*R Programming for Demographers*

# Basic R

Tim Riffe*

September 19, 2017

## Contents



## 1 Statistical Programming

### Requirements and Aims:

- In advance: no programming experience or advanced statistical knowledge is needed
- Afterward: feeling comfortable with simple statistical programming

### What is Statistical Programming?

In general statistical programming involves controlling computers, telling them what calculations to do, what to display and doing computations to aid in statistical analysis. Practically it is the contrary of a "click-and-tick" software, and you are supposed to know what you are going to do.

- More in detail:
  - summarizing and displaying data
  - fitting models to the data
  - displaying results

*First, a big thanks to Giancarlo Camarda, who has let me recycle his impeccable material for teaching this course. Higher order thanks are also due to the very same people that Giancarlo himself thanked, especially Sabine Zinn and Roland Rau. Sabine was my teacher in this very course in 2009 (cohort 5)! Part of these lectures has been freely inspired by courses that both Roland, Sabine, and Giancarlo taught in the past. Furthermore, in recent years, Adam Lenart, Fernando Colchero, and Silvia Rizzi have aided in teaching this course and improving handouts. On the other hand, it goes without saying that I am uniquely responsible for any deficiencies and mistakes you may find in handouts.

- Advanced:
  - implementation of (sophisticated) statistical methods
  - simulation of stochastic models (the real world is full of randomness)
- Statistical programming is closely related to numerical programming. It involves
  - (complex) computations that have to be done numerically
  - optimization
  - approximation of mathematical functions

## 1.1 Why R?

There are a number of different statistical software tools you can use for statistical computing (Microsoft EXCEL, SAS, SPSS, S-Plus, Stata, MATLAB, R, etc.). Statistical programming can be performed by most of these.

The following questions may hence arise:

- How much time does it take install and use a software tool?
- How is the user interface?
- What kind of methods and functions are used within a software tool?
- How are these methods and functions are implemented?
- Are implemented methods modifiable?
- Does a software tool offer the option to implement your own methods and functions (easily)?
- . . .
- How much does it cost?
- How widely used is the software? Does it have a user community?

What exactly is R? Why is `R` useful for us?

- `R` is a statistical software tool that allows for statistical programming within a wide range of applications.
- `R` provides a broad spectrum of statistical and graphical techniques in a large number of packages.
- With `R` it is possible to program your own methods for individual jobs, or for widespread sharing.
- It is also possible to integrate already implemented statistical methods in own programs.
- `R` is open source software, which means that it is software whose source code is available under a license that permits users to use, change, and improve the software, and to redistribute it in modified or unmodified form. You can download `R` from `http://www.r-project.org/`.

`R` is a command-driven software tool, so you type in text and ask `R` to execute it. Nowadays most statistical software tools provide an interactive graphical user interface with menus, etc. So, is `R` an old-fashioned tool? No, it is not. Here are some reasons.

- Menu-based software tools are very convenient when only a limited set of commands should be executed. A command-line software tool is instead open ended.
- It is rather simple to write some pieces of code for a new problem nobody has faced before and execute them within `R` . (This is also possible with some menu-driven software tools. However, it is much easier in a command-driven tool.)
- Command-line software tools allow the user to be more independent from any particular organization structure of a tool.

## 2 Basic operations and assignment

The simplest use of R is as a calculator. Once you open the program, you can type a simple calculation next to the greater-than symbol > in the console. The result is given upon pressing the **Enter** key:

```
5+4

## [1] 9
```

You can see that R provides the outcomes just in the next line. The number in square-brackets [1] indicates that this is the first result from the command. For longer outcomes, each line of results will be labeled accordingly to aid the user in scanning quickly though the output.

Though not particularly common in R programming, multiple calculations could be arranged within the same line by separating them with a semi-colon:

```
4*8 ; 21/6 ; 5^3

## [1] 32
## [1] 3.5
## [1] 125
```

It is intuitive that + stands for a sum, * is the symbol for multiplication, ^ is the power operator, etc.

Basic operators as well as more complex functions are available. For instance we can compute modular arithmetic. For instance, a life-time of 6000 days is equal to

```
6000%/%365 ; 6000%%365

## [1] 16
## [1] 160
```

i.e. 16 years and 160 days. First we take the integer division between 6000 and 365, assuming that a year is always 365 days long. Then we compute the remainder after division: 6000 (mod 365).

Of course we cannot write all our ideas in a single line. R has a workspace that can be used to store the results of calculations, and many other kinds of objects. As a first example, suppose we aim to store the result of a basic demographic concept: the arithmetic growth rate. We take the population in Sweden on January, 1st 2000 which was of 8860859 and, assuming an arithmetic increase of 5.3 ‰ per year for ten years, we compute the Swedish population on January, 1st 2010. Data for this example are taken from the Human Mortality Database (2015).

If we assume a uniform arithmetic (a.k.a. geometric) increase, the population at time $T$, denoted by $N_T$, is given by

$$N_T = N_0(1+r)^T$$

where $N_0$ is the population at the beginning of the period.

We start "translating" $N_0$ in R :

```
N0 <- 8860859
```

In this command, we have assigned a certain number to an R object named N0 by an arrow that points to the left: the combination of the left-than sign (<) with the hyphen (-). Did you notice that no result was returned in the following lines. We have just made an assignment without asking to show/print us the result. If we want to see it just type:

```
N0

## [1] 8860859
```

Remember that the name of an R object cannot start with numbers, and it is always a good habit to choose sensible and meaningful names. Moreover if we write something else using the same name (e.g. N0), the previous object will be automatically overwritten.

Let's now compute the population in 2010:

```
# Swedish population in 2010
# assuming arithmetic growth rate
r <- 5.3/1000
T <- 10
NT <- N0 * (1 + r)^T
NT

## [1] 9341845
```

For the record, Swedish population in 2010 was equal to 9340560: the error is only of

```
NT - 9340560

## [1] 1284.875
```

persons.

Perhaps you have noticed that everything that follows the # sign is assumed to be a comment and is ignored by R . Note that R is case-sensitive, that means:

```
Nt

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

are not recognized and produce errors.

```
## Warning in rm(my_pdf):  object 'my_pdf' not found
```

Sometimes we want to know the objects in the workspace or to remove them. These operations can be done in the following way:

```
# list of the object
ls()

## [1] "N0" "NT" "r"  "T"

# remove N0
rm(N0)
# list of the objects, without N0
ls()

## [1] "NT" "r"  "T"
```

# 3 Using an Editor

When you write a program, it normally does not consist of only one or two lines. Therefore it is a good idea to write the code of the program in a text editor and then run the whole code or parts of it in R . We suggest to use R Studio, which is free, and can be downloaded from rstudio.org, respectively. A good alternative might be Emacs or Eclipse (gnu.org/software/emacs, www.eclipse.org), which are particularly flexible and useful for all programming and document markup languages, but it takes some time to really learn them

R Studio is pretty user-friendly, and learning about all its features goes beyond the scope of these lectures. Nevertheless, before we continue with the next sections, take some minutes to get acquainted of your selected editor while typing and compiling the previous code snippets.

# 4 The user friendly R

## 4.1 Functions

Most of the work in R is done through functions. There are many functions already implemented, but we can also code our own functions (see Module 4). Just as an introduction we will try three simple functions: `c()`, `sum()` and `mean()`. The parenthesis surround the argument list.

The function `c()` combines values into a vector. For instance:

```
my.vec <- c(1,2,3,4,5,6,7,8,9,10)
my.vec

##  [1]  1  2  3  4  5  6  7  8  9 10
```

In this case the arguments of the function `c()` are a series of numbers separated by commas. The resulting vector is assigned to the object `my.vec`. We can calculate the mean of such vector manually:

```
(1+2+3+4+5+6+7+8+9+10)/10

## [1] 5.5
```

Alternatively we could first sum them up and then divide by 10:

```
sum.vec <- sum(my.vec)
sum.vec/10

## [1] 5.5
```

Obviously R provides a function for computing the arithmetic mean of a vector, namely `mean()`:

```
my.mean <- mean(my.vec)
my.mean

## [1] 5.5
```

In this last case we assign the outcomes to the object `my.mean`.

In the remainder of these materials, elementary functions will often be introduced without formal presentation. The next section shows what to do when these functions are not completely intuitive in their usage.

## 4.2 Getting Help

This material can not cover all aspects of R . However, R is very user friendly and provides an extensive and detailed help documentation. For different problems there are different ways to get the help.

- If information you want information on a function whose name you know, the command `help(function_name)` is used. Alternatively the command `?function_name` can be typed in. For example, just ask for the documentation on the function `mean`:

  ```
  ?mean
  ```

- If only a keyword is known and not a respective function, it is possible to search across all functions using the command `help.search`(keyword")". If we do not know the function for computing the mean value we just type:

  ```
  help.search("mean value")
  ```

- There is also the possibility to use simply the menu item help which is provided in R or R Studio.

Sometimes the most interesting part of the function help is the one related to the examples. A useful way of looking directly at this section is the function `example()`:

```
example(mean)
```

An alternatively way would be to go at the end of the documentation file and run each line of the example by your own.

### 4.3 The Package System

R consists of a lot of packages written for specific purposes. These packages contain fundamental statistical functions (`stats`), demographic functions (`demography`), biodemographical functions (`Biodem`), etc.

A few packages are automatically loaded on the start up of R , e.g. the packages `base` and `stats`. All R -packages are downloadable from the R web-page by typing

```
install.packages("package_name")
```

and selecting your (nearest) CRAN mirror. You need to be connected to the Internet to install packages in this way.

If installed, a package can be loaded in R by the command:

```
library(package_name)
```

R Studio also has functionality for finding and loading packages.

A loaded package is only available during a current session and must be re-loaded in the next session. Information about a package and its internal functions can be obtained using the command:

```
library(help=package_name)
```

## 5 Data Handling

The main task of any statistical software tool is the analysis of data. In this section we explain how to read data into R and how to store the results of the analysis (or again data).

Often it is useful to check in which directory R is open in order to possibly change it and look at the current list of files in it. The commands to know the current working directory and the associated list of files and/or sub-directories are given by

```
getwd()
dir()
```

We can eventually change the working directory to a specific one on our machine:

```
setwd("~/WORK/data/") # for a Linux OS
setwd("C:\\WORK\\data\\") # for a Windows OS
```

In R pure text data can be read in using the function `read.table()`. It reads the specified file and transforms it into a specific data format, a `data.frame` (see Module 2). The following arguments are often used for this function:

- The only required argument in any case is the file name with the directory where the file can be found (unless the session is open in the folder where the data are saved).
- The argument `header` is a logical variable indicating whether variable names are included in the top of each column or not.
- The argument `sep` specifies which delimiter has been used to separate the columns. If the text file is comma separated we type in `sep=","`. If a space has been used for separation we write `sep=" "`. If a file is tab-separated we enter `sep="\t"` (which is the default value).

- If leading lines of data should be skipped we use the argument `skip`.
- The argument `nrows` specifies the maximum number of rows to read in.

As an example we read a dataset which contains the mid-year population and number of deaths in Sweden and Kazakhstan by age-groups for females in 1992 (Preston et al., 2001, p. 22). The variables are:

- `pop.swe`: mid-year population of Sweden
- `dea.swe`: deaths during the year for Sweden
- `pop.kaz`: mid-year population of Kazakhstan
- `dea.kaz`: deaths during the year for Kazakhstan

In `R` we read these data as follows:

```
SweKaz <- read.table(file="swed_kaza.txt", header=TRUE, sep="\t")
```

We may inspect the data using the function `head()`, which presents the first six data rows.

```
head(SweKaz)

##   pop.swe dea.swe pop.kaz dea.kaz
## 1   59727     279  174078    3720
## 2  229775      42  754758    1220
## 3  245172      31  879129     396
## 4  240110      33  808510     298
## 5  264957      61  720161     561
## 6  287176      87  622988     673
```

For a more general overview of the data we can type:

```
fix(SweKaz)
```

This function will open a pseudo-spreadsheet which is best to close before continuing with our work. R Studio also has an editable view tab that lets you examine data in a spreadsheet layout. In general, it is best to not use these options to edit data, but rather just to view them.

`R` also provides functions to read in other data formats as `*.txt`. In Table 1 common data formats and the corresponding function are displayed.

| Program | Format | `R` Function | Package |
|---|---|---|---|
| | *.txt | read.table() | base |
| EXCEL | *.csv | read.csv() | base |
| SPSS | *.sav | read.spss() | foreign |
| Stata | *.dta | read.dta() | foreign |

Table 1: Functions to read in data

The standard function in `R` to export data is `write.table()`. For example we store the data-object `SweKaz` including the variable names. We separate the columns using tabs (default value).

```
write.table(x=SweKaz, file="MyData.txt", col.names=TRUE)
```

You can check the presence on your directory of the exported object `MyData` by typing `dir()`.

When you close `R` all the created objects will not be automatically saved. Nevertheless, all your code should be preserved in `.R`-files by the editor. You would only need to run the scripts again the to obtain the previous objects. This can be done by:

```
source("MyScript.R")
```

Rarely, you may need to save the workspace. Based on our experience, this is principally useful when time consuming calculations have been run. In `R` you need to type:

```
save.image("FirstExerciseR.RData")
```

And to retrieve the session `FirstExerciseR.RData`:

```
load("FirstExerciseR.RData")
```

R Studio does this pretty much automatically unless you opt out on closing the session.

# 6 Data types

R has a wide range of data types. The basic ones are the following:

- numeric
- character
- logical
- factor
- date and times
- complex

An example of a numeric object is given by `my.vec`. We can ask for the storage mode of the object by

```
mode(my.vec)

## [1] "numeric"
```

Vectors as well as scalars can be also made up of strings instead of numbers. We simply have to enclose the characters/strings by quotation marks like in the following example:

```
country <- c("Japan", "USA", "France")
country

## [1] "Japan"  "USA"    "France"

mode(country)

## [1] "character"
```

In many situations, a sequence of strings masks realizations from categorical variables such as religion, political party and blood type. R offers an efficient way to store these sequences as factors:

```
blood <- c("A", "B", "O", "AB", "A", "B", "O", "O", "AB", "B")
blood <- factor(blood)
blood

##  [1] A  B  O  AB A  B  O  O  AB B
## Levels: A AB B O
```

In the first line we list the blood type of 10 observations, then we force the vector of characters to be factor and then we print it. Note that R recognizes that `blood` is made up of a limited number of possible values. To query them and their number:

```
levels(blood)

## [1] "A"  "AB" "B"  "O"

nlevels(blood)

## [1] 4
```

Factors are internally stored as integers (another storage mode of the object) implying lower memory usage than alternatives. Moreover factors will be important when we later run statistical models (see Module 8). If factor levels are assumed to be ordered, the internal argument `ordered` must be set to `TRUE`:

```
educ <- c("Prim", "Ter", "Sec", "Sec", "Ter", "Prim")
educ <- factor(educ, ordered=TRUE)
educ

## [1] Prim Ter  Sec  Sec  Ter  Prim
## Levels: Prim < Sec < Ter
```

When strings are provided, `R` orders the factor levels alphabetically.

Missing values are often present in demographic datasets. In `R` such instance is coded with the symbol `NA` (Not-Available) and it can lead to misleading outcomes when operators are applied:

```
my.na <- c(1,2,3,4,NA,6,7,NA,NA,10)
my.na

##  [1]  1  2  3  4 NA  6  7 NA NA 10

mean(my.na)

## [1] NA
```

There are various ways to omit missing cases, and they depend on the analysis undertaken.

Other special cases are undefined values, which are represented by `NaN` (Not-a-Number) and infinite values (`Inf`):

```
0/0

## [1] NaN

1/0

## [1] Inf

-1/0

## [1] -Inf
```

Vectors made up of logical values (`TRUE/FALSE`) are also possible in `R` :

```
my.log <- c(TRUE, FALSE, TRUE, FALSE, FALSE)
my.log

## [1]  TRUE FALSE  TRUE FALSE FALSE

mode(my.log)

## [1] "logical"
```

Also in case of logical values, `R` efficiently stores them as 0/1 values. Note that both summation and mean of a logical vector are feasible as well as sensible operations:

```
sum(my.log)

## [1] 2

mean(my.log)

## [1] 0.4
```

These results tells us that in `my.log` we have two values equal to `TRUE` which represents 40% of the total number of elements. Logical values can be very useful for selection, as we will see later.

Logical values are also results of relational operators. We can test some of these operators on the mean value of `my.vec` as follows:

```
my.mean > 5.5  # test my.mean greater than 5.5

## [1] FALSE

my.mean <= 5.5 # test my.mean smaller than or equal to 5.5

## [1] TRUE

my.mean == 5.5 # test my.mean exactly equal to 5.5

## [1] TRUE

my.mean != 5.5 # test my.mean not equal to 5.5

## [1] FALSE
```

Other relational operators with intuitive meanings are >= and <. These operators can be also vectorized to test element-wise relations. For example we can test which values of `my.vec` are greater than its own mean (`my.mean`):

```
my.vec > my.mean

##  [1] FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
```

Multiple relational operators can be useful. Here we test which elements of `my.vec` are equal to either 1,3,7 or 9:

```
my.vec%in%c(1,3,7,9)

##  [1]  TRUE FALSE  TRUE FALSE FALSE FALSE  TRUE FALSE  TRUE FALSE
```

or different from the mentioned values:

```
!my.vec%in%c(1,3,7,9)

##  [1] FALSE  TRUE FALSE  TRUE  TRUE  TRUE FALSE  TRUE FALSE  TRUE
```

Here we use the command `!` which indicates logical negation, i.e. we swap `TRUE` and `FALSE` from the previous command line.

An additional relational operator can be used to detect missing cases within an object:

```
is.na(my.na)

##  [1] FALSE FALSE FALSE FALSE  TRUE FALSE FALSE  TRUE  TRUE FALSE
```

`R` provides several options for dealing with date and time data. The function `as.Date()` can be used for converting strings and objects of class `Date` representing calendar dates. For instance, we can assign to two objects dates of birth and death of a given individual:

```
birth <- as.Date('1961-7-31')
death <- as.Date('1989/11/9')
birth

## [1] "1961-07-31"

death

## [1] "1989-11-09"
```

The default format is a four digit year, followed by a month, then a day, separated by either dashes or slashes. Moreover, despite the print version resembles strings objects, R recognizes them as dates and numerical functions are feasible:

```
life <- death-birth
life

## Time difference of 10328 days
```

and tells that life-time of this observation was 10328 days.

For the mathematically oriented readers, we inform that complex numbers can also be handled within R . A small instance is given here in which we assign to z the complex number $3 + 2i$ where $i = \sqrt{-1}$:

```
z <- 3+2i
z

## [1] 3+2i
```

We can take the complex conjugate of z $(3 - 2i)$ and extract its real or complex parts as follows:

```
Conj(z)

## [1] 3-2i

Re(z)

## [1] 3

Im(z)

## [1] 2
```

These items can be particularly helpful in demography when working with projection matrices, for instance.

It is noteworthy to remember that if different data types are included within a vector containing strings, all elements will be coerced to character strings:

```
my.mix <- c("Japan", "USA", 20, TRUE, z)
my.mix

## [1] "Japan" "USA"   "20"    "TRUE"  "3+2i"
```

# 7 Exercises

## Exercise 1

Solve the following problems using your editor and R :

(i) $2 + 3 * (4 + 5 * (6 + 7 * (8 + 9)))$

(ii) $1e{+}07 * 4^3$

(iii) $log_e(2)$

(iv) $log_{10}(2)$

(v) $\sqrt{333}$.

### Exercise 2

Compute

$$\frac{1335}{4}y^6 + x^2(11x^2y^2 - y^6 - 121y^4 - 2) + \frac{11}{2}y^8$$

where $x = 0.2$ and $y = 3.5$.

### Exercise 3

Read in the first 10 rows of the dataset `GDPe02012.txt`.

### Exercise 4

Create a vector for 10 observations and 3 religions. It is up to you the types of religions and their order.

Then, search for and apply a function, which informs you about the number of observations in each category (i.e. build a contingency table of the object).

### Exercise 5

The 20th Century experienced two World Wars. The first started on 28 July, 1914 and lasted until 11 November, 1918. The Second World War started with the invasion of Poland on September 1, 1939 and the final armistice was signed on 14 August, 1945.

Compute the length of these two important conflicts, test which one has lasted longer and how many days more.

### Exercise 6

Show in `R` that:

$$z^\star \cdot z = |z|^2$$

where $z^\star$ is the complex conjugate of the complex number $z = 4 + 3i$. Note that a simple logical operator is not sufficient here: just print the outcomes of the right- and left-end-side of the equation.

## References

Human Mortality Database (2015). University of California, Berkeley (USA) and Max Planck Institute for Demographic Research (Germany). Available at www.mortality.org or www.humanmortality.de (data downloaded on July 10, 2014).

Preston, S. H., P. Heuveline, and M. Guillot (2001). *Demography: Measuring and modeling population processes*. Oxford: Blackwell.