# A Short Introduction to R

Jochen Voss

January 27, 2020

Copyright © 2020 Jochen Voss <J.Voss@leeds.ac.uk>

# Contents


**1 The Basics** **1**
1.1 Commands . . . 1
1.2 Variables . . . 2

**2 Loading Data into R** **5**
2.1 Importing CSV Files . . . 5
2.2 Importing Microsoft Excel Files . . . 5
2.3 Checking the Imported Data . . . 6
2.4 Common Problems . . . 7

**3 Data Types in R** **9**
3.1 Numerical Data . . . 9
3.1.1 Numbers . . . 9
3.1.2 Vectors . . . 9
3.1.3 Matrices . . . 10
3.2 Categorical Data . . . 11
3.3 Text Data . . . 11
3.4 Data Frames . . . 11
3.5 Selecting Columns/Variables . . . 12
3.6 Selecting Rows/Individuals . . . 12

**4 Working with Data** **15**
4.1 Mathematical Operations . . . 15
4.2 Summary Statistics for Numerical Data . . . 15
4.3 Contingency Tables . . . 16
4.4 Counting . . . 17
4.5 Correlation and Covariance . . . 17

**5 Creating Plots** **19**
5.1 Scatter Plots . . . 19
5.2 Line Plots . . . 19
5.3 Histograms . . . 20
5.4 Box Plots . . . 21

**6 Linear Regression** **23**
6.1 Fitting a Model . . . 23
6.2 Working with the fitted model . . . 24
6.3 Making predictions . . . 25

**A Important R Commands** **27**

# Chapter 1

# The Basics

## 1.1 Commands

R is a programming language, like Python and Java, but with a focus on statistical data analysis. The biggest advantage of R is that it has built-in knowledge of many statistical concepts and methods. The language is also good at producing scientific plots.

The way to interact with R is by 'executing' commands which tell R what to do. These commands can either be typed directly, or stored in a file and then later be loaded straight from this file. Here are some examples of R executing commands:

```
> 1 + 2 + 3
[1] 6
> 2 * (3 + 4)
[1] 14
> c(1, 2, 3, 4, 5, 6)
[1] 1 2 3 4 5 6
> mean(c(1, 2, 3, 4, 5, 6))
[1] 3.5
```

The first commands, `1 + 2 + 3` and `2 * (3 + 4)`, show that basic arithmetic in R looks very much like it does in school. The next command in the example above, `c(1, 2, 3, 4, 5, 6)`, illustrates how vectors are written in R. For this command, the *name* of the command is '`c`' and `1, 2, 3, 4, 5, 6` are *arguments* of the command, *i.e.* values which determine what the command actually does. In this case, the `c()` command builds a vector, and the arguments `1, 2, 3, 4, 5, 6` specify the contents of the vector. The result is a representation of the vector $(1, 2, 3, 4, 5, 6) \in \mathbb{R}^6$ in R. Commands like this, which take arguments in brackets, are called *functions*. The final command in the list computes the average of the elements of the vector $(1, 2, 3, 4, 5, 6)$. There are many different functions available in R, too many to make a complete list. An alphabetical list of important R functions can be found in appendix A.

Often, commands must be combined to achieve the desired effect. A simple case of this is shown in the last command of the example. This command uses the function `mean()` to compute the average of the elements of a vector. This function needs an argument which specifies the vector to use, and in the example this argument consists of a second command, `c(1, 2, 3, 4, 5, 6)`, to construct the required vector. Such nested commands are often best read 'inside out': first `c()` constructs the vector, and the `mean()` computes the average of the elements.

The main hurdle in learning to use R is the need to learn which commands to use for which task, and how these commands works. The most important thing to keep in mind is, that you won't need to learn about *all* commands to successfully use R. Basic tasks can be solved using a small subset of commands, maybe two dozens in total. The following chapters systematically explain the commands needed to perform typical tasks in R.

Finally, one important function in R is `help()`, which shows the documentation for any function given as an argument. For example, the command

```
help(plot)
```

shows the documentation for the `plot()` command.

## 1.2 Variables

In R you can use variables to store intermediate results of computations. The following transscript illustrates the use of variables:

```
> a <- 1
> b <- 7
> c <- 2
> root1 <- (-b + sqrt(b^2 - 4*a*c)) / (2*a)
> root2 <- (-b - sqrt(b^2 - 4*a*c)) / (2*a)
> root1
[1] -0.2984379
> root2
[1] -6.701562
> root1*root2
[1] 2
```

You can freely choose names for your variables, consisting of letters, digits and the dot character (but starting with a letter), and you can assign values to variables using the assignment operator `<-` (you can also write `=` instead of `<-`). After a value is assigned to a variable, the name of the variable can be used as a shorthand for the assigned value. Variables can not only be used to store just single numbers, they can also store more complex objects and whole data sets.

Since variable names in R cannot contain accented or greek characters, some creativity is required when translating a mathematical formula into R code: instead of $X_k$ we could, for example, write `Xk` and instead of $\tilde{\alpha}$ we could, for example, write `alpha.tilde`.

There is a subtle difference between the use of variables in mathematics and in programming. While in mathematics expressions like $x = x + 1$ are not very useful, the corresponding expression `x <- x + 1` in R has a useful meaning:

```
> x <- 6
> x <- x + 1
> x
[1] 7
```

What happens here is that in the assignment `x <- x + 1`, the right-hand side `x + 1` is evaluated first: by the rules for the use of variables, `x` is replaced by its value `6`, and then `6 + 1` is evaluated to `7`. Once the value to be assigned is determined, this value (the `7`) is assigned to `x`. Consequently, the effect of the command `x <- x + 1` is to increase the value stored in the variable `x` by 1.

One consequence of the fact that `<-` indicates an assignment, *i.e.* some action the computer will perform, is that the order of commands can matter when assignments are involved:

```
x <- 3
y <- x + 1
```

sets `x` to 3 and `y` to 4, whereas

```
y <- x + 1
x <- 3
```

also sets `x` to 3 but set `y` to whatever value `x` previously had, plus 1. If the value of `x` has not been previously set, an error message to this effect will be shown when the line `y <- x + 1` is executed.

The main use of variables is to store data and results of computations for later reference. This has several advantages: First, once a value is stored, the variable name can be used to refer to this value, potentially saving much typing and making

the program shorter. If a descriptive name is chosen for the variable, this can also make the intention of a command much clearer to human readers. Secondly, if the result of a time-consuming download or computation is stored in a variable, the result from the variable can be re-used without performing the computation again. Sometimes, use of variables can make an analysis much faster.

# Chapter 2

# Loading Data into R

Before we can analyse data using R, we have to "import" the data into R. How exactly this is done depends on how the data is stored, and more specifically on which file format is used. Here we consider two commonly used formats: comma-separated values (`.csv`) and Microsoft Excel files (`.xls` or `.xlsx`).

## 2.1 Importing CSV Files

The `read.csv()` command can be used to import `.csv` files into R: if we use the command

```
x <- read.csv("file.csv")
```

then the contents of the file `file.csv` will be stored in the variable `x`. Optional arguments to `read.csv()` can be used to specify whether or not the file includes column names, and allow to deal with variations of how the data may be stored in the file. These are explained in Appendix A (page 30).

The function can not only read data from the local computer, but can also download data from the internet. If the file is on the local computer, you may need to change R's current directory to the directory where the file is stored before calling `read.csv()`. In RStudio you can use the menu "Session ▷ Set Working Directory ▷ Choose Directory. . . " to do this.

**Example 2.1.** In the 2016 version of the MATH1712 module, I performed a small questionnaire in the first lecture. The following R command can be used to load the data from the questionnaire into R

```
x <- read.csv("http://www1.maths.leeds.ac.uk/~voss/rintro/Q2016.csv")
```

The variable `x` now contains the questionnaire results. Instead of directly downloading the data from the internet into R, you can alternatively first download the data using a web browser, and then import the data directly from your computer:

```
x <- read.csv("Q2016.csv")
```

Both approaches give the same result.

## 2.2 Importing Microsoft Excel Files

The easiest way to import Excel files into R is to first convert these files to `.csv` format. To do this:

a) Open the file with the data in Excel.

b) Open a new, empty file (choosing "Blank workbook" in Excel).

c) Copy and paste the relevant cells into the empty file. It is important to just copy the required data and to leave out any explanatory text and and empty

rows/columns. The data must form a tidy rectangle, with one individual per row and one variate per column. Optionally, you can put column headers into the first row.

d) Save the new file in `.csv` format in a folder where you will find it again.

e) Read the resulting `.csv` into R as explained above.

## 2.3 Checking the Imported Data

The following commands can be used to get a quick overview over the data:

- `dim(x)` gives the number of rows and columns of the data frame. Similarly, `nrow(x)` shows the number of rows and `ncol(x)` shows the number of columns in the data frame.
- `str(x)` shows the structure of any R object. This command is often an excellent way to understand the nature of any problems one may encounter while importing data into R.
- `summary(x)` prints, for each variate, the values of various summary statistics. For variates with numeric values, these are the minimum, first quartile, median, mean, third quartile, maximum, and the number of missing values. For attribute data this gives the counts for each observed value.
- `head(x)` shows the first few rows of the data.

Every time you import a data set into R, you should use some of these commands to check that everything went well. In case you discover problems, you should either fix the data file (*e.g.* using Microsoft Excel) or by using the correct options for the `read.csv()` command.

**Example 2.2.** Continuing with the data set from example 2.1, we can try the following commands: We first check that the data has plausible dimensions:

```
> dim(x)
[1] 220   5
```

This tells us that the data has $n = 220$ rows and $p = 5$ columns, as expected. Next, we get some details about the five columns of the data frame:

```
> str(x)
'data.frame':   220 obs. of  5 variables:
 $ gender     : Factor w/ 2 levels "F","M": 2 2 2 2 1 1 1 1 1 2 ...
 $ height     : num  180 183 183 182 164 ...
 $ handedness : Factor w/ 3 levels "L","R","both": 2 2 2 2 2 2 2 2 2 2 ...
 $ travel.time: int  20 25 20 12 5 20 15 10 11 7 ...
 $ R.skills   : Factor w/ 4 levels "basic","good",..: 1 1 1 1 1 1 1 3 1 1 ...
```

This shows, for each column, the name, the "data type" and the first few values. The "data type" is a good indicator to detect problems; it should read `int` for integer valued numeric data, `num` for all other numeric data, and `Factor` for attribute data. For attribute data, the range of observed values is also shown, for example the column `gender` takes the values `f` and `m`, as expected.

Finally, we can print summary statistics for all columns:

```
> summary(x)
 gender      height      handedness  travel.time       R.skills
 F:102   Min.   :147.0   L   : 18   Min.   :  0.00   basic :157
 M:118   1st Qu.:167.0   R   :201   1st Qu.:  5.00   good  :  4
         Median :175.0   both:  1   Median : 15.00   medium: 28
         Mean   :173.7              Mean   : 19.15   none  : 31
         3rd Qu.:180.3              3rd Qu.: 25.00
         Max.   :195.6              Max.   :150.00
         NA's   :23
```

## 2.4 Common Problems

There are many things which can go wrong when importing data. Some commonly encountered problems are the following:

- If the data file contains no line with headers, but the `headers=FALSE` option for `read.csv()` has been omitted, the first row of data will be used in place of the column names. This can, for example, be seen in the `str()` output. The solution to this problem is to correctly use the `headers=FALSE` option.
- If the data in a `.csv` file is not separated by commas but by some other character like a semicolon, R will be unable to separate the columns. When this is the case, the imported data will appear to only have one column, where each entry shows as some garbled version of the data for the whole row. The solution to this problem is to use the `sep=...` option.
- If attribute values are encoded inconsistently, *e.g.* if a mix of `m` and `M` is used to encode the gender "male", this will be visible in the `str()` output. One solution to this problem is to fix the `.csv` file using Microsoft Excel, before trying to import it into R again.
- If a numeric column in the input file contains one or more entries which are neither numbers nor empty, R will interpret the whole column as attribute data. This problem can be detected in the `str()` output, when a numeric column is listed as having data type `Factor`. One solution to this problem is to use Microsoft Excel to remove or fix the offending entry from the file.

# Chapter 3

# Data Types in R

Every object in R represents one of a handful of possible 'data types'. In the examples above we have already seen numbers and strings (short for 'character strings').

## 3.1 Numerical Data

### 3.1.1 Numbers

The basic data type in R are numbers. Most of the time, numbers in R programs are written in the same way which is used in mathematics:

```
> 2+3
[1] 5
> 97*128
[1] 12416
```

R uses a special notation for very small or very large numbers:

```
> exp(100)
[1] 2.688117e+43
```

The expression $x\texttt{e}y$, where $x$ and $y$ are ordinary numbers, stands for $x \cdot 10^y$. Thus, `1e6` is a shorthand notation for one million and `1.3e-5` stands for 0.000013. The output of the command shown above tells us that $\mathrm{e}^{100}$ is approximately equal to $2.69 \cdot 10^{43}$.

### 3.1.2 Vectors

Vector objects in R are useful to represent mathematical vectors in a program; they can also be used as a way to store data for later processing. An easy way to create vector objects is the function `c` (short for 'concatenate') which collects all its arguments into a vector. The vector

$$x = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$$

can be represented in R as follows:

```
> c(1, 2, 3)
[1] 1 2 3
```

The elements of a vector can be accessed by using square brackets: if `x` is a vector, `x[1]` is the first element, `x[2]` the second element and so on:

```
> v <- c(7, 6, 5, 4)
> v
[1] 7 6 5 4
> v[1]
[1] 7
> v[1] + v[2]
[1] 13
> v[1] <- 99
```

```
> v
[1] 99  6  5  4
```

Vectors consisting of consecutive, increasing numbers can be created using the colon operator:

```
> 1:15
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
> 10:20
 [1] 10 11 12 13 14 15 16 17 18 19 20
```

More complicated vectors can be generated using the function `seq`:

```
> seq(from=1, to=15, by=2)
[1]  1  3  5  7  9 11 13 15
> seq(from=15, to=1, by=-2)
[1] 15 13 11  9  7  5  3  1
```

Vectors in R programs are mostly used to store data sets, but they can also be used to store the components of mathematical vectors, *i.e.* of elements of the space $\mathbb{R}^d$. Mathematical operations on vectors work as expected:

```
> c(1, 2, 3) * 2
[1] 2 4 6
> c(1, 2, 3) + c(3, 2, 1)
[1] 4 4 4
```

The function `length()` returns the length of a vector, *i.e.* how many numbers it contains:

```
> v <- c(0,0,1,0)
> length(v)
[1] 4
```

### 3.1.3 Matrices

Individual elements of a matrix can be accessed using square brackets, just like for vectors: if `A` is a matrix, `A[1,1]` denotes the top-left element of the matrix, `A[1,]` denotes the first row of the matrix (as a vector), and `A[,1]` denotes the first column of `A`:

```
> A <- matrix(c(1,2,3,4), nrow=2, ncol=2, byrow=TRUE)
> A[1,1] <- 9
> A
     [,1] [,2]
[1,]    9    2
[2,]    3    4
> A[1,]
[1] 9 2
> A[,1]
[1] 9 3
```

The sum of matrices and the product of a matrix with a number can be computed using `+` and `*`, the matrix-matrix and matrix-vector products from linear algebra are given by `%*%`. (Careful, `A * A` is not the matrix product, but the element-wise product!)

```
> A <- matrix(c(1, 2,
+               2, 3),
+             nrow=2, ncol=2, byrow=TRUE)
> A
     [,1] [,2]
[1,]    1    2
[2,]    2    3
> A %*% A
     [,1] [,2]
[1,]    5    8
[2,]    8   13
> v <- c(0, 1)
```

```
> A %*% v
     [,1]
[1,]    2
[2,]    3
> v %*% A %*% v
     [,1]
[1,]    3
```

The last command shows that vectors are automatically interpreted as row vectors or column vectors as needed: in R it is not required to transpose the vector $x$ when evaluating expressions like $x^\top Ax$.

## 3.2 Categorical Data

## 3.3 Text Data

Strings, short for 'character strings', are used in R to represent textual data, represented as a sequence of characters. Strings are used, for example, to specify the name of a file to load in `read.csv()`, or the axis labels in a plot. Strings in R are enclosed in quotation marks.

Sometimes a bit of care is needed when using strings: the string `"12"` represents the text consisting of the digits one and two; this is different from the number 12:

```
> 12 + 1
[1] 13
> "12" + "1"
Error in "12" + "1" : non-numeric argument to binary operator
```

R complains that it cannot add `"12"` and `"1"`, because these values are not numbers.

Strings can be stored in variables just like numbers and the function `paste()` can be used to concatenate strings:

```
> s <- paste("this", "is", "a", "test")
> paste(s, ": ", "a", "bra", "ca", "da", "bra", sep="")
[1] "this is a test: abracadabra"
> paste("x =", 12)
[1] "x = 12"
```

The argument `sep` for the function `paste` specifies a 'separator' which is put between the individual strings. The default value is a single space character. If the arguments of `paste` are not strings, they are converted to a string before the concatenation:

```
> paste("x=", 12, sep="")
[1] "x=12"
```

## 3.4 Data Frames

A data frame is a two-dimensional data structure which can hold a complete data set. The rows of the data frame correspond to individual observations, the columns correspond to the variables observed. All values within one column must be of the same type, *e.g.* numerical or categorical, but different columns can have different types. (In contrast, for a matrix all elements must have the same type.)

Data loaded via `read.csv()` comes in the form of a data frame. The command `str()` can be used to understand the different columns of a data frame. For example the questionnaire data set considered earlier has the following columns:

```
> str(x)
'data.frame':   220 obs. of  5 variables:
 $ gender     : Factor w/ 2 levels "F","M": 2 2 2 2 1 1 1 1 1 2 ...
 $ height     : num  180 183 183 182 164 ...
 $ handedness : Factor w/ 3 levels "L","R","both": 2 2 2 2 2 2 2 2 2 2 ...
 $ travel.time: int  20 25 20 12 5 20 15 10 11 7 ...
 $ R.skills   : Factor w/ 4 levels "basic","good",..: 1 1 1 1 1 1 1 3 1 1 ...
```

The output shows in the first line that `x` is indeed a data frame containing 220 observations of 5 variables. The following rows show that the variables `gender`, `handedness` and `R.skills` are categorical, while `height` and `travel.time` are numerical.

## 3.5 Selecting Columns/Variables

To access all the data for one or more variables, *i.e.* columns of a data frame, the following commands can be used:

- `colnames(x)` shows the names of the columns.
- The columns of a data frame or matrix `x` can be accessed using either `x$`*name* or `x[,"`*name*`"]`, where *name* stands for the name of the column. For the data from example 2.1, the `height` column can be accessed as either `x$height` or `x[,"height"]`.
- When working with a data set without column names, you can access the columns by number, *e.g.* `x[,2]` is the second column of `x`.

The columns of a data frame or matrix are vectors.

## 3.6 Selecting Rows/Individuals

Commands to access selected rows of the data frame:

- `x[`*i*`,]` gets row *i* of the data frame, *e.g.* `x[1,]` gets the first row.
- If `r` is a vector of integers, `x[r,]` gets the corresponding rows. For example, `x[c(2,5,10),]` gives rows 2, 5 and 10 of the data frame and `x[1:10,]` gets the first ten rows.
- In the R expressions above, you can write a minus sign in front of the row specification to get all the data except for the specified rows. For example, `x[-1,]` gets all rows but the first and `x[-c(2,5,10),]` gets the data with rows 2, 5 and 10 omitted.
- You can combine row and column selection, *e.g.* `x[1:10,"height"]` gives the first ten height values.

You can find the row numbers of "interesting" samples using the `which()` command. The command returns the indices of all elements of a vector which satisfy some condition. For example, we can use this to find all rows corresponding to left-handed students:

```
> rows <- which(x$handedness == "L")
> rows
 [1]  15  20  23  38  60  61  62  98 102 105 111 113 114 131 147 166
[17] 190 199
```

We can use this list to select the corresponding rows of the data set:

```
> x[rows,]
    gender height handedness travel.time R.skills
15       M 185.42          L          20    basic
20       M 176.00          L          20    basic
23       F 165.10          L          30    basic
38       M 185.42          L           5    basic
60       M 190.50          L           2    basic
61       M 147.00          L          20    basic
62       M 185.42          L          10    basic
98       F 165.10          L          10    basic
102      M 185.42          L           5    basic
105      F     NA          L          25    basic
111      F 167.64          L          20    basic
```

```
113      M 187.96          L          22   medium
114      F 170.00          L          20    basic
131      F 167.64          L          25    basic
147      M 176.00          L          10    basic
166      F 177.80          L           0   medium
190      F 170.18          L          35    basic
199      M 180.00          L          20    basic
```

To select samples inside a `which()` statement, you can use `<`, `==`, and `>` for comparisons (note that testing for equality requires a double equality sign). Conditions can be combined using `&` (when both conditions must be true) or `|` (when either condition can be true). For example, the following command lists all left-handed students with 'medium' R skills:

```
> x[which(x$handedness=="L" & x$R.skills=="medium"),]
    gender height handedness travel.time R.skills
113      M 187.96          L          22   medium
166      F 177.80          L           0   medium
```

Using `|` we can find all students which are either very small or very tall:

```
> x[which(x$height<150 | x$height>190),]
    gender height handedness travel.time R.skills
10       M 190.50          R           7    basic
60       M 190.50          L           2    basic
61       M 147.00          L          20    basic
67       M 190.50          R           5    basic
72       M 195.58          R          20    basic
77       F 149.86          R          20    basic
79       M 147.32          R          10    basic
116      M 190.50          R          20    basic
191      M 195.00          R           5    basic
209      M 190.50          R          70    basic
```

# Chapter 4

# Working with Data

## 4.1 Mathematical Operations

Some of the simplest commands available are the ones which correspond directly to mathematical operations. In the first example of this section, we could just type `3 + 4` to compute the corresponding sum. The following table lists the R equivalent of the most important mathematical operations.

| operation | example | R code |
|---|---|---|
| addition | $8+3$ | `8 + 3` |
| subtraction | $8-3$ | `8 - 3` |
| multiplication | $8 \cdot 3$ | `8 * 3` |
| division | $8/3$ | `8 / 3` |
| power | $8^3$ | `8 ^ 3` |
| modulus | $8 \mod 3$ | `8 %% 3` |
| absolute value | $\|x\|$ | `abs(x)` |
| square root | $\sqrt{x}$ | `sqrt(x)` |
| exponential | $e^x$ | `exp(x)` |
| natural logarithm | $\log(x)$ | `log(x)` |
| sine | $\sin(2\pi x)$ | `sin(2 * pi * x)` |
| cosine | $\cos(2\pi x)$ | `cos(2 * pi * x)` |

When these operations are applied to a vector, they operate on the individual elements of the vector. This allows to efficiently operate on a whole data set with a single instruction.

```
> x <- c(-1, 0, 1, 2, 3)
> abs(x)
[1] 1 0 1 2 3
> x^2
[1] 1 0 1 4 9
```

## 4.2 Summary Statistics for Numerical Data

The R commands for computing summary statistics operate on vectors. For example, these commands can be applied to a column of a data frame, selected as explained above. Assuming that $v$ is a vector of numbers in R,

- `sum(v)` computes the sum of the elements of `v`,
- `mean(v)` computes the sample mean of the elements of `v`,
- `var(v)` computes the sample variance of the elements of `v`,
- `sd(v)` computes the sample standard deviation of the elements of `v`,
- `median(v)` computes the median of the elements of `v`,

- `min(v)` computes the minimum of the elements of `v`,
- `max(v)` computes the maximum of the elements of `v`,
- `range(v)` computes the range of the elements of `v`,
- `quantile(v, alpha)` computes the `alpha`-quantile of the elements of `v`, and
- `IQR(v)` computes the interquartile range of the elements of `v`.

One potential pitfall is the fact is that a function `mode()` exists in R, but this function does *not* compute the mode of a sample.

Many of these function have optional arguments to modify their behaviour. For example, the `quantile` function allows to choose which value should be returned in case the quantile is not unique. These options are explained on the help pages for the corresponding commands; for example `help(quantile)` shows, amongst other things, the options available for the `quantile` command. One option, `na.rm=TRUE` is shared by all of the commands above; it can be used to tell R to ignore missing values in a vector and to compute the summary statistic only from the non-missing values.

**Example 4.1.** Continuing with the data set from the questionnaire, considered above, we can try the following commands:

```
> h <- x$height
> min(h)
[1] NA
> min(h, na.rm=TRUE)
[1] 147
> max(h, na.rm=TRUE)
[1] 195.58
> range(h, na.rm=TRUE)
[1] 147.00 195.58
> quantile(h, 0.9, na.rm=TRUE)
   90%
185.42
```

Since `h` has missing values (not all students gave their heights), we need to use the `na.rm=TRUE` option to get useful results. From the R output we see that height values range from 147 cm to 195.58 cm, and that 90% of the responses are $\leq$ 185.32 cm.

## 4.3 Contingency Tables

The R command for summarising attribute data in tabular form is `table()`. This command can be used both for a single variate, or for a pair of variates.

**Example 4.2.** Contingency tables showing either just the counts for both genders, or for all combinations of handedness and gender, can be produced as follows:

```
> table(x$gender)

  F   M
102 118
> table(x$handedness, x$gender)

         F   M
  L      8  10
  R     94 107
  both   0   1
```

The `table()` command returns the counts as a vector or a matrix. This matrix can be stored in a variable, and elements of this matrix can be used in further computations:

```
> t <- table(x$handedness, x$gender)
> t

        F   M
  L     8  10
  R    94 107
  both  0   1
> t[,1]
   L    R both
   8   94    0
> sum(t[1,])
[1] 18
```

## 4.4 Counting

Finally, a count of how often certain cases of observations appear in the data can obtained in different ways:

- `table()` can be used to get the counts for different groups, and the result can then be read off the resulting table.
- `which()` can be used to get a list of all observations in the group (see section 3.6), and then the length of the resulting vector gives the count.
- Alternatively, the same result can be obtained by taking the sum of the boolean vector which contains `TRUE` for all rows where the observation should be counted; this works because `TRUE` and `FALSE` are interpreted as 1 and 0, respectively, in contexts where numbers are expected.

**Example 4.3.** We can get the number of students which gave a height between 160 cm and 170 cm using either of the following two methods:

```
> length(which(x$height >= 160 & x$height <= 170))
[1] 52
> sum(x$height >= 160 & x$height <= 170, na.rm=TRUE)
[1] 52
```

## 4.5 Correlation and Covariance

The functions to compute sample covariances and correlations in R are `cov()` and `cor()`.

```
> x <- c(1, 2, 3, 4)
> y <- c(1, 2, 3, 5)
> cov(x, y)
[1] 2.166667
> cor(x, y)
[1] 0.9827076
```

Both functions have an optional argument `use=...`, which controls how missing data is handled.

- If `use="everything"` (or if `use=...` is not specified), the functions return `NA`, if any of the input data are missing.

  ```
  > z <- c(1, 2, NA, 4)
  > cor(x, z)
  [1] NA
  ```

- If `use="all.obs"`, the functions abort with an error, if any of the input data are missing.

  ```
  > cor(x, z, use="all.obs")
  Error in cor(x, z, use = "all.obs") : missing observations in cov/cor
  ```

- If use="complete.obs", any pairs $(x_i, y_i)$ where either $x_i$ or $y_i$ is missing are ignored, and the covariance/correlation is computed using the remaining samples.

```
> cor(x, z, use="complete.obs")
[1] 1
> cor(y, z, use="complete.obs")
[1] 0.9958706
```

# Chapter 5

# Creating Plots

In this section we show different ways to use graphs to explore a data set.

## 5.1 Scatter Plots

Scatter plots can be used to visualise the joint distribution of paired data $(x_i, y_i)$ for $i \in \{1, 2, \ldots, n\}$. Scatter plots still provide a good means to get an idea of the dependency structure between $x$ and $y$.

If you have stored the data in two vectors `x` and `y`, you can use the command `plot(x, y)` to produce a scatter plot; each sample is represented by a small circle. R provides many optional arguments for the `plot()` command which can be used to adjust the plot.

**Example 5.1.** We can use the following commands to produce a scatter plot of 20 random samples. The options for `plot()` adjust the $x$-range displayed, change the label on the vertical axis, and use diamonds to indicate the data points.

```
x <- rnorm(20)
y <- rexp(20)
plot(x, y, xlim=c(-3, 3), ylab="height", pch=5)
```

(see figure 5.1.)

## 5.2 Line Plots

Line plots can be used to show the functional relationship between two variables. Such plots can, for example, be used to display time series data (where a value changes as a function of time), or to display mathematical functions like the density of a distribution.

Line plots can be produced by adding the `type="l"` (short for 'line') option to the plot command: Instead of marking the points $(x_i, y_i)$ with a small circle, a line connecting the points is drawn.

Additional straight lines can be added to an existing plot using the `abline()` command:

- `abline(h=...)` adds a horizontal line at the given $y$-coordinate.
- `abline(v=...)` adds a vertical line at the given $x$-coordinate.
- `abline(a=..., b=...)` adds a straight line with a given intercept `a` and a given slope `b`.
- The `lwd=...` and `lty=...` options can be used as described above.

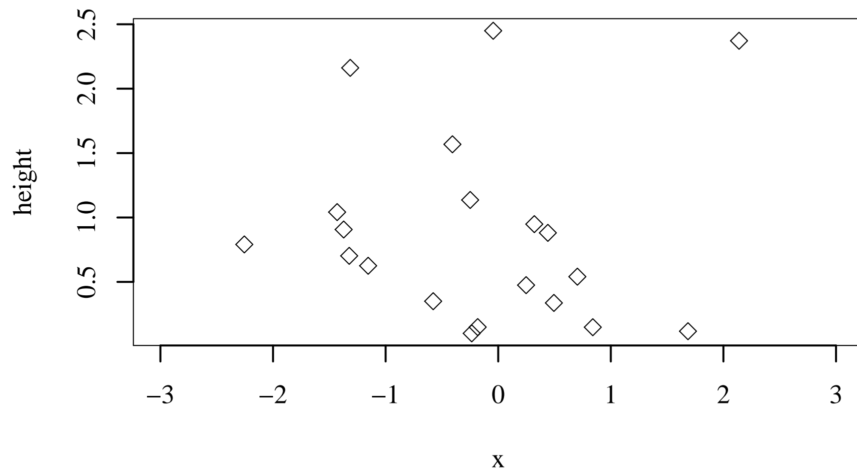


**Figure 5.1.** An illustration of the effect of different options for the `plot()` command for scatter plots.

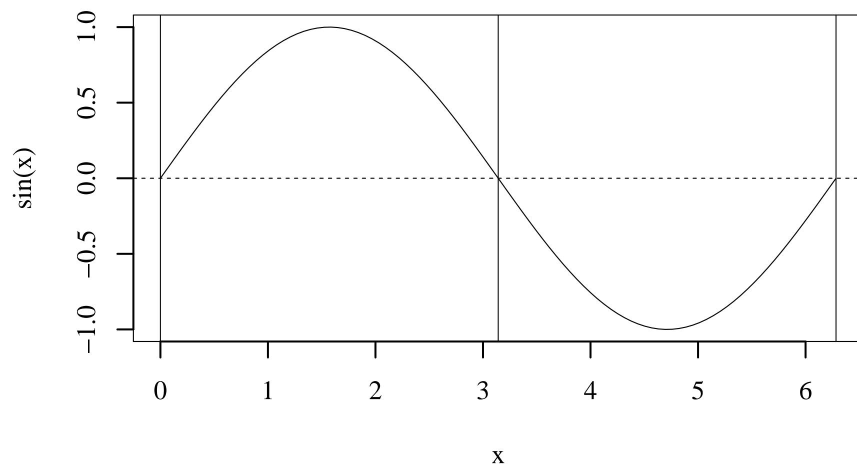


**Figure 5.2.** An example of a line plot in R.

---

**Example 5.2.** The dashed horizonal line, and the three vertical lines were added using the commands

```
x <- seq(0, 2*pi, length.out=100)
plot(x, sin(x), type="l")
abline(h=0, lty=2)
abline(v=0, lwd=0.5)
abline(v=pi, lwd=0.5)
abline(v=2*pi, lwd=0.5)
```

(see figure 5.2.)

## 5.3 Histograms

Histograms are used to illustrate the spread and distribution of a sample of numerical data. The data are summarised by splitting the range into a number of "buckets" and, for each bucket, showing how many (or which fraction) of the data fall into this bucket. Histograms can only be used for single, numerical variables.

If `x` is a numerical vector, the command `hist(x)` can be used to plot a histogram

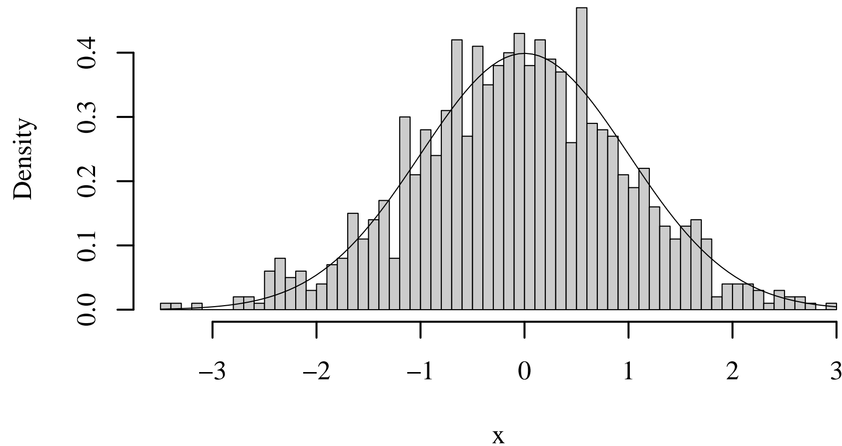


**Figure 5.3.** An example of a histogram plotted in R.

---

of `x`. As for the plot command, there are many optional arguments for `hist()` to adjust the appearance of the histogram:

- The option `main="..."` can be used to change the title text above the histogram. `main=NULL` can be used to remove the title. `xlab=...` and `ylab=...` can be used to change the axis labels as explained above.

- `breaks=...` can be used to adjust the number and size of histogram buckets: `breaks=`$n$ where $n$ is a number tells R to use approximately $n$ histogram buckets, `breaks=a` where `a` is a vector tells R to use the numbers in `a` as the bucket boundaries. For example, the command `hist(x, breaks=seq(-0.5, 10.5, by=1))` uses buckets $[-0.5, 0.5]$, $(0.5, 1.5]$, $\ldots$, $(9.5, 10.5]$ for the histogram.

- `freq=FALSE` can be used to change the $y$-axis value from counts to relative frequencies. If this option is used, the total area under the histogram equals 1 and the histogram forms a probability density.

- `col="gray80"` can be used to fill the histogram bars with light grey.

Example:

```
x <- rnorm(1000)
hist(x, breaks=50, col="grey80", freq=FALSE, main=NULL, xlab="x")
curve(dnorm, add=TRUE)
```

(see figure 5.3.) Since `rnorm()` produces standard normally distributed samples, and since the `freq=FALSE` option was used, the histogram approximates the density of the standard normal distribution. For comparison, the solid line gives the exact density.

## 5.4 Box Plots

# Chapter 6

# Linear Regression

## 6.1 Fitting a Model

The function `lm()` can be used to fit a linear model in R, using the least squares method. The basic use of `lm()` is to call the function as

```
lm(y ~ x)
```

where `x` is the explanatory variable and `y` is the response. This command shows the estimated regression coefficients.

**Example 6.1.** After we have stored the input and response vectors in the variables `x` and `y`, we can execute the following command:

```
> lm(y ~ x)

Call:
lm(formula = y ~ x)

Coefficients:
(Intercept)            x
      2.006        0.720
```

The output indicates that the intercept $\alpha$ was estimated as $\hat{\alpha} = 2.006$ and the slope $\beta$ was estimated as $\hat{\beta} = 0.720$.

Often, it is a good idea to store the fitted model in a variable, *e.g.* using an assignment like

```
m <- lm(y ~ x)
```

In this way, we can later refer to `m` when we work with the fitted model.

There are different ways to specify the form of the model and the data to be used for fitting the model.

a) The most basic way to call `lm()` is the case where the explanatory variables and the response variable are stored as separate vectors. Assuming, for example, that the explanatory variables are `x1`, `x2`, `x3` and that the response variable is `y` in R, we can tell R to fit the linear model $y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + \varepsilon$ by using the following command:

```
lm(y ~ x1 + x2 + x3)
```

Note that R automatically added the intercept term $\beta_0$ to this model. If we want to fit a model without an intercept, *i.e.* the model $y = \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + \varepsilon$, we have to add `0 +` in front of the explanatory variables:

```
lm(y ~ 0 + x1 + x2 + x3)
```

The general form of a model specification is the response variable, followed by the tilde character `~`, followed by a `+`-separated list of explanatory variables. This is the method we used in the example above.

For this form of calling `lm()`, the variables `y`, `x1`, `x2`, and `x3` in the examples above must be already defined before `lm()` is called. It may be a good idea to double-check that the variables have the correct values before trying to call `lm()`.

b) Both for the response and for explanatory variables we can specify arbitrary R expressions to compute the numeric values to be used. For example, to fit the model $\log(y) = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \varepsilon$ (assuming that all $y_i$ are positive) we can use the following command:

```
lm(log(y) ~ x1 + x2)
```

Some care is needed, because `+`, `*` and `^` have a special meaning inside the first argument of `lm()`; any time we want to compute a variable for `lm()` using these operations, we need to surround the corresponding expression with `I()`, to tell R that `+`, `*` or `^` should have their usual, arithmetic meaning. For example, to fit a model of the form $y \sim \beta_0 + \beta_1 x + \beta_2 x^2 + \varepsilon$, we can use the following R command:

```
lm(y ~ x + I(x^2))
```

Here, the use of `I()` tells R that `x^2` is to be interpreted as the vector $(x_1^2, \ldots, x_n^2)$. Similarly, we can fit a model of the form $y = \beta_0 + \beta_1(x_1 + x_2) + \varepsilon$:

```
lm(y ~ I(x1+x2))
```

Here, the use of `I()` tells R that `x1+x2` indicates the vector $(x_{1,1}+x_{2,1}, \ldots, x_{1,n}+x_{2,n})$ instead of two separate explanatory variables.

Details about how to specify models in calls to `lm()` can be found by using the command `help(formula)` in R.

c) If the response and the explanatory variables are stored in the columns of a data frame, we can use the `data=...` argument to `lm()` to specify this data frame and then just use the column names to specify the regression model. For example, the `stackloss` data set built into R consists of a data frame with columns `Air.Flow`, `Water.Temp`, `Acid.Conc.`, `stack.loss`. To predict `stackloss$stack.loss` from `stackloss$Air.Flow` we could write

```
lm(stack.loss ~ Air.Flow, data=stackloss)
```

As a special case, a single dot "`.`" can be used in place of the explanatory variables in the model to indicate that all columns except for the given response should be used. Thus, the following two commands are equivalent:

```
lm(stack.loss ~ ., data=stackloss)
lm(stack.loss ~ Air.Flow + Water.Temp + Acid.Conc., data=stackloss)
```

To check the numerical values `lm()` is using for each column, `lm()` can be replaced with `model.frame()`, using the same formula and `data=` argument. The result is a data frame where the first column holds the values used for the response variable, and the following columns hold the values used for the explanatory variables.

## 6.2 Working with the fitted model

The output of the `lm()` function is an R object which can be used the extract information about the fitted model. A good way to work with this object is to store is in a variable and then use commands like the ones listed below to work with this variable. For example, the following R command fits a model for the `stackloss` data set and stores it in the variable `m`. Many operations are available to use with this object `m`:

a) The command `summary()` can be used to print additional information about the fitted model. Much of this output is beyond the scope of this text.

```
> summary(m)

Call:
lm(formula = y ~ x)
```

```
Residuals:
      Min        1Q    Median        3Q       Max
-0.008357 -0.007073 -0.004975  0.006933  0.018391

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 2.006483   0.002854   703.1   <2e-16 ***
x           0.719967   0.003304   217.9   <2e-16 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 0.009822 on 10 degrees of freedom
Multiple R-squared:  0.9998,    Adjusted R-squared:  0.9998
F-statistic: 4.748e+04 on 1 and 10 DF,  p-value: < 2.2e-16
```

b) The coefficients, as a vector containing the intercept and the slope, can be obtained using `coef(m)`.

```
> coef(m)
(Intercept)           x
  2.0064835   0.7199673
> alpha <- coef(m)[1]
> beta <- coef(m)[2]
> alpha + x[1] * beta
(Intercept)
   1.908622
```

The last command compute the fitted value $\hat{y}_1 = \alpha + \beta x_1$. (The title `(Intercept)` above the result is spurious and should be ignored.)

c) The fitted values $\hat{y}_i = \alpha + \beta x_i$ can be obtained using the command `fitted(m)`.

```
> fitted(m)
        1         2         3         4         5         6
1.9086222 1.9771110 2.7340385 1.8925397 0.4537749 2.3654929
        7         8         9        10        11        12
1.4627322 2.5670394 2.5497549 1.2148465 2.1269700 1.9854003
```

Note that the first of these values coincides with the result we computed above using `coef()`.

d) The estimated residuals $\hat{\varepsilon}_i = y_i - \hat{y}_i$ can be obtained using the command `resid(m)`. These are the distances of the data to the fitted regression line. Positive values indicate that the data are above the line, negative values indicate that the data are below the line.

```
> resid(m)
           1            2            3            4            5
-0.007662386 -0.006876113  0.013866847 -0.007820780  0.018390555
           6            7            8            9           10
 0.001005794 -0.008356636  0.007389788  0.006780861 -0.005424833
          11           12
-0.004524426 -0.006768671
```

## 6.3 Making predictions

One of the main aims of fitting a linear model is to use the model to make predictions for new, not previously observed $x$-values, *i.e.* to compute $y_{\text{new}} = \alpha + \beta x_{\text{new}}$. The command for prediction is `predict(m, newdata=...)`, where `m` is the model previously fitted using `lm()`, and `newdata` specifies the new $x$-values to predict responses for. The argument `newdata` should be a data frame with a column, which has the name of the original variable and contains the new values.

```
> predict(m, newdata=data.frame(x=1))
       1
2.726451
> predict(m, newdata=data.frame(x=c(1,2,3,4,5)))
       1        2        3        4        5
2.726451 3.446418 4.166385 4.886353 5.606320
```

# Appendix A

# Important R Commands

## Summary

**c()** combines its arguments to form a vector.

**colnames()** returns the column names of a matrix or data frame.

**cor()** computes sample correlations.

**cov()** computes sample covariances.

**dim()** returns the number of rows and columns for two-dimensional data.

**lm()** fits linear models, using least squares regression.

**plot()** shows line and scatter plots.

**read.csv()** loads data stored in ‘comma separated values’ (CSV) format into R.

**str()** shows how given data is stored in R, explaining the type and structure of the data.

**table()** creates tables, showing how often each class occurred in a sample of categorical data.

The following sections describe these commands in more detail.

## c()

The function `c()` combines its arguments to form a vector. The arguments can be either numbers or vectors. The result is a single vector, composed of all given numbers and the *elements* of the given vectors, in order. All arguments are converted to a common type while the resulting vector is formed. Details, in particular about the case where arguments with different types are given, can be found using `help(c)`.

## colnames()

The command `colnames()` returns the column names of a matrix or data frame.

## cor()

If `x` and `y` are vectors of the same length, `cor(x,y)` returns the sample correlation between vectors `x` and `y`. The optional argument `use=...` controls how missing data is handled:

- If `use="everything"` (or if `use=...` is not specified), the functions return `NA`, if any of the input data are missing.
- If `use="all.obs"`, the functions abort with an error, if any of the input data are missing.
- If `use="complete.obs"`, any pairs $(x_i, y_i)$ where either $x_i$ or $y_i$ is missing are ignored, and the covariance/correlation is computed using the remaining samples.

The `help(cor)` command gives more information about the use of this function.

## cov()

If `x` and `y` are vectors of the same length, `cov(x,y)` returns the sample covariance between vectors `x` and `y`. An optional argument `use=...` controls how missing data is handled (see the description of `cor()`, above) The `help(cov)` command shows gives more information about the use of this function.

## dim()

For a matrix or a data frame `x`, the command `dim(x)` returns a vector which gives the number of rows and columns of `x`.

## lm()

The `lm()` command can be used to fit a linear model to data, using least squares regression. The only required argument is a "formula" to specify the form of the model being fitted (see section 6.1).

The `help(lm)` command shows gives more information about the use of the `lm()` function. Details about how to specify models can be found using `help(formula)`.

## plot()

The `plot()` command produces line and scatter plots from paired, numerical data. If `x` contains a vector $(x_1, \ldots, x_n)$ and `y` contains a vector $(y_1, \ldots, y_n)$, then the command

```
plot(x, y)
```

produces a scatter plot, showing the observations $(x_i, y_i)$. There are many optional arguments, which can be used to adjust the plot. The most important ones are:

- `xlab="..."` and `ylab="..."` can be used to adjust the axis labels for the $x$- and $y$-axis, respectively.
- `xlim=c(`$a$`,`$b$`)` adjusts the plot so that the $x$-coordinate range from $a$ to $b$ is shown in the plot. This option can, for example, be used to exclude outliers from the plot.
- `pch=...` can be used to change the symbol used to represent samples in the scatter plot. Many values are possible, *e.g.* `pch=0` gives squares, `pch=2` gives triangles and `pch=5` gives diamonds.

The command

```
plot(x, y, type="l")
```

produces a line plot, by connecting consecutive points using straight line segments. In addition to the arguments listed above, the following options can be used:

- `lwd=...` can be used to adjust the line width, *e.g.* `lwd=2` gives thicker lines, `lwd=0.5` gives thinner lines.
- `lty=1` up to `lty=6` can be used to get different forms of dashed or dotted lines.

Further details about the function `plot()` can be found using the command `help(plot)` in R. Graphics parameters used to adjust the plot are explained at `help(par)`.

## read.csv()

The `read.csv()` command imports data from a CSV file into R: if we use the command

```
x <- read.csv("file.csv")
```

then the contents of the file `file.csv` will be stored in the data frame `x`, as a data frame. The most important things to know about the function `read.csv()` are:

a) The filename can either denote a file on the local computer, or a file available for download from the internet. If you are not sure what the filename for a given input file is, you can use the command `file.choose()` in place of the file name, to choose a file interactively. If you want R to read the file directly from the internet, replace the file name with the web address (starting with `http://` or `https://`).

b) By default, R uses the first row of the `.csv` file to set the column names of the data frame and assumes that the actual data starts in row 2 of the `.csv` file. If the file does not contain column names, you can use the `header=FALSE` option with `read.csv()` to tell R that the column names are not included in the data:

```
x <- read.csv("file.csv", header=FALSE)
```

You can see whether this option is needed by opening the file in Excel and looking whether the first row contains headers or not. Alternatively you can inspect the column names and the contents of the first data row in R to see whether everything looks right after importing the data.

c) Sometimes, the columns in a `.csv` file are separated not by a comma, but using a semicolon instead. In this case you need to use the option `sep=";"` when you import the data:

```
x <- read.csv("file.csv", sep=";")
```

d) Missing values should be represented by empty cells in the `.csv` file and are represented as special `NA` values in the data frame. If the `.csv` file uses a different encoding for missing values, the `na.strings` option can be used to tell `read.csv()` which cell values should be interpreted as missing values. For example, `read.csv("file.csv", na.strings=c("", "-"))` can be used for a file where missing values are indicated by either empty cells or cells containing a hyphen.

Further details about the function `read.csv()` can be found using the command `help(read.csv)` in R.

## str()

The command `str(x)` shows a compact description of the structure of the data stored in `x`. If `x` is a data frame, a one-line summary for every column of `x` is shown.

## table()

The command `table(x)` returns a vector, which shows how often each unique value occurs in `x`.

The command `table(x, y)` returns a matrix, which shows how often each unique pair of values $(x_i, y_i)$ occurs. Here, `x` and `y` must be vectors of the same length, The rows of the matrix correspond to values of `x`, the columns correspond to values of `y`.