Introduction to R and basics in statistics
Lecture notes

Stefanie von Felten & Pius Korner-Nievergelt, September 2012

Contents

Preface 3
1 First steps in R 3
1.1 What is R? 3
1.2 R Download and Environment 3
1.3 A first R session 4
1.3.1 Exploring the R console 4
1.3.2 Functions and objects 5
1.4 More specific topics 6
1.4.1 Adding comments and layout 6
1.4.2 Vectors and data frames 7
1.4.3 Reading data from a file 7
1.4.4 Looking at data 8
1.4.5 Manipulating data 10
1.5 Additional Tips 10
1.5.1 The working directory 10
1.5.2 The R workspace 11
1.5.3 Trouble shooting 11
1.5.4 Write data created in R to a file 12
1.5.5 Changing basic settings 12
1.5.6 Date and time formats 12
1.6 Add-on packages 13
1.7 R-help 14
1.8 Further reading 14
2 Graphics 15
2.1 Some basic comments 15
2.2 A worked example 17
2.2.1 Setting up the frame 18
2.2.2 Customizing axes 20
2.2.3 Colors and background elements 21
2.2.4 The actual data 22

2.3 Exporting graphics 22
2.4 Some more options 23
2.4.1 More custom plots and log-axes 23
2.4.2 Getting values from the graphic 24
2.4.3 Overlaying graphs; figure within a figure 25
2.4.4 More than one graph 25
2.4.5 Symbols and fonts and pixel images 27
2.5 Specific graphics packages 28
2.6 Literature 28
3 Probability distributions 29
3.1 The binomial distribution 29
3.2 The Poisson distribution 31
3.3 Discrete and continuous distributions 33
3.4 The normal distribution 33
3.4.1 The central limit theorem 35
3.5 Note on the generation of random numbers 36
3.6 Literature 36
4 Summary statistics 37
4.1 Measures of Location 37
4.2 Measures of dispersion 38
4.3 Quantiles and the boxplot 38
4.4 The standard error of the mean 39
4.5 Confidence intervals 39
4.6 Mean and Variance of different distributions 40
4.7 Literature 40
5 Classical statistical tests 41
5.1 Null-hypothesis testing 41
5.1.1 Test statistics 42
5.2 The t test family 42
5.2.1 One-sample t test 42
5.2.2 The two-sample t test 44
5.2.3 The t test for paired samples 47
5.3 Rank-based alternatives to t tests 48
5.4 Tests for categorical data 49
5.4.1 Compare a proportion to a reference value: the binomial test 49
5.4.2 Compare two proportions: χ^2 test 49
5.5 Outlook: linear models 52

5.6 Literature 53

Preface

We wrote these lecture notes between July and September 2012 in order to accompany several courses we teach. The notes aim to provide a basic introduction to using R for drawing graphics and doing basic statistical analyses. For each chapter, we provide a text file with the plain R-Code, ready to be run in R.
We hope that you are going to find this document and the contributed R-Code useful. If you find mistakes or have feedback of any kind, we will be grateful to know, in order to make improvements.
Regarding the contents, we have drawn heavily on various books and other sources. We do not attempt to claim these contents to be our own intellectual property and give you the references used at the end of each chapter. However, we have of course chosen topics and bits of R-Code which we find useful in our own work as statisticians and biologists.

1 First steps in R

1.1 What is R?

R is a software package for statistics and graphics, which is free in two ways: free download and free source code (see www.r-project.org). More technically, R is a language and environment for statistical computing and graphics under the terms of the (www.gnu.org) Free Software Foundation's GNU General Public License in source code form.
The current R is the result of a collaborative effort with contributions from all over the world. R was initially written by Robert Gentleman and Ross Ihaka—also known as "*R & R*" of the Statistics Department of the University of Auckland. Since mid-1997 there has been a core group with write access to the R source (see www.r-project.org/contributors.html).
R is similar to the S language and environment which was developed at Bell Laboratories (formerly AT&T, now Lucent Technologies) by John Chambers and colleagues. Most code written for S runs unaltered in R.
A strength of R is that along with statistical analyses, well-designed publication-quality graphics can be produced. R runs on all operating systems (Linux, Mac, Windows).

1.2 R Download and Environment

R is freely available from a network of CRAN mirror sites (CRAN: Comprehensive R Archive Network). To download and install R go to www.r-project.org and select a CRAN mirror nearby.
R works code driven via a console, not with menus that you may be used to from other software. The R-console is just a calculator. To document the steps of your analyses, you will write your R code in a text editor (except short bits of code that you do not need to save). From the text editor, you can copy or send (if your editor interacts with R) the code to the R console to execute the function calls. You can save results produced by R to text files or produce graphics in various formats. The R-console itself is normally not saved when you close your R session. However, to be able to reconstruct your analyses any time, you should save the text file(s) containing your R code.
Although you can use any text editor to write and save R code (e.g., Notepad), it is recommended to install a text editor that recognises the R language, such as Tinn-R (http://www.sciviews.org/Tinn-R), RStudio (www.rstudio.org), or Emacs

(http://www.gnu.org/software/emacs/). Advantages of such editors are direct interaction with R and syntax-highlighting. The latter means that different colours are used for commands, arguments and comments, and that corresponding brackets in nested commands are visible. Such syntax highlighting is extremely useful once you have more than just a few lines of code. You can also use the editor that comes with the R installation. However, syntax-highlighting is only provided in the Mac version. We thus recommend using Tinn-R for Windows and the internal editor for Mac.

1.3 A first R session

To start an R session, you can start Tinn-R. Then start R from Tinn-R ("R" in the menu bar, choose "Start/Close and connections", then "RGui"). Alternatively, you can start R and your preferred text editor separately. If you use the editor provided by R itself, open it from within R using the "open script" or "new script" buttons. An advantage of the R editor over the other editors is that it works on all systems without additional installation efforts and normally it corresponds with the R console without problems (the short key "Ctrl + R" sends lines or selections to the R console).

First, we will explore the R-console. Although it is not necessary for this purpose to save all your R code, we recommend that you do so. Write and save all the code you wish to keep in your text file. However, to explore the behaviour of the console, you will sometimes write into the console directly.

1.3.1 Exploring the R console

When you have started R and a text editor, you can write a mathematical expression such as 15.3 * 5 into the text editor and then send the line to the R console by using the predefined short key or copy/paste. You will see your input followed by the output (R's answer) in the R-console:

```
> 15.3 * 5
[1] 76.5
>
```

The > sign is the prompt sign. It means that the R console is ready to accept commands. Our command (15.3*5) appears next to the prompt sign. The next line shows the result. The [1] tells us that this is the first element of the output (there is only one element in this example). The next line shows the prompt sign again. This means that R has done the calculations and is ready to accept the next command. If your command is not complete within one line, a "+" appears instead of the prompt sign and you can simply add the missing code on this line.

```
> 15.3 *
+ 5
[1] 76.5
>
```

If one command is complete at the end of the line, R is ready to accept the next command on the next line. Two commands on the same line need to be separated by a semicolon. The output is given on separate lines, in the same order as the commands were given.

```
> 15.3 * 5; 3 * (4 + 5)
[1] 76.5
[1] 27
>
```

If your cursor is next to the prompt sign, you can use up and down arrows to go back to previous commands. While typing commands, use the horizontal arrows to move within the line. With long commands, it can save time to go back to a previous command and quickly edit it. For now, just try to go back to 15.3 * 5 by using the up arrow. As from now, we will give R code without the prompt signs.

1.3.2 Functions and objects

Instead of arithmetic signs you can use inbuilt functions such as mean, log(), sqrt(), and sin().

```
sqrt(30)
[1] 5.477226
```

You will see later, that you can also write your own functions.

R is an object oriented programming language. This means that you can create objects, using the left pointing arrow "<-" (means assign) to define them, and use these objects for further analyses.

```
x <- 5+39
4*x
[1] 176
sqrt(x)
[1] 6.63325
```

R code consists of two fundamentally different elements: functions and objects. Functions are commands that tell R to do something. A function may be applied to an object, and the result of applying a function is usually an object, too.
All function calls need to be followed by parentheses. Functions can either do something on their own, as e.g., help.start(), or they can do something with something, e.g., sqrt(x). Here, the "object" x is given as an argument to the function sqrt(). Most functions need several arguments (not all of them being objects) to do what they are supposed to. Here is an example:

```
mean(c(0,6,8,3,NA,3,6,6), na.rm=TRUE)
[1] 4.571429
```

To obtain a mean of a vector of numbers that contains a missing value, specified by "NA" in the R language, a second argument, na.rm=TRUE (NA-remove is true), has to be given to tell R to ignore the missing value. Note that we have not given the argument name for the first argument (mean(x=c(...), na.rm=TRUE)), because the function mean() expects as first argument x, a vector that contains the values of which the arithmetic mean should be calculated. But we have to give the name of the argument na.rm because as second argument "trim" is expected rather than na.rm. The rule is that argument names do not need to be given if they appear in the expected order (positional matching). Function arguments do not need to be given in the expected order but then, argument names are required (keyword matching). What is the expected order? We see this in the "usage" section of the help file for each function. Help files can be opened by typing the name of the function preceded by a question mark, such as:

```
?mean
```

From the line:

```
mean(x, trim = 0, na.rm = FALSE, ...)
```

we see that the first argument is x, the second trim, the third na.rm, and that trim and na.rm have default values 0 and FALSE, respectively. This means that you do not need to specify a value for these arguments if you are happy with the default value. However, since x has no default value, we must specify x to make the function mean() work. The following variants are allowed (among others):

```
mean(c(0,6,8,3,NA,3,6,6), 0, TRUE)
```

TRUE is now the third argument and therefore matched with na.rm.

```
mean(na.rm=TRUE, x=c(0,6,8,3,NA,3,6,6))
```

Here, the argument names are required because the order is not as expected.

In this example, you see why we need to use the function c() to combine (or concatenate) the numbers into a vector. If we use

```
mean(4,7,2,6)
```

the function returns 4, because it has calculated the arithmetic mean of the first element within the brackets which is 4. Unfortunately, it does not warn you that trim=7 and na.rm=2 do not make sense! The correct way to obtain a mean of 4, 7, 2, and 6 is

```
mean(c(4,7,2,6))
```

R is case-sensitive. Mean(c(4,7,2,6)) does not work.

1.4 More specific topics

1.4.1 Adding comments and layout

In addition to functions and objects, R code may contain comments that are not interpreted by R. In fact, comments are extremely important to enable yourself and others to understand the individual steps of your analysis, even after months or years. Comments are introduced by the hash sign (#). Everything on a line that is preceded by "#" is not interpreted by R. Comments may include explanations (what is done and why), references (to other files or to literature), titles that help structuring your code into logical units, or lines of code that you currently do not want to be executed. Here is a short example:

```
## Use R as a pocket calculator
1+1
33 + 56            # addition
sqrt(100)          # square root
```

When documenting an analysis, your layout options are somewhat limited. For example, you cannot use different font types. However, some of the options include for example:

- use a different number of # to indicate the importance of a comment (sections, subsections, individual comments)
- use other symbols to follow #, for example # ---------- to mark the beginning of a new section
- use capital letters for titles
- number sections of your code

- use enough white space (empty lines, space between characters) to make your code easy to read

1.4.2 Vectors and data frames

Create two vectors and see what you can do with them:

```
x <- c(1,4,6,7)     # x is a vector with 4 elements
x
y <- c(2,3,4,5)     # y is another vector with 4 elements
y
x + y               # adding elements of vectors
x / y               # divide one vector by another
```

Data sets are usually stored as object of the class data.frame. A data.frame is composed of columns (one column per variable) and rows (one row per observation). Small data sets can be easily created in R directly:

```
## individual data vectors - all with 6 elements

# two numeric vectors
weight <- c(60,72,57,90,95,72)
height <- c(1.75,1.80,1.65,1.90,1.74,1.91)

# a factor (nominal vector)
sex <- rep(c("female", "male"), each = 3)

# a logical vector, i.e. containing only TRUE and FALSE
smoking <- rep(c("TRUE","FALSE"), 3)

## combine vectors to data.frame
data <- data.frame(weight, height, sex, smoking); data
```

Tip: if you name objects, choose meaningful names that make it easy to remember what the object is, but also keep names short. Long names are a hassle if you have to type them repeatedly. Where you use an abbreviation, explain it with a comment (after a #-sign). Remember that R is case-sensitive, so an object called fish cannot be accessed with Fish.

Some R packages contain their own data sets, which can be accessed easily:

```
library(ISwR)             # loads the package ISWR
data(package="ISwR")      # to see all data sets in that package
data(bcmort)              # loads the data set bcmort
bcmort                    # to inspect bcmort
```

1.4.3 Reading data from a file

In most cases, however, you will prepare and store your data in a spreadsheet (Excel, Access, LibreOffice). To read the data into R, the easiest way is to safe them as tab- or comma-separated text file (.txt, .csv), and use the function read.table().

```
# general function read.table
read.table(file, header=T)    # default sep="" (space)
# to read a comma-separated file
read.table("D:/birds.txt", header=T, sep=",")
```

Give the file path (where the file is located on your computer) and the file name within quotation marks. For the path use forward slashes or double backward slashes (not single

backward slashes!). A file path can be given in “absolute” or “relative” form. An absolute path works independent from your current working directory and starts with the drive on your computer where the file is located (for example “D:/birds.txt” for a file birds.txt on drive D). A relative path starts at your working directory (see section 4.3.3, below) that you can find using the function getwd(). The argument header = TRUE tells R that the first line in the text file contains the variable names. The default with read.table() is header = FALSE; since in most cases you will have variable names in the first row, you need to type header=TRUE. If you add the argument sep="\t", R will interpret tabulators (and only tabulators) as column separators. By default, read.table() will assume that columns are separated by "white space", i.e. space or tab.

As an example, and to look at a data set more closely, read the data file parusater.txt from the web (a url can be specified instead of a local file path):

```
dat <- read.table("http://www.oikostat.ch/data/parusater.txt", header=TRUE)
dat
```

This text-file is a tab delimited text file. We do not have to define the argument sep, because by default, the separator is ‘white space’.
If your data contains space that you don't want to be interpreted as column break (data columns such as "comments" often contain space) you either have to tell R to only use tabs as column separators by providing the argument sep="\t", or you first replace all spaces with e.g. underscore in excel or word. Spreadsheet programs like Excel provide additional options for saving data, such as comma-separated text (file extension usually .csv). Depending on your settings, Excel will use “,” or “;” as separator (the latter is often more useful because “;” is rarely used except as separator).
Note that there should be only one first row that contains the variable names and no second (or more) rows containing e.g. explanations of the variables or their units of measurement. Such information needs to be stored elsewhere. For example you may have an excel file with the data in one worksheet and explanations on a second worksheet.
Tip: common error messages when trying to read in a .txt file are "incomplete line" or "line X did not have Y elements". If you think that you do have the same number of elements on each line, possibly you have a "strange" symbol in your data file; symbols that potentially cause problems may be mutated vowels (Umlaute ä, ö, ü), semicolons, comas, single and double quotation marks, dash, slashes. You may want to remove such symbols or replace them with underscores (or ä with ae etc).

1.4.4 Looking at data

To have a look at the object "dat" that we have just created above, we can type "dat" into the R console. However, for large data files this output is not convenient. Better use

```
str(dat)
'data.frame':   28 obs. of  6 variables:
 $ Land: Factor w/ 5 levels "Algerien","Bulgarien",..: 5 5 3 3 5 5 5 5 5 5
...
 $ age : int  4 4 3 4 3 3 3 4 3 4 ...
 $ sex : int  0 0 2 1 0 0 0 1 0 2 ...
 $ Gew : num  9.1 9.2 9.4 8.9 9.5 9.9 8.9 9.5 9 8 ...
 $ P8  : num  47 50.5 46 48.5 48.5 49 49.5 50 49 47 ...
 $ wing: num  61.5 65 59.5 63.5 64 64 65.5 64 64.5 61 ...
```

The function str() gives an overview over the object "dat". It tells you that "dat" is a data.frame with 28 observations (rows) of 6 variables (columns). For each variable you see how R has interpreted it, i.e., as a numeric variable ("int" for integer numbers, "num" for decimal numbers) or as categorical variable ("Factor") variable.
Another useful function to look at the first 6 rows of a data.frame is head(). Try tail() too.

```
head(dat)
      Land age sex Gew   P8 wing
1  Schweiz   4   0 9.1 47.0 61.5
2  Schweiz   4   0 9.2 50.5 65.0
3 Russland   3   2 9.4 46.0 59.5
4 Russland   4   1 8.9 48.5 63.5
5  Schweiz   3   0 9.5 48.5 64.0
6  Schweiz   3   0 9.9 49.0 64.0

tail(dat)
```

Note that a vector that consists of numbers will always be interpreted by R as integer or numeric, unless explicitly specified as factor (using the function factor() or as.factor()). For example, the variable “sex” in “dat” is recognised as integer, because it contains numeric codes (0, 1, 2). Alternatively, we could have used character strings to code this vector (“unknown”, “male”, “female”), as we did in the data frame “data”. Character vectors in data frames are automatically interpreted as factors. As a consequence, a numeric vector becomes a factor if you mistype 0.2 as 0,2 when “.” is defined as the decimal point (since “,” is a character).

```
str(dat)
str(data)
```

Try also:

```
summary(data)
```

This function shows a summary of each variable in the dataset. Note the difference between factors and numeric variables.

Elements of a data.frame can be accessed using squared brackets [row, columns]:

```
dat[3,5]          # the third element of the fifth column
[1] 46
dat[,2]           # whole second column (as no row is specified)
[1] 4 4 3 4 3 3 3 4 3 4 4 4 5 4 0 0 3 3 3 0 5 4 4 4 5 4 4 5
dat[,c(2,4)]      # second and fourth column
dat[3,]           # whole third row (as no column is specified)
      Land age sex Gew P8 wing
3 Russland   3   2 9.4 46 59.5
dat[dat$Land=="Schweiz",]     # all Swiss observations
```

Alternatively, a column can be extracted by using the dollar sign.

```
dat$age          # this is equivalent to dat[,2]
 [1] 4 4 3 4 3 3 3 4 3 4 4 4 5 4 0 0 3 3 3 0 5 4 4 4 5 4 4 5
```

The function table() gives contingency tables of factors. This can be useful for checking a variable and for detecting typing errors.

```
table(dat$age)
```

```
 0  3  4  5
 3  8 13  4
table(dat$age, dat$sex)
    0 1 2
  0 2 1 0
  3 4 3 1
  4 2 8 3
  5 0 1 3
table(age=dat$age, sex=dat$sex) # to make the table self-explanatory
   sex
age 0 1 2
  0 2 1 0
  3 4 3 1
  4 2 8 3
  5 0 1 3
```

To apply a function such as mean() or sd() to groups, e.g. the mean of P8 (length of the 8th primary) by sex:

```
tapply(dat$P8, dat$sex, mean)
       0        1        2
48.25000 50.03846 48.00000

?tapply     # to find out more
```

1.4.5 Manipulating data

Simple data frame manipulations can easily be done in R. This is very convenient to document all changes (in contrast to manipulations in Excel). Two basic examples:

```
dat$sex <- as.factor(dat$sex)       # turns sex into a factor
str(dat)

# add a new variable (nonsense example)
dat$age.weight <- dat$age * dat$wing  # product of age and wing length
```

1.5 Additional Tips

1.5.1 The working directory

R refers to a working directory. To find your current working directory, type

```
getwd()
```

If you want to read a file from your working directory, you do not need to specify a file path (just the filename). To read a file from another directory, you can either specify the file path in the read.table() function (as we did above), or you can set the working directory accordingly, using setwd(). Try to set the working directory on the folder “datasets” containing the datasets used in our course. Then read the data set size.txt:

```
size <- read.table("size.txt", header=TRUE)
```

A quick way around is the function file.choose(), which will allow you to browse your folders and choose a file manually:

```
size <- read.table(file.choose(), header=T) # choose size.txt manually
```

When you open an R script written with a specific R editor , the working directory is usually automatically set to the directory where your script is located (try if it works with your editor). If you want to read data and always get the error "cannot open the connection", type getwd() to see where you currently are and adjust your working directory (setwd()) or your path.

1.5.2 The R workspace

During an R session, all objects you create are stored in the workspace.

```
ls()              # lists all objects
```

It can be useful to remove all objects from the workspace, for example before you start working on a new analysis. When quitting R, you are asked whether a “workspace image” should be saved. If you do so, this workspace image is restored the next time you start R. To remove objects (do only if you really want to):

```
rm(list=ls())     # removes all objects
```

For example, if you have accidentally saved the workspace when quitting R, you can open R again (all your objects are back, as can be seen e.g. by typing ls()), remove all objects (using rm(list=ls()) and select "save" when you now quit R again.

It may be advisable especially in long R scripts to have a first section that loads everything needed such as data files and packages and that does preparatory steps, e.g. transforming variables or calculate new variables. Each time you work on this script you sent the entire first section to the R console and then are ready to continue, because all objects created in this first part are now in the workspace. Sending the entire content of a script is often too time consuming.

1.5.3 Trouble shooting

When something does not work (i.e. you get an error, or a result that does not make sense), go through the code line by line and check each object that is created (using **`str()`** or **`head()`** if needed). You may find that you misspelled a variable name (e.g. due to an upper-lower-case mistake). Here is an example: you typed

```
apply(matrix(c(1,1,1,2,2,2),nrow=2),2,sum)
```

(this returns the sum of each column of the object given as the first argument) expecting R to return the vector (3,3,3), because you think that the matrix command produces a matrix

1 1 1
2 2 2

To check that, send only

```
matrix(c(1,1,1,2,2,2),nrow=2)
```

and you will find that the values (1,1,1,2,2,2) are filled in column-wise in the matrix function, resulting in another matrix than you intended. This step-wise inspection of a doubtful bit of code usually helps to locate the error. In this case, the error may be corrected by adding the argument byrow=TRUE in the matrix function (by the way: byrow=T suffices, too).

1.5.4 Write data created in R to a file

You may want to open a data frame (e.g. containing results) that you created in R in another program or just to save a data frame you created for later use. We try this with the data frame dat used above (where we added a variable age.weight):

```
write.table(dat, "mydata.txt", row.names=F)
```

Hereby, dat is the object you want to save and "mydata.txt" is the name of the text file created in the current working directory (unless you specify a path before here.txt). row.names=F prevents that row names are exported (often just line numbers). You can now open mydata.txt e.g. in Word or from within Excel (be sure that in the "open" window in Excel "all readable files" or "all files" is selected, otherwise you may not see here.txt). Excel then opens its text import assistant, allowing you to specify that space is the separator (the default in write.table()) so that your results are now in a table format analogous to the original data frame in R.

1.5.5 Changing basic settings

Some interesting aspects can be controlled by the function options(). Type ?options to see the help file. For example, you can change the number of digits printed in R. Try the following:

```
sqrt(8)
options(digits=12)
sqrt(8)
```

Note that this changes only what is printed. R stores more digits than are printed. Try also:

```
options(OutDec=",")
sqrt(8)
```

This changes the decimal sign in the R console as well as in R graphics. Furthermore, the notation of the output can be changed by the argument scipen (e.g., exponential: 1.23e+10 or fixed: 12300000000).

1.5.6 Date and time formats

We found that handling dates and times is tricky. Date/time variables can be defined as such, allowing you to calculate differences between times. However, when defining a variable to be date/time, it depends on your computer system time how exactly R defines your time object. You easily get in troubles with time zones and summer time or if system time changes (due to traveling or sending data to be analysed elsewhere). It is possible to define time zones, but as far as we know it is e.g. not possible to measure time on CET (central European time) but forcing it not to use summer time (CET during summer becomes CEST = central European summer time). Many field data are measured on the local time but ignoring summer time. Something measured at 2:30 in the night summer time ends will produce NA because during this night the clock jumps from 2:00:00 h to 3:00:01 h and 2:30:00 does not exist.
One option is to use the technical time zone UTC (universal time coordinated) that has no summer time. May be the safest way is to define a starting time point such as 1. January 00:00:00 h of the year your study started, and then calculate the seconds since then (or whatever is the precision of time measured). Don't forget leap days (29th February) if needed and relevant.

We recommend to store the original time variable as a character variable and not to alter it (it's the backup). Useful functions are strptime(), POSIXct(), date(). For example:

```
strptime("14.3.2009", format="%d.%m.%Y")$yday
```

provides the Julian day for the 14th of March. However, note that the Julian day for the 1st of January is 0, so usually you want to add 1 to this value. Moreover, the leap day is always counted, even if the year is not a leap year! So maybe you only want to add 1 for the Julian days < 58...

1.6 Add-on packages

Maybe the most important advantage of R is that useful functions and procedures written by users can be published and made available for everyone else on CRAN. For example, the two ecologists J. Oksanen and B. O'Hara have written the function diversity() that calculates different ecological diversity indices such as Shannon, Simpson, Fisher and others. The authors have packed this function together with other functions for the analysis of community ecology into the package "vegan" that can now be downloaded from CRAN together with documentation and help files. There are currently more than 3000 packages available. These packages are reviewed in the Task Views on CRAN by topics (see http://cran.rakanu.com/). This Task Views help to find the right package for your problem.
Note that there are three different types of R-packages: the most commonly used, the fairly often used, and the specific packages (only used by a small part of the R community). The most commonly used packages, such as "base" or "stats" are automatically downloaded and installed when you install R on your computer. These packages are automatically loaded to the R console when you start R, and their functions, e.g. mean() or weighted.mean(), are available without the need to load or install any package. By the way, all R functions are part of a package. The package name is given in the header of the help file in curly brackets. Fairly often used packages, such as "nlme", are automatically downloaded and installed on your computer when you install R but they are not automatically loaded to the R console when you start up R. To use the function lme() that is part of the package "nlme", you need to load the package "nlme" to the R console by using the function library():

```
library(nlme)
```

Otherwise the function lme() cannot be found by R. Specific packages, such as "arm" or "vegan", are not automatically installed on your computer when you install R. Such packages have to be downloaded and installed manually. The simplest way to install a package on your computer is to use the function install.packages()

```
install.packages("arm")   # the name of the package has to be given in quotation marks
```

A pop-up window will appear and you have to choose a CRAN mirror. Everything else follows automatically. Of course, this only works when you are connected to the internet. An alternative way is to download a zip-file of the package from the CRAN and then use the "Install packages from local zip-files..." button from the menu packages to extract and install the package.
This system of different package types helps to keep the basic installation of R slim, and the time needed for installation and start up of R short.

1.7 R-help

As we have already seen, typing a question mark and the name of a function opens the documentation of the function. However, this only works if the package containing the function is loaded. R-help is therefore related to the previous section on add-on packages.

```
?lm          # works, lm belongs to the package stats
?lmer        # doesn't work, lmer belongs to lme4

library(lme4)
?lmer        # now it works
```

You can also search for a topic, within packages on your hard drive

```
??"linear model"        # (fuzzy matching)
??"arcus sinus"
help.search("linear model")
??lm                    # (expression matching)
help.start()
```

Or you can search CRAN for any function and package that contains it: www.r-project.org -> Search -> Google – Toolbar.
Alternatively, just use your preferred search engine and type for example "arcus sinus in R". It can also help to look at the help file of a similar function of which you know it exists, e.g. ?sin if you look for the arcus sinus, and check the "See also" section at the bottom. (If you try ?sin, you will see that this is the help file for all trigonometric functions and your function asin is already listed at the top; but often you find hints in the "See also" section).

You will use the R-help extensively, to find out what arguments can be provided in a function, what values the function returns and how exactly they are calculated. (For Mac users: the Mac console and editor list the arguments in the lower window frame whenever you are inside the brackets of a function.)

1.8 Further reading

This is a very short introduction to R. If you are keen to learn more about R and statistics, we recommend Daalgard (2008) and Crawley (2007) for an introduction to applied statistics with R, and Chambers (2008) for more advanced R users. Korner-Nievergelt (2010) give a German introduction to R, similar to this one. A more detailed English introduction to R written by W. N. Venables and D. M. Smith can be downloaded from the net (http://cran.r-project.org/doc/manuals/R-intro.pdf).

Chambers, J. M. (2008). Software for Data Analysis, Programming with R. New York, Springer.
Crawley, M. J. (2007). The R Book. West Sussex, John Wiley & Sons.
Daalgard, P. (2008). Introductory statistics with R. New York, Springer.
Korner-Nievergelt, F. and Hüppop. O. (2010). Das freie Statistikpaket R: Eine Einführung für Ornithologen. Vogelwarte 48: 119-135.

2 Graphics

Apart from being a powerful statistics software package, R is very convenient for visualizing data. Many built-in functions allow you to create graphics quickly and easily. However, in order to design a graphic exactly as you want, it is often necessary to create your graphic step by step. The flexibility you have in R is a great advantage. You can export graphics to various formats for publication or in order to open the graphic in another program, e.g. in Adobe Illustrator.

There are so many details about how to produce graphics that we do not attempt to give a complete overview here. Literature listed at the end of this chapter can be consulted for a more comprehensive overview. After a few basic comments we create an example graphic, introducing the functions we found most useful in our own ecological/statistical work. Thereafter, we discuss how to export graphics and focus on other topics we often encounter.

2.1 Some basic comments

The code to produce graphics can be added to the file containing the statistical analysis. For large analyses needing extensive R-code, it can be useful to write R-code for graphics in a separate file. When you send a graphics-function to the console, a graphics window opens where the graphic is drawn. The window can simply be closed again, or saved to a file, or you can copy-paste the graphic. The main function to plot data is `plot()`.

`plot()` is a generic function which means that it changes its behaviour depending on the arguments you specify. The first argument can be a function of the form
`y-value ~ x-value`
(the wave line is called "tilde" and it is used in R in the sense of "is a function of").
Alternatively, the first argument of `plot()` can contain the x-values and the second argument the y-values, separated by a comma.

```
plot(data$weight ~ data$height)    # scatterplot
plot(data$height, data$weight)     # same plot, xy-data provided differently
plot(height~sex, data)             # boxplot, because sex is a factor
```

In the last example, the data frame "data" is provided as an argument, allowing the use of variable names in the formula. This only works if you use "~" (called tilde) in the first argument. Note that you need to have the objects called data, and dat for the next couple of commands. Both objects were created in the first chapter).

```
plot(dat)                      # plots all variables in the data set
plot(wing ~ P8, dat)           # scatterplot
plot(wing ~ factor(sex), dat) # boxplot
plot(dat$wing)                 # index plot
```

Also try the following commands:

```
hist(dat$wing)
boxplot(wing~sex, dat)

demo(graphics)
```

Many general plotting settings can be changed with the function `par()`. For example, if you want to have two separate panels in the graphics window next to one another, `par(mfrow=c(1,2))` (1 row- 2 column layout) has the effect that your first graphic only uses up half the space of the window (left panel) and the next graphic will be drawn to the right of it (right panel). Whenever you close the graphics window, the par-arguments are set back to their defaults. Try out:

```
par(mfrow = c(1,2))
plot(wing ~ P8, dat)
plot(wing ~ factor(sex), dat)
```

The graphics so far have used standard x- and y-axes. Sometimes you want to plot special axes - we do that in the worked example afterwards. As a preparation, we first deal with a simpler case: We have 3 values each for the months March, April, and June. Here's the data:

```
mo <- data.frame(
      month = c("Mar","Mar","Mar","Apr","Apr","Apr","Jun","Jun","Jun"),
      month.n = c(3,3,3,4,4,4,6,6,6),    # the number for the month
      value = c(5,3,7,7,6,6.5,1,4,3))

plot(mo$month.n, mo$value)
```

This produces a very first sketch of the graph which does not satisfy us.

```
plot(mo$month.n, mo$value, las=1, xlab="month", ylab="value")
```

las=1 has turned all axis labels horizontal, xlab and ylab receive the axes titles we want. We want the y-axis to start at 0, and we want filled symbols:

```
plot(mo$month.n, mo$value, las=1, xlab="month", ylab="value",
     ylim=c(0,max(mo$value)), pch=16)
```

pch means point character (see 2.4.5 for some of the available characters). The ylim-argument sets the limits of the y-axis. By default, R adds about 4% on the bottom and on the top, so that data points don't lie on the graphics frame. If you want the x-axis to go through y=0, add the argument `yaxs="i"` (y-axis style). Because some points now lie on the upper edge of the graphics frame, we increase the upper limit of the ylim-argument by a small factor, e.g. `ylim=c(0, max(mo$value)*1.04)`. We will do this in the worked example, too. For now we are happy with what we have, except for the x-axis: we want to have tick-marks betweenmonths and the month-names centred in-between. We first plot the graphic without an x-axis:

```
plot(mo$month.n, mo$value, las=1, xlab="month", ylab="value",
     ylim=c(0,max(mo$value)), pch=16, xaxt="n")
```

Now we are free to draw tick marks and labels where we need them. Let's agree that at x=3 we have the centre of March (where we plot the data points). Thus, March starts at 2.5 and ends at 3.5. We can add tick marks at 2.5, 3.5, ... 6.5 - but first we need to enlarge the range of the x-axis, because it starts somewhere just before 3 and does not include 2.5. Thus, we also set axis limits by using the xlim-argument:

```
plot(mo$month.n, mo$value, las=1, xlab="month", ylab="value",
     ylim=c(0,max(mo$value)), pch=16, xaxt="n", xlim=c(2.3, 6.7))
```

Note that we don't see where x=2.5 is, because we have not drawn an x-axis yet, but the space is there; we have more space between the data points and the left and right edges of the graph.

```
axis(1, 2.5:6.5, labels=F)
```

This draws the tick marks on axis 1 (first argument of axis: 1 = x-axis), at the values 2.5, 3.5, 4.5, 5.5, and 6.5 (abbreviated with 2.5:6.5), and it puts no labels to the tick marks (labels = FALSE or, as seen, F is enough). Now we want the month names in-between, i.e. at the positions 3, 4, 5, 6:

```
axis(1, 3:6, labels=c("Mar","Apr","May","Jun"), tick=F)
```

Again, we add an axis at side 1 (the x-axis), at the positions 3 through 6, at these positions we write the labels given in the labels-argument, and, since we don't want to have tick marks at these positions, we set tick=F.
If you don't want to have the May-label because you don't have data from May (but you still want the data points to be spaced correctly), the last command needs to be replaced with:

```
axis(1, c(3,4,6), labels=c("Mar","Apr","Jun"), tick=F)
```

You see that you are completely free and flexible to draw whatever you want. However, this also involves the danger of making errors. As soon as you draw the axes of a graph by hand, you need to be extremely careful and make surethe data points come to lie at the right position with respect to the axes.
Now you are prepared to go through the worked example. It is not a simple example (you find many simple cases in beginners books), because its aim is to show you the flexibility you have. Execute commands step by step, and may be you have to do the example more than once to start to understand your possibilities (which, of course, are not exhausted in the example!).

2.2 A worked example

Imagine we have measured brain activity in turtles every minute over a number of days. Because we are especially interested in the course of the brain activity during dusk we calculate the mean brain activity during three dusk periods (civil, nautical, and astronomical, each lasting 30 min) and for each hour relative to dusk (i.e., 12 hours before and after dusk). We want to plot mean brain activity values with standard errors per hour. We also want to have background shades showing the dusk periods and night. We want to draw brain activity on a log scale but have the y-axis labelled with the original values. The plot we aim at looks like this:

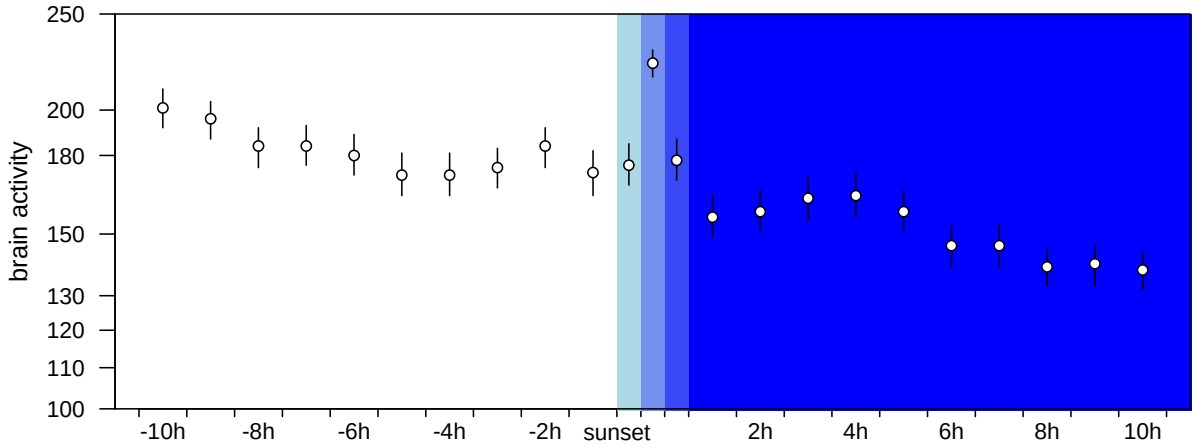

Figure: Turtle brain activity during at different times of the day.

Before plotting we have calculated the means and standard errors from the data with a mixed-effects model, correctly dealing with repeated measurements of individuals, autocorrelation and so forth. By the way, the data we are using here are fake data, derived from original data on a different animal with a different parameter measured. The results were saved in the data frame "ba" for brain activity - for our purpose we can generate it as follows:

```
ba <- data.frame(
     hrdusk=c("-10h","-9h","-8h","-7h","-6h","-5h","-4h","-3h","-2h","-1h","c","n","a",
               "1h","2h","3h","4h","5h","6h","7h","8h","9h","10h"),
     pred=c(201,196,184,184,180,172,172,175,184,173,176,223,178,156,158,163,164,158,146,146,139,140,138),
     se_u=c(192,187,175,176,172,164,164,167,175,164,168,216,170,149,151,155,156,151,139,139,133,133,132),
     se_l=c(210,204,192,193,189,181,181,183,192,182,185,230,187,164,166,171,173,166,153,153,145,146,144))
))
```

Note the spacing we have used to align code pieces in a nice way. hrdusk is the hour relative to dusk, with the twilight periods given as "c", "n", and "a" (for civil, nautical, and astronomical). pred contains the predicted values, se_u and se_l contain the upper and lower ends of the standard errors.

We create this graphic entirely "by hand", defining all its elements ourselves. Note that at this stage we don't have useful values for the x-axis yet - hrdusk does not contain the x-values at which we want to draw the data, but it informs us of the period (the hour before or after dusk and the dusk-periods) the data belong to.

2.2.1 Setting up the frame

`plot()` opens a plotting window with a pre-defined aspect ratio that is unsuitable for our wide plot. To set up a plotting window with the desired dimensions, we write

```
windows(8,3)
```

This opens the plotting window with width and height in the given ratio (according to the help file the units are in inches, but since the graphic will be a vector graphic it can be scaled to any size with the given ratio with no loss of quality).
This is one of the few moments where Windows and Mac users have to do something else: `windows()` is for Windows, on Mac it is:

```
quartz(,8,3)
```

The comma before the 8 is not a typo. Rather, quartz takes as first argument a title (of the plotting window), which we don't want to specify here, so we leave it empty. Unfortunately, a graphics code written on Windows cannot be expected to look exactly the same when you run it on Mac and vice versa: some proportions and sizes will change and you will have to adjust the code. The code given here was written on a Mac, so on Windows the result probably looks a bit odd - may be you want to do the adjustments as an exercise!

We now have our plotting window. Before we use `plot()` we adjust the white space that is left around the plot (the margins, i.e. space for the axes labels). The default leaves space on all four sides. We need only little space on top and on the right, and we set that with

```
par(mar=c(1.5,4,0.5,0.5)).
```

The mar-argument in par contains the width of the bottom, left, upper, and right margin. The unit is "lines", fractions are allowed as seen in the example.

Now we plot the data. `plot()` usually takes as its first two arguments the x- and y-values (or the formula notation) as seen above. Thus, it expects two vectors of equal length. In our case we don't have the numeric values for the x-axis in our data frame (ba$hrdusk is a factor due to the twilight periods, i.e. it does not give the spacing we want), so we simply type NA as the first two arguments of `plot()`. We can do that, because we set the arguments xlim and ylim in the plot function, i.e. we set our custom axes limits. If we don't do that but provide x- and y-values, R calculates the limits from the data. We have decided to put "sunset" at the x-value 0, thus our fist tick mark on the x-axis has to be at -10 and the rightmost tick is at x=11.5 (dusk periods are 0.5 hours wide).

```
plot(NA, NA, xlim=c(-10.5,12), ylim=c(log(100),log(250)), type="n",
     axes=F, xlab="", ylab="", yaxs="i", xaxs="i")
```

The y-limits we need are between 100 and 250, which we read from the data frame but which we also could calculate e.g. with `min(ba$se_l)` etc. We want to draw the values on a log scale, thus the log (in R, this is the natural logarithm). The type-argument defines the type of plot, default is "p" for points, alternatives are e.g. "l" for line, "b" for both points and lines, "h" for histogram-like, "s" for step, and "n" for nothing. In our case the argument could have been omitted because of the NA's as first arguments.
Because we want to draw our axes by hand, we write axes=F (FALSE) and we want no x- and y-labels (xlab="", ylab=""), too. Normally, R adds about 4% space on each side to the axes limits. If we want the axes to precisely cover the range specified by ylim or xlim, we add yaxs="i" and xaxs="i" (axis style). This is necessary, for instance, when you want the x-axis to go through y=0. In such a case you may write yaxs="i" and ylim=c(0, max(yvalues)*1.04); if you don't multiply the maximal y-value by a little more than 1 the largest points will lie on the upper edge of the plot which looks not nice (thus, we remove the 4 % added by R only at the lower end of the axis).

This plot-command does not draw anything - not even a box around the plotting area! But everything is set up to start to draw things we can see! If at this stage you want to check whether you are satisfied with the general layout, you can write `box()`, which draws a box around the plotting region. It is not really needed, because as the next steps we draw the axes and quickly see whether we have enough margin space.

2.2.2 Customizing axes

We use the function `axis()` to draw the axis. After drawing it with the arguments we just explain, we saw that the labelling was a bit too large. Many plotting functions have the argument cex, which means character extension. `axis()` does not. In such a case, or if we want to change the character extension for all following commands, we can change the "global" default cex-value of 1:

```
par(cex=0.9)
```

For the x-axis, we write:

```
axis(1,c(-10:0,0.5,1,1.5:11.5), labels=NA, tck=-0.02)
```

The first argument, which we set at 1, defines which axis we want to draw; 1=below, 2=left, 3=top, 4=right. Then we have to tell R where we want to have axis ticks and potentially labels. We want to have the numbers from -10 to 0, abbreviated by -10:0, then 0.5 and 1 where the dusk periods change, and then the values 1.5, 2.5 ... 11.5. The next argument takes the labels we want to show at the tick marks. It has to be of equal length as the previous argument, or NA. We choose NA because we don't want to see labels at the ticks. tck=-0.02 reduces the length of the tick marks a bit (see `?par` for what the value means, and try different values; with tck=F, no ticks are drawn). We now have our x-axis with the tick marks where we want them, and we can add the labels with a second axis-command:

```
axis(1,c(seq(-9.5,-1.5,by=2),seq(3,11,by=2)),
     labels=paste(seq(-10,10,by=2)[-6],"h",sep=""),tick=F,line=-0.8)
```

Again the first argument is 1, i.e. the x-axis. Then we provide the x-values where we want to have our labels (they are centred at this values). We only want to have every second hour labelled, so we need a sequence of the values from -9.5 (the middle between -10 and -9) with steps of 2. `seq(-9.5,-1.5,by=2)` does that up to the value -1.5. We omit the 0, we add "sunset" afterwards (we could also do it here, actually), but we need the x-values from 3 to 11 with step=2. Now the labels - have a look at the plot: We want a sequence from -10 to 10, step=2, but not the 0, and add "h" to these numbers. `paste()` pastes numbers and/or characters together. The seq-function produces the sequence from -10 to 10 with step 0, including the 0. By adding [-6] after the brackets we omit the 6th value of the sequence with is 0. The numbers are to be pasted to "h". We don't have to repeat the "h" - `paste()` simply recycles values if they are unequally long. By default, R places a space between pasted characters, which we can omit with the argument sep="" (sep=";" would use a semicolon, or you can place anything you want between the quotation marks). This time we want no ticks, thus tick=F. And the default position is a bit too low, line=-0.8 suits better (trial and error!). Finally, we add "sunset" at the right point. Here we use the function `text()` to illustrate it. Usually, we use `text()` to write text at a given position in the plotting region, but we can also use it outside.

```
text(0,4.57,"sunset",adj=c(0.5,1), xpd=NA)
```

0 is the x-value. 4.57 is the value on the y-axis (remember, it starts at log(100)=4.605), found by trial and error. Then we provide the text, i.e. "sunset". the adj-argument (adjust) defines where the text should appear relative to the x-y-coordinates we have provided and it needs a two-number vector (i.e. two numbers combined with `c()`). The first number shifts the text from 0="to the right of the xy-coordinate" to 1="to the left of the xy-coordinate"; 0.5 centers the text. The second value shifts the text from 0="above the xy-coordinate" to 1="below the

xy-coordinate". Values outside [0;1] are usually allowed, too. Usually, text (as well as e.g. lines, rectangles etc) are clipped where they leave the plotting area. Since our text is outside the plotting area, it would be clipped, i.e. not appear at all. xpd=NA overrides this setting, allowing plotting all the way to the end of the plotting window.

Now the y-axis:

```
t.y <- c(100,110,120,130,150,180,200,250)
axis(2,log(t.y),labels=t.y,las=1)
```

We save the y-values at which we want to have labels in t.y. Then we plot the y-axis, thus with the first argument=2, the ticks are to be at the log-values of t.y but the labels are t.y themselves. By default labels are written in the direction of the axis. Because we want the y-labels to be horizontal, too, we set the argument las=1.

We don't need a axis title for the x-axis since it is self-explanatory. We want, though, a y-axis title. We could use `text()` again, with the argument srt=90 (string rotation 90°), but there's an easier way with the margin text function:

```
mtext("brain activity", 2, line=2.7)
```

Write "brain activity" in the second margin (i.e. the left margin). line=2.7 defines the distance to the y-axis. You could use the ylab argument in the plot-command above, but if you are not happy with the distance of the axis title to the axis line, using mtext is one option.

2.2.3 Colors and background elements

We want four background colours indicating the dusk periods and night in increasingly dark blues (if you have a b/w-copy of this script, you only see greys, of course). We want to save the four colours in t.col. We could search all colours available as characters (type `colours()` in the console! colors or colours can be used). On the web you find tables with all these. You can also use `rgb()` and other colour-functions (see `?rgb` and other help files indicated in the section "See also"), allowing transparency, too (possibly device dependent).
Here we used a nice function that allows you to define your own gradual colours:

```
t.col <- colorRampPalette(c("lightblue","blue"))(4)
```

`colorRampPalette()` with one argument of a vector of two or more colours produces a function, and this function has one argument only, namely how many colours should be found that span between the colours indicated. The (4) thus makes that 4 colours are required, and we want them to gradually change from "lightblue" to "blue".
The function `gray()` produces gray scales. For a b/w-version, we could use

```
t.col <- gray(c(0.9,0.8,0.7,0.6))
```

The background colours in the plot are, in fact, rectangles:

```
rect(c(0,0.5,1,1.5),log(100),c(0.5,1,1.5,12.5),log(250), border=NA,
     col=t.col)
```

The function `rect()` first needs four arguments providing the x-value of the left edge of the rectangle, then the y-value of the bottom, then the x-value of the right edge, then the y-value

of the upper edge. Since we want to draw four rectangles, we provide four values for each side, unless it is always the same value (no need to provide the log(100) four times, it is recycled as many times as needed). We could also use a smaller/larger y-value for the bottom/top of the rectangle because it is clipped to the plotting region (unless we add xpd=NA, as we did above with the text "sunset" in the margin). We don't want borders (default is border="black"). col gets the colours for the rectangles, we have prepared these above.
To draw more complicated background shapes you could use the function `polygon()`.
Finally we send `box()` to the console to draw a box around the plotting area, which also closes the gaps between the axes (actually, there are now two lines where the axes are).

2.2.4 The actual data

Everything is fine and ready: we can plot our data. We first draw the error bar lines and then the points on top of it. Note that we have prepared the graphic with the background shades first so that our actual data points now lie above these background shades. The order you plot elements in a graphic is the order you see these elements, staked on top of each other, so to say.

```
t.x <- c(seq(-9.5,-0.5),0.25,0.75,1.25,2,seq(3,11))
segments(t.x, log(ba$se_l), t.x, log(ba$se_u))
points(t.x, log(ba$pred), pch=21, bg="white")
```

We store the x-values at which we want to have our data in t.x, using the same functions we did before when plotting the x-axis. The error lines are best drawn using `segments()` that takes the four arguments x-value, y-value of starting coordinate, x-value, y-value of end coordinate. Further arguments could be used to change line width (lwd), line type (lty), colour (col), and possibly xpd=NA if you want to draw outside the plotting area (the xpd-argument is not explained in `?segments` - in such a case, always look for info in `?par`). To draw a continuous line between a number of points, use `lines()`.
Finally, we plot points at the right xy-coordinates. pch is point character - you see the first 25 when you type

```
plot(0:25,0:25, pch=0:25)
```

Types 21 to 25 are symbols who's fill colour can be specified with the argument bg (background colour).

Of course it takes some time to get a routine drawing custom plots in R. Once you have the code you can adjust it easily, you can draw almost everything you want, and you can also automatically draw individual plots in a loop (e.g., for a number of species) and save each plot into a file. How to save a plot is explained next.

2.3 Exporting graphics

You can left-click on the graphics window and using the drop-down menu, copy the graphic to the clipboard or save it to a file; try copy-paste, too (possibly device dependent). There are more "save" options in the menu bar at the top. Possibly you don't find the format you need (e.g. on Mac there's only the pdf-option), or you want to automatically save graphics

generated in a loop. For this, rather than `windows()` or `quartz()`, you use one of the following graphics devices: `pdf()`, `postscript()`, `tiff()`, `jpeg()`,etc . Use `?Devices` to find out more. Independent of the device used, you need to specify the file path and the name of the file in which you want to save your graphic. In order to signal R that you are finished with the graphic and that it should safe it, you write `dev.off()`, which means device off. The general pattern, thus, is (for a pdf-example):

```
pdf("filepath/filename.pdf", further arguments if desired)
all the code needed to create the plot
dev.off()      # no argument needed
```

If you develop your graphics using windows() or quartz(),you see step by step what you draw (and you will be able to adjust details until you are satisfied). Once you are happy with the graph, you can pack it into such a device - code - dev.off - frame.
However, using the functions pdf() or postscript() saves your graphic directly to a file (which can also be viewed directly, e.g., with GSview, see http://www.ghostscript.com/GSview.html). These functions also allow you to specify the height and width of the graphic. For LaTeX users (and all others who like it), this is the preferable option because you will bind the graphic file directly into your output file, and by changing the graphic file, the final output (e.g., your manuscript) updates the Figures automatically after running latex.

2.4 Some more options

2.4.1 More custom plots and log-axes

We have seen `boxplot()` and `hist()` above. If you want to customize a histogram and you are not served well enough with the available arguments in `hist()`, you can draw rectangles "by hand" on a blank plot similar to the way we did above. You can write

```
data <- rnorm(1000)    # creates 1000 normally-distributed random numbers
t.h <- hist(data, plot=F)
```

to get valuable information saved in t.h, e.g. break points; have a look at `t.h` or `str(t.h)`.

```
plot(t.h$mids, t.h$density, type="n", xlim=range(t.h$breaks), las=1,
     xlab="", ylab="density")  # prepares the drawing region
n.b <- length(t.h$breaks)      # the number of breaks there are
rect(t.h$breaks[1:(n.b-1)],rep(0,(n.b-1)),t.h$breaks[2:n.b],t.h$density,
     col=1:n.b)
```

What a colourful graph! Let's add lines that enclose 95% of the values symmetrically between them, i.e. since we have a normal distribution with mean=0 and standard deviation=1 this is at x=-1.96 and x=1.96. We want a dotted line type (lty=3), with line width 2 (lwd=2), and in gray:

```
abline(v=c(-1.96,1.96), lty=3, col="gray", lwd=2)
```

If you don't want the histogram to hover above the x-axis, you need to specify `yaxs="i"` and the ylim-argument so that the largest bar does not touch the top border of the graphics box, e.g. `ylim=c(0,max(t.h$density)*1.04)`. Or, adjust the graphics box using `par(bty="L")` for box type = L-shaped (left and bottom, only), a command that you have to execute before the plot command. In the plot command, you still need `yaxs="i"`, but not the ylim-argument.

Other custom plots we regularly use are **`pairs()`**, which draws a pairs-plot, i.e. one plot for each pair of the variables provided. The variables are given as the first argument either as matrix or data frame. This is a helpful plot to inspect correlations among variables at the start of a modelling process.

```
dat <- read.table("http://www.oikostat.ch/data/parusater.txt", header=TRUE)
head(dat)   # look at the data head
pairs(dat[,c("age","sex","P8","wing")])
```

From this we see the strong correlation between P8 (the 8. primary feather of the bird wing) and wing. Males (sex=1) have longer wings than females (sex=2), non-determined birds (sex=0) are inbetween (best seen in the 2. panel in the bottom line). Age and sex have many data points falling on top of each other. Try:

```
plot(jitter(dat$age),jitter(dat$sex))
```

`jitter` jitters the data points a little bit so that they are not overlaying any more. Type `?jitter` to see what control options you have and what exactly `jitter` does.

`acf()` plots an autocorrelation plot. To inspect whether your residuals of the model are autocorrelated, you can use `acf(resid(model), type="partial")` ("model" is the model-object of interest; the data need to be in chronological order!); with type="partial" partial autocorrelations are drawn.

In the worked example above we created a log-axis ourselves. A more automatic alternative is to use `plot(x.values, y.values, log="x")` for a log-x-axis, or log="y" for a log-y-axis, or log="xy" for a log-log-plot.

2.4.2 Getting values from the graphic

If you want to place text or something else at a specific point on the plot you can do that by trial and error. A helpful function is `locator()`. This sent to or typed in the console, you can change to the plotting window and left-click in the plot as many times as you want. Then you right-click to finish and the xy-coordinates of the point or points are written in the console. `locator(1)` will stop locating after 1 click.

```
plot(1:10, 1:10, pch=16)
t.l <- locator(4)    # click four times into the plot, and the four
                     # coordinates are saved in t.l
t.l
```

If you have a scatterplot and you want to know the identity of a specific point (e.g. an outlier), you can use the function `identify()`; you have to give the x- and y-axis coordinates as the first two arguments, i.e. often the same values you provide to the plot-function. See `?identify` for details.

```
dat <- data.frame(x= c(1:10,2), y=c(1:10,9))
plot(dat$x, dat$y, pch=16)
identify(c(1:10,2), c(1:10,9))    # click on the outlier to find out its
                                  # row-number in the dataframe
```

If for some reason you need to know the actual values of whatever parameter set by par, e.g. the margin sizes in your plot, you can type `par("mar")`. See `?par` for all the settings that can be set or asked. `par("mar")[2]`, of course, returns the margin width of the second = left margin only.

2.4.3 Overlaying graphs; figure within a figure

Sometimes one wants to have two plots overlaying one another. For this:

```
dat <- read.table("http://www.oikostat.ch/data/parusater.txt", header=TRUE)
par(mar=c(4,4.5,0.5,4.5))  # margin size on the bottom-left-top-right;
                # we do it here especially to get more space on the right
                # side; par("mar") shows you the current values.
plot(dat$wing,dat$P8,las=1,pch=16,xlab="wing",ylab="P8 (black)")
par(new=T)
```

For the second plot we usually have the argument axes=F, and then, possibly, `axis(4, ..)` to draw an axis on the right side pertaining to the info provided in the second plot:

```
plot(dat$wing,dat$Gew,axes=F, xlab="",ylab="",pch=10, col="orange")
axis(4, las=1)
mtext("weight (orange)", 4, line=3)
```

Sometimes one wants to draw a second figure inside an existing figure (e.g. an inlet figure). Using dat from just above, we want a scatterplot P8 againts wing, and within the figure a smaller histogram of wing:

```
plot(dat$wing,dat$P8,las=1,pch=16,xlab="wing",ylab="P8")
par(fig=c(0.6,1,0.1,0.55), new=T)
hist(dat$wing, main="", xlab="", ylab="", las=1, cex.axis=0.8, col="gray")
```

The fig-argument in par() defines what area of the figure region should be used for plotting; the arguments are left, right, lower edge, upper edge, and the unit is "portion of the figure region" (reasonable values, thus, are between 0 and 1). In our example, the inlet figure is placed close to the bottom right of the first plot. Possibly, you also want to set mar=c(0,0,0,0) in the par-function if your inlet figure needs no axes.

You can also set two plots next to one another:

```
par(fig=c(0,0.5,0,1))      # use the left half of the figure region only
```
plot first plot

```
par(fig=c(0.5,1,0.1), new=T)  # use the right half of the figure region
```
plot second plot

Of course, any other layout, possibly with more figures, can be used. If you want to plot several plots of equal size, however, it is probably easier to use the mfrow-argument explained next.

2.4.4 More than one graph

If more than one plot needs to be plotted in one plotting window, e.g. 4 plots, use:

```
par(mfrow=c(2,2))
plot(1,1,pch="A")
```

```
plot(1,1,pch="B")
plot(1,1,pch="C")
plot(1,1,pch="D")
```

In order to have all four next to one another:

```
windows(8,3)    # for Mac: quartz(,8,3) - try also without this line!
par(mfrow=c(1,4))
plot(1,1,pch="A")
plot(1,1,pch="B")
plot(1,1,pch="C")
plot(1,1,pch="D")
```

Thus, the first of the two values required by mfrow is the number of rows of plots, the second is the number of columns of plots. Sequential plots are filled in row-wise; mfcol=c(2,2) fills them in column-wise.

If you need legends only on the leftmost and/or bottom plots, because it is the same for all plots in a row/column, you may want to reduce the white space between the plots but have enough space to the right and at the bottom of the group of plots:

```
par(mfrow=c(2,2),mar=c(1,1,1,1), oma=c(4,4,0,0))
```

mar is the margin around each plot (use mar=c(0,0,0,0) for adjacent plots), oma is the outer margin around the set of plots (again bottom-left-upper-right; default is c(0,0,0,0)).
You could continue as follows:

```
par(las=1)  # draw all axis labels horizontal
plot(1:10,1:10,xaxt="n")    # no x-axis in the first = upper left plot
plot(1:10,1:10,xaxt="n",yaxt="n")
            # neither x- nor y-axis in the second = upper right plot
plot(1:10,1:10)             # both axes in the bottom left plot
plot(1:10,1:10,yaxt="n")    # no y-axis in the bottom right plot
mtext("x-axis", 1, outer=T, line=2)  # x-axis title in the outer margin
par(las=0)  # otherwise the following axis title is horizontal
mtext("y-axis", 2, outer=T, line=2)  # y-axis title in the outer margin
```

The result:

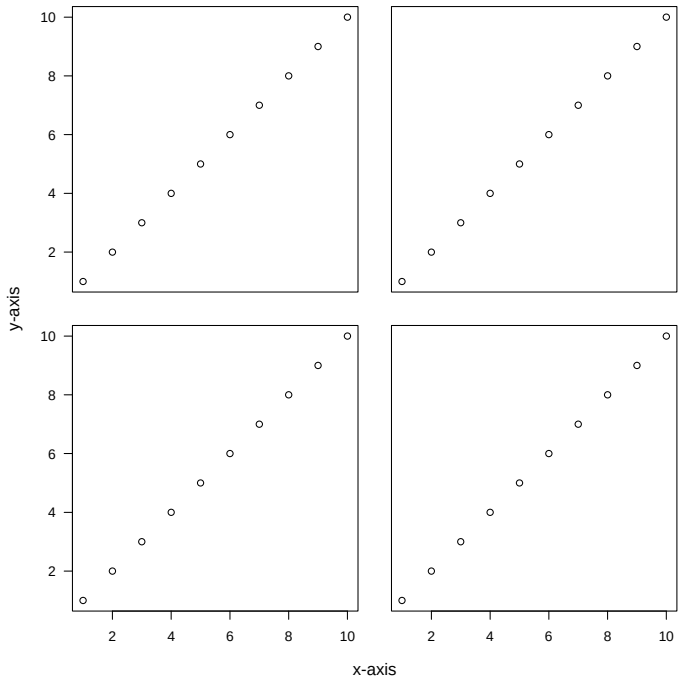

Figure: One graphic composed of four panels.

2.4.5 Symbols and fonts and pixel images

Male (mars) and female (venus) symbols are sometimes needed in our plots:

```
text(x,y, "\\MA", vfont=c("sans serif", "bold"))
text(x,y, "\\VE", vfont=c("sans serif", "bold"))
```

x and y are the coordinates where the symbol should appear. Possibly you have to make a single back slash, i.e.: "\MA" and "\VE" (device dependent).

More symbols can be explored as follows:

```
plot(1:20,1:20,type="n")
i <- 1
for(x in 1:20) for(y in 1:20) { points(x,y,pch=i); i <- i+1 }
```

You'll get more or less symbols (and correspondingly less or more warnings) depending on your device - not all allow for the same number of point characters, apparently.

The font-topic is a difficult one. It is simple to get bold, italic or bold-italic text:

```
par(font=2)  # 1=plain, 2=bold, 3=italic, 4=bold italic
```

Further options are the arguments vfont in `text()` and family in `par()`, but we are not expert in this, and it seems to be device dependent, too. Despite good advice from the web we sometimes don't manage to get the desired font. Let's hope that not too many journals start to require specific fonts for graphics... Of course, if you have Illustrator (or a similar program) you can produce e.g. a postscript file (see above) and change fonts in Illustrator.

It is also possible to place a pixel image in your graph, e.g. as a background picture. We don't use that often, so we don't have much experience, especially regarding file size. Murrell (2006) Murrell (2006) suggests using the package pixmap and then the function addlogo(), and to work with bitmap images.

2.5 Specific graphics packages

There are two major graphics packages that we want to mention here: The package `lattice` brings the proven design of Trellis graphics (developed for S by William S. Cleveland and colleagues at Bell Labs) to R. With lattice, it is very easy to draw nice graphs with a few lines of code. Basically, you can customize them too, but this is a bit harder than with the basic plot function. There is a whole book about lattice graphics (Sarkar, 2008). We just make one simple example, the result of which is shown below:

```
library(lattice)
library(ISwR)
data(bp.obese)
bp.obese$SEX <- factor(bp.obese$sex, labels = c("male", "female"))
xyplot(obese ~ bp | SEX, data = bp.obese, ylab = "Obesity index", xlab =
"Systolic blood pressure (mm Hg)")
```

The package `ggplot2` is newer than lattice and works quite differently. It is based on the insights from Leland Wilkinson's Grammar of Graphics and written by Hadley Wickham. Using ggplot2, it is easier to superpose multiple layers of a graph (points, lines, maps, etc.). Read Wickham (2009) to learn more (http://had.co.nz/ggplot2).

A further recommendable book on graphics is Murrell (2006) Murrell (2006).

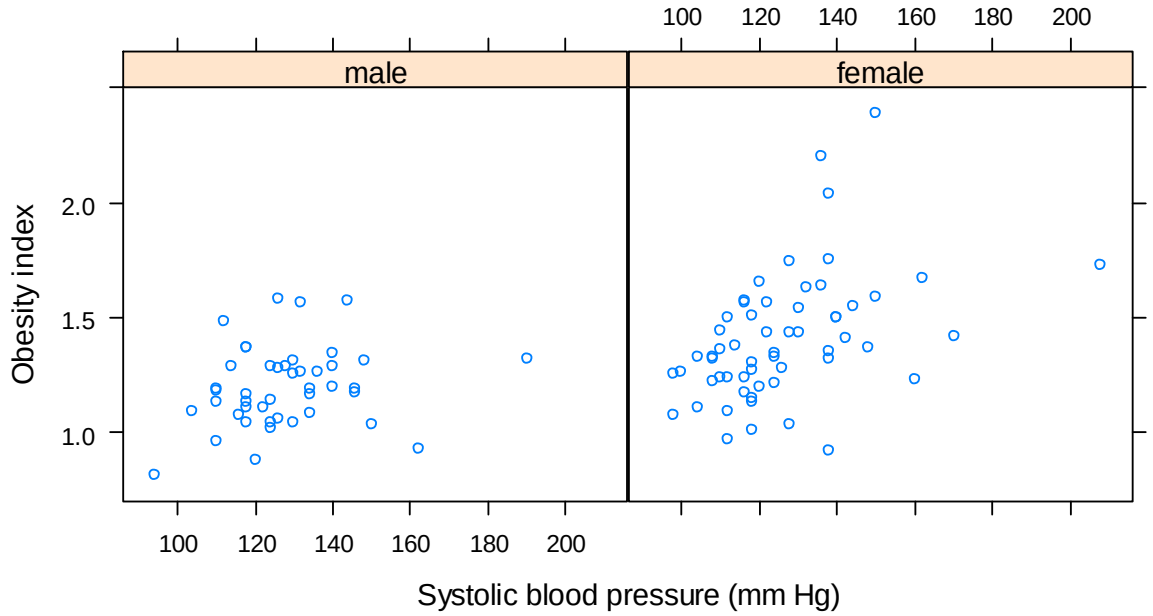

Figure: Scatterplot of obesity index vs. systolic blood pressure for men and women.

2.6 Literature

Murrell, P. (2006). R Graphics. Boca Raton, Chapman&Hall/CRC.
Sarkar, D. (2008) Lattice - Multivariate data visualization with R. Springer.
Wickham, H. (2009). ggplot2. Heidelberg, Springer.

3 Probability distributions

Before diving into statistical analyses and tests, we want to focus on probability distributions. Daalgard (2008, p. 55) says: "the view of data as something coming from a statistical distribution is vital to understanding statistical methods". On the one hand, we can hardly ever study the whole "population" (group of organisms or other entities) of interest. Instead, we investigate a sample. To draw valid conclusions from the sample about the population of interest, it is necessary to make reasonable assumptions about the type of probability distribution that best describes our data.
In this chapter, we will focus on the three probability distributions we find most important for analyzing ecological / biological data: the normal distribution, the Poisson distribution and the binomial distribution. R has convenient functions to simulate data from and handle many different theoretical distributions. Without going into much theoretical detail, we will play with these functions.

3.1 The binomial distribution

The binomial distribution describes repetitions of a binary experiment with „success" or „failure" as outcomes (1 or 0). Examples you may well know from probability theory are flipping a coin or throwing dice. For example, you may flip a coin 10 times and count the number of heads (successes). The number of repetitions, n (here n=10), and the probability of success, p (here p=0.5), sufficiently describe the binomial distribution, abbreviated as B(n,p). We can write X ~B(n,p) to describe a binomially distributed random variable X.
In our example, we have B(n=10, p=0.5). If you know the parameters n and p, the Binomial distribution function, f(x) = P(X=x) describes the point probabilities of the outcome x (for example, 3 heads out of 10):

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

$$\binom{n}{x} = \frac{n!}{x!(n-x)!}$$

The cumulative distribution function, f(x) = P(X≤x), describes the probability of a range of outcomes, for example, less than 4 heads out of 10.
Biological examples, which may be well described by the binomial distribution are:

- the number of young birds from a clutch, which are still alive two weeks after hatching (e.g., 2 out of 4, 3 out of 4…)
- the number of seedlings germinated after 1 week from a petri dish with 100 seeds (e.g., 52, 94, 31...).

A special case of the binomial distribution, when n=1, is the Bernoulli distribution. So Bern(p) is equivalent to B(1,p). In a Bernoulli trial, the outcome is simple "success" or "failure" (0 or 1), without summing up successes. Binary data like presence vs. absence or dead vs. alive may be described by a Bernoulli distribution (or B(1,p)).

Now, let's go to R: we now simulate data from a Bernoulli distribution using the function rbinom() for the example of flipping a coin 10 times. These are 10 independent Bernoulli trials, the outcome will follow B(1,p) or Bern(p)

```
X1 <- rbinom(n = 10, size = 1, prob = 0.5)     # one possible outcome
X1                                             # look at X1
```

Note that the argument size = 1 witin rbinom() is actually the Binomial parameter n in B(n,p), whereas the argument n = 10 in rbinom() is the length of our outcome vector. The function

rbinom uses the name “n” for the number of random values simulated (sample size). Thus, X1 is Bernoulli distributed with p=0.5. Next, we let everyone in this class flip a coin 10 times and count the number of heads. So we repeat the above experiment several times, each time summing up the heads.

```
n.class <- 15     # adapt this for your class
X2 <- rbinom(n = n.class, size = 10, prob = 0.5); X2
# creates and prints X2
```

Now the size of our trial is 10, as we simulate data from B(10,p), and n in rbinom() is now the number of people in class.

We can get point probabilities from the Binomial distribution for specific outcomes using the function dbinom(). For example: the probability of getting exactly 7 heads when flipping a coin ten times is:

```
dbinom(x = 7, size = 10, prob = 0.5)
```

Alternatively, you could do this calculation “by hand”, using the distribution function for the binomial distribution given above. The function choose(n, x) gives the number of possibilities for choosing x out of n:

```
choose(10,7)*0.5^7*0.5^3      # gives the same result as above
```

The probability of getting more than 7 heads (8, 9, or 10 heads) when flipping a coin ten times is:

```
pbinom(q = 7, size = 10, prob = 0.5, lower.tail = F)
# lower.tail = F: P(X > 7)
```

Conversely, the probability of getting <= 7 heads (up to 7 heads) when flipping a coin ten times is:

```
pbinom(q = 7, size = 10, prob = 0.5, lower.tail = T)
# lower.tail = T: P(X <= 7)
```

This is exactly 1-P(X > 7) as calculated before.
There is a fourth function, the quantile function qbinom() that can be used for the binomial distribution. We will apply the quantile function for the normal distribution later. In fact, these four functions are available for all distributions implemented in R.

A good way to graphically inspect the distribution of values in a sample is to draw a histogram:

```
hist(X2)
# create a large binomial sample
X3 <- rbinom(n =1000, size = 10, prob = 1/6)
par(mfrow = c(1,2))
hist(X3, breaks=seq(0,10), right=FALSE)
```

We add a graph of the theoretical point probabilities (see Figure below).

```
plot(x = seq(0,10), y = dbinom(x = seq(0,10),size = 10, prob = 1/6),
main="Point probabilities of B(10,1/6) ", xlab = "x", ylab = "P(X=x)")
```

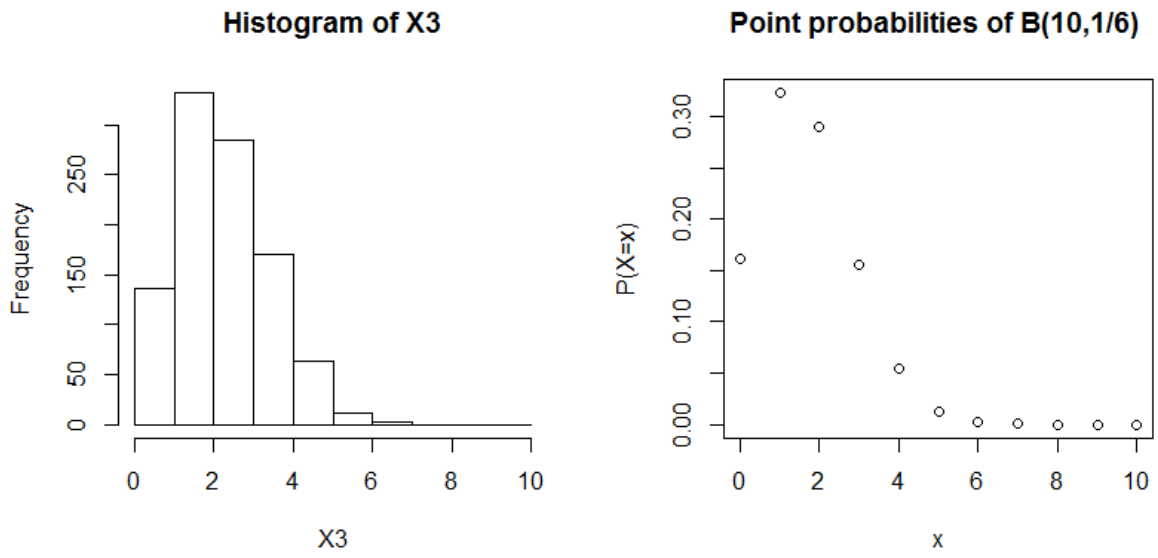

Figure: Histogram of a binomial sample of size 1000 (left) and point probabilities of the underlying distribution B(n=10, p=1/6) (right).

Note that while the histogram does show the general shape of the Binomial distribution B(10,1/6), it does not emphasize the fact that the distribution is discrete, i.e., there are no values between 0 and 1 or between 1 and 2 (etc.), the values are exactly 0, 1, 2 etc. The function hist() is great due to its general use, but in the Binomial case a more precise version would be:

```
plot(table(X3), xlim=c(0,10), ylab="Frequency"); axis(1, 0:10)
```

3.2 The Poisson distribution

The Poisson distribution can be used to describe countable phenomena that happen with constant probability in time or space. The Poisson distribution is thus used to describe count data related to a certain time period (e.g., 1 hour), or a given area (e.g., 1m2). Unlike the binomial distribution, the maximum count is not strictly limited or it is unkown, so we cannot calculate proportions. The Poisson distribution is defined by a single parameter λ, which is at the same time the mean and the variance of the distribution:

$$P(X = x) = \frac{\lambda^x e^{-\lambda}}{x!}$$

Biological data that may be described using a Poisson distribution are:

- the number of seedlings emerged per m2
- the number of birds travelling across an alpine pass per day
- the number of mutations in a given stretch of DNA after a certain amount of radiation

Assume that we know from the literature that the average clutch size for Blue tits (*Parus caeruleus*) is 5. Let's simulate the clutch size of 20 pairs

```
rpois(lambda = 5, n = 20)
```

Now we want to know the probability of a clutch size of exactly 7:

```
dpois(x = 7, lambda = 5)
```

and the probability of a clutch size larger than 7:

```
ppois(q = 7, lambda = 5, lower.tail = FALSE)
```

Note: you can also get the point probabilities and cumulative probabilities for several possible outcomes:

```
dpois(x = c(0:15), lambda = 5)
ppois(q = c(0:15), lambda = 5, lower.tail = TRUE)
```

Let's graphically inspect the point probabilities of different Poisson distributions:

```
par(mfrow=c(1,1))
# lambda = 1
plot(x = seq(0,20), y = dpois(x = seq(0,20), lambda = 1), pch=16, main =
"Point probabilities of the Poisson distribution", xlab = "x", ylab =
"P(X=x)")
# add points for lambda = 0.5, 3, 5 and 10
points(x = seq(0,20), y = dpois(x = seq(0,20), lambda = 0.5), pch=16,
col="red")
points(x = seq(0,20), y = dpois(x = seq(0,20), lambda = 3), pch=16,
col="blue")
points(x = seq(0,20), y = dpois(x = seq(0,20), lambda = 5), pch=16,
col="green")
points(x = seq(0,20), y = dpois(x = seq(0,20), lambda = 10), pch=16,
col="orange")
legend(x=15, y = 0.35, legend=
c("lambda=1","lambda=0.5","lambda=3","lambda=5","lambda=10"), pch=16, col=
c("black","red","blue","green","orange"))
```

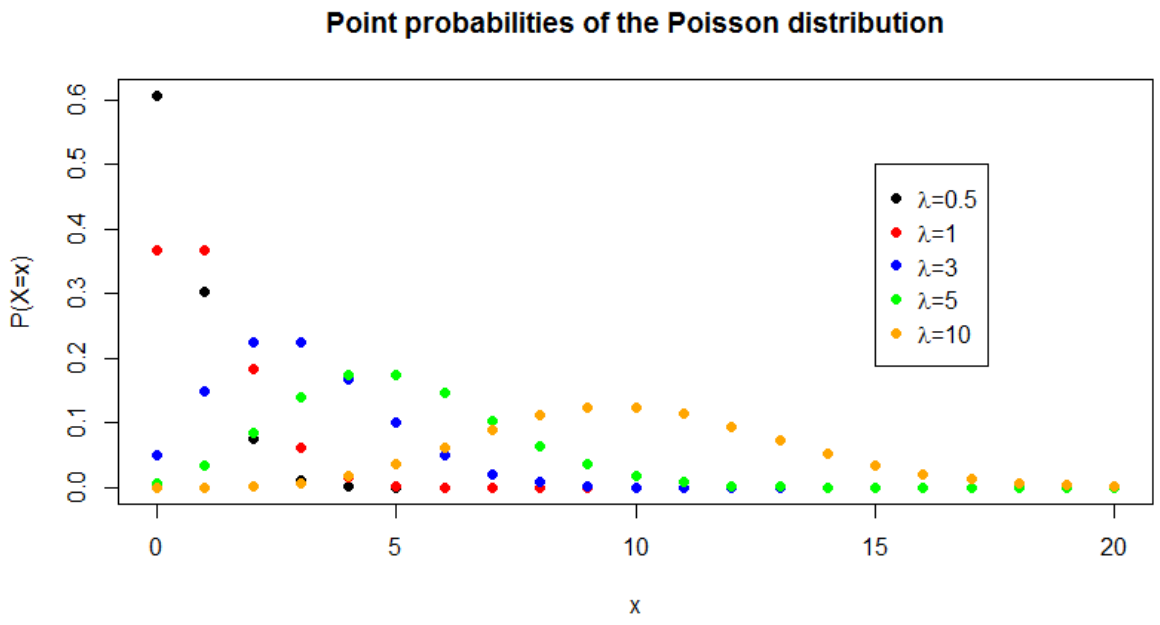

Figure: Point probabilities of Poisson distributions with different λ.

3.3 Discrete and continuous distributions

The Poisson and the Binomial distribution are so called *discrete distributions*. This means that the outcome is a discrete number. It can be any integer number for the Poisson (0, 1, 2, 3,...), an integer between zero and n (upper limit) for the Binomial (0, 1, 2, 3,..,n), or a binary variable (0,1), as in case of the Bernoulli distribution to describe either of two possible outcome categories (presence vs. absence or dead vs. alive). For discrete probabilities, the function $f(X) = P(X = x)$ describes the *point probabilities*.
The normal distribution, which we look at next, is a *continuous distribution*. This means that random values from the normal distribution can lie everywhere on a continuous scale and thus there is an infinite number of possible values. In practice, every measurement on a continuous scale has a limited number of digits and is thus discrete on a very small scale. For example, a measure taken in meters with three digits after the comma (e.g. 2.755 m) is an integer number when we express it as millimeters (2755 mm). Nevertheless we model such data as continuous data because that is what they are in nature. The distribution function for continuous distributions describes densities instead of point probabilities. Densities are a bit hard to interpret, but it helps to remember that the total area under the density curve is always 1. More technically, a density is the infinitesimal probability of hitting a small region around x divided by the size of that region. There are many other continuous distributions apart from the normal distribution, several of which are important in statistics (t, F, χ^2, Uniform, Gamma, Weibull, etc.)

3.4 The normal distribution

The normal distribution, also known as Gaussian distribution is useful to describe, at least approximately, any (continuous) variable that tends to cluster around the mean. As an example, the heights of men in your country cluster around a mean of let's say 180 cm (depending on the country), with only few men less than 160 cm or more than 200 cm tall. Although we would probably measure the height in cm, there is an infinite number of possible heights (even within a 1 cm interval). The density function of the normal distribution has a perfectly symmetric, bell-shaped form, defined by the parameters μ for the mean and σ for the standard deviation (σ^2 for the variance). A histogram of a sample of normally distributed measurements will also have a more or less bell-shaped form, depending on the size of the sample.
Other biological examples of data that may be described by a normal distribution are:

- the wing span or beak length of birds (in cm or mm)
- weight of seeds (mg)

Note that these are typically measured variables (instead of counted, as for Poisson or binomially distributed variables), involving some sort of unit (as cm or g, etc.).

The normal distribution can also be used to approximate other distributions. This works well for example for a binomial distribution B(n,p) with large n and p far enough from zero or one. The approximation is then $N(\mu=np, \sigma^2=np(1-p))$. As a rule of thumb, the approximation works well if $np > 5$ and $n(1-p) > 5$. A Poisson distribution with sufficiently large λ can also be approximated as $N(\mu=l, \sigma^2=l)$. This approximation is excellent for $\lambda > 1000$, and is probably ok for $\lambda > 10$.

Let's go back to the Blue tits example above, for which we simulated the sizes of 20 clutches. Imagine we know that 18 cm is the average wingspan for female Blue tits, with a standard

deviation of 0.5 cm. We simulate the wing spans of 20 female birds using the function rnorm():

```
rnorm(mean = 18, sd = 0.5, n = 20)
```

Note that if mean and sd are not specified, the default is mean = 0 and sd = 1, corresponding to the standard normal distribution. We now have a closer look at the standard normal distribution by plotting the density curve:

```
x <- seq(-4, 4, 0.1)       # x-values used in the plot
plot(x, dnorm(x), type = "l", main = expression(paste("Standard normal
distribution (", sigma," = 1, ", mu, " = 0)")))
```

Note that the area under the density function is exactly 1. Now we want to know the the 97.5 % quantile (value x1 on the x-axis)

```
x1 <- qnorm(0.975); x1                          # x1, where P[X <= x] = 0.975
```

The probability, that any random number X, drawn from the standard normal distribution is <= x1 is 0.975. Conversely, X > x2 is 0.975 for x2.

```
x2 <- qnorm(0.975, lower.tail = FALSE); x2     # x2, where P[X > x] = 0.975
x3 <- qnorm(0.025, lower.tail = TRUE); x3      # alternative way, x3 = x2
```

The probability of a random number > 3, ≤ 1.96 and ≤1 is:

```
pnorm(3, lower.tail = FALSE)          # probability of a value > 3, P[X > 3]
pnorm(1.96)                           # P[X ≤ 1.96]
pnorm(1)                              # P[X ≤ 1]
```

Specifying lower.tail = FALSE is not necessary, because this is the default.

Let's graphically inspect the density curves of different Normal distributions (see Figure below):

```
x <- seq(-6,6,0.1)
par(mfrow = c(1,2))
plot(x, dnorm(x), type = "l", ylim = c(0,0.8), main =
expression(paste("Normal distribution with different ", sigma)), ylab =
"Probability density")
lines(x, dnorm(x, sd = 2),col="red")
lines(x, dnorm(x, sd = 0.5),col="violet")
legend(-6,0.8, legend =
c(expression(paste(sigma,"=1")),expression(paste(sigma,"=2")),
expression(paste(sigma,"=0.5"))), col = c("black","red","violet"), lty=1,
bty="n")
plot(x, dnorm(x), type = "l", ylim = c(0,0.8), main =
expression(paste("Normal distribution with different ", mu)), ylab =
"Probability density")
lines(x, dnorm(x, mean = 2), col="red")
lines(x, dnorm(x, mean = -1), col="violet")
legend(-6,0.8, legend =
c(expression(paste(mu,"=0")),expression(paste(mu,"=2")),
expression(paste(mu,"=-1"))), col = c("black","red","violet"), lty=1,
bty="n")
```

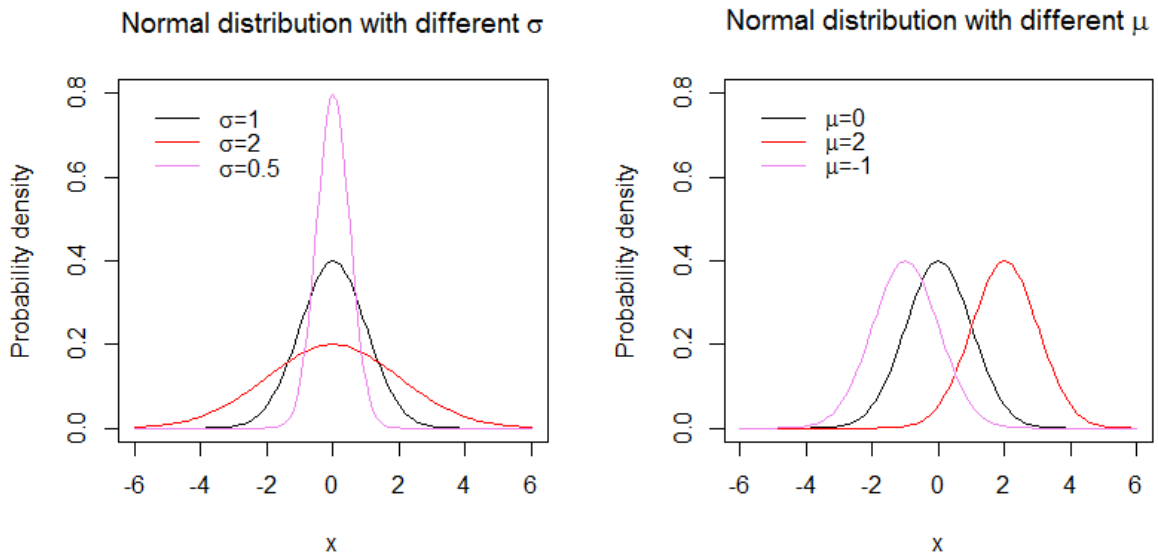

Figure: Left: normal distributions with equal mean (μ = 0) but different standard deviations. Right: normal distributions with equal standard deviation (σ = 1) but different m.

The black curve is the standard normal distribution (with μ=0 and σ=1).

3.4.1 The central limit theorem

In its classical form, the central limit theorem states that if x_1, x_2,..., x_n is a random sample from a population (more technically, the sample should be iid, independent and identically distributed) with mean μ and standard deviation σ, and $\bar{x}$ is the sample average, then the distribution of $\bar{x}$ is approximately normal with mean μ and standard deviation σ^2/n.

Let's try this for the average of a sample from a binomial distribution (example adapted from Crawley 2002, p. 127):

```
# Histogram of 1000 random numbers from a binomial distribution with size =
12 and p = 0.2

par(mfrow=c(1,2))
y <- rbinom(n = 1000, size = 12, p = 0.05)
table(y)
hist(y, breaks = c(0:(max(y)+1)), right = FALSE)

# Histogram of 1000 means of 30 random numbers each from a binomial
distribution with size = 12 and p = 0.2

my <- numeric(1000)
for (i in 1:1000){
    y <-rbinom(n = 1000, size = 12, p = 0.05)
    my[i] <- mean(y)}
hist(my)

# look at the means
mean(y)
mean(my)

# look at the variances
# theoretical variance of a binomial distribution
```

```
var.binom <- 0.05 * (1-0.05) * 12   # sqrt(p * (1-p) * n)
# compare with sample variance of y
var.binom; var(y)
# theoretical variance of sample averages (sample size 30)
sd.mean.30 <- var.binom / 30
# compare with sample standard deviation of my
sd.mean.30; sd(my)
```

A more relaxed formulation says that the sum z of many small independent random variables, z_i, will be a random variable with an approximate normal distribution $z \sim N(\mu_z, \sigma^2_z)$, with μ_z and σ^2_z being the sums of the means μ_{zi} and variances σ^2_{zi} of the z_i's. This works in practice, if the individual σ^2_{zi} are small compared to the total variance σ^2_z. For more details, see for example Gelman and Hill (2007) or Crawley (2002). Here, we focus on the practical implications:

- The distribution of the sample mean of any sample of random variables (also if these are themselves not normally distributed), tends to have a normal distribution. The larger the sample size, the better works this approximation.
- The approximation of the binomial distribution by the normal distribution (see above): since if n is large, the random variable is the sum of many independent random variables (Bernoulli trials).
- Any variable that is the result of a combination of a large number of small effects (like phenotypic characteristics that are determined by many genes) tends to show a bell-shaped density estimate. This justifies the common use of the normal distribution to describe such data.
- For the same reason, the normal distribution can be used to describe error variation (residual variance) in linear models (where this normality of errors is an underlying assumption). In practice, the error is often a result of many unobserved variables.

3.5 Note on the generation of random numbers

Try the following in R and compare the results with the results from your neighbor:

```
rnorm(3, mean=10, sd=4) # draw 3 random numbers from Normal(10, 4)
```

You almost certainly got different values. To make a draw of random numbers reproducible, we need to set starting values (random seed) for the random number generator. Now try:

```
set.seed(3634)
rnorm(3, mean=10, sd=4)
```

Now you should get the same numbers. Note: the random numbers generated in R (and other software) are actually pseudo-random, generated by certain algorithms. Otherwise they could not be reproduced. Setting a seed is very handy whenever you simulate data and want to be able to reproduce them.

3.6 Literature

Dalgaard, P. (2008). Introductory statics with R. New York, Springer.
Gelman A. & Hill J. (2007) Data analysis using regression and multilevel/hierarchical models. Cambridge University Press.
Crawley, M. (2002) Statistical computing – An introduction to data analysis using S-Plus. Wiley.

4 Summary statistics

Summary statistics help us summarize a set of observations (data), in order to communicate as much information as possible in a few numbers. A statistic is a sample property, i.e., it can be calculated from the observations in the sample. In contrast, a parameter is a property of the population from which the sample was taken. As it is usually unknown (unless we simulate data from a known distribution), we use statistics to estimate parameters.
Statistics inform us about the distribution of observed values in the sample. Lots of statistics can easily be calculated in R. Here is an overview of some statistics, given a sample $\{x_1, x_2, x_3, \ldots, x_n\}$, ordered with x_1 being the smallest, x_n being the largest value (ordering is only important for the median):

Statistic	R-function	Formula	Parameter
arithmetic mean	mean()	$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$	population mean (μ)
median	median()	$median = x_{(n+1)/2} \ (for\ uneven\ n)$ $median = \frac{1}{2}\left(x_{n/2} + x_{(n/2)+1}\right) (for\ even\ n)$	population median
sample variance	var()	$s^2 = \frac{1}{n-1}\sum_{i=1}^{n} (\bar{x} - x_i)^2$	population variance (σ^2)
sample standard deviation	sd()	$s = \sqrt{s^2}$	population standard deviation (σ)

4.1 Measures of Location

The most important measure of location (or central tendency) is the arithmetic mean (or average). It is very useful to describe the "center" of symmetric distributions (such as the normal distribution). Its disadvantage is, that it is sensitive to extreme values. The median is an alternative measure of location that is much less sensitive to extreme values. It is the central value of the ordered sample (the formula given in the table only applies if the sample is ordered). If n is even, it is the arithmetic mean of the two most central values.
Let's simulate some data in R and calculate both the mean and the median:

```
x <- rnorm(20, mean = 5, sd = 4)          # x: sample data, n=20
x2 <- c(x, 45)                            # add an extreme value of 45 to x
mean(x);   mean(x2)
```

You see that the arithmetic mean of x is something close to 5, the true mean of the normal distribution we have sampled 20 values from. The mean of x2, the sample x including an extreme value is considerably increased. The median, however, is changed only little by the extreme value.

```
median(x)
median(x2)
z <- c(1,4,7,9,10)   # small sample with uneven n
median(z)
z2 <- c(1,4,7,9)     # small sample with even n
median(z2)
```

For a perfectly symmetric distribution, the median equals the mean.

4.2 Measures of dispersion

Measures of dispersion measure the spread / variability of the data.
The variance of a sample is the sum of the squared deviations from the sample mean over all observations in the sample, divided by (n-1). The variance is hard to interpret, as it is usually quite a large number. The standard deviation, which is the square root of the variance, is easier. It is approximately the average deviation of an observation from the sample mean (it would be exactly the average deviation from the sample mean, if we would use n instead of n-1 in the denominator of the formula for the standard deviation). Try for the sample x created above:

```
var(x)            # sample variance
sd(x)             # sample standard deviation
sqrt(var(x))      # dito
```

4.3 Quantiles and the boxplot

The p-quantile is the value x with the property that there is probability p of getting a value less than or equal to it, i.e. $p(X \leq x) = p$. The median is the 50 % quantile. The 25 % quantile and the 75 % quantile are also called the lower and the upper quartile (quartile because together with the median, they divide the distribution into quarters). The difference between the 25 % and the 75 % quantile is called the inter-quartile range. This range includes 50 % of the distribution and is also used as a measure of dispersion.

We have already seen how to calculate quartiles for the binomial, the Poisson and the normal distribution (functions qbinom(), qpois(), qnorm()). The 25 %, 50 % and the 75 % quantile of the standard normal distribution can be calculated as follows:

```
q25 <- qnorm(0.25); q25
q50 <- qnorm(0.5); q50
q75 <- qnorm(0.75); q75
```

The boxplot is a way to graphically display a distribution, using the median, the quartiles and the inter-quartile range. Let's load the dataset tlc from the library ISwR and create a boxplot of the body sizes of women and men.

```
library(ISwR)
data(tlc); ?tlc
par(mfrow=c(1,1))
boxplot(height ~ sex, data = tlc, names=c("Women","Men"), ylab = "Body
height (cm)", col = "blue")
```

In a boxplot, the box (here in blue) includes the interquartile range. The median is shown as a line within the box. The range defined by the whiskers which extend at the upper and lower end of the box is less clearly defined (software and user dependent). The default in R (see `?boxplot.stats`) is that the whiskers the whiskers extend to the most extreme data point which is no more than 1.5 times the length of the box away from the box (interquartile-ranges). Data points beyond this range are shown separately as "outliers" (see Figure below).

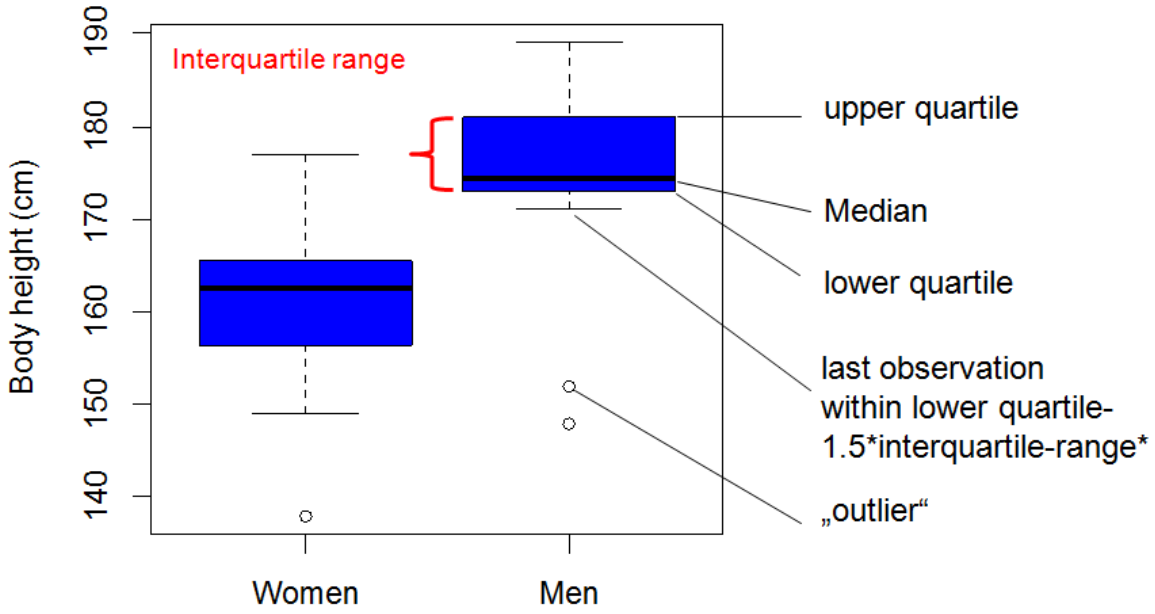

Figure: Comparison of body height of women and men by box plots.

4.4 The standard error of the mean

If we have a sample of n observations, which all come from a normal distribution with mean μ and standard deviation σ, then it is known that the arithmetic mean $\bar{x}$ of the sample is normally distributed around μ with standard deviation $SD_{\bar{x}} = \sigma/\sqrt{n}$.
In practice, however, we do not know σ, but estimate it by the sample standard deviation s. The true standard deviation of the sample mean is also estimated by the "standard error of the mean" (SEM), which is calculated as $SEM = s/\sqrt{n}$. Here is an example.

```
x <- rnorm(n = 100, s = 4, mean = 20)          # a sample of size n=100
sem <- sd(x) / sqrt(length(x)); sem            # calculate SEM
sd.mean <- 4/sqrt(100)                         # theoretical SD.mean
```

While s describes the variability of individual observations (average difference from the sample mean, SEM describes the variability of the sample mean of n random values from a sample with sample standard deviation s.

4.5 Confidence intervals

If (as above) we have a sample of n observations, which all come from a normal distribution with mean μ and standard deviation σ, and we know that the arithmetic mean $\bar{x}$ of the sample is normally distributed around μ with standard deviation $SD_{\bar{x}} = \sigma/\sqrt{n}$ we can calculate a 95 % confidence interval for μ as: $\bar{x} + \sigma/\sqrt{n} \cdot N_{0.025} \leq \mu \leq \bar{x} + \sigma/\sqrt{n} \cdot N_{0.975}$, where $N_{0.025}$ and $N_{0.975}$ are the 2.5 % quantiles of the standard normal distribution. For the example above this would be:

```
x <- rnorm(n = 100, s = 4, mean = 20)          # a sample of size n=100
sample.mean <- mean(x)
sd.mean <- 4/sqrt(100)                         # theoretical SD.mean
ci95.lower <- sample.mean + sd.mean * qnorm(0.025)
ci95.upper <- sample.mean + sd.mean * qnorm(0.975)
```

The 95 % confidence interval specifies a plausible range for μ. If we would estimate a confidence interval for the true parameter μ an infinite number of times, by calculating $\bar{x}$ from an infinite number of samples, each of size n, then μ would lie within the confidence interval in 95 % of times (e.g., about 50 times out of 1000). The meaning of the 95 % confidence interval is sometimes mistaken as containing μ with 95 % probability. This is not the case, since one specific confidence interval either contains μ or it does not.
However, in practice we do not know σ and have to estimate s (and SEM) from the sample. Using the normal distribution then results in a confidence interval that is too narrow. Confidence intervals that are estimated from data are therefore based on the t-distribution (see below) instead of the normal distribution. This is necessary to correct for having estimated σ from the sample, especially if the sample size is small.

4.6 Mean and Variance of different distributions

As we have seen in Chapter 3, the Normal distribution $N(\mu,\sigma^2)$ is defined by the mean and the standard deviation (or the variance, if we take the square of σ). These two parameters can be estimated directly from the arithmetic mean and standard deviation of a sample. If we take a sample large enough (n=10000) from a normal distribution with μ = 5 and σ = 4, this estimate is fairly accurate:

```
normal <- rnorm(n = 10000, mean = 5, sd = 4)  # normal sample of n=10000
mean(normal)              # arithmetic mean of the sample
sd(normal)                # standard deviation of the sample
```

The binomial distribution B(n,p) is also defined by two parameters, but not by the mean and the standard deviation. The mean of a binomial distribution is np, the variance is np(1-p). Let's check that, again with a sample of n=10000.

```
n <- 10000
p <- 1/6
binom <- rbinom(n = n, size = 10, prob = p) # binomial sample
n*p                       # expected value for the mean of B(n,p)
n * p * (1-p)             # expected value for the variance of B(n,p)
mean(binom)               # arithmetic mean
var(binom)                # sample variance
```

The Poisson distribution is defined by only one parameter, l, which is the mean and the variance at the same time.

```
poisson <- rpois(n = 100000, lambda = 5)
mean(poisson)             # expected value: lambda
var(poisson)              # expected value: lambda
```

4.7 Literature

Dalgaard, P. (2008). Introductory statics with R. New York, Springer.

5 Classical statistical tests

5.1 Null-hypothesis testing

Classical statistical methods work with null-hypothesis testing which is based on falsification. From a logical point of view, it is impossible to prove a theory or a derived hypothesis, since this would require making all observations related to the hypothesis, and we usually work with samples. Falsificationism thus strives for questioning, for falsification, of hypotheses instead of proving them.

For a long time, for example, only white swans had been observed in Europe and it was hypothesized that all swans are white. However, the observation of a single black swan in Australia was sufficient to disprove this hypothesis. When doing classical statistical tests, we therefore make a detour. We propose a null-hypothesis (H_0) which is typically uninteresting and try to falsify (reject) it, instead of proving our actual hypothesis. H_0 is complemented by an alternative hypothesis (H_A), which is mutually exclusive with regard to H_0. As a consequence, rejecting H_0 means support for H_A.

For example, we might have measured the metabolic rates of Northern fulmars (Eissturmvogel, *Fulmarus glacialis*) and want to know whether males and females differ in metabolic rate. The corresponding H_0 would be that the metabolic rates of males and females are equal. H_A would be that the metabolic rates differ. Note that H_A is two-sided in this case, which means that we do not specify whether males or females have a higher metabolic rate. To test H_0, we estimate the probability that the observed outcome (e.g., the difference between male and female metabolic rates) or an even more extreme outcome (even larger difference) could have occurred by chance alone (if H_0 were true). This probability is the P-value. If the P-value is $< \alpha$ (see below), we reject H_0. We say, the test is significant.

When testing H_0, four typical decision situations can arise:

		Truth	
		H_0 is true	**H_A is true**
Decision (test result)	**accept H_0**	1-α	type II error (β)
	reject H_0	type I error (α)	1-β

We are fine if we accept H_0 when it is actually true or if we reject H_0 when it is actually false. We can make two types of error: a type I error occurs if we reject H_0 although it is true. The probability of such an error is α. A type II error occurs if we accept H_0 although H_A is true (failure to reject H_0). This error occurs with probability β. α is also known as the significance level, and it is usually set to 5 % (or 0.05). 1- α is called the confidence level (95 % if α is 5 %). 1-β is the statistical power, the probability to reject H_0 when H_A is true (the probability to detect a true difference).

Let's assume H_0 is wrong for the fulmar example, and there is a true difference between the metabolic rates of males and females. Whenever we get a p-value < 0.05 as a result from the statistical test applied (below we will apply a two-sample t test for this example), we conclude that there is a difference between males and females. However, the power for showing this difference is never 100 % because we only look at a sample from a larger population (all fulmars). If we collected a sample of fulmars 1000 times, with a certain sample size n, we would find a significant test result in (1-β) *100 % of the repetitions. The power increases with the sample size n (for a given difference in metabolic rates, standard deviation of metabolic rates and α). In order to guarantee a certain power, a sample size estimation is

necessary. Sample size estimations are required for medical studies where a power of 80 or 90 % is usually anticipated ($\beta = 0.2$ or 0.1). Unfortunately, sample size estimations are rarely performed for biological studies meaning that the power is usually unknown and often quite small (β is large). This means that you might get a non-significant result despite of a large difference (effect size). However, "absence of evidence is not evidence of absence" (Altman & Bland 1995), i.e., a non-significant effect is no proof of H_0 (but may be lack of power to reject it). The take home message here is that statistical significance is only one aspect of an analysis.
A much more important aspect is to estimate the size of the effect of interest. In our example, this would mean to estimate the difference between male and female metabolic rates. We can estimate the size of this difference together with a confidence interval (see above). While providing more information than just the significance test alone (size, direction, and plausible range of the difference), the confidence interval also provides a test of H_0. We conclude that the difference between metabolic rates is significant if the confidence interval does not contain zero, the difference expected under H_0.But, again, if H_0 is included in the confidence interval, this does not mean that we have proven H_0.

5.1.1 Test statistics

As already said earlier, the probability that the observed outcome or a more extreme outcome could have occurred by chance alone (if H_0 were true), is the P-value. The P-value is usually derived via a test-statistic and the distribution of that test statistic. An example of a test statistic is the t statistic and the t-distribution in the case of t tests. Other test statistics are for instance $\chi 2$ ($\chi 2$ test, likelihood-ratio test) or F (F-tests in Analysis of Variance).

5.2 The t test family

There are three types of t tests that we are going to look at, which are used depending on the kind of comparison we want to make. All of them are based on the assumption that the sample data come from a normal distribution. This also means that t tests are for continuous data. Let's start with the one-sample t test.

5.2.1 One-sample t test

The one-sample t test is used to compare one mean (of a sample) to a reference value (an a priori chosen value). Here is an example from Daalgard (2008): We have: a sample of measurements of daily energy intake (in Joules) from 11 women, $x_1, x_2, \ldots, x_{11}$. Now we want to know whether these data conform with the recommended value of 7725 Joules per day.

```
daily.intake <- c(5260,5470,5640,6180,6390,6515,6805,7515,7515,8230,8770)
Mean <- mean(daily.intake); Mean
```

The Null hypothesis, H_0, is: $\mu = \mu_0$, the alternative Hypothesis, H_A: $\mu \neq \mu_0$. Hereby, μ is the true population mean that we estimate by the sample mean $\bar{x}$. μ_0 is the reference value (here the commended energy intake). If the assumption that our sample is normally distributed with mean μ and variance σ^2, i.e. $x_1, x_2, \ldots, x_n \sim N(\mu, \sigma^2)$, is reasonable, we can use a one-sample t test to answer our question. A key concept for this test is the standard error of the mean $SEM = s/\sqrt{n}$ (s is the sample standard deviation, i.e. the sum of residuals divided by n-1) that we have met in the previous chapter. The test statistic of the t test is t (what a surprise), and it is calculated as follows:

$$\frac{\bar{x} - \mu_0}{SEM}$$

t basically measures how many SEM our sample mean differs from the reference value. Remember that for normally distributed data, the probability of staying within $\mu \pm 2\sigma$ is approximately 95 % (corresponding to 5 % outside this range). Under H_0, we would thus expect μ_0 to fall within this range. In small samples (say n < 30), however, it is necessary to correct for the fact that we have estimated SEM from the sample. The distribution of the test statistic t (t-distribution) therefore has slightly increased probabilities for extreme values, i.e., the t-distribution has heavier tails. The shape of the t-distribution depends on the sample size and approximates the normal distribution for n increasing to infinity (see Figure below). To get the correct quantiles to define the acceptance region of H_0 we take them for the t-distribution with n-1 degrees of freedom. If our observed t falls outside the acceptance region for a given significance level, i.e. $[t_{0.025,\ n-1}/t_{0.975,n-1}]$ for α=5 %, we reject H_0 and say that the sample mean differs significantly from μ_0. Equivalently, we would get a p-value < 0.05 for this test, which is the probability of observing a t-value as large or larger than the one observed, given H_0 is true. So, if we observe a t-value that is sufficiently unlikely to be observed if H_0 were true, and we have defined what "sufficiently unlikely" means by setting our significance level, we conclude that H_0 must be wrong (reject H_0).
Note that for large samples ($n \geq 30$), you can use the rule of thumb: $t \geq 2 \rightarrow \mu \neq \mu_0$. Now we do the calculations just described by hand:

```
# One sample t test by hand
Mu0 <- 7725
> SD <- sd(daily.intake); SD
[1] 1142.123
> N <- length(daily.intake); N
[1] 11
> SEM <-SD/sqrt(N); SEM              # standard error of the mean
[1] 344.3631
> Tval <- (Mean-Mu0)/SEM; Tval       # to calculate t manually
[1] -2.820754
> pvalue <- 2*pt(Tval, N-1); pvalue
[1] 0.01813724
```

Doing this test using the t.test() function in R is much quicker:

```
# One-sample t test using t.test()
> t.test(daily.intake, mu = 7725)    # 7725: recommended value

        One Sample t-test

data:  daily.intake
t = -2.8208, df = 10, p-value = 0.01814
alternative hypothesis: true mean is not equal to 7725
95 percent confidence interval:
 5986.348 7520.925
sample estimates:
mean of x
 6753.636
```

We get the same result as we got by hand, but with a more coherent output. We get the data set that was used, the observed t-value, the degrees of freedom and the p-value, the probability of the observed t-value (or a more extreme one) from the t-distribution for . In

addition, R reminds us about our H_A and the formulation "not equal" reminds us of the fact that we have a two-sided H_A. R also provides a 95 % confidence interval for μ, which we could calculate by hand as follows:

$$\bar{x} + t_{0.025,\ n-1} \times SEM < \mu < \bar{x} + t_{0.975,\ n-1} \times SEM$$

This confidence interval does not include the reference value 7725, which means we can conclude that $\mu \neq \mu_0$.

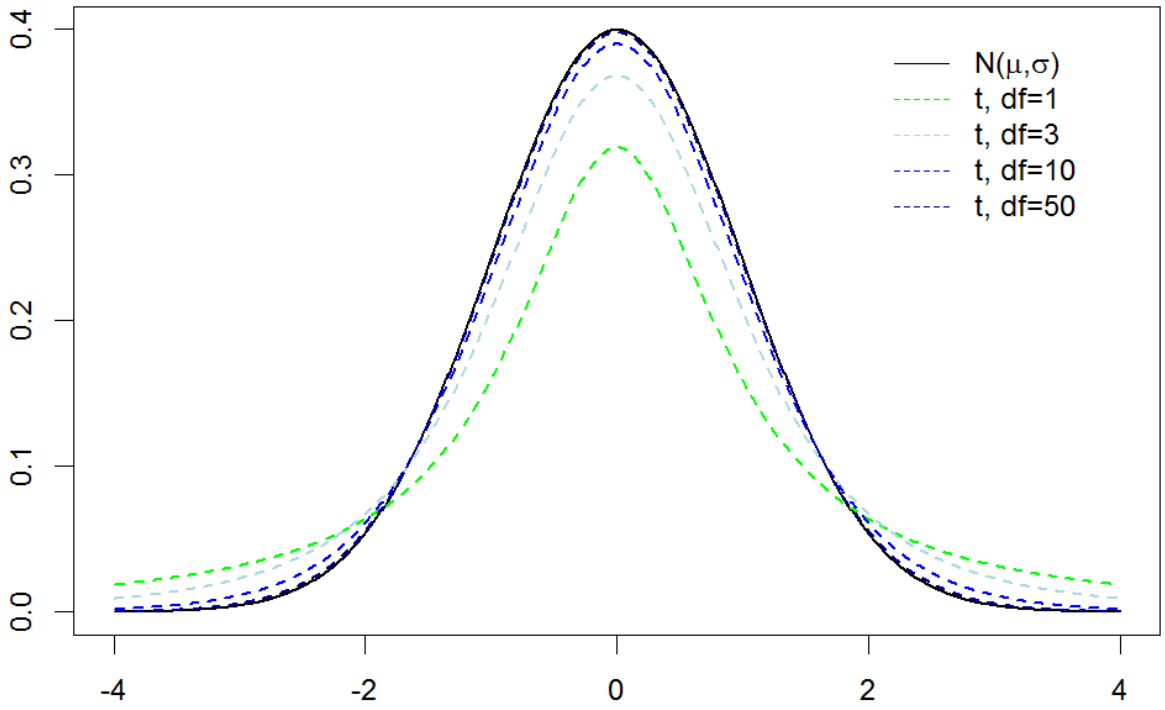

Figure: Comparison of the t-distribution with 1,3, 10 and 50 degrees of freedom with the standard normal distribution. For df=50, the difference is hardly visible.

5.2.2 The two-sample t test

Instead of comparing one mean to a reference value, we might more often want to compare two means (two samples). We could also say we want to test the Null-hypothesis that two samples come from distributions with the same mean. Here is an example from Quinn & Keough (2002): they have measured the metabolic rates of Northern fulmars (Eissturmvogel, *Fulmarus glacialis*) and want to know whether males and females differ in metabolic rate. The corresponding Null hypothesis, H_0, is: $\mu_M = \mu_F$ or equivalently $\mu_M - \mu_F = 0$, and the (two-sided) alternative hypothesis, H_A is,: $\mu_M \neq \mu_F$ or: $\mu_M - \mu_F \neq 0$. We assume that both male and female metabolic rates are normally distributed, i.e., male metabolic rate ~ $N(\mu_M, \sigma_M^2$ and female metabolic rate ~ $N(\mu_F, \sigma_F^2)$. Another important assumption is that the two samples are independent. This means that each measurement of metabolic rate was taken on an independent "unit" (here bird). We will see an example of non-independent samples in the next subchapter.
The two-sample t test works in a similar way as the one-sample t test. The test statistic t is:

$$\frac{\bar{x}_2 - \bar{x}_1}{SEDM}$$

SEDM is the standard error of the difference of means. Under the assumption that σ_M and σ_F are equal, it is calculated as the pooled standard error of the mean (classical t test):

$$SEDM = s_p \sqrt{\frac{1}{n_1} + \frac{1}{n_2}}$$

$$s_p = \sqrt{\frac{(n_1 - 1)s_1^2 + (n_2 - 1)s_2^2}{n_1 + n_2 - 2}}$$

The corresponding t will follow a t-distribution with n_1+n_2-2 degrees of freedom. However, the assumption of equal variances is not always suitable, and R provides by default the Welch test, which does not make this assumption. The distribution of the resulting t can be approximated by a t distribution with degrees of freedom calculated from s_1, s_2, n_1, and n_2 (non-integer df). The Welch test is considered the safer one. But as long as the group sizes and standard deviations do not differ a lot, the two tests will usually give similar results. Now let's look at these variants of the two-sample t test in R. First, you need to read the data set "furness.csv", for example by setting the working directory to your folder "datasets":

```
setwd("..your filepath…/datasets")        # set the working directory
furness <- read.csv("furness.csv")        # read the data
```

It is always a good idea to graphically inspect data before carrying out any test. A boxplot by sex is a good way to do this for the furness data:

```
plot(METRATE ~ SEX, data = furness)
```

The graphic suggests that the variances are unequal. We start with the default Welch test:

```
t.test(METRATE ~ SEX, data = furness)     # default: Welch two-sample t test
```

```
        Welch Two Sample t-test

data:  METRATE by SEX
t = -0.7732, df = 10.468, p-value = 0.4565
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -1075.3208   518.8042
sample estimates:
mean in group Female   mean in group Male
            1285.517             1563.775
```

To get the classical two-sample t test we have to explicitly say that the variances are assumed to be equal:

```
t.test(METRATE ~ SEX, data = furness, var.equal = T)  # classical t test,
assuming equal variances
```

```
        Two Sample t-test

data:  METRATE by SEX
t = -0.7009, df = 12, p-value = 0.4968
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -1143.3057   586.7891
```

```
sample estimates:
mean in group Female   mean in group Male
            1285.517             1563.775
```

The output of both test variants is similar as seen for the one-sample t test. But the confidence interval is now for the difference in means. Zero, the difference under H_0, is contained in the 95 % confidence interval, suggesting that male and female fulmars do not differ (much) in metabolic rate. The same is suggested by the p-values > 0.05. The degrees of freedom for the classical test are $n_F + n_M - 2 = 6 + 8 – 2 = 12$.
Note that the specification METRATE ~ SEX corresponds to a model formula (metabolic rate explained by sex) and is usually very handy to use for data stored in data frames.
Alternatively, we could specify both groups separately, but that is more complicated to write:

```
# alternative specification
t.test(furness$METRATE[furness$SEX == "Female"],
       furness$METRATE[furness$SEX == "Male"], var.equal = T)
```

We recommend using graphics (as the boxplot below) and your own reasoning to decide which test you want to use. There is also a formal test for the equality of variances in R, var.test(), which we try here quickly:

```
var.test(METRATE ~ SEX, data = furness)
```

The non-significant test result suggests that variances might be equal (not different). However, this test suffers the same drawback as any statistical test: For small samples, you are likely to get a non-significant result (due to a lack of statistical power), in this case leading you to reject the assumption of equal variances even if the difference is in fact rather large (as we might conclude from the box plot). For large samples, you are very likely to get a significant test result and reject the assumption of equal variances (high power). For this reason we prefer judging the variances graphically.

A demonstration how one could do the two-sample t test ”by hand” is provided in the R-Code file of this chapter.

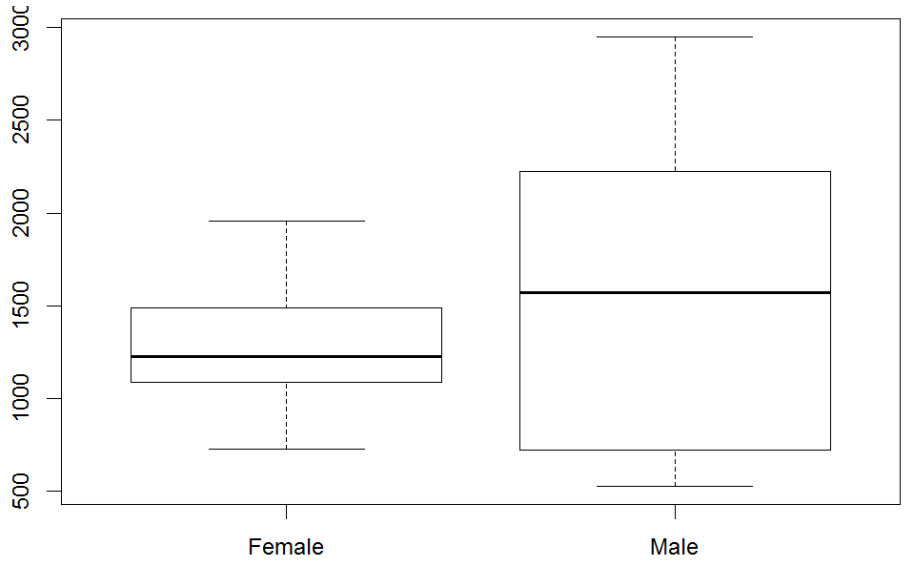

Figure: Box-plot of the metabolic rates of female and male Northern fulmars.

5.2.3 The t test for paired samples

Paired samples occur if there are two measurements on the same experimental unit. This means that these two measurements are not independent. For example, you might have measured daily energy intake in 11 women twice, pre- and postmenstrual. A large premenstrual value is likely to be paired with a large postmenstrual value (of the same woman). To assess whether pre- and postmenstrual energy intake differ, we can calculate the difference between the two for every woman and perform a one-sample t test, comparing the sample of differences against zero (the reference value under H_0 of no difference). An important assumption for this test is that the differences have a distribution that is independent from the level (differences between large values should not have a larger variation than those between small values).

Other examples of paired data are measurements on pairs of plants growing in the same pot or measurements on pairs of siblings. It is very important to understand the difference between independent and non-independent data. You will come across them again whenever you have "hierarchical" or "clustered" data.

Let's look at the example and graphically inspect the differences between pre- and postmenstrual measurements. The data are from the library ISwR that accompanies Daalgard (2008):

```
library(ISwR)
data(intake)
# inspect differences graphically
difference <- intake$post - intake$pre
average <- (intake$post + intake$pre)/2
plot(average, difference)
```

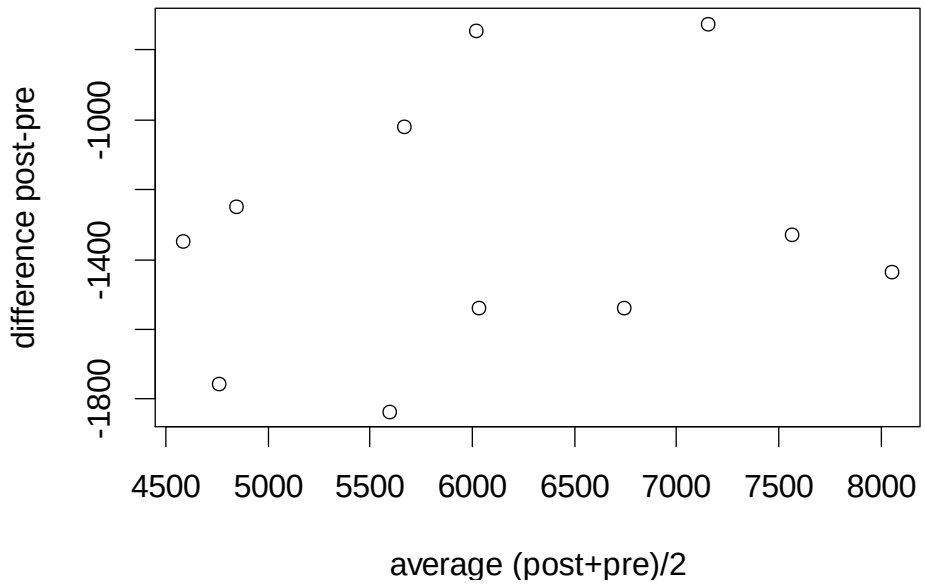

Figure: Graphical check whether the size of the differences is independent of the level of energy intake.

The sample of just n=11 women is very small. But the graphic indicates that assuming differences to be independent from the level of energy intake is reasonable. and applying a paired t test should be save. I It is equivalent to use both samples and specify that they are paired, or we can just compare the differences to zero by a one-sample t-test:

```
# using both samples and say they are paired
> t.test(intake$post, intake$pre , paired = TRUE)

        Paired t-test

data:  intake$post and intake$pre
t = -11.9414, df = 10, p-value = 3.059e-07
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -1566.838 -1074.072
sample estimates:
mean of the differences
              -1320.455


# alternative: use one-sample t test on difference
> t.test(difference, mu = 0)

        One Sample t-test

data:  difference
t = -11.9414, df = 10, p-value = 3.059e-07
alternative hypothesis: true mean is not equal to 0
95 percent confidence interval:
 -1566.838 -1074.072
sample estimates:
mean of x
-1320.455
```

The output is numerically equal, just the description (data, mean of the differences vs. mean of x) is a bit different. Note that such a paired design is very "efficient" if it fits the research question asked (i.e., if you are interested in things that differ within-subject). Through the pairing of subjects, you can basically get rid of the variance between subjects. Ignoring the paired nature of the data would in this case decrease the statistical power. (However, ignoring the paired or clustered nature of data in other settings can lead to the opposite, also known as pseudo-replication.)

Final remark: If you understand in which situation the paired t test should be used instead of the two-sample t test, you will find it much easier to understand when you should take into account hierarchical structure in a more complex dataset. For example, you might have multiple measurements per individual animal (or plant) and several individuals may have the same mother (come from the same nest, have the same genotype, etc.). Such data need more complex hierarchical models (also called mixed-effects models).

5.3 Rank-based alternatives to t tests

All t tests are based on the assumption that our samples come from populations of normally distributed values. Although t tests are fairly robust against departures from this assumption, they are not a good choice if this assumption is severely violated; or you might just not want to make the normality assumption. In this case you can use a non-parametric test (which does not rely on any parametric distributional assumption). For this sort of test, the data are replaced by ranks (rank-based tests).
A rank-based analogue for the one-sample t test is the Wilcoxon signed-rank test (also called one-sample Wilcoxon test). Instead of the two-sample t test you can use the Wilcoxon rank sum test (two-sample Wilcoxon test, also known as Mann-Whitney test or U-test) and instead of the t test for paired samples you can use a matched-pairs Wilcoxon test.

Just as you can do all t tests using the t.test() function, you can do all these tests using the wilcox.test() function. Read the description of wilxox.test if you want to know more.
The lack of distributional assumptions can mislead people to use non-parametric tests inappropriately. You can never get around thinking about your data. For example, if your observations are not independent, a non-parametric test is not the right solution. Moreover, although no specific distribution is assumed for the two samples compared by the rank sum test, the distributions of both groups should be equal (have the same shape).

5.4 Tests for categorical data

5.4.1 Compare a proportion to a reference value: the binomial test

You have put 215 seeds in a petri dish and 39 germinated. Now you want to check whether this germination rate p = 39/215 conforms to the germination rate given by the trader, p_0 = 0.15. The null-hypothesis is: p = p_0. You can do this in R by using the function binom.test() which is based on the probabilities of the binomial distribution B(n,p):

```
> binom.test(39,215,0.15)

        Exact binomial test

data:  39 and 215
number of successes = 39, number of trials = 215, p-value = 0.2135
alternative hypothesis: true probability of success is not equal to 0.15
95 percent confidence interval:
 0.1322842 0.2395223
sample estimates:
probability of success
             0.1813953
```

The output includes a confidence interval for the probability to germinate. Since the confidence interval contains p_0, the value under H_0, the observed germination rate conforms to the information given by the trader. Alternatively, you could do this test using the function prop.test() which makes use of the normal approximation N(np,np(1-p)):

```
prop.test(39,215,0.15)

        1-sample proportions test with continuity correction

data:  39 out of 215, null probability 0.15
X-squared = 1.425, df = 1, p-value = 0.2326
alternative hypothesis: true p is not equal to 0.15
95 percent confidence interval:
 0.1335937 0.2408799
sample estimates:
        p
0.1813953
```

The result is similar to binom.test() because n is large (n = 215). By default, Yates continuity correction is used (see for example Daalgard 2008 for more detail).

5.4.2 Compare two proportions: χ^2 test

You have sown 108 seeds of plant species A and 117 seeds of plant species B in petri dishes. After a week, 81 seeds of species A and 36 seeds of species B germinated. Are the germination rates p_A and p_B of the two species different? The corresponding H_0 is: $p_A = p_B$.

There are several tests that may be used for this example. We will look at the $\chi 2$ test first and do it "by hand" in R. We start by making a matrix of the observed frequencies:

```
sown <- c(108,117)
germinated <- c(81,36)
notgerminated <- sown-germinated
observed <- matrix(c(notgerminated, germinated), ncol=2,
dimnames=list(c("species A","species B"),c("not
germinated","germinated"))); observed

          not germinated germinated
species A             27         81
species B             81         36
```

The $\chi 2$ test compares observed frequencies with the frequencies expected under the assumption that the germination rate is independent of the species (and vice versa, which in this example makes little biological sense). Expected frequencies are calculated by multiplying the marginal totals for each of the four cells of the matrix. For seeds germinated from species A this would be:

(all seeds of species A * all germinated seeds) / Total number of seeds

(27+81) * (27 + 81) / (27 + 81 + 81 + 36) = 51.84

```
# Calculate the marginal totals
speciesA <- sum(observed["species A",])      # Total of species A
speciesB <- sum(observed["species B",])      # Total of species B
germ <- sum(observed[,"germinated"])         # Total germinated
notgerm <- sum(observed[,"not germinated"])  # Total not germinated
Total <- sum(observed)                       # Total seeds overall

# calculate expected frequencies, assuming independence of species and
germination rate
germA <- speciesA * germ / Total
germB <- speciesB * germ / Total
notgermA <- speciesA * notgerm / Total
notgermB <- speciesB * notgerm / Total
# matrix of expected frequencies
expected <- matrix(c(notgermA, notgermB, germA, germB),
dimnames=list(c("species A","species B"),c("not germinated","germinated")),
ncol = 2); expected
          not germinated germinated
species A          51.84      56.16
species B          56.16      60.84
```

The test statistic is then

$$\chi^2 = \sum \frac{(observed - expected)^2}{expected}$$

```
# calculate Chi-square observed
Chi2 <- sum( ((observed - expected)^2) / expected ); Chi2
[1] 44.01812
# p-value
pchisq(Chi2, df = 1, lower.tail = FALSE)
[1] 3.253498e-11
```

The test result indicates strong dependence between plant species and germination rates. Of course, R has a function that does the calculation much faster. We get the same result as calculated by hand if we use:

```
chisq.test(observed, correct=FALSE)       # agrees with our own calculation

        Pearson's Chi-squared test

data:  observed
X-squared = 44.0181, df = 1, p-value = 3.253e-11
```

The default in R is correct = TRUE for a $\chi 2$ test using Yates continuity correction (as in prop.test() above). This correction makes the 95 % confidence interval a bit wider.

```
chisq.test(observed)

        Pearson's Chi-squared test with Yates' continuity correction

data:  observed
X-squared = 42.2639, df = 1, p-value = 7.975e-11
```

The function prop.test() is equivalent to chsiq.test() for a 2 x 2 contingency table as in this example:

```
> prop.test(germinated,sown, correct = FALSE)

        2-sample test for equality of proportions without continuity
correction

data:  germinated out of sown
X-squared = 44.0181, df = 1, p-value = 3.253e-11
alternative hypothesis: two.sided
95 percent confidence interval:
 0.3254180 0.5591974
sample estimates:
   prop 1    prop 2
0.7500000 0.3076923

> prop.test(germinated,sown)

        2-sample test for equality of proportions with continuity
correction

data:  germinated out of sown
X-squared = 42.2639, df = 1, p-value = 7.975e-11
alternative hypothesis: two.sided
95 percent confidence interval:
 0.3165148 0.5681005
sample estimates:
   prop 1    prop 2
0.7500000 0.3076923
```

With small frequencies (one or more frequencies < 5), the test statistic is not nicely $\chi 2$-distributed and Fisher's exact test is recommended. Let's look at an example with one frequency < 5:

```
> sown<-c(12,13)
> germinated <-c(9,4)
```

```
> notgerminated <- sown-germinated
> observed <- matrix(c(notgerminated, germinated), ncol=2,
dimnames=list(c("species A","species B"),c("not
germinated","germinated"))); observed
          not germinated germinated
species A              3          9
species B              9          4

> chisq.test(observed)

        Pearson's Chi-squared test with Yates' continuity correction

data:  observed
X-squared = 3.2793, df = 1, p-value = 0.07016

> fisher.test(observed)

        Fisher's Exact Test for Count Data

data:  observed
p-value = 0.04718
alternative hypothesis: true odds ratio is not equal to 1
95 percent confidence interval:
 0.01746573 1.11027182
sample estimates:
odds ratio
 0.1617985
```

5.5 Outlook: linear models

To analyze more complex data such simple tests as described in this chapter have to be replaced by models. We will have a look at a very simple *General Linear Model* that could be used to analyze the furness data. Instead of doing a two-sample t test, we can fit a linear model to explain metabolic rates by using the function lm() as follows:

```
> furness$Male <- as.numeric(furness$SEX == "Male")
> model <- lm(METRATE ~ Male, data = furness)
> summary(model)

Call:
lm(formula = METRATE ~ Male, data = furness)

Residuals:
     Min       1Q   Median       3Q      Max
-1037.97  -510.43   -59.37   524.33  1386.22

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   1285.5      300.1   4.283  0.00106 **
Male           278.3      397.0   0.701  0.49676
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 735.2 on 12 degrees of freedom
Multiple R-squared: 0.03932,    Adjusted R-squared: -0.04073
F-statistic: 0.4912 on 1 and 12 DF,  p-value: 0.4968
```

Note that we have created an indicator variable for Male fulmars (1 = Male, 0 = Female). This model estimates two parameters. The intercept is the estimated metabolic rate for female

fulmars, the estimate Male is the difference between the rate of males and females. The fitted model is: 1285.5 + 278.3 * Male = 1563.8. For male fulmars the estimate 278.3 is added (multiplied by Male = 1), for female fulmars it is not added (because Male = 0).

We got the two metabolic rates directly by using t.test():

```
t.test(METRATE ~ SEX, data = furness)

        Welch Two Sample t-test

data:  METRATE by SEX
t = -0.7732, df = 10.468, p-value = 0.4565
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -1075.3208   518.8042
sample estimates:
mean in group Female   mean in group Male
            1285.517             1563.775
```

5.6 Literature

Dalgaard, P. (2008). Introductory statics with R. New York, Springer.
Quinn, G. P. and Keough, M. J. (2002) Experimental design and data analysis for biologists. Cambridge University Press.
Altman D. G. & Bland J. M. (1995) Absence of evidence is not evidence of absence. BMJ 311:485.