DEPT. OF MATHEMATICAL SCIENCES
UNIVERSITY OF COPENHAGEN

HELLE SØRENSEN
NOVEMBER 2017

Getting started with R and RStudio

R is a programming language and software environment for statistical computing and graphics. The flexibility and large number of user-contributed packages make it an extremely useful statistical toolbox for almost any data analysis and visualization problem. RStudio is an integrated development environment for R. It provides a nice environment for working with R, but you can also use R without RStudio. Some of the comments below, *e.g.* regarding handling of files, only apply directly if you run RStudio, but all actual computationcs can be carried out without using RStudio. You can read more about the programs at the webpages `www.r-project.org` and `www.rstudio.com`. You can also download and install the programs from the pages, see also Appendix A..

This note gives a short introduction to R and RStudio which hopefully makes it easy to get started. The note shows how to import data, how to make simple graphs, and how to compute simple statistics (like sample mean and standard deviation). No actual statistical analyses are illustrated.

The note is intended for use in the courses *Sandsynlighedsreging og Statistik* (SS) and *Statistik for Biokemikere* (StatBK) at University of Copenhagen. Not all sections are relevant for both courses. In the courses, supplementary material on usage of R for specific statistical analyses is included in the course book or handed out seperately.

Contents

1 Working with R and RStudio

We assume that R and RStudio have already been installed (see Appendix A for details).

RStudio windows, R console and prompt When you start RStudio, three or four windows appear, see Figure 1. The windows will be commented upon in the note as they become relevant.

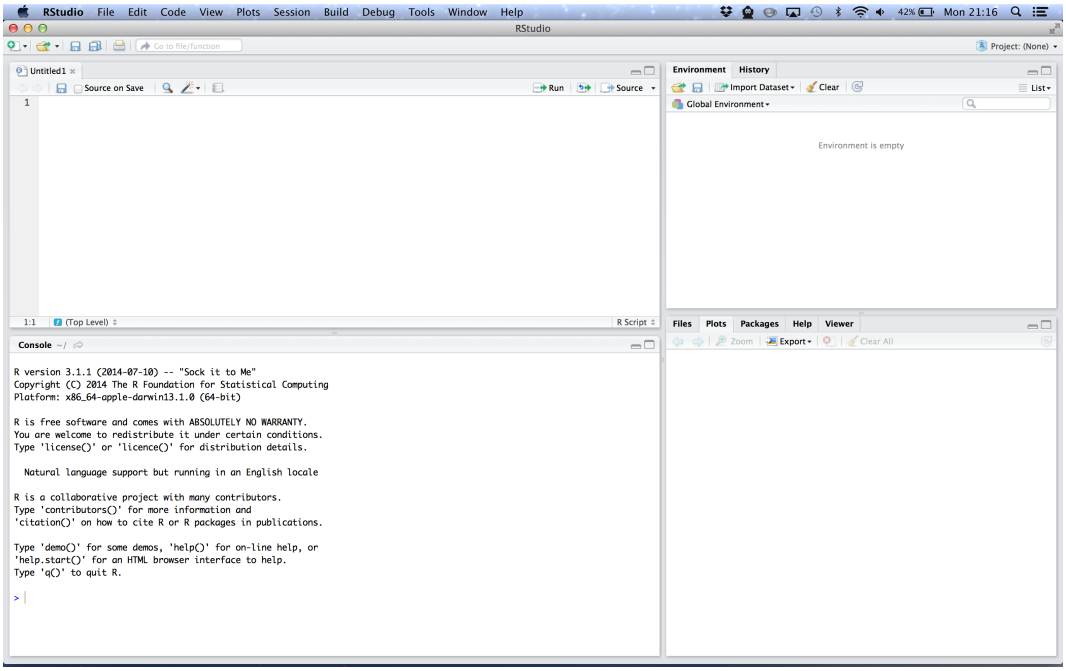


Figure 1: RStudio windows.

First, look at the lower left window, the so-called console. At the bottom you meet the prompt

```
>
```

which indicates that R is ready to receive a command. You may simply write commands after the prompt (followed by Enter), and the result appears. For example,

```
> 3+2
[1] 5
```

R as a pocket calculator R may be used as a simple pocket calculator. All standard functions (powers, exponential, squareroot, logarithm, *etc.*) are of course built-in. For example,

```
> (17-3) / (3-1) + 1
[1] 8
> 3*4 - 7 + 1.5
[1] 6.5
> sin(pi/2)
```

```
[1] 1
> log(exp(0.3))
[1] 0.3
> 1.8^2 * exp(0.5) - sqrt(2.4)
[1] 3.792664
```

The last example shows that $1.8^2 \cdot e^{0.5} - \sqrt{2.4}$ equals 3.79.

Comments Everything after # is read as a comment and is disregarded by R. For example, R is not 'disturbed' by the command `R is great at multiplication`:

```
> 6*5  # R is great at multiplication
[1] 30
```

Comments are very useful when commands are saved in files for later use.

Arrow keys Notice that you can use the arrow keys in the console, to easily access previous commands.

Closing R You quit R either by via the File menu or by typing the command

```
q()
```

at the prompt. You are then asked if you want to 'save workspace image', *i.e.* all commands and output. In general, you do not want to do so, so you should answer 'No'. You would rather want to save the commands in a file as to be explained now.

R scripts Instead of typing the commands directly at the prompt, you can write your commands in a so-called R script or R program, and then run the commands from the script. This is generally recommended. Make sure to save all relevant commands — but not all the irrelevant ones!

An R script is just a file consisting of commands. RStudio has a built-in editor which you can use to edit your R-programs. In the File menu you can start a new R-script or open an existing one. Write commands in the script and run a command line by pressing `ctrl-r` (for Mac, `cmd+Enter` has the same effect), or via the Code menu. The command is then automatically transferred to the R console and executed, and the command as well as the result appear as before.

Make a folder on your computer for data files and R scripts related to the course/project, and save the R script in the folder via the File menu.

Working directory R uses the current working directory for saving files and looking for data files, if you do not specify a different folder specifically. The default choice for working directory is rarely a good one, so it is convenient to change it in the beginning of each session. This is done via the Session menu. Alternatively, the default working directory can be permanently changed via the Tools menu: Choose Global options, then R General, and browse to the preferred folder.

Help system The `help` function provides information on the various functions and their syntax in R. For example,

```
> help(log)
> help(mean)
```

will give information on logarithms and exponentials, and about the `mean` function, respectively. The commands could be replaced by `?log` and `?mean`.

Unfortunately this strategy only works when you know what function to use and remember its name. Quite often you remember that something can be done, but have forgotten how. The reference card by Short (2007) is very well suited for this situation.

If you want an overview of all functions, you can start the help page with the command `help.start()`. Then you can browse around the available reference pages.

Books and manuals There are numerous book and manuals to R. Readers interested in a broader introduction to R can consult the books by Crawley (2007) or Venables and Ripley (2002), which is in principle written for the S language, but everything applies to R, too. The handbook by Ekstrøm (2012) answers many basic as well as advanced questions about R. There are also numerous documents on the R project homepage, `www.r-project.org`. In particular, you may find the manual *An Introduction to R* useful.

2 Variables, vectors, matrices

Variables You can assign values to variables and use them for later computations. For example, the following commands (once again) compute $1.8^2 \cdot e^{0.5} - \sqrt{2.4}$:

```
> x <- 1.8^2
> y <- exp(0.5)
> z <- x*y - sqrt(2.4)
> z
[1] 3.792664
```

R is quite flexible with the names that are allowed for variables, but note that variable names cannot start with a digit. Also notice that that variable names are case sensitive so `x` and `X` are different variables. Notice that the variable names appear in the Environment window (top right window in RStudio), telling you that you have now defined objects with those names.

Vectors The combine function, `c`, is used to construct a vector from several elements:

```
> daysPerMonth <- c(31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31)
> daysPerMonth
 [1] 31 28 31 30 31 30 31 31 30 31 30 31
```

The `[1]` in front means that 31 is the first element of the vector. This information is only relevant is the vector is so long that is exceeds one line, see the examples below.

The length (number of elements) of a vector is computed with `length`:

```
> length(daysPerMonth)
[1] 12
```

Computations with vectors Standard arithmetic in R is 'vectorized' in the sense that mathematical functions work element-wise on vectors; *i.e.* `x + y` adds each element of `x` to the corresponding element of `y`.

Assume for example that the weight in kg and the height in cm are given for 5 people, and that we want to compute the body mass index (BMI) for each person. The BMI is defined as the weight in kg divided by the height in metres squared. For example, the BMI for a 158 cm tall person with weight 65 kg is $65/1.58^2 = 26.04$.

```
> weight <- c(65, 77, 84, 82, 93)      # Vector with weights in kg
> weight
[1] 65 77 84 82 93
> height <- c(158, 180, 186, 179, 182) # Vector with heights in cm
> height
[1] 158 180 186 179 182
> heightM <- height/100                # Height in metres
> heightM
[1] 1.58 1.80 1.86 1.79 1.82
> bmi <- weight/heightM^2              # Compute BMI
> bmi
[1] 26.03749 23.76543 24.28026 25.59221 28.07632
```

Usually computations that combine vectors are meaningful only if the vectors are of the same length, and you should be very careful if you operate on vectors of different lengths.

Functions applied to vectors are also computed element-wise. For example the natural logarithm to the five BMI values are easily computed in one command:

```
> log(bmi)
[1] 3.259538 3.168232 3.189664 3.242288 3.334926
```

Extracting elements from vectors One may use extract specific elements in a vector with brackets:

```
> daysPerMonth[2]        ## Element no. 2
[1] 28
> daysPerMonth[-2]       ## All elements but no. 2
[1] 31 31 30 31 30 31 31 30 31 30 31
> daysPerMonth[1:4]      ## Elements 1-4
[1] 31 28 31 30
> daysPerMonth[c(1,3,5)] ## Elements 1, 3, 5
[1] 31 31 31
```

Logical expressions Consider the weight and height data from above, and assume that we are only interested in people taller than 180 cm. Consider the following vector:

```
> (height>180)
[1] FALSE FALSE  TRUE FALSE  TRUE
```

It has values `TRUE` or `FALSE` according to whether the criterion "height being above 180" is true of not. The logical vector can be combined with the bracket syntax such that the height, weight and BMI for those people are extracted:

```
> height[height>180]    # height where height is above 180
[1] 186 182
> weight[height>180]    # weight where height is above 180
[1] 84 93
> bmi[height>180]       # BMI where height is above 180
[1] 24.28026 28.07632
```

If we had wanted to include the person who is 180 cm, then we should have replaced `>` by `>=`. Other logical operators are `<=`, `<`, `==`, and `!=` where `==` and `!=` mean equal and not equal, respectively.

Assume for example that the two first people are women and the last three are men, as coded in the variable `sex` below. The BMI for the men in the sample, and the BMI for the people in the sample that are exactly 180 cm, respectively, are then extracted as follows:

```
> sex <- c("F","F","M","M","M")  # New sex variable
> sex
[1] "F" "F" "M" "M" "M"
> bmi[sex=="M"]                  # BMI for males
[1] 24.28026 25.59221 28.07632
> bmi[height==180]               # BMI where height is 180
[1] 23.76543
```

Notice that we put `""` around the value `M` for the `sex` variable. This is necessary for variables with character values.

Matrices Matrices are constructed with the `matrix` function. Notice that, by default, values are filled in column-wise (not row-wise):

```
> A <- matrix(c(1,4,5,6,9,11), nrow=2, ncol=3)
> A
     [,1] [,2] [,3]
[1,]    1    5    9
[2,]    4    6   11
> B <- matrix(c(1,4,5,6,9,11), nrow=2, ncol=3, byrow=TRUE)
> B
     [,1] [,2] [,3]
[1,]    1    4    5
[2,]    6    9   11
```

Cells, as well as complete rows and columns can be extracted using brackets:

```
> B[2,3]
[1] 11
> B[1,]
[1] 1 4 5
> B[,2]
[1] 4 9
```

The bracket syntax can also be used if we want to modify values in the matrix, either in just one cell or in rows and columns:

```
> C <- matrix(0, nrow=4, ncol=3)
> C
     [,1] [,2] [,3]
[1,]    0    0    0
[2,]    0    0    0
[3,]    0    0    0
[4,]    0    0    0
> C[,2] <- 1
> C[4,] <- c(7,9,13)
> C[1,3] <- 4
> C
     [,1] [,2] [,3]
[1,]    0    1    4
[2,]    0    1    0
[3,]    0    1    0
[4,]    7    9   13
```

R has all sorts of functionalities for matrix computations. See for example the help pages for `solve`, `t`, `det`, `matmult` and `eigen`.

3 Data frames and how to read data into R

In the course we will use R for statistical analyses, and thus work with data that are somehow supplied to us by other people. There are several ways of reading data into R, some of which we will describe here.

Typing data directly into R For small datasets you may simply type the data directly into R with `c` as described in Section 2. However, this above method is rarely recommendable as most datasets are much larger (more observations, more than one variable) and are already stored in a plain text file or in an excel sheet. We now show how to import data from such files.

Reading data from plain text files Consider a dataset on fecundity of crickets (Samuels and Witmer, 2003, Example 12.2). The dataset consists of 39 observations of two variables, namely the body weight and the number of matrure eggs for each of 39 crickets. Suppose the data are stored in the text file `crickets.txt` as follows:

```
eggs bodywt
15 3.68
33 3.69
84 4.56
.
.     [More datalines here]
.
10 2.51
```

Then you may input the data using the `read.table` as follows (without the commment):

```
> cricketdata <- read.table("crickets.txt", header=TRUE)  # Read from
                                                          # text file
```

R has now created a dataset (or data frame) named `cricketdata`. Notice that the name of dataset appears in the Environment window. The dataset has two vectors or variables: `bodywt` and `eggs`. Notice the option `header=TRUE` in the first line of the `read.table` command. It is appropriate if the text file contains the names of the variables (as is the case for our dataset here). If this is not the case, then it should be left out.

You should always check that a new dataset has the appropriate content. If you click on the name, the dataset is shown in the upper left window. You can see the first few datalines with the `head` command, and get summaries of each variable with the `summary` command:

```
> head(cricketdata)
  eggs bodywt
1   15   3.68
2   33   3.69
3   84   4.56
4   10   3.33
5   55   4.22
6   82   3.88

> summary(cricketdata)
      eggs            bodywt
 Min.   :  0.00   Min.   :1.900
 1st Qu.: 16.00   1st Qu.:3.100
 Median : 32.00   Median :3.490
 Mean   : 39.74   Mean   :3.518
 3rd Qu.: 53.50   3rd Qu.:3.955
 Max.   :121.00   Max.   :5.230
```

We will comment on the `summary` output in Section 4. It can also be very useful to use the `plot` command on a dataset. This plots each pair of variables against each other. In our case, since there are only two variables, the command `plot(cricketdata)` gives a single scatterplot (shown in Section 5).

Reading data from Excel sheets The easiest way to read data from Excel sheets is to use the `Import Dataset` facility in the Environment window (top right). Click it, choose Excel, and

browse to the excel sheet with the data. A preview of the code that will be used for reading, is now shown in the bottom, right part of the window, and that a preview of the dataset is shown in the middle. Here, you can also change the data type for each variable. In the bottom, left part of the window you can change various things, e.g. the name that is designated for the data frame. When you finally click `Import`, the code is run, and a R data frame is generated and shown.

Assessing vectors in a data frame Recall that the data frame `cricketdata` consists of two vectors, `bodywt` and `eggs`. Notice that we must tell R in which dataset to look for the variables. If not, we get an error message:

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

The vectors can be assessed with the $-syntax, where we explicitly tell R the dataset as well as the variable name:

```
> cricketdata$bodywt
 [1] 3.68 3.69 4.56 3.33 4.22 3.88 3.24 3.47 2.70 3.58 2.88 4.91
[13] 3.97 4.22 3.49 5.23 3.51 3.94 3.99 3.04 4.00 3.32 2.88 2.82
[25] 3.98 2.97 3.46 3.26 3.53 3.05 4.43 2.65 1.90 3.87 3.18 3.50
[37] 3.15 3.21 2.51

> cricketdata$eggs
 [1]  15  33  84  10  55  82   4  32   9  55   0 121  52  13  18
[16] 120  27  92  47  23  69  17  28  37  45   8  56   5  14  47
[31]  50  26  27  34  24  86  29  46  10
```

As an alternative, we can use the `with` function. Say that we want to make a histogram of the `bodywt` variable. As will be explained in Section 5, this can be done with the function `hist`. The following command tells R to make a histogram of the variable `bodywt` from the dataset `cricketdata`:

```
> with(cricketdata, hist(bodywt))
```

The general syntax is `with(dataset, command)`, where we tell R to carry out a certain command, using (among others) the variables from the specified dataset.

Another alternative is to tell R that it is allowed to look for variables in a specific data set even if the user does not tell is explicitly in the commands. This is done using the `attach` function:

```
> attach(cricketdata)  # Gives direct access to variables in dataset
```

Now R looks for the variables in the `cricketdata` dataset, and `bodywt` can be assessed directly:

```
> bodywt
```

```
 [1] 3.68 3.69 4.56 3.33 4.22 3.88 3.24 3.47 2.70 3.58 2.88 4.91
[13] 3.97 4.22 3.49 5.23 3.51 3.94 3.99 3.04 4.00 3.32 2.88 2.82
[25] 3.98 2.97 3.46 3.26 3.53 3.05 4.43 2.65 1.90 3.87 3.18 3.50
[37] 3.15 3.21 2.51
```

Similarly R is told to no longer look in the data frame by writing

```
> detach(cricketdata)
```

Attaching dataset can be convenient, but is also dangerous if one has are several variables with the same names (in different dataset). Then it is almost impossible to control which of the variables is used by R. The safe way is therefore to use `with` or the $-syntax. In the following we sometimes assume, however, that `cricketdata` has been attached in order to make the commands more easy to read.

Subsets of data frames Sometimes we are interested in subsets of the original data. Suppose, for example, that we should only use data crickets with a body weight less than 4. The command

```
> smallcrickets <- subset(cricketdata, bodywt<4)
```

creates a new dataset, `smallcrickets` with only 32 observations. All the logical expressions described in Section 2 can be used in combination with `subset`.

Transformation of vectors in data frames Sometimes we use transformations of the variables in the dataset rather than the original variables. Suppose we need the `cricketdata` data frame to contain a vector with the natural logarithm to the body weight as values. The following command creates a data frame with such a vector:

```
> cricketdata <- transform(cricketdata, logbodywt=log(bodywt))
> cricketdata
   eggs bodywt logbodywt
1    15   3.68 1.3029128
2    33   3.69 1.3056265
3    84   4.56 1.5173226
.
.     [More datalines here]
.
39   10   2.51 0.9202828
```

4 Simple summary statistics

Sample mean, standard deviation, variance We are often interested in summary statistics for the variables of interest. The sample mean, the sample variance and the sample standard deviation are computed by the functions `mean`, `var` and `sd` respectively. For example,

```
> mean(bodywt)  # Sample mean
[1] 3.517949
> var(bodywt)   # Sample variance
[1] 0.4424430
> sd(bodywt)    # Sample standard deviation
[1] 0.6651639
```

shows that the `bodywt` variable has mean 3.52, variance 0.44, and standard deviation 0.66. Recall that the standard deviation is the squareroot of the variance.

Quantiles The functions `min` and `max` finds the smallest and the largest value, `median` computes the median, and the function `quantile` furthermore computes the quartiles:

```
> min(bodywt)      # Smallest observation
[1] 1.9
> max(bodywt)      # Largest observations
[1] 5.23
> median(bodywt)   # Median (the 'middle' observation when sorted)
[1] 3.49
> quantile(bodywt) # Min, max, median, 25% and 75% quantile
   0%   25%   50%   75%  100%
1.900 3.100 3.490 3.955 5.230
```

Summary of a variable As an alternative to some of the above commands, the `summary` command is useful:

```
> summary(bodywt)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  1.900   3.100   3.490   3.518   3.955   5.230
```

Recall that we already used the `summary` function in Section 3, at that time with a dataset rather than a variable as argument. This gave a summary of each variable in the dataset.

Correlation coefficient Pearson's correlation coefficient of two variables are computed with `cor` as follows:

```
> cor(bodywt, eggs)  # Correlation coefficient
[1] 0.6879256
```

5 Simple graphs

One of the advantages of R is its ability to produce high level plots. In the following we show how to make scatterplots, histograms, boxplots, and QQ-plots, but there are numerous other high-level plots in R.

Scatterplots The scatterplot below (left) of the two variables `bodywt` and `eggs` is produced by the function `plot` as follows:

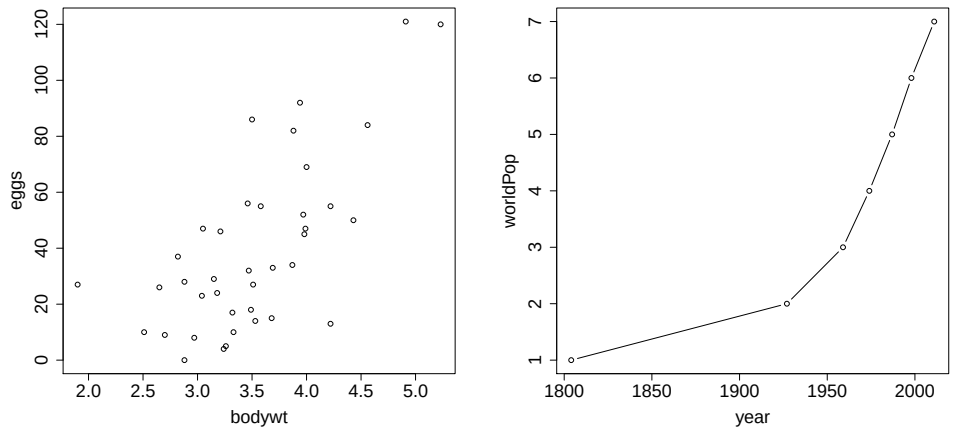
```
> plot(bodywt,eggs)  # Scatter plot of eggs against bodywt (left)
```

It is sometimes useful to connect the datapoints with lines, with or without plotting the points themselves. For example, consider the growth of the world population. The variable `year` contains the years where the world population reached 1 up to 7 billions people (given in the variable `worldPop`). The plot to the right below shows a scatterplot where the points have been connected — this is obtained with `type="b"`. If we wanted the line segments only, without the points themselves, we should have used `type="l"` (`b` for "both", `l` for "lines").

Here come the commands:

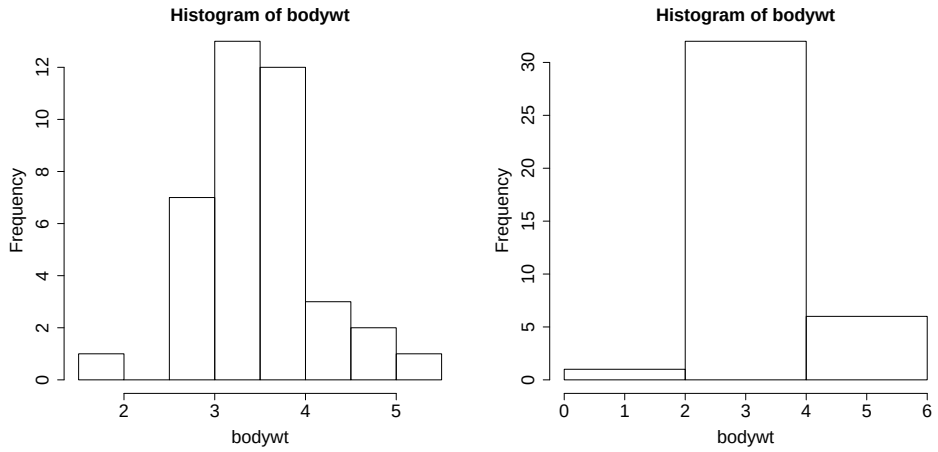
```
> year <- c(1804, 1927, 1959, 1974, 1987, 1998, 2011)
> worldPop <- c(1,2,3,4,5,6,7)
> plot(year, worldPop, type="b")   # Scatter plot with points and lines (right)
> plot(year, worldPop, type="l")   # Only line segments (not shown)
```

And here come the scatter plots:



Histograms Histograms are produced by the function `hist` as follows:

```
> hist(bodywt)                    # Histogram with default breaks
> hist(bodywt,breaks=c(0,2,4,6)) # Histogram with breaks at 0,2,4,6
```



Notice how you can choose the breakpoints if you are not satisfied with those that R chooses by default. In this situation the default (to the left) is fine, so there is no reason to choose them differently.

As default R has the frequencies, *i.e.* the number of observations, on the y-axis. If you prefer *relative* frequencies, then use the option `prob=TRUE`. Then the total area of the rectangles is one, as is useful when the histogram is to be compared with a density function.

Boxplots A boxplot gives a nice view of the distribution of the observations in a dataset, and are easily made with R:

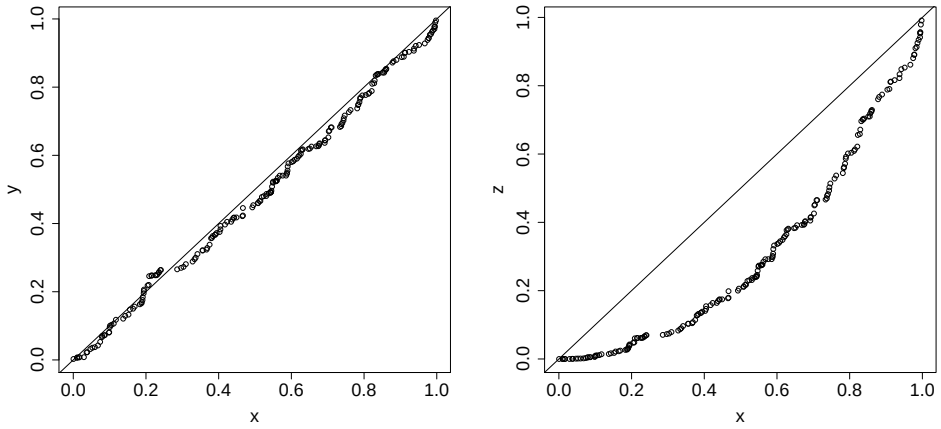
```
> boxplot(bodywt)     # Boxplot (left in figure below)
```

The boxplot is shown below to the left. The box represents the 25% quantile ($Q1$) and the 75% quantile ($Q3$), so the height of the box is the inter-quartile range, $\text{IQR} = Q3 - Q1$. The median is shown as a bold line inside the box. The whiskers represent the smallest and largest observations that are still within the interval $[Q1 - 1.5 \cdot \text{IQR}; Q3 + 1.5 \cdot \text{IQR}]$. If there are observations outside this interval, they are plotted as individual points.

QQ-plots for comparing two samples QQ-plots can be used for comparison of two samples. Consider two variables `x` and `y`. The command `qqplot(x,y)` sorts the two variables separately, and plots the ordered observations against each other. For example, the smallest `x` is plotted against the smallest `y`, the second-smallest `x` is plotted against the second-smallest `y`, *etc.* If the two variabes have the same distribution, then the points will lie around the line with intercept 0 and slope 1.

As example, consider the commands and the resulting graphs:

```
> x <- runif(200, 0, 1)   # 200 independent uniform variable
> y <- runif(200, 0, 1)   # 200 independent uniform variable
> z <- y^2
> qqplot(x,y)             # QQ-plot of x and y (left graph below)
> abline(0,1)             # Add x=y line
> qqplot(x,z)             # QQ-plot of x and y (right graph below)
> abline(0,1)             # Add x=z line
```

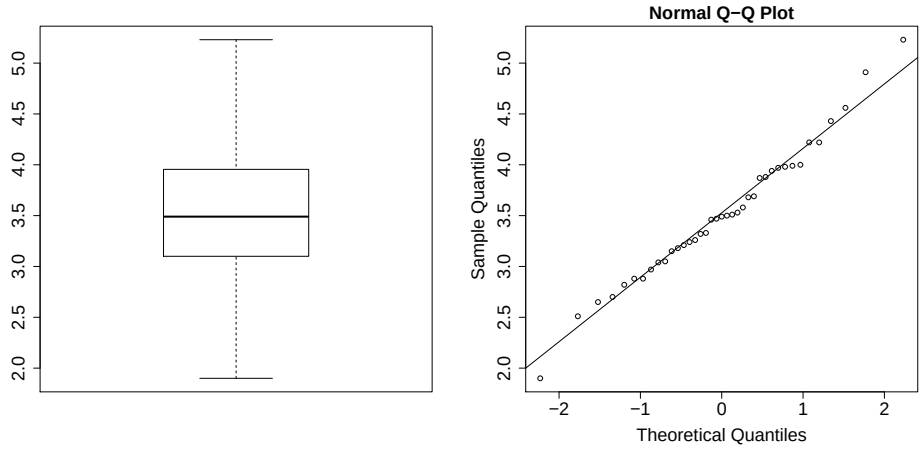


Since `x` and `y` are both drawn from the uniform distribution on $(0, 1)$, they have the same distribution and we see that the points are scattered around the unit line. In contrast `z=y^2` and `x` obviously have different distributions.

Normal QQ-plots A normal QQ-plot is a plot that compares the sample quantiles to the (theoretical) quantiles of the normal distribution. It is used to check if data can be assumed to be distributed according to a normal distribution is produced by `qqnorm`. A straight line going through the 25% and the 75% quantile is drawn on top with `qqline`. For the `bodywt` variable we write as follows:

```
> qqnorm(bodywt)    # Normal QQ-plot (right in figure below)
> qqline(bodywt)    # Adding a straigt line
```

The result is shown to the right in the following figure:

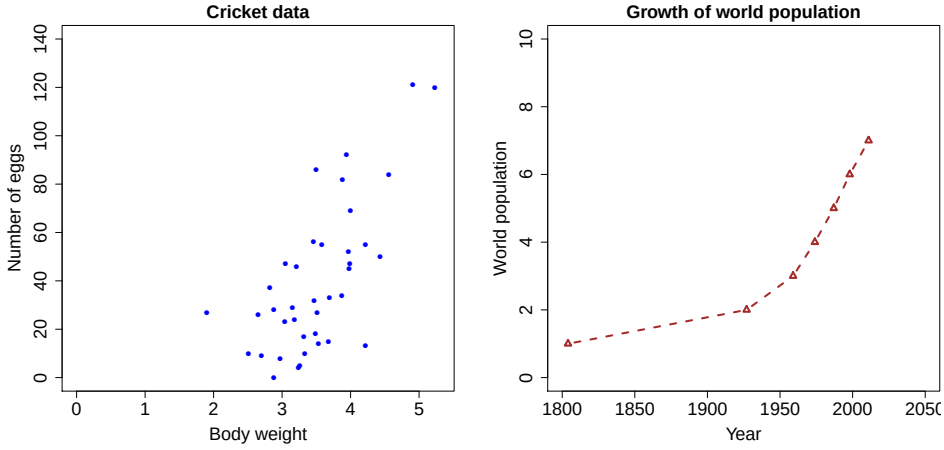


The boxplot indicates that the distribution is symmetric, and the QQ-plot shows no deviations from a straight line; hence it is reasonable to assume that the `bodywt` values come from a normal distribution.

Layout It is quite easy to change axes, labels, symbols, line types *etc.*; let us here mention a few of the possibilities. The following lines of code produce the figures below — compare to the original figures on page 12.

```
> plot(bodywt, eggs,
+      xlab="Body weight", ylab="Number of eggs",  # Labels
+      xlim=c(0,5.3), ylim=c(0,140),               # Range of axes
+      pch=16,                                     # Type of points
+      col="blue",                                 # Colour
+      main="Cricket data")                        # Title

> plot(year, worldPop,
+      type="b",                                 # both points and lines
+      xlab="Year", ylab="World population",     # Labels
+      xlim=c(1800,2050), ylim=c(0,10),          # Range og axes
+      col="brown",                              # Colour
+      lty=2, lwd=3,                             # Line type and line width
+      pch=2,                                    # Type of points
+      main="Growth of world population")        # Title
```



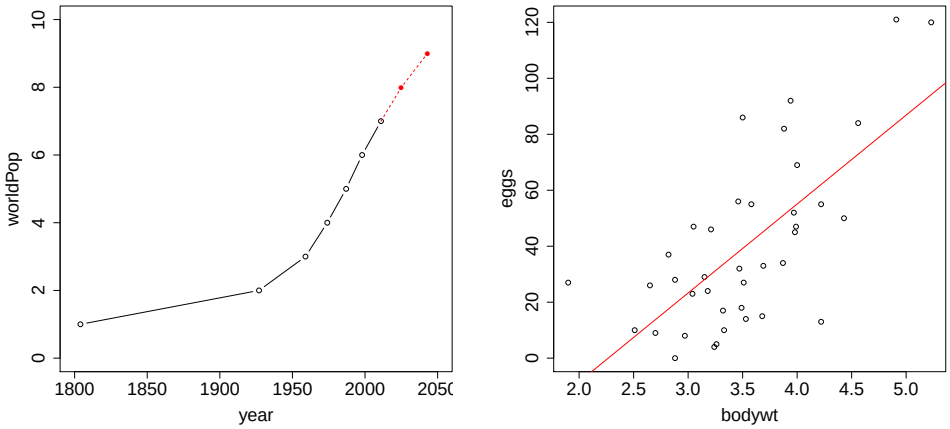
Arguments `cex.lab`, `cex.axis`, `cex.main` can be used to magnify the size of the axis labels, the number of the axes and the title. The defalt is 1, and the value 2, say, makes the text twice as large.

Adding points, line segments and lines One can add point, line segments, or straight lines to an existing plot with `points`, `lines`, and `abline`, respectively. For example, it is predicted that the world population will grow to 8 billions in 2025 and to 9 billions in 2043. These predicted data may be added to the graph as points or line segments (left plot below):

```
> plot(year, worldPop, type="b", xlim=c(1800,2050), ylim=c(0,10))
                                    # Original plot, but new y-axis
> points(yearPred, worldPopPred, col="red", pch=16)    # Red points
> lines(c(2011,2025,2043), c(7,8,9), col="red", lty=2) # Dashed line
```

The best-fitting straight line to cricketdata has intercept -72.05 and slope 31.78. In the right plot below, this line has been added to the scatterplot with `abline`:

```
> plot(bodywt, eggs)                 # Original plot
> abline(-72.05, 31.78, col="red")   # Straigt line added
```



Printing a graph to a file We often want to insert a graph in a document. In that case we need the graph to be printed to a file. There are several ways to do this. One way is to use the Export menu in the graph window. Another is to use the `dev.print` function as follows:

```
> plot(x,y)                          # Scatter plot
> dev.print(pdf, "my-plot.pdf")      # Copy graph to pdf-file
```

The first command makes the graph on the screen. The second command copies the graph to the pdf file `my-plot.pdf`. The file is saved in the working directory.

It you want to control the size of the gtaph, you can use the `pdf` function. See the help page, *i.e.*, write `?pdf` for details.

6 Probability distributions and random numbers

Cumulative distribution functions and probability or density functions are built-in functions in R for all the standard distributions (and many more). Moreover, it is possible to draw (simulate) random numbers from the distributions. This can be extremely useful: Numbers are drawn at random from a certain distribution and the properties of the simulated values can investigated to learn about the distribution.

The normal distribution The density and cumulative distribution function for the normal distribution are computed with `dnorm` and `pnorm`, respectively. For example, if X is normally distributed with mean 5 and standard deviation 1.5 (variance 2.25), then it turns out that $P(X \leq 4) = 0.2525$ and that $f(4) = 0.2130$ where f is the density of X:

```
> pnorm(4, mean=5, sd=1.5)
[1] 0.2524925
> dnorm(4, mean=5, sd=1.5)
[1] 0.2129653
```

Notice that you should specify the standard deviation, not the variance. The default values for `mean` and `sd` are 0 and 1, respectively, so `pnorm(0.5)`, say, computes $P(Y \leq 0.5)$ for $Y \sim N(0,1)$. Quantiles in the normal distribution can be computed with `qnorm`. For example, $P(Y \leq 1.645) = 0.95$, and $P(Y \leq 1.960) = 0.975$ for $Y \sim N(0,1)$:

```
> qnorm(0.95)
[1] 1.644854
> qnorm(0.975)
[1] 1.959964
```

We use `rnorm` to simulate data from the normal distribution. The following command gives us 6 random numbers drawn from the normal distribution with mean 5 and standard deviation 1.5:

```
> rnorm(n=6, mean=5, sd=1.5)
[1] 4.974034 5.594782 6.297043 8.920571 4.414532 6.634663
```

The binomial distribution In order to compute probabilities or draw random numbers from the binomial distribution we must specify the number of trials (the size) and the probability of success in each trial. Here we use 10 trials and a success probability of 0.25:

```
> pbinom(4, size=10, prob=0.25)    # P(X leq 4) if X is binomial (10, 0.25)
[1] 0.9218731
> dbinom(3, size=10, prob=0.25)    # P(X = 3) if X is binomial (10, 0.25)
[1] 0.2502823
> rbinom(n=5, size=10, prob=0.25)  # Five random draws from bin(10,0.25)
[1] 3 0 3 2 3
```

The hypergeometric distribution In order to compute probabilities or draw random numbers from the hypergeometric distribution we must specify the number of white and black balls in the urn and the number of balls in the sample drawn without replacement. For example, assume that there are 21 white balls and 79 balls in the urn, and that we draw five of them without replacement and count the number, X, of white balls in the sample:

```
> phyper(2, m=21, n=79, k=5)  # P(X leq 2) if X is hypergeom. (21,79,5)
[1] 0.9390218
> dhyper(1, m=21, n=79, k=5)  # P(X=1) if X is hypergeom. (21,79,5)
[1] 0.4190936
> rhyper(7, m=21, n=79, k=5)  # Seven random raws from hypergeom. (21,79,5)
[1] 0 0 2 0 1 0 1
```

The uniform distribution In order to compute probabilities or draw random numbers from the uniform distribution on (a,b), we must specify the end points a and b. For the uniform distribution on $(-1,2)$, we get:

```
> punif(0.5, min=-1, max=2)   # P(X leq 0.5) if X is uniform on (-1,2)
[1] 0.5
> dunif(0.25, min=-1, max=2)  # Density at 0.25 for uniform dist. on (-1,2)
[1] 0.3333333
> runif(n=2, min=-1, max=2)   # Two draws from uniform dist. on (-1,2)
[1] -0.1287942  0.1154072
```

The default values of `min` and `max` are 0 and 1. In particular, simulation from the uniform distribution on $(0,1)$ is done as follows:

```
> runif(n=4)    # Four random draws from uniform dist on (0.1)
[1] 0.14453894 0.12972789 0.74320940 0.03650309
```

Other distributions You probably noticed the system in the naming of the functions above. Each distribution type has a name, *e.g.* `norm` or `binom`, and the letter in front determines which feature of the distribution is considered: `d` gives the probability or density function, `p` gives the cumulative distribution function, `q` gives the quantiles, and `r` gives random draws from the distribution. Parameters in the distribution are given as extra arguments.

Random sampling from a finite set The `sample` functions make random samples from a finite set of elements. For example, the two next commands draw eight numbers at random between 1 and 10, with and without replacement:

```
> sample(x=1:10, size=8, replace=TRUE)  # 8 draws from 1,..,10; w. replacem.
[1] 9 1 1 4 3 3 9 3
> sample(x=1:10, size=8, replace=FALSE) # 8 draws from 1,..,10; no replacem.
[1] 4 6 2 5 1 8 3 7
```

By default, all elements are assumed to have the same probability (1/10 in the above example), but that can be changed with the `prob` option. The next command makes a sample of 12 numbers, all chosen at random from the numbers 2, 4 and 6, with probilities 0.5, 0.4, and 0.1, respectively:

```
> sample(x=c(2,4,6), size=12, replace=TRUE, prob=c(0.5,0.4,0.1))
 [1] 2 2 2 2 4 2 2 2 4 2 2 4
```

For this particular sample, 6 was not selected at all, due to its low probability.

7 Functions

As you have probably already noticed, functions perform almost anything you do in R, and it is worthwhile considering their use a bit more generally.

Arguments A function has a name and one or more named *arguments*. For example, the function `sqrt` has a single argument which you put inside the parentheses following the name of the function: `sqrt(2.4)`. Another example is `dnorm` computing the density in a normal distribution. It has four arguments, named `x`, `mean`, `sd` and `log`.

When you call a function, you write the function name followed by a left and right parenthesis with the arguments, separated by commas, in between. If you include the names of each argument, the arguments can come in any order, whereas you may omit the names if the arguments come in the right order. So the following commands give same values, namely the density of $N(0,1)$ at 0.5:

```
> dnorm(0.5, 0, 1, FALSE)
[1] 0.3520653
> dnorm(sd=1, log=FALSE, x=0.5, mean=0)
[1] 0.3520653
```

You can get to know the arguments and their order with the help function:

```
> help(dnorm)
```

Sometimes you call a function with no arguments. Then you still need the parentheses. For example, `getwd()` gives as result the full path name of the current working directory (see page 3).

Default values of arguments Conveniently, some arguments have *default values*. For example, in `dnorm` the default values of `mean`, `sd` and `log` are 0, 1, and `FALSE`, respectively, so we could as well just have written

```
> dnorm(0.5)
[1] 0.3520653
```

There is no default value of `x`.

Writing your own functions It is sometimes useful to define functions in R yourself. The following commands define functions `f` and g corresponding to $f(x) = x^3 + 4x - 8$ and $g(x) = x^2 + y^2$, respectively, and evaluate $f(3)$ and $g(2,5)$:

```
> f <- function(x) x^3 +4*x - 8
> g <- function(x,y) x^2 + y^2
> f(3)
[1] 31
> g(2,5)
[1] 29
```

Plotting functions It is easy to make graphs of functions with a single argument. For example, the function `f` from above can be plotted on the interval $(-1,5)$ with either `plot` or `curve` as follows:

```
> curve(f, from=-1, to=5)
> plot(f, from=-1, to=5)
```

The default values of `from` and `to` are 0 and 1, respectively, so the function is plotted on $(0,1)$ if no values are specified in the commands. Notice that `plot` can be used to make scatter plots as well as function graphs (and many other things). R uses the arguments to decides which type of plot is appropriate.

A Installation

Installation of R and RStudio Installation of R and RStudio is usually quite unproblematic if you use the default settings along the way. Install R before RStudio. R is installed from the R homepage, `http://www.r-project.org`. Click "CRAN" (top left) and choose Denmark as your CRAN mirror. Then you will find versions for Windows, Mac, and several Linux distributions. RStudio is installed from `http://www.rstudio.com`. Again, there are versions for all platforms. In any case, choose the free version.

Installation and loading of packages Apart from the basic functionalities in R, there exists a huge number of R packages. An R package is a collection of R functions (and datasets). The functions cannot be used before the package is installed and loaded.

In order to install a package, choose Packages in the menu in the bottom, right window. Then click Install and write the name of the package in the pop-up window. Then the package will be installed, and it will appear in the list of installed packages. This needs to be done only once on your computer. The package should be loaded in order to use its functions. Use the Package menu again, and mark the package you want to load.

Alternatively, installation and loading can be carried out as follows. Assume that we need the package called "isdals". It contains datasets used in the book by Ekstrøm and Sørensen (2015). Use the commands

```
> install.packages("isdals")  # Install the foreign package
> library("isdals")           # Loads the foreign package
```

The first command should be run only once; the second in every R session where you need functions or data from the package.

Notice that some packages comes with the base installation, such that they should not be installed seperately. They should still be loaded, though, in order to use the functions or datasets. The MASS package, accompanying the highly recommended book by Venables and Ripley (2002) is an example of such a pre-installed package.

Loading datasats Sometimes datasets from packages should be loaded before they can be used (even if the package itself has been loaded). For example, the `isdals` dataset contains

a dataset named `antibio`. The dataset is not available before the command `data(antibio)` has been used.

B Exercises

Exercise 1 (Get started)

1. Start R
2. Use R to compute $3 \cdot 2 + 8$, 5^3 and $\sqrt{36}$.
3. Assign the value 0.7 to a variable `x`, and write the value of `x` to the screen. Create a new variable `y`, assigning to it the exponential of `x`. Write the value of `y` to the screen. Compute also the natural logarithm of `x` (use the R function `log`).
4. Use the command `help(log)` and find out what the logarithm with base 10 is called in R. Use the function to compute $\log_{10}(100)$ and $\log_{10}(2)$.
5. Make a directory on your computer that you would for files from the course. Go to R and change the working directory to your course directory. Open a new R script.
6. Write a few commands in the R script, and run them in R.
7. Save the file, and close R
8. Open R again, and open the file you wrote just before.

Exercise 2 (Fecundity of crickets)

1. Download the file `crickets.txt` from Absalon and save it in your course directory. Start R and change directory (via the File menu) to the directory where you saved the file.
2. Make an R dataset with `read.table` as in Section 3. Print the dataset on the screen.
3. Try the command `eggs`. Why do you get an error message? Attach the dataset and try the command again.
4. Make a histogram and a boxplot of the `eggs` variable.
5. Compute the sample mean, the sample variance, the sample standard deviation, and the median of the `eggs` variable.

Exercise 3 (Manipulation of datasets) A greenhouse experiment was carried out to examine how the spread of a disease in cucumbers depends on climate and amount of fertilizer (de Neergaard *et al.*, 1993). Two different climates were used: (A) change to day temperature 3 hours before sunrise and (B) normal change to day temperature. Fertilizer was applied in 3 different doses: 2.0, 3.5, and 4.0 units. The amount of infection on standardized plants was recorded after a number of days, and two plants were examined for each combination of climate and dose. The data are available on Absalon in the file `cucumber.txt`.

1. Read the data to a data frame with `read.table`. Name the data frame `cucumber`. Print the data frame on the screen, and make sure that you understand its structure. Compute the mean of the `disease` variable.

2. Use `transform` to make a new variable in the dataset with values equal to the squareroot of the `disease` variable. Compute the mean of the new variable.

3. Use `subset` to make a new dataset that contains only the datalines corresponding to dose larger than 3.

4. Use `subset` to make a new dataset that contains only the datalines corresponding to dose equal to 2.

5. Use `subset` to make a new dataset that contains only the datalines with climate A. Remember that the `climate` variable has character values.

Exercise 4 (Simulation from Bernoulli distribution) Let X be a random variable which is either 0 or 1. The distribution of X is called the Bernoulli distribution, and is given by the probability $p = P(X = 1)$.

1. Explain why the Bernoulli distribution is a special case of the binomial distribution.

2. Make a vector `x` with five simulated values from the Bernoulli distribution with $p = 0.25$. Use the `mean` function to compute the relative frequency of ones among the five values. Explain the relation between the relative frequency and the probability p.

3. Repeat question 2 ten times. Do you get the same relative frequency each time? Why not?

4. Repeat question 2, but now with 500 random numbers. What happens?

Exercise 5 (Simulation from the normal distribution)

1. Make a vector `x` with 200 simulated values from the normal distribution with mean 1 and variance 0.1.

2. Make a histogram of `x`. Does the plot look as you would expect it to do?

3. Compute the mean, the sample standard deviation and the sample variance of `x`. Are the numbers as you would expect them to be?

Exercise 6 (Simulation from the binomial distribution)

1. Make a vector `x` with 500 simulated values from the binomial distribution with 8 trials and probability 0.7.

2. Make a histogram of `x`.

3. Compute the mean and the sample variance of `x`. Are the numbers as you would expect them to be?

4. Try the commands below and explain what the output means:

```
> x[x==6]
> length(x[x==6])
```

5. The command `dbinom(6, size=8, prob=0.7)` computes the probability $P(X = 6)$ if X has a binomial distribution with 8 trials and success probability 0.7. Compare to the result from the previous question.

References

Crawley, M. (2007). *The R Book*. John Wiley & Sons, New York.

de Neergaard, E., Haupt, G., and Rasmussen, K. (1993). Studies of *Didymella bryoniae*: the influence of nutrition and cultural practices on the occurrence of stem lesions and internal and external fruit rot on different cultivars of cucumber. *Netherlands Journal of Plant Pathology*, **99**, 335–343.

Ekstrøm, C. (2012). *The R Primer*. Taylor & Francis group/CRC Press.

Ekstrøm, C. and Sørensen, H. (2015). *Introduction to Statistical Data Analysis for the Life Sciences*. Taylor & Francis group/CRC Press, second edition.

Samuels, M. L. and Witmer, J. A. (2003). *Statistics for the Life Sciences*. Pearson Education International, New Jersey.

Short, T. (2007). R reference card.
`http://cran.r-project.org/doc/contrib/Short-refcard.pdf`.

Venables, W. and Ripley, B. (2002). *Modern Applied Statistics with S*. Springer, fourth edition.