice`

Figure 6.1 shows some histograms of simulated data from and exponential distribution. The same histograms can be plotted with one command as a multipanel plot using the `lattice` library, as in Figure 6.5 below.

For more details on this package see

```
help(lattice)
```

6.12 Exercises

1. The following commands load the `DAAG` package and extract from the `possum` dataset in that package a vector of the lengths of tails of possums.

   ```
   library(DAAG)

   ## Warning:  package 'DAAG' was built under R version 4.0.5

   data(possum)
   tails <- possum$taill
   ```

   (a) Plot a histogram of the tail data, using 20 class intervals.

   (b) The position of the cell boundaries can be controlled by the `breaks` argument. Plot two histograms on one plot, one with break points starting at 31.5 and going up in intervals of 1.5, the other with break points starting at 31 and going up in intervals of 2. Also limit the y-axis appropriately to fit the range of frequency values.

```
options(width=50)
library(lattice)
       ## Warning:  package 'lattice' was built under R version 4.0.5

par(cex=0.7)
rmt <- matrix(rexp(1000 * 16), nrow = 16)
mns <- cbind(rmt[ 1, ],  apply(rmt[ 1:4, ], 2, mean), apply(rmt[ 1:16, ], 2, mean)
)
print(histogram(~ mns | ssz,
data = data.frame(mns = c(mns),ssz = gl(3, 1000,
labels = c("1", "4", "16"))),
layout = c(3, 1),
main = "Histograms of means by sample
size"))
```

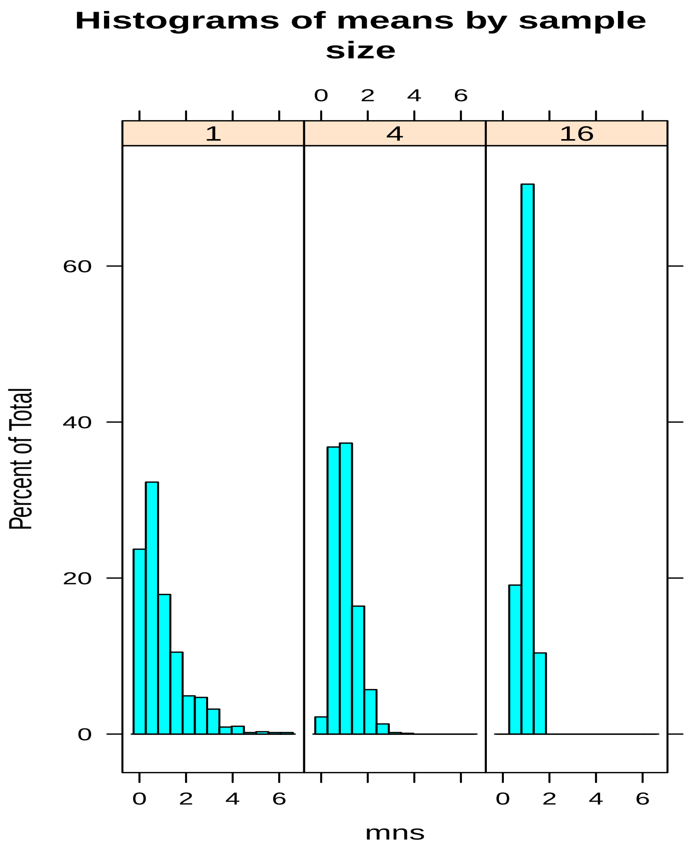


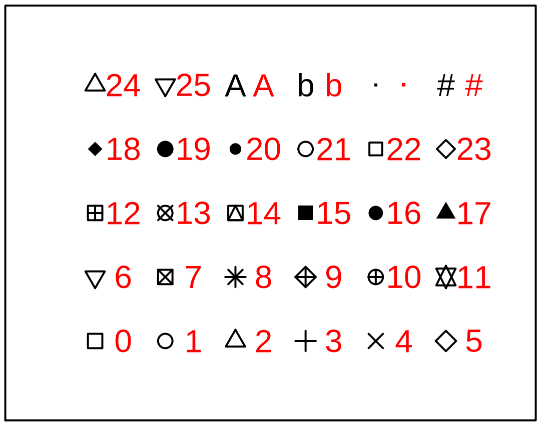
Figure 6.5: Histogram using `lattice` functions.

(c) Comment on the visual impression of the distribution of tail length that you get from each histogram.

2. The `primate` dataset in the `DAAG` package contains the brain weight and the body weight of five primates. Load the data into `R` (see the last exercise). Produce a scatter plot of the brain weights against body weights with the following specifications.

   - Limit the x-axis range to $(0, 300$ and the y-range to $(0, 1500)$ so there is space for labels.
   - Put the axes labels as `Body weight (kg)` and `Brain weight (g)` appropriately.
   - Plot the points as filled dots.
   - Next to the points, print the name of the primate it corresponds to.

3. Produce the following plot.



4. For the `battery` data from section 3, produce appropriately labelled boxplots of the lifetimes against material type and temperature.

6.13 Final comments

There is a lot of detail in this section, and we will get practice with the common ones in the laboratory classes and in further work. This section should serve as a reasonably comprehensive guide and reference to graphics in `R`. In particular, we have not discussed the `ggplot2` package, which produces very good graphics and allows user control of all the features of graphics.

Exploratory data analysis should precede any formal data analysis. At least some appropriate graphs and summary statistics should be obtained. These will provide insight into data and help identify and data anomalies. Commonly, a seven (or five) point data summary is provided, including the minimum, lower quartile, median, mean, upper quartile, maximum and standard deviation. (The five point summary does not include the mean and standard deviation).

In describing data, point out any key features of the data, and any unusual aspects. Features such as central tendency, spread or variability, skewness and any groupings should be highlighted. Particular data may be expected to have certain features. For example, if we have data on per capita income for Indonesia, Singapore, Australia and New Zealand, then we would expect a histogram of this data to exhibit some groupings. If groupings were not observed in the histogram then this would be surprising and would demand an explanation.

7 Parametric Models

7.1 Introduction

R has the full suite of statistical methods, either in the base or other packages. Several specialist packages are also available, such as bioconductor for analysis of microarray data and spatstat for analysis of spatial data. We will cover the basic univariate techniques first, including model diagnostics, before covering Linear Models and Generalised Linear Models GLMs).

This section discusses parametric models, beginning with univariate models, then General Linear Models and finally Generalised Linear Models (GLMs), including model diagnostics and remedial action. The details of all the commands used below can be found using the help facility in R.

Fitting any model in R produces an object of a corresponding class. This model object can be interrogated to obtain information about the model. This is a very powerful feature of R. Using the help facility information on model that is fitted can be obtained.

7.2 Sampling Distribution of the Sample Mean

Take a random sample $X_1, X_2, , \ldots, X_n$ from a population with mean μ and standard deviation σ. Then the sample mean $\overline{X}$ has mean μ and variance σ^2/n or standard deviation $\sigma/\sqrt{n}$. We want to make inferences about the population mean μ based on the sample mean $\overline{X}$. For this we first need the distribution of $\overline{X}$ so that we can evaluate relevant probabilities.

1. If the population is normally distributed then the sample mean also has a normal distribution, that is,

$$\overline{X} \sim N\left(\mu, \sigma^2/n\right) \quad \text{or} \quad \frac{\overline{X}-\mu}{\sigma/\sqrt{n}} \sim N(0,1)$$

.

2. If the population is normally distributed but the population variance σ^2 s unknown, then we estimate it by the sample variance

$$S^2 = \frac{1}{n-1}\sum_{i=1}^{n}\left(X_i - \overline{X}\right)^2 .$$

Then

$$\frac{\overline{X}-\mu}{S/\sqrt{n}} \sim t_{n-1}.$$

3. If the population is not normal but the sample size is large ($n \geq 30$) then by the *Central Limit Theorem*

$$\frac{\overline{X}-\mu}{\sigma/\sqrt{n}} \mathrel{\dot{\underset{\cdot}{\sim}}} N(0,1) \quad \text{or} \quad \frac{\overline{X}-\mu}{S/\sqrt{n}} \mathrel{\dot{\underset{\cdot}{\sim}}} t_{n-1}.$$

Note that in most cases the population variance in unknown and needs to be estimated from sample data.

7.3 One sample t-test

Consider the following data on the heights and genders of people. We are interested in testing if the heights of the men are more than 69 inches. The hypotheses of interest are:

$$H_0 : \mu = 69 \qquad H_1 : \mu > 69$$

Note that the null hypothesis always states equality, and the alternative is appropriately formulated depending on the question of interest. We test the hypotheses in R using the function `t.test()` as follows.

```
height<-read.table("Data/height.txt",header=T)
summary(height)

##       Men             Women      
##  Min.   :63.31   Min.   :58.17  
##  1st Qu.:69.04   1st Qu.:63.48  
##  Median :70.56   Median :64.69  
##  Mean   :70.23   Mean   :64.70  
##  3rd Qu.:72.13   3rd Qu.:65.69  
##  Max.   :74.85   Max.   :69.85  

sapply(height, sd, na.rm = TRUE)

##      Men    Women 
## 2.776952 2.405681

ht.t<-t.test(height$Men,mu=69,alternative="greater")
ht.t

## 
##  One Sample t-test
## 
## data:  height$Men
## t = 2.2119, df = 24, p-value = 0.01837
## alternative hypothesis: true mean is greater than 69
## 95 percent confidence interval:
##  69.27828      Inf
## sample estimates:
## mean of x 
##  70.22849
```

Note how R computes the **one-sided confidence interval**: for an upper-tail test,

$$\text{Conf Int} = \left(\overline{\mathrm{x}} - \mathrm{t}_{\mathrm{n}-1}(\alpha)\ \frac{\mathrm{s}}{\sqrt{\mathrm{n}}}, \infty\right)$$

where $t_{n-1}(\alpha)$ is the upper tail critical value of the t-distribution with $n-1$ degrees of freedom. Similarly for a lower-tail test,

$$\mathrm{Conf\ Int} = \left(-\infty, \bar{\mathrm{x}} + \mathrm{t}_{\mathrm{n}-1}(\alpha)\ \frac{\mathrm{s}}{\sqrt{\mathrm{n}}}\right)$$

```r
options(width=50)
par(mfrow=c(2,1),cex=0.6)
res<-height$Men-69
#£
hist(res,breaks=c(-7,-5,-3,-1,1,3,5,7),
main="Histogram of residuals for the height data")
box()
qqnorm(res,main="Q-Q plot of residuals of height data")
```

The model diagnostics show that there is no evidence against the assumption that the heights are normally distributed. We can also compute the power of the test and the sample size required for a specified effect size and power.

```r
power.t.test(delta=2.0,sd=2.777,power=0.95,type="one.sample",alternative="two.sided")

##
##      One-sample t test power calculation
##
##               n = 27.05577
##           delta = 2
##              sd = 2.777
##       sig.level = 0.05
##           power = 0.95
##     alternative = two.sided

power.t.test(n=25,delta=2.0,sd=2.777,type="one.sample",alternative="two.sided")

##
##      One-sample t test power calculation
##
##               n = 25
##           delta = 2
##              sd = 2.777
##       sig.level = 0.05
##           power = 0.9323566
##     alternative = two.sided
```

For power calculations, any one of `n,delta,power,sd` and `sig.level` must be passed as `NULL` and will be determined from the other parameters.

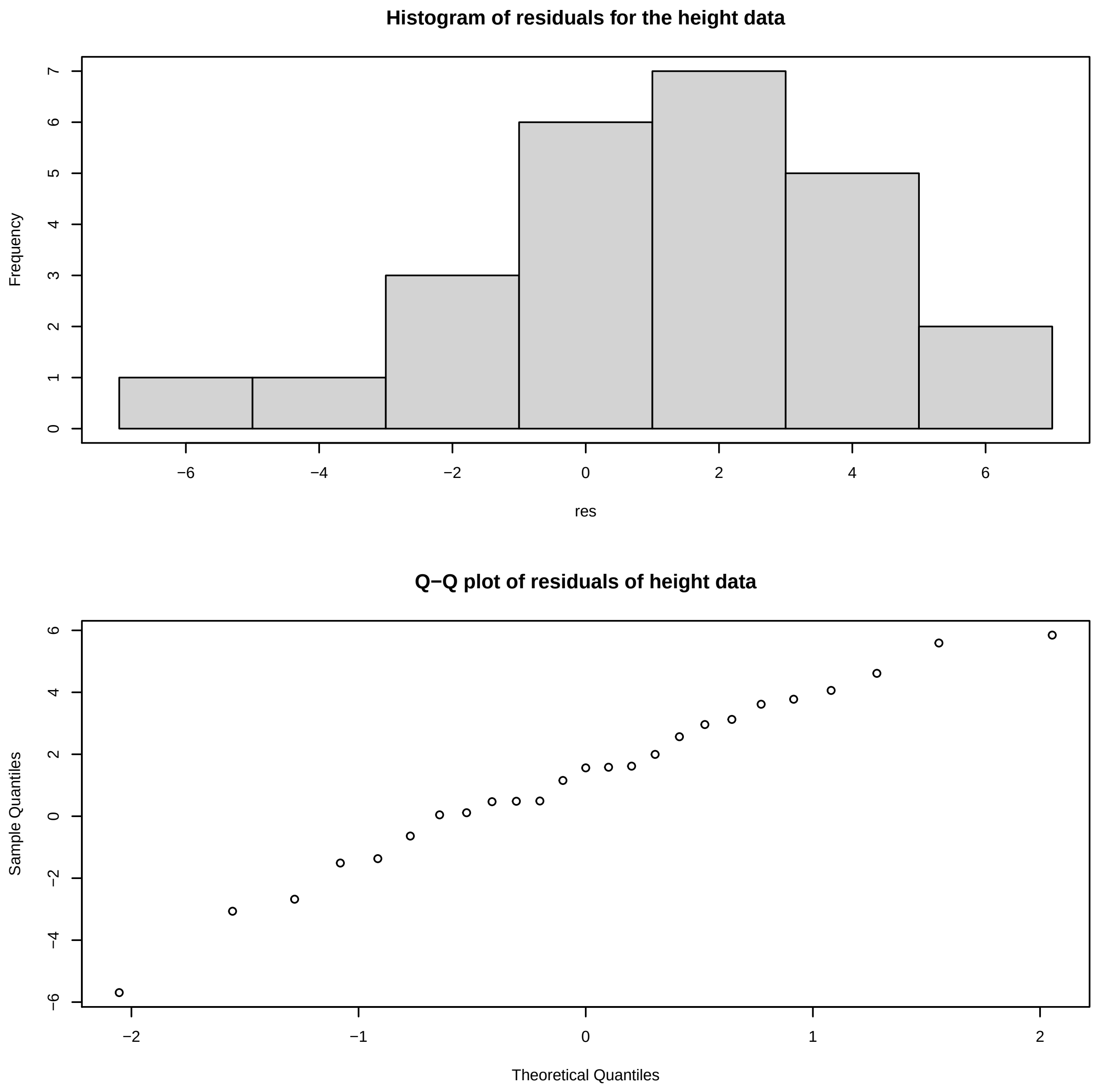


Figure 7.1: Diagnostic of residuals for the height data.

7.4 Paired-samples *t*-test

When two samples are observations on the same subjects, or matched subjects, then a paired-samples t-test is appropriate. The method here is similar to a single sample t-test. The data are differenced first, following which a single sample t-test is performed. In `R` (t.test() can be used with the argument `paired=TRUE)` — the default value of this argument is `FALSE`.

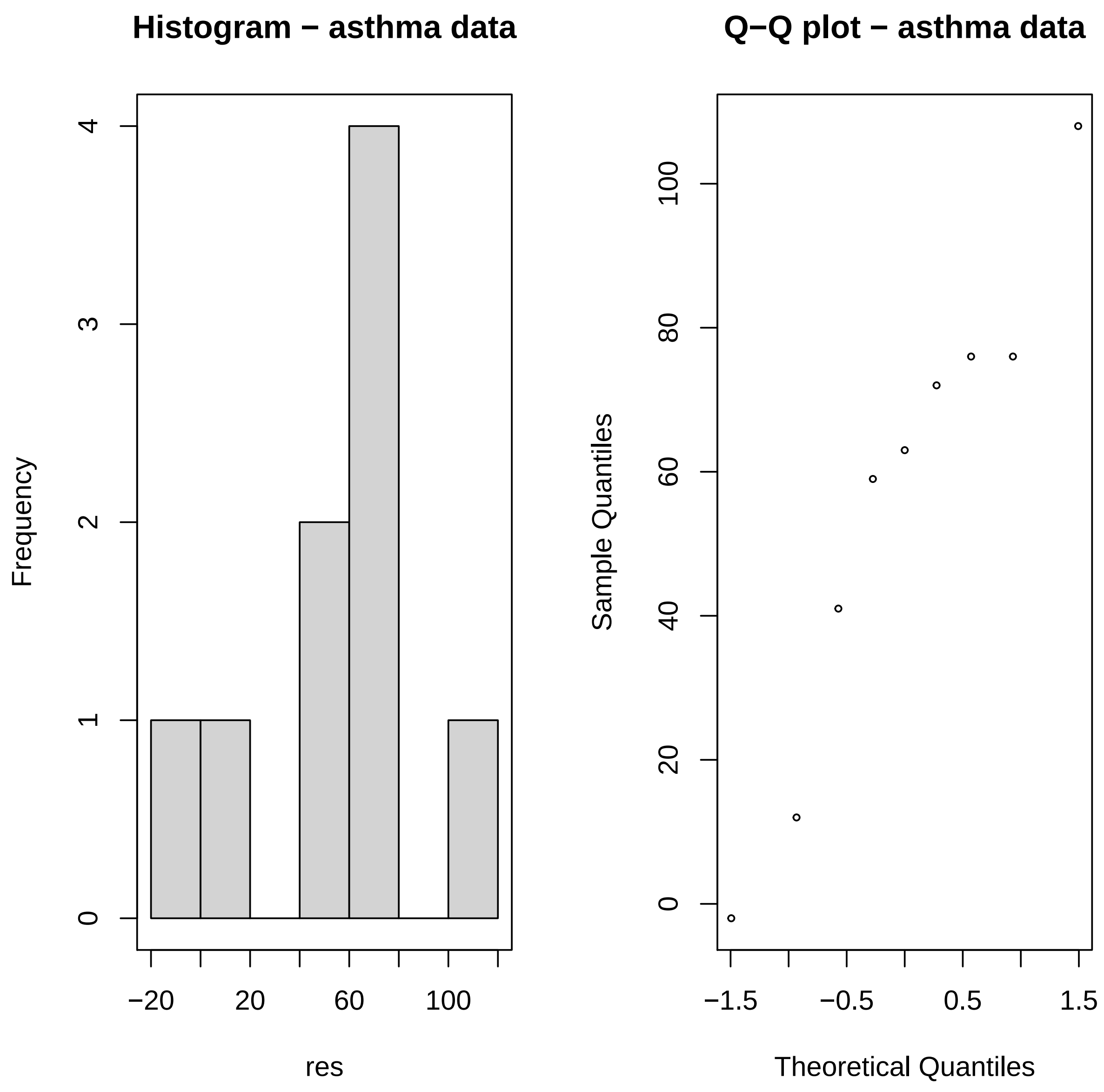
7.4.1 Example

An experiment Compares the peak expiratory flow rate (PEFR) before and after a walk on a cold winter's day for a random sample of nine asthmatics. The data is given below.

Subject	1	2	3	4	5	6	7	8	9
Before	312	242	340	388	296	254	391	402	290
After	300	201	232	312	220	256	328	330	231

```
options(width=70)
Subject<-c(1:9)
Before<-c(312, 242, 340, 388, 296, 254, 391, 402, 290)
After<-c(300, 201, 232, 312, 220, 256, 328, 330, 231)
summary(Before)

##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   242.0   290.0   312.0   323.9   388.0   402.0

summary(After)

##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   201.0   231.0   256.0   267.8   312.0   330.0

summary(After-Before)

##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
## -108.00  -76.00  -63.00  -56.11  -41.00    2.00

t.test(Before,After,paired=TRUE,mu=0)

##
##  Paired t-test
##
## data:  Before and After
## t = 4.9258, df = 8, p-value = 0.001156
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  29.84266 82.37956
## sample estimates:
## mean of the differences
##                 56.11111
```

```r
res<-Before-After
par(mfrow=c(1,2))
hist(res,main="Histogram - asthma data", cex=0.5)
box()
qqnorm(res,main="Q-Q plot - asthma data",cex=0.5)
```



7.5 Two sample *t*-test

```
ht.t2<-t.test(height$Men,height$Women,alternative="greater",var.equal=T)
ht.t2

##
##  Two Sample t-test
##
## data:  height$Men and height$Women
## t = 7.5212, df = 48, p-value = 5.869e-10
## alternative hypothesis: true difference in means is greater than 0
## 95 percent confidence interval:
##  4.294199      Inf
## sample estimates:
## mean of x mean of y
##  70.22849  64.70184

res2<-c(height$Men-mean(height$Men),height$Women-mean(height$Women))
```

From the diagnostics the assumption of normality seems to be satisfied. Also, there is no evidence against the common variance assumption. A formal test of equality of variance may be performed, as below.

```
var.test(height$Men,height$Women)

##
##  F test to compare two variances
##
## data:  height$Men and height$Women
## F = 1.3325, num df = 24, denom df = 24,
## p-value = 0.4873
## alternative hypothesis: true ratio of variances is not equal to 1
## 95 percent confidence interval:
##  0.5871824 3.0237660
## sample estimates:
## ratio of variances
##            1.33248
```

Note that the observed ratio of variances is 1.33 and yet the p-value is very large. As another illustration, consider the following.

```
x<-rnorm(100,0,1)
y<-rnorm(100,0,2)
var.test(y,x)
```

```
options(width=50)
par(mfrow=c(1,2),cex=0.6)
hist(res2,breaks=c(-9,-7,-5,-3,-1,1,3,5,7),
main="Histogram of residuals")
box()
qqnorm(res2,main="Q-Q plot of residuals")
```

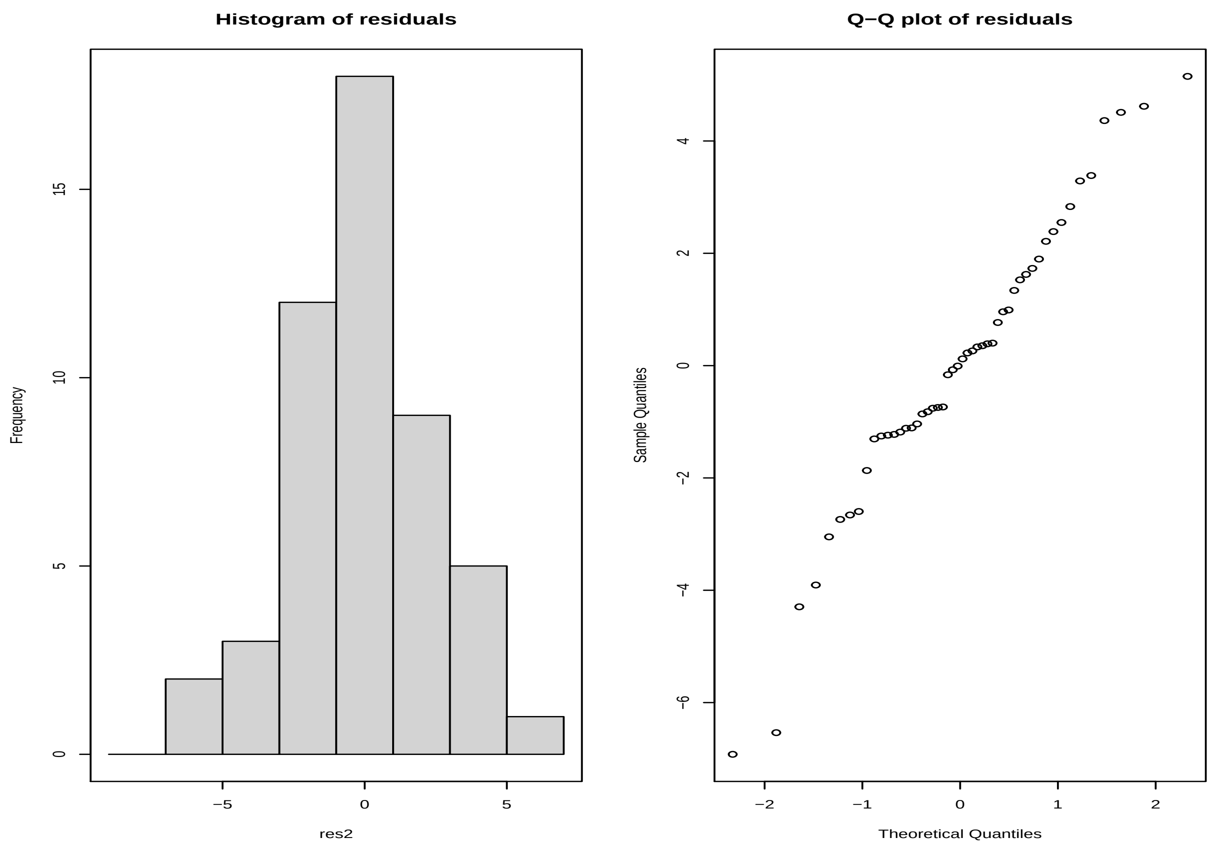


Figure 7.2: Diagnostic of residuals for two sample t-test.

```
##
##  F test to compare two variances
##
## data:  y and x
## F = 3.7025, num df = 99, denom df = 99,
## p-value = 3.291e-10
## alternative hypothesis: true ratio of variances is not equal to 1
## 95 percent confidence interval:
##  2.491172 5.502727
## sample estimates:
## ratio of variances
##           3.702464
```

7.6 Test for binomial proportions

1. **Single binomial proportion**

   Below is a test for a single binomial proportion.

```
## Conover (1971), p. 97f.
## Under (the assumption of) simple Mendelian inheritance, a cross
## between plants of two particular genotypes produces progeny 1/4 of
## which are "dwarf" and 3/4 of which are "giant", respectively.
## In an experiment to determine if this assumption is reasonable, a
## cross results in progeny having 243 dwarf and 682 giant plants.
## If "giant" is taken as success, the null hypothesis is that p =
## 3/4 and the alternative that p != 3/4. Note that this is an exact
##binomial test.
##In the first example, the first argument is a vector specifying the
##number of successes
##and the number of failures.
binom.test(c(682, 243), p = 3/4)

##
##  Exact binomial test
##
## data:  c(682, 243)
## number of successes = 682, number of trials
## = 925, p-value = 0.3825
## alternative hypothesis: true probability of success is not equal to 0.75
## 95 percent confidence interval:
##  0.7076683 0.7654066
## sample estimates:
## probability of success
##              0.7372973
```

```r
##Now the first argument gives the number of successes and the second
##the number of trials.
binom.test(682, 682 + 243, p = 3/4)

##
##  Exact binomial test
##
## data:  682 and 682 + 243
## number of successes = 682, number of trials
## = 925, p-value = 0.3825
## alternative hypothesis: true probability of success is not equal to 0.75
## 95 percent confidence interval:
##  0.7076683 0.7654066
## sample estimates:
## probability of success
##              0.7372973
```

2. **Test of two or more binomial proportions**

   Below a chi-square test is used with the expected counts computed under the null hypothesis. The overall proportion of smokers is computed from the data.

```r
## Data from Fleiss (1981), p. 139.
## H0: The null hypothesis is that the four populations from which
##     the patients were drawn have the same true proportion of smokers.
## A:  The alternative is that this proportion is different in at
##     least one of the populations.

smokers  <- c( 83, 90, 129, 70 )
patients <- c( 86, 93, 136, 82 )
prop.test(smokers, patients)

##
##  4-sample test for equality of proportions
##  without continuity correction
##
## data:  smokers out of patients
## X-squared = 12.6, df = 3, p-value = 0.005585
## alternative hypothesis: two.sided
## sample estimates:
##    prop 1    prop 2    prop 3    prop 4
## 0.9651163 0.9677419 0.9485294 0.8536585
```

7.7 Some Examples

7.7.1 Example 1

A tyre company has found that the mean time required for a mechanic to replace a set of four tyres is 18 minutes. A new machine and installation procedure is expected to reduce this time. A random sample of 40 mechanics using the new setup to replace a set of 4 tyres produced the following data.
18.2, 11.7, 16.3, 19.2, 16.0, 17.0, 13.3, 16.7, 20.1, 17.8, 14.9, 17.4, 15.1, 17.1, 16.5, 14.3, 14.8, 16.7, 16.2, 19.2, 15.5, 13.4, 17.4, 11.1, 15.9, 16.0, 14.6, 16.5, 15.6, 16.7, 16.2, 16.1, 18.3, 19.3, 16.4, 14.9, 13.1, 16.0, 16.9

Has the new set up reduced the mean time required to replace a set of tyres?

Solution Let μ denote the mean time to replace a set of four tyres. The hypotheses of interest are:

$$H_0 : \mu = 18 \qquad H_1 : \mu < 18$$

The analysis is performed in `R` and the output is given below.

```
options(width=70)
tyre<-c(18.2, 11.7, 16.3, 19.2, 16.0, 17.0, 13.3, 16.7, 20.1, 17.8, 14.9, 17.4, 15.1, 17.1,
14.3, 14.8, 16.7, 16.2, 19.2, 15.5, 13.4, 17.4, 11.1, 15.9, 16.0, 14.6, 16.5, 15.6, 16.7,
16.2, 16.1, 18.3, 19.3, 16.4, 14.9, 13.1, 16.0, 16.9)
summary(tyre)

##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   11.10   15.00   16.20   16.11   17.05   20.10

#Summary statistics to give some ideas of data behaviour.
t.test(tyre, mu=18, alternative="less")

##
##  One Sample t-test
##
## data:  tyre
## t = -6.029, df = 38, p-value = 2.597e-07
## alternative hypothesis: true mean is less than 18
## 95 percent confidence interval:
##      -Inf 16.64055
## sample estimates:
## mean of x
##  16.11282
```

The p-value is much less than 0.025, so the data provides very strong evidence against the null hypothesis. We conclude that the data indicates that the new set up has reduced the mean time to replace a set of four tyres.

Important Note For a two-tailed test the default significance level is 0.05, and for a one-tailed test we use a half of this, that is, 2.5% level of significance as a default. This is for reasons of consistency. Otherwise, for example, if the p-value for a two-tailed test is say 0.06, then we fail to reject the null hypothesis at the 5% level of significance. However, the p-value for a corresponding one-tailed test is 0.03, which is less than 0.05, so we will reject the null hypothesis in this case and conclude that the mean is different from the null value. This contradiction is avoided if we halve the default p-value for a one-tailed test.

7.7.2 Further examples

See separate handout.

8 Linear Models

Linear models are an important and useful tool in applied statistics. Linear models comprise wide range of techniques, including Anova, Regression, General Linear Models and Generalised Linear Models (GLMs). Model building is an important aspect of linear modelling, and writing the model equation is usually a start to model fitting.

In R, the function `lm()` is used to fit a linear model, and this produces an object of class `lm`. Table 8.1 gives examples of model fitting using the function `lm()`.

An object of class "lm" is a list containing at least the following components.

coefficients : a named vector of coefficients

residuals : the residuals, that is response minus fitted values.

fitted.values : the fitted mean values.

rank : the numeric rank of the fitted linear model.

weights : (only for weighted fits) the specified weights.

df.residual : the residual degrees of freedom.

call : the matched call.

terms : the terms object used.

contrasts : (only where relevant) the contrasts used.

xlevels : (only where relevant) a record of the levels of the factors used in fitting.

offset : the offset used (missing if none were used).

y : if requested, the response used.

x : if requested, the model matrix used.

model : if requested (the default), the model frame used.

na.action : (where relevant) information returned by model.frame on the special handling of NAs.

In addition, non-null fits will have components `assign, effects` and (unless not requested) `qr` relating to the linear fit, for use by extractor functions such as `summary` and `effects`.

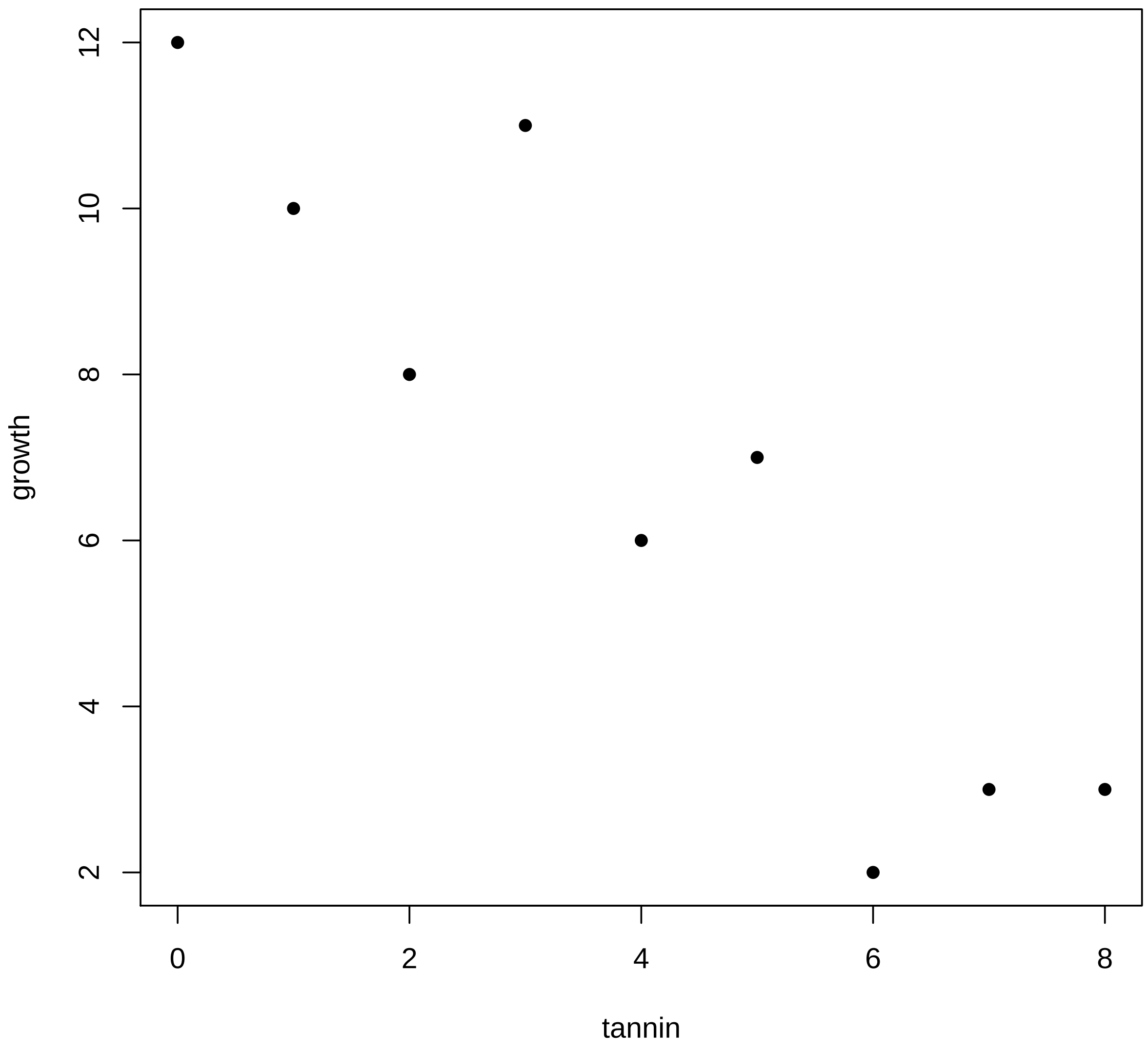
The model object can be interrogated by `extractor` functions to obtain information regarding the model. Other functions can also be applied to the model object. The example below illustrates the use of some functions that operate on the `lm` model object.

```
cat<-read.table("Data/tannin.txt",header=T)
cat
```

```
##   growth tannin
## 1     12      0
## 2     10      1
## 3      8      2
## 4     11      3
## 5      6      4
## 6      7      5
## 7      2      6
## 8      3      7
## 9      3      8

#Data is on growth of caterpillars fed on diet with differing tannin content.
attach(cat)
```

```
cat.lm<-lm(growth~tannin,data=cat)
summary(cat.lm)

##
## Call:
## lm(formula = growth ~ tannin, data = cat)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -2.4556 -0.8889 -0.2389  0.9778  2.8944
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  11.7556     1.0408  11.295 9.54e-06 ***
## tannin       -1.2167     0.2186  -5.565 0.000846 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.693 on 7 degrees of freedom
## Multiple R-squared:  0.8157,Adjusted R-squared:  0.7893
## F-statistic: 30.97 on 1 and 7 DF,  p-value: 0.0008461
```

```
plot(growth~tannin,pch=16,xlab="tannin",ylab="growth")
```



Model	Model Formula	Comments
Null	$y \sim 1$	1 is the intercept in a regression model, and here this model fits just the mean of y. This is equivalent to a single sample `t-test`.
Regression	$y \sim x$	x is a continuous explanatory variable.
On-way Anova	$y \sim gender$	gender is a two-level categorical variable.
Two-way Anova	$ysimgender + genotype$	genotype is a four-level categorical variable.
Factorial Anova	$y \sim N * P * K$	N, P and K are two-level factors to be fitted along with all their interactions.
Three-way Anova	$y \sim N * P * K - N : P : K$	As above, but do not fit the three-way interaction.
Analysis of covariance	$y \sim x + gender$	A common slope for y against x, but with two intercepts, one for each gender.
Analysis of covariance	$y \sim x * gender$	Two slopes and two intercepts.
Nested Anova	$y \sim a/b/c$	Factor c is nested within factor b within factor a.
Split-plot Anova	$y \sim A * B * C + Error(a/b/c)$	A factorial experiment but with three plot sizes and three different error variances, one for each plot size.
Multiple regression	$y \sim x + z$	Two continuous explanatory variables.
Multiple regression	$y \sim x * z$	Fit an interaction term as well. $(x+z+x : z)$
Multiple regression	$y \sim x + l(x^2) + z + l(z^2)$	Fit a quadratic term for each of x and z.
Multiple regression	$y < -poly(x, 2) + z$	Fit a quadratic polynomial for x and linear z.
Multiple regression	$y \sim (x + z + w)^2$	Fit three variables plus all their two-way interactions.
Non-parametric model	$y \sim s(x) + lo(z)$	y is a function of smoothed x and `loess` z.
Transformed response and explanatory variables	$\log(y)\ \log(1/x) + sqrt(z)$	All three variables are transformed in the model.

Table 8.1: Examples of `R` model formulae