A (very basic) R tutorial

Johannes Karreth

Applied Introduction to Bayesian Data Analysis

1 Getting started

The purpose of this tutorial is to show the very basics of the R language so that participants who have not used R before can complete the first assignment in this workshop. For information on the thousands of other features of R, see the suggested resources below.
In this tutorial, R code that you would enter in your script file or in the command line is preceded by the > character, and by + if the current line of code continues from a previous line. You do not need to type this character in your own code.

1.1 Installing R and RStudio

The most recent version of R for all operating systems is always located at `http://www.r-project.org/index.html`. Go directly to `http://lib.stat.cmu.edu/R/CRAN/`, and download the R version for your operating system. Then, install R.

To operate R, you should rely on writing R scripts. We will write these scripts in RStudio. Download RStudio from `http://www.rstudio.org`. Then, install it on your computer. Some text editors also offer integration with R, so that you can send code directly to R. RStudio is generally the best solution for running R and maintaining a reproducible workflow.

Lastly, install LaTeX in order to compile PDF files from within RStudio. To do this, follow the instructions under `http://www.jkarreth.net/latex.html`, "Installation". You won't have to use LaTeX directly or learn how to write LaTeX code in this workshop.

1.2 Opening RStudio

Upon opening the first time, RStudio will look like Figure 1.
The window on the left is named "Console". The point next to the blue "larger than" sign > is the "command line". You can tell R to perform actions by typing commands into this command line. We will rarely do this and operate R through script files instead.

1.3 R packages

Many useful and important functions in R are provided via packages that need to be installed separately. You can do this by using the Package Installer in the menu (Packages & Data > Package Installer in R or Tools > Install Packages... in RStudio), or by typing

```
> install.packages("foreign")
```

in the R command line. Next, every time you use R, you need to load the packages you want to use: type

```
> library(foreign)
```

in the R command line.

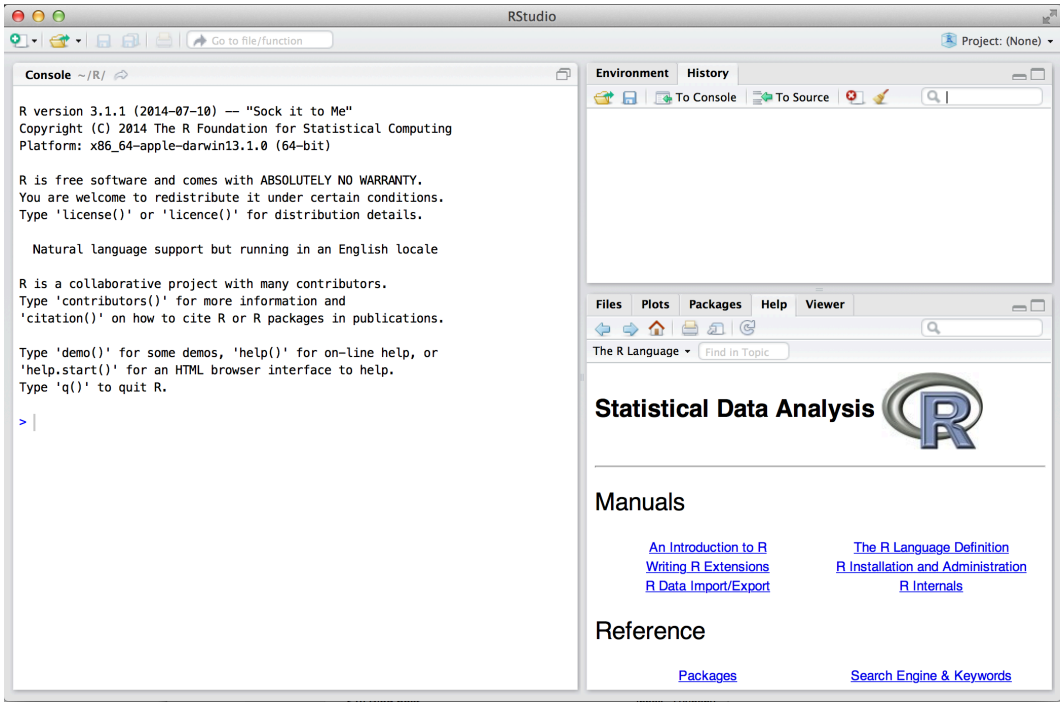


Figure 1: RStudio.

1.4 Working directory

In most cases, it is useful to set a project-specific working directory—especially if you work with graphics that you want to have printed to .pdf or .eps files. You can set the WD with this command:

```
> setwd("/Users/johanneskarreth/Documents/Uni/Teaching/CPH-Bayes/Tutorials/Intro to R")
```

RStudio offers a very useful function to set up a whole project (File > New Project...). Projects automatically create a working directory for you.

1.5 R help

Within R, you can access the help files for any command that exists by typing `?commandname` or, for a list of the commands within a package, by typing `help(package = packagename)`. So, for instance:

```
> ?rnorm
> help(package = "foreign")
```

1.6 Workflows and conventions

There are many resources on how to structure your R workflow (think of routines like the ones suggested by J. Scott Long in *The Workflow of Data Analysis Using Stata*), and I encourage you to search for and maintain a consistent approach to working with R. It will make your life much, much easier—with regards to collaboration, replication, and general efficiency. A few really important points that you might want to consider as you start using R:

- Never type commands into the R command line. Always use a script file, from which you can send (via RStudio, Emacs, ...) commands to R, or at least copy and paste them into R.
- Comment your script files! Comments are indicated by the # sign:

```
> # This is a comment
```

- Save your script files in a project-specific working directory.

- Use a consistent style when writing code. A good place to start is Google's style guide: `http://google-styleguide.googlecode.com/svn/trunk/google-r-style.html`.
- Do not use the `attach()` command.

1.7 Error messages

R will return error messages when a command is incorrect or when it cannot execute a command. Often, these error messages are informative by themselves. You can often get more information by simply searching for an error message on the web. Here, I try to add 1 and the letter a, which does not (yet) make sense:

```
> 1 + a

## Error in eval(expr, envir, enclos):  object 'a' not found
```

As your coding will become more complex, you may forget to complete a particular command. For example, here I want to add 1 and the product of 2 and 4. But I forget to close the parentheses around the product:

```
> 1 + (2 * 4
+ )

## [1] 9
```

You will notice that the little > on the left changes into a +. This means that R is offering you a new line to finish the original command. If I type a right parenthesis, R returns the result of my operation.

1.8 Useful resources

As R has become one of the most popular programs for statistical computing, the number of resources in print and online has increased dramatically. Searching for terms like "introduction to R software" will return a huge number of results.
Some (of the many) good books and e-books that I have encountered and found useful are:

- Fox and Weisberg, *An R and S-Plus Companion to Applied Regression* (2011, print).
- `statmethods.net`. This website offers well-explained computer code to complete most of the data analysis tasks we use in this workshop.
- Maindonald and Braun, *Data Analysis and Graphics Using R* (2006, print).
- Verzani, *simpleR - Using R for Introductory Statistics* (`http://cran.r-project.org/doc/contrib/Verzani-SimpleR.pdf`).

2 R and object-oriented programming

R is an object-based programming language. This means that you, the user, create objects and work with them. Objects can be:

- Numbers:

```
> x <- 1
> x

## [1] 1

> y <- 2
> x + y

## [1] 3

> x * y
```

```
## [1] 2

> x / y

## [1] 0.5

> y^2

## [1] 4

> log(x)

## [1] 0

> exp(x)

## [1] 2.718282
```

- Vectors:

```
> xvec <- c(1, 2, 3, 4, 5)
> xvec

## [1] 1 2 3 4 5

> xvec2 <- seq(from = 1, to = 5, by = 1)
> xvec2

## [1] 1 2 3 4 5

> yvec <- rep(1, 5)
> yvec

## [1] 1 1 1 1 1

> zvec <- xvec + yvec
> zvec

## [1] 2 3 4 5 6
```

- Matrices:

```
> mat1 <- matrix(data = c(1, 2, 3, 4, 5, 6), nrow = 3, byrow = TRUE)
> mat1

##      [,1] [,2]
## [1,]    1    2
## [2,]    3    4
## [3,]    5    6

> mat2 <- matrix(data = seq(from = 6, to = 3.5, by = -0.5),
+                nrow = 2, byrow = T)
> mat2

##      [,1] [,2] [,3]
## [1,]  6.0  5.5  5.0
## [2,]  4.5  4.0  3.5

> mat1 %*% mat2

##      [,1] [,2] [,3]
## [1,]   15 13.5   12
## [2,]   36 32.5   29
## [3,]   57 51.5   46
```

- Data frames (equivalent to data sets):

```
> y <- c(1, 1, 3, 4, 7, 2)
> x1 <- c(2, 4, 1, 8, 19, 11)
> x2 <- c(-3, 4, -2, 0, 4, 20)
> name <- c("Student 1", "Student 2", "Student 3", "Student 4",
+           "Student 5", "Student 6")
> mydata <- data.frame(name, y, x1, x2)
> mydata


##        name y x1 x2
## 1 Student 1 1  2 -3
## 2 Student 2 1  4  4
## 3 Student 3 3  1 -2
## 4 Student 4 4  8  0
## 5 Student 5 7 19  4
## 6 Student 6 2 11 20
```

2.1 Random numbers and distributions

You can use R to generate (random) draws from distributions. This will be important in the first assignment. For instance, to generate 1000 draws from a normal distribution with a mean of 5 and standard deviation of 10, you would write:

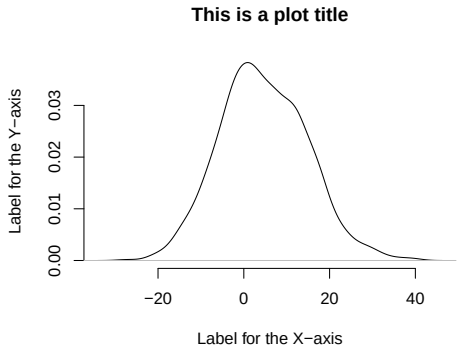
```
> draws <- rnorm(1000, mean = 5, sd = 10)
> summary(draws)

##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
## -26.750  -1.392   5.065   4.951  11.070  34.460
```

You can then use a variety of plotting commands (see for more below) to visualize your draws:

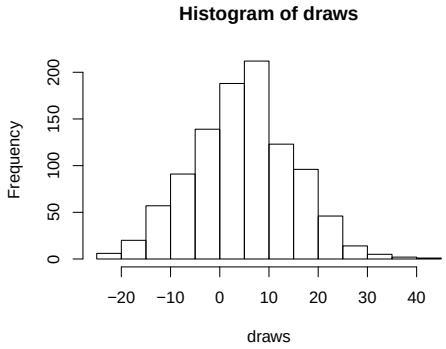
- Density plots:

```
> draws <- rnorm(1000, mean = 5, sd = 10)
> plot(density(draws), main = "This is a plot title",
+      xlab = "Label for the X-axis", ylab = "Label for the Y-axis",
+      frame = FALSE)
```



- Histograms:

```
> draws <- rnorm(1000, mean = 5, sd = 10)
> hist(draws)
```



2.2 Extracting elements from an object

- Elements from a vector:

```
> vec <- c(4, 1, 5, 3)
> vec[3]

## [1] 5
```

- Variables from a data frame:

```
> mydata$x1

## [1]  2  4  1  8 19 11

> mydata$names

## NULL
```

- Columns from a matrix:

```
> mat1[ ,1]

## [1] 1 3 5
```

- Rows from a matrix:

```
> mat1[1, ]

## [1] 1 2
```

- Elements from a list

```
> mylist <- list(x1, x2, y)
> mylist[[1]]

## [1]  2  4  1  8 19 11
```

3 Working with data sets

In most cases, you will not type up your data by hand, but use data sets that were created in other formats. You can easily import such data sets into R. Here are the most common options. Most of them use the foreign package, hence you need to load that package before using these commands:

```
> library(foreign)
```

Note that for each command, many options (in R language: arguments) are available; you will most likely need to work with these options at some time, for instance when your source dataset (e.g., in Stata) has value labels. Check the help files for the respective command in that case.

- **Tables:** If you have a text file with a simple tab-delimited table, where the first line designates variable names:

```
> mydata.table <- read.table("http://www.jkarreth.net/files/data.txt",
+                            header = TRUE)
> head(mydata.table)


##            y         x1         x2
## 1 -0.1629267  1.6535472  0.3001316
## 2  1.3985720  1.4152763 -0.9544489
## 3  0.8983962  0.4199516 -0.4580181
## 4 -1.6484948  0.7212208  0.9356037
## 5  0.2285570 -1.1969352 -1.1368931
```

- **CSV files:** If you have a text file with a simple tab-delimited table, where the first line designates variable names:

```
> mydata.csv <- read.csv("http://www.jkarreth.net/files/data.csv",
+                        header = TRUE)
> head(mydata.csv)


##            y         x1         x2
## 1 -0.1629267  1.6535472  0.3001316
## 2  1.3985720  1.4152763 -0.9544489
## 3  0.8983962  0.4199516 -0.4580181
## 4 -1.6484948  0.7212208  0.9356037
## 5  0.2285570 -1.1969352 -1.1368931
```

- **SPSS files:** If you have an SPSS data file, you can do this:

```
> # mydata.spss <- read.spss("http://www.jkarreth.net/files/data.sav",
> # use.value.labels = TRUE)
```

- **Stata files:** If you have a Stata data file, you can do this:

```
> mydata.dta <- read.dta("http://www.jkarreth.net/files/data.dta",
+                        convert.dates = TRUE, convert.factors = TRUE)
> head(mydata.dta)


##            y         x1         x2
## 1 -0.1629267  1.6535472  0.3001316
## 2  1.3985720  1.4152763 -0.9544489
## 3  0.8983962  0.4199516 -0.4580181
## 4 -1.6484948  0.7212208  0.9356037
## 5  0.2285570 -1.1969352 -1.1368931
```

Describing data

To obtain descriptive statistics of a dataset, or a variable, use the `summary` command:

```
> summary(mydata.dta)

##        y                 x1                x2
##  Min.   :-1.6485   Min.   :-1.1969   Min.   :-1.1369
##  1st Qu.:-0.1629   1st Qu.: 0.4200   1st Qu.:-0.9544
##  Median : 0.2286   Median : 0.7212   Median :-0.4580
##  Mean   : 0.1428   Mean   : 0.6026   Mean   :-0.2627
##  3rd Qu.: 0.8984   3rd Qu.: 1.4153   3rd Qu.: 0.3001
##  Max.   : 1.3986   Max.   : 1.6535   Max.   : 0.9356

> summary(mydata$y)

##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    1.00    1.25    2.50    3.00    3.75    7.00
```

You can access particular quantities, such as standard deviations and quantiles (in this case the 5th and 95th percentiles), with the respective functions:

```
> sd(mydata$y)

## [1] 2.280351

> quantile(mydata$y, probs = c(0.05, 0.95))

##   5%  95%
## 1.00 6.25
```

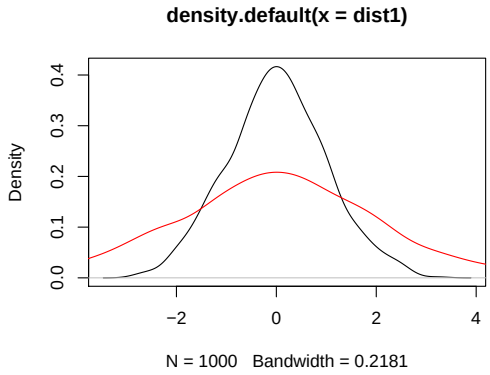
4 Creating figures

R offers several options to create figures. We will work with the so-called "base graphics", mostly using the `plot()` function, and the ggplot2 package.

4.1 Base graphics

R's base graphics are very versatile and, in our workshop, ideal for creating quick plots to inspect objects. These graphs are built sequentially, beginning with the `plot()` command applied to an object. So, for instance to plot the density of 1000 draws from a normal distribution, you would use the following code. I'm using the `set.seed()` command here before every simulation to ensure that the same values are drawn when you try these commands and make these plots.

```
> set.seed(123)
> dist1 <- rnorm(n = 1000, mean = 0, sd = 1)
> set.seed(123)
> dist2 <- rnorm(1000, mean = 0, sd = 2)
> plot(density(dist1))
> lines(density(dist2), col = "red")
```

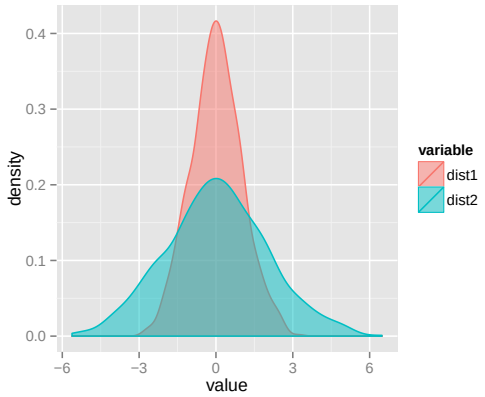


4.2 The ggplot2 package

The ggplot2 package has become popular because its language and plotting sequence can be somewhat more convenient (depending on users' background), especially when working with more complex datasets. For plotting Bayesian model output, ggplot2 offers some useful features.
ggplot2 needs to be first loaded as an external package. Its key commands are `ggplot()` and various plotting commands. All commands are added via +, either in one line or in a new line to an existing ggplot2 object. The command below contains a couple more data manipulation steps that will come in handy for us later; we will discuss them in the workshop. When trying the code below, have a look at the structure of the `dist.dat` object to see what's going on.

```
> library(ggplot2); library(reshape2)
> set.seed(123)
> dist1 <- rnorm(n = 1000, mean = 0, sd = 1)
> set.seed(123)
```

```
> dist2 <- rnorm(1000, mean = 0, sd = 2)
> dist.df <- data.frame(dist1, dist2)
> dist.df <- melt(dist.df)
> normal.plot <- ggplot(data = dist.df, aes(x = value, colour = variable, fill = variable))
> normal.plot <- normal.plot + geom_density(alpha = 0.5)
> normal.plot
```



ggplot2 offers plenty of opportunities for customizing plots; we will also encounter these later on in the workshop. You can also have a look at Winston Chang's *R Graphics Cookbook* for plenty of examples of ggplot2 customization: `http://www.cookbook-r.com/Graphs`.

4.3 Exporting graphs

Plots created via base graphics can be printed to a PDF file using the `pdf()` command. This code:

```
> set.seed(123)
> dist1 <- rnorm(n = 1000, mean = 0, sd = 1)
> set.seed(123)
> dist2 <- rnorm(1000, mean = 0, sd = 2)
> pdf("normal_plot.pdf", width = 5, height = 5)
> plot(density(dist1))
> lines(density(dist2), col = "red")
> dev.off()

## pdf
##   2
```

will print a plot named normal_plot.pdf of the size 5×5 inches to your working directory.
Plots created with ggplot2 are best saved using the `ggsave()` command:

```
> ggsave(plot = normal.plot, filename = "normal_ggplot.pdf", width = 5, height = 5)
```

5 Integrating writing and data analysis

For project management and replication purposes, it is a great idea to combine your data analyis and writing in one framework. RMarkdown, Sweave and knitr are great solutions for this. The RStudio website has a good explanation of these options: `http://rmarkdown.rstudio.com` and `https://support.rstudio.com/hc/en-us/articles/200552056-Using-Sweave-and-knitr`. This tutorial was written using knitr.

A (very) short introduction to R

Paul Torfs & Claudia Brauer

Hydrology and Quantitative Water Management Group

Wageningen University, The Netherlands

3 March 2014

1 Introduction

R is a powerful language and environment for statistical computing and graphics. It is a public domain (a so called "GNU") project which is similar to the commercial S language and environment which was developed at Bell Laboratories (formerly AT&T, now Lucent Technologies) by John Chambers and colleagues. R can be considered as a different implementation of S, and is much used in as an educational language and research tool.

The main advantages of R are the fact that R is freeware and that there is a lot of help available online. It is quite similar to other programming packages such as MatLab (not freeware), but more user-friendly than programming languages such as C++ or Fortran. You can use R as it is, but for educational purposes we prefer to use R in combination with the RStudio interface (also freeware), which has an organized layout and several extra options.

This document contains explanations, examples and exercises, which can also be understood (hopefully) by people without any programming experience. Going through all text and exercises takes about 1 or 2 hours. Examples of frequently used commands and error messages are listed on the last two pages of this document and can be used as a reference while programming.

2 Getting started

2.1 Install R

To install R on your computer (legally for free!), go to the home website of R*:

`http://www.r-project.org/`

and do the following (assuming you work on a windows computer):
- click `download CRAN` in the left bar
- choose a download site
- choose `Windows` as target operation system
- click `base`
- choose `Download R 3.0.3 for Windows` † and choose default answers for all questions

It is also possible to run R and RStudio from a USB stick instead of installing them. This could be useful when you don't have administrator rights on your computer. See our separate note "How to use portable versions of R and RStudio" for help on this topic.

2.2 Install RStudio

After finishing this setup, you should see an "R" icon on you desktop. Clicking on this would start up the standard interface. We recommend, however, to use the RStudio interface. ‡ To install RStudio, go to:

`http://www.rstudio.org/`

and do the following (assuming you work on a windows computer):
- click `Download RStudio`
- click `Download RStudio Desktop`
- click `Recommended For Your System`
- download the `.exe` file and run it (choose default answers for all questions)

2.3 RStudio layout

The RStudio interface consists of several windows (see Figure 1).

- Bottom left: **console window** (also called **command window**). Here you can type simple commands after the ">" prompt and R will then execute your command. This is the most important window, because this is where R actually does stuff.
- Top left: **editor window** (also called **script window**). Collections of commands (scripts) can be edited and saved. When you don't get

*On the R-website you can also find this document: `http://cran.r-project.org/doc/contrib/Torfs+Brauer-Short-R-Intro.pdf`

†At the moment of writing 3.0.3 was the latest version. Choose the most recent one.

‡There are many other (freeware) interfaces, such as Tinn-R.

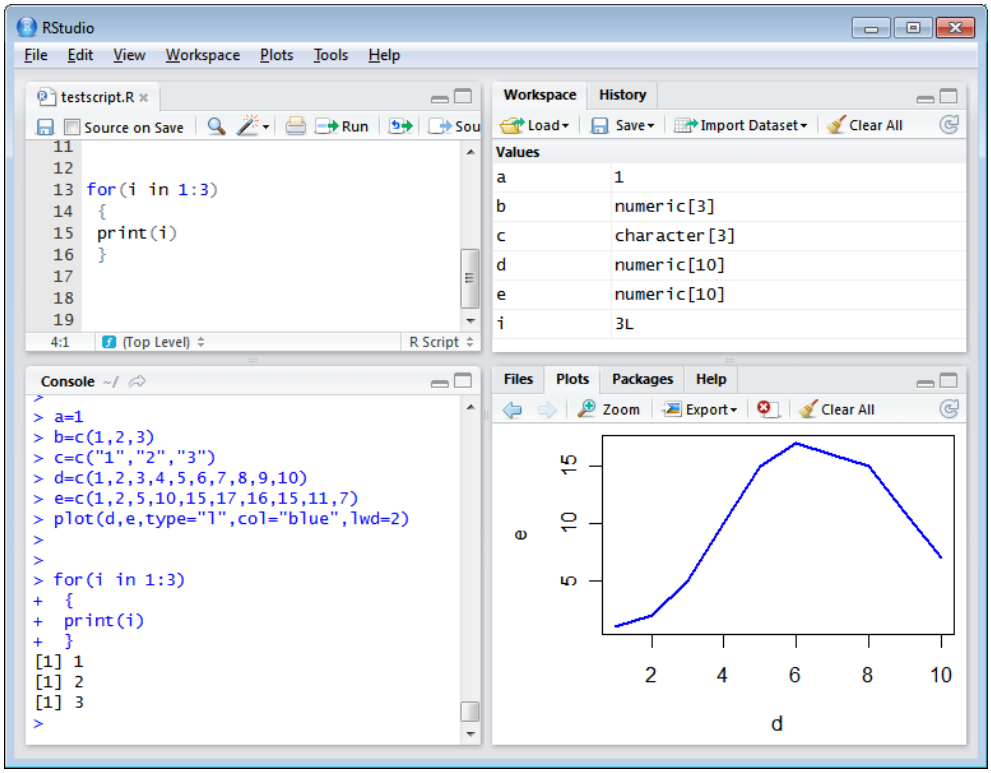


Figure 1 The editor, workspace, console and plots windows in RStudio.

this window, you can open it with `File` → `New` → `R script`
Just typing a command in the editor window is not enough, it has to get into the command window before R executes the command. If you want to run a line from the script window (or the whole script), you can click `Run` or press `CTRL+ENTER` to send it to the command window.

- Top right: **workspace / history window**. In the workspace window you can see which data and values R has in its memory. You can view and edit the values by clicking on them. The history window shows what has been typed before.

- Bottom right: **files / plots / packages / help window**. Here you can open files, view plots (also previous plots), install and load packages or use the help function.

You can change the size of the windows by dragging the grey bars between the windows.

2.4 Working directory

Your *working directory* is the folder on your computer in which you are currently working. When you ask R to open a certain file, it will look in the working directory for this file, and when you tell R to save a data file or figure, it will save it in the working directory.

Before you start working, please set your working directory to where all your data and script files are or should be stored.

Type in the command window: `setwd("directoryname")`. For example:

```
> setwd("M:/Hydrology/R/")
```

Make sure that the slashes are forward slashes and that you don't forget the apostrophes (for the reason of the apostrophes, see section 10.1). R is case sensitive, so make sure you write capitals where necessary.

Within RStudio you can also go to `Tools / Set working directory`.

2.5 Libraries

R can do many statistical and data analyses. They are organized in so-called *packages* or *libraries*. With the standard installation, most common packages are installed.

To get a list of all installed packages, go to the packages window or type `library()` in the console

window. If the box in front of the package name is ticked, the package is loaded (activated) and can be used.

There are many more packages available on the R website. If you want to install and use a package (for example, the package called "geometry") you should:

• Install the package: click `install packages` in the packages window and type `geometry` or type `install.packages("geometry")` in the command window.

• Load the package: check box in front of `geometry` or type `library("geometry")` in the command window.

3 Some first examples of R commands

3.1 Calculator

R can be used as a calculator. You can just type your equation in the command window after the ">":

```
> 10^2 + 36
```

and R will give the answer

```
[1] 136
```

ToDo

Compute the difference between 2014 and the year you started at this university and divide this by the difference between 2014 and the year you were born. Multiply this with 100 to get the percentage of your life you have spent at this university. Use brackets if you need them.

If you use brackets and forget to add the closing bracket, the ">" on the command line changes into a "+". The "+" can also mean that R is still busy with some heavy computation. If you want R to quit what it was doing and give back the ">", press `ESC` (see the reference list on the last page).

3.2 Workspace

You can also give numbers a name. By doing so, they become so-called variables which can be used later. For example, you can type in the command window:

```
> a = 4
```

You can see that `a` appears in the workspace window, which means that R now remembers what `a` is.[§] You can also ask R what `a` is (just type `a` `ENTER` in the command window):

```
> a
[1] 4
```

or do calculations with `a`:

```
> a * 5
[1] 20
```

If you specify `a` again, it will forget what value it had before. You can also assign a new value to `a` using the old one.

```
> a = a + 10
> a
[1] 14
```

To remove all variables from R's memory, type

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

or click "clear all" in the workspace window. You can see that RStudio then empties the workspace window. If you only want to remove the variable `a`, you can type `rm(a)`.

ToDo

Repeat the previous ToDo, but with several steps in between. You can give the variables any name you want, but the name has to start with a letter.

3.3 Scalars, vectors and matrices

Like in many other programs, R organizes numbers in *scalars* (a single number – 0-dimensional), *vectors* (a row of numbers, also called arrays – 1-dimensional) and *matrices* (like a table – 2-dimensional).

The `a` you defined before was a scalar. To define a vector with the numbers 3, 4 and 5, you need the function[¶] `c`, which is short for concatenate (paste together).

```
b=c(3,4,5)
```

Matrices and other 2-dimensional structures will be introduced in Section 6.

[§]Some people prefer te use `<-` instead of `=` (they do the same thing). `<-` consists of two characters, `<` and `-`, and represents an arrow pointing at the object receiving the value of the expression.

[¶]See next Section for the explanation of functions.

3.4 Functions

If you would like to compute the mean of all the elements in the vector `b` from the example above, you could type

```
> (3+4+5)/3
```

But when the vector is very long, this is very boring and time-consuming work. This is why things you do often are automated in so-called *functions*. Some functions are standard in R or in one of the packages. You can also program your own functions (Section 11.3). When you use a function to compute a mean, you'll type:

```
> mean(x=b)
```

Within the brackets you specify the *arguments*. Arguments give extra information to the function. In this case, the argument `x` says of which set of numbers (vector) the mean should computed (namely of `b`). Sometimes, the name of the argument is not necessary: `mean(b)` works as well.

ToDo

Compute the sum of 4, 5, 8 and 11 by first combining them into a vector and then using the function `sum`.

The function `rnorm`, as another example, is a standard R function which creates random samples from a normal distribution. Hit the `ENTER` key and you will see 10 random numbers as:

```
1 > rnorm(10)
2 [1] -0.949  1.342 -0.474 0.403
3 [5] -0.091 -0.379  1.015  0.740
4 [9] -0.639  0.950
```

• Line 1 contains the command: `rnorm` is the function and the 10 is an argument specifying how many random numbers you want — in this case 10 numbers (typing `n=10` instead of just `10` would also work).
• Lines 2-4 contain the results: 10 random numbers organised in a vector with length 10.

Entering the same command again produces 10 new random numbers. Instead of typing the same text again, you can also press the upward arrow key (↑) to access previous commands. If you want 10 random numbers out of normal distribution with mean 1.2 and standard deviation 3.4 you can type

```
> rnorm(10, mean=1.2, sd=3.4)
```

showing that the same function (`rnorm`) may have different interfaces and that R has so called *named arguments* (in this case `mean` and `sd`). By the way, the spaces around the "," and "=" do not matter.

Comparing this example to the previous one also shows that for the function `rnorm` only the first argument (the number 10) is compulsory, and that R gives default values to the other so-called optional arguments.‖

RStudio has a nice feature: when you type `rnorm(` in the command window and press `TAB`, RStudio will show the possible arguments (Fig. 2).

3.5 Plots

R can make graphs. The following is a very simple ** example:

```
1 > x = rnorm(100)
2 > plot(x)
```

• In the first line, 100 random numbers are assigned to the variable `x`, which becomes a vector by this operation.
• In the second line, all these values are plotted in the plots window.

ToDo

Plot 100 normal random numbers.

4 Help and documentation

There is a large amount of (free) documentation and help available. Some help is automatically installed. Typing in the console window the command

```
> help(rnorm)
```

gives help on the `rnorm` function. It gives a description of the function, possible arguments and the values that are used as default for optional arguments. Typing

```
> example(rnorm)
```

‖Use the help function (Sect. 4) to see which values are used as default.
**See Section 7 for slightly less trivial examples.

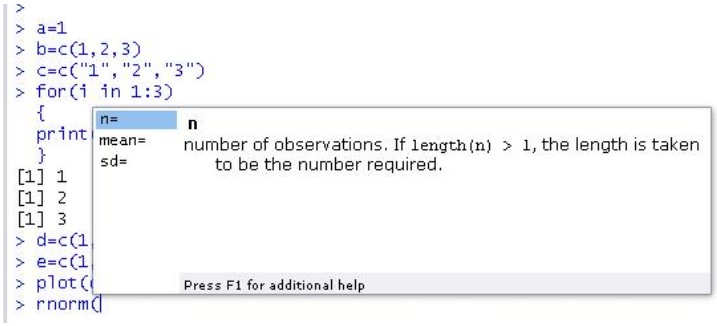


Figure 2 RStudio shows possible arguments when you press `TAB` after the function name and bracket.

gives some examples of how the function can be used.

An HTML-based global help can be called with:

```
> help.start()
```

or by going to the help window.

The following links can also be very useful:

- `http://cran.r-project.org/doc/manuals/R-intro.pdf` A full manual.
- `http://cran.r-project.org/doc/contrib/Short-refcard.pdf` A short reference card.
- `http://zoonek2.free.fr/UNIX/48_R/all.html`

A very rich source of examples.

- `http://rwiki.sciviews.org/doku.php`

A typical user wiki.

- `http://www.statmethods.net/`

Also called Quick-R. Gives very productive direct help. Also for users coming from other programming languages.

- `http://mathesaurus.sourceforge.net/`

Dictionary for programming languages (e.g. R for Matlab users).

- Just using Google (type e.g. "R rnorm" in the search field) can also be very productive.

ToDo

Find help for the `sqrt` function.

5 Scripts

R is an interpreter that uses a command line based environment. This means that you have to type commands, rather than use the mouse and menus. This has the advantage that you do not always have to retype all commands and are less likely to get complaints of arms, neck and shoulders.

You can store your commands in files, the so-called *scripts*. These scripts have typically file names with the extension `.R`, e.g. `foo.R`. You can open an editor window to edit these files by clicking `File` and `New` or `Open file...` [††].

You can run (send to the console window) part of the code by selecting lines and pressing `CTRL+ENTER` or click `Run` in the editor window. If you do not select anything, R will run the line your cursor is on. You can always run the whole script with the console command `source`, so e.g. for the script in the file `foo.R` you type:

```
> source("foo.R")
```

You can also click `Run all` in the editor window or type `CTRL+SHIFT + S` to run the whole script at once.

ToDo

Make a file called `firstscript.R` containing R-code that generates 100 random numbers and plots them, and run this script several times.

6 Data structures

If you are unfamiliar with R, it makes sense to just retype the commands listed in this section. Maybe you will not need all these structures in the beginning, but it is always good to have at least a first glimpse of the terminology and possible applications.

6.1 Vectors

Vectors were already introduced, but they can do more:

[††] Where also the options `Save` and `Save as` are available.

```
1  > vec1 = c(1,4,6,8,10)
2  > vec1
3  [1]  1  4  6  8 10
4  > vec1[5]
5  [1] 10
6  > vec1[3] = 12
7  > vec1
8  [1]  1  4 12  8 10
9  > vec2 = seq(from=0, to=1, by=0.25)
10 > vec2
11 [1] 0.00 0.25 0.50 0.75 1.00
12 > sum(vec1)
13 [1] 35
14 > vec1 + vec2
15 [1]  1.00 4.25 12.50 8.75 11.00
```

• In line 1, a vector `vec1` is explicitly constructed by the concatenation function `c()`, which was introduced before. Elements in vectors can be addressed by standard `[i]` indexing, as shown in lines 4-5.
• In line 6, one of the elements is replaced with a new number. The result is shown in line 8.
• Line 9 demonstrates another useful way of constructing a vector: the `seq()` (sequence) function.
• Lines 10-15 show some typical vector oriented calculations. If you add up two vectors of the same length, the first elements of both vectors are summed, and the second elements, etc., leading to a new vector of length 5 (just like in regular vector calculus). Note that the function `sum` sums up the elements within a vector, leading to one number (a scalar).

6.2 Matrices

Matrices are nothing more than 2-dimensional vectors. To define a matrix, use the function `matrix`:

```
1 mat=matrix(data=c(9,2,3,4,5,6),ncol=3)
2 > mat
3      [,1] [,2] [,3]
4 [1,]    9    3    5
5 [2,]    2    4    6
```

The argument `data` specifies which numbers should be in the matrix. Use either `ncol` to specify the number of columns or `nrow` to specify the number of rows.

ToDo

Put the numbers 31 to 60 in a vector named `P` and in a matrix with 6 rows and 5 columns named `Q`. Tip: use the function `seq`. Look at the different ways scalars, vectors and matrices are denoted in the workspace window.

Matrix-operations are similar to vector operations:

```
1 > mat[1,2]
2 [1] 3
3 > mat[2,]
4 [1] 2 4 6
5 > mean(mat)
6 [1] 4.8333
```

• Elements of a matrix can be addressed in the usual way: `[row,column]` (line 1).
• Line 3: When you want to select a whole row, you leave the spot for the column number empty (the other way around for columns of course).
• Line 5 shows that many functions also work with matrices as argument.

6.3 Data frames

Time series are often ordered in *data frames*. A data frame is a matrix with names above the columns. This is nice, because you can call and use one of the columns without knowing in which position it is.

```
1  > t = data.frame(x = c(11,12,14),
2   y = c(19,20,21), z = c(10,9,7))
3  > t
4     x y z
5  1 11 19 10
6  2 12 20 9
7  3 14 21 7
8  > mean(t$z)
9  [1] 8.666667
10 > mean(t[["z"]])
11 [1] 8.666667
```

• In lines 1-2 a typical data frame called `t` is constructed. The columns have the names `x`, `y` and `z`.
• Line 8-11 show two ways of how you can select the column called `z` from the data frame called `t`.

ToDo

Make a script file which constructs three random normal vectors of length 100. Call these vectors `x1`, `x2` and `x3`. Make a data frame called `t` with three columns (called `a`, `b` and `c`) containing respectively `x1`, `x1+x2` and `x1+x2+x3`. Call the following functions for this data frame: `plot(t)` and `sd(t)`. Can you understand the results? Rerun this script a few times.

6.4 Lists

Another basic structure in R is a *list*. The main advantage of lists is that the "columns" (they're not really ordered in columns any more, but are more a collection of vectors) don't have to be of the same length, unlike matrices and data frames.

```
1  > L = list(one=1, two=c(1,2),
2   five=seq(0, 1, length=5))
3  > L
4  $one
5  [1] 1
6  $two
7  [1] 1 2
8  $five
9  [1] 0.00 0.25 0.50 0.75 1.00
10 > names(L)
11 [1] "one"  "two"  "five"
12 > L$five + 10
13 [1] 10.00 10.25 10.50 10.75 11.00
```

• Lines 1-2 construct a list by giving names and values. The list also appears in the workspace window.
• Lines 3-9 show a typical printing (after pressing `L ENTER`).
• Line 10 illustrates how to find out what's in the list.
• Line 12 shows how to use the numbers.

7 Graphics

Plotting is an important statistical activity. So it should not come as a surprise that R has many plotting facilities. The following lines show a simple plot:

```
> plot(rnorm(100), type="l", col="gold")
```

Hundred random numbers are plotted by connecting the points by lines (the symbol between quotes after the `type=`, is the letter l, not the number 1) in a gold color.

Another very simple example is the classical statistical histogram plot, generated by the simple command

```
> hist(rnorm(100))
```

which generates the plot in Figure 3.

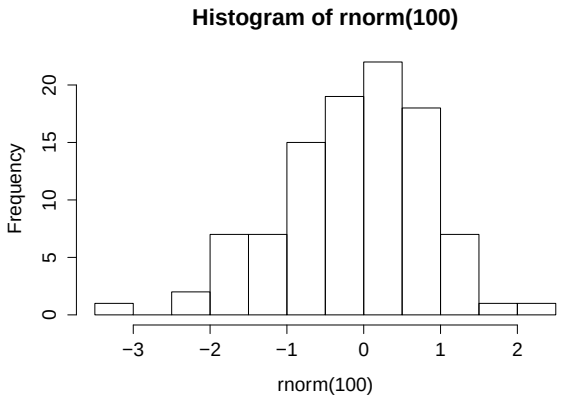


Figure 3 A simple histogram plot.

The following few lines create a plot using the data frame `t` constructed in the previous ToDo:

```
1 plot(t$a, type="l", ylim=range(t),
2  lwd=3, col=rgb(1,0,0,0.3))
3 lines(t$b, type="s", lwd=2,
4  col=rgb(0.3,0.4,0.3,0.9))
5 points(t$c, pch=20, cex=4,
6  col=rgb(0,0,1,0.3))
```

ToDo

Add these lines to the script file of the previous section. Try to find out, either by experimenting or by using the help, what the meaning is of `rgb`, the last argument of `rgb`, `lwd`, `pch`, `cex`.

To learn more about formatting plots, search for `par` in the R help. Google "R color chart" for a pdf file with a wealth of color options.

To copy your plot to a document, go to the plots window, click the "Export" button, choose the nicest width and height and click `Copy` or `Save`.

8 Reading and writing data files

There are many ways to write data from within the R environment to files, and to read data from files. We will illustrate one way here. The following lines illustrate the essential:

```
1  > d = data.frame(a = c(3,4,5),
2   b = c(12,43,54))
3  > d
4    a  b
5  1 3 12
6  2 4 43
7  3 5 54
8  > write.table(d, file="tst0.txt",
9   row.names=FALSE)
10 > d2 = read.table(file="tst0.txt",
11  header=TRUE)
12 > d2
13   a  b
14 1 3 12
15 2 4 43
16 3 5 54
```

• In lines 1-2, a simple example data frame is constructed and stored in the variable `d`.
• Lines 3-7 show the content of this data frame: two columns (called `a` and `b`), each containing three numbers.
• Line 8 writes this data frame to a text file, called `tst0.txt` The argument `row.names=FALSE` prevents that row names are written to the file. Because nothing is specified about `col.names`, the default option `col.names=TRUE` is chosen and column names are written to the file. Figure 4 shows the resulting file (opened in an editor, such as Notepad), with the column names (`a` and `b`) in the first line.
• Lines 10-11 illustrate how to read a file into a data frame. Note that the column names are also read. The data frame also appears in the workspace window.

ToDo

Make a file called `tst1.txt` in Notepad from the example in Figure 4 and store it in your working directory. Write a script to read it, to multiply the column called `g` by 5 and to store it as `tst2.txt`.

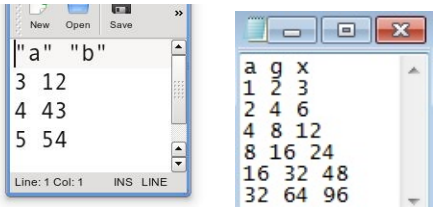


Figure 4 The files `tst0.txt` of section 8 (left) and `tst1.txt` from the ToDo below (right) opened in two text editors.

9 Not available data

ToDo

Compute the mean of the square root of a vector of 100 random numbers. What happens?

When you work with real data, you will encounter missing values because instrumentation failed or because you didn't want to measure in the weekend. When a data point is *not available*, you write `NA` instead of a number.

```
> j = c(1,2,NA)
```

Computing statistics of incomplete data sets is strictly speaking not possible. Maybe the largest value occurred during the weekend when you didn't measure. Therefore, R will say that it doesn't know what the largest value of `j` is:

```
> max(j)
[1] NA
```

If you don't mind about the missing data and want to compute the statistics anyway, you can add the argument `na.rm=TRUE` (Should I remove the NAs? Yes!).

```
> max(j, na.rm=TRUE)
[1] 2
```

10 Classes

The exercises you did before were nearly all with numbers. Sometimes you want to specify something which is not a number, for example the name of a measurement station or data file. In that case you want the variable to be a character string instead of a number.

An object in R can have several so-called *classes*. The most important three are *numeric*, *character* and *POSIX* (date-time combinations). You can ask R what class a certain variable is by typing `class(...)`.

10.1 Characters

To tell R that something is a character string, you should type the text between apostrophes, otherwise R will start looking for a defined variable with the same name:

```
> m = "apples"
> m
[1] "apples"
> n = pears
Error: object 'pears' not found
```

Of course, you cannot do computations with character strings:

```
> m + 2
Error in m + 2 : non-numeric argument to
binary operator
```

10.2 Dates

Dates and times are complicated. R has to know that 3 o'clock comes after 2:59 and that February has 29 days in some years. The easiest way to tell R that something is a date-time combination is with the function `strptime`:

```
1 > date1=strptime( c("20100225230000",
2  "20100226000000", "20100226010000"),
3  format="%Y%m%d%H%M%S")
4  > date1
5 [1] "2010-02-25 23:00:00"
6 [2] "2010-02-26 00:00:00"
7 [3] "2010-02-26 01:00:00"
```

• In lines 1-2 you create a vector with `c(...)`. The numbers in the vectors are between apostrophes because the function `strptime` needs character strings as input.
• In line 3 the argument `format` specifies how the character string should be read. In this case the year is denoted first (%Y), then the month (%m), day (%d), hour (%H), minute (%M) and second (%S). You don't have to specify all of them, as long as the format corresponds to the character string.

ToDo

Make a graph with on the x-axis: today, Sinterklaas 2014 and your next birthday and on the y-axis the number of presents you expect on each of these days. Tip: make two vectors first.

11 Programming tools

When you are building a larger program than in the examples above or if you're using someone else's scripts, you may encounter some programming statements. In this Section we describe a few tips and tricks.

11.1 If-statement

The *if-statement* is used when certain computations should *only* be done when a certain condition is met (and maybe something else should be done when the condition is not met). An example:

```
1 > w = 3
2 > if( w < 5 )
3    {
4    d=2
5    }else{
6    d=10
7    }
8 > d
9 2
```

• In line 2 a condition is specified: `w` should be less than 5.
• If the condition is met, R will execute what is between the first brackets in line 4.
• If the condition is *not* met, R will execute what is between the second brackets, after the `else` in line 6. You can leave the `else{...}`-part out if you don't need it.
• In this case, the condition is met and `d` has been assigned the value 2 (lines 8-9).

To get a subset of points in a vector for which a certain condition holds, you can use a shorter method:

```
1 > a = c(1,2,3,4)
2 > b = c(5,6,7,8)
3 > f = a[b==5 | b==8]
4 > f
5 [1] 1 4
```

• In line 1 and 2 two vectors are made.
• In line 3 you say that `f` is composed of those elements of vector `a` for which `b` equals 5 or `b` equals 8.

Note the double `=` in the condition. Other conditions (also called logical or Boolean operators) are `<`, `>`, `!=` ($\neq$), `<=` ($\leq$) and `>=` ($\geq$). To test more than one condition in one if-statement, use `&` if both conditions have to be met ("and") and `|` if one of the conditions has to be met ("or").

11.2 For-loop

If you want to model a time series, you usually do the computations for one time step and then for the next and the next, etc. Because nobody wants to type the same commands over and over again, these computations are automated in for-loops.

In a *for-loop* you specify what has to be done and how many times. To tell "how many times", you specify a so-called counter. An example:

```
1 > h = seq(from=1, to=8)
2 > s = c()
3 > for(i in 2:10)
4    {
5    s[i] = h[i] * 10
6    }
7 > s
8 [1] NA 20 30 40 50 60 70 80 NA NA
```

• First the vector `h` is made.
• In line 2 an empty vector (`s`) is created. This is necessary because when you introduce a variable within the for-loop, R will not remember it when it has gotten out of the for-loop.
• In line 3 the for-loop starts. In this case, `i` is the counter and runs from 2 to 10.
• Everything between the curly brackets (line 5) is processed 9 times. The first time `i=2`, the second element of `h` is multiplied with 10 and placed in the second position of the vector `s`. The second time `i=3`, etc. In the last two runs, the 9^{th} and 10^{th} elements of `h` are requested, which do not exist. Note that these statements are evaluated without any explicit error messages.

ToDo

Make a vector from 1 to 100. Make a for-loop which runs through the whole vector. Multiply the elements which are smaller than 5 and larger than 90 with 10 and the other elements with 0.1.

11.3 Writing your own functions

Functions you program yourself work in the same way as pre-programmed R functions.

```
1 > fun1 = function(arg1, arg2 )
2    {
3    w = arg1 ^ 2
4    return(arg2 + w)
5    }
6 > fun1(arg1 = 3, arg2 = 5)
7 [1] 14
8
```

• In line 1 the function name (`fun1`) and its arguments (`arg1` and `arg2`) are defined.
• Lines 2-5 specify what the function should do if it is called. The return value (`arg2+w`) is shown on the screen.
• In line 6 the function is called with arguments 3 and 5.

ToDo

Write a function for the previous ToDo, so that you can feed it any vector you like (as argument). Use a for-loop in the function to do the computation with each element. Use the standard R function `length` in the specification of the counter. [a])

[a] Actually, people often use more for-loops than necessary. The ToDo above can be done more easily and quickly without a for-loop but with regular vector-computations.

12 Some useful references

12.1 Functions

This is a subset of the functions explained in the R reference card.

Data creation
• `read.table`: read a table from file. Arguments: `header=TRUE`: read first line as titles of the columns; `sep=","`: numbers are separated by commas; `skip=n`: don't read the first `n` lines.
• `write.table`: write a table to file
• `c`: paste numbers together to create a vector
• `array`: create a vector, Arguments: `dim`: length
• `matrix`: create a matrix, Arguments: `ncol` and/or `nrow`: number of rows/columns
• `data.frame`: create a data frame
• `list`: create a list
• `rbind` and `cbind`: combine vectors into a matrix by row or column

Extracting data
• `x[n]`: the n^{th} element of a vector
• `x[m:n]`: the m^{th} to n^{th} element
• `x[c(k,m,n)]`: specific elements
• `x[x>m & x<n]`: elements between `m` and `n`
• `x$n`: element of list or data frame named `n`
• `x[["n"]]`: idem
• `[i,j]`: element at i^{th} row and j^{th} column
• `[i,]`: row `i` in a matrix

Information on variables
• `length`: length of a vector
• `ncol` or `nrow`: number of columns or rows in a matrix
• `class`: class of a variable
• `names`: names of objects in a list
• `print`: show variable or character string on the screen (used in scripts or for-loops)
• `return`: show variable on the screen (used in functions)
• `is.na`: test if variable is `NA`
• `as.numeric` or `as.character`: change class to number or character string
• `strptime`: change class from character to date-time (POSIX)

Statistics
• `sum`: sum of a vector (or matrix)
• `mean`: mean of a vector
• `sd`: standard deviation of a vector
• `max` or `min`: largest or smallest element
• `rowSums` (or `rowMeans`, `colSums` and `colMeans`): sums (or means) of all numbers in each row (or column) of a matrix. The result is a vector.
• `quantile(x,c(0.1,0.5))`: sample the 0.1 and 0.5^{th} quantiles of vector `x`

Data processing
• `seq`: create a vector with equal steps between the numbers
• `rnorm`: create a vector with random numbers with normal distribution (other distributions are also available)
• `sort`: sort elements in increasing order
• `t`: transpose a matrix
• `aggregate(x,by=ls(y),FUN="mean")`: split data set `x` into subsets (defined by `y`) and computes means of the subsets. Result: a new list.
• `na.approx`: interpolate (in `zoo` package). Argument: vector with `NA`s. Result: vector without `NA`s.
• `cumsum`: cumulative sum. Result is a vector.
• `rollmean`: moving average (in the `zoo` package)
• `paste`: paste character strings together
• `substr`: extract part of a character string

Fitting
• `lm(v1~v2)`: linear fit (regression line) between vector `v1` on the y-axis and `v2` on the x-axis
• `nls(v1~a+b*v2, start=ls(a=1,b=0))`: non-linear fit. Should contain equation with variables (here `v1` and `v2` and parameters (here `a` and `b`) with starting values
• `coef`: returns coefficients from a fit
• `summary`: returns all results from a fit

Plotting
• `plot(x)`: plot `x` (y-axis) versus index number (x-axis) in a new window
• `plot(x,y)`: plot `y` (y-axis) versus `x` (x-axis) in a new window
• `image(x,y,z)`: plot `z` (color scale) versus `x` (x-axis) and `y` (y-axis) in a new window
• `lines` or `points`: add lines or points to a previous plot
• `hist`: plot histogram of the numbers in a vector
• `barplot`: bar plot of vector or data frame
• `contour(x,y,z)`: contour plot
• `abline`: draw line (segment). Arguments: `a,b` for intercept `a` and slope `b`; or `h=y` for horizontal line at `y`; or `v=x` for vertical line at `x`.
• `curve`: add function to plot. Needs to have an

x in the expression. Example: curve(x^2)
• legend: add legend with given symbols (lty or pch and col) and text (legend) at location (x="topright")
• axis: add axis. Arguments: side – 1=bottom, 2=left, 3=top, 4=right
• mtext: add text on axis. Arguments: text (character string) and side
• grid: add grid
• par: plotting parameters to be specified before the plots. Arguments: e.g. mfrow=c(1,3)): number of figures per page (1 row, 3 columns); new=TRUE: draw plot over previous plot.

Plotting parameters
These can be added as arguments to plot, lines, image, etc. For help see par.
• type: "l"=lines, "p"=points, etc.
• col: color – "blue", "red", etc
• lty: line type – 1=solid, 2=dashed, etc.
• pch: point type – 1=circle, 2=triangle, etc.
• main: title - character string
• xlab and ylab: axis labels – character string
• xlim and ylim: range of axes – e.g. c(1,10)
• log: logarithmic axis – "x", "y" or "xy"

Programming
• function(arglist){expr}: function definition: do expr with list of arguments arglist
• if(cond){expr1}else{expr2}: if-statement: if cond is true, then expr1, else expr2
• for(var in vec) {expr}: for-loop: the counter var runs through the vector vec and does expr each run
• while(cond){expr}: while-loop: while cond is true, do expr each run

12.2 Keyboard shortcuts

There are several useful keyboard shortcuts for RStudio (see Help → Keyboard Shortcuts):
• CRL+ENTER: send commands from script window to command window
• ↑ or ↓ in command window: previous or next command
• CTRL+1, CTRL+2, etc.: change between the windows

Not R-specific, but very useful keyboard shortcuts:
• CTRL+C, CTRL+X and CTRL+V: copy, cut and paste
• ALT+TAB: change to another program window
• ↑, ↓, ← or →: move cursor
• HOME or END: move cursor to begin or end of line
• Page Up or Page Down: move cursor one page up or down
• SHIFT+↑/↓/←/→/HOME/END/PgUp/PgDn: select

12.3 Error messages

• No such file or directory or Cannot change working directory
Make sure the working directory and file names are correct.
• Object 'x' not found
The variable x has not been defined yet. Define x or write apostrophes if x should be a character string.
• Argument 'x' is missing without default
You didn't specify the compulsory argument x.
• +
R is still busy with something or you forgot closing brackets. Wait, type } or) or press ESC.
• Unexpected ')' in ")" or Unexpected '}' in "}"
The opposite of the previous. You try to close something which hasn't been opened yet. Add opening brackets.
• Unexpected 'else' in "else"
Put the else of an if-statement on the same line as the last bracket of the "then"-part: }else{.
• Missing value where TRUE/FALSE needed
Something goes wrong in the condition-part (if(x==1)) of an if-statement. Is x NA?
• The condition has length > 1 and only the first element will be used
In the condition-part (if(x==1)) of an if-statement, a vector is compared with a scalar. Is x a vector? Did you mean x[i]?
• Non-numeric argument to binary operator
You are trying to do computations with something which is not a number. Use class(...) to find out what went wrong or use as.numeric(...) to transform the variable to a number.
• Argument is of length zero or Replacement is of length zero
The variable in question is NULL, which means that it is empty, for example created by c(). Check the definition of the variable.

list of some useful R functions

Charles DiMaggio

February 27, 2013

1 help

- *help()* opens help page (same as *?topic*)
- *apropos()*displays all objects matching topic (same as *??topic*)
- *library(help=packageName)* help on a specific package
- *example()* ; *demo()*
- *vignette(package="packageName")*; *vignette(package="topic")*
- RSiteSearch("packageName")
- ?NA - handling missing data values
- args() - arguments for a function
- *functionName* - just writing the name of the function returns the function source code
- help with math:
  - ?Control - Help on control flow statements (e.g. if, for, while)
  - ?Extract - Help on operators acting to extract or replace subsets of vectors
  - ?Logic - Help on logical operators
  - ?regex - Help on regular expressions used in R
  - ?Syntax - Help on R syntax and giving the precedence of operators

2 General

- append() - add elements to a vector
- cbind() - Combine vectors by row/column
- grep() - regular expressions

- identical() - test if 2 objects are exactly equal
- length() - no. of elements in vector
- ls() - list objects in current environment
- range(x) - minimum and maximum
- rep(x,n) - repeat the number x, n times
- rev(x) - elements of x in reverse order
- seq(x,y,n) - sequence (x to y, spaced by n)
- sort(x) - sort the vector x
- order(x) - list the sorted *element numbers* of x
- tolower(),toupper() - Convert string to lower/upper case letters
- unique(x) - remove duplicate entries from vector
- round(x), signif(x), trunc(x) - rounding functions
- getwd() - return working directory
- setwd() - set working directory
- choose.files() - get path to a file (useful for virtual machines)
- month.abb/month.name - abbreviated and full names for months
- pi,letters,(e.g. letters[7] = "g") LETTERS

3 Math

- sqrt(),sum()
- log(x),log10(),,exp(),sqrt()
- cos(),sin(),tan(),
- %% modulus
- %/% integer divisilln
- %*% matrix multiplication
- %o% outer product (a%o% equivalent to outer(a,b,"*"))
- union(),intersect(),setdiff(),setequal() - set operations
- eigen() - eigenvalues and eigenvectors
- deriv() - symbolic and algorithmic derivatives of simple expressions

- integrate() - adaptive quadrature over a finite or infinite interval.

4 Plotting

- plot() - generic R object plotting
- par() - set or query graphical parameters
- curve(equation,add=T) - plot an equation as a curve
- points(x,y) - add additional set of points to an existing graph
- arrows() - draw arrows
- abline() - ddd a straight line to an existing graph
- lines() - join specified points with line segments
- segments() - draw line segments between pairs of points
- hist() - histogram
- pairs() - plot matrix of scatter plots
- matplot() - plot columns of matrices
- persp() - perspective plot
- contour() - contour plot
- image() - plot an image file
- loess(), lowess() - scatter plot smoothing
- splinefun() - spline interpolation
- smooth.spline() - Fits a cubic smoothing spline
- jitter() - Add a small amount of noise to a numeric vector
- pdf()/ png() / jpeg() - send plot to .pdf / .png / .jpeg file

5 Statistics

- help(package=stats) - list all stats functions
- lm - fit linear model
- glm - fit generalized linear model
- cor.test() - correlation test

- cumsum() cumprod() - cumuluative functions for vectors
- density(x) - kernel density estimates
- ks.test() - one or two sample Kolmogorov-Smirnov tests
- mean(x), weighted.mean(x), median(x), min(x), max(x), quantile(x)
- rnorm(), runif() - generate random data with Gaussian/uniform distribution
- sd() - standard deviation
- summary(x) - a summary of x (mean, min, max)
- t.test() - Student's t-test
- var() - variance
- sample() - random samples
- qqplot() - quantile-quantile plot

6 regression

(Functions in italics, packages in quotation marks.)

- Linear models
  - *aov* ("stats"), *Anova()* ("car"): ANOVA models
  - *coef*: extract model coefficients ("stats")
  - *confint*: Computes confidence intervals for one or more parameters in a fitted model. ("stats")
  - *fitted*: extracts fitted values ("stats")
  - *lm*: fit linear models. ("stats")
  - *model.matrix*: creates a design matrix ("stats")
  - *predict*: predicted values based on linear model object ("stats")
  - *residuals*: extracts model residuals ("stats")
  - *summary* summary method for class "lm" (stats)
  - *vcov*: variance-covariance matrix of the main parameters of a fitted model object ("stats")
  - *AIC*: Akaike information criterion for one or several fitted model objects ("stats")
  - *extractAIC*: Computes the (generalized) Akaike An Information Criterion for a fitted parametric model ("stats")

  - *offset*: An offset is a term to be added to a linear predictor, such as in a generalised linear model

- Generalized Linear Models (GLM)
  - *glm*: is used to fit generalized linear models ("stats")

"family=" specify the details of the models used by *glm* ("stats")

  - *glm.nb*: fit a negative binomial generalized linear model ("MASS")

- Diagnostics
  - *cookd*: cook's distances for linear and generalized linear models ("car") "cooks.distance": Cooks distance ("stats")
  - *influence.measures*: suite of functions to compute regression (leave-one-out deletion) diagnostics for linear and generalized linear models ("stats")
  - *lm.influence*: provides the basic quantities used in diagnostics for checking the quality of regression fits ("stats")
  - *outlier.test*: Bonferroni outlier test ("car")
  - *rstandard*: standardized residuals ("stats")
  - *rstudent*: studentized residuals ("stats")
  - *vif*: variance inflation factor ("car")

- Graphics
  - *influence.plot*: regression influence plot ("car")
  - *leverage.plots*: regression leverage plots ("car")
  - *plot*: four residual plots ("stats")
  - *qq.plot*: quantile-comparison plots ("car")
  - *qqline*: adds a line to a normal quantile-quantile plot which passes through the first and third quartiles ("stats")
  - *qqnorm*: normal QQ plot of the values in y ("stats")
  - *reg.line*: plot regression line ("car")
  - *scatterplot*: scatterplots with boxplots ("car")

- Tests and Transformations
  - *durbin.watson*: Durbin-Watson Test for autocorrelated errors ("car")
  - *dwtest*: Durbin-Watson test ("lmtest")
  - *levene.test*: Levene's test ("car")
  - *lillie.test*: Lilliefors (Kolmogorov-Smirnov) test for normality ("nortest")

- *pearson.test*: Pearson chi-square test for normality ("nortest")
- *box.cox*: Box-Cox family of transformations ("car")
- *boxcox*: Box-Cox transformations for linear models ("MASS")

- Survival analysis
  - *anova.survreg*: ANOVA tables for survreg objects ("survival")
  - *clogit*: Conditional logistic regression ("survival")
  - *cox.zph*: Test the proportional hazards assumption of a Cox regression ("survival")
  - *coxph*: proportional hazards regression ("survival")
  - *coxph.detail*: details of a Cox model fit ("survival")
  - *coxph.rvar*: robust variance for a Cox model ("survival")
  - *ridge*: ridge regression ("survival")
  - *survdiff*: test survival curve differences ("survival")
  - *survexp*: compute expected survival ("survival")
  - *survfit*: compute a survival curve for censored data ("survival")
  - *survreg*: regression for a parametric survival model ("survival")

- Linear and nonlinear mixed effects models
  - *ACF*: autocorrelation function ("nlme")
  - *ACF.lme*: autocorrelation Function for lme Residuals ("nlme")
  - *intervals*: confidence intervals on coefficients ("nlme")
  - *intervals.lme*: confidence intervals on lme parameters ("nlme")
  - *lme*: linear mixed-effects models ("nlme")
  - *nlme*: nonlinear mixed-effects models ("nlme")
  - *predict.lme*: predictions from an lme object ("nlme")
  - *predict.nlme*: predictions from an nlme object ("nlme")
  - *qqnorm.lme*: normal plot of residuals or random effects from an lme object ("nlme")
  - *ranef.lme*: extract lme random effects ("nlme")
  - *residuals.lme*: extract lme residuals ("nlme")
  - *simulate.lme*: simulate lme models ("nlme")
  - *summary.lme*: summarize an lme object ("nlme")

- Structural Equation, Principal Components, Partial Least Squares Regression Models
  - *sem*: general structural equation models ("sem")
  - *systemfit*: fits a set of linear structural equations using ordinary least squares
  - *biplot.mvr*: biplots of PLSR and PCR Models ("pls")
  - *coefplot*: plot regression coefficients of plsr and pcr models ("pls")
  - *mvr*: partial least squares and principal components regression ("pls")
  - *scores*: extract scores and loadings from plsr and pcr models ("pls")
- Recursive Partitioning and Regression Trees
  - *cv.tree*: cross-validation for choosing tree complexity ("tree")
  - *deviance.tree*: extract deviance from a tree object ("tree")
  - *labels.rpart*: create split labels for an rpart object ("rpart")
  - *misclass.tree*: misclassifications by a classification tree ("tree")
  - *partition.tree*: plot the partitions of a simple tree model ("tree")
  - *path.rpart*: follow paths to selected nodes of an rpart object (rpart)
  - *plotcp*: plot a complexity parameter table for an rpart fit ("rpart")
  - *printcp*: displays cp table for fitted rpart object ("rpart")
  - *prune.misclass*: cost-complexity pruning of tree by error rate ("tree")
  - *rpart*: recursive partitioning and regression trees ("rpart")
  - *rsq.rpart*: plots the approximate r-square for the different splits ("rpart")
  - *tile.tree*: add class barplots to a classification tree plot ("tree")
  - *tree.control*: select parameters for tree (tree)
  - *tree.screens*: split screen for plotting trees ("tree")
  - *tree*: fit a classification or regression tree ("tree")

This list is based on material posted online by Alastair Sanderson and Vito Ricci.