Chapter 1
Introduction

R is a statistical computing environment. It is free (open source) software for statistical computation and graphics [40] and a computer language designed for typical statistical and graphical applications. The R distribution includes the ability to save and run commands stored in script files, and an integrated editor in the R Graphical User Interface (R-GUI). It is available for most platforms including unix/linux, PC, and Macintosh platforms. Thousands of contributed packages are available, and users are provided tools to make packages.

At the core of R is an interpreted computer language. This language provides the logical control of branching and looping, and modular programming using functions. The base R distribution contains functions and data to implement and illustrate most common statistical procedures, including regression and ANOVA, classical parametric and nonparametric tests, cluster analysis, density estimation, and much more. An extensive suite of probability distribution functions and generators are provided, as well as a graphical environment for exploratory data analysis and creating presentation graphics.

On the history and evolution of R, see the R-FAQ [26] and resources on the R home page at http://www.R-project.org/.

1.1 Getting Started

R is an interpreted language; that is, the system processes commands entered by the user, who types the commands at the command prompt, or submits the commands from a file called a script. We assume that our readers use R at a graphics workstation running a windowing system, such as Windows, Macintosh, or X window systems. In a window system, users interact with R through the R console. Except for the simplest operations, most users will prefer to type commands in a script (see Section 1.1.3) to save retyping and

J. Albert and M. Rizzo, *R by Example*, Use R, DOI 10.1007/978-1-4614-1365-3_1,

to separate commands from results. However, let us begin by working directly at the command prompt.

When we use the command line interface, each command or expression to be evaluated is typed at the command prompt, and immediately evaluated when the Enter key is pressed at the end of a syntactically complete statement. It is helpful to remember the following tips.

- Press the up-arrow key to recall commands and edit them.
- Use the Esc (Escape) key to cancel a command.

1.1.1 Preliminaries

Remarks or tips about R are identified by the symbol $\mathbf{R}_\mathbf{x}$ to set them apart from the main text.

$\mathbf{R}_\mathbf{x}$ 1.1 *The right-to-left assignment operators are the left arrow* `<-` *and equal sign* `=`*. For example, borrowing a line from Example 1.3, either method below*

```
> x = c(109, 65, 22, 3, 1)
> x <- c(109, 65, 22, 3, 1)
```

creates the vector $(109, 65, 22, 3, 1)$ *and assigns it to* `x`*. Borrowing another line from Example 1.3, either method below*

```
> y = rpois(200, lambda=.61)
> y <- rpois(200, lambda=.61)
```

assigns the result of the `rpois` *function to* `y`*. Notice that the equal sign inside the parentheses is not an assignment operator; it passes the value of an argument (lambda) to the function* `rpois`*.*

The R manuals and examples in the help files use the arrow assignment operators `<-` and `->`. However, in this book we have used the equal sign `=` operator for assignment, rather than `<-`, as novice users may find it easier to type the `=` symbol.

In the examples, R code and output appears in bold monospaced type as in the remark $\mathbf{R}_\mathbf{x}$ 1.1 above. Code that would be typed interactively by the user or submitted from an R script is identified by the leading prompt symbol `>`. Scripts for some of the functions in the examples are provided in files available from the book web site; these functions are shown in the book without the prompt character.

Data files and scripts used in the examples are available on our web site at `personal.bgsu.edu/~mrizzo/Rx`. Some data files can be downloaded directly from a connection to a url.

1.1.2 Basic operations

Some basic operations with vectors are illustrated in the following example. The R commands are entered at the prompt in the R console window. The prompt character is > and when a line is continued the prompt changes to +. (The prompt symbols can be changed.)

Example 1.1 (Temperature data). Average annual temperatures in New Haven, CT, were recorded in degrees Fahrenheit, as

Year	1968	1969	1970	1971
Mean temperature	51.9	51.8	51.9	53

(This data is part of a larger data set in R called *nhtemp*.) The combine function `c` creates a vector from its arguments, and the result can be stored in user-defined vectors. We use the combine function to enter our data and store it in an object named `temps`.

```
> temps = c(51.9, 51.8, 51.9, 53)
```

To display the value of `temps`, one simply types the name.

```
> temps
[1] 51.9 51.8 51.9 53.0
```

Suppose that we want to convert the Fahrenheit temperatures (F) to Celsius temperatures (C). The formula for the conversion is $C = \frac{5}{9}(F - 32)$. It is easy to apply this formula to all of the temperatures in one step, because arithmetic operations in R are *vectorized*; operations are applied element by element. For example, to subtract 32 from every element of `temp`, we use

```
> temps - 32
[1] 19.9 19.8 19.9 21.0
```

Then `(5/9)*(temps - 32)` multiplies each difference by 5/9. The temperatures in degrees Celsius are

```
> (5/9) * (temps - 32)
[1] 11.05556 11.00000 11.05556 11.66667
```

In 1968 through 1971, the mean annual temperatures (Fahrenheit) in the state of Connecticut were 48, 48.2, 48, 48.7, according to the National Climatic Center Data web page. We store the state temperatures in `CT`, and compare the local New Haven temperatures with the state averages. For example, one can compute the annual differences in mean temperatures. Here `CT` and `temps` are both vectors of length four and the subtraction operation is applied element by element. The result is the vector of four differences.

```
> CT = c(48, 48.2, 48, 48.7)
> temps - CT
[1] 3.9 3.6 3.9 4.3
```

The four values in the result are differences in mean temperatures for 1968 through 1971. It appears that on average New Haven enjoyed slightly warmer temperatures than the state of Connecticut in this period.

Example 1.2 (President's heights). An article in Wikipedia [54] reports data on the heights of Presidents of the United States and the heights of their opponents in the presidential election. It has been observed [53, 48] that the taller presidential candidate typically wins the election. In this example, we explore the data corresponding to the elections in the television era. In Table 1.1 are the heights of the presidents and their opponents in the U.S. presidential elections of 1948 through 2008, extracted from the Wikipedia article.

Table 1.1 Height of the election winner in the Electoral College and height of the main opponent in the U.S. Presidential elections of 1948 through 2008.

Year	Winner	Height		Opponent	Height	
2008	Barack Obama	6 ft 1 in	185 cm	John McCain	5 ft 9 in	175 cm
2004	George W. Bush	5 ft 11.5 in	182 cm	John Kerry	6 ft 4 in	193 cm
2000	George W. Bush	5 ft 11.5 in	182 cm	Al Gore	6 ft 1 in	185 cm
1996	Bill Clinton	6 ft 2 in	188 cm	Bob Dole	6 ft 1.5 in	187 cm
1992	Bill Clinton	6 ft 2 in	188 cm	George H.W. Bush	6 ft 2 in	188 cm
1988	George H.W. Bush	6 ft 2 in	188 cm	Michael Dukakis	5 ft 8 in	173 cm
1984	Ronald Reagan	6 ft 1 in	185 cm	Walter Mondale	5 ft 11 in	180 cm
1980	Ronald Reagan	6 ft 1 in	185 cm	Jimmy Carter	5 ft 9.5 in	177 cm
1976	Jimmy Carter	5 ft 9.5 in	177 cm	Gerald Ford	6 ft 0 in	183 cm
1972	Richard Nixon	5 ft 11.5 in	182 cm	George McGovern	6 ft 1 in	185 cm
1968	Richard Nixon	5 ft 11.5 in	182 cm	Hubert Humphrey	5 ft 11 in	180 cm
1964	Lyndon B. Johnson	6 ft 4 in	193 cm	Barry Goldwater	5 ft 11 in	180 cm
1960	John F. Kennedy	6 ft 0 in	183 cm	Richard Nixon	5 ft 11.5 in	182 cm
1956	Dwight D. Eisenhower	5 ft 10.5 in	179 cm	Adlai Stevenson	5 ft 10 in	178 cm
1952	Dwight D. Eisenhower	5 ft 10.5 in	179 cm	Adlai Stevenson	5 ft 10 in	178 cm
1948	Harry S. Truman	5 ft 9 in	175 cm	Thomas Dewey	5 ft 8 in	173 cm

Section 1.5 illustrates several methods for importing data from a file. In this example we enter the data interactively as follows. The continuation character + indicates that the R command is continued.

```
> winner = c(185, 182, 182, 188, 188, 188, 185, 185, 177,
+   182, 182, 193, 183, 179, 179, 175)
> opponent = c(175, 193, 185, 187, 188, 173, 180, 177, 183,
+   185, 180, 180, 182, 178, 178, 173)
```

(Another method for entering data interactively is to use the `scan` function. See Example 3.1 on page 79.) Now the newly created objects `winner` and `opponent` are each vectors of length 16.

```
> length(winner)
[1] 16
```

The year of the election is a regular sequence, which we can generate using the sequence function `seq`. Our first data value corresponds to year 2008, so the sequence can be created by

```
> year = seq(from=2008, to=1948, by=-4)
```

or equivalently by

```
> year = seq(2008, 1948, -4)
```

According to the Washington Post blog [53], Wikipedia misstates "Bill Clinton's height, which was measured during official medical exams at 6 foot-2-1/2, making him just a tad taller than George H.W. Bush." We can correct the height measurement for Bill Clinton by assigning a height of 189 cm to the fourth and fifth entries of the vector `winner`.

```
> winner[4] = 189
> winner[5] = 189
```

The sequence operator `:` allows us to perform this operation in one step:

```
> winner[4:5] = 189
```

The revised values of `winner` are

```
> winner
 [1] 185 182 182 189 189 188 185 185 177 182 182 193 183 179 179 175
```

Are presidents taller than average adult males? According to the National Center for Health Statistics, in 2005 the average height for an adult male in the United States is 5 feet 9.2 inches or 175.768 cm. The sample mean is computed by the `mean` function.

```
> mean(winner)
[1] 183.4375
```

Interestingly, the opponents also tend to be taller than average.

```
> mean(opponent)
[1] 181.0625
```

Next, we use vectorized operations to compute the differences in the height of the winner and the main opponent, and store the result in `difference`.

```
> difference = winner - opponent
```

An easy way to display our data is as a data frame:

```
> data.frame(year, winner, opponent, difference)
```

The result is displayed in Table 1.2. Data frames are discussed in detail in Section 1.4.

We see that most, but not all, of the differences in height are positive, indicating that the taller candidate won the election. Another approach to determining whether the taller candidate won is to compare the heights with the logical operator `>`. Like the basic arithmetic operations, this operation is vectorized. The result will be a vector of logical values (`TRUE/FALSE`) having the same length as the two vectors being compared.

Table 1.2 Data for Example 1.2.

```
> data.frame(year, winner, opponent, difference)
   year winner opponent difference
1  2008    185      175         10
2  2004    182      193        -11
3  2000    182      185         -3
4  1996    189      187          2
5  1992    189      188          1
6  1988    188      173         15
7  1984    185      180          5
8  1980    185      177          8
9  1976    177      183         -6
10 1972    182      185         -3
11 1968    182      180          2
12 1964    193      180         13
13 1960    183      182          1
14 1956    179      178          1
15 1952    179      178          1
16 1948    175      173          2
```

```
> taller.won = winner > opponent
> taller.won
 [1]  TRUE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE
[10] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
```

On the second line, the prefix `[10]` indicates that the output continues with the tenth element of the vector.

The `table` function summarizes discrete data such as the result in the vector `taller.won`.

```
> table(taller.won)
taller.won
FALSE  TRUE
    4    12
```

We can use the result of `table` to display percentages if we divide the result by 16 and multiply that result by 100.

```
> table(taller.won) / 16 * 100
taller.won
FALSE  TRUE
   25    75
```

Thus, in the last 16 elections, the odds in favor of the taller candidate winning the election are 3 to 1.

Several types of graphs of this data may be interesting to help visualize any pattern. For example, we could display a barplot of differences using the `barplot` function. For the plot we use the `rev` function to reverse the order of the differences so that the election year is increasing from left to right. We also provide a descriptive label for both axes.

```
> barplot(rev(difference), xlab="Election years 1948 to 2008",
+   ylab="Height difference in cm")
```

The barplot of differences in heights is shown in Figure 1.1.

It would also be interesting to display a scatterplot of the data. A scatterplot of loser's heights vs winner's height for election years 1798 through 2004 appears in the Wikipedia article [54]. A simple version of the scatterplot (not shown here) can be obtained in R by

```
> plot(winner, opponent)
```

Chapter 4 "Presentation Graphics" illustrates many options for creating a custom graphic such as the scatterplot from the Wikipedia article.

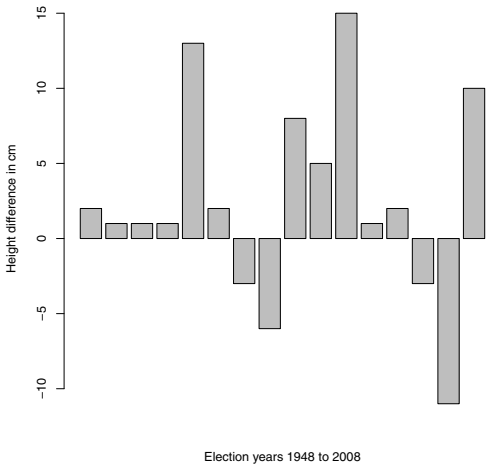


Fig. 1.1 Barplot of the difference in height of the election winner in the Electoral College over the height of the main opponent in the U.S. Presidential elections. Height differences in centimeters for election years 1948 through 2008 are shown from left to right. The electoral vote determines the outcome of the election. In 12 out of these 16 elections, the taller candidate won the electoral vote. In 2000, the taller candidate (Al Gore) did not win the electoral vote, but received more popular votes.

Example 1.3 (horsekicks). This data set appears in several books; see e.g. Larsen and Marx [30, p. 287]. In the late 19th century, Prussian officers collected data on deaths of soldiers in 10 calvary corps recording fatalities due to horsekicks over a 20 year period. The 200 values are summarized in Table 1.3.

To enter this data, we use the *combine* function `c`.

Table 1.3 Fatalities due to horsekick for Prussian calvary in Example 1.3

Number of deaths, k	Number of corps-years in which k fatalities occurred
0	109
1	65
2	22
3	3
4	1
	200

```
> k = c(0, 1, 2, 3, 4)
> x = c(109, 65, 22, 3, 1)
```

To display a bar plot of the frequencies, we use the `barplot` function. The function `barplot(x)` produces a barplot like Figure 1.2, but without the labels below the bars. The argument `names.arg` is optional; it assigns labels to display below the bars. Figure 1.2 is obtained by.

```
> barplot(x, names.arg=k)
```

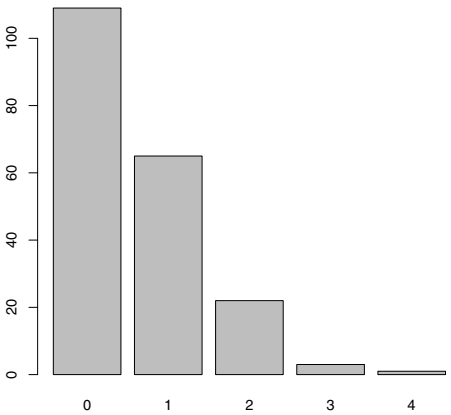


Fig. 1.2 Frequency distribution for Prussian horsekick data in Example 1.3.

The relative frequency distribution of the observed data in `x` is easily computed using vectorized arithmetic in R. For example, the sample proportion of 1's is $65/200 = 0.545$. The expression `x/sum(x)` divides every element of

the vector `x` by the sum of the vector (200). The result is a vector the same length as `x` containing the sample proportions of the death counts 0 to 4.

```
> p = x / sum(x)
> p
[1] 0.545 0.325 0.110 0.015 0.005
```

The center of this distribution can be estimated by its sample mean, which is

$$\frac{1}{200}\sum_{i=1}^{200} x_i = \frac{109(0)+65(1)+22(2)+3(3)+1(4)}{200}.$$
$$= 0.545(0)+0.325(1)+0.110(2)+0.015(3)+0.005(4).$$

The last line is simply the sum of `p*k`, because R computes this product element by element ("vectorized"). Now we can write the sample mean formula as the `sum` of the vector `p*k`. The value of the sample mean is then assigned to `r`.

```
> r = sum(p * k)
> r
[1] 0.61
```

Similarly, one can compute an estimate of the variance. Apply the computing formula for variance of a sample $y_1, \ldots, y_n$:

$$s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(y_i - \overline{y})^2.$$

Here the sample mean is the value `r` computed above and

$$s^2 = \frac{1}{n-1}\left\{109(0-r)^2+65(1-r)^2+22(2-r)^2+3(3-r)^2+1(4-r)^2\right\},$$

so the expression inside the braces can be coded as `x*(k-r)^2`. The sample variance `v` is:

```
> v = sum(x * (k - r)^2) / 199
> v
[1] 0.6109548
```

Among the counting distributions that might fit this data (binomial, geometric, negative binomial, Poisson, etc.) the Poisson is the one that has equal mean and variance. The sample mean 0.61 and sample variance 0.611 are almost equal, which suggests fitting a Poisson distribution to the data. The Poisson model has probability mass function

$$f(k) = \frac{\lambda^k e^{-\lambda}}{k!}, \quad k \geq 0, \tag{1.1}$$

where $\lambda = \sum_{k=0}^{\infty} kf(k)$ is the mean of the distribution. The sample mean 0.61 is our estimate of the population mean λ. Substituting the sample mean for λ in the density (1.1), the corresponding Poisson probabilities are

```
> f = r^k * exp(- r) / factorial(k)
> f
[1] 0.5433509 0.3314440 0.1010904 0.0205551 0.0031346
```

R has probability functions for many distributions, including Poisson. The R density functions begin with "d" and the Poisson density function is `dpois`. The probabilities above can also be computed as

```
> f = dpois(k, r)
> f
[1] 0.5433509 0.3314440 0.1010904 0.0205551 0.0031346
```

R$_{\mathbf{x}}$ 1.2 *R provides functions for the density, cumulative distribution function (CDF), percentiles, and for generating random variates for many commonly applied distributions. For the Poisson distribution these functions are* `dpois`*,* `ppois`*,* `qpois`*, and* `rpois`*, respectively. For the normal distribution these functions are* `dnorm`*,* `pnorm`*,* `qnorm`*, and* `rnorm`*.*

How well does the Poisson model fit the horsekick data? In a sample of size 200, the expected counts are $200f(k)$. Truncating the fraction using `floor` we have

```
>  floor(200*f)  #expected counts
[1] 108  66  20   4   0
>  x             #observed counts
[1] 109  65  22   3   1
```

for $k = 0, 1, 2, 3, 4$, respectively. The expected and observed counts are in close agreement, so the Poisson model appears to be a good one for this data.

One can alternately compare the Poisson probabilities (stored in vector `f`) with the sample proportions (stored in vector `p`). To summarize our comparison of the probabilities in a matrix we can use `rbind` or `cbind`. Both functions bind vectors together to form matrices; with `rbind` the vectors become rows, and with `cbind` the vectors become columns. Here we use `cbind` to construct a matrix with columns `k`, `p`, and `f`.

```
> cbind(k, p, f)
     k     p         f
[1,] 0 0.545 0.5433509
[2,] 1 0.325 0.3314440
[3,] 2 0.110 0.1010904
[4,] 3 0.015 0.0205551
[5,] 4 0.005 0.0031346
```

It appears that the observed proportions `p` are close to the Poisson(0.61) probabilities in `f`.

1.1.3 R Scripts

Example 1.3 contains several lines of code that would be tedious to retype if one wants to continue the data analysis. If the commands are placed in a file, called an R script, then the commands can be run using `source` or copy-paste. Using the `source` function causes R to accept input from the named source, such as a file.

Open a new R script for editing. In the R GUI users can open a new script window through the *File* menu. Type the following lines of "horsekicks.R" (below) in the script. It is a good idea to insert a few comments. Comments begin with a # symbol.

Using the `source` function, auto-printing of expressions does not happen. We added `print` statements to the script so that the values of objects will be printed.

horsekicks.R

```
# Prussian horsekick data
k = c(0, 1, 2, 3, 4)
x = c(109, 65, 22, 3, 1)
p = x / sum(x)        #relative frequencies
print(p)

r = sum(k * p)   #mean
v = sum(x * (k - r)^2) / 199  #variance
print(r)
print(v)
f = dpois(k, r)
print(cbind(k, p, f))
```

At this point it is convenient to create a working directory for the R scripts and data files that will be used in this book. To display the current working directory, type `getwd()`. For example, one may create a directory at the root, say `/Rx`. Then change the working directory through the File menu or by the function `setwd`, substituting the path to your working directory in the quotation marks below. On our system this has the following effect.

```
> getwd()
[1] "C:/R/R-2.13.0/bin/i386"
> setwd("c:/Rx")
> getwd()
[1] "c:/Rx"
```

Save the script as "horsekicks.R" in your working directory. Now the file can be sourced by the command

```
source("horsekicks.R")
```

and all of the commands in the file will be executed.

R$_x$ 1.3 *Unlike* Matlab *.m files, an R script can contain any number of functions and commands.* Matlab *users may be familiar with defining a function*

by writing an .m file, where each .m file is limited to exactly one function. Function syntax is covered in Section 1.2.

Rx 1.4 *Here are a few helpful shortcuts for running part of a script.*

- *Select lines and click the button 'Run line or selection' on the toolbar.*
- *Copy the lines, and then paste the lines at the command prompt.*
- *(For Windows users:) To execute one or more lines of the file in the R GUI editor, select the lines and type Ctrl-R.*
- *(For Macintosh users:) One can execute lines of a file by selecting the lines and typing Command-Return.*

Example 1.4 (Simulated horsekick data). For comparison with Example 1.3, in this example we use the random Poisson generator `rpois` to simulate 200 random observations from a Poisson($\lambda = 0.61$) distribution. We then compute the relative frequency distribution for this sample. Because these are randomly generated counts, each time the code below is executed we obtain a different sample and therefore the results of readers will vary slightly from what follows.

```
> y = rpois(200, lambda=.61)
> kicks = table(y)   #table of sample frequencies
> kicks
y
  0   1   2   3
105  67  26   2
> kicks / 200        #sample proportions
y
    0     1     2     3
0.525 0.335 0.130 0.010
```

Comparing this data with the theoretical Poisson frequencies:

```
> Theoretical = dpois(0:3, lambda=.61)
> Sample = kicks / 200
> cbind(Theoretical, Sample)
  Theoretical Sample
0  0.54335087  0.525
1  0.33144403  0.335
2  0.10109043  0.130
3  0.02055505  0.010
```

The computation of mean and variance is simpler here than in Example 1.3 because we have the raw, ungrouped data in the vector `y`.

```
> mean(y)
[1] 0.625
> var(y)
[1] 0.5571608
```

It is interesting that the observed Prussian horsekicks data seems to fit the Poisson model better than our simulated Poisson($\lambda = 0.61$) sample.

1.1.4 The R Help System

The R Graphical User Interface has a Help menu to find and display online documentation for R objects, methods, data sets, and functions. Through the Help menu one can find several manuals in PDF form, an html help page, and help search utilities. The help search utility functions are also available at the command line, using the functions `help` and `help.search`, and the corresponding shortcuts `?` and `??`. These functions are described below.

- `help("keyword")` displays help for "keyword".
- `help.search("keyword")` searches for all objects containing "keyword".

The quotes are usually optional in `help`, but would be required for special characters such as in `help("[")`. Quotes are required for `help.search`. When searching for help topics, keep in mind that R is case-sensitive: for example, `t` and `T` are different objects.

One or two question marks in front of a search term also search for help topics.

- `?keyword` (short for `help(keyword)`).
- `??keyword` (short for `help.search("keyword")`).

Try entering the following commands to see their effect.

```
?barplot        #searches for barplot topic
??plot          #anything containing "plot"

help(dpois)         #search for "dpois"
help.search("test") #anything containing "test"
```

The last command above displays a list including a large number of statistical tests implemented in the R.

One of the features of R online help is that most of the keywords documented include examples appearing at the end of the page. Users can try one or more of the examples by selecting the code and then copy-paste to the console. R also provides a function `example` that runs all of the examples if any exist for the keyword. To see the examples for the function `mean`, type `example(mean)`. The examples are then executed and displayed at the console with a special prompt symbol (`mean>`) that is specific to the keyword.

```
> example(mean)

mean> x <- c(0:10, 50)

mean> xm <- mean(x)

mean> c(xm, mean(x, trim = 0.10))
[1] 8.75 5.50
```

```
mean> mean(USArrests, trim = 0.2)
  Murder  Assault UrbanPop     Rape
    7.42   167.60    66.20    20.16
>
```

For many of the graphics functions, the documentation includes interesting examples. Try `example(curve)` for an overview of what the `curve` function can do. The system will prompt the user for input as it displays each graph.

A glossary of R functions is available online in "Appendix D: Function and Variable Index" of the manual "Introduction to R" [49], and the "R Reference Manual" [41] has a comprehensive index by function and concept. These manuals are included with the R distribution, and also available online on the R project home page[1] at the line "Manuals" under "Documentation".

1.2 Functions

The R language allows for modular programming using functions. R users interact with the software primarily through functions. We have seen several examples of functions above. In this section, we discuss how to create user-defined functions.

The syntax of a function is

```
f = function(x, ...) {
  }
```

or

```
f <- function(x, ...) {
  }
```

where `f` is the name of the function, `x` is the name of the first argument (there can be several arguments), and `...` indicates possible additional arguments. Functions can be defined with no arguments, also. The curly brackets enclose the body of the function. The return value of a function is the value of the last expression evaluated.

Example 1.5 (function definition). R has a function `var` that computes the unbiased estimate of variance, usually denoted by s^2. Occasionally, one requires the maximum likelihood estimator (MLE) of variance,

$$\hat{\sigma}^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \overline{x})^2 = \frac{n-1}{n}\,s^2.$$

A function to compute $\hat{\sigma}^2$ can be created as follows.

[1] www.r-project.org

```
var.n = function(x) {
  v = var(x)
  n = NROW(x)
  v * (n - 1) / n
  }
```

The NROW function computes the number of observations in x. The value v * (n-1)/n evaluated on the last line is returned. Note: it would also be correct (but unnecessary) to replace the last line of the function var.n with

```
return(v * (n - 1) / n)
```

Before this user-defined function can be used, one must input the code so that the function, in this case var.n, is an object in the R workspace. Normally, one places functions in a script file and uses the source function (or copy and paste to the command line) to submit them. Here is an example that computes s^2 and $\hat{\sigma}^2$ for the temperature data of Example 1.1.

```
> temps = c(51.9, 51.8, 51.9, 53)
> var(temps)
[1] 0.3233333
> var.n(temps)
[1] 0.2425
```

Example 1.6 (functions as arguments). Many of the available R functions require functions as arguments. An example is the integrate function, which implements numerical integration; one must supply the integrand as an argument. Suppose that we need to compute the beta function, which is defined as

$$B(a,b) = \int_0^1 x^{a-1}(1-x)^{b-1}\,dx,$$

for constants $a > 0$ and $b > 0$. First we write a function that returns the integrand evaluated at a given point x. The additional arguments a and b specify the exponents.

```
f = function(x, a=1, b=1)
    x^(a-1) * (1-x)^(b-1)
```

The curly brackets are not needed here because there is only one line in the function body. Also, we defined default values $a = 1$ and $b = 1$, so that if a or b are not specified, the default values will be used. The function can be used to evaluate the integrand along a sequence of x values.

```
> x = seq(0, 1, .2)  #sequence from 0 to 1 with steps of .2
> f(x, a=2, b=2)
[1] 0.00 0.16 0.24 0.24 0.16 0.00
```

This *vectorized* behavior is necessary for the function argument of the integrate function; the function that evaluates the integrand must accept a vector as its first argument and return a vector of the same length.

Now the numerical integration result for $a = b = 2$ can be obtained by

```
> integrate(f, lower=0, upper=1, a=2, b=2)
0.1666667 with absolute error < 1.9e-15
```

Actually, R provides a function `beta` to compute this integral. We can compare our numerical integration result to the value returned by the `beta` function:

```
> beta(2, 2)
[1] 0.1666667
```

See `?Special` for more details on the beta and other special functions of mathematics.

R$_x$ 1.5 *The* ***integrate*** *function is an example of a function syntax with extra arguments (...). The complete function syntax is*

```
integrate(f, lower, upper, ..., subdivisions=100,
          rel.tol = .Machine$double.eps^0.25, abs.tol = rel.tol,
          stop.on.error = TRUE, keep.xy = FALSE, aux = NULL)
```

The three dots are additional arguments to be passed to the integrand function `f`*. In our example, these extra arguments were* `a` *and* `b`*.*

Example 1.7 (graphing a function using ***curve****).* R provides the `curve` function to display the graph of a function. For example, suppose that we wish to graph the function

$$f(x) = x^{a-1}(1-x)^{b-1}$$

for $a = b = 2$, which is the integrand in Example 1.6. This is easily obtained as

```
> curve(x*(1-x), from=0, to=1, ylab="f(x)")
```

See Figure 1.3 for the result. The function argument in `curve` is always written as a function of `x`. The optional argument `ylab` specifies the label for the vertical axis.

1.3 Vectors and Matrices

A *vector* in R contains a finite sequence of values of a single type, such as a sequence of numbers or a sequence of characters. A *matrix* in R is a two dimensional array of values of a single type.

Common operations on vectors and matrices are illustrated with the following probability example. A more detailed introduction to vectors and matrices in R is provided in the Appendix.

Example 1.8 (Class mobility). The following model of class mobility is discussed in Ross [42, Example 4.19, p. 207]. Assume that the class of a child (lower, middle, or upper class) depends only on the class of his/her parents.

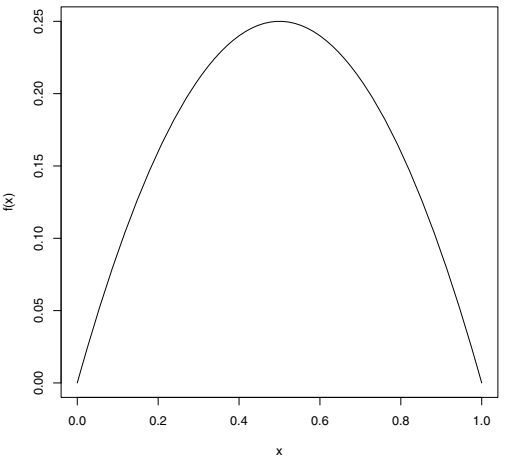


Fig. 1.3 Plot of the beta function for parameters $a = 2, b = 2$ in Example 1.7.

The class of the parents is indicated by the row label. The entries in the table below correspond to the chance that the child will transition to the class indicated by the column label.

	lower	middle	upper
lower	0.45	0.48	0.07
middle	0.05	0.70	0.25
upper	0.01	0.50	0.49

To create a matrix with these transition probabilities, we use the `matrix` function. First, the vector of probabilities `probs` is constructed to supply the entries of the matrix. Then the matrix is defined by its entries, number of rows, and number of columns.

```
> probs = c(.45, .05, .01, .48, .70, .50, .07, .25, .49)
> P = matrix(probs, nrow=3, ncol=3)
> P
     [,1] [,2] [,3]
[1,] 0.45 0.48 0.07
[2,] 0.05 0.70 0.25
[3,] 0.01 0.50 0.49
```

Notice that the values are entered by column; to enter the data by row, use the optional argument `byrow=TRUE` in the `matrix` function. Matrices can optionally have row names and column names. In this case, row names and column names are identical, so we can assign both using

```
> rownames(P) <- colnames(P) <- c("lower", "middle", "upper")
```

and the updated value of `P` is

```
> P
       lower middle upper
lower   0.45   0.48  0.07
middle  0.05   0.70  0.25
upper   0.01   0.50  0.49
```

In the matrix $P = (p_{ij})$, the probability p_{ij} in the i-th row and j-th column is the probability of a transition from class i to class j in one generation.

This type of matrix has rows that sum to 1 (because each row is a probability distribution on the three classes). This fact can be verified by the `rowSums` function.

```
> rowSums(P)
 lower middle  upper
     1      1      1
```

Another approach uses the `apply` function. It requires specifying the name of the matrix, `MARGIN` (row=1, column=2), and `FUN` (function) as its arguments.

```
> apply(P, MARGIN=1, FUN=sum)
lower middle  upper
     1      1      1
```

It can be shown that the transition probabilities for two generations are given by the product $P^2 = PP$, which can be computed by the matrix multiplication operator `%*%`.

```
> P2 = P %*% P
> P2
        lower middle  upper
lower  0.2272 0.5870 0.1858
middle 0.0600 0.6390 0.3010
upper  0.0344 0.5998 0.3658
```

Rx 1.6 *Here we did not use the syntax* `P^2` *because* `P^2` *squares every element of the matrix and the result is the matrix* (p_{ij}^2)*, not the matrix product.*

To extract elements from the matrix, the [row, column] syntax is used. If the row or column is omitted, this specifies all rows (columns). In two generations, the probability that descendants of lower class parents can transition to upper class is in row 1, column 3:

```
> P2[1, 3]
[1] 0.1858
```

and the probability distribution for lower class to (lower, middle, upper) is given by row 1:

```
> P2[1, ]
 lower middle  upper
0.2272 0.5870 0.1858
```

After several generations, each row of the transition matrix will be approximately equal, with probabilities $p = (l, m, u)$ corresponding to the percentages of lower, middle, and upper class occupations. After eight transitions, the probabilities are `P8`:

```
> P4 = P2 %*% P2
> P8 = P4 %*% P4

> P8
            lower    middle     upper
lower  0.06350395 0.6233444 0.3131516
middle 0.06239010 0.6234412 0.3141687
upper  0.06216410 0.6234574 0.3143785
```

It can be shown that the limiting probabilities are 0.07, 0.62, and 0.31. For the solution p, see Ross [42, p. 207].

R$_x$ 1.7 *To enter a matrix of constants, as in this example, it is usually easier to enter data by rows using* `byrow=TRUE` *in the* `matrix` *function. Compare the following with the example on page 17.*

```
> Q = matrix(c(  0.45,  0.48,  0.07,
+                0.05,  0.70,  0.25,
+                0.01,  0.50,  0.49), nrow=3, ncol=3, byrow=TRUE)
> Q
     [,1] [,2] [,3]
[1,] 0.45 0.48 0.07
[2,] 0.05 0.70 0.25
[3,] 0.01 0.50 0.49
```

The "by row" format makes it easier to see (visually) the data vector as a matrix in the code.

R$_x$ 1.8 *Matrix operations for numeric matrices in R:*

1. *Elementwise multiplication:* `*`
   If matrices $A = (a_{ij})$ and $B = (b_{ij})$ have the same dimension, then `A*B` *is evaluated as the matrix with entries $(a_{ij}b_{ij})$.*
2. *If $A = (a_{ij})$ is a matrix then* `A^r` *is evaluated as the matrix with entries (a_{ij}^r).*
3. *Matrix multiplication:* `%*%`
   If $A = (a_{ij})$ is an $n \times k$ matrix and $B = (b_{ij})$ is a $k \times m$ matrix, then `A %*% B` *is evaluated as the $n \times m$ matrix product AB.*
4. *Matrix inverse:*
   If A is a nonsingular matrix, the inverse of A is returned by `solve(A)`.

   For eigenvalues and matrix factorization, see `eigen`, `qr`, `chol`, *and* `svd`.

1.4 Data Frames

Data frames are special types of objects in R designed for data sets that are somewhat like matrices, but unlike matrices, the columns of a `data.frame` can be different types such as numeric or character. Several data sets are installed with R; a list of these data sets can be displayed by the command `data()`. Most data sets that are provided with R are in data frame format.

1.4.1 Introduction to data frames

The data frame format is similar to a spreadsheet, with the variables corresponding to columns and the observations corresponding to rows. Variables in a data frame may be numeric (numbers) or categorical (characters or factors).

To get an initial overview of a data frame, we are usually interested in knowing the names of variables, type of data, sample size, numbers of missing observations, etc.

Example 1.9 (USArrests). The *USArrests* data records rates of violent crimes in the US. The statistics are given as arrests per 100,000 residents for assault, murder, and rape in each of the 50 US states in 1973. The percentage of the population living in urban areas is also given. Some basic functions to get started with a data frame are illustrated with this data set.

Display all data

To simply display the data, type the name of the object, `USArrests`.

Display top of data

To display the first few lines of data:

```
> head(USArrests)
           Murder Assault UrbanPop Rape
Alabama      13.2     236       58 21.2
Alaska       10.0     263       48 44.5
Arizona       8.1     294       80 31.0
Arkansas      8.8     190       50 19.5
California    9.0     276       91 40.6
Colorado      7.9     204       78 38.7
```

The result shows that we have four variables named Murder, Assault, UrbanPop, and Rape, and that the observations (rows) are labeled by the name of the state. We also see that the states appear to be listed in alphabetical order. All of the variables appear to be quantitative, which we expected from the description above.

Sample size and dimension

How many observations are in this data set? (`NROW`, `nrow`, or `dim`)

```
> NROW(USArrests)
[1] 50
> dim(USArrests)  #dimension
[1] 50  4
```

The dimension (`dim`) of a data frame or a matrix returns a vector with the number of rows and number of columns. `NROW` returns the number of observations. We have 50 observations corresponding to the 50 states in the U.S.

Names of variables

Get (or set) names of variables in the data frame:

```
> names(USArrests)
[1] "Murder"   "Assault"  "UrbanPop" "Rape"
```

Structure of the data

Display information about the structure of the data frame (`str`):

```
> str(USArrests)
'data.frame':   50 obs. of  4 variables:
 $ Murder  : num  13.2 10 8.1 8.8 9 7.9 3.3 5.9 15.4 17.4 ...
 $ Assault : int  236 263 294 190 276 204 110 238 335 211 ...
 $ UrbanPop: int  58 48 80 50 91 78 77 72 80 60 ...
 $ Rape    : num  21.2 44.5 31 19.5 40.6 38.7 11.1 15.8 31.9 25.8 ...
```

The result of `str` gives the dimension as well as the name and type of each variable. We have two numeric type and two integer type variables. Although we can think of integer as a special case of numeric, they are stored differently in R.

R$_x$ 1.9 *For many data sets, like `USArrests`, all of the data are numbers and in this case the data can be converted to a matrix using `as.matrix`. But in order to store the data in a matrix, all variables must be of the same type so R will convert the integers to numeric. Compare the result in matrix form:*

```
> arrests = as.matrix(USArrests)
> str(arrests)
 num [1:50, 1:4] 13.2 10 8.1 8.8 9 7.9 3.3 5.9 15.4 17.4 ...
 - attr(*, "dimnames")=List of 2
  ..$ : chr [1:50] "Alabama" "Alaska" "Arizona" "Arkansas" ...
  ..$ : chr [1:4] "Murder" "Assault" "UrbanPop" "Rape"
```

This output shows that all of the data was converted to numeric, listed on the first line as `num`. The attributes (`attr`) are the row and column names (`dimnames`). The conversion preserved the row labels and converted the variable names to column labels. We used `names` to get the names of the variables in the data frame, but we would use `rownames`, `colnames` or `dimnames` (to get both) to get the row and/or column names. These last three functions can also be used on data frames.

Missing values

The `is.na` function returns `TRUE` for a missing value and otherwise `FALSE`. The expression `is.na(USArrests)` will return a data frame the same size as `USArrests` where every entry is `TRUE` or `FALSE`. To quickly check if any of the results are `TRUE` we use the `any` function.

```
> any(is.na(USArrests))
[1] FALSE
```

We see that USArrests does not contain missing values. A data set with missing values is discussed in Example 2.3 on page 53.

1.4.2 Working with a data frame

In this section we illustrate some operations on data frames, and some basic statistics and plots.

Example 1.10 (USArrests, cont.).

Compute summary statistics

Obtain appropriate summary statistics for each variable using `summary`. For numeric data, the `summary` function computes a five-number summary and sample mean.

```
> summary(USArrests)
     Murder          Assault         UrbanPop          Rape
 Min.   : 0.800   Min.   : 45.0   Min.   :32.00   Min.   : 7.30
 1st Qu.: 4.075   1st Qu.:109.0   1st Qu.:54.50   1st Qu.:15.07
 Median : 7.250   Median :159.0   Median :66.00   Median :20.10
 Mean   : 7.788   Mean   :170.8   Mean   :65.54   Mean   :21.23
 3rd Qu.:11.250   3rd Qu.:249.0   3rd Qu.:77.75   3rd Qu.:26.18
 Max.   :17.400   Max.   :337.0   Max.   :91.00   Max.   :46.00
```

If there were any missing values, the number of missing values would be included in the summaries; see e.g. Example 2.3 on page 53. If any of our variables were categorical, `summary` would tabulate the values for those variables.

From the summary it appears that the mean and median are approximately equal for all variables except `Assault`. The mean for `Assault` is larger than the median, indicating that the assault data is positively skewed.

Extract data from a data frame

The simplest way to extract data from a data frame uses the matrix-style [row, column] indexing.

```
> USArrests["California", "Murder"]
[1] 9
> USArrests["California", ]
           Murder Assault UrbanPop Rape
California      9     276       91 40.6
```

Extract a variable using $

Variables can be extracted using the `$` operator followed by the name of the variable.

```
> USArrests$Assault
 [1] 236 263 294 190 276 204 110 238 335 211  46 120 249 113
[15]  56 115 109 249  83 300 149 255  72 259 178 109 102 252
[29]  57 159 285 254 337  45 120 151 159 106 174 279  86 188
[43] 201 120  48 156 145  81  53 161
```

Histograms

In the summary of the *USArrests* data frame, we observed that the distribution of assaults may be positively skewed because the sample mean is larger than the sample median. A histogram of the data helps to visualize the shape of the distribution. We show two versions of the histogram of `Assault`. The result of

```
> hist(USArrests$Assault)
```

is shown in Figure 1.4(a). The skewness is easier to observe in the second histogram (Figure 1.4(b)), obtained using

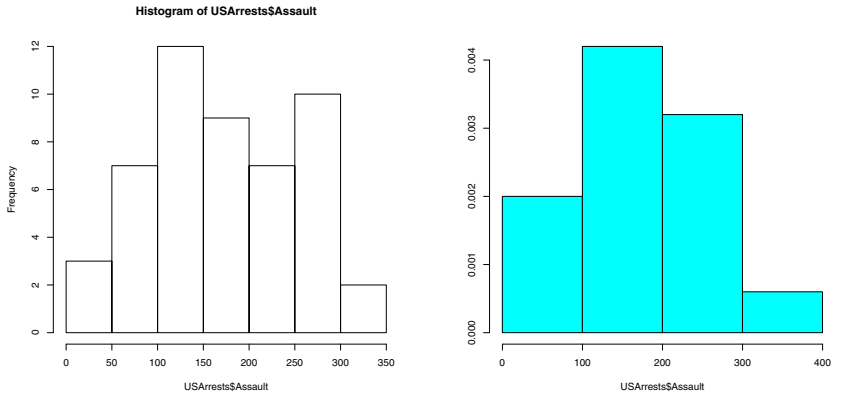
```
> library(MASS)    #need for truehist function
> truehist(USArrests$Assault)
```

There are obvious differences in the histograms; one is that the vertical scales differ. Another is that there are a different number of bins. Figure 1.4(a) is a frequency histogram, and Figure 1.4(b) is a probability histogram.

R$_x$ 1.10 *Try the following command and compare the result with* ***truehist*** *in Figure 1.4(b):*

```
hist(USArrests$Assault, prob=TRUE, breaks="scott")
```

The two histogram functions ***hist*** *and* ***truehist*** *have different default methods for determining the bin width, and the* ***truehist*** *function by default produces a probability histogram. The optional arguments of* ***hist*** *above match the defaults of* ***truehist****.*



(a) (b)

Fig. 1.4 Frequency histogram (a) using `hist` and probability histogram (b) using `truehist` for the assault data in `USArrests`.

Attaching a data frame

If we `attach` the data frame, the variables can be referenced directly by name, without the dollar sign operator. For example, it is easy to compute the percent of crimes that are murders using vectorized operations.

```
> attach(USArrests)
> murder.pct = 100 * Murder / (Murder + Assault + Rape)
> head(murder.pct)
[1] 4.881657 3.149606 2.431702 4.031150 2.764128 3.152434
```

If a data frame is attached, it can be detached when it is no longer needed using the `detach` function.

An alternative to attaching a data frame is (sometimes) to use the `with` function. It is useful for displaying plots or summary statistics. However, variables created using `with` are local to it.

```
> with(USArrests, expr={
+    murder.pct = 100 * Murder / (Murder + Assault + Rape)
+    })
> murder.pct
Error: object 'murder.pct' not found
```

The documentation for `with` states that "assignments within expr take place in the constructed environment and not in the user's workspace." So all computations involving the variable `murder.pct` created in the `expr` block of the `with` function would have to be completed within the scope of the `expr` block. This can sometimes lead to unexpected errors in an analysis that can easily go undetected.

Rx 1.11 *Is it good programming practice to* `attach` *a data frame? A disadvantage is that there can be conflicts with names of variables already in the workspace. Many functions in R have a* `data` *argument, which allows the variables to be referenced by name within the list of arguments, without attaching the data frame. Unfortunately, some R functions (such as* `plot`*) do not have a* `data` *argument. The use of* `with` *could lead to unexpected programming errors, especially for novices. Overall, we find that attaching a data frame is sometimes helpful to make code more readable.*

Scatterplots and correlations

In the `USArrests` data all of the variables are numbers, so it is interesting to display scatterplots of different pairs of data to look for possible relations between the variables. We can display a single scatterplot using the `plot` function. Recall that above we have attached the data frame using `attach` so the variables can be referenced directly by name. To obtain a plot of murders vs percent urban population we use

```
> plot(UrbanPop, Murder)
```

and the result is shown in Figure 1.5. The plot does not reveal a very strong relation between murders and percent population. The `pairs` function can be used to display an array of scatterplots for each pair of variables.

```
> pairs(USArrests)
```

The pairs plot shown in Figure 1.6 conveys much information about the data. There appears to be a positive association between murder and assault rates, but weak or no association between murder and percent urban population. There is a positive association between rape and percent urban population.

The correlation statistic measures the degree of linear association between two variables. One can obtain correlation for a pair of variables or a table of correlation statistics for a group of variables using the `cor` function. By default it computes the Pearson correlation coefficient.

```
> cor(UrbanPop, Murder)
[1] 0.06957262
> cor(USArrests)
             Murder   Assault   UrbanPop      Rape
Murder   1.00000000 0.8018733 0.06957262 0.5635788
Assault  0.80187331 1.0000000 0.25887170 0.6652412
UrbanPop 0.06957262 0.2588717 1.00000000 0.4113412
Rape     0.56357883 0.6652412 0.41134124 1.0000000
```

All of the correlations are positive in sign. The small correlation of $r \doteq 0.07$ between `Murder` and `UrbanPop` is consistent with our interpretation of the scatterplot in Figure 1.5. There is a strong positive correlation ($r = 0.80$) between `Murder` and `Assault`, also consistent with the plot in Figure 1.6.

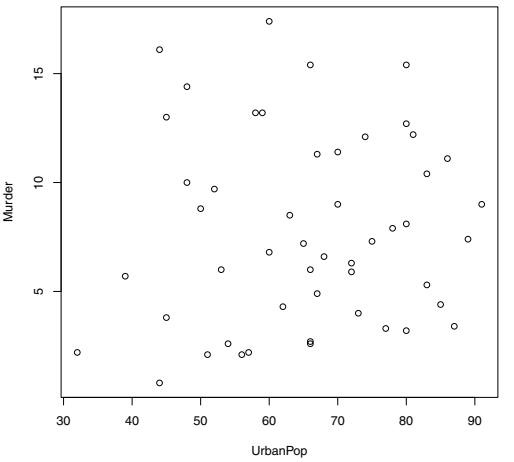


Fig. 1.5 Scatterplot of murder rate vs percent urban population in `USArrests`.

1.5 Importing Data

A preliminary task for data analysis is to import data into R. Data sets may be found on the web, in plain text (space delimited) files, spreadsheets, and many other formats. R contains utility functions to import data in a variety of formats. When a data set is imported into R, typically we store it in an R `data.frame` object. See the examples of Section 1.4.

In this section we cover various methods of importing data, including:

- Entering data manually.
- Importing data from a plain text (ASCII) local file.
- Importing data given in tabular form on a web page.

1.5.1 Entering data manually

In the first few examples of this chapter we have seen how to enter data vectors using the `c` (combine) function. The `scan` function is sometimes useful for entering small data sets interactively. See Example 3.1 "Flipping a Coin" in Chapter 3 for an example.

The concept for this book was originally inspired by students' questions in a statistics course. After learning some R basics, students were eager to try their textbook problems in R, but typically needed help to enter the data.

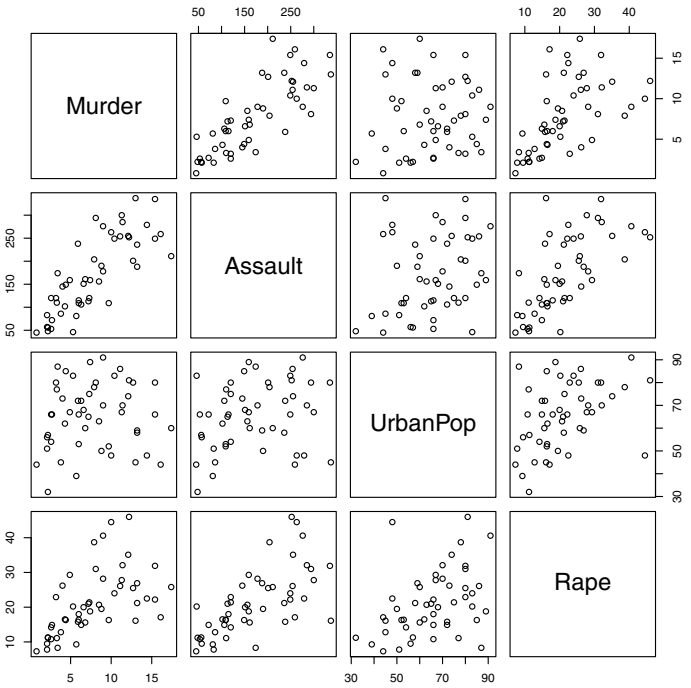


Fig. 1.6 An array of scatterplots produced by `pairs` for the `USArrests` data.

The next example shows how to enter a very small data set that one might find among the exercises in a statistics textbook.

Example 1.11 (Data from a textbook). A table of gas mileages on four new models of Japanese luxury cars is given in Larsen & Marx [30, Question 12.1.1], which is shown in Table 1.4. The reader was asked to test if the four models give the same gas mileage, on average.

Table 1.4 Gas mileage on four models of Japanese luxury cars, from a problem in Larsen & Marx [30, 12.1.1].

Model			
A	B	C	D
22	28	29	23
26	24	32	24
29	28		

To solve the stated problem, we need to enter the data as two variables (gas mileage and model). One method of doing so is shown below. The `rep` (replicate) function is used to create the sequences of letters.

```
> y1 = c(22, 26)
> y2 = c(28, 24, 29)
> y3 = c(29, 32, 28)
> y4 = c(23, 24)
> y = c(y1, y2, y3, y4)
> Model = c(rep("A", 2), rep("B", 3), rep("C", 3), rep("D", 2))
```

We see that `y` and `Model` have been entered correctly:

```
> y
 [1] 22 26 28 24 29 29 32 28 23 24
> Model
 [1] "A" "A" "B" "B" "B" "C" "C" "C" "D" "D"
```

The data frame is created by

```
> mileages = data.frame(y, Model)
```

The character vector `Model` is converted to a factor by default when the data frame is created. We can check that the structure of our data frame is correct using the structure (`str`) function.

```
> str(mileages)
'data.frame':   10 obs. of  2 variables:
 $ y    : num  22 26 28 24 29 29 32 28 23 24
 $ Model: Factor w/ 4 levels "A","B","C","D": 1 1 2 2 2 3 3 3 4 4
```

and

```
> mileages
    y Model
1  22     A
2  26     A
3  28     B
4  24     B
5  29     B
6  29     C
7  32     C
8  28     C
9  23     D
10 24     D
```

Our data set is ready for analysis, which is left as an exercise for a later chapter (Exercise 8.1).

1.5.2 Importing data from a text file

It is often the case that data for analysis is contained in an external file (external to R) in plain text (ASCII) format. The data is typically delimited

or separated by a special character such as a space, tab, or comma. The `read.table` function provides optional arguments such as `sep` for the separator character and `header` to indicate whether or not the first row contains variable names.

Example 1.12 (Massachusetts Lunatics). Importing data from a plain text file can be illustrated with an example of a data set available on the website "Data and Story Library" (DASL). The *Massachusetts lunatics* data is available at http://lib.stat.cmu.edu/DASL/Datafiles/lunaticsdat.html.

These data are from an 1854 survey conducted by the Massachusetts Commission on Lunacy. Fourteen counties were surveyed. The data can be copied from the web page, and simply pasted into a plain text file. Although the result is not nicely formatted, it is space delimited (columns separated by spaces). The data is saved in "lunatics.txt" in our current working directory. This data set should be imported into a data frame that has 14 rows and six columns, corresponding to the following variables:

1. COUNTY = Name of county
2. NBR = Number of lunatics, by county
3. DIST = Distance to nearest mental health center
4. POP = County population , 1950 (thousands)
5. PDEN = County population density per square mile
6. PHOME = Percent of lunatics cared for at home

Use the `read.table` function to read the file into a data frame. The argument `header=TRUE` specifies that the first line contains variable names rather than data. Type `?read.table` for a description of other possible arguments.

```
> lunatics = read.table("lunatics.txt", header=TRUE)
```

The `str` (structure) function provides a quick check that 14 observations of six variables were successfully imported.

```
> str(lunatics)
'data.frame':   14 obs. of  6 variables:
 $ COUNTY: Factor w/ 14 levels "BARNSTABLE","BERKSHIRE",..
 $ NBR   : int  119 84 94 105 351 357 377 458 241 158 ...
 $ DIST  : int  97 62 54 52 20 14 10 4 14 14 ...
 $ POP   : num  26.7 22.3 23.3 18.9 82.8 ...
 $ PDEN  : int  56 45 72 94 98 231 3252 3042 235 151 ...
 $ PHOME : int  77 81 75 69 64 47 47 6 49 60 ...
```

Now since `lunatics` is a relatively small data set, we can simply print it to view the result of `read.table` that we used to import the data. Typing the name of the data set causes it to be printed (displayed) at the console.

```
> lunatics
       COUNTY NBR DIST     POP PDEN PHOME
1   BERKSHIRE 119   97  26.656   56    77
2    FRANKLIN  84   62  22.260   45    81
3   HAMPSHIRE  94   54  23.312   72    75
4     HAMPDEN 105   52  18.900   94    69
```

```
5   WORCESTER 351   20  82.836   98    64
6   MIDDLESEX 357   14  66.759  231    47
7       ESSEX 377   10  95.004 3252    47
8     SUFFOLK 458    4 123.202 3042     6
9     NORFOLK 241   14  62.901  235    49
10    BRISTOL 158   14  29.704  151    60
11   PLYMOUTH 139   16  32.526   91    68
12 BARNSTABLE  78   44  16.692   93    76
13  NANTUCKET  12   77   1.740  179    25
14      DUKES  19   52   7.524   46    79
```

The *Massachusetts lunatics* data set is discussed in Example 7.8 of our regression chapter.

In the example above, the data in the file "lunatics.txt" is delimited by space characters. Often data is found in a spreadsheet format, delimited by tab characters or commas.

For tab-delimited files, simply change the `sep` argument to the tab character `\t`. An example appears in Chapter 5 on exploratory data analysis, where the tab-delimited data file "college.txt" is imported using the command

```
dat = read.table("college.txt", header=TRUE, sep="\t")
```

R$_x$ 1.12 *The simplest way to import spreadsheet data is to save it in .csv format (comma separated values) or a tab-delimited format. The worksheet should contain only data and possibly the header with names. For .csv files, use the* `read.table` *function with* `sep=","` *as shown below*

```
> lunatics = read.table("lunatics.csv", header=TRUE, sep=",")
```

or use `read.csv` *for this type of file:*

```
> lunatics = read.csv("lunatics.csv")
```

1.5.3 Data available on the internet

Many of the interesting data sets that one may wish to analyze are available on a web page. R provides an easy way to access data from a file on the internet using the URL of the web page. The function `read.table` can be used to input data directly from the internet. This is illustrated in the following example.

Example 1.13 (Digits of π). The data file "PiDigits.dat" contains the first 5000 digits of the mathematical constant $\pi = 3.1415926535897932384\ldots$. The data is one of the *Statistical Reference Datasets* provided by the National Institute of Standards and Technology (NIST).[2] Documentation is inserted at the top of the file, and the digits start on line 61. We use the `read.table` function

[2] http://www.itl.nist.gov/div898/strd/univ/pidigits.html

with the complete URL[3] (web address) and `skip=60` to read the data starting at line 61. We display the first six digits to check the result:

```
pidigits = read.table(
  "http://www.itl.nist.gov/div898/strd/univ/data/PiDigits.dat",
  skip=60)
head(pidigits)

  V1
1  3
2  1
3  4
4  1
5  5
6  9
```

Here although we have only one variable, our data `pidigits` is a data frame. The data frame was automatically created because we used `read.table` to import the data and a default label of `V1` assigned to the single variable.

Are the digits of π uniformly distributed? The digits can be summarized in a table by the `table` function and summarized graphically in a plot.

```
> table(pidigits)

pidigits
  0   1   2   3   4   5   6   7   8   9
466 531 496 461 508 525 513 488 491 521
```

For easier interpretation, it is more convenient to summarize proportions. We can convert the table to proportions in one step by dividing by 5000; this is another example of vectorized operations.

```
> prop = table(pidigits) / 5000   #proportions
> prop

pidigits
     0      1      2      3      4      5      6      7      8      9
0.0932 0.1062 0.0992 0.0922 0.1016 0.1050 0.1026 0.0976 0.0982 0.1042
```

Recall that the variance of a sample proportion is $p(1-p)/n$. If the true proportion is 0.1 for every digit, then the standard error (se) is

```
> sqrt(.1 * .9 / 5000)
[1] 0.004242641
```

However, if the true proportions are unknown, the sample estimates of proportions are used. In this case we obtain slightly different results for se. In the calculation of `se` the constants 0.1 and 0.9 are replaced by vectors of length 10, so the result is a vector of length 10 rather than a scalar. We can display the sample proportion plus or minus two standard errors using vectorized arithmetic. The `rbind` function is handy to collect the results together into a

[3] The URL should be enclosed in quotes and on a single line; otherwise an error message "cannot open the connection" occurs.

matrix for display. We round the result and include the estimate of standard error in the display.

```
> se.hat = sqrt(prop * (1-prop) / 5000)
> round(rbind(prop, se.hat, prop-2*se.hat, prop+2*se.hat), 4)
            0      1      2      3      4      5      6
prop   0.0932 0.1062 0.0992 0.0922 0.1016 0.1050 0.1026
se.hat 0.0041 0.0044 0.0042 0.0041 0.0043 0.0043 0.0043
       0.0850 0.0975 0.0907 0.0840 0.0931 0.0963 0.0940
       0.1014 0.1149 0.1077 0.1004 0.1101 0.1137 0.1112
            7      8      9
prop   0.0976 0.0982 0.1042
se.hat 0.0042 0.0042 0.0043
       0.0892 0.0898 0.0956
       0.1060 0.1066 0.1128
```

Here we see that none of the sample proportions falls outside of the interval $0.1 \pm 2\,\widehat{se}$.

A barplot helps to visualize the tabulated data. A horizontal reference line is added through 0.1 using `abline`. The plot is shown in Figure 1.7.

```
barplot(prop, xlab="digit", ylab="proportion")
abline(h = .1)
```

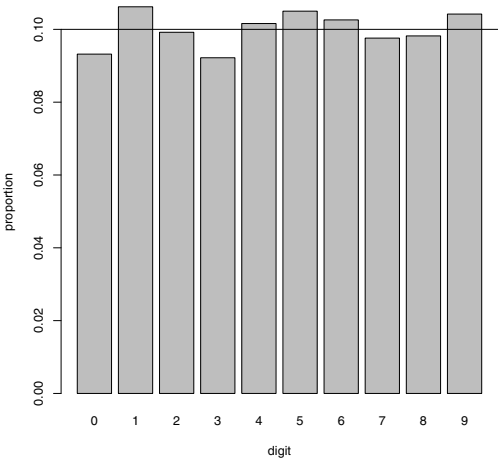


Fig. 1.7 Barplot of proportion of digits in the mathematical constant π.

1.6 Packages

R functions are grouped into packages, such as the `base` package, `datasets`, `graphics`, or `stats`. A number of recommended packages are included in the R distribution; some examples are `boot` (bootstrap), MASS [50], and `lattice` (graphics). In addition, thousands of contributed packages are available to install. Type the command

```
library()
```

to display a list of installed packages. For example, on our system currently this command produces a list starting with

```
Packages in library 'C:/R/R-2.13.0/library':

base                    The R Base Package
boot                    Bootstrap R (S-Plus) Functions (Canty)
bootstrap               Functions for the Book "An Introduction to the
                        Bootstrap"
    ...
```

Each of these packages is currently installed on our system. The `base` and `boot` packages were automatically installed. We installed the `bootstrap` package [31] through the Packages menu. Although there is excellent support for bootstrap in the `boot` package, we also want to have access to the data sets for the book *An Introduction to the Bootstrap* by Efron and Tibshirani [14], which are available in the `bootstrap` package.

Example 1.14 (Using the `bootstrap` package). Suppose that we are interested in the examples related to the "law" data in the `bootstrap` package. Typing `law` at the prompt produces the following error because no object named "law" is found on the R search path.

```
> law
Error: object 'law' not found
```

We get a similar warning with

```
> data(law)
Warning message:
In data(law) : data set 'law' not found
```

In order to use `law` we first install the `bootstrap` package. This can be done in the R GUI using the Packages menu, or by typing the command

```
install.packages("bootstrap")
```

The system will prompt the user to select a server:

```
--- Please select a CRAN mirror for use in this session ---
```

and after the server is selected, the package will be installed. The only time that the package needs to be re-installed is when a new version of R is installed. A list of data files in the `bootstrap` package is displayed by

```
> data(package="bootstrap")
```

However, to use the objects in the package, one first needs to load it using the `library` function,

```
> library(bootstrap)
```

and then the objects in the package will be available. Loading the package with the `library` function typically needs to be done once in each new R session (each time the program is opened). Another useful feature of the `library` function is to get help for a package: either of the following display a summary of the package.

```
library(help=bootstrap)
help(package=bootstrap)
```

Finally, we want to use the `law` data, which is now loaded and accessible in the R workspace.

```
> library(bootstrap)
> law
   LSAT GPA
1   576 339
2   635 330
3   558 281
4   578 303
5   666 344
6   580 307
7   555 300
8   661 343
9   651 336
10  605 313
11  653 312
12  575 274
13  545 276
14  572 288
15  594 296
```

For example, we may want to compute the sample means or the correlation between LSAT scores and GPA.

```
> mean(law)
    LSAT      GPA
600.2667 309.4667
> cor(law$LSAT, law$GPA)
[1] 0.7763745
```

After installing R, and periodically thereafter, one should update packages using `update.packages` or the Packages menu.

Searching for data or methods in packages

With thousands of contributed packages available for R, it is likely that whatever method one would like to apply has been implemented in an R package.

Many well known data sets can also be found in R packages. To search any installed packages, use `help.search` or `??`. Keep in mind that `help.search` and `??` do not search outside of the current R installation.

For example, suppose that we are trying to find a data set on heights of fathers and sons. We can try `??height` but none of the hits seem relevant. Next we go to the R homepage at www.r-project.org and click on *Search*. Using the R site search we get several relevant hits. The data set `galton` in package `UsingR` [51] is relevant ("Galton's height data for parents and children") and we also find a link to the data set `father.son` in `UsingR`, which is exactly what we were looking for. We can install the package `UsingR` and the `father.son` data will be available after loading the package with `library(UsingR)`.

One way to find an implementation of a particular method is to search on the R home page at www.r-project.org. A good general resource is to consult one or more of the *Task Views* (see page 38).

1.7 The R Workspace

When the user ends an R session by closing the R GUI or typing the quit command `q()`, a dialog appears asking "Save workspace image?". Usually one would not need to save the workspace image. Data is typically saved in files and reusable R commands should be saved in scripts.

One can list the objects in the current workspace by the command `ls()` or `objects()`. Starting with a new R session and no previously saved workspace, `ls()` returns an empty list.

```
> ls()
character(0)
```

After running the script "horsekicks.R" on page 11, a few objects have been added to the workspace. These objects will persist until the R session is ended or until removed or redefined by the user.

```
> source("/Rx/horsekicks.R")
[1] 0.545 0.325 0.110 0.015 0.005
[1] 0.61
[1] 0.6109548
     k     p           f
[1,] 0 0.545 0.543350869
[2,] 1 0.325 0.331444030
[3,] 2 0.110 0.101090429
[4,] 3 0.015 0.020555054
[5,] 4 0.005 0.003134646
```

The `ls` function displays the names of objects that now exist in the R workspace.

```
> ls()
[1] "f" "k" "p" "r" "v" "x"
```

To remove an object from the workspace use the `rm` or `remove` function.

```
> rm("v")
> ls()
[1] "f" "k" "p" "r" "x"

> remove(list=c("f", "r"))
> ls()
[1] "k" "p" "x"
```

If the workspace is saved upon exiting at this point, `k`, `p`, and `x` will be saved. However, saving a workspace combined with human error can lead to unnoticed serious programming errors. A typical example is when one forgets to define an object such as `x` but because it was already in the workspace (unintended and incorrect values for `x`) the R code may run without any reported errors. The unsuspecting user may never know that the analysis was completely wrong.

Finally to restore the R workspace to the "fresh" condition with no user-defined objects in the workspace, we can use the following code. This removes **all** objects listed by `ls()` without warning.

```
rm(list = ls())
```

This is best done at the beginning or the end of a session. Wait until the end of this chapter to try it.

1.8 Options and Resources

The `options` *function*

For more readable tables of data, we may want to round the displayed data. This can be done by explicitly rounding what is to be displayed:

```
> pi
[1] 3.141593
> round(pi, 5)
[1] 3.14159
```

Alternately, there is an option that governs the number of digits to display that can be set. The default is `digits=7`.

```
> options(digits=4)
> pi
[1] 3.142
```

To see the current values of options, type `options()`. Another option that helps to control the display is `width`; it controls how many characters are printed on each line. Illustrated below are two ways to get the current value of the width option, and changing the width option.

```
> options()$width      #current option for width
[1] 70
> options(width=60)    #change width to 60 characters
> getOption("width")   #current option for width
[1] 60
```

Graphical parameters: `par`

Another set of options, for graphical parameters, is controlled by the `par` function. A useful one to know is how to change the "prompt user for next graph" behavior. It can be turned on/off by changing the graphics parameter `ask`; for example, to turn this prompt off:

```
par(ask = FALSE)
```

Another option that we use in this book is `mfrow` or `mfcol` to control how many figures are displayed in the current graphics window. For example, `par(mfrow=c(2, 2))` will present figures in a 2 by 2 array, by row. The graphical parameters that can be set using `par` are described in the help topic `?par`.

R$_x$ 1.13 *There are so many possible parameters to graphics functions, that usually only a subset of them are listed in the documentation. For example, the* `plot` *help page starts with*

```
Generic function for plotting of R objects.  For more
details about the graphical parameter arguments, see 'par'.
```

The `par` *help page contains further documentation for* `plot` *and other graphics functions.*

Graph history

In Windows, when a graphics window is active (on top), one can select *Recording* from the *History* menu. This has the effect of storing any graphs that are subsequently displayed, so that the user can use the page up/page down keys to page through the graphs.

If you construct multiple plots on a Macintosh, then with the graph selected, you can go Back or Forward from the Quartz menu to see previous graphs.

Other resources

In addition to the manuals, frequently asked questions (FAQ), and online help included with R, there are many contributed files and web pages with excellent tutorials, examples, and explanations. A list is available on the R web site.[4]

Task Views

"Task Views" for different types of statistical analyses are available on CRAN. Go to the R project home page at www.r-project.org and click on CRAN, then choose a mirror site near you. The CRAN page has a link to Task Views on several subjects. A Task View lists functions and packages that are related to the named task, such as "Bayesian", "Multivariate" or "Time Series". A direct link is http://cran.at.r-project.org/web/views/.

External resources

In addition to materials found on the R project website, there are many useful materials to be found on the web. There is an interesting collection of information and examples in R Wiki,[5] including the list of examples in the R Graph Gallery. There is also a list of other R Wiki's. A nicely organized external resource is "Quick R: for SAS/SPSS/Stata Users" at http://www.statmethods.net/index.html.

The R Graph Gallery and R Graphical Manual

For more experienced R users, a great resource for graphics is the *R Graph Gallery*.[6] We display the Gallery's home page and click on 'Thumbnails' to view small images of the graphs. Each graph includes the corresponding R code to produce the graph. Alternately one can select graphs by keyword or simply browse. The *R Graphical Manual*[7] illustrates thousands of graphs organized by image, task view, data set, or package.

[4] Contributed documentation, http://cran.r-project.org/other-docs.html
[5] R Wiki, http://rwiki.sciviews.org/doku.php
[6] R Graph Gallery, http://addictedtor.free.fr/graphiques/
[7] R Graphical Manual, http://rgm2.lab.nig.ac.jp/RGM2/images.php?show=all

1.9 Reports and Reproducible Research

Most data analysis will be summarized in some type of report or article. The process of "copy and paste" for commands and output can lead to errors and omissions. Reproducible research refers to methods of reporting that combine the data analysis, output, graphics, and written report together in such a way that the entire analysis and report can be reproduced by others. Various formats for reports may include word processing documents, LaTeX, or HTML.

The `Sweave` function in R facilitates generating this type of report. There is a LaTeX package (*Sweave*) that generates a .tex file from the Sweave output. Various other packages such as *R2wd* (R to Word), *R2PPT* (R to PowerPoint), *odfWeave* (open document format), *R2HTML* (HTML), can be installed. Commercial packages are also available (e.g. *RTFGen* and *Inference for R*). For more details see the Task View "Reproducible Research" on CRAN.[8]

Exercises

1.1 (Normal percentiles). The `qnorm` function returns the percentiles (quantiles) of a normal distribution. Use the `qnorm` function to find the 95^{th} percentile of the standard normal distribution. Then, use the `qnorm` function to find the quartiles of the standard normal distribution (the quartiles are the 25^{th}, 50^{th}, and 75^{th} percentiles). Hint: Use `c(.25, .5, .75)` as the first argument to `qnorm`.

1.2 (Chi-square density curve). Use the `curve` function to display the graph of the $\chi^2(1)$ density. The chi-square density function is `dchisq`.

1.3 (Gamma densities). Use the `curve` function to display the graph of the gamma density with shape parameter 1 and rate parameter 1. Then use the `curve` function with `add=TRUE` to display the graphs of the gamma density with shape parameter k and rate 1 for 2,3, all in the same graphics window. The gamma density function is `dgamma`. Consult the help file `?dgamma` to see how to specify the parameters.

1.4 (Binomial probabilities). Let X be the number of "ones" obtained in 12 rolls of a fair die. Then X has a Binomial$(n = 12, p = 1/3)$ distribution. Compute a table of binomial probabilities for $x = 0, 1, \ldots, 12$ by two methods:

a. Use the probability density formula

$$P(X = k) = \binom{n}{k} p^k (1-p)^{n-k}$$

[8] http://cran.at.r-project.org/web/views/ReproducibleResearch.html.

and vectorized arithmetic in R. Use `0:12` for the sequence of x values and the `choose` function to compute the binomial coefficients $\binom{n}{k}$.

b. Use the `dbinom` function provided in R and compare your results using both methods.

1.5 (Binomial CDF). Let X be the number of "ones" obtained in 12 rolls of a fair die. Then X has a Binomial$(n = 12, p = 1/3)$ distribution. Compute a table of cumulative binomial probabilities (the CDF) for $x = 0, 1, \ldots, 12$ by two methods: (1) using `cumsum` and the result of Exercise 1.4, and (2) using the `pbinom` function. What is $P(X > 7)$?

1.6 (Presidents' heights). Refer to Example 1.2 where the heights of the United States Presidents are compared with their main opponent in the presidential election. Create a scatterplot of the loser's height vs the winner's height using the `plot` function. Compare the plot to the more detailed plot shown in the Wikipedia article "Heights of Presidents of the United States and presidential candidates" [54].

1.7 (Simulated "horsekicks" data). The `rpois` function generates random observations from a Poisson distribution. In Example 1.3, we compared the deaths due to horsekicks to a Poisson distribution with mean $\lambda = 0.61$, and in Example 1.4 we simulated random Poisson$(\lambda = 0.61)$ data. Use the `rpois` function to simulate very large ($n = 1000$ and $n = 10000$) Poisson$(\lambda = 0.61)$ random samples. Find the frequency distribution, mean and variance for the sample. Compare the theoretical Poisson density with the sample proportions (see Example 1.4).

1.8 (*horsekicks*, continued). Refer to Example 1.3. Using the `ppois` function, compute the cumulative distribution function (CDF) for the Poisson distribution with mean $\lambda = 0.61$, for the values 0 to 4. Compare these probabilities with the empirical CDF. The empirical CDF is the cumulative sum of the sample proportions `p`, which is easily computed using the `cumsum` function. Combine the values of `0:4`, the CDF, and the empirical CDF in a matrix to display these results in a single table.

1.9 (Custom standard deviation function). Write a function `sd.n` similar to the function `var.n` in Example 1.5 that will return the estimate $\hat{\sigma}$ (the square root of $\hat{\sigma}^2$). Try this function on the temperature data of Example 1.1.

1.10 (Euclidean norm function). Write a function `norm` that will compute the Euclidean norm of a numeric vector. The Euclidean norm of a vector $x = (x_1, \ldots, x_n)$ is

$$\|x\| = \sqrt{\sum_{i=1}^{n} x_i^2}.$$

Use vectorized operations to compute the sum. Try this function on the vectors $(0, 0, 0, 1)$ and $(2, 5, 2, 4)$ to check that your function result is correct.

1.11 (Numerical integration). Use the `curve` function to display the graph of the function $f(x) = e^{-x^2}/(1+x^2)$ on the interval $0 \le x \le 10$. Then use the `integrate` function to compute the value of the integral

$$\int_0^\infty \frac{e^{-x^2}}{1+x^2}\,dx.$$

The upper limit at infinity is specified by `upper=Inf` in the `integrate` function.

1.12 (Bivariate normal). Construct a matrix with 10 rows and 2 columns, containing random standard normal data:

```
x = matrix(rnorm(20), 10, 2)
```

This is a random sample of 10 observations from a standard bivariate normal distribution. Use the `apply` function and your `norm` function from Exercise 1.10 to compute the Euclidean norms for each of these 10 observations.

1.13 (*lunatics* data). Obtain a five-number summary for the numeric variables in the *lunatics* data set (see Example 1.12). From the summary we can get an idea about the skewness of variables by comparing the median and the mean population. Which of the distributions are skewed, and in which direction?

1.14 (Tearing factor of paper). The following data describe the tearing factor of paper manufactured under different pressures during pressing. The data is given in Hand et al. [21, Page 4]. Four sheets of paper were selected and tested from each of the five batches manufactured.

Pressure	Tear factor			
35.0	112	119	117	113
49.5	108	99	112	118
70.0	120	106	102	109
99.0	110	101	99	104
140.0	100	102	96	101

Enter this data into an R data frame with two variables: tear factor and pressure. Hint: it may be easiest to enter it into a spreadsheet, and then save it as a tab or comma delimited file (.txt or .csv). There should be 20 observations after a successful import.

1.15 (Vectorized operations). We have seen two examples of vectorized arithmetic in Example 1.1. In the conversion to Celsius, the operations involved one vector `temps` of length four and scalars (32 and 5/9). When we computed differences, the operation involved two vectors `temps` and `CT` of length four. In both cases, the arithmetic operations were applied element by element. What would happen if two vectors of different lengths appear

together in an arithmetic expression? Try the following examples using the colon operator `:` to generate a sequence of consecutive integers.
a.

```
x = 1:8
n = 1:2
x + n
```

b.

```
n = 1:3
x + n
```

Explain how the elements of the shorter vector were “recycled” in each case.