

Third VALUE training workshop

Spatial and Temporal Variability in Statistical and Dynamical Downscaling

November 3-8 2014

Abdus Salam International Centre for Theoretical Physics, Trieste,
Italy

Preparatory material for Rglimclim training sessions

Richard Chandler (r.chandler@ucl.ac.uk)

Contents

1.	Introduction	1
2.	What is R?	1
3.	RStudio	1
4.	Obtaining the software and setting up	1
4.1	Obtaining R	1
4.2	Obtaining RStudio	2
4.3	Setting yourself up	2
5.	Getting started	2
5.1	Getting out	2
5.2	Simple manipulations; numbers and vectors	3
5.3	Stored objects	3
5.4	Syntax for R commands	4
5.5	Creating vectors in R	4
5.6	R vector arithmetic	5
5.7	Current working directory	6
5.8	Getting unstuck	6
6.	Self-study exercise: analysis of Galapagos tortoise data	6
6.1	Reading data from a file	8
6.2	Examining data frames	9
6.3	Customizing graphics	11
6.4	Fitting statistical models	12
6.5	Running commands in batch mode	14
7.	Additional R packages	14
8.	Postscript: the case study area	15

1. Introduction

During the training sessions on the afternoons of Wednesday 5th and Thursday 6th November, we will use the Rglimclim software package to build and evaluate a stochastic weather generator for daily temperature and precipitation in northern Iberia. Rglimclim is a powerful and flexible weather generation package that runs within the R statistical software environment. A full introduction to Rglimclim will be provided during the training sessions. To ensure that delegates are able to make the most of the sessions however, they are requested to install all necessary software on their laptops prior to arrival: instructions are provided in Sections 4 and 7 below. Delegates who already have R installed should ensure that they also have RStudio (see Section 4.2) and that they have installed the necessary add-on packages (Section 7). Delegates who are unfamiliar with R should also work through the preparatory exercises in Sections 5 and 6. These exercises do not have any direct connection to statistical downscaling or to stochastic weather generation: their purpose is merely to provide an introduction to the “look and feel” of R.

2. What is R?

R describes itself as “a free software environment for statistical computing and graphics”. It is built around the award-winning S language, which was developed in the late 1980s; and it is the computing environment of choice for much modern statistical work. Initially its use was confined largely to the academic community, but it is now used much more widely. Its popularity stems from a number of attractive features:

- It is free.
- It has many “inbuilt” statistical commands (e.g. to fit linear and nonlinear models, carry out classical statistical tests, do time-series analysis, classification, clustering, ...) and graphical techniques: its capabilities equal or exceed those of all commercial statistics packages.
- It provides an environment for users to define their own new procedures, and hence offers great flexibility.
- The Comprehensive R Archive Network (<http://cran.r-project.org/mirrors.html>) distributes user-contributed extensions, or add-on “packages” that extend the basic capabilities of the environment. There are literally hundreds of these, all freely available and contributed by researchers all over the world. Thus, tested implementations of the latest statistical methods are freely and widely available to anybody with an internet connection.
- It has excellent facilities for producing a wide range of publication-quality graphics; and graphical displays are easily customized so that the user has full control over their appearance.
- It runs on almost all modern operating systems. In this introduction, it is assumed that most delegates use Microsoft Windows, and the instructions below are tailored to the Windows implementation. However, the differences for Linux or Macintosh operating systems are largely cosmetic.

3. RStudio

Although R itself provides a limited user interface (on Windows and Macintosh operating systems, at least), its inbuilt editor is rather basic. Moreover, it can be difficult to keep track of your files, your graphics windows, the objects in your workspace (see Section 5.3 below) and so forth. The RStudio package addresses these issues: it describes itself as “an integrated development environment (IDE) for R [including] a console, syntax-highlighting editor that supports direct code execution, as well as tools for plotting, history, debugging and workspace management”. In these training sessions, we will use RStudio to make our lives easier. Instructions for installing both R and RStudio are in the next section. Install R *first*, and then RStudio!

4. Obtaining the software and setting up

4.1 Obtaining R

The R homepage is at <http://www.r-project.org/>. To install the software, follow the [CRAN](#) link on the left-hand side of the page. This takes you to a list of mirror sites: click the link for a site near you and follow the appropriate links to download an appropriate binary installation file for your operating system (at the time of writing, these binary files are for R version 3.1.1). **Note** to Windows users: at this stage, you need only the installation file for the R base package – we will install additional contributed packages later (see Section 7).

When you have downloaded the installation file, run it to install the software onto your computer. You may need administrative privileges to do this. You should probably accept the default settings during the installation process, unless you know what you’re doing. Installation under some Linux systems requires careful reading of the instructions!

4.2 Obtaining RStudio

Having installed R, you should install RStudio from <http://www.rstudio.com/products/rstudio/download/>. Download and install the appropriate version for your operating system – the installers can be found under the heading “Installers for ALL Platforms”.


4.3 Setting yourself up

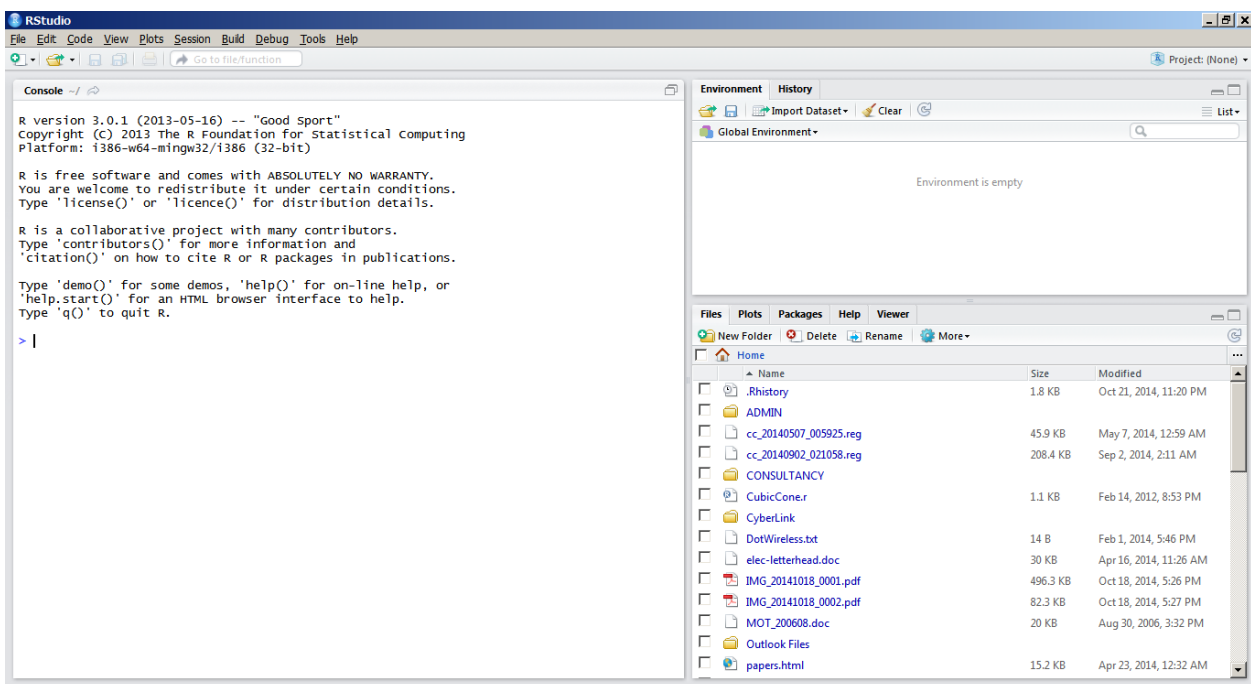
You may download a zip archive `RglimclimPrep.zip` from the workshop web page at <http://www.value-cost.eu/node/1143> (the same address that you used to download the present document). This archive contains data files and sample R code for the self-study exercise below. Create a separate folder / directory in which to store these files, and unpack the zip archive into this folder. We will return to this later.

You will need to install some additional software as well. However, you need to know something about how R works before doing this. We will start by learning some key concepts therefore, and install the additional software later (Section 7).

5. Getting started

The instructions here are mostly for the Windows operating system; details for other operating systems are similar. In the Windows version of RStudio, commands appear in blue; the results of those commands are in black; and information and error messages appear in red. This convention is followed throughout the remainder of this document.

So: if you are using Windows, and your installation was successful, you will have an RStudio icon  on your desktop.¹ Double-click this icon to start the program, and you get something like this:



The left-hand subwindow is entitled `Console`: this is where most of the action takes place. We will look at the other subwindows later. When you start R, the console contains some text followed by a prompt

```
>
```

which indicates that R is waiting for a command from you.

5.1 Getting out

The most important thing to know about any software package is how to get out of it. To quit RStudio, type `q()` (including the brackets) at the prompt, and press the `<Return>` key:

```
> q()
```

¹ Similar icons appear on other operating systems, e.g. on the Launcher in Ubuntu if you're using the default Unity desktop.

If it asks you Save workspace image to ~/.RData? [y/n/c]:, type n (for “No”) and press <Return>. Now start up RStudio again, and we will look at some very basic features of the R environment.

5.2 Simple manipulations; numbers and vectors

Elementary R commands are either *expressions* or *assignments*.

- An *expression* is a command to display the result of a calculation, which is *not* retained in the computer's memory.
- An *assignment* passes the result of a calculation to the name of an R *object* which is stored (but the result will not necessarily be printed out on the screen).

Example. Here is an expression, which displays the result of the calculation $5 + 2.6$:

```
> 5 + 2.6
[1] 7.6
```

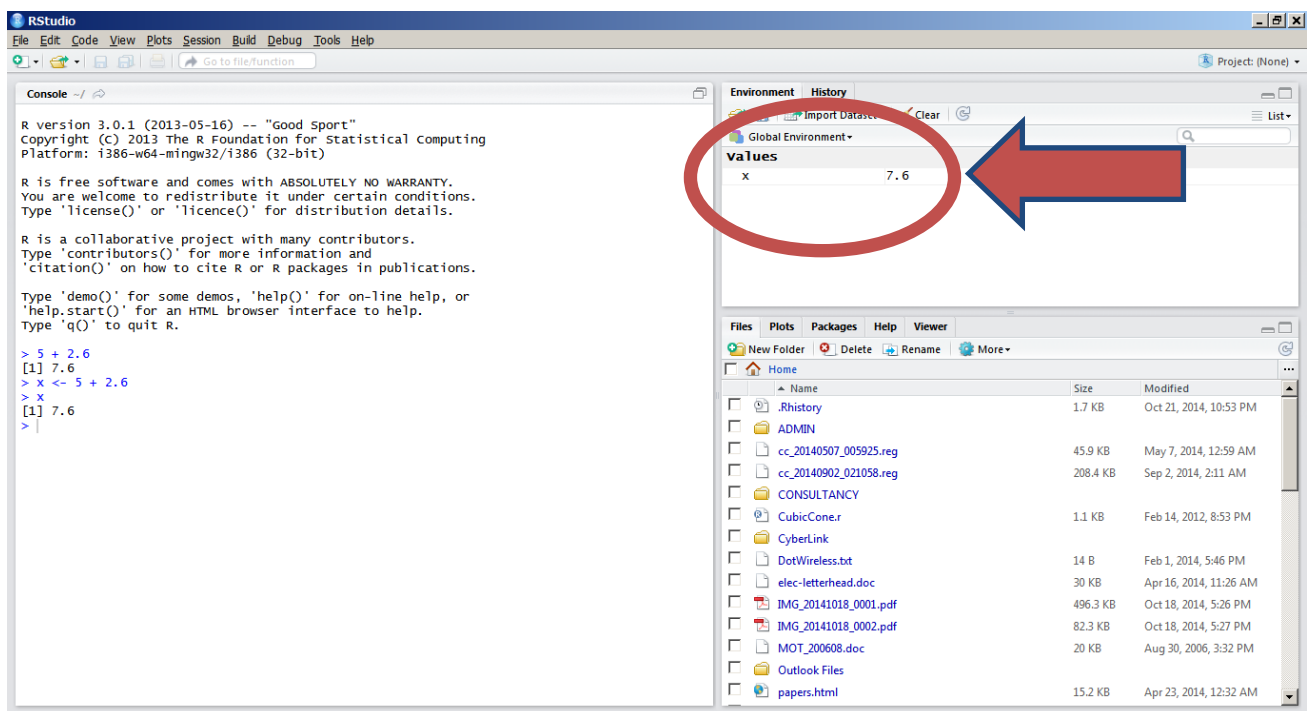
Don't worry about the [1] for the moment! Here is the same calculation, but with the result assigned to an object called x:

```
> x <- 5 + 2.6
> x
[1] 7.6
```

Note the use of the *assignment operator* <- above. This reads as an arrow pointing to the object x – you can interpret the command `x <- 5 + 2.6` as “take the value of the expression $5 + 2.6$, and store it in x”.

5.3 Stored objects

All assigned R objects are automatically stored by the computer in your *R workspace*, until you finish your R session. When you finish (by typing `q()`), R gives you the option to save your workspace for future use. If you always save it, all objects will remain on disk until overwritten or explicitly deleted by the command `rm()` (for *remove*). This means that, after a while, your R files can take up a lot of disk space: periodically therefore, you may need to delete those which are no longer required. RStudio displays the objects in your workspace in the top right-hand subwindow. You should see that the object x is listed there now, together with its value of 7.6:



Another way to see what variables are stored is to type `ls()` (for *list*) or `objects()`.

Example:

```
> x <- 8
```

```

> x
[1] 8
> y <- 3.1415
> ls()
[1] "x" "y"
> rm(x)
> objects()
[1] "y"

```

Note that R is case-sensitive: `x` and `X` are different objects. Take care when typing, therefore!

5.4 Syntax for R commands

Notice that all R commands, e.g. `ls()`, `rm()`, are followed by parentheses which may or may not contain additional information (*arguments*). Writing a command name without parentheses simply makes R write out the underlying code.

Example:

```

> rm(y)
> rm
function (..., list = character(0), pos = -1, envir = as.environment(pos),
  inherits = FALSE)
{
  dots <- match.call(expand.dots = FALSE)$...
  if (length(dots) && !all(sapply(dots, function(x) is.symbol(x) ||
    is.character(x))))
    stop("... must contain names or character strings")
  names <- sapply(dots, as.character)
  if (length(names) == 0)
    names <- character(0)
  list <- .Primitive("c")(list, names)
  .Internal(remove(list, envir, inherits))
}
<bytecode: 0x08e2f8c4>
<environment: namespace:base>

```

Don't worry about the details here – the point is that if you omit the parentheses, R simply tells you how the `rm()` command is defined.

5.5 Creating vectors in R

The command `c()` (for *combine*) creates R vectors.

Example:

```

> x <- c(2.3, 1.2, 2.4)
> x
[1] 2.3 1.2 2.4
> y <- c(x, 9.0, x)
> y
[1] 2.3 1.2 2.4 9.0 2.3 1.2 2.4

```

If you make a mistake while typing, you can use the up arrow key \uparrow to recall your previous commands and correct them – you don't have to type everything again from the beginning.

Sometimes we want to create sequences. The expression `1:n` denotes the sequence 1, 2, ..., n-1, n. More generally, `seq(i, j, k)` is a sequence from *i* to *j* in steps of *k*.

Example:

```

> z <- 1:50
> z
[1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27
[28] 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
> seq(3, 10, 2)
[1] 3 5 7 9

```

We can now clarify the meaning of the `[1]` in the output for many of the earlier examples: it indicates that the object being printed is a vector, and that the current line of output starts with its first element. In the first part of the example above, the vector is too long to fit on one line, and we see that the second line starts with the 28th element of the vector (which is obvious in this particular case, but not in general).

Notice also that the “Environment” tab in the top right-hand subwindow in RStudio now contains more details of the current objects in the workspace, for example `num [1:3] 2.3 1.2 2.4` alongside `x`. This tells us something about what kind of object `x` is – in this case, a numeric vector indexed from 1 to 3 (because it has three elements).

To extract specific elements of a vector, we can use square brackets `[]`.

Example:

```
> z <- c(3,1,4,1,5,9,2,7)
> z[3]
[1] 4
> z[c(3,2,5)]
[1] 4 1 5
```

The last command above uses a vector `c(3,2,5)` within the square brackets `[]`, to extract respectively the third, second and fifth elements of `z`.

5.6 R vector arithmetic

R uses the symbols `+`, `-`, `*` and `/` for the basic arithmetic operations, and `^` for exponentiation (raising to a power). Vector operations are done element by element, with recycling of short vectors if required.

Example:

```
> x <- c(2,3)
> y <- c(1,4,5,6)
> 2*x
[1] 4 6
> 2 + x
[1] 4 5
> y^2
[1] 1 16 25 36
> x + y
[1] 3 7 7 9
> x*y
[1] 2 12 10 18
```

The last two examples above illustrate the “recycling” convention: `x` has two elements but `y` has four, so the third and fourth elements of `y` are paired with the first and second elements of `x` respectively.

It is important to understand the order in which R does things. For the standard arithmetic operations `+`, `-`, `*`, `/` and `^`, R follows the usual rules: `^` is done first, then `*` and `/`, and finally `+` and `-`.

Example:

```
> 4+2*3
[1] 10
> 2^2*3
[1] 12
```

Note, however, that it is difficult to read these commands. In general, it is good practice to make explicit the order in which you *want* the calculations to be performed, using brackets:

```
> 4+(2*3)
[1] 10
> (4+2)*3
[1] 18
```

Not only does this make the commands easier to read; it also guarantees that R is not making any decisions for you. Much of the time, its decisions are very intuitive. However, this is not always the case and it is always best to err on the side of caution. Look at this:

```
> x <- 3
> 1:x
[1] 1 2 3
> 1:x+2
[1] 3 4 5
```

You might expect the last command to produce the sequence 1 to $x+2$ i.e. 1,2,3,4,5. However, in R the sequence operator `:` is executed before *any* of the arithmetic operators. So R interprets the last command above as $(1:x)+2$. To count from 1 to $x+2$, we must go

```
> 1:(x+2)
[1] 1 2 3 4 5
```


5.7 Current working directory

In some of the examples below, we will use R to read data and commands from files. Whenever you ask R to access information from a file, by default it assumes that the file is located in the *current working directory*. Similarly, if you ask R to save anything then it will usually be written to the current working directory. To find out what this is, use the `getwd()` command:

```
> getwd()
[1] "C:/Users/richard/Documents"
```

You will get a different result here, depending on your system setup.² You should usually change the current working directory to the place where you have saved the files for your current project. The easiest way to do this in RStudio is via the `Set Working Directory` option on the `Session` drop-down menu. This allows you to browse your directories and select (by clicking) the directory you want. Do this now, and change to the directory into which you unpacked `RglimclimPrep.zip` in Section 4.3. Check that R can see all of the files that you unpacked – the contents of the directory are displayed in the “Files” tab in the lower right-hand subwindow. The first few files in the list should be `galapagos.dat`, `R_SelfStudy.r` and `R_SelfStudy_Clean.r`. If you cannot see these and other files, then EITHER you have not changed to the correct working directory, OR you did not unpack `RglimclimPrep.zip` correctly in Section 4.3.

5.8 Getting unstuck

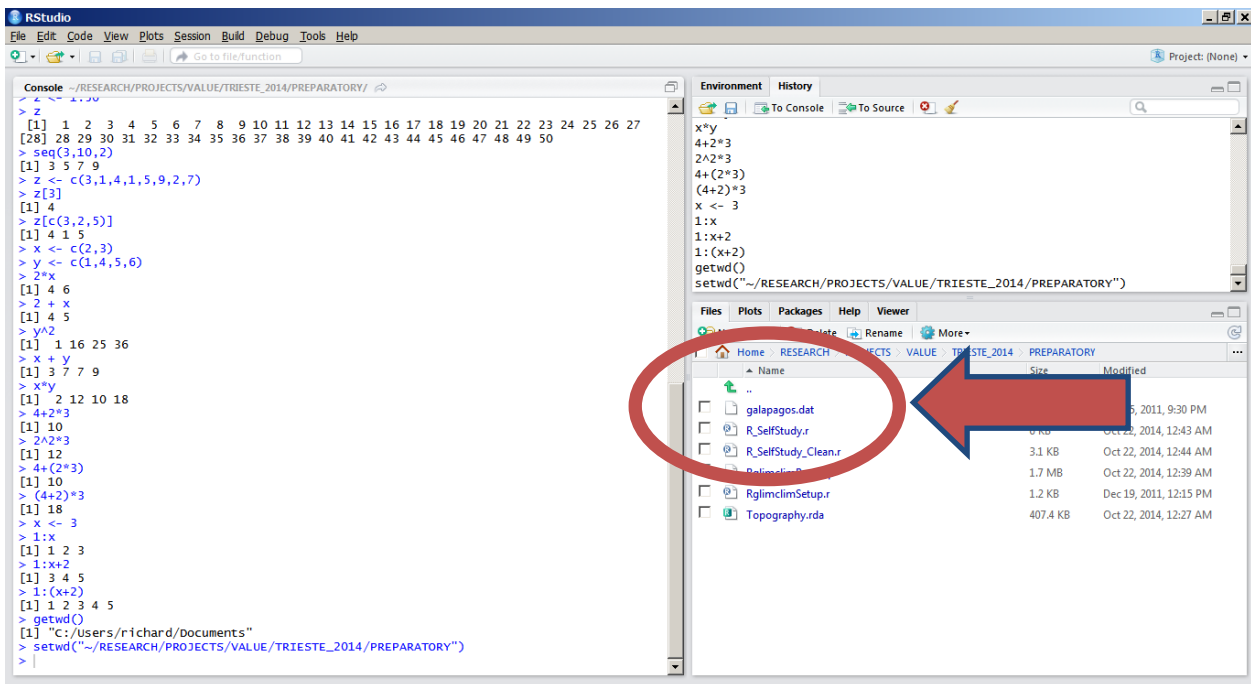
R and RStudio are usually very stable (although RStudio occasionally hangs in Ubuntu). However, any software package can occasionally have problems, and all package users can do silly things. Sometimes therefore, you may need to intervene to stop a particular operation that seems to be taking too long or was started in error. If you’re lucky, this can be done by clicking on the “STOP” icon  at the top of the console subwindow. If this doesn’t work (or if the “STOP” icon doesn’t appear), try the `Restart R` option on the `Session` drop-down menu – or, if all else fails, the `Terminate R` option. If you use either of the last two options, you will need to reset the current working directory afterwards.

6. Self-study exercise: analysis of Galapagos tortoise data

We can now demonstrate some of the basic graphical and analytical capabilities of R. **Please note:** you should not expect to remember everything instantly. The purpose of this exercise is merely to give you a feel for how the system works, so that you can recognize the key concepts when we start using `Rglimclim`.

For this demonstration, we will analyse some data on tortoise species in the Galapagos islands: these data can be found in file `galapagos.dat`, which was unpacked from `RglimclimPrep.zip` in Section 4.3. To view this file, set the working directory appropriately (if you have just worked through Section 5.7, you should already be in the right place) and click on `galapagos.dat` in the “Files” tab of the lower right-hand subwindow (see screenshot below):

² Windows users: notice that R uses a forward slash `/` as a directory separator, whereas Windows usually uses a backslash `\`. Yours is not to reason why ...



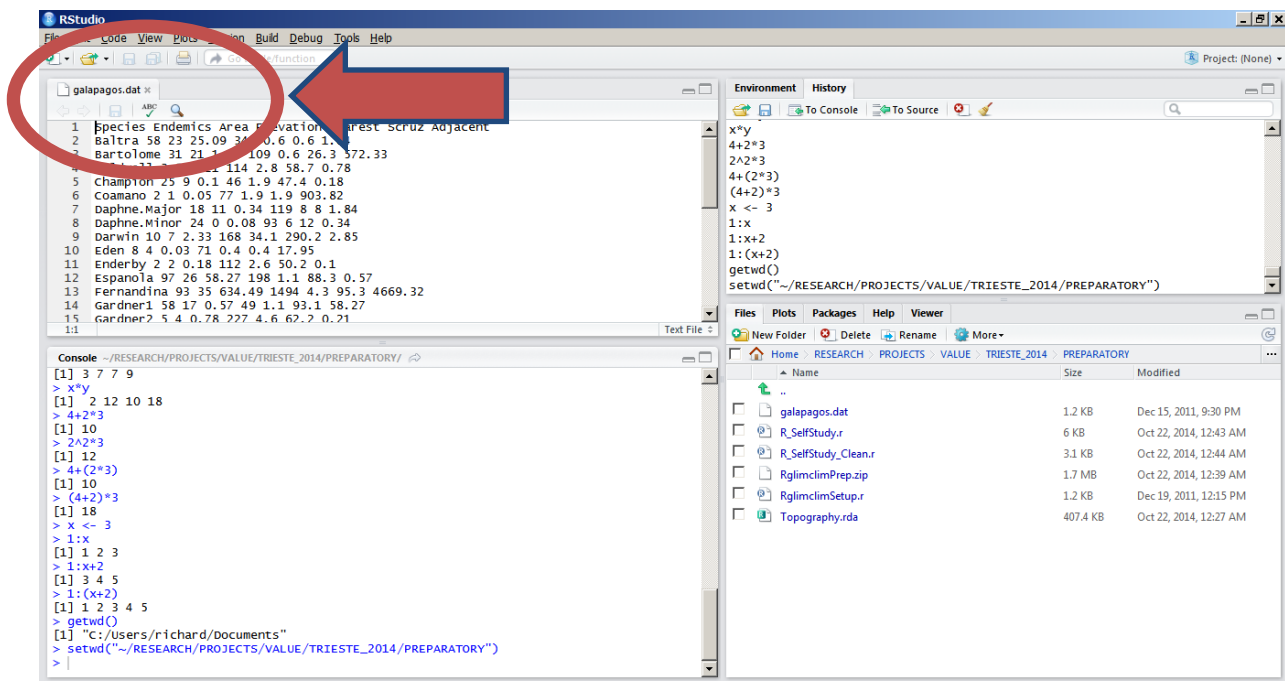
When you click on the filename, a new subwindow opens displaying the file contents. The first few rows look like this:

```

Species Endemics Area Elevation Nearest Scruz Adjacent
Baltra 58 23 25.09 346 0.6 0.6 1.84
Bartolome 31 21 1.24 109 0.6 26.3 572.33
Caldwell 3 3 0.21 114 2.8 58.7 0.78
Champion 25 9 0.1 46 1.9 47.4 0.18

```

The data file contains data on seven variables for each of 30 islands. The first row contains the variable names; each remaining row contains the name of an island, along with the values of the seven variables for that island. Fields are separated by spaces and are not aligned in columns. We will find out what the variables represent in a moment. First, however: we don't need to keep the data file on display, so close it by clicking the "x" symbol in its tab:



To analyse these data, we will use a script containing a sequence of R commands, and which was also provided in RglimclimPrep.zip. This script is called R_SelfStudy.r. To open it, click its name in the "Files" tab. Do *not* open R_SelfStudy_Clean.r yet – we'll look at that later!

The script opens in a new subwindow again. The first few lines look like this:

```
#####
#####
#####
#####
#####          3rd VALUE training workshop, 2014          #####
#####          #####
#####          Preparatory material for Rglimclim training:  #####
#####          self-study exercises                          #####
#####          #####
#####          Analysis of Galapagos Islands data           #####
#####          #####
#####          This script contains the commands used in a short #####
#####          (and non-definitive) analysis of the dataset on  #####
#####          Galapagos islands tortoises that is considered in #####
#####          #####
#####          Faraway, J. (2005). Linear Models with R. Chapman & #####
#####          Hall / CRC Press, Boca Raton.                 #####
```

Notice the use of the “hash” sign #. This indicates a *comment* in R code: R will ignore everything after a hash sign on any line of code.³ Comments are used to make code easier to read.

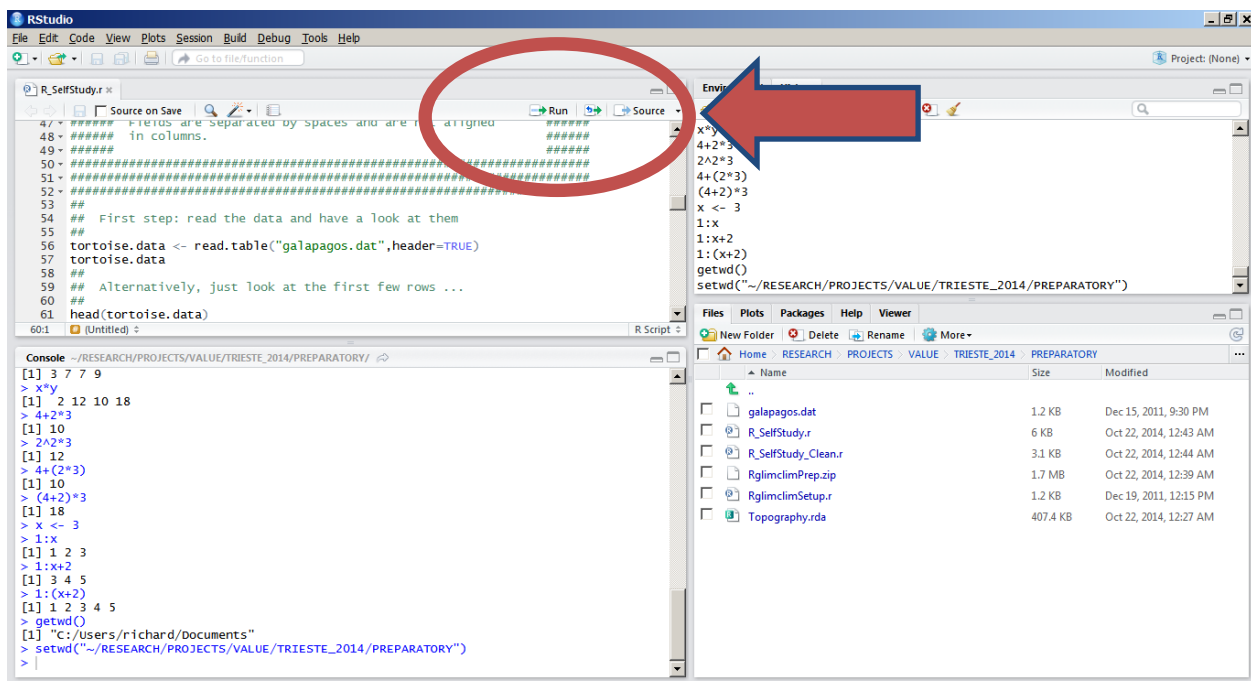
Read through the comments at the top of the script – they tell you what the data represent. Ignore the part about the `faraway` package – add-on packages are discussed in Section 7 below.

6.1 Reading data from a file

After the script header, you find the following:

```
##
##   First step: read the data and have a look at them
##
tortoise.data <- read.table("galapagos.dat",header=TRUE)
```

The first three lines are comments. The last reads data from the file `galapagos.dat` and assigns it to the object `tortoise.data`. To run this command, copy and paste it to the console prompt. The easiest way to do this is to position the cursor anywhere on the line, and either press `<Ctrl-R>` or click the “Run” button at the top of the subwindow:



³ Unless the hash is obviously part of a character string and should be interpreted as such.

```
> tortoise.data <- read.table("galapagos.dat",header=TRUE)
```

Hopefully, you won't see anything when you do this. If you see the following:

```
Error in file(file, "rt") : cannot open the connection
In addition: Warning message:
In file(file, "rt") :
  cannot open file 'galapagos.dat': No such file or directory
```

it means that R could not find the data file `galapagos.dat`. The reason is probably EITHER that you have not changed to the correct working directory (see Section 5.7), OR that you did not unpack `RglimclimPrep.zip` correctly in Section 4.3.

The `read.table()` command is often the easiest way to read data into R. It can be used whenever your data are provided in an ASCII file, in fields separated by spaces. The fields may be aligned in columns, but (as here) this is not necessary – all that is required is that each row of the data file contains the same number of entries. The argument `header=TRUE` tells R that the first row of the data file contains variable names rather than data values.

To see what the above command has done, copy and paste the next command to the prompt:

```
> tortoise.data
      Species Endemics   Area Elevation Nearest Scruz Adjacent
Baltra      58      23 25.09      346      0.6  0.6      1.84
Bartolome   31      21  1.24      109      0.6 26.3     572.33
Caldwell     3       3  0.21      114      2.8 58.7      0.78
Champion    25       9  0.10       46      1.9 47.4      0.18
[output truncated]
```

R has read the data and stored it in a *data frame*, which can be thought of as the R equivalent of a spreadsheet: each column contains the values of a single variable. In this case, the variables are all numeric, and R has named them using the information from the first row of the data file. Notice, however, that there is no variable name for the first column in the output above: this column contains the *row names* of the data frame, which in this case are the names of the individual Galapagos islands. R figured this out from the format of the data file: there are seven variable names in the header row, but eight fields in each subsequent line. It is not necessary to include row names in data files: if they are omitted (so that the number of variable names is the same as the number of fields in each subsequent line, R will just number the rows. Later, we will see one way in which row names can be used.

6.2 Examining data frames

For this particular example, it is feasible (although not particularly informative!) to look at `tortoise.data` on screen. For larger data frames, perhaps containing thousands or even hundreds of thousands of rows, this is not feasible. One way to get an idea of the structure of a data frame is to use the `head()` command, which just prints the first few rows. The next script command illustrates this:

```
> head(tortoise.data)
      Species Endemics   Area Elevation Nearest Scruz Adjacent
Baltra      58      23 25.09      346      0.6  0.6      1.84
Bartolome   31      21  1.24      109      0.6 26.3     572.33
Caldwell     3       3  0.21      114      2.8 58.7      0.78
Champion    25       9  0.10       46      1.9 47.4      0.18
Coamano      2       1  0.05       77      1.9  1.9     903.82
Daphne.Major 18      11  0.34      119      8.0  8.0      1.84
```

By looking at the first few lines of the data frame, you can see what variables are present and how they are recorded. In particular, you can see whether they are numeric – as here – or not. Nonetheless, sometimes you may only want to know what variables are present in the data frame. If variable names have been defined, you can use the `names()` command to discover this:

```
> names(tortoise.data)
[1] "Species"  "Endemics" "Area"      "Elevation" "Nearest"  "Scruz"
[7] "Adjacent"
```

Notice the numbers [1] and [7] here: the variable names are a *character vector* i.e. a vector whose elements are character strings. Whenever R prints character strings, they are enclosed in quotation marks "...".

Usually of course, it is not adequate merely to know what variables are present. Before doing anything complicated, we might want to compute some simple summary statistics:

```
> summary(tortoise.data)
  Species      Endemics      Area      Elevation
Min.   : 2.00   Min.   : 0.00   Min.   : 0.0100   Min.   : 25.00
1st Qu.: 13.00  1st Qu.: 7.25   1st Qu.: 0.2575   1st Qu.: 97.75
Median : 42.00  Median :18.00   Median : 2.5900   Median : 192.00
Mean   : 85.23  Mean    :26.10   Mean    :261.7087   Mean    : 368.03
3rd Qu.: 96.00  3rd Qu.:32.25   3rd Qu.: 59.2375   3rd Qu.: 435.25
Max.   :444.00  Max.    :95.00   Max.    :4669.3200   Max.    :1707.00

  Nearest      Scruz      Adjacent
Min.   : 0.20   Min.   : 0.00   Min.   : 0.03
1st Qu.: 0.80   1st Qu.: 11.03  1st Qu.: 0.52
Median : 3.05   Median : 46.65  Median : 2.59
Mean   :10.06   Mean    : 56.98  Mean    :261.10
3rd Qu.:10.03   3rd Qu.: 81.08  3rd Qu.: 59.24
Max.   :47.40   Max.    :290.20  Max.    :4669.32
```

This summarizes each variable in the data frame. In this particular case, the variables are all quantitative (i.e. numeric): R recognizes this and produces an appropriate set of summary statistics. If some of the variables had been categorical, R would have reported a frequency table for these variables (i.e. a tally indicating how often each category occurred).

Of course, summary statistics on their own do not tell the whole story. We might in addition want to look at relationships between the variables. The next command in the script provides an easy way of doing this:

```
> plot(tortoise.data)
```

In response to this, R produces a *scatterplot matrix* showing the relationship between each pair of variables in the data frame. This matrix is displayed in the “Plots” tab of the bottom-right subwindow in RStudio.⁴ In this case, the variables are all numeric so the scatterplots are easily interpretable. The scatterplot matrix shows, for example, that the variables `Species` and `Endemics` are strongly related – this is not surprising, given that the former variable represents the total number of tortoise species on an island and the latter represents the number of these species that are endemic.

Plotting the data enables us to visualize potential relationships between variables. It also provides an opportunity to identify data values that seem out of line with the majority: such values may have a large effect on the subsequent analysis, or may suggest that there is some error in the recorded data. In this particular example, there is a clear outlier on all of the plots involving `Area`, as well as those involving `Adjacent`. Recall (from the comments at the top of the script file) that both of these variables represent the areas of islands. The outlying data point thus represents an unusually large island. The plots indicate that the area of this island exceeds 3000km², whereas all of the remaining islands have areas well below 3000km². On the basis of this, we can find out which is the outlying island as follows:

```
> big.island <- (tortoise.data$Area > 3000)
> big.island
 [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[13] FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[25] FALSE FALSE FALSE FALSE FALSE FALSE
> tortoise.data[big.island,]# It's Isabela, with an area of 4669.32km^2
   Species Endemics   Area Elevation Nearest Scruz Adjacent
Isabela   347      89 4669.32    1707     0.7  28.1   634.49
```

There are only three commands here, but several new concepts:

- The first command assigns the result of `(tortoise.data$Area > 3000)` to an object `big.island`. Note firstly the use of the dollar symbol `$`, which is used to extract a named component of an object. In this case, `tortoise.data$Area` extracts the `Area` column of `tortoise.data`. Note secondly that an expression of this form evaluates to either `TRUE` (in this case, if the island area exceeds 3000km²) or `FALSE` (otherwise). Expressions of this form are called *logical expressions*. Thus, the object `big.island` is a vector, the same length as `tortoise.data$Area`, containing the value `TRUE` for islands larger than 3000km² and `FALSE` for the remaining islands. The output from the `big.island` command shows that there is only one island with an area greater than 3000km², and that this is the 16th island in the data set.
- In the final command, we have used square brackets `[]` again. In Section 5.5, we used square brackets to extract specific elements of a vector. Of course, a data frame consists of both rows and columns: in the

⁴ If the subwindow is too small to see the plots clearly, click the “Zoom” button to view the graphics in an external window – which can be closed again when you’re done.

square bracket notation, these are referenced as `[row(s), column(s)]`. If either index is omitted in an expression, all available values are used. Thus, the expression `tortoise.data[big.island,]` extracts the rows of `tortoise.data` corresponding to `big.island`, and all columns.

There is another difference between the use of square brackets here and in Section 5.5. There, the elements of interest were defined using their numeric positions. Here however, they are defined using a logical vector with `TRUE` elements in the positions required, and `FALSE` elements elsewhere. Both methods are valid. Thus, instead of typing `tortoise.data[big.island,]`, we could have typed `tortoise.data[16,]`: the first version translates as “all rows of `tortoise.data` for which `big.island` is `TRUE`”, and the second as “the 16th row of `tortoise.data`”.

We have thus seen three different ways of extracting subsets of an object:

- By using square brackets `[]` with numeric vectors to select the components of interest – as in `z[c(3,2,5)]` or `tortoise.data[16,]`.
 - By using square brackets `[]` with logical vectors to select the components of interest – as in `tortoise.data[big.island,]`.
 - By using the dollar sign `$` to extract named components of an object – as in `tortoise.data$Area`.
- The final command above contains a comment at the end of the line: `# It's Isabel, with an area of 4669.32km^2`. A comment does not have to occupy a line by itself: sometimes, as here, it is convenient to add a comment at the end of a line of code.

6.3 Customizing graphics

The `plot()` command above is a very convenient way to examine relationships between variables quickly. However, to produce publication-quality graphics it is necessary to exercise finer control over the output. We will illustrate this by plotting the relationship between `Elevation` and `Endemics` – from the earlier scatterplot matrix, it looks as though there may be a roughly linear relationship here. The next line in the script produces a plot involving just these two variables:

```
> plot(tortoise.data$Elevation,tortoise.data$Endemics)
```

Remember that the dollar sign `$` is used to extract named components of an object: thus the arguments to this `plot()` command are vectors containing the `Elevation` and `Endemics` columns of `tortoise.data`. If the first two arguments to the `plot()` command are vectors of equal length, R produces a scatterplot with the first and second vectors on the horizontal and vertical axes respectively. Notice that `plot()` behaves differently here from previously: when we typed `plot(tortoise.data)` we obtained a scatterplot matrix. This is because `plot()` is a *generic function* with different *methods* for different *classes* of object. In plain English: R is intelligent enough to realize that plotting a data frame is different from plotting two vectors, and to act appropriately.

To control the appearance of the plot, we can proceed as follows:

```
> plot(tortoise.data$Elevation,tortoise.data$Endemics,
+       xlab="Elevation (m)",ylab="No. of species",
+       main="Variation of endemic species numbers with\nisland elevation",
+       pch=15,col="blue")
> box(lwd=2)# Nice frame round the plot
```

The first command here is `plot()` again, but this time with many more arguments. These are as follows:

- `xlab`, `ylab`: the labels for the x- and y-axes of the plot.
- `main`: title for the plot. Notice the use of the special code `\n` in the middle of the title: this is a *new line* character, and forces R to write the remainder of the title on a new line (look at the output to see this).
- `pch`: the *plotting character* to use. Plotting character 15 is a filled square. If you want to see what other plotting characters are available, type `help(points)` at the R prompt. Notice that the help page opens in the “Help” tab of the lower right-hand subwindow.
- `col`: the colour to use for the points.

The final command draws a box around the plot, using lines that are twice as wide as the default (`lwd=2`). This can be helpful for reproduction when graphics are submitted for publication or in conference presentations.

Of course, beautiful graphics are of little use unless they can be saved to a file. Fortunately, R can export graphics to many different file formats. The next few lines in the script illustrate this:

```
> dev.copy(pdf, "endemics.pdf", width=6, height=6)
pdf
  4
> dev.off()
RStudioGD
  2
```

The first of these commands translates as “copy the current contents of the graphics window to a PDF file called `endemics.pdf`, which is 6 inches wide and 6 inches high”. R responds with a note to say that graphics output is currently being written to *device 4*, which is a *pdf device*. The second command – `dev.off()` – tells R to stop copying and close the PDF file. R responds to say that graphics output has reverted to *device 2* which is a *RStudioGD device* (i.e. the “RStudio Graphics Device”, which is the “Plots” panel in the lower-right). After running **both** of these commands, you should find the file `endemics.pdf` in your current working directory (verify this by clicking the “Files” tab).

As indicated above, R is able to export graphics to a wide variety of file formats. The script contains comments to show how to create JPEG and encapsulated postscript (EPS) files (publishers often require graphics files in EPS format). To try out these commands, uncomment them and then run them in the usual way. If you want to know what other graphics formats are available, type `help(Devices)` at the R prompt.

6.4 Fitting statistical models

Given the apparent linear relationship between the elevation (i.e. maximum altitude) of an island and the number of endemic species found on it, it may be of interest to quantify this relationship by calculating the least-squares regression line. The command for this is `lm()`, which stands for “linear model”:

```
> endemics.model <- lm(Endemics ~ Elevation, data=tortoise.data)
```

The first argument to the `lm()` command here is `Endemics ~ Elevation`. This is an example of a *model formula*: the special character `~` means “depends on”. The second argument to the `lm()` command tells R that the variables in the model formula can be found in the `tortoise.data` data frame. Thus the code above means “Fit a linear model in which `Endemics` depends upon `Elevation`, where these variables can be found in `tortoise.data`; and store the result in the object `endemics.model`”.

To view this result, type `endemics.model`:

```
> endemics.model
```

Call:

```
lm(formula = Endemics ~ Elevation, data = tortoise.data)
```

Coefficients:

```
(Intercept)    Elevation
      7.1827      0.0514
```

The output contains the original call to the `lm()` function, along with the coefficients of the fitted regression line which is $y = 7.1827 + 0.0514x$, where y and x denote respectively the number of endemic species and the elevation. On its own however, this is not very informative. We might, for example, want to test for an association between `Elevation` and `Endemics` or to calculate the proportion of variance explained by the model (otherwise known as the coefficient of determination, or R^2):

```
> summary(endemics.model)
```

[output truncated]

Coefficients:

```
              Estimate Std. Error t value Pr(>|t|)
(Intercept)  7.182682   4.138088   1.736   0.0936 .
Elevation    0.051401   0.007465   6.886 1.75e-07 ***
---
```

```
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Residual standard error: 16.95 on 28 degrees of freedom

Multiple R-squared: 0.6287, Adjusted R-squared: 0.6154

F-statistic: 47.41 on 1 and 28 DF, p-value: 1.751e-07

The result is very similar to regression model output for other statistical software packages: it contains a table giving the estimated regression coefficients along with standard errors, test statistics and p -values for testing

the null hypothesis that the true value of each coefficient is zero. The tiny p -value for `Elevation` suggests that at any reasonable level of significance we should reject this null hypothesis: we conclude that there is a genuine relationship between the elevation of an island and the number of endemic species found there (remember, however, that we cannot draw any conclusions about cause and effect from this!). The output also gives the multiple and adjusted R^2 statistics.

This is the second time that we have used the `summary()` command. The first time was to summarise the variables in `tortoise.data` (Section 6.2). This time however, it produces completely different output. Like `plot()`, `summary()` is a generic function, and it does different things for different classes of object. We will see this several times when we start to use `Rglmclim`.

You may want to add the fitted regression line to the scatterplot produced earlier. Your graphics window should still contain this scatterplot (if not, repeat the earlier commands to recreate it). To add the regression line:

```
> abline(endemics.model, col="red", lty=2, lwd=2)
```

The command `abline()` adds the line with equation $y = a + bx$ to the current plot. There are many ways of specifying the values of a and b – to see all of them, type `help(abline)` at the R prompt. In this particular case, R extracts the appropriate coefficients automatically from the `endemics.model` object, recognizing that this corresponds to a linear regression model.

The remaining arguments to the `abline()` command above are optional, and have been included to improve the appearance of the plot. They are as follows:

- `col`: the colour of the line (recall that the same argument was used in the earlier `plot()` command to control the colour of the points).
- `lty`: the line type. A value of 1 corresponds to a solid line, a value of 2 to a dashed line, a value of 3 to a dotted line and so on.
- `lwd`: the line width. As with the earlier `box()` command, we use a double-width line for emphasis.

Finally: all statistical models are based on assumptions, and the validity of these assumptions should be checked before drawing conclusions from the model (strictly speaking therefore, we should have checked the assumptions before testing hypotheses about the relationship between `Elevation` and `Endemics!`). For most statistical models, a wide variety of graphical diagnostics are available for checking assumptions. The `plot()` command can be used again:

```
> par(mfrow=c(2,2))
> plot(endemics.model)
> par(mfrow=c(1,1))
```

The first command here tells R to split the graphics window into two rows and two columns: this is because the subsequent `plot()` command produces four plots in total, and it is helpful to see them all at the same time. The final command ensures that any subsequent plots will occupy the entire graphics window again.

For linear models, the `plot()` command produces a different result again (recall that we have already used it to plot a data frame and to produce a scatterplot of two variables). Here, the plots produced are as follows:

- A plot of the *residuals* against the *fitted values*. The fitted values are the predicted numbers of endemic species according to the fitted model; the residuals are the differences between the observed and predicted numbers. The idea is that if the model is an adequate fit, the residuals should be randomly scattered around zero and should not show any obvious relationship with the fitted values. The red curve on the plot is a *nonparametric regression curve* and is intended to indicate any potential relationships that may be present. Notice also that the three points with the largest (positive or negative) residuals are labeled on the plot: the labels are taken from the row names of `tortoise.data` (see Section 6.1). In this case, the row names are the names of the individual islands, so we can see immediately the islands for which the model fit is potentially problematic.
- A *normal quantile-quantile (Q-Q) plot* of the residuals. This is intended to assess the assumption that the residuals from the linear model are normally distributed: if so, the points on this plot should fall on a straight line. Here, there are apparent departures from linearity at each end of the plot, suggesting that the normality assumption may not be satisfied. Once again, the worst offenders are labeled: this enables us to investigate these data points in more detail and, potentially, to identify ways of improving the model.
- A *scale-location plot*, which is intended to check the assumption that the residuals have a common variance. In this example, it seems that the variability of the residuals increases with the fitted values, so that this assumption is not satisfied.
- A plot of *residuals* versus *leverage*, with contours of *Cook's distance* superimposed.

If you haven't seen these kinds of plot before, don't worry – they aren't critical, and we will see similar ideas when we start to use Rglimclim. The main message for the moment is that it is easy to check modelling assumptions using R!

There are a few other commands at the end of the script – don't run these yet, because they require an additional package (see Section 7 below).

6.5 Running commands in batch mode

So far, we have used R by typing (or copying and pasting) individual commands to the prompt; and the output in the Console subwindow consists of commands interspersed with the results that they produce. For more extensive analyses however, this is slightly cumbersome. Fortunately, R also offers the opportunity to run commands in *batch mode* and to write the results to a file if required. The main things to know about this are:

- To run commands in batch mode, you need to save them in a script file and to run this using the `source()` command.
- To get output to appear when commands are run in batch mode, you need to use the `cat()` and `print()` commands.
- To write output to a file instead of to the console, use the `sink()` command; and to return output to the console afterwards, use the `sink()` command again.

The script `R_SelfStudy_Clean.r` illustrates this. It is a slightly amended version of `R_SelfStudy.r`. Open the new script and read through it, noticing the use of `sink()`, `cat()` and `print()`. Then run it by clicking the "Source" button at the top of its subwindow. You will not see anything in the console, although you will see new plots in the graphics window. After the script has finished however, you will find a file called `Galapagos_Results.txt` in your working directory: click on this in the "Files" tab to view the results.

Hopefully, this example has given you a sense of how R works and what it can do. However, it has not been possible to explain everything here. If you have any questions, please note them down – they can be addressed during the training sessions.

7. Additional R packages

As explained previously, one of the key features of R is that many add-on packages are available. One way to see them is to browse the "Packages" pages on the CRAN (see Section 4.1). The number of available packages is quite bewildering, however! Fortunately, if you know which packages you want, and if they are available from CRAN, it is very easy to locate and install them providing you have an internet connection.

Using RStudio, the easiest way to install a small number of additional packages is via the `Install Packages` option in the drop-down `Tools` menu. If you have a large number of packages to install however, it is usually easier to use the `install.packages()` command at the R prompt. There are a few packages to install here, so we'll use a script that uses `install.packages()` to automate the procedure – details are below.

Not all packages are available from CRAN, however. For example, Rglimclim is not on CRAN. Instead, it is available from <http://www.homepages.ucl.ac.uk/~ucakarc/work/glimclim.html>. To make things easier however, the installation files `Rglimclim_1.2-4.zip` (for Windows) and `Rglimclim_1.2-4.tar.gz` (for other operating systems) are provided in `RglimclimPrep.zip`: you should find them in the folder you created in Section 4.3 (if not, you made a mistake either creating the folder or unpacking the zip archive!). The installation script contains the necessary code to install it automatically for you.

To install the required add-on packages now, proceed as follows:

1. Ensure that your computer is connected to the internet.
2. Start up RStudio, if it is not already running; and ensure that you have sufficient administrative privileges to install software from within it.⁵
3. Change to the directory where you unpacked the `RglimclimPrep.zip` (see Section 5.7 for details on how to do this)
4. Type `source("RglimclimSetup.r")` at the console prompt, and press <Return>.

This should produce the following result (if you do *not* see this, read on for possible explanations!):

⁵ If you are using Windows, you may need to right-click the icon and select "Run as administrator". Under Linux, you may need to start RStudio from a terminal, using a command such as `sudo rstudio &` (note that this is case-sensitive) to ensure that you have the necessary permissions.


```
Installing Rglimclim ...
Installing package into 'C:/Users/richard/Documents/R/win-library/3.0'
(as 'lib' is unspecified)
package 'Rglimclim' successfully unpacked and MD5 sums checked
Installing packages from CRAN ...
[output truncated]
```

If you see something like this:

```
> source("RglimclimSetup.r")
Error in file(filename, "r", encoding = encoding) :
  cannot open the connection
In addition: Warning message:
In file(filename, "r", encoding = encoding) :
  cannot open file 'RglimclimSetup.r': No such file or directory
```

it means that R could not find the file `RglimclimSetup.r`. The reason is probably EITHER that you have not changed to the correct working directory (see Section 5.7), OR that you did not unpack `RglimclimPrep.zip` correctly in Section 4.3.⁶

To check that Rglimclim has installed properly now, from the RStudio console type

```
> library(Rglimclim)
```

If you see the message

```
Use 'help("Rglimclim-package")' to get started
```

then you are ready to come to Trieste. Well done!

8. Postscript: the case study area

In case you feel disappointed that this introduction has not covered anything that is directly relevant to downscaling or to the development of stochastic weather generators: you might be interested at least to see the area that we will study during the Rglimclim practical sessions. There are a few commands at the end of the `R_SelfStudy.r` script (see Section 6) to produce a map of the area. By now you should know how to start up RStudio, set the working directory correctly (if you're not already there), open the script and scroll to the bottom.

We're going to use a command from the `lattice` graphics package, to produce a map of the area with an altitude scale (there are other commands that can be used to produce maps, without loading additional packages; but they don't automatically produce altitude scales). The first step is to load the `lattice` package:

```
> library(lattice)
```

If you already had R on your computer before installing the `lattice` package, you may get a warning message such as `package 'lattice' was built under R version 3.0.3` (or something). This means that the version of R on your machine is slightly out of date. This is not usually a problem, providing your R installation is not *too* old.

The next step is to read the data that will be used to produce the map. These data are provided in file `Topography.rda`, which is a binary file in "R data format".

```
> load("Topography.rda")
```

If you look at the "Environment" tab in the top right-hand subwindow now, you should see an object called `topo.data` which is described as a "Large matrix (273600 elements, 2.1Mb)"; also some vectors `topo.lats` and `topo.longs`. The `topo.data` matrix contains a gridded representation of the topography of the area, at roughly $1 \times 1 \text{ km}^2$ resolution; the data are altitudes in hundreds of metres, and are from the GTOPO30 Digital Elevation Model at http://webmap.ornl.gov/wcsdown/dataset.jsp?ds_id=10003. The `topo.lats` and `topo.longs` vectors merely contain the latitude and longitude coordinates corresponding to each dimension of the matrix.

⁶ Users of other operating systems may have more difficulties here – for example, Linux users may need to type `sudo apt-get install tcl8.5-dev` and `sudo apt-get install tk8.5-dev` from a terminal before attempting the installation above. If you're using Linux, I assume you know what you're doing ...

The study region is not even approximately the same shape as the RStudio graphics window. To produce a reasonable map, therefore, it is helpful to open a new graphics device with specified dimensions:

```
> x11(width=8,height=4)
```

The `width` and `height` arguments here are in inches (an inch is about 25mm). Now you can produce your map:

```
> filled.contour(topo.longs,topo.lats,100*topo.data,
+   color.palette=terrain.colors,
+   plot.title=
+     title("Topographic map of northern Iberia",
+       xlab="Longitude (degrees)",
+       ylab="Latitude (degrees)"),
+   key.title=title("Altitude (m)",cex.main=0.8))
```

The syntax for this command is quite complicated; however, hopefully you can look at the resulting plot and figure out what the various options are doing. If you want to know more about this command, type `help(filled.contour)`. For most graphics commands in R, the syntax is simpler than this: the `lattice` package is not very intuitive. The results are impressive, however.

You have already seen (Section 6.3) the use of the `dev.copy()` command to export graphics to a file. We'll finish by creating a PDF file containing the map:

```
> dev.copy(pdf,"NorthernIberiaMap.pdf",width=8,height=4)
pdf
  4
> dev.off()
RStudioGD
  2
```

Notice that by specifying the same width and height (8 inches and 4 inches) that were used to open the graphics window, you can guarantee that the resulting graphics file will look exactly the same as it did on screen. This can be very useful.