# Chapter 1 A brief introduction to R

## 1.1 What is R?

R is an object-oriented programming language and environment for statistical computing that provides relatively simple access to a wide variety of statistical techniques (R Core Team [2016](https://remot-technologies.com/introduction-to-r/module-1/intro.html#ref-R-base)). R offers a complete programming language with which to add new methods by defining functions or automating iterative processes.

Many statistical techniques, from the classic to the latest methodologies, are available in R, with the user in charge of locating the package that best suits their needs.

R can be considered as an integrated set of programs for data manipulation, calculation and graphics. Among other features R allows:

* effective data storage and manipulation,
* operators for calculation on indexed variables,
* a comprehensive, coherent and integrated collection of data analysis tools,
* plotting possibilities, which work directly on screen or printer, and
* a well-developed, simple and effective programming language, including conditionals, loops, recursive functions and the possibility of inputs and outputs.

R is distributed as open source software, so obtaining it is completely free.

R is also multiplatform software which means it can be installed and used in various operating systems (OS), mainly Windows and Linux. However, the available functions and packages syntax is practically the same in any OS. From an operational point of view, R consists of a base system and additional packages that extend its functionality. Among the main types of packags we found:

* Those that are part of the base system (ctest).
* Those that are not part of the base system, but are recommended (*survival*, *nlme*). In GNU/Linux and Windows these packages are already part of the standard distribution.
* Other packages such as *UsingR*, *foreing*, or *Maptools*. These must be selected and installed individually. We will see how to do this later.

The functions included in the packages installed by default, that is, those that are predefined in the basic installation R, are available for use at any time. However, **in order to use the functions of new packages, specific calls must be made to those packages**.

### 1.1.1 Getting and installing R

The installation of R depends on the operating system to be used. You can find all the necessary information in:

* <http://cran.r-project.org/>
* Windows: <http://cran.r-project.org/bin/windows/>
* Linux: <http://cran.r-project.org/bin/linux/>

For the development of this course I will use the Windows version but feel free to use whatever version fits your needs. The last version of R is downloadable from [here](https://cran.r-project.org/bin/windows/base/). We will install the latest version available. Remember that you have to install the version that corresponds to the architecture of your OS (32 or 64 bits). In case of doubt install both versions or at least the 32-bit version, which will always work on our computer.

Installation in Windows is very simple. Just run the executable (.exe) file and follow the installation steps (basically say *Yes* to everything). Once R is installed, we will install **RStudio** an integrated development environment (IDE) that is more user-friendly than the basic R interface. RStudio provides a more complete environment and some useful tools such as:

* Autocomplete instructions[1](https://remot-technologies.com/introduction-to-r/module-1/intro.html#fn1).
* Object management.
* Data display and visualization.
* Exporting plots and figures.

We will install the latest version of [RStudio](https://www.rstudio.com/products/rstudio/download/). You can see the installation steps in the installation video tutorial.

* Note that RStudio is just an *interface*. Any code block or instruction will work in any other R environment.
* Before installing RStudio you need to have already installed the standard R software.

### 1.1.2 Documentation, manuals and help

Being an open source software and with a strong collaborative component R has a large amount of resources and documentation relative to the specific syntax of the language itself (control structures, function creation, calls to objects …) and to every single package available as well.

On the other hand, R counts on a series of manuals which are available right after installing the software. You can find them in the installation directory of R (C:Files-X.X.X). These manuals and many others can also be downloaded from the [R project website](https://cran.r-project.org/):

* Writing R extensions.
* R data import / export.
* The R language definition.
* R installation and administration.
* An introduction to R.

Finally, in addition to the wide repertoire of manuals available, there is also a wide range of resources and online help including:

* <http://www.r-bloggers.com/>. A website dedicated to R development of tutorials.
* <http://www.r-project.org/mail.html>. R help mailing lists with various interest groups including the R (r-sig-geo) GIS user community.
* <http://stackoverflow.com/>. Website devoted to questions-and-answers on programming languages among which is R also available[2](https://remot-technologies.com/introduction-to-r/module-1/intro.html#fn2).
* <http://www.r-tutor.com/>. A website devoted to teach statistics. An useful one if you are not much familiar with basic statistical methods.

It is relatively important to become familiar from the beginning with the various alternatives for getting help. A key part of your success in using R lies in your ability to be self-relient and be able to get help and apply it to your own problems.

## 1.2 Starting with R

### 1.2.1 The R environment

R is basically a command line environment that allows the user to interact with the system to enter data, perform mathematical calculations or visualize results through plots and maps.

Figure [1.1](https://remot-technologies.com/introduction-to-r/module-1/intro.html#fig:figure1) shows the *standard* appearance of the R console, which can be considered as a windows *cmd* like *terminal* or *console*. We have also seen what the terminal and working environment looks like in RStudio at the end of the installation process (Figure [1.2](https://remot-technologies.com/introduction-to-r/module-1/intro.html#fig:figure2)). A third possibility is to work directly on the *cmd* terminal (Figure [1.3](https://remot-technologies.com/introduction-to-r/module-1/intro.html#fig:figure3)). The commands and instructions are the same regardless of the environment that we chose. In this course we will focus on the use of RStudio, since it is the simplest of them all.

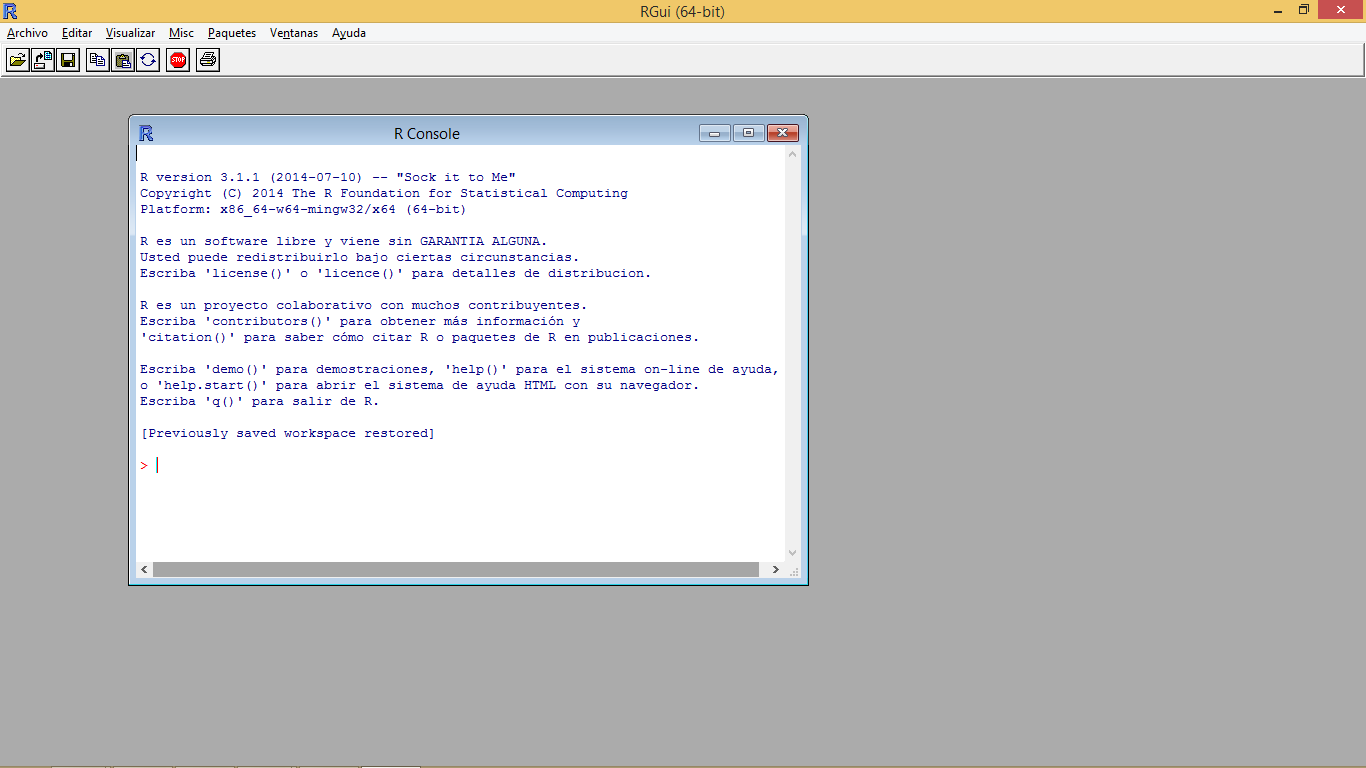


Figure 1.1: R standard terminal

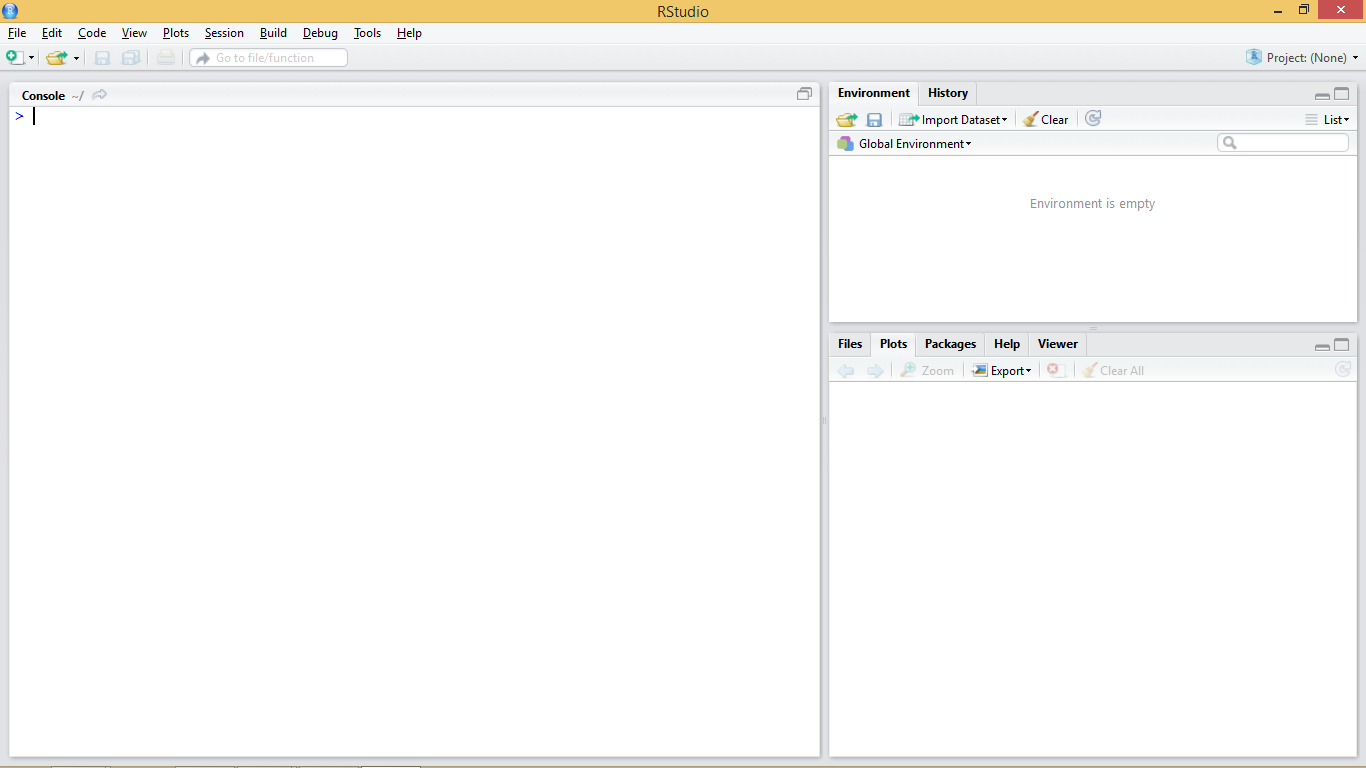


Figure 1.2: RStudio interface

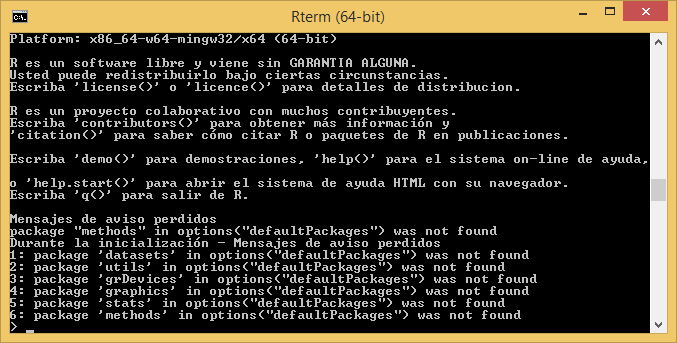


Figure 1.3: R cmd terminal

The terminal (regardless of the chosen option) is usually the main working window and is where we will introduce the necessary instructions to carry out our operations. It is in this window where we will visualize the results from most instructions and objects we generate. An exception to this are plots and maps which are displayed in a specifically-devoted window located in the bottom-right corner.

## 1.3 Working with R

Let’s get started and insert our first command in the R’s terminal. When R is ready or awaiting us to input an instruction the terminal shows a cursor right after a *>* symbol to indicate it.

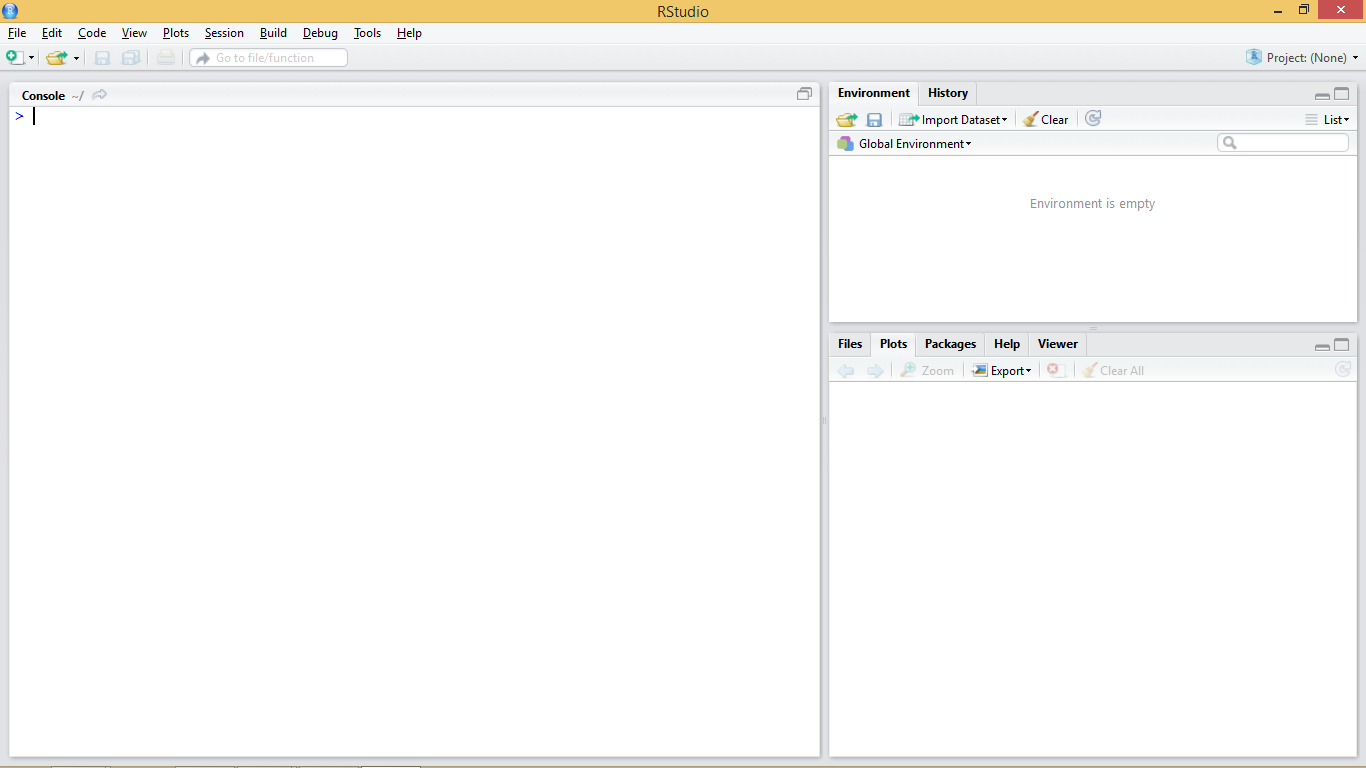


Figure 1.4: R waiting for instructions

At this point it is necessary to take into account that R is an interpreted language, which means that the different instructions or functions that we specify are read and executed one by one. The procedure is more or less as follows. We introduce an instruction in the R console, the application interprets and executes it, and finally generates or returns the result. To better understand this concept we will do a little test using the R console as a calculator. Open the working environment of RStudio if you did not already open it and enter the following statement and press enter:

10+2

## [1] 12

What just happened is that the R interpreter has read the instruction, in this case a simple arithmetic operation, executed it and returned the result. This is the basic way to proceed to enter operations. However, it will not be necessary for us to always enter the instructions manually. Later we will see how to create scripts or introduce blocks of instructions.

## 1.4 Objects in R

We have previously mentioned some features of R such as that R is an object-oriented language. But, what does this mean? It basically means that to perform any type of task we use objects. Everything in R is an object (functions, variables, results …). Thus, entities that are created and manipulated in R are called objects, including data, functions and other structures.

Objects are stored and characterized by their name and content. Depending on the type of object we create that object will have a given set of characteristics. Generally the first objects one creates are those of variable type, in which we will be able to store a piece of data and information. The main objects of variable type in R are:

* Number: an integer or decimal number depending on whether we specify decimal figures.
* Factor:a categorical variable or text.
* Vector: a list of values of the same type.
* Array: a vector of k
* dimensions.
* Matrix: a particular case of array where k=2
* (rows, cols).
* Data.frame: table composed of vectors.
* List: vector with values of different types.

Obviously there are other types of objects in R. For example, another object with which we are going to familiarize ourselves is the *model* objects. We can create them by storing the output of executing some kind of model like a linear regression model for instance. Spatial data also fits in its particular variety of objecto. Thororugh the course we will see both models and spatial information (vector[3](https://remot-technologies.com/introduction-to-r/module-1/intro.html#fn3) and raster).

### 1.4.1 Creating objects

Objects in R are created by declarating a variable by specifing its name and then assign it a value using the <- operator. We can also use = but the <- operator is most commonly found in examples and manuals.

So, to create an object and assign it a value the basic instruction is composed of object name <- value.

n <- 4

Prueba a introducir las siguientes instrucciones para crear distintos tipos de objeto:

n <- 15x <- 1.0name <- "Marcos"

We can also store in an object the result form any operarion:

n <- 10+2

So here is the thing. The type of object we create depends on the content that we assign. Therefor, if we assign a numeric value, we are creating an object of type number (integer or decimal) and if we assign a text string (any quoted text, either with single or double quotes), we are creating a text type object or string. Once created, the objects are visualized using calls using the name that we have assigned to the object. That is, we will write to the terminal in the name of the object and then its value will be shown.

n

## [1] 12

name

## [1] "Marcos"

Some considerations to keep in mind when creating objects or working with R in general lines:

1. R is **case-sensitive** so radio ≠ Radio
2. If a new value is assigned to an object it is overwritten and **deletes the previous value**.
3. Textual information (also knwon as string or char) is entered between quotation marks, either single ('') or double ("").
4. The function ls() will show us in the terminal the objects created so far.
5. If the value obtained from an instruction is not assigned in an object it is only displayed in the terminal, it is not stored.

### 1.4.2 Vectors

One of the most common objects in R is the vector. A vector can store several values, which must necessarily be of the same type (all numbers, all text, and so forth). There are several ways to create vectors. Try entering the following instructions and viewing the created objects:

v1 <- c(1, 2, 3, 4, 5)v1

## [1] 1 2 3 4 5

v2 <- 1:10v2

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

v3 <- -5:3v3

## [1] -5 -4 -3 -2 -1 0 1 2 3

v4 <- c('spatial','statistics','rules!!')v4

## [1] "spatial" "statistics" "rules!!"

We have just covered the basic methods for vector creation. The most common approach is use the function c() which allow as to introduce values manually by separatting them using ,.

v1 <- c(1, 2, 3, 4, 5)v1

## [1] 1 2 3 4 5

v4 <- c('spatial','statistics','rules!!')v4

## [1] "spatial" "statistics" "rules!!"

Another option that only works for vectors contaning integer values is the use of : which produces a ordered sequence of numbers by adding 1 starting from the first value and finishing in the last.

v2 <- 1:10v2

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

v3 <- -5:3v3

## [1] -5 -4 -3 -2 -1 0 1 2 3

Vectors, lists, arrays, and dataframes are indexed objects. This means that they store several values and assign to each of them a numerical index that indicates their position within the object. We can access the information stored in each of the positions by using name[position]:

v1[1]

## [1] 1

Note that opposite to most of the other programming languages, the index for the first position in an indexed object is 1, whereas Python, C++ and others use 0.

As with an unindexed object, it is possible to modify the information of a particular position using the combination name[position] and the assignment operator <-. For example:

v3[9] <- 10000v3[9]

## [1] 10000

Let’s see some specific functions and basic operations for vectors and other indexed objects:

* length(vector): Returns the number of positions of a vector.
* Logical operators <,>, ==,!=: Applying these operators on a vector returns a new vector with values TRUE/FALSE for each of the positions of the vector, depending on whether the given values satisfies or not the condition.

length(v3)

## [1] 9

v4<-1:5v4>3

## [1] FALSE FALSE FALSE TRUE TRUE

### 1.4.3 Lists

Once we have seen vectors we go to explore how objects of type list work. A list is an object similar to a vector with the difference that lists allow to store values of different type. Lists are created using the list(value1, value2, ...) function. For example:

list1 <- list(1,7,'Marcos')list1

## [[1]]## [1] 1## ## [[2]]## [1] 7## ## [[3]]## [1] "Marcos"

To access the values stored in the different positions proceed in the same way we did with vectors, ie name[position]:

list1[3]

## [[1]]## [1] "Marcos"

We can use the length() function with list too:

length(list1)

## [1] 3

### 1.4.4 Arrays

Arrays are an extension of vectors, which add additional dimensions to store information. The most common case is the 2-dimensional matrix (rows and columns). To create an array, we use array(values, dimensions). Both values and dimensions are specified using vectors. In the following example we see how to create a matrix with 4 rows and 5 columns, thus containing 20 values, in this case correlative numbers from 1 to 20:

myarray<- array(1:20,dim=c(4,5))myarray

## [,1] [,2] [,3] [,4] [,5]## [1,] 1 5 9 13 17## [2,] 2 6 10 14 18## [3,] 3 7 11 15 19## [4,] 4 8 12 16 20

To access the stored values we will use a combination of row-and-column positions like matrix[row, col], where row indicates the row postition and col the column one. If we only assign value to one of the coordinates ([row,] or [,col]) we get the vector corresponding to the specified row or column.

myarray[3,2]

## [1] 7

myarray[3,]

## [1] 3 7 11 15 19

myarray[,2]

## [1] 5 6 7 8

### 1.4.5 Data.frame

A data frame is used for storing data tables. It is a list of vectors of equal length. For example, the following variable df is a data frame containing three vectors n, s, b.

n <- c(2, 3, 5) s <- c("aa", "bb", "cc") b <- c(TRUE, FALSE, TRUE) df <- data.frame(n, s, b)

As you can see, the function data.frame() is used to create the data.frame. However, we will seldom use these function to create objects or store data. Normally, we will call an instruction to read text files containing data or call data objects available in some packages.

## 1.5 Object management

We have seen so far aspects related to the creation of objects. However we should be also know how many objects we have created in our session and how to remove them if necessary. To display all created objects we use ls(). Deleting objects in R is done by the command *remove* rm(object) and then call to the *garbage collector* with gc() to free-up the occupied memory.

ls()

## [1] "b" "col2" "cols" "cols2" "cols3" ## [6] "col.Y\_INDEX" "data" "desv" "df" "fires" ## [11] "list1" "myarray" "n" "name" "ndvi" ## [16] "regression" "row1" "row2" "row3" "s" ## [21] "table" "v1" "v2" "v3" "v4" ## [26] "x"

rm(n)gc()

## used (Mb) gc trigger (Mb) max used (Mb)## Ncells 1249546 66.8 2164898 115.7 2164898 115.7## Vcells 2482625 19.0 5372895 41.0 5372893 41.0

If we want to removal **al** the objects we have currently in our working session we can pass a list object containing the names of all objects to the gc() function. If you are thinking on combining gc()and ls()you are right. This would be the way:

rm(list=ls())gc()

## used (Mb) gc trigger (Mb) max used (Mb)## Ncells 1249167 66.8 2164898 115.7 2164898 115.7## Vcells 2480615 19.0 5372895 41.0 5372893 41.0

## 1.6 Functions and arguments

Up to this point we have seen and executed some instructions in R, generally oriented to the creation of objects or realization of simple arithmetic operations.

However, we have also executed some function-type statements, such as the length() function. A function can be defined as a group of instructions that takes an input, uses this input to compute other values and returns a result or product. We will not go into very deep details, at least for now. It suffices to know that to execute a function it is enough to invoke the instruction that calls the desired function (length) and to specify the necessary inputs, also knwon as *arguments*. These inputs are always included between the parentheses of the instruction (length(vector)). If several arguments are needed we separate them usign ,.

Sometimes we can refer to a given argument by using the argument’s name as is the case of the example we saw to delete all the objects in a session rm(list=ls()).

## 1.7 Scripts in R

So far we have inserted instructions in the console but this is not the most efficient way to work. We will focus on the use of scripts which are an **ordered set of instructions**. This means we can write a text file with the instructions we want to insert and then run them at once.

RStudio has an script development environment which opens in the top-left window. We can access the scripting window pressing *File/New File/R script*.

For additional information visit the [RStudio support site](https://support.rstudio.com/hc/en-us/articles/200484448-Editing-and-Executing-Code).

### References

R Core Team. 2016. *R: A Language and Environment for Statistical Computing*. Vienna, Austria: R Foundation for Statistical Computing. [https://www.R-project.org/](https://www.r-project.org/).

# Chapter 2 Working with files

So far we have seen how to enter data and create objects manually, but it is also possible, and in fact is most common, to read data from files and store it in an object. If the target data file is properly structured, we will create a matrix or a ‘data.frame’ object which we can manipulated afterwards.

## 2.1 The working directory

Using data files normally requires us to specify the location of that file using paths. To avoid this, R has a tool that allow us to specify a target folder -the so-called **working directory or folder**- to work with.

The working directory is the default path for reading and writing files of any kind. We can know the path to the current working directory using the getwd() command. To set a new working directory, we use the command setwd("path"). Remember that setwd() requires a string argument (whereas getwd() does not) to specify the path to the working directory (“path”).

getwd()setwd('C:/Users/Marcos/Desktop/')getwd()

By default, in Windows the working directory is set to the **Documents** folder.

Once the working directory is specified, everything we do in R (read files, export tables and/or graphics …) will be done in that directory. However, it is possible to work with other file system locations, specifying a different path through the arguments of some functions.

## 2.2 Reading data files

R allows you to read any type of file in ASCII format (text files). The most frequently used functions are:

* read.table() and its different variations
* scan()
* read.fwf()

For the development of this course we will focus on using the read.table() function as it is quite versatile and easy to use. Before starting to use a new function we should always take a look at the available documentation. I will take this oportunity to show you how to do this in R so that you begin to become familiar R help.

Any available function in R, regardless of being a stantard one or belonging to an imported package has a documentation entry in the R help. R help describes us in detail the use of any function, providing information of the different arguments of the function, argument types, defaults, reference to the method (when applies) and even short code examples. To access the help we will use the help() function as follows:

help(read.table)

Entering the help() function we access the manual. In this case we can see a brief description of the function read.table() and its different variants (read.csv(), …). Below is the description of the arguments of the function, followed by some examples of application. This is the usual procedure for all functions available in R.

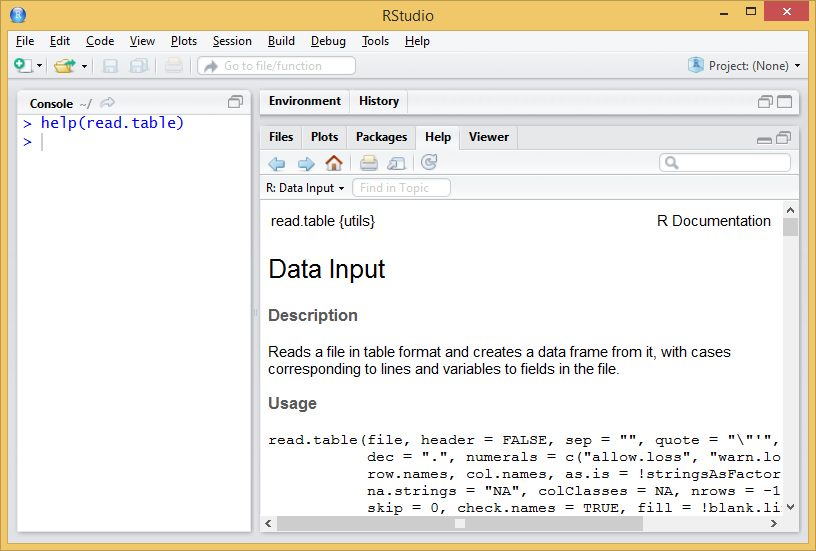


Figure 2.1: Help example

In case we use the regular R interface or the *cmd* terminal, the help entry will open in our default web browser.

At first glance the read.table() function seems to require many arguments, while its variations seem simpler. As already mentioned, arguments are just parameters that we can specify or change to execute a function, thus *tunning* the operation of the functions and the result to be obtained.

Arguments are specified within the function separated by commas (“,”). However, it is not necessary to assign a value to each one of them, since in the case of omitting an argument it is assigned a default value (always reported in the help entry). In the case of read.table() *versus* other functions like read.csv() the main advantage of using the first is that we can manipulate any argument, whereas most of them are fixed in the later. read.csv() is desigend to open comma separate files following the north-american standard (, as field delimiter and . as decimal separator). On the other hand, read.table() can potentially open any text file regardless of the separator, encoding, decimal format and so on so forth. Most of the executions of read.table() consist of:

read.table(file, header = TRUE, sep = “;”, dec = “,”)

* file: path and name of the file to open.
* header: TRUE/FALSE argument to determine whether the first row of the data file contains column names.
* sep: field or column separator[4](https://remot-technologies.com/introduction-to-r/module-1/working-with-files.html#fn4).
* dec: decimal separator.

Let’s see an example of reading file. We will read the file *coordinates.txt*, located in the Data directory. The file is structured in 3 columns with heading and separated by “;”. Al data are integer so no decimal separator is needed. See Table [2.2](https://remot-technologies.com/introduction-to-r/module-1/working-with-files.html#fig:table1).

Show

entries

Search:

|  |  |  |  |
| --- | --- | --- | --- |
|  | **FID\_1** | **X\_INDEX** | **Y\_INDEX** |
| 1 | 364011 | 82500 | 4653500 |
| 2 | 371655 | 110500 | 4661500 |
| 3 | 487720 | 55500 | 4805500 |
| 4 | 474504 | 28500 | 4783500 |
| 5 | 436415 | 85500 | 4729500 |
| 6 | 457549 | 38500 | 4757500 |
| 7 | 469377 | 39500 | 4775500 |
| 8 | 397162 | 124500 | 4687500 |
| 9 | 434666 | 41500 | 4727500 |
| 10 | 478383 | 49500 | 4789500 |

Showing 1 to 10 of 14 entries

Previous12Next

Figure 2.2: Structure of the coordinates.txt file.

The process to follow is:

1. Set working directory.
2. Use the function to save the data in an object.

If we do…

setwd('/media/marcos/6C0C-8C4F/module-1/')read.table('Data/coordinates.txt',header=TRUE,sep=';')

…as we have not specified any object in which to store the result of the function, the contents of the file are simply printed on the terminal. This is a usuall mistake, don’t worry. To store and later access the contents of the file we will do the following:

setwd('/media/marcos/6C0C-8C4F/module-1/Data')table<-read.table('coordinates.txt',header=TRUE,sep=';')table

Please note that:

* You have to specify your own working directory.
* The path to the directory is specified in text format, so you type **“in quotation marks”**.
* The name of the file to be read is also specified as text.
* The header argument only accepts TRUE or FALSE values.
* The sep argument also requires text values to enter de separator[5](https://remot-technologies.com/introduction-to-r/module-1/working-with-files.html#fn5).
* It is advisable to save the data an object (table <-).

The result of read.table() is an array stored in an object named table. Since read.table() returns an array we can manipulate our table using the same procedure described in the [Arrays](https://remot-technologies.com/introduction-to-r/module-1/intro.html#arrays) section.

table[1,]

## FID\_1 X\_INDEX Y\_INDEX## 1 364011 82500 4653500

table[,1]

## [1] 364011 371655 487720 474504 436415 457549 469377 397162 434666 478383## [11] 488973 394153 426962 362216

table[1,1]

## [1] 364011

Below we found some interesting functions to preview and verify the information, and to know the structure of the data we have just imported. These functions are normally used to take a quick look into the first and last rows of an array or data.frame object and also to describe the structure of a given object.

* head(): displays the first rows of the array.
* tail(): displays the last rows of the array.
* str(): displays the structure and data type (factor or number).

head(table)

## FID\_1 X\_INDEX Y\_INDEX## 1 364011 82500 4653500## 2 371655 110500 4661500## 3 487720 55500 4805500## 4 474504 28500 4783500## 5 436415 85500 4729500## 6 457549 38500 4757500

tail(table)

## FID\_1 X\_INDEX Y\_INDEX## 9 434666 41500 4727500## 10 478383 49500 4789500## 11 488973 329500 4807500## 12 394153 77500 4684500## 13 426962 36500 4718500## 14 362216 148500 4651500

str(table)

## 'data.frame': 14 obs. of 3 variables:## $ FID\_1 : int 364011 371655 487720 474504 436415 457549 469377 397162 434666 478383 ...## $ X\_INDEX: int 82500 110500 55500 28500 85500 38500 39500 124500 41500 49500 ...## $ Y\_INDEX: int 4653500 4661500 4805500 4783500 4729500 4757500 4775500 4687500 4727500 4789500 ...

As already mentioned, read.table() is adequate to start reading of files to incorporate data into our working session in R. In any case we must be aware that there are other possibilities such as the read.csv() and read.csv2() that we have already seen when accessing the read.table() description. These functions are variations that defaults some arguments such as the field separator (columns) or the decimal character. In the help of the function you have information about it.

## 2.3 Writing text files

Of course, we can also write text files from our data. The procedure is quite similar to read data but using write.table() instead of read.table. Remember that the created files are saved into the working directory, unless you specify an alternative path in the function arguments. As always, before starting the first thing is to consult the help of the function.

help("write.table")

The arguments of the function are similar to those already seen in read.table ():

write.table (object, file, row.names, sep)

* object: object of type matrix (or dataframe) to write.
* file: name and path to the created file (in text format).
* row.names: add or not (TRUE or FALSE) queue names. FALSE is recommended.
* sep: column separator (in text format).
* dec: decimal separator (in text format).

Try the following instructions[6](https://remot-technologies.com/introduction-to-r/module-1/working-with-files.html#fn6) and observe the different results:

write.table(table,'table1.txt',row.names=TRUE,sep='\t')write.table(table,'table2.txt',row.names=FALSE,sep='\t')write.table(table,'table3.txt',row.names=FALSE,sep=';')write.table(table,'C:/Users/Marcos/Desktop/table4.txt',sep=';')write.csv(table,'C:/Users/Marcos/Desktop/tabla5.csv')

## 2.4 Data manipulation

Let’s see the most common instructions for manipulating and extracting information in R. Specifically we will see how to extract subsets of data from objects of type vector, array or dataframe. We will also see how to create new data sets from the aggregation of several objects. There are many commands that allow us to manipulate our data in R. Many things can be understood as *manipulation* but for the moment we will focus on:

* Select or extract information
* Sort tables
* Add rows or columns to a table

As you might already guess, we will work with tabular data like arrays and data.frames which I further refer to as **tables**.

### 2.4.1 Working with columns

The first thing we will do is access the information stored in the column(s) of a given table object. There are two basic ways to do this:

1. Using the position index of the column.
2. Using the name (header) of the column.

These two basic forms are not always interchangeable, so we will use one or the other depending on the case. It is recommended that you use the one that feels most comfortable for you. However, in most examples the column position index is used since it is a numerical value that is very easily integrated with loops and other iterative processes.

#### 2.4.1.1 Extracting columns

To extract columns using the position index we will use a series of instructions similar to those already seen in extracting information from [Arrays](https://remot-technologies.com/introduction-to-r/module-1/intro.html#arrays), [Vectors](https://remot-technologies.com/introduction-to-r/module-1/intro.html#vectors) and [Lists](https://remot-technologies.com/introduction-to-r/module-1/intro.html#lists). The following statement returns the information of the second column of the array object table and stores it in a new object that we called col2:

col2 <- table[,2]col2

## [1] 82500 110500 55500 28500 85500 38500 39500 124500 41500 49500## [11] 329500 77500 36500 148500

It is also possible to extract a range of columns, proceeding in a similar way to what has already been seen in the creation of vectors. The following statement extracts columns 2 and 3 from the table object and stores them in a new object called cols:

cols <- table[,2:3]cols

## X\_INDEX Y\_INDEX## 1 82500 4653500## 2 110500 4661500## 3 55500 4805500## 4 28500 4783500## 5 85500 4729500## 6 38500 4757500## 7 39500 4775500## 8 124500 4687500## 9 41500 4727500## 10 49500 4789500## 11 329500 4807500## 12 77500 4684500## 13 36500 4718500## 14 148500 4651500

Now let’s see how to select columns using their name. Name extraction is performed using a combination of object and column name object using $ to separte object from column name. The following statement selects the column named Y\_INDEX from the array object table and stores it in col.Y\_INDEX:

col.Y\_INDEX <- table$Y\_INDEXcol.Y\_INDEX

## [1] 4653500 4661500 4805500 4783500 4729500 4757500 4775500 4687500## [9] 4727500 4789500 4807500 4684500 4718500 4651500

A key piece of information here is the name of the column which we need to know in advance. Well, we can check the original text file or inspect the object table using str(). We can also take look to the top-right window activaing the *Environment* sub-window and unwrap table but be aware this can be only accessed using RStudio.

The main difference between these two methods is that index selection makes it possible to extract column ranges easily. To do this using the name of the columns you have to use functions like subset():

cols2 <- subset(table, select = c(X\_INDEX,Y\_INDEX))

Using the argument select we can point the columns that we want to extract using a vector with column names. Using subset() it is also possible to specify the columns that we **do NOT want to extract**. To do this proceed as follows:

cols2 <- subset(table, select = -c(X\_INDEX,Y\_INDEX))

In this way we would only extract the first column, excluding X\_INDEX and Y\_INDEX. Of cours, we can do this using the column index as well:

cols2 <- table[,-(2:3)]

#### 2.4.1.2 Merge columns and tables

The main reason why we are learning how to manipulate table columns is to be able to prepare our data for other purposes. It may be the case we need to join tables or columns that proceed from the same original table. The instruction cbind() allow us to merge together several tables and/or vectors **provided they have the same number of rows**. We can merge as many objects as we want to, by separating them using ,:

cols3 <- cbind(col2,cols2)cols3

## col2 cols2## [1,] 82500 364011## [2,] 110500 371655## [3,] 55500 487720## [4,] 28500 474504## [5,] 85500 436415## [6,] 38500 457549## [7,] 39500 469377## [8,] 124500 397162## [9,] 41500 434666## [10,] 49500 478383## [11,] 329500 488973## [12,] 77500 394153## [13,] 36500 426962## [14,] 148500 362216

#### 2.4.1.3 Changing column names

It is often the case we need to alter or change the name of a table object. If we wanted to rename all the columns of an object we would to pass a vector with names to the function colnames() in case we are renaming an array or names() if we are dealing with a data.frame. Note that the vector should have the same length as the total number of columns. Lets rename our table cols3:

colnames(cols3)<- c("COL1","COL2")

What if we want to change only a given name. Then we just point to the column header using the position index like this:

colnames(cols3)[2]<- "RENAMED COLUMN"

You may be wondering *How can I know what kind of object is my table?*. That is a very good question. Specially at the begining is quite difficult to be in control this stuff. If you use RStudio you already see a description of the objects in the top-right window. array or matrix objects show something like [1:14,1:2] indicating multiple dimensions, vectors are similar but with only 1 dimension [1:14] and data.frames show the word data.frame in their description. However, this is not the fancy way to deal with object types. Just for the record, when I mean **type** an actual code developer means **class**. Of course there is a function called class() that returns the class an object belongs to:

class(cols3)

## [1] "matrix"

#### 2.4.1.4 Sorting our data

Finally, let’s look at how to sort columns and arrays. To sort a column in R, the sort() function is used. The general function of the function is:

sort(cols3[,1])

## [1] 28500 36500 38500 39500 41500 49500 55500 77500 82500 85500## [11] 110500 124500 148500 329500

If we want to reorder an array based on the values of one of its columns, we will use the order () function. The general operation of the function is:

table[order(table$X\_INDEX),]

## FID\_1 X\_INDEX Y\_INDEX## 4 474504 28500 4783500## 13 426962 36500 4718500## 6 457549 38500 4757500## 7 469377 39500 4775500## 9 434666 41500 4727500## 10 478383 49500 4789500## 3 487720 55500 4805500## 12 394153 77500 4684500## 1 364011 82500 4653500## 5 436415 85500 4729500## 2 371655 110500 4661500## 8 397162 124500 4687500## 14 362216 148500 4651500## 11 488973 329500 4807500

To be honest, in this last example we are actually working with rows. Take a look at the position of the ,. The brackets are also something that we will use later to extract data from a table. But it feels right to bring here the order() command right after sort().

### 2.4.2 Working with rows

Let us now turn to the manipulation of rows. The procedure is basically the same as in the case of columns, except for the fact that we normally do not work with names assigned to rows (although that’s a possibility), but we refer to a row using its position. To extract rows or combine several objects according to their rows we use the following expressions:

row1 <- table[1:5,]row2 <- table[-(6:7),]row3 <- rbind(row1,row2)

Same as with [Arrays](https://remot-technologies.com/introduction-to-r/module-1/intro.html#arrays) we point to rows instead of columns when we use the index value to the left of the [row,col]. So that’s the thing, we just change that and we are dealing with rows. We can join rows and tables using the rbind() function rather than cbind(). r stands for row and c for column.

We can extract a subsample of rows that meet a given criteria:

table[criteria,]

table[table$X\_INDEX==82500,]

## FID\_1 X\_INDEX Y\_INDEX## 1 364011 82500 4653500

table[table$X\_INDEX>82500,]

## FID\_1 X\_INDEX Y\_INDEX## 2 371655 110500 4661500## 5 436415 85500 4729500## 8 397162 124500 4687500## 11 488973 329500 4807500## 14 362216 148500 4651500

Oh, I expect you to have found out this by yourself but evidently we can combine row and column manipulation if that fits our purpose.

# Chapter 3 Basic instructions

## 3.1 Descriptive statistics and summaries

There are a large number of functions in the basic installation of R. It would be practically impossible to see all of them so we will see some of the most used, although we must remember that not only is there the possibility of using predesigned functions, but R also offers the possibility to create your own functions.

Below are some of the basic statistical functions that we can find in R. These functions are generally applied matrix-array or data.frame data objects. Some of them can be applied to the whole table and others to single columns or rows.

* sum() Add values.
* max() Returns the maximum value.
* min() Returns the minimum value.
* mean() Calculates the mean.
* median() Returns the median.
* sd() Calculates the standard deviation.
* summary() Returns a statistical summary of the columns.

We are going to applie them using some example data. We will use the data stored in the file *fires.csv*, inside the *Data* folder (Table [3.1](https://remot-technologies.com/introduction-to-r/module-1/basic-instructions.html#fig:table2)). This file contains data on the annual number of fires between 1985 and 2009 in several European countries.

Show

entries

Search:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | **YEAR** | **PORTUGAL** | **SPAIN** | **FRANCE** | **ITALY** | **GREECE** | **EUMED** |
| 1 | 1985 | 8441 | 12238 | 6249 | 18664 | 1442 | 47034 |
| 2 | 1986 | 5036 | 7570 | 4353 | 9398 | 1082 | 27439 |
| 3 | 1987 | 7705 | 8679 | 3043 | 11972 | 1266 | 32665 |
| 4 | 1988 | 6131 | 9247 | 2837 | 13588 | 1898 | 33701 |
| 5 | 1989 | 21896 | 20811 | 6763 | 9669 | 1284 | 60423 |
| 6 | 1990 | 10745 | 12913 | 5881 | 14477 | 1322 | 45338 |
| 7 | 1991 | 14327 | 13531 | 3888 | 11965 | 858 | 44569 |
| 8 | 1992 | 14954 | 15955 | 4002 | 14641 | 2582 | 52134 |
| 9 | 1993 | 16101 | 14254 | 4769 | 14412 | 2406 | 51942 |
| 10 | 1994 | 19983 | 19263 | 4618 | 11588 | 1763 | 57215 |

Showing 1 to 10 of 25 entries

Previous123Next

Figure 3.1: Structure of the fires.csv file.

The first thing is to import the file into a table.

fires<-read.table('Data/fires.csv', header= TRUE, sep=',')

Once data is imported we can take a look at the structure to make sure that everything went well. We should have integer values for each region:

str (fires)

## 'data.frame': 25 obs. of 7 variables:## $ YEAR : int 1985 1986 1987 1988 1989 1990 1991 1992 1993 1994 ...## $ PORTUGAL: int 8441 5036 7705 6131 21896 10745 14327 14954 16101 19983 ...## $ SPAIN : int 12238 7570 8679 9247 20811 12913 13531 15955 14254 19263 ...## $ FRANCE : int 6249 4353 3043 2837 6763 5881 3888 4002 4769 4618 ...## $ ITALY : int 18664 9398 11972 13588 9669 14477 11965 14641 14412 11588 ...## $ GREECE : int 1442 1082 1266 1898 1284 1322 858 2582 2406 1763 ...## $ EUMED : int 47034 27439 32665 33701 60423 45338 44569 52134 51942 57215 ...

Now we will calculate some descriptive statistics for each column to have a first approximation to the *distribution* of our data. To do this we will use the function summary() that will return some basic statistical values such as:

* Quantile
* Mean
* Median
* Maximum
* Minimum

summary(fires)

## YEAR PORTUGAL SPAIN FRANCE ## Min. :1985 Min. : 5036 Min. : 7570 Min. :2781 ## 1st Qu.:1991 1st Qu.:14327 1st Qu.:12913 1st Qu.:4002 ## Median :1997 Median :21870 Median :16771 Median :4618 ## Mean :1997 Mean :20848 Mean :16937 Mean :4907 ## 3rd Qu.:2003 3rd Qu.:26488 3rd Qu.:20811 3rd Qu.:6249 ## Max. :2009 Max. :35697 Max. :25827 Max. :8005 ## ITALY GREECE EUMED ## Min. : 4601 Min. : 858 Min. :27439 ## 1st Qu.: 7134 1st Qu.:1322 1st Qu.:45623 ## Median : 9540 Median :1486 Median :55215 ## Mean : 9901 Mean :1656 Mean :54249 ## 3rd Qu.:11965 3rd Qu.:1898 3rd Qu.:62399 ## Max. :18664 Max. :2582 Max. :75382

The summary() function is not only used to obtain summaries of data through descriptive statistics, but can also be used in model type objects to obtain a statistical summary of the results, coefficients, significance … Later we will see an example of this applied on a linear regression model.

Let’s now see what happens if we apply some of the functions presented above. Try running the following instructions:

sum(fires)

## [1] 2762377

max(fires)

## [1] 75382

min(fires)

## [1] 858

As can be seen these 3 instructions work with a tablebut mean(), median() and sd() will not. We have to apply them to a single column:

mean(fires$SPAIN)

## [1] 16937.24

median(fires$SPAIN)

## [1] 16771

sd(fires$SPAIN)

## [1] 5260.519

## 3.2 The *apply()* function

So far we have seen how to apply some of the basic statistical functions to our data, applying those functions to the data contained in the matrix or some of its columns. In the case of columns we have manually specified which one to apply a function. However, we can apply functions to all elements of an array (columns or rows) using iteration functions.

The apply() function allows you to apply a function to all elements of a table. We can apply some of the functions before the rows or columns of an array. There are different variants of this function. First, as always, invoke the help of the function to make sure what we are doing.

help(apply)

According to the specified in the function help we can see that the apply () function works as follows:

apply (X, MARGIN, FUN, …)

Where:

* X: data matrix.
* MARGIN: argument to specify whether the function is applied to rows (1) or columns (2).
* FUN: function to be applied (mean,sum…)

For example, if we want to sum the values of each country/region we will do as follows:

apply(fires,2,mean)

## YEAR PORTUGAL SPAIN FRANCE ITALY GREECE EUMED ## 1997.00 20848.20 16937.24 4907.16 9900.64 1655.80 54249.04

Well, almost there. We have included the *YEAR* column which is the first one:

apply(fires[,-1],2,mean)

## PORTUGAL SPAIN FRANCE ITALY GREECE EUMED ## 20848.20 16937.24 4907.16 9900.64 1655.80 54249.04

**EXERCISE 1**: Calculate the total number of fires on a yearly basis.

**EXERCISE 2**: - Open the file “barea.csv” and save it in an array named “barea” - Select and store in a new matrix the data from the year 2000 to 2009 - Calculate descriptive statistics 2000-2009 - Calculate the mean, standard deviation, minimum and maximum of all columns and save it to a new object. - Save the object into a text file.

## 3.3 Installing packages

We have already seen how some of the basic functions of R. work. However, we have the possibility of extending the functionalities of R by importing new packages into our environment. These packages are developed by different research groups and/or individuals.

There are currently over 4600 packages available in the R project repository ([CRAN](https://cran.r-project.org/web/packages/available_packages_by_name.html)). Obviously we do not need to know how each one of them works, but only focus on those that fit our needs.

The import and installation of new packages is carried out in 2 stages:

1. obtaining and installing and internal call to the package.
2. Loading the package into our session.

### 3.3.1 Installing an external package

We will install the “foreign” package. This package provides functions for reading and writing data stored in different statistical software formats such as Minitab, S, SAS, SPSS, Stata, Systat … and to read and write *dBase* files such as attribute tables of vector layers in format *shapefile*. The first thing we will do is get the package via download. This can be done from the menu of R:

* Packages / Install packages …
* Select download directory (mirror)
* Search for “foreing”

If you are using RStudio simply go to the *Packages* tab in the lower right box and select the *Install* option. In the popup window select the *Repository (CRAN, CRANextra)* option in the drop-down menu and type the name of the package.

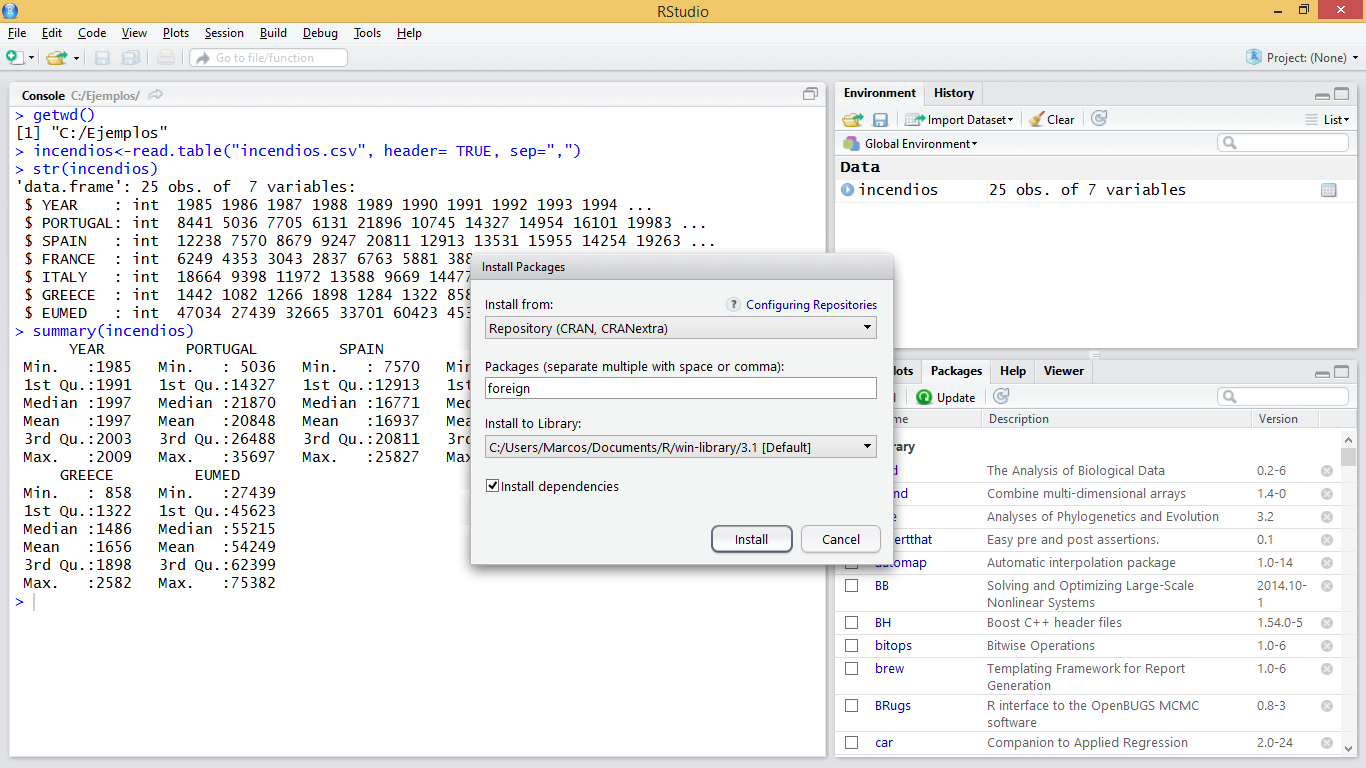


Figure 3.2: Installing a package using RStudio

Another option is to obtain the package in *.zip* format directly from the webpage of project R and use the function *Install package (s) from local zip files …* (or *Package archives* in RStudio).

It is also possible to install packages through instructions in the R window, which is the most recommended method[7](https://remot-technologies.com/introduction-to-r/module-1/basic-instructions.html#fn7):

#install.packages('foreign')

At this point we would have installed the package in our personal library of R. However, in order to use the functions of the new package in our R environment it is necessary to make an internal call to the package. This is generally done using the library() function[8](https://remot-technologies.com/introduction-to-r/module-1/basic-instructions.html#fn8):

library(foreign)

Once this is done we have all the functions of the package ready to be used. All that is left is to learn how to use the functions of the package… which is easy to say but maybe not to do.

## 3.4 Create a function

So far we have mainly seen how to use pre-designed functions either in the default installation or from other packages. However, there is the possibility of creating our own functions.

A function is a group of instructions that takes an input or input data, and uses them to calculate other values, returning a result or product. For example, the mean() function takes as input a vector and returns as a result a numeric value that corresponds to the arithmetic mean.

To create our own functions we will use the object called function that constitute new functions. The usual syntax is:

FunctionName <- function (arguments){commands}

Where:

* arguments are the arguments we want to pass to our function.
* commands are the instructions needed to do whatever the function does.

Let’s look at a simple example. We will create a function to calculate the [standard deviation](https://en.wikipedia.org/wiki/Standard_deviation) of a vector with numerical data. The standard deviation formula looks like this:

μ=√∑Ni=1(xi−¯¯¯x)2N−1

Which is essentally the square root (sqrt()) of the variance (var()):

desv <- function(x){sqrt(var(x))}

The function name is desv(). This function requires a single argument (x) to be executed. Once the function is defined, it can be called and used as any other predefined function in the system.

data <- 1:10desv(data)

## [1] 3.02765

Let’s see an example with 2 arguments. We will create a function to calculate the NDVI. The function will take as arguments two objects of vector type corresponding to the sensor Landsat TM channels 3 and 4:

The normalized difference vegetation index (NDVI) is a simple graphical indicator that can be used to analyze remote sensing measurements, typically but not necessarily from a space platform, and assess whether the target being observed contains live green vegetation or not [wikipedia.org](https://en.wikipedia.org/wiki/Normalized_Difference_Vegetation_Index).

NDVI=NIR−RedNIR+Red

Where NIR

(*x*) is the near-infrarred band and Red

(*y*) is the red band.

ndvi <- function(x,y){(y-x)/(y+x)}

As we have said, once created we can use our functions in the same way as the rest of functions. This includes using the apply() function and its derived versions to iterate over rows and columns of an array or dataframe.

Let’s see an example with the function desv(), previously created applied to the data of number of fires:

apply(fires,2,desv)

## YEAR PORTUGAL SPAIN FRANCE ITALY ## 7.359801 9187.114305 5260.519400 1393.670265 3467.510729 ## GREECE EUMED ## 496.662025 13547.393887

# Chapter 4 Plotting with R

## 4.1 Basic plots

R is not just an environment for the implementation and use of functions for statistical calculation but it is also a powerful environment for generating and displaying plots. Creating plots is besides an effective and quick way to visualize our data. By doing so we can verify whether data was correctly imported or not. However, creating graphics is also done by command-line instructions, which can sometimes be a bit tricky, especially at the beginning.

In R we can create many types of plots. With some packages it is also possible to generate maps similar to those created by GIS, although for the moment we will only see some basic types such as:

* Dot charts
* Line charts
* Barplots
* Histograms
* Scatterplots

Before going into detail with the specific types of graphics we will see some general concepts which apply to the majority of plots:

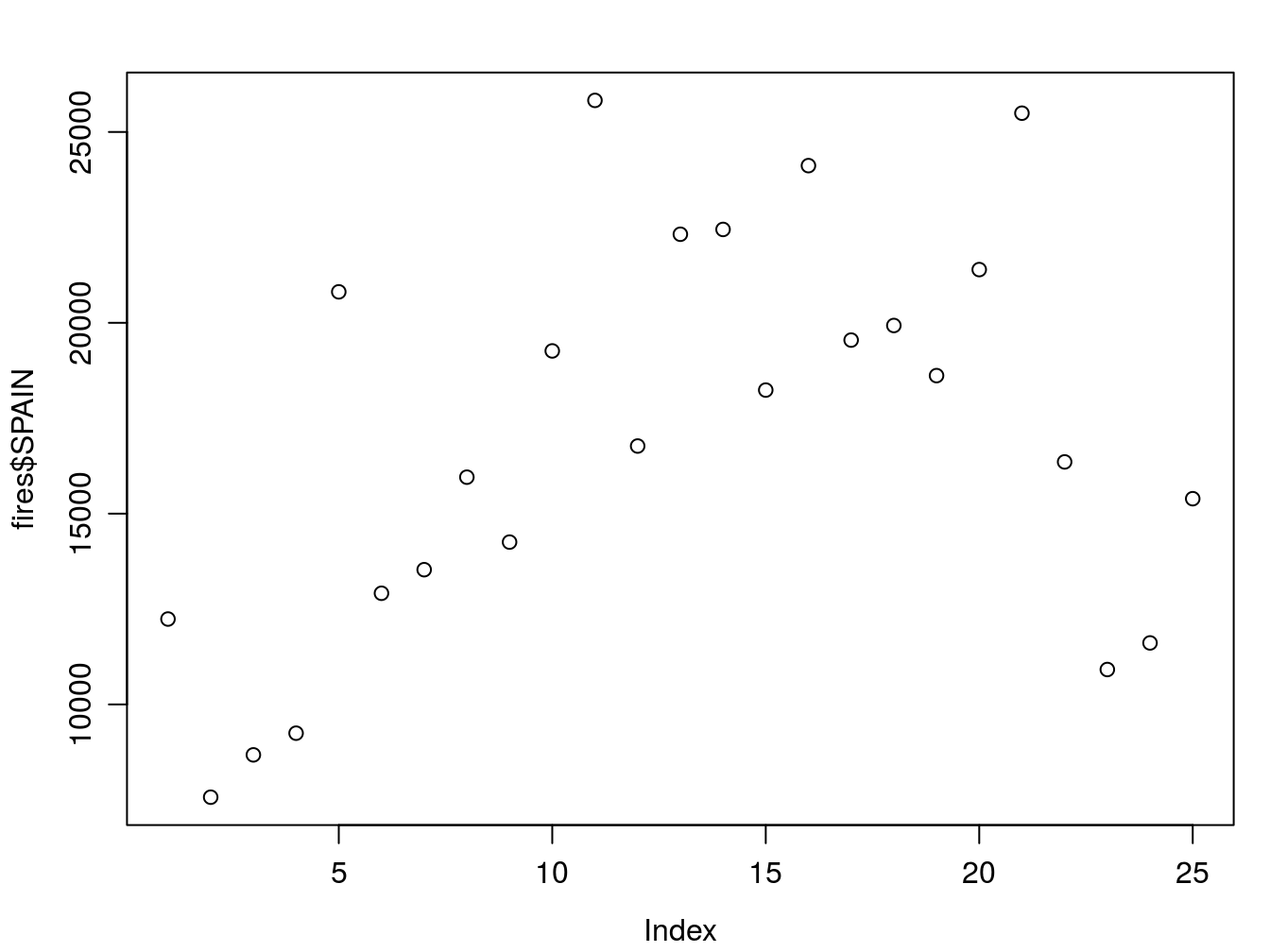
1. All graphics always require an object that contains the data to be drawn. This object is usually specified in the first argument of the function corresponding to each type of chart.
2. There are a number of arguments to manipulate axis labels or the chart title:

* main: text with the title of our plot.
* xlab: text for x axis label.
* ylab: text for y axis label.
* xlim: vector with upper and low range for the x-axis.
* ylim: vector with upper and low range for the y-axis.
* cex: number indicating the aspect ratio between plot elements and text. 1 by default.
* col: changes de color the plotted element. See <http://www.stat.columbia.edu/~tzheng/files/Rcolor.pdf>
* legend(): adds a legend element describing symbology.

### 4.1.1 Dot or point plots

This is one of the most basic types of graphics if not the most basic one we can create. To create such a graph we will use the plot() function. Let’s look at a simple example using data from *fires.csv*[9](https://remot-technologies.com/introduction-to-r/module-1/plotting-with-r.html#fn9):

par(mar=c(3.5, 3.5, 2, 1), mgp=c(2.4, 0.8, 0))plot(fires$SPAIN)



As you can see, we are ploting data from the *SPAIN* column, ie, yearly fire occurrence data in Spain.

Remember RStudio displays plots in the right-bottom window. In addition, if you need to take closer look use the *Zoom* button to pop-up a new plot window.

Le’s tune up and enhace our plot. We can change the symbol type using the pch argument. You’ll find a list of symbol types (not just dot charts) at <http://www.statmethods.net/advgraphs/parameters.html>.

par(mar=c(3.5, 3.5, 2, 1), mgp=c(2.4, 0.8, 0))plot(fires$SPAIN,pch=4)



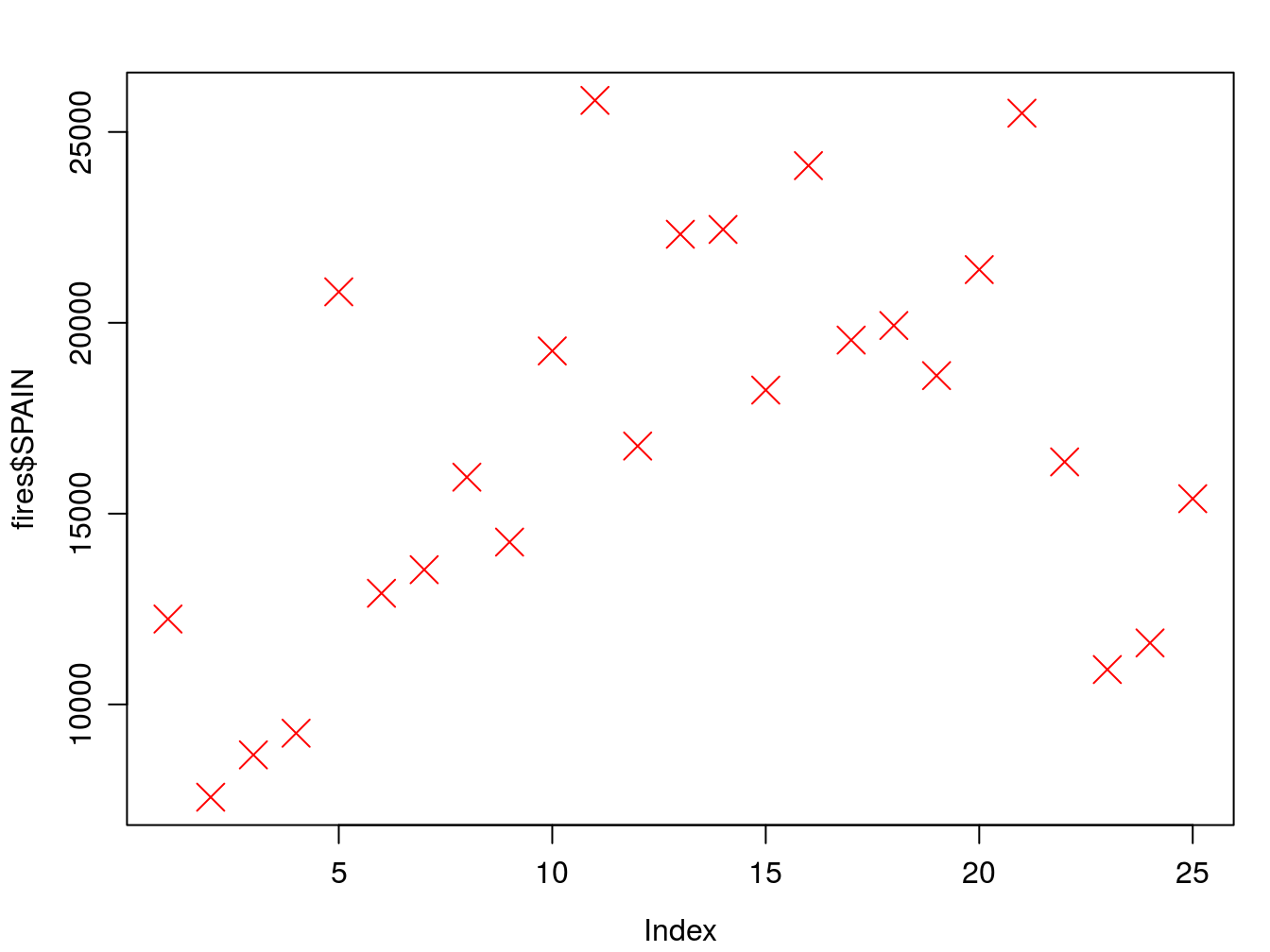
We can change symbol size using cex:

par(mar=c(3.5, 3.5, 2, 1), mgp=c(2.4, 0.8, 0))plot(fires$SPAIN,pch=4,cex=2)



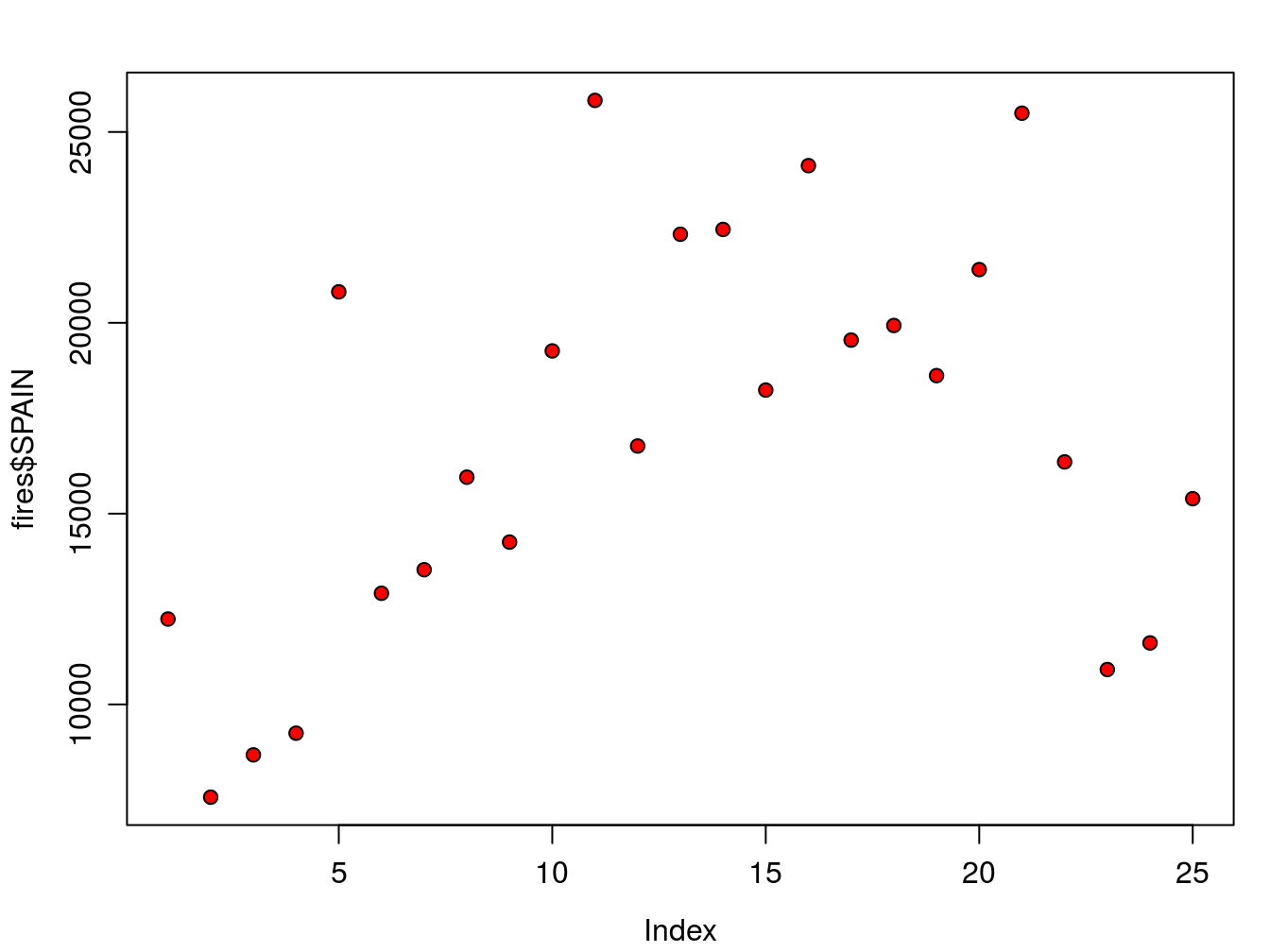
And we can change the color of the symbol with `col’:

par(mar=c(3.5, 3.5, 2, 1), mgp=c(2.4, 0.8, 0))plot(fires$SPAIN,pch=4,cex=2,col='red')



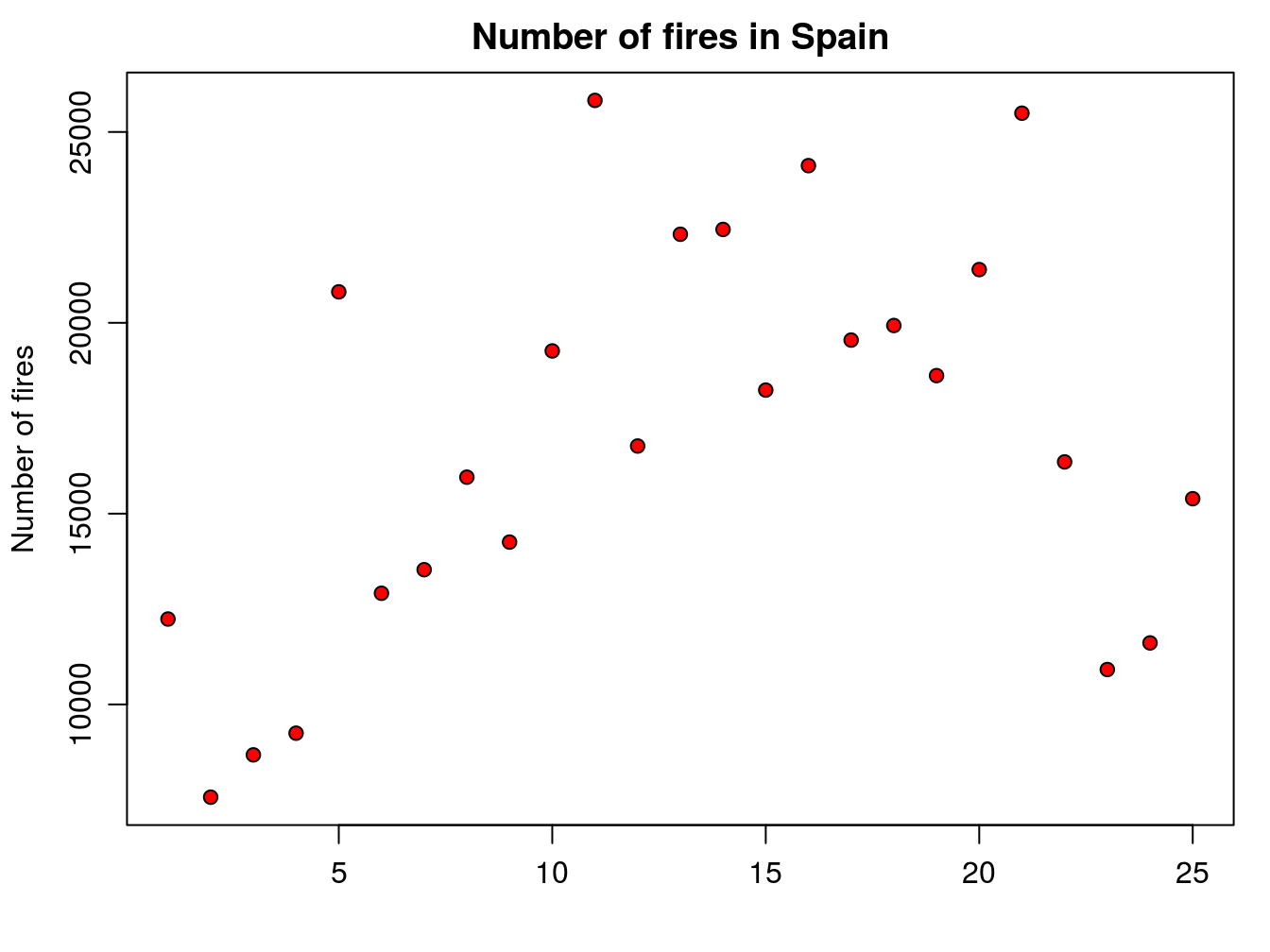
The col argument can be specified either using the color name as in the example or using its code number, hexadecimal or RGB so that col = 1, col = "white", and col = "#FFFFFF" are equivalent. In some types of symbols we can also change the color of the symbol background in addition to the symbol itself using the argument bg:

par(mar=c(3.5, 3.5, 2, 1), mgp=c(2.4, 0.8, 0))plot(fires$SPAIN,pch=21,cex=1,col='black',bg='red')

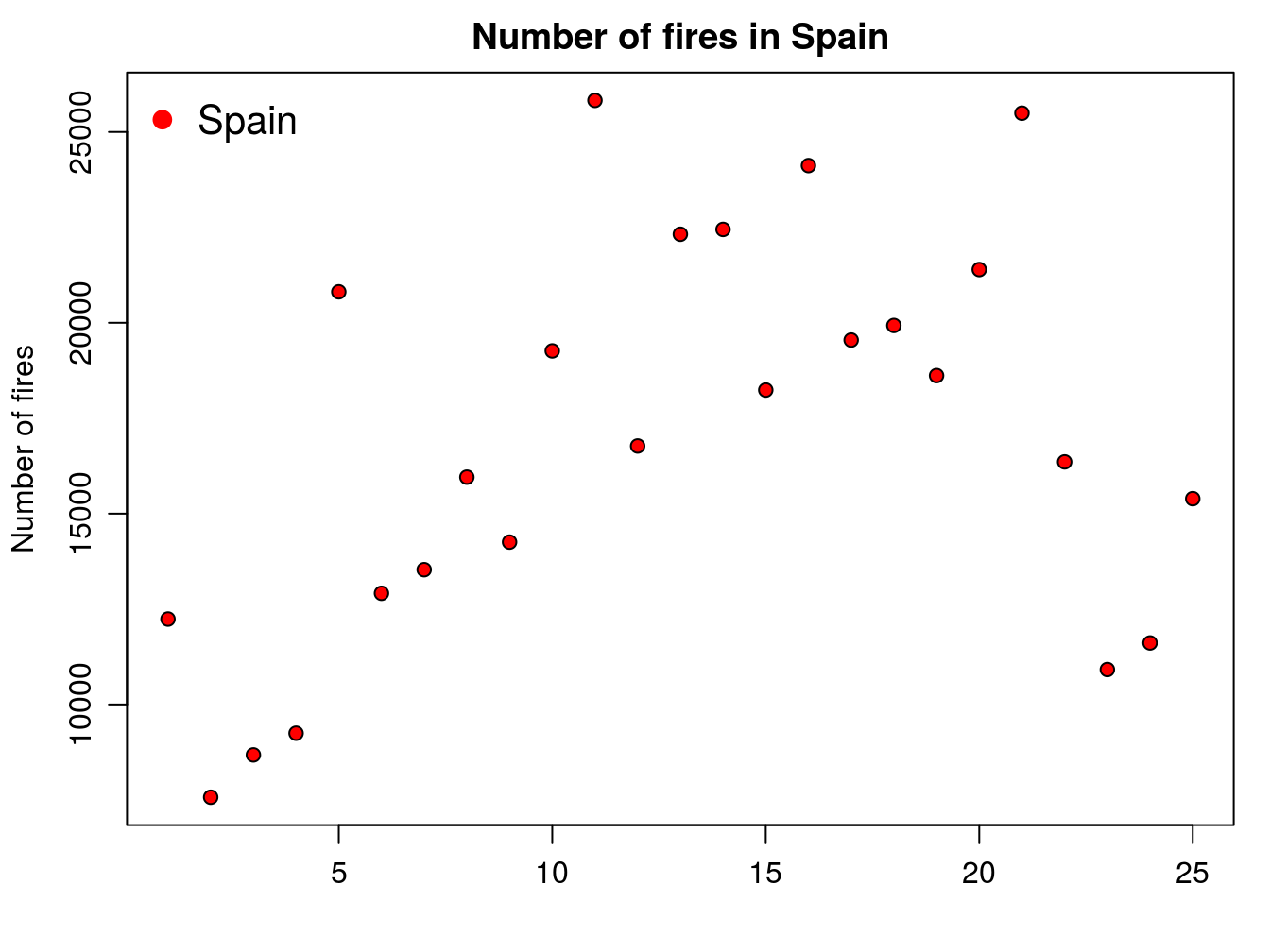


The finish our plot we will modify axis labels, add a title and a legend:

par(mar=c(3.5, 3.5, 2, 1), mgp=c(2.4, 0.8, 0))plot(fires$SPAIN,pch=21,cex=1,col='black',bg='red', main='Number of fires in Spain',ylab = 'Number of fires',xlab = '')

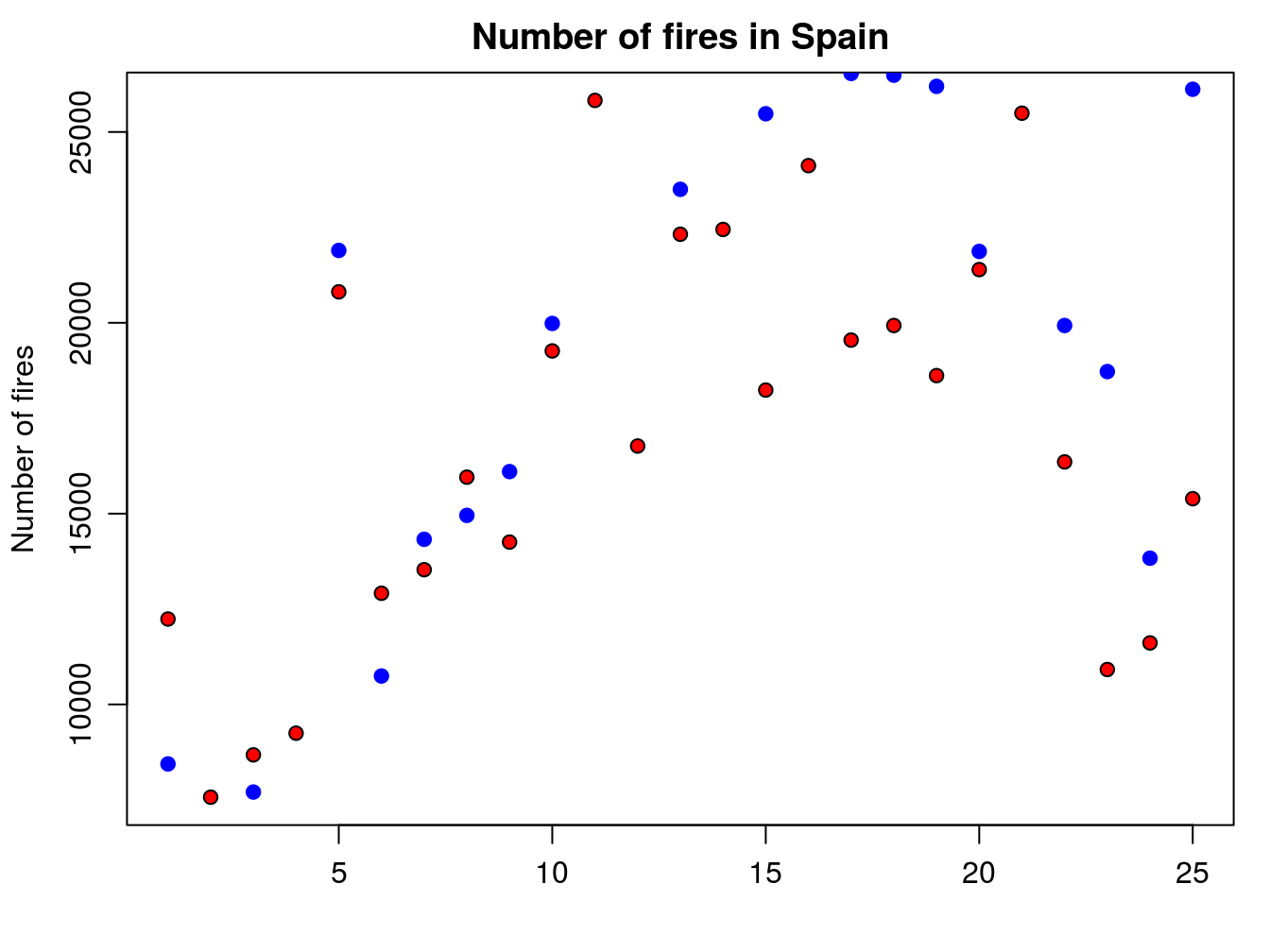
Note that we have used the xlab argument to leave the x-axis label *blank*. Now we add the legend. It is important you bear in mind that the legend is added with an additional command right after the plot statement. Legends in basic plots are just an image we add to an existing plot by emulating the symbol used in that plot using the legend() function:

par(mar=c(3.5, 3.5, 2, 1), mgp=c(2.4, 0.8, 0))plot(fires$SPAIN,pch=21,cex=1,col='black',bg='red', main='Number of fires in Spain',ylab = 'Number of fires',xlab = '')legend( "topleft" , cex = 1.3, bty = "n", legend = c("Spain"), , text.col = c("black"), col = c("red") , pt.bg = c("red") , pch = c(21) )



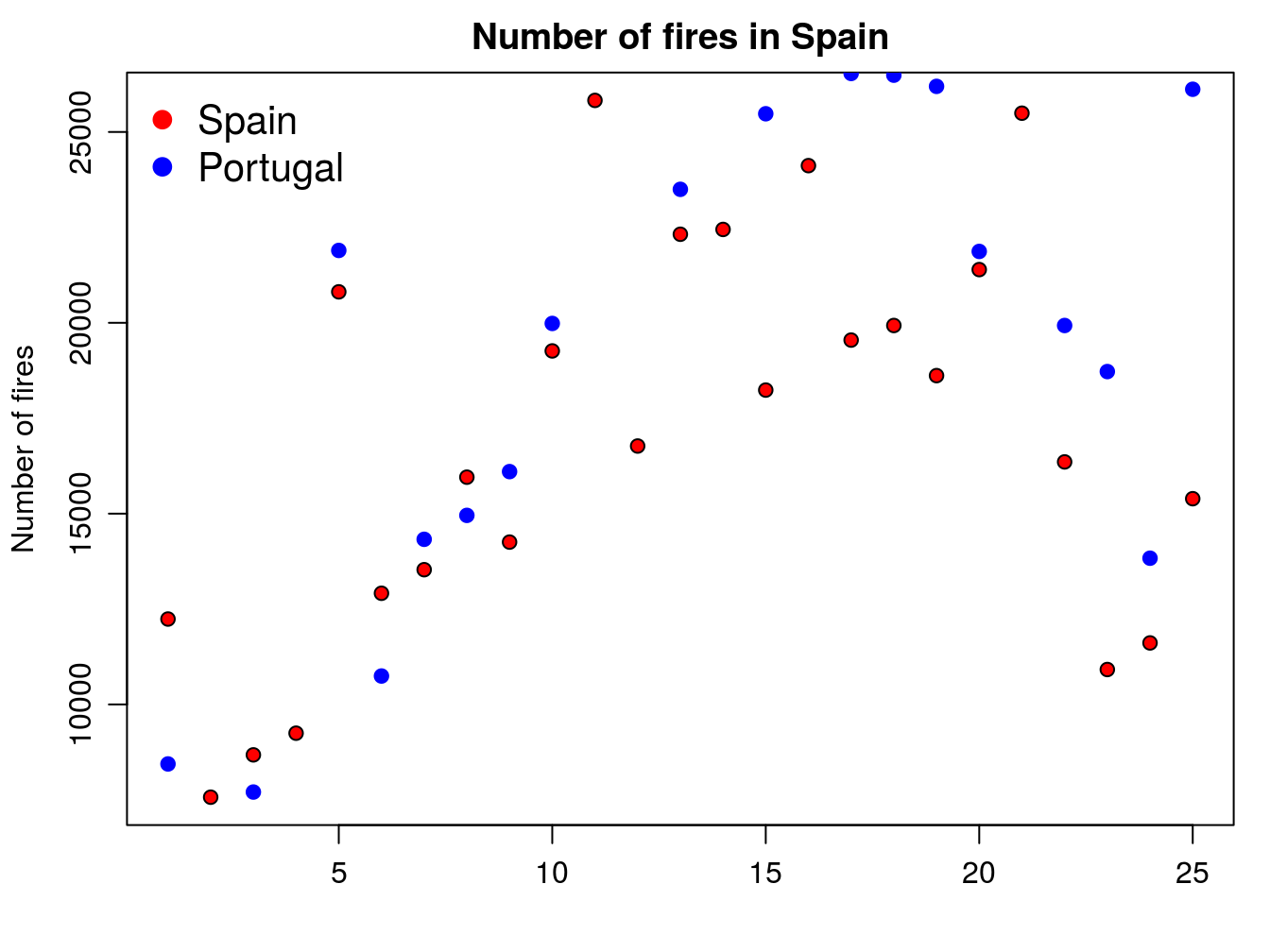
But, what if we want to add a second series of data? Then we proceed in a way similar to the legend() statement, but using the function points() similar to what we have seen in the plot() example. In the following code we add a second point data series with portugal:

par(mar=c(3.5, 3.5, 2, 1), mgp=c(2.4, 0.8, 0))plot(fires$SPAIN,pch=21,cex=1,col='black',bg='red', main='Number of fires in Spain',ylab = 'Number of fires',xlab = '')points(fires$PORTUGAL,pch=21,cex=1,col="blue",bg="blue")



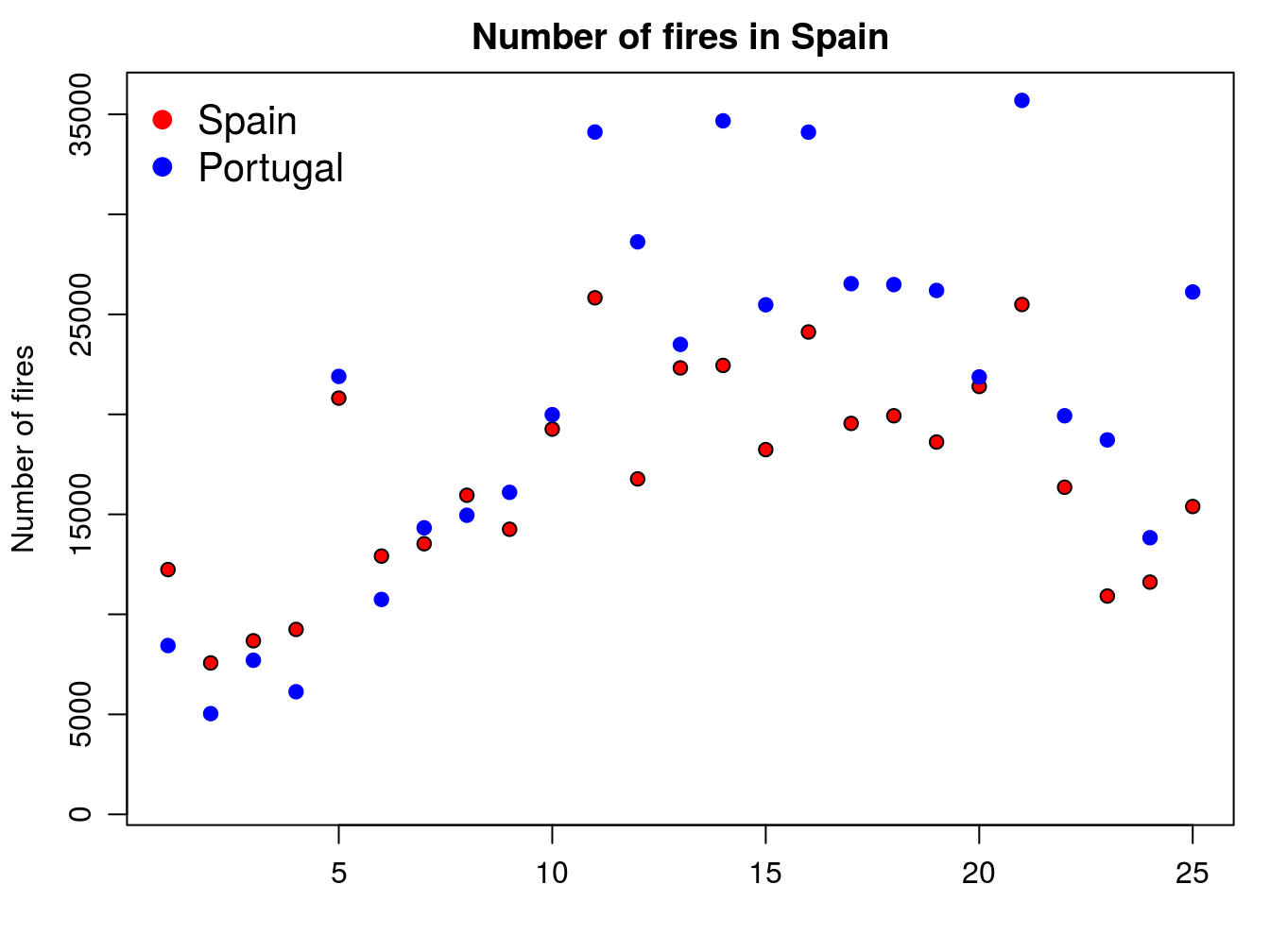
Finally, we update the legend to fit the new plot with the second series of data. To do that we simply include a second value on each argument using a vector:

par(mar=c(3.5, 3.5, 2, 1), mgp=c(2.4, 0.8, 0))plot(fires$SPAIN,pch=21,cex=1,col='black',bg='red', main='Number of fires in Spain',ylab = 'Number of fires',xlab = '')points(fires$PORTUGAL,pch=21,cex=1,col="blue",bg="blue")legend( "topleft" , cex = 1.3, bty = "n", legend = c("Spain","Portugal"), , text.col = c("black"), col = c("red","blue") , pt.bg = c("red","blue") , pch = c(21) )



The problem we are now experiencing is that data for Portugal doesn’t fit in the extent of the plot as it is. We should modify this using the xlim and ylim arguments. At this point x-axis works fine, so we’ll leave it as it is. The problem comes from y-axis. We can solve it by passing the ylim argument passing the minimum and maximum values of the fires data. We bring here some functions from [Descriptive statistics and summaries](https://remot-technologies.com/introduction-to-r/module-1/basic-instructions.html#descriptive-statistics-and-summaries):

par(mar=c(3.5, 3.5, 2, 1), mgp=c(2.4, 0.8, 0))plot(fires$SPAIN,pch=21,cex=1,col='black',bg='red', main='Number of fires in Spain',ylab = 'Number of fires',xlab = '', ylim = c(min(fires[,2:6]),max(fires[,2:6]))) points(fires$PORTUGAL,pch=21,cex=1,col="blue",bg="blue")legend( "topleft" , cex = 1.3, bty = "n", legend = c("Spain","Portugal"), , text.col = c("black"), col = c("red","blue") , pt.bg = c("red","blue") , pch = c(21) )



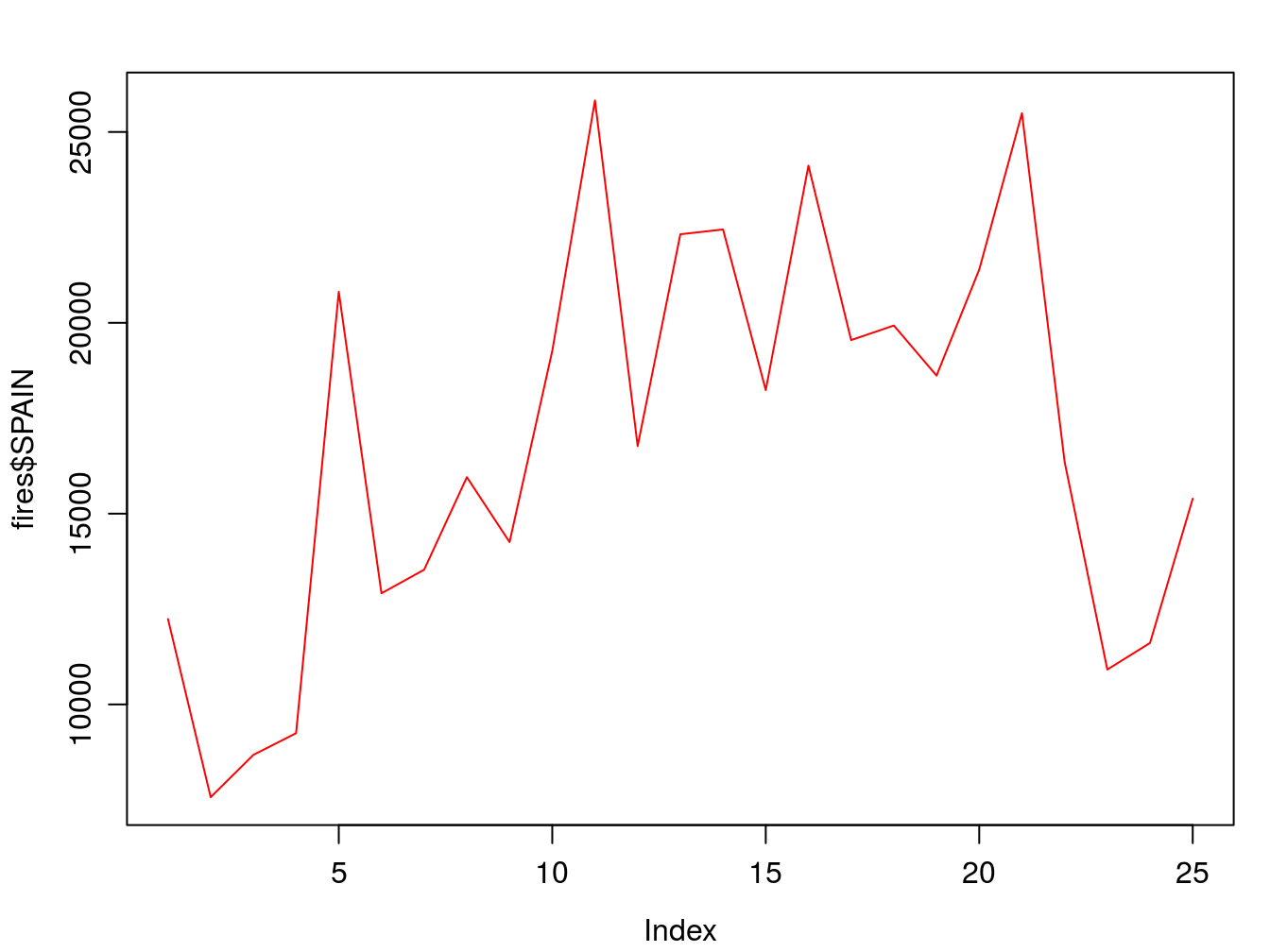
**EXERCISE 3**:

Explain in detail how the statement ylim = c(min(fires[,2:6]),max(fires[,2:6])) works in terms of the max() and min() functions and its interaction with the firesobject.

### 4.1.2 Line plots

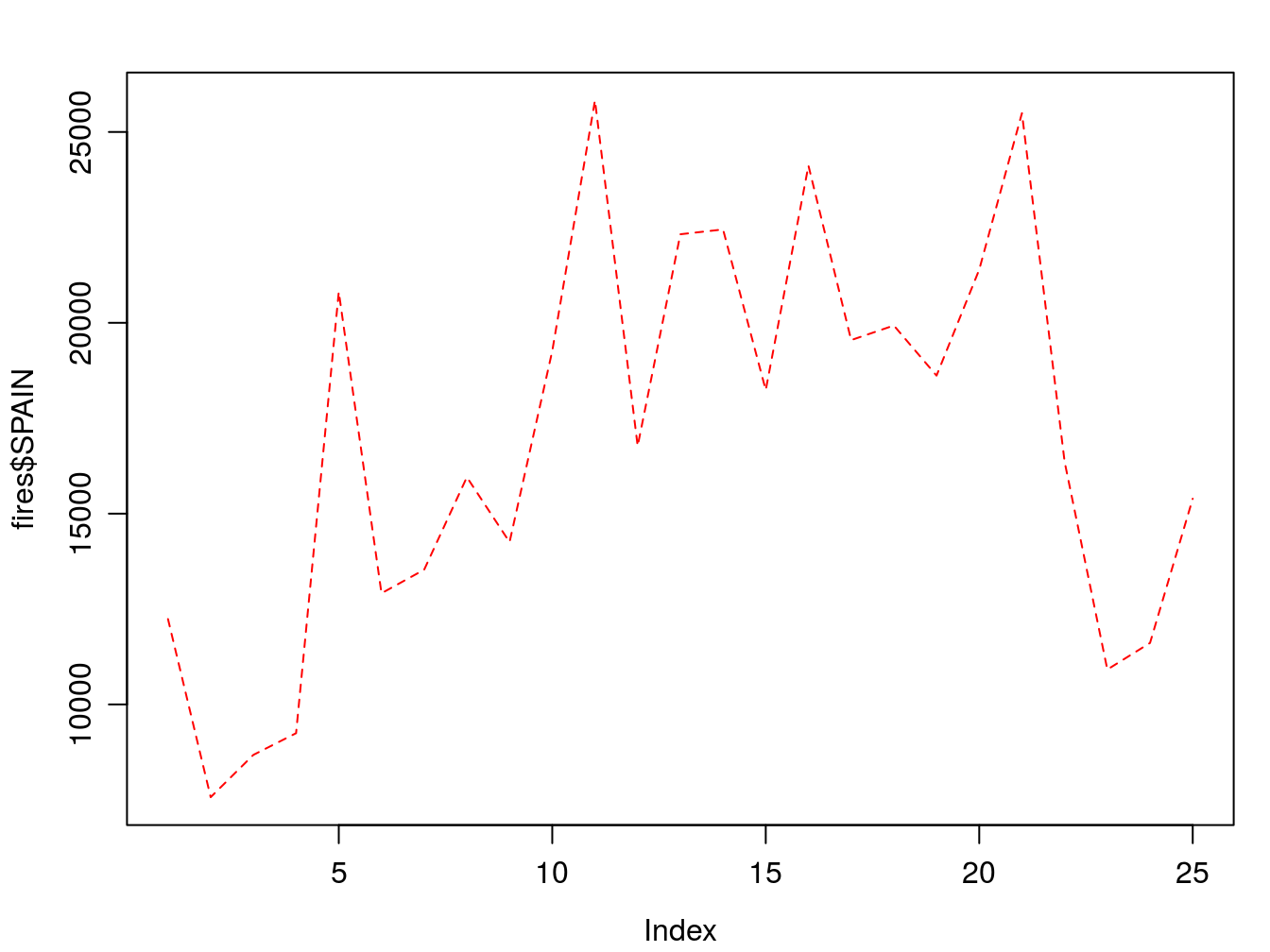
Well, this is quite easy. In line plots we use lines to represent our data series instead of points which are the default symbol. How do we do that? Just adding an additionla argument to specify we want to use lines with type = 'l':

par(mar=c(3.5, 3.5, 2, 1), mgp=c(2.4, 0.8, 0))plot(fires$SPAIN,pch=4,type='l',col='red')



Of course we can change the line [style](http://www.statmethods.net/advgraphs/parameters.html):

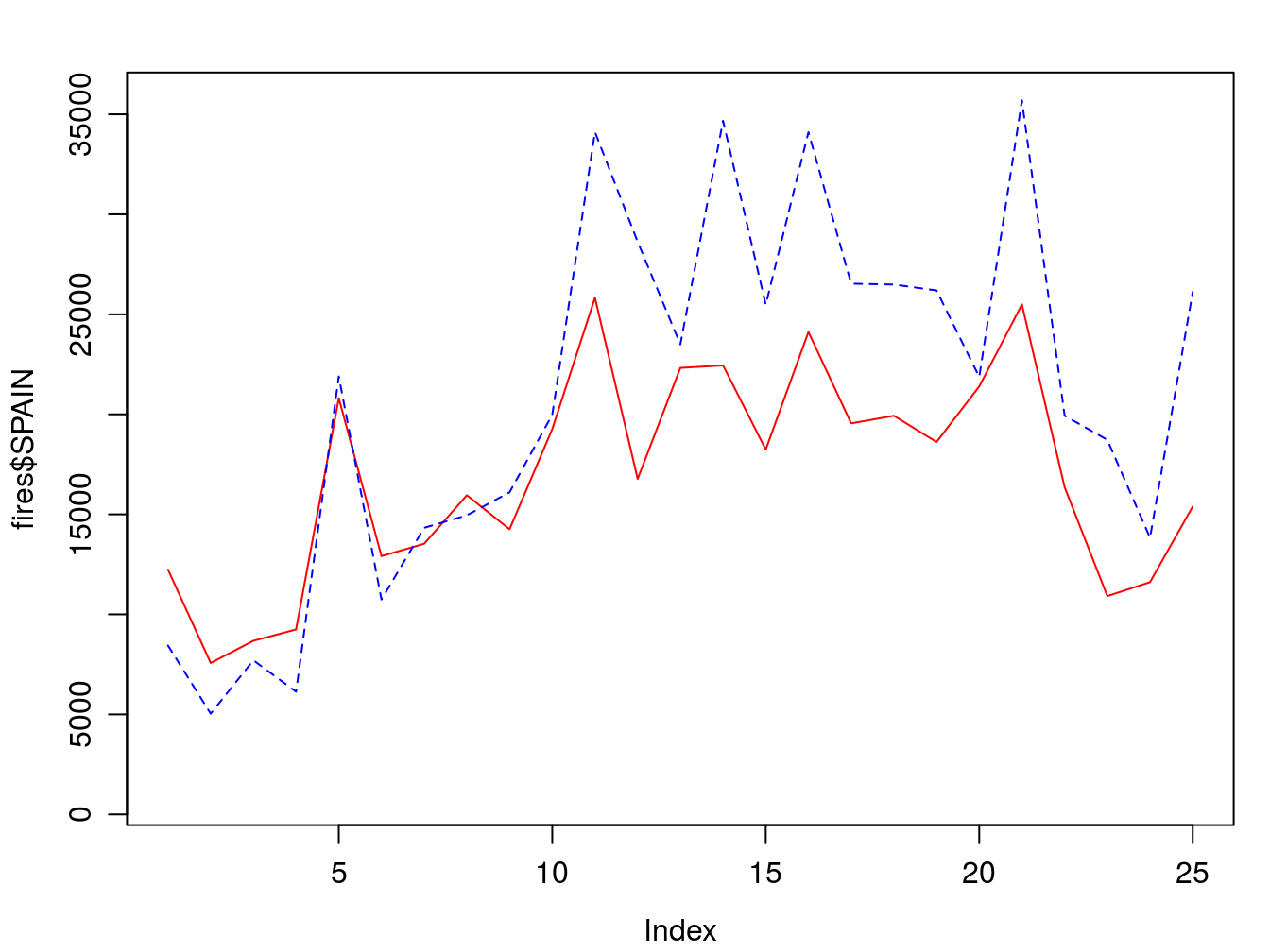
par(mar=c(3.5, 3.5, 2, 1), mgp=c(2.4, 0.8, 0))plot(fires$SPAIN,pch=4,type='l', lty=2, col='red')



Adding a second (or third, fourth,…,n

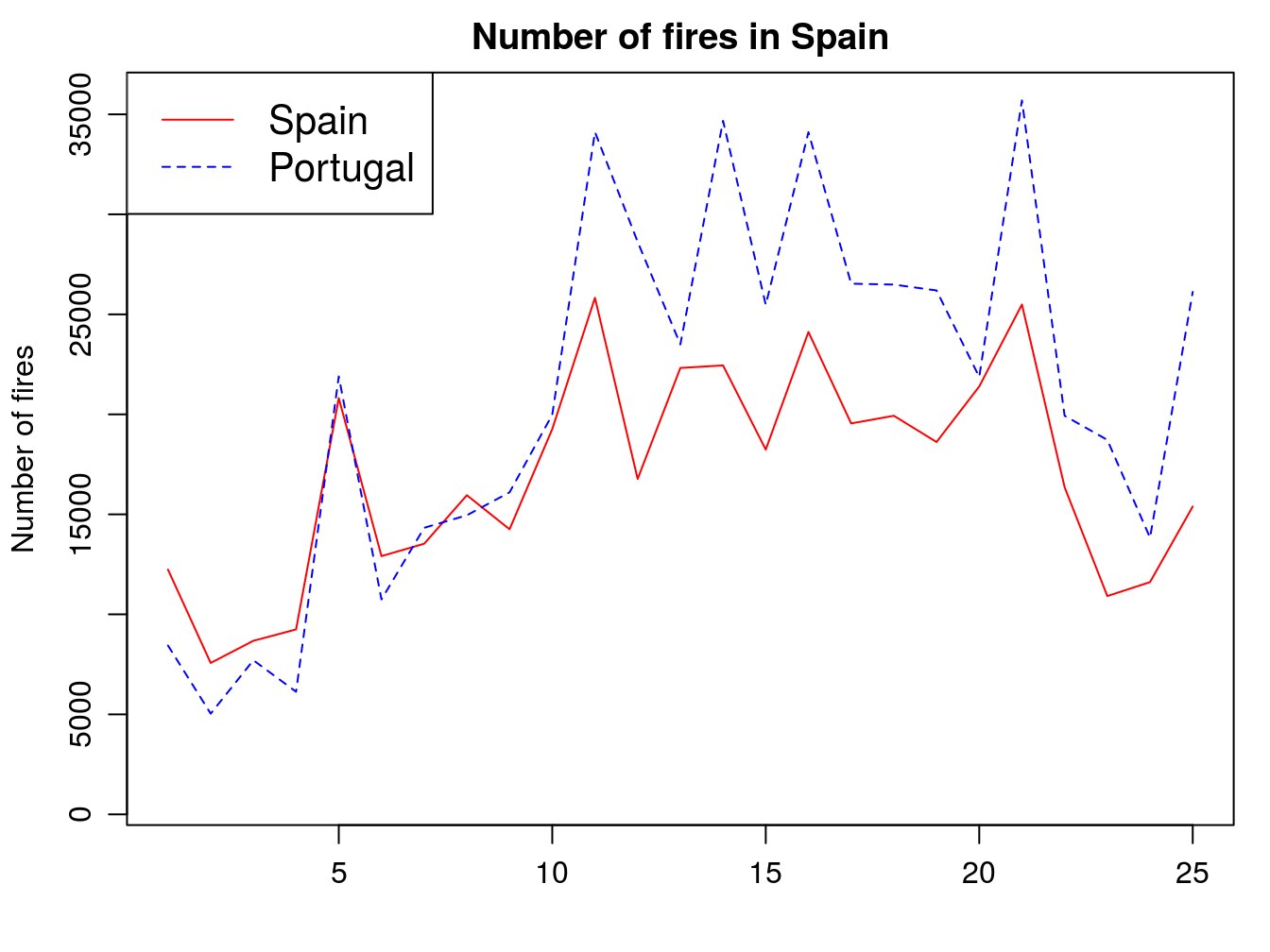
) series is done with lines():

par(mar=c(3.5, 3.5, 2, 1), mgp=c(2.4, 0.8, 0))plot(fires$SPAIN,pch=4,type='l', col='red', ylim = c(min(fires[,2:6]),max(fires[,2:6])))lines(fires$PORTUGAL,cex=1,col="blue",lty=2)



We now adapt the legend and add some titles and we are good to go:

par(mar=c(3.5, 3.5, 2, 1), mgp=c(2.4, 0.8, 0))plot(fires$SPAIN,pch=4,type='l', col='red', main='Number of fires in Spain',ylab = 'Number of fires',xlab = '', ylim = c(min(fires[,2:6]),max(fires[,2:6])))lines(fires$PORTUGAL,cex=1,col="blue",lty=2)legend( "topleft" , cex = 1.3, , lty = c(1,2), legend = c("Spain","Portugal"), text.col = c("black"), col = c("red","blue"))



## 4.2 Histograms

The next type of chart we will see is the frequency histogram. It is a bar chart that represents the number of elements of a sample (frequency) that we find within a certain range of values.

To draw this type of plot R uses the function hist(), which requires as a mandatory argument a vector (or column/row of an array) with the data to be represented. As with all other chart types we have seen, we can use main, xlab… Let’s see at an example using the example data from the *regression.txt* file[10](https://remot-technologies.com/introduction-to-r/module-1/plotting-with-r.html#fn10)(See Table [4.1](https://remot-technologies.com/introduction-to-r/module-1/plotting-with-r.html#fig:table3)):

Show

entries

Search:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | **TavgMAX** | **Tavg** | **long** | **d\_atl** | **d\_medit** | **lat** | **elevation** |
| 1 | 20.3 | 14.55 | 614350 | 305753 | 152147 | 4489750 | 1423 |
| 2 | 23.8 | 16.1 | 593950 | 271251 | 183121 | 4522150 | 1100 |
| 3 | 24.4 | 18.15 | 567250 | 279823 | 204123 | 4512650 | 1229 |
| 4 | 20.1 | 14.05 | 609150 | 319257 | 150357 | 4475450 | 1610 |
| 5 | 20.3 | 13.85 | 629450 | 306673 | 138390 | 4491350 | 1402 |
| 6 | 21.9 | 16.2 | 666750 | 317312 | 103530 | 4489850 | 1022 |
| 7 | 24.4 | 17.35 | 659450 | 335694 | 102739 | 4468550 | 882 |
| 8 | 20.9 | 15.45 | 654050 | 347119 | 101473 | 4455350 | 828 |
| 9 | 21.2 | 15.2 | 666850 | 359349 | 85698 | 4446050 | 1308 |
| 10 | 24 | 17.3 | 654150 | 369714 | 91221 | 4432150 | 1186 |

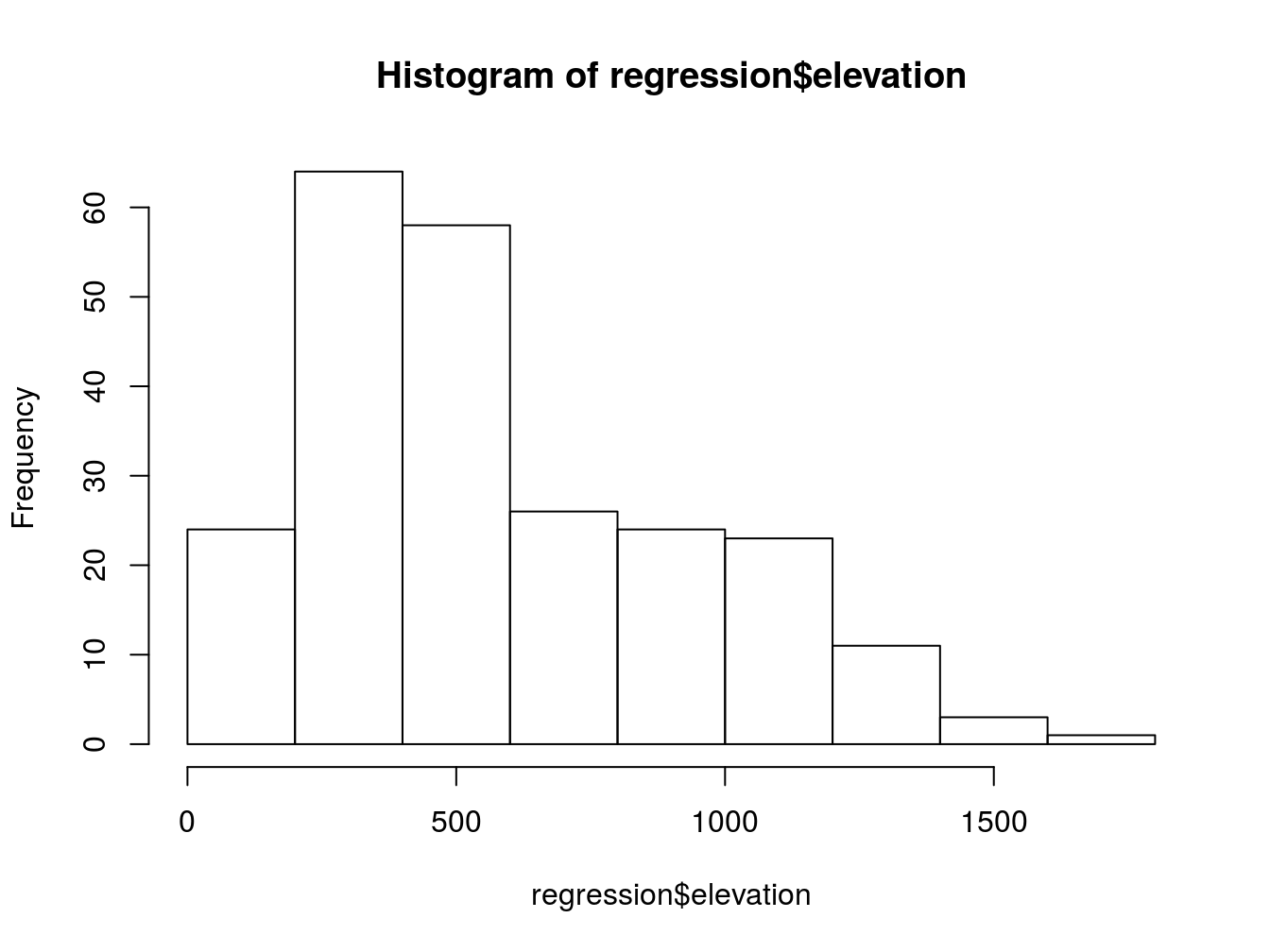
Showing 1 to 10 of 234 entries

Previous12345…24Next

Figure 4.1: Structure of the regression.txt file.

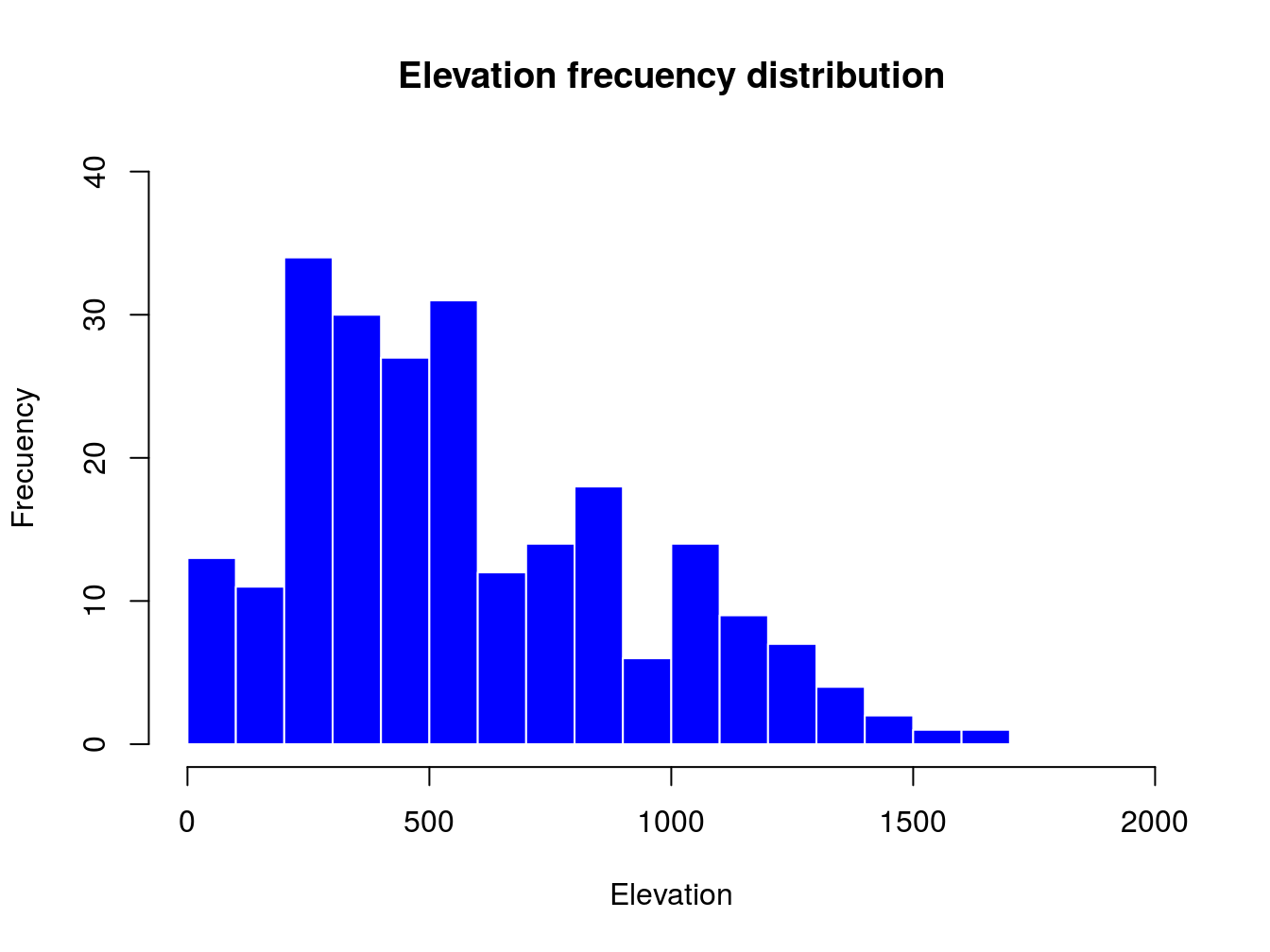
* Tavg\_max: maximum average temperture in June.
* Tavg: average temperture in June.
* long: longitude in UTM values EPSG:23030.
* lot: latitude in UTM values EPSG:23030.
* d\_atl: distance in meters to the Atlantic sea.
* d\_medit: distance in meters to the Mediterranean sea.
* elevation: elevation above sea level in meters.

hist(regression$elevation)



Then we use the arguments we have seen to customize the plot:

hist(regression$elevation,breaks=15,main="Elevation frecuency distribution",xlab="Elevation",ylab="Frecuency",col="blue",border="white",ylim=c(0,40),xlim=c(0,2000))



The only new parameters are:

* breaks, used to specify the number of bars in the histrogram.
* border, used the change the color of bar’s borders.

## 4.3 Scatterplots

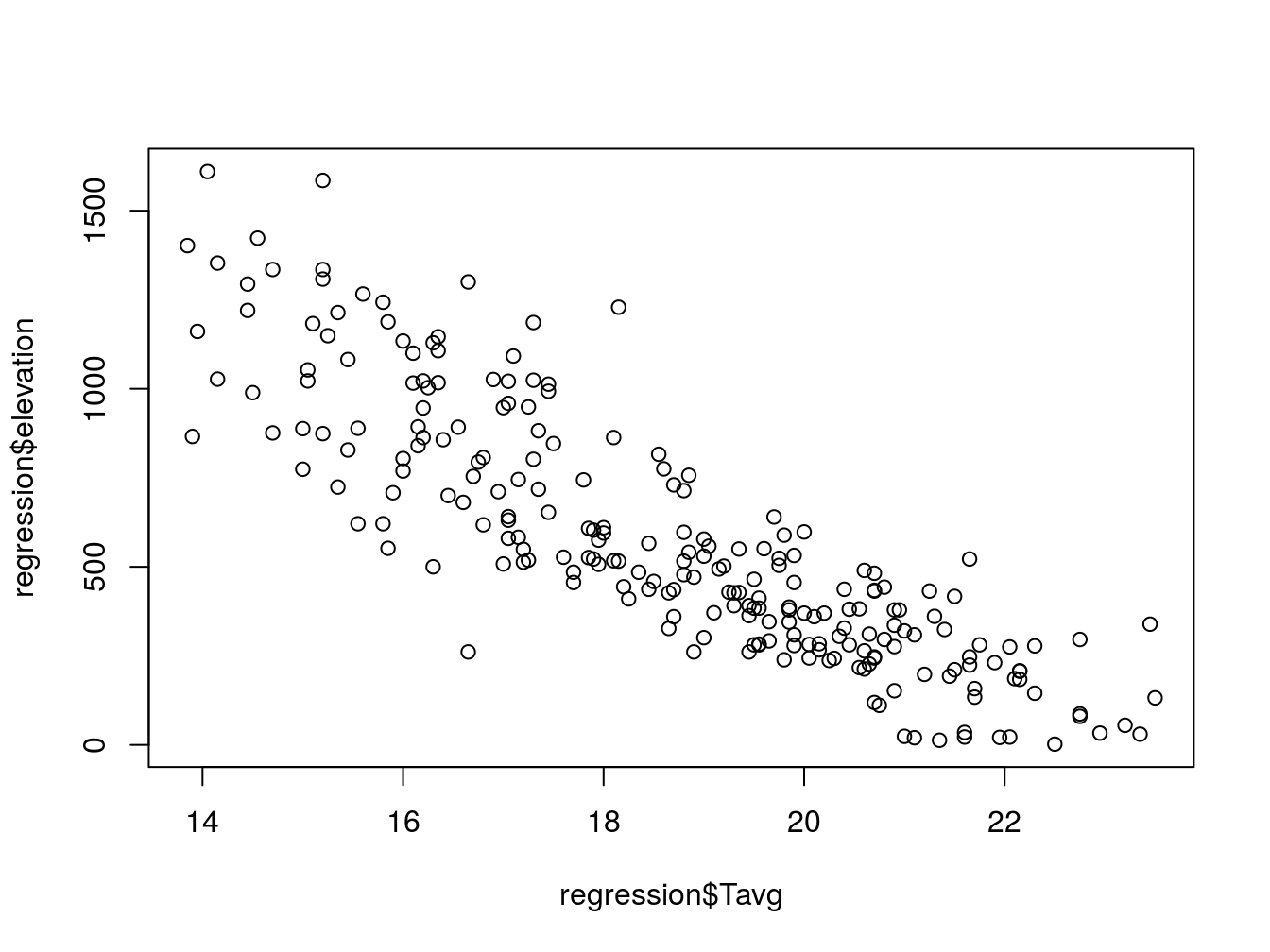
So far we have seen how to construct univariate graphs, ie, represent a single variable or data series. Next we will see a type of bivariate graph, the scatterplot. This type of chart is interesting to visualize relations between two variables, almost mandatory to explore correlation or collinearity in regression analysis. Let’s look at an example with our fire data.

In this case we introduce in the function plot() a second data argument (y

) with a second vector with the data right after the first data argument (x

):

plot(regression$Tavg,regression$elevation)

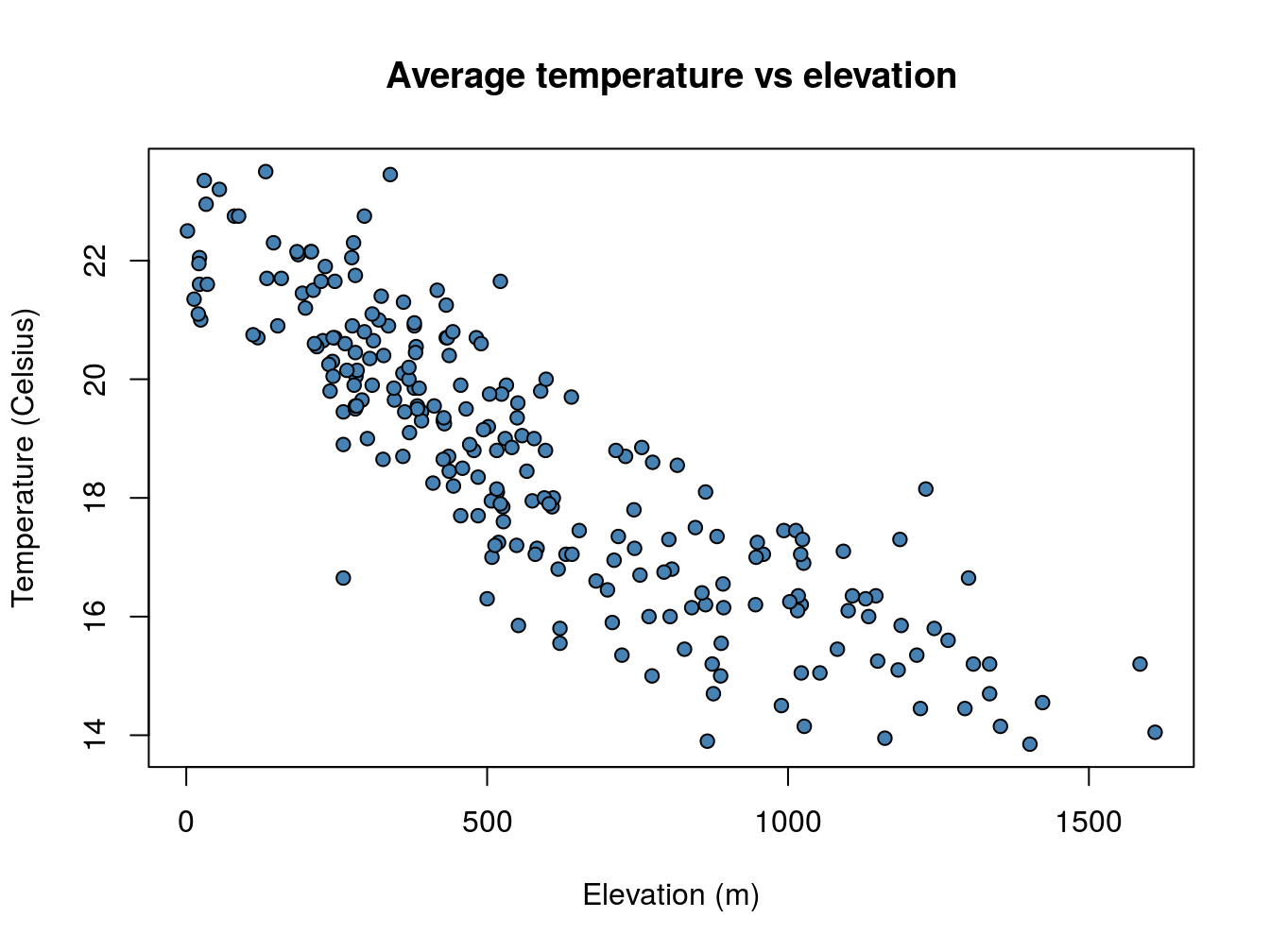


Note that the first data series goes to x−axis

and the second to the y

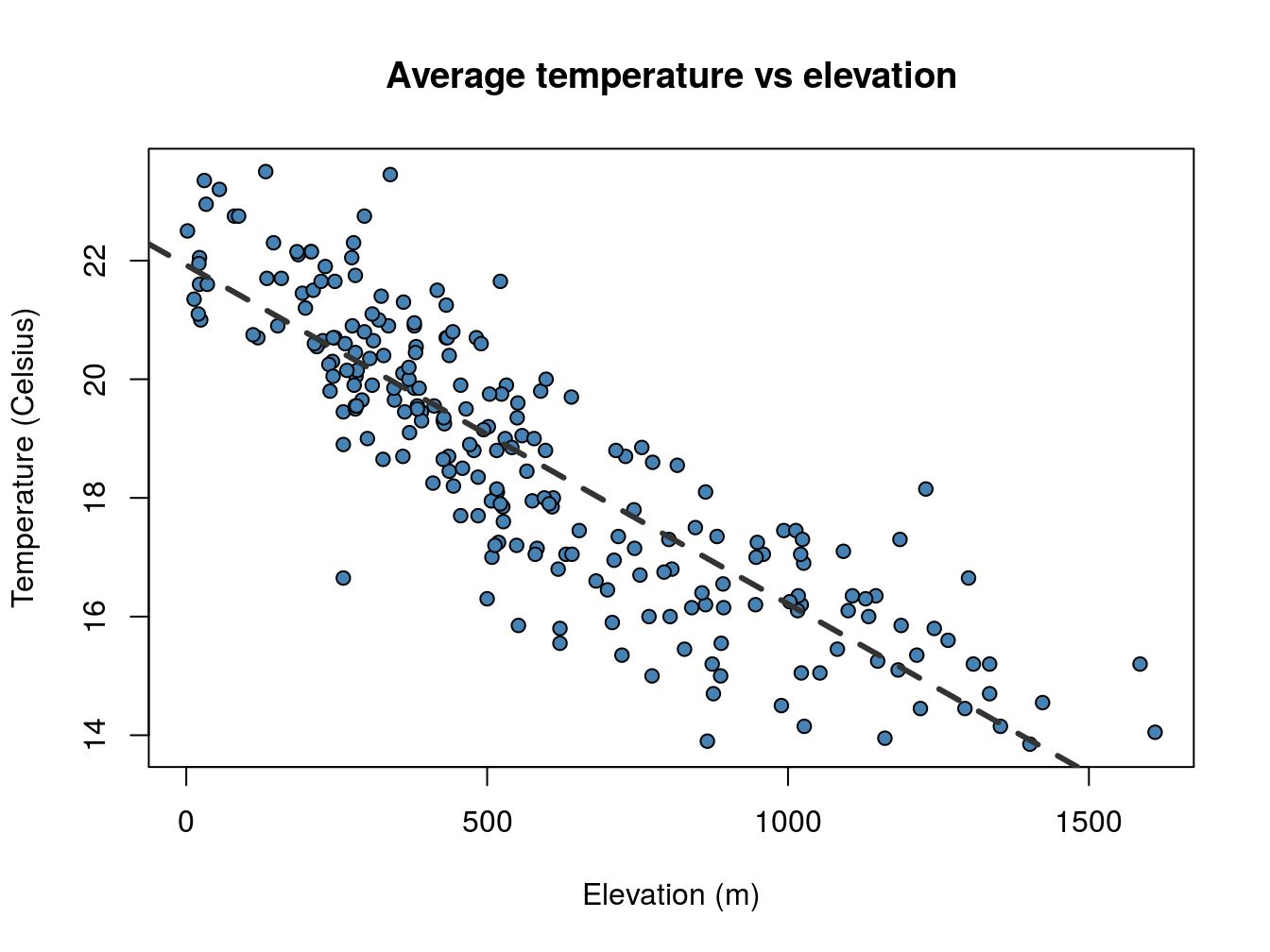
. Again, we can customize the appearnce and symbols similar to the regural do plot:

plot(regression$elevation, regression$Tavg, main='Average temperature vs elevation',ylab = 'Temperature (Celsius)', xlab = 'Elevation (m)', pch=21, col='black', bg='steelblue')



A particularly useful function in combination with scatterplots is abline() which allows to incorporate a *trend line* to the plot. We will further explore this later.

plot(regression$elevation, regression$Tavg, main='Average temperature vs elevation',ylab = 'Temperature (Celsius)', xlab = 'Elevation (m)', pch=21, col='black', bg='steelblue')abline(lm(regression$Tavg~regression$elevation), lty=2, col='gray20',lwd=3)



Pay attention to the order in which I have introduced the *x* and *y* data in plot() and abline(). It is the opposite!!.

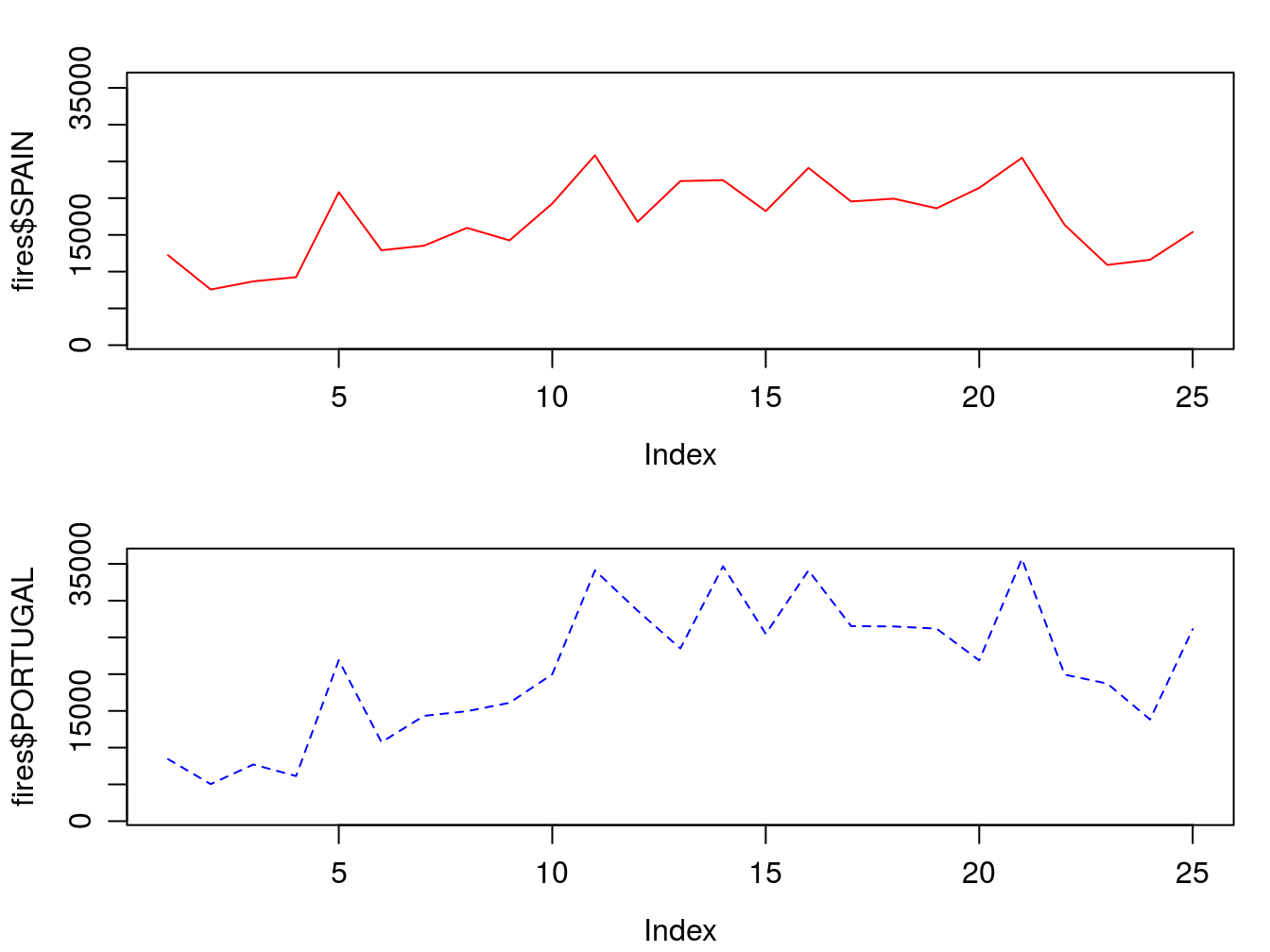
## 4.4 Multiple plots

To finish with plot creation we will see two last possibilities. The first one is how to combine several charts in a single figure and the second how to export an image file from our graphics.

### 4.4.1 Create a combined plot

Combining several graphs in R is possible thanks to the function par(mfrow = c(rows, columns)). Using this function we prepare the display window to include several graphs simultaneously:

par(mfrow = c(2, 1),mar=c(3.5, 3.5, 2, 1), mgp=c(2.4, 0.8, 0))plot(fires$SPAIN,pch=4,type='l', col='red', ylim = c(min(fires[,2:6]),max(fires[,2:6])))plot(fires$PORTUGAL,type='l',col="blue",lty=2, ylim = c(min(fires[,2:6]),max(fires[,2:6])))



### 4.4.2 Export an image file

Since R runs on so many different operating systems, and supports so many different graphics formats, it’s not surprising that there are a variety of ways of saving your plots, depending on what operating system you are using, what you plan to do with the graph, and whether you’re connecting locally or remotely.

The first step in deciding how to save plots is to decide on the output format that you want to use. The following table lists some of the available formats, along with guidance as to when they may be useful.

Here’s a general method[11](https://remot-technologies.com/introduction-to-r/module-1/plotting-with-r.html#fn11) that will work on any computer with R, regardless of operating system or the way that you are connecting.

1. Choose the format that you want to use. In this example, I’ll save a plot as a JPG file, so I’ll use the jpeg driver.
2. The only argument that the device drivers need is the name of the file that you will use to save your graph. Remember that your plot will be stored relative to the current directory. You can find the current directory by typing getwd() at the R prompt. You may want to make adjustments to the size of the plot before saving it. Consult the help file for your selected driver to learn how.
3. Now enter your plotting commands as you normally would. You will not actually see the plot - the commands are being saved to a file instead.
4. When you’re done with your plotting commands, enter the dev.off() command. This is very important - without it you’ll get a partial plot or nothing at all. So if I wanted to save a jpg file called “rplot.jpg” containing a plot of x and y, I would type the following commands:

jpeg('rplot.jpg', width = 800, height = 600, res=100)# Here goes the plotpar(mfrow = c(2,1),mar=c(3.5, 3.5, 2, 1), mgp=c(2.4, 0.8, 0))plot(fires$SPAIN,pch=4,type='l', col='red', ylim = c(min(fires[,2:6]),max(fires[,2:6])))plot(fires$PORTUGAL,type='l',col="blue",lty=2, ylim = c(min(fires[,2:6]),max(fires[,2:6])))dev.off()

## png ## 2