

An Introduction to R

Franz X. Mohr

March, 2018

Contents

1	Introduction	1
2	Installation	2
2.1	R	2
2.2	A development environment: RStudio	3
3	R(eal) Basics	4
3.1	Functions and packages	4
3.2	Before you start	5
3.3	Getting help	6
3.4	Importing data	7
4	Data transformation	9
4.1	Standard method	9
4.2	Manipulating data with the dplyr package	11
5	Saving and exporting data	13
5.1	Saving data in an R format	13
5.2	Exporting data into a csv file	13
5.3	Exporting data with the openxlsx package	14
6	Graphical data analysis with the ggplot2 package	15
6.1	Line plots	15
6.2	Bar charts	17
6.3	Scatterplots	19
6.4	Saving plots	20
7	Loops and random numbers	20
7.1	Cross-sectional data	21
7.2	Time series data	23
8	Estimating econometric models	25
8.1	Linear regression with cross sectional data	25
8.2	ARIMA models	27
9	References	28
9.1	Literature	28
9.2	Further resources	29

1 Introduction

Data analysis is becoming increasingly important in today's professional life and learning some basic data science skills appears to be a worthwhile effort. It can boost your productivity at work and improve your attractiveness on the job market. Among the myriad of programming languages out there, R has emerged as one of the most popular starting points for such an endeavour. It is a powerful application to analyse data, it is free and has a steep learning curve. Therefore, it enjoys great popularity among data scientists, statisticians and economists both in the private sector, the public sector and academia.

This introduction goes through an exemplary workflow in R and, by that, presents the basic syntax of the language and how to deal with common issues that can arise during a project. Although originally intended for people with an economic background, the text should be accessible to a broad range of readers. It does not require any previous programming skills or deep knowledge of economic methods. Only the last chapter on the estimation of econometric models requires an intuitive understanding of regression analysis.¹

After reading this guide the readers should have a basic understanding of R and the `dplyr`, `ggplot2` and `openxlsx` packages. Most sections end with a paragraph on further resources, which should provide interested readers with a starting point from which they can deepen their knowledge on the topic by themselves.

This introduction begins with the installation of the recommended software and continues with the explanation of some basic terminology. Afterwards it presents a workflow, which consists of the following steps:

- [Set up a new script for a new research project](#)
- [Read data into R](#)
- [Basic data manipulation with and without the `dplyr` package](#)
- [Create basic graphs with the `ggplot2` package](#)
- [Saving and exporting data](#)
- [Estimating basic econometric models](#)

Since in my experience data manipulation and creating graphs are among the most time consuming processes in the analysis of data, this introduction covers these steps a bit more intensively. However, if your data has already been pre-processed and you are only interested in estimating some models, it will suffice to read section [R\(eal\) Basics](#) and then skip to section [Estimating econometric models](#).

Most of the information presented in this guide can also be found on my website R-Econometrics.com². If you have any further suggestions on how to improve this text, feel free to send me an [e-mail](#) or leave a comment on the website.³

2 Installation

For this introduction I recommend to install two software packages on your system:

- *R*: The software, which does all the calculations.
- *RStudio*: The working environment, where all the code will be developed.

Note that the instructions of this section were written for a standard installation on a private computer. If you work on a corporate machine, the installation could be different due to different IT requirements of your company.

2.1 R

A proper installation of R is the prerequisite of everything that will follow. Fortunately, installing R is easy and can be done quickly by following these steps:

- Go to www.r-project.com.
- Click on *CRAN* in the download section on the left.
- Search the list for a server that is in or at least close to your country and click on it.

Windows

- Click on *Download R for Windows*.
- Click on *base*.
- Click on *Download R for Windows* and download the file.
- Execute the file and follow the installation instructions.

¹A basic understanding of linear regression models as presented, for example, on [Wikipedia: Linear regression](#) or in Wooldridge (2013) will suffice.

²The link will forward you to my Wordpress blog econometricswithr.wordpress.com. So you do not have to worry if the address of the website shown in your internet browser changes.

³I am grateful to Stefan Kavan, Stefan Kerbl and Karin Wagner for valuable comments.

Mac OS X

- Click on *Download R for (Mac) OS X*.
- Click on *R-.pkg* and download the file.
- Install the file.

Linux

If you use Linux, I trust that you are experienced enough that you can follow the installation instructions on the website. Alternatively, consult the person that put Linux on your system in the first place.

Once the installation is finished open R, enter $1+2$ into the so-called *command line* and press return so that you get

```
1+2
```

```
## [1] 3
```

Congratulations, you just did your first calculation in R! You can close the window now...

The simple addition of numbers was probably not the main reason why you started to learn R. Usually, people want to do much more calculations of a more complex nature, which includes multiple operations and code that goes over many lines. Your current installation of R is perfectly able to do that. The main challenge is just to write your code and bring it into R's command line. One solution to this problem could be to write several lines of code in a text document, copy and paste them into R for execution. But as you might imagine, this is a rather laborious way to proceed, especially when you are still developing your code. Therefore, I recommend to install further software, which makes it easier to work with R. This would be a so-called (integrated) development environment (IDE).

2.2 A development environment: RStudio

An IDE⁴ facilitates the work with a programming language. It assists in writing code, helps to find mistakes⁵, structures your workplace and allows for the quick execution of commands. This guide uses RStudio, which is widely accepted as a standard IDE for R projects. The software can be found at www.rstudio.com/. Search for the (free) version that matches your operating system, download it and install it by following the installation instructions.⁶

Once the installation is finished, open RStudio and take some seconds to familiarise yourself with its appearance. A first thing you might notice is that the left side looks somehow familiar to the R window from before. That is because it is that window. This so-called *R console* is the place, where all the calculations are done and the results and error messages show up. This integration of the R console in RStudio can be interpreted as if RStudio is a program that is built around R and can interact with it.

Next, click *File – New File* (or press the button with the plus sign on the top left) and then *R Script*. A new window will appear above the console, which is called the *editor*. This is the place, where all the code is written into a script, which can be used by you and your colleagues to develop your analysis.

A very useful function of RStudio is that you can execute single or multiple lines of code directly from the editor. Try it out by typing $1+2$ into the script and hit return. Nothing else will happen than the text cursor jumping to the next line. This is straightforward, because the editor is nothing else than a mere text editor. In order to execute the code from the editor window you have multiple choices:

- Use your mouse cursor and mark the area of the script, which you want to execute. Then click *Run* on the top right of the editor window.
- Mark the relevant area in the script again and use the keyboard shortcut `Ctrl + Return`.
- Put the text cursor in the same line as the command, which you want to execute and press `Ctrl + Return`. Note that this method will execute all lines around the cursor, which belong to the command.

⁴See, for example, [Wikipedia: Integrated development environment](#).

⁵Mistakes in the code are also called *bugs* and the process of cleaning your code from such bugs is referred to as *debugging*.

⁶Note that there are also other IDEs for R – e.g., Bio7, Eclipse or IntelliJ IDEA – and you are free to use them. But since RStudio comes with a lot of nice additional tools, which facilitate the typical R workflow, I will stay with it until something better shows up.

R	Package <i>name</i>	Function x
		Function y
		Function z
	Package <i>ggplot2</i>	ggplot
		geom_line
	Package <i>dplyr</i>	group_by
		summarise
		mutate

Figure 1: Packages and functions in R.

All of those three methods should lead to the same output in the console, [1] 3. For the time being, you can neglect the [1] in front of the answer.

3 R(eal) Basics

3.1 Functions and packages

R can be comprehended as a language that provides the basis for a system of code chunks, which have been developed by many people across the globe, who wanted to share their work. These code chunks are called *packages*. And each of those packages usually consist of at least one *function*.⁷ This structure is presented in the following graph. Beside a schematic description of a package and its functions in the upper part of the graph it also contains some real world examples using the packages *ggplot2* and *dplyr*.

Functions

Functions are what makes R work. So it is necessary to understand what inputs they require from the user in order to work properly. Usually, a function has a *name* and can contain further *arguments*, which are additional options the function should consider when it is executed. In R, the syntax of a function is as follows:

```
name_of_the_function(argument 1, argument 2, ...).8
```

Basically, a function requires the right arguments as inputs to produce some sort of output. In this introduction functions will usually require data and additional arguments, which determine the operations that should be applied to the data, e.g., `name_of_function(data = your_data, argument1 = 1, argument2 = "something", argument2 = "somethingelse")`. This rather abstract depiction should become clearer when we start with the examples.

Packages

A package can be comprehended as a collection of functions. Usually, a package serves a distinct purpose. For example, there are packages which are designed to generate nice graphics like the *ggplot2* package. There are packages, which implement certain statistical methods such as *igraph* or *sna* that were developed for network analysis. And the packages *foreign* and *haven* can be used to load data files, which were originally stored in the file format of a different statistics software such as SAS, SPSS or Stata.

The basic installation of R comes with a relatively small amount of pre-installed packages. Therefore, most packages must be installed by the user manually, before they can be used. This is a relatively simple process in RStudio. If you wanted to install, for example, the *dplyr* package, you would just have to do the following:

- Click on the tab *Packages* of the bottom right window
- Click *Install*

⁷R specialists may forgive me the slightly sloppy terminology. However, I have found it informative and just right for beginners with no previous knowledge of the topic.

⁸Note that this form resembles the mathematical way to write a function such as $F(a_i, b_i)$.

- Make sure that the field *Install from* says *Repository (CRAN, CRANextra)*⁹
- Enter *dplyr* into the field *Packages*
- Check the box *Install dependencies*
- Sometimes – for example when using a corporate computer – it can be necessary to check whether the path in *Install to library* is the right one. But this should be of less concern when working on a private computer.
- Click *Install*

The progress of the installation is shown in the console. During the installation of those packages, you will probably notice that some additional packages are installed, which you did not explicitly specify. This is because packages can depend on functions of other packages and, hence, they have to be installed as well. By checking the box *Install dependencies* R takes care of that and installs all the required packages automatically.

During the installation process R might also occasionally state that the *Source* package is newer than the *Binary* version and ask *Do you want to install from sources the package which needs compilation?*. This decision should not be of great importance for this guide. However, I recommend to answer this question by typing *n* into the console, press *Return* and install the *Binary* version, since it seems to cause less problems.

For the purpose of this introduction you *could* repeat these steps for the following packages:

- *ggplot2* for creating and analysing data with graphs.
- *reshape2* for further data transformation functions.
- *openxlsx* for reading and writing files in the *xlsx* format.
- *zoo* for structuring time series data.

Alternatively, you could just execute the following line:

```
install.packages(c("ggplot2", "reshape2", "openxlsx", "zoo"))
```

Once the installation of the packages is complete, you can click on *Packages* in the lower right window of RStudio and search for the packages you just installed. The round button with the *x* on the right side of the package list deinstalls a package. And if you click on a package's name, you will arrive at an index page, which contains a list of the package's functions.

3.2 Before you start

3.2.1 Create a working directory

Usually, people want to keep the folder structure on their computers tidy, so that they and their co-workers understand, which documents and data were used for a particular project – even if that project was finished months ago. This should also be the case when you work with R. Therefore, I recommend to create a new folder for every new project, which becomes the so-called working directory. There you put all the files, which are necessary for a project. For example, for this introduction I created the folder *r_intro* in my file explorer and I recommend to do the same on your computer now.

3.2.2 Obtain the original data

This introduction uses economic data from the [World Penn Tables \(WPT\) 9.0](#)¹⁰. The WPT contain annual time series data of macroeconomic variables for several countries. The data should be downloaded from the website and saved in the working directory now. Please, make sure that you download the Excel version of the data, i.e. the *pwt90.xlsx* file.

3.2.3 Create a new R script

You can either use the script you have already open from the 1+2 example above or click on the button with the plus sign on the top left of the editor window and then choose *R Script* to create a new script. Eventually,

⁹CRAN is a server structure, where approved R packages are stored and can be downloaded for free. Beside *Bioconductor* it is the most important source of R packages.

¹⁰Feenstra, R. C., Inklaar, R., & Timmer, M. P. (2016). PWT 9.0: A User's Guide. <http://www.ggd.net/pwt/>.

you should save that script in your working directory by clicking on the blue save button or on *File – Save As...*. It is recommended that the file has a telling name. For this intro the name *my_first_r_experience* will do. After clicking on *Save* the script saved in the *.R* format.

3.2.4 Set the working directory in R

Before we start our analysis, we should set the path to the working directory in R. The reason for this is a bit technical. Let's suppose that we want to use R to access a certain file on our computer. There is no way, that R would automatically know in which folder this file is located, if we only specified its name. Therefore, we would not only have to provide the filename, but also the path to it. This can become cumbersome if we want to access many files with R. But given that we have put all the necessary data for our project in the folder *r_intro*, we only have to enter the path once by setting the working directory. Basically, it tells R to access this directory by default, whenever it has to use a file on the hard drive.¹¹ The working directory of R can be set with the `setwd` function:

```
setwd("C:/path/to/your/working/directory/r_intro")
```

Recall that a function in R has a name and at least one argument. In the previous line `setwd` is the name of the function and the path to the working directory is the argument. Note that the path to the working directory has to be in quotation marks `"path/to/your/working/directory"` and the folders have to be separated by slashes `/` and *not backslashes* `\` as they would be inserted when you directly copy a path from the Windows explorer into R.

3.2.5 Load the packages you need

Having installed all the necessary packages does not mean, that we can use them without any further action. Before we use the functions of a particular package, we have to load the package whenever R is started.¹² This is commonly done with the `library(packagename)` function.¹³

```
library(dplyr)
library(ggplot2)
library(reshape2)
library(openxlsx)
```

For the purpose of this introduction you can ignore the warning messages that appear when the `dplyr` package is loaded. They just indicate that there is another package – in this case `base` and `stats` – which contains functions with the same name and R will from now on associate these names with the functions of the `dplyr` package. This is exactly what we want, but it can cause problems in more complex projects, like writing a package. Note that if we started R again, the functions would be associated with the `base` and `stats` packages again until we load the `dplyr` package again.

3.3 Getting help

Once the necessary packages are loaded, we can import the data into R. Since they come in an `xlsx` file, we need a function that can read it. Such a function is not included in the standard installation of R, but the `openxlsx` Package contains it: `read.xlsx`. To understand what kind of input this function needs in order to work, you can read its documentation by using the help function:

```
?read.xlsx
```

¹¹However, you can also import data from different folders than your working directory, if you provide the full path to a file. This can be useful if you have a common data base, which should not be copied too often in order to save memory or to ensure that you always access the most recent data.

¹²Note that you have to install a package only once, whereas you have to load a package every time you start R.

¹³There is also a function called `require` which, basically, does the same as `library`. However, it also gives out the value `TRUE` if the package was loaded successfully and `FALSE` if not. This can be useful when you want your script to check, whether a certain package is installed on a machine. For example, if a co-worker uses the script and misses a certain package, `require` can be used to indicate this.

which can also be written as `help("read.xlsx")`. Alternatively, you can click on the tab *Help* in the lower right window and enter `read.xlsx` into the search field. As soon as you start typing, RStudio will give you some suggestions on functions that fit your input and for which documentation is available.

After the execution of a help command a new page should appear in the lower right window. If not, click on the tab *Help* in the lower right window. The help page contains the title of the function, *Read from an Excel file or Workbook object*, followed by a short description. In our context the most important parts of the documentation are the following:

- *Usage* describes the arguments of the function, which can be specified by the user. The values to the right of the equality signs are the standard values, which are used if the users do not specify those values manually.
- *Arguments* provides further information on the possible specifications of the arguments.

For the purpose of this introduction only the first two arguments have to be specified:

- `xlsxFile` contains the name of the xlsx file you want to read.
- `sheet` is the number or name of the sheet you want to read.

It might also be worthwhile to take a minute to go through the other arguments that could be specified.¹⁴ Furthermore, you could look at the **Vignettes** of the package, which contain more detailed examples on how to use its functions. They can be accessed by browsing to the bottom of the help page and clicking on *Index*. You will arrive at the index page, which contains a list of all the package's functions for which documentation is available. Then click *User guides, package vignettes and other documentation* on the top of the page and on *openxlsx::Introduction* to open the document.¹⁵

3.4 Importing data

Since the name of our data file is `pwt90.xlsx`, the first argument of the `read.xlsx` function is `xlsxFile = "pwt90.xlsx"`. Furthermore, we know that the xlsx file consists of three sheets from looking into it with a spreadsheet program. We only want to read the content of the third sheet named *Data*. Thus, we can either use the argument `sheet = 3` or `sheet = "Data"`. I recommend to use the latter approach, because it is more reliable in case the order of the Excel sheets is changed in the future. So, the following line of code should be just right for our purposes:

```
read.xlsx(xlsxFile = "pwt90.xlsx", sheet = "Data")
```

If you hit `Ctrl + Return` and execute this line and the R console gives you an error message like

```
## Error in read.xlsx.default(xlsxFile = 'pwt90.xlsx', sheet = 'Data') :
##     Expecting a single string value: [type=character; extent=0].
```

open the `pwt90.xlsx` file with your spreadsheet program, click *Save as...*, save it under the same name – i.e., replace the existing file – and try to execute the line again.¹⁶

Note that the order of a function's arguments is not important as long as you provide the names of the arguments followed by the equality sign. The space between the entries is also not important, but it is good practice to use it, since it makes the code more readable. Furthermore, make sure that you write the names of objects and arguments correctly. This is especially true for lower and upper cases. R is very sensitive in this regard.

If the command worked, the console will be filled with a lot of numbers and signs and it will indicate that some rows were omitted. This is because the function `read.xlsx` only reads the xlsx file and R does not know what else to do with the output than just to display its content in the console. In order to change this behaviour, we use the operator `<-`.¹⁷ It assigns a certain input to an *object*, or so-called flat-file. In this example, it takes the output of the `read.xlsx` function as input and assigns it to a new object. The following

¹⁴The additional arguments can be quite useful, because they allow to import data from xlsx sheets which have not been prepared in a special way so that they could be imported into R.

¹⁵Note that not all packages have vignettes, but if they have, it is worthwhile to read them.

¹⁶According to [the package's development site](#) the problem might occur, because the xlsx file was automatically generated by another program. I have encountered this problem also when using the `read_xlsx` function of the `readxl` package.

¹⁷Instead of `<-` it is also possible to use the equality sign `=` to assign values to an object. However, `<-` is the more frequently used method although there are good arguments for `=`, which are of a rather technical nature.

code line assigns the content of the sheet *Data* of the *pwt90.xlsx* file to the object *pwt*. Of course, you could also use other names for the object like, e.g., *this_is_data_from_pwt90*, but *pwt* seems more convenient.

```
pwt <- read.xlsx(xlsxFile = "pwt90.xlsx", sheet = "Data")
```

Note that this line also represents the *typical form of code in R*: A function with some arguments is executed and its output is assigned to an object via the `<-` operator.

Also note that object names may never contain spaces, so `pwt data <- read.xlsx(xlsxFile = "pwt90.xlsx", sheet = "Data")` would result in an error.

If your data file has a different format, you have to search for the package, which is able to read it. This might seem a bit laborious, but fortunately the structure of those functions is very similar, so that this example should be representative for most other file formats. If not, open a search engine and enter, for example, *r read xyz data* and search the results for answers on the website stackoverflow.com. It is a forum, where people post questions on coding issues. Usually, the respondents give the right advice. The quality of the answers to a question can be assessed by the approval rating of the community and, even better, a check sign, which indicates that this answer can be regarded as the final solution to the respective problem.

Once the file is successfully loaded into R, an object will appear in the upper right window. This window contains all the objects, which are loaded in R and which can be used for further processing. At the moment this is only one object, *pwt*. But with every use of the `<-` operator you can create a new object or overwrite an existing one with something else. For example, you could just copy the object *pwt* and save the copy as *data_copy*:

```
data_copy <- pwt
```

Now there are two objects in the upper right window, where one is obviously redundant. In order to get rid of it, you can use the command `rm(data_copy)` which will **remove** the object from the memory. This can be very helpful when working with large samples that have millions of observations.

A very useful application of the `rm` function is `rm(list = ls())`, which removes all the objects in the upper right window and, thus, from your computer's memory. I usually put this command on top of a script to make sure that no remaining objects from a different project can cause problems in the new project. Feel free to try the command out and load the data into R again by executing

```
# Remove objects in the memory
rm(list = ls())
```

```
pwt <- read.xlsx(xlsxFile = "pwt90.xlsx", sheet = "Data") # Re-load PWT data
```

Note that the lines above contain *comments*, which are indicated by the route `#` signs. Comments are an essential part of every code, because they allow other people to understand your code more easily. It is good practice to use comments to explain, why you added a certain line of code. You should definitely make a habit out of using them.

R can handle a broad variety of data formats such as boolean¹⁸ or numeric values, dates or text¹⁹. These different kinds of data can also be structured differently. These categories of data formats and structures are called *classes*. And one of the most popular classes in R are a so-called *data frames*. They serve as the data input for most of the basic R functions and, hence, we will focus on them in this introduction.

Data frames are quite similar to standard spreadsheets, which might become a bit clearer when you either execute `View(pwt)` or, equivalently, click on *pwt* in the upper right window. This will open an additional tab in the editor window, where you can explore the data just like in a spreadsheet program.

Following a different approach you can execute the lines

```
names_pwt <- names(pwt) # Save names column names as distinct object
names_pwt # Show object
```

to extract the column names of the data frame *pwt* and save them as a new object with the name *names_pwt* (first line) and to display the names in the console (second line). Note that R will print the content of an

¹⁸Boolean values are either TRUE or FALSE.

¹⁹Textual data can also be referred to as *strings* or *character values*.

object in the console, if the object name is entered and executed just like before when we used the `read.xlsx` function without assigning its output to an object.

The output of the `names` function is new object named `names_pwt`. It is not a data frame, but a so-called *character vector*, which is another way to structure data in R. The quotation marks around its elements indicate that the content of the vector is text. Note that this is also indicated by the abbreviation *chr* to the right of the object name in the upper right window. Moreover, we get the same information when using the command `class(names_pwt)`, which gives "character" as output.

If you extend the data frame object by clicking on the white button to the left of the object name, you will also notice the abbreviation *chr* in some lines, whereas *num* (*numeric*) will appear in others. This indicates that every column of a data frame is a vector of a certain format. Note that although it is possible for data frames to consist of vectors with different data formats, a vector can only contain one kind of data.

4 Data transformation

This introduction presents two approaches to transform data by calculating the time series of GDP per capita for Austria. First, it uses the standard method to explain the basic syntax for vector and data frame manipulation in R. Then, it presents the solution to the same task by using the `dplyr` package, which is regarded to be more user-friendly.

4.1 Standard method

Since the original data frame `pwt` contains information on many countries and not just on Austria, it seems a good idea to start with the extraction of the Austrian observations. To achieve this, it is important to understand how R handles data frames.

Like in the case of spreadsheet tables, each cell of a data frame is defined by rows and columns. This means that each cell can be accessed by specifying a pair of row and column values. In R this is done by adding brackets to the object name and entering the respective row and column indices. For example, executing `pwt[1, 2]` gives the first value of the second column of the object `pwt`. Alternatively, it would be equivalent to write `pwt[1, "country"]` to access the same cell. The only difference would be that in the latter case, the name of the column was provided as character instead of its numeric index. Like in the case of extracting a particular sheet of an `xlsx` file, providing the column name is also preferable to merely specifying the column index, because it extracts the right column if the order of columns is changed.

Note that you can also access multiple rows and columns by using a series of row and column values like

```
pwt[21:25, 1:5]
```

where `21:25` in the row field specifies that you want to access rows 21 to 25 of the `pwt` object and `1:5` gives the first to fifth columns of the data frame. This command is equivalent to

```
pwt[c(21, 22, 23, 24, 25), c(1, 2, 3, 4, 5)]
```

and also to

```
pwt[c(21, 22, 23, 24, 25), c("countrycode", "country", "currency_unit",
                             "year", "rgdpe")]
```

where the row and column indices are provided individually as vectors, which are indicated by `c`.

After clicking on the `pwt` object in the upper right window and inspecting the data, we notice that the first two columns of the data frame contain information on the country name. The task to extract the Austrian observations becomes then to find those row numbers, where either the values in the column `countrycode` are equal to `AUT` or the values in the column `country` are equal to `Austria`.

In this example, we take the second approach – for no particular reason – and search for those values in the column `country`, which are equal to `"Austria"` and save the resulting logical vector as an object named `aut` by executing

```
aut <- pwt[, "country"] == "Austria"
```

In this line `pwt[, "country"]` tells R to access the column `country` in the `pwt` data frame.²⁰ The double equality sign is used for logical comparisons²¹ and gives TRUE if the value on the left hand side is exactly equal to the value on the right hand side. So, in this example, R checks for each entry of the `country` column whether it has the value "Austria" or not. Note that "Austria" has to be written in quotes, since it is a character.

Executing `aut` gives a series of TRUE and FALSE values, where TRUE indicates observations for Austria. This logical vector can now be used to extract the Austrian observations from the original sample by inserting it into the row field of `pwt`. When this is done R keeps all rows of the original sample, where `aut` takes the value TRUE, and drops all the other rows. The column field can remain empty, so that all columns of the original data frame are preserved.

```
data_aut <- pwt[aut, ]
```

Note that you could also use the command `which(aut)` to extract the positions, where the values of vector `aut` are TRUE and use `data_aut <- pwt[which(aut),]` to get the same result.

The new object `data_aut` is a data frame and contains only observations for Austria and all columns from the original sample. The next step is to calculate the GDP per capita ratio. This step will be presented in different ways, which lead to the same outcome. This should help to get a better intuition of the R syntax and make it easier to understand the code of other people, how might have a different coding style.

First, the columns "rgdpe" and "pop" are extracted into separate vectors and the values of the vector `rgdpe` are divided by the corresponding values of the vector `pop`. The result is saved in the vector `gdp.pc`.

```
rgdpe <- data_aut[, "rgdpe"] # Extract real GDP
pop <- data_aut[, "pop"] # Extract population
gdp.pc <- rgdpe / pop # Calculate GDP per capita
```

Using a different approach, the ratio is calculated more directly, because the step of saving each column separately is skipped:

```
gdp.pc <- data_aut[, "rgdpe"] / data_aut[, "pop"]
```

Again, each value of the vector resulting from `data_aut[, "rgdpe"]` is divided by its corresponding value of the vector resulting from `data_aut[, "pop"]`.

The following approach does the same as the previous, but uses a different, still equivalent, way to access the columns of the data frame:

```
gdp.pc <- data_aut$rgdpe / data_aut$pop
```

Note that this time, there is no need to use quotation marks for the column names, since the operator `$` automatically looks at the column names. However, this only works, because the column names do not contain spaces.

The following line of code tries to summarise the approaches presented above:

```
gdp.pc <- pwt[pwt[, "country"] == "Austria", "rgdpe"] / pwt[pwt[, 1] == "AUT", ]$pop
```

If you understand every part of this line, you basically understand the whole concept of data frames and, by that, most of the data syntax in R. However, although the line leads to the right result, it is messy, uses an inconsistent syntax and leads to redundant calculations, because the logical vector to extract the Austrian rows is calculated twice. Personally, I recommend to stay with the following style:

```
aut <- pwt[, "country"] == "Austria" # Find Austrian values
# Calculate GDP per capita ratio for Austria
gdp.pc <- pwt[aut, "rgdpe"] / pwt[aut, "pop"]
```

²⁰Note that it would be equivalent to write `pwt[, "countrycode"] == "AUT"`, because the values in `countrycode` are just the abbreviations of `country`.

²¹Other comparison signs are `<`, `<=`, `>`, `>=` and `!=` for checking if the value on the left side of the sign is lower, lower or equal, larger, larger or equal and different, respectively, from the value on the right side.

It minimises the amount of calculations and avoids the generation of additional objects in your memory. This might be of less concern when small samples are involved, but can become very important with large samples.

The numeric vector `gdp.pc` does not yet allow to attribute a GDP per capita value with a certain year. This can be changed by using the function `data.frame` to create a new data frame, where the first column contains the time values and the second the GDP values:

```
# Combine the year and GDP per capita series
data_aut <- data.frame("time" = pwt[aut, "year"], gdp.pc)
```

In this case, we overwrite the previous data frame `data_aut` with the result of the `data.frame` function. The argument `"time" = pwt[aut, "year"]` specifies that *time* should be the name of the first column of the resulting data frame and that this column should contain the values from `pwt[aut, "year"]`. The second argument specifies that the object `gdp.pc` is added to the right of the *time* column without any further specification of the column name. In this case, the function automatically derives the name of the second column from the object name. But it would be equivalent to write `"gdp.pc" = gdp.pc` instead.

Note that the calculations from above and the combinatino of time and GDP values only work, because the objects have the same length and the same order. This is something to keep in mind when working with functions that might drop certain observations or change the order of your data.

Assuming that the column names could be improved a bit more, we can execute

```
names(data_aut) <- c("year", "GDPPC")
```

to replace the existing column names of the data frame with new names. Again, this only works if the number of columns in the data frame matches the number of elements in the character vector, which contains the new names. We could also access single elements of the column names by providing the index. For example, if we wanted to change the name of the second column, we could do this using

```
names(data_aut)[1] <- "Year"
names(data_aut)[2] <- "GDP.pc"
```

Finally, executing

```
data_aut
```

prints the new data frame and we see in the console that we have successfully calculated GDP per capita for Austria from 1950 to 2014.

4.2 Manipulating data with the dplyr package

The standard method of data transformation presented above can be useful for many applications. However, there are tasks, where it can be quite laborious and slow. An alternative way to transform data is to use the `dplyr` package. It uses so-called *pipes*²² – indicated by the operator `%>%` – which forward the output of the part to its left as input for the part to its right. This results in slightly different code, but with potentially significant efficiency gains.

As above, we start with the extraction of Austrian observations. This is done with the `%>%` operator in combination with the function `filter`. The output is saved as a new object named `data_aut_dplyr`.

```
data_aut_dplyr <- pwt %>%
  filter(countrycode == "AUT")
```

The `%>%` operator tells R that the function on its right side, `filter`, should take the information on its left side, the data in `pwt`, as its data input. The `filter` function then extracts the Austrian observations.

Note that a further advantage of the `dplyr` package is that it does not require to put column names in quotes – at least as long as they do not contain spaces.

In the next step the GDP per capita ratio is calculated and added as an additional column to the existing sample by using the `mutate` function. The existing object `data_aut_dplyr` is overwritten by the same sample with an additional column named `GDP.pc`.

²²For further information on pipes see <http://r-posts.com/pipes-in-r-tutorial-for-beginners/>.

```
data_aut_dplyr <- data_aut_dplyr %>%
  mutate(GDP.pc = rgdpe / pop)
```

To keep things simple we only keep the columns `year` and `GDP.pc` by using the `select` function.

The column names can be changed with the function `rename`:

```
data_aut_dplyr <- data_aut_dplyr %>%
  rename(Year = year)
```

The resulting data frame should contain the same values as the result obtained with the standard method. It might seem as if the efficiency gain was not that high compared to the standard method. This is true, especially, because the example is relatively simple. However, this assessment might change a bit after executing the next code block, where the operations are performed with a single command. This is achieved by the sequential use of pipes.

```
# Use the pwt data frame %>% Extract data for Austria %>%
# Calculate GDP per capita in a new column of the data frame %>%
# Select year and GDP.pc columns
data_aut_dplyr <- pwt %>%
  filter(countrycode == "AUT") %>%
  mutate(GDP.pc = rgdpe / pop) %>%
  select(year, GDP.pc) %>%
  rename(Year = year)

head(data_aut_dplyr) # Look at the first values of the result
```

Note that it is also possible to split the code into multiple lines which are connected by pipes at the end of a line.

Again, we have successfully calculated the GDP per capita time series for Austria. However, knowing those numbers is nice, but it might be interesting to compare it to the GDP per capita of other countries, for example France, Germany and Switzerland.

This task only requires to change the specification of the `filter` function, where the equality sign is changed to a `%in%` followed by a vector of country names. And in order to distinguish the countries from each other, the column `country` is added to the `select` function and renamed so that it starts with a capital letter.

```
data_multi <- pwt %>%
  filter(country%in%c("Austria", "France", "Germany", "Switzerland")) %>%
  mutate(GDP.pc = rgdpe / pop) %>%
  select(year, country, GDP.pc) %>%
  rename(Year = year, Country = country)
```

Sometimes it might also be useful to calculate aggregate values over a group of observations. This is where the `dplyr` package becomes very useful. It can be used to group variables with the `group_by` function in combination with `summarise`. The following code uses the `pwt` data frame, extracts data for Austria, France, Germany and Switzerland, and uses the `group_by` function to tell R that it should apply the `summarise` function to each year separately. Afterwards, the column name of `year` is renamed again, so that it starts with a capital letter. The result is a time series of the aggregate GDP of the four countries.

```
data_agg <- pwt %>%
  filter(country%in%c("Austria", "France", "Germany", "Switzerland")) %>%
  group_by(year) %>%
  summarise(GDP = sum(rgdpe)) %>%
  rename(Year = year)
```

Working with the `dplyr` packages requires some practise. This is especially true, when somebody just started to work with it. However, you have now already learned the basic functions and syntax of the package and there is a lot of additional material out there, which assists in becoming more familiar with the functions of `dplyr`. And once somebody is quite familiar with the syntax of the package, I find so-called *cheatsheets* as

they are provided on the RStudio website especially useful.²³ Other sources might be blog articles, online training courses or stockoverflow.com for a particular problem.

At the end of this section on data manipulation I quickly want to mention a further package, which is useful for large data sets – e.g. above a million observations. It is called `data.table` and although the syntax might be a bit more complicated, it can be very quick and handy once you know how to use it properly. The package has a good introductory vignette and there is a cheatsheet on the RStudio website as well.

5 Saving and exporting data

5.1 Saving data in an R format

Data in the memory, i.e. data shown in the upper right window of RStudio, can be saved in R-specific file formats with the function `save`. The file formats can be `.RData` or `.rda`²⁴, where the latter is more or less just the short form of the former. A single object in the memory can be saved, for example, with

```
save(pwt, file = "pwt9.0.rda")
```

and multiple objects can be saved, for example, with

```
save(list = c("data_aut", "data_agg", "data_multi", "pwt"), file = "data.rda")
```

Note that in the first example, no argument has to be provided for the object that should be saved. By contrast, in the second example it is required to specify the argument `list` and provide a character vector, which contains the names of the objects we want to save.

To see how the data can be loaded back into R, delete all the objects in the memory with the `rm` function and use `load` to load the objects, which were saved in the last step.

```
rm(list = ls()) # Clear memory
load("data.rda") # Load saved data
```

5.2 Exporting data into a csv file

It can be necessary to export data into a csv file, because people, who do not use R, want to use the data too. For example, the task could be to export data on the GDP per capita of some countries in a way, where the time series of each country is put into a separate column. By that, it becomes easy for other people to produce a line plot in a spreadsheet program. Looking at the data in the object `data_multi` we see that the columns of the object are `Year`, `Country` and `GDP.pc`. But for standard spreadsheet programs it would be necessary to have the values of GDP per capita in a single column per country. This can be done with the function `dcast`, which comes with the `reshape2` package:

```
data_export <- dcast(data_multi, Year ~ Country, value.var = "GDP.pc")
```

The first argument in `dcast` specifies the data used as input. The second argument provides a formula of how the data should be casted, where variables on the left side of the tilde `~` are used as row attributes and variables to its right are used as further columns. The argument `value.var` contains the name of the column in the original data frame, which contains the values used to fill the cells in the resulting data frame.

Note that the function `melt` does the opposite of `dcast`. This can be very convenient when you import data from a csv or xlsx file and have to transform it so that the `ggplot` function can use it. It is used in the following way:

```
data_multi_2 <- melt(data_export, id.vars = "Year",
                    variable.name = "Country", value.name = "GDP.pc")
```

²³Cheatsheets are compact summaries for the use of packages. They can be found on the RStudio website: <https://www.rstudio.com/resources/cheatsheets/>.

²⁴Theoretically, there is also an `.rds` file format, which is hardly used.

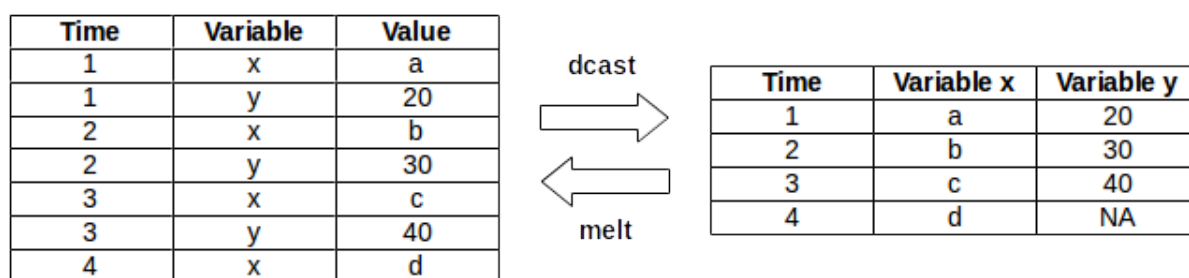


Figure 2: Table formats and the use of dcast and melt.

The function requires a data input – in our case `data_export` – and the argument `id.vars` tells R which columns were used to sort the data. The other arguments are optional. `variable.name` allows to specify a name for the column which contains the group names, i.e. the country names in our example. Otherwise the default column name would just be `variable`. And `value.name` allows to specify a name for the column of the output data frame, which contains the values of GDP per capita per country and year. This column would be named `value` by default.

The following table illustrates how the `dcast` and `melt` functions work. For further information visit the help page of these functions with `?dcast` and `?melt`, respectively.

Once we are finished with these data transformations, our results can be exported. This step is quite similar to the way data was imported at the beginning of this introduction. But since the output of the function is saved in the working directory by default, it is not necessary to assign it to an object with `<-`. The command is the following:

```
write.csv(data_export, file = "gdppc_multi.csv", row.names = FALSE)
```

Note that the function `write.csv` uses dots "." as decimal points. Depending on your location, this might cause problems. But it would be possible to use the `write.csv2` function or to use the argument `dec = ","` to specify the sign manually. See the documentation `?write.table` for further information.

The export to other file formats such as SAS, SPSS or Stata data files is usually analogous to this example.

5.3 Exporting data with the openxlsx package

A significant drawback of csv files is that they do not allow for multiple sheets in a single file. However, xlsx files can contain multiple sheets and the `openxlsx` package is able to read and create them. With this package the export of data is a little bit more complicated, but follows a straightforward logic.

First, a new – totally empty – workbook is created. This is equivalent to an xlsx file that does not contain any sheet.

```
wb <- createWorkbook()
```

Then, a new sheet called *GDP per Capita* is added to the new workbook.

```
addWorksheet(wb, "GDP per Capita")
```

The new sheet is filled with the content of the `data_export` data frame.²⁵

```
writeData(wb, "GDP per Capita", data_export)
```

Afterwards, the workbook is saved as xlsx file in the working directory, overwriting a target file, which might already exist.

²⁵Note that the formatting of the sheet in the final xlsx file might depend on the object class of the data you use. I recommend to always use data frames and not matrices in this context, because they allow for different data formats per column. Objects like matrices, which are only mentioned in this introduction, may lead to the problem that all columns in the final xlsx sheet are formatted as text. This might lead to issues, when people want to use those values for further calculations in a spreadsheet program.

```
saveWorkbook(wb, file = "R-Output.xlsx", overwrite = TRUE)
```

For further information I recommend to go through the vignettes of the package as already described above.

6 Graphical data analysis with the ggplot2 package

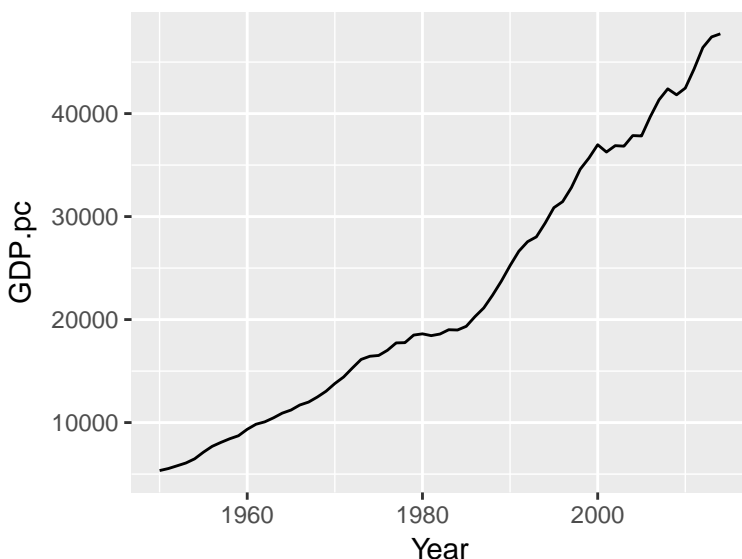
A major challenge in data analysis is to summarise and present data with informative graphs. The ggplot2 package was specifically designed to help with this task. Since it is a very powerful and well documented package²⁶, this introduction will only focus on its basic syntax, so that the user gets a better understanding of how to read the supporting material on the internet.

ggplot graphs are built with some kind of blocks, which usually start with the function `ggplot`. Its first argument contains the data object and the second argument is a further function called `aes`.²⁷ It controls, which columns of the data frame are used for the axes, colours, shapes of the data points and further features of the graph. The remaining blocks are separated by plus + signs and – if not specified otherwise – take the information from the first block and add a certain aspect to the graph, for example additional lines or data points. To understand what this all means, let us look at some basic examples.

6.1 Line plots

In order to plot the time series of the Austrian GDP per capita we use the following code:

```
ggplot(data_aut, aes(x = Year, y = GDP.pc)) +  
  geom_line()
```



After the execution of the line the graph should appear in the lower right window of RStudio. If not, just click on the tab *Plots* in that window and you should see it.

The first block in the line you just executed – `ggplot(data_aut, aes(x = Year, y = GDP.pc))` – tells R that it should use the data frame `data_aut` as the main source of data for everything that follows. And the `aes` function tells R that the column `Year` in `data_aut` should be used to map the data on the x axis and the corresponding values in column `GDP.pc` should be used to map the data on the y axis.

The second block – `geom_line()` – does not require any further specifications, because all the necessary information was specified in the first block. It just adds the line to the graph. To see that more clearly, you could just execute the first block and notice that only an empty graph with predefined x- and y-axes –

²⁶Beside the help function in R the [website of the ggplot2 package](#) contains very good documentation and examples of the package's functions. There is also a [cheatsheet on the website of RStudio](#).

²⁷This is short for "aesthetic mappings".

that have the range of the actual data – is displayed. The only thing missing is the line of the time series. Therefore, it is necessary to add the second block.

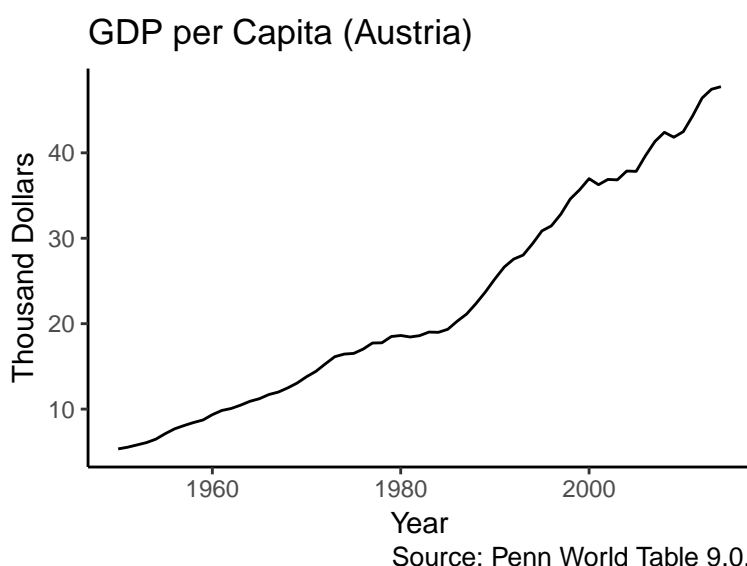
Note that it was not necessary here to write `data = data_aut` to specify the data used for the graph and – unless the column names contain spaces – it is not necessary to write the variable names in quotations marks neither. However, if you want to add a title manually, quotation marks still will be required. But do not worry. If this requirement is not met, R will give an error message anyway.

Note that in a lot of examples on the internet the first block of a ggplot graph is saved as a separate object. This can be very convenient, if somebody wanted to try out different graph specifications with the same data. In our time series example this could look like this:

```
g <- ggplot(data_aut, aes(x = Year, y = GDP.pc))
g + geom_line()
```

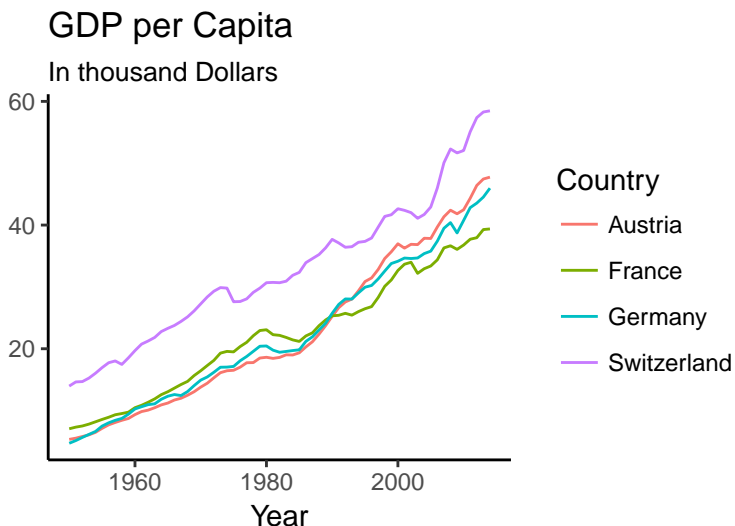
The layout of the current graph does not look very appealing. But this can be changed relatively quickly with some more blocks. In the following code the explanation of each line is given in the form of a comment.

```
# Specify the data and rescale the values of the y axis
ggplot(data_aut, aes(x = Year, y = GDP.pc/1000)) +
  geom_line() + # Add a line to the plot
  labs(title = "GDP per Capita (Austria)", # Add a title
       caption = "Source: Penn World Table 9.0.", # Add a caption at the bottom
       x = "Year", y = "Thousand Dollars") + # Rename the title of the x-axis
  theme_classic() # Usa a predefined theme of the graph
```



If you wanted to compare the evolution of GDP per capita across Austria, France, Germany and Switzerland as calculated in the last section, you could basically use the same code. The only thing we have to change would be that we use a different data source – `data_multi` – and to add `colour = Country` as an additional argument to the `aes` function. This tells R that it has to separate the GDP per capita values per year according to the name in the column `Country` in the `data_multi` data frame.

```
# Specify the data and rescale the values of the y axis
ggplot(data_multi, aes(x = Year, y = GDP.pc / 1000, colour = Country)) +
  geom_line() + # Add a line to the plot
  # The function labs allows to add and change labels like a title
  labs(title = "GDP per Capita",
       subtitle = "In thousand Dollars", # Add a subtitle
       caption = "Source: Penn World Table 9.0.", # Add a caption
       x = "Year") + # Rename the title of the x-axis
  theme_classic() + # Usa a predefined theme of the graph
  theme(axis.title.y = element_blank()) # theme() changes further features of a graph.
```

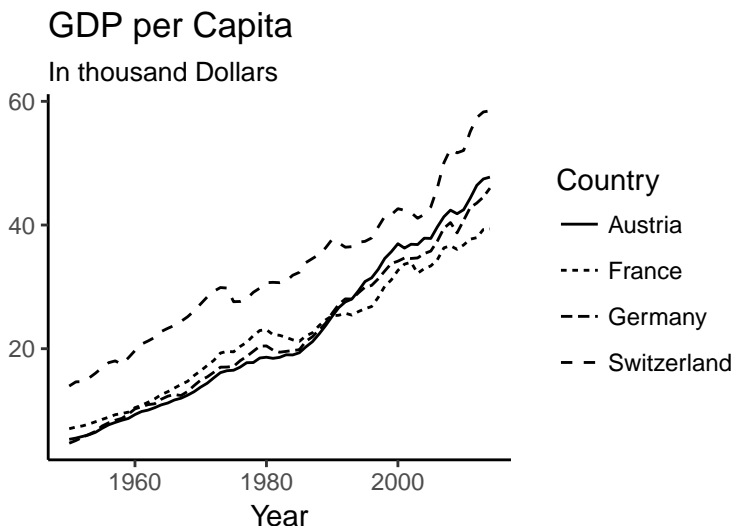


Source: Penn World Table 9.0.

```
# Here: delete the title of the y axis)
```

If you do not want to use colours, but instead prefer different line types to separate the countries' values, this can be done by replacing the colour argument with `linetype`.

```
# Specify the data and rescale the values of the y axis
ggplot(data_multi, aes(x = Year, y = GDP.pc / 1000, linetype = Country)) +
  geom_line() + # Add a line to the plot
  labs(title = "GDP per Capita", # Add a title
        subtitle = "In thousand Dollars", # Add a subtitle
        caption = "Source: Penn World Table 9.0.", # Add a caption
        x = "Year") + # Rename the title of the x-axis
  theme_classic() + # Usa a predefined theme of the graph
  theme(axis.title.y = element_blank()) # Delete the title of the y axis)
```



Source: Penn World Table 9.0.

6.2 Bar charts

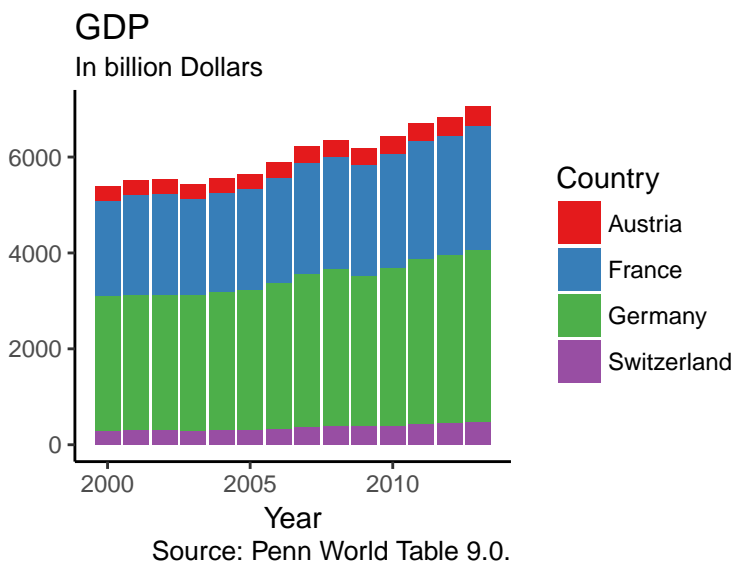
Another popular type of graphs are bar charts, which can be useful when the share of single components in an aggregate are of interest. In the following example a new data frame is created, which contains real GDP values for Austria, France, Germany and Switzerland from the year 2000 until 2013. Afterwards the bar chart is created. Again, nearly the same code as in the previous sub-section is used.

```

# Extract data
data_gdp <- pwt %>%
  filter(country%in%c("Austria", "France", "Germany", "Switzerland"),
         year >= 2000, year <= 2013) %>%
  select(year, country, rgdpe) %>% # Select the relevant columns
  rename(Year = year, Country = country, GDP = rgdpe) # Rename columns

# Create graph
# Specify the data and rescale the values of the y axis
ggplot(data_gdp, aes(x = Year, y = GDP/1000, fill = Country)) +
  geom_bar(stat = "identity") + # Add bars to the plot
  labs(title = "GDP", # Add a title
       subtitle = "In billion Dollars", # Add a subtitle
       caption = "Source: Penn World Table 9.0.", # Add a caption
       x = "Year") + # Rename the title of the x-axis
  # Change the standard set of colours used to fill the bars
  scale_fill_brewer(palette = "Set1") +
  theme_classic() + # Usa a predefined theme of the graph
  theme(axis.title.y = element_blank()) # Delete the title of the y axis)

```



The only differences are:

- A new data source, `data_gdp`, is used.
- Instead of `geom_line()` we use `geom_bar()` to indicate that the output should be a bar chart. Note that in this case the argument `stat = "identity"` must be used. Otherwise this will result in an error message. Furthermore, if you were exclusively interested in the distribution of the countries' shares in aggregate GDP and not in the overall size of GDP, you could add the argument `position = "fill"` to the `geom_bar` function and see what happens.
- The colour argument from above was replaced by `fill = Country` in the `aes` function. If this had not been done, the resulting graph would consist of grey bars with coloured frames.
- Instead of filling the bars with standard colours we use a different palette by adding the block `scale_fill_brewer(palette = "Set1")`. Note that this is also possible with line plots, but if the argument `colour = Country` is used in the `aes` function, we will have to use `scale_colour_brewer(palette = "Set1")` instead.

Note that if we had not specified the `fill` argument in the `aes` function, `ggplot` would just have summed up the values over all countries per year.

6.3 Scatterplots

Scatterplots can be useful to illustrate the correlation between two variables of a sample. For example, the PWT contains information on human capital, GDP and the number of persons who are engaged. Therefore, it could be interesting to check, whether there is a relationship between human capital and output per worker across countries. In order to do this, we filter the observations for the last available year, i.e. 2014, calculate the ratio of GDP per worker, select the relevant columns and rename their titles.

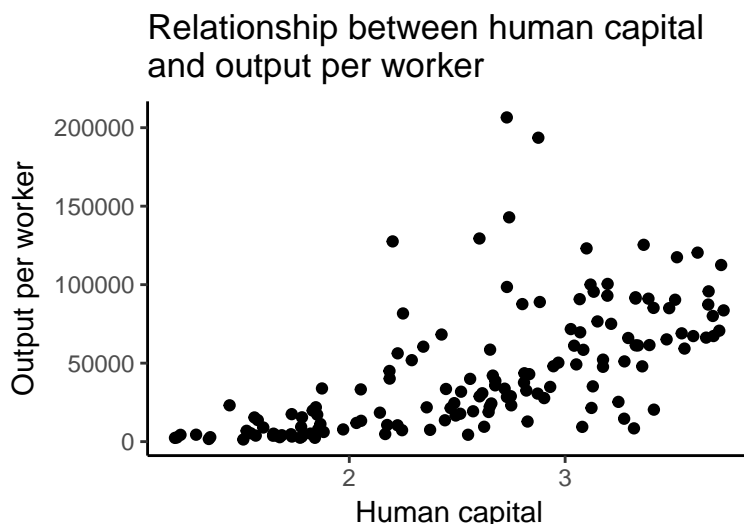
Looking at the resulting data frame with `data_hc` we notice that there are some cells which contain the value NA. This means that these values are **not available**. When plotting the data, `ggplot` will notice these data points, omit them from the sample and give a warning that they were removed. In order to avoid this warning, the missing values can be manually dropped with the function `na.omit`.

```
# Prepare data
# Use the pwt data frame and extract observations from 2014
data_hc <- pwt %>% filter(year == 2014) %>%
  # Calculate GDP per worker and add it to the existing data frame
  mutate(GDP.worker = rgdpe / emp) %>%
  select(country, GDP.worker, hc) %>% # Select the relevant columns
  rename(Country = country, Human.Capital = hc) # Rename columns

data_hc <- na.omit(data_hc) # Omit NA values
```

The scatterplot is created in a similar manner as the graphs created above. The first function contains the data frame `data_hc` with the data and the function `aes` specifies that the columns `Human.Capital` and `GDP.worker` contain data on the position of the points on the x- and y-axis, respectively. The second function, `geom_point()`, introduces the points of the scatterplot. The argument `title` contains the character `\n`, which indicates, that at this point a new line should be started. This results in a plot title that has two lines and the break occurs at the position of the `\n` sign.

```
# Specify the data and rescale the values of the y axis
ggplot(data_hc, aes(x = Human.Capital, y = GDP.worker)) +
  geom_point() + # Add points of the scatterplot
  labs(title = "Relationship between human capital\nand output per worker", # Add title
       caption = "Source: Penn World Table 9.0.", # Add a caption
       x = "Human capital", y = "Output per worker") + # Rename axes titles
  theme_classic() # Usa a predefined theme of the graph
```



Source: Penn World Table 9.0.

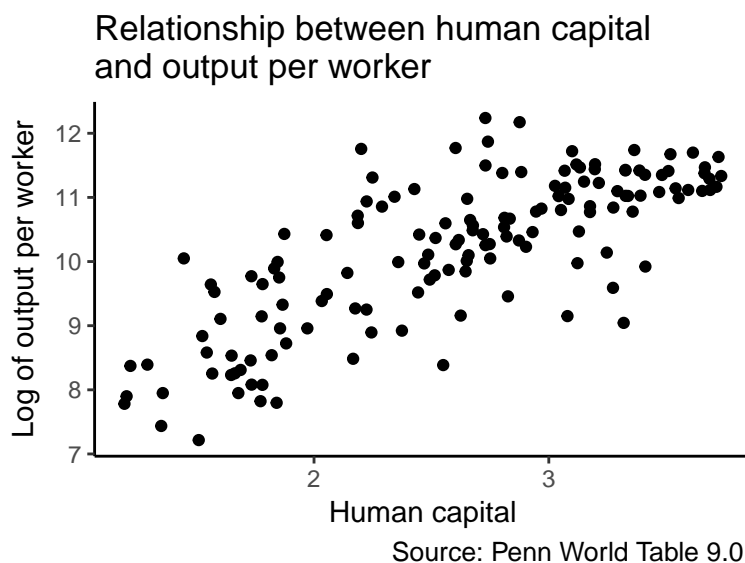
The plot implies that there is a positive correlation between human capital and output per worker, but output per worker seems to become more dispersed as the value of human capital increases. One solution to this issue might be to take the (natural) logarithm of output per worker and plot the graph again. The functions, which are required for this step, are already familiar. But instead of plotting the graph immediately, it is saved

as an object named `g`, which is plotted afterwards by executing `g`.

```
# Adjust data
data_hc_log <- data_hc %>% # Use the data_hc data frame
  mutate(GDP.worker = log(GDP.worker)) # Replace GDP per worker with its natural logarithm

# Create graph and save it as object g without plotting
g <- ggplot(data_hc_log, aes(x = Human.Capital, y = GDP.worker)) +
  geom_point() + # Add points of the scatterplot
  labs(title = "Relationship between human capital\and output per worker", # Add title
        caption = "Source: Penn World Table 9.0.", # Add a caption
        x = "Human capital", y = "Log of output per worker") + # Rename axes titles
  theme_classic() # Usa a predefined theme of the graph

g # Plot graph
```



The positive relationship in the graph appears more clearly now, although the effect seems to weaken as the value of human capital increases, which might indicate decreasing marginal returns of human capital in terms of GDP per worker. But that is not important here.

6.4 Saving plots

Since it is usually the case that a graph should be used in a text editor or presentation, it is necessary to export it. One solution is to plot the graph in the *Plots* tab in the lower right window and use one of the *Export* functions in that tab. Alternatively, it is possible to export a saved graph into the working directory – or any other folder when a path is specified – by using the function `ggsave`.

```
ggsave(plot = g, filename = "Graph_Human_Capital_Versus_Productivity.png",
        width = 4, height = 4)
```

Further practical information on the use of the `ggplot2` package can be found, for example, on the packages website <http://ggplot2.tidyverse.org/reference/> or on <http://www.cookbook-r.com/Graphs/>.

7 Loops and random numbers

Loops are among the most frequently used concepts in programming. They are particularly useful when it comes to the repeated execution of tasks with slightly different specifications per run. For example, loops can be used to find the best specification of an econometric model. Moreover, they can be used to create

artificial samples. In this section we will create two samples, which will be used to present the estimation of econometric models in the following section.

7.1 Cross-sectional data

Our first task is to generate a cross-sectional sample with one thousand observations and the following features:

$$y_i = 300 + 3a_i - 3b_i + \epsilon_i, \text{ with } a_i \sim U(30, 60), b_i \sim N(100, 100) \text{ and } \epsilon_i \sim N(0, 40) \text{ for } i = 1, \dots, 1000.$$

First, we set the seed of R's random number generator with the function `set.seed`. By that the random numbers generated by R should be the same on every other computer. Note that the seed number can be chosen freely.

```
set.seed(2018)
```

Then, we generate vectors with one thousand observations for each of the two randomly distributed variables a and b . Since a is uniformly distributed, we use the function `runif`. For the simulation of variable b , which is normally distributed with mean 100 and variance 100, we use the `rnorm` function. Note that `rnorm` requires the standard deviation of the normal distribution, and not its variance. Therefore, we must calculate the square root of the variance with the function `sqrt` before we can draw values from the distribution.

```
n <- 1000 # Set number of observations

# Random uniform distribution for variable a
a <- runif(n, 30, 60)
# Random normal distribution for variable b
b <- rnorm(n, 100, sqrt(100))
```

Next, we create an empty vector for the variable y with the function `rep`, which repeats the value NA one thousand times.

```
# Empty vector of dependent variable
y <- rep(NA, n)
```

Then we apply a loop to obtain values of the endogenous variable. There are different kinds of loops. The most frequently used type is a so-called *for* loop. It is based on the idea that a certain operation is repeated for different values of a single variable. In each round of the loop this variable takes a different value, which is then used in the same operation. Given the present example, this means that for each of the one thousand observations we want to apply the values in the vectors a and b to the equation from above to calculate a value of the endogenous variable y . In R this can be done with

```
for (i in 1:n){
  y[i] <- 300 + 3 * a[i] - 3 * b[i] + rnorm(1, 0, sqrt(40))
}
```

Basically, this code does the following: **for** every value i , which sequentially takes every value **in** the vector produced by `1:n`, we access the i th position of the vectors a and b , insert them into the model from above, add a normally distributed number and save the output as the i th value of vector y .

For later use, we produce the final sample by combining the vectors in a single data frame with

```
data_cs <- data.frame("y" = y, "a" = a, "b" = b)
```

Now the artificial sample is finished.

Equivalently, you could have created an empty data frame in the first place and fill it with the random samples afterwards. In that case, the loop does not access a single position in a vector, but uses the i th row of the respective column in `data_cs` to use it in the model equation.

```
set.seed(2018) # Reset the random number generator

# Create an empty data frame
data_cs <- data.frame("y" = rep(NA,n), "a" = NA, "b" = NA)
```

```

# Fill column a
data_cs[, "a"] <- runif(n, 30, 60)
# Fill column b
data_cs[, "b"] <- rnorm(n, 100, sqrt(100))

for (i in 1:n){
  data_cs[i,"y"] <- 300 + 3 * data_cs[i,"a"] - 3 * data_cs[i,"b"] + rnorm(1, 0, sqrt(40))
}

```

The resulting data frame should be identical to the one created before.

Note that the artificial sample could have been created without a loop in the following way

```

set.seed(2018) # Reset the random number generator

# Create a data frame with the
data_cs <- data.frame("y" = rep(NA,n),
                      "a" = runif(n, 30, 60),
                      "b" = rnorm(n, 100, sqrt(100)))

# Apply formula
data_cs[, "y"] <- 300 + 3 * data_cs[, "a"] - 3 * data_cs[, "b"]

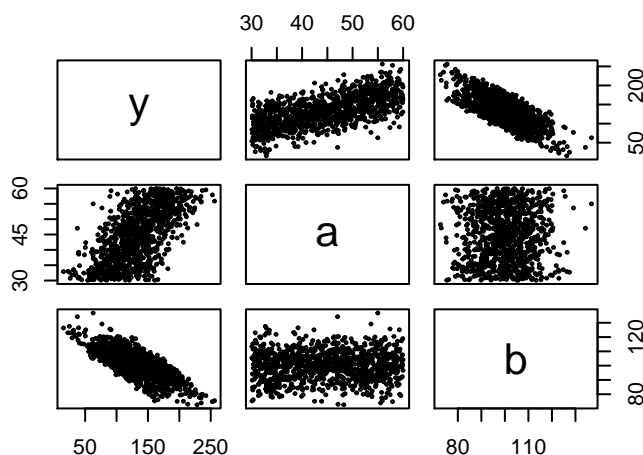
# Add random numbers
data_cs[, "y"] <- data_cs[, "y"] + rnorm(n, 0, sqrt(40))

```

Again, the resulting sample should be identical to the samples generated above. Also note that this last approach is always preferable to the application of a loop, since loops are relatively slow in terms of calculation time. This might not be a problem with a small sample as ours, but can become a serious issue with large data sets.

Finally, you could take a quick look at the data using R's standard `plot` function.

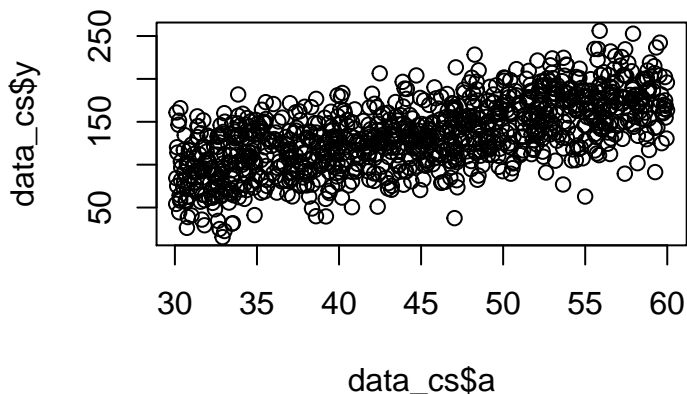
```
plot(data_cs)
```



The resulting graph is a scatterplot matrix, which describes the relationship between all variables in the data frame. The first row contains a box with the variable name and two scatterplots illustrating the correlation between variable y on the y -axis and variables a and b on the x -axis, respectively. The second row shows the same relationships, except that variable a is mapped on the y -axis and the other variables on the x -axis.

If you wanted to look only at the relationship between variable y and a , you could execute

```
plot(data_cs$a, data_cs$y)
```

Please, consult the help page of the `plot` function for further details on how to refine the graph.

7.2 Time series data

Next, we simulate a time series with one thousand observations that is generated by a simple autoregressive process of order one:

$$y_t = 0.75y_{t-1} + \epsilon_t, \quad \text{with } \epsilon_t \sim N(0,1) \text{ for } i = 1, \dots, T \text{ and } y_0 \sim N(0,1).$$

Again, we start by setting the seed of the random number generator to ensure reproducibility. Then, we save the desired number of observations plus one as object `tt`. We have to add one additional observation, because we also have to simulate a value for period 0 and R can only operate with indices larger than zero. This means that the first position of the resulting vector is reserved for period 0, the second for period 1 etc.²⁸ Note that although it would be convenient to name the object containing the maximum amount of observations `T`, it is not good to create such an object, because this letter is also interpreted as the abbreviation of the logical value `TRUE`. This can cause some unnecessary issues and, thus, should be avoided.

```
set.seed(2018) # Set the seed of the random number generator
tt <- 1000 + 1 # Specify number of observations plus number of initial values
```

Next, we create an empty vector `y.ts` with length $T + 1$ and fill its first position – the value in period 0 – with a random number from a standard normal distribution, i.e. from a normal distribution with zero mean and a unity variance. This is the standard specification of the `rnorm` function and, therefore, only the desired amount of random draws must be given as argument in this case.

```
y.ts <- rep(NA, tt) # Create empty vector of length T+1
y.ts[1] <- rnorm(1) # Generate a random value for y0
```

Then, we use a so-called *while* loop to simulate the autoregressive process. This type of loop differs from the *for* type, due to the alternative behaviour of the running indicator `i`.²⁹ A *for* loop goes through each value of the pre-specified vector that follows the `in` part at the beginning of the loop code. By contrast, a *while* loop requires an initial value of the indicator `i`, which is updated at the end of each round of the loop. The loop continues to run until `i` takes a value, which meets a condition that was specified by the user.

In our example, this means that we have to set the initial value of `i` before we can run the loop. The decision on this value depends on how the values of the time series are calculated during the run of the loop. Since the autoregressive process is of order one, it is necessary to access the past value of the vector in each round to calculate the contemporaneous value. To obtain the value in period 1 we have to multiply the value of period 0 with 0.75 and add a normally distributed error. In R this calculation is done by multiplying the first value of the vector `y.ts` with 0.75 and adding a random number with `rnorm`. Again, note that the first value of the vector has the index 1, although it represents the value in period 0. This means that the value of period 1 has the index 2 in vector `y.ts` and period 2 has the index 3 etc. Consequently, we set the initial value of `i` to 2 and start the loop.

The loop consists of two steps. First, we apply the formula of the data generating process to the data of the last period and obtain the contemporary value. Then, we update `i` by replacing it with its current value

²⁸By contrast, other programming languages like, for example, C++ use the index zero for the first position of a vector.

²⁹Instead of `i` any other object name can be used.

increased by 1. This new value of i will be used in the next round of the loop. These two steps are repeated until i is equal to $T + 1$ as specified by the exit condition $i \leq tt$. The resulting code is

```
i <- 2 # Set initial value of i
while (i <= tt) {
  # Calculate the value for period i based on the value in period i-1
  y.ts[i] <- 0.75 * y.ts[i-1] + rnorm(1)
  i <- i + 1 # Update the value of i
}
```

A drawback of such a *while* loop is that it is very important to include a line, which updates i , and to specify the right exit condition. Otherwise, R could be trapped in an infinite loop, which can only be stopped manually.

Alternatively, we could also use a *for* loop for the same task, which would result in the following code:

```
set.seed(2018) # Set seed of the random number generator

y.ts <- rep(NA, tt) # Create empty vector
y.ts[1] <- rnorm(1) # Generate a random value for y0

for (i in 2:tt){
  y.ts[i] <- 0.75 * y.ts[i-1] + rnorm(1)
}
```

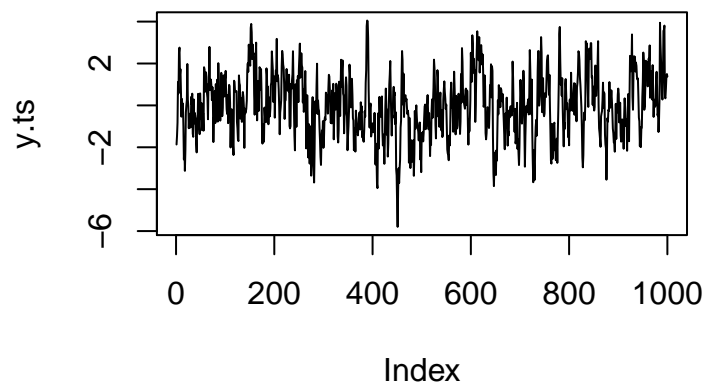
Personally, I would recommend to use this approach when working with time series data, because it is less prone to accidental mistakes, which might lead to an infinite loop.

Finally, we omit the first value of the vector $y.ts$, i.e. the value of period 0, to obtain a time series, which starts in period 1 and ends in period T . This is achieved by adding a minus sign in front of the index number.³⁰

```
y.ts <- y.ts[-1]
```

You can also take a quick look at the generated time series using R's standard `plot` function augmented by the argument `type = "l"` which indicates that the points should be connected by lines.

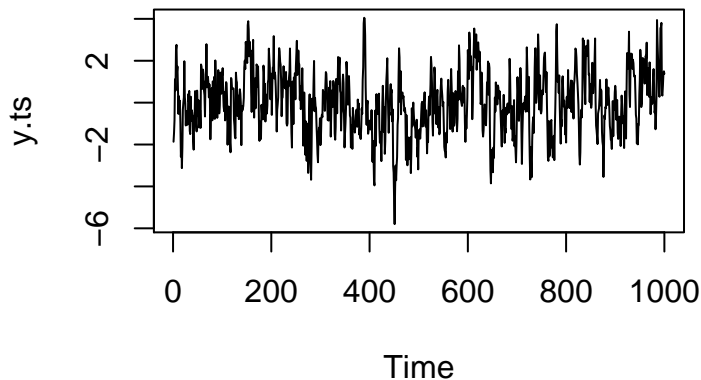
```
plot(y.ts, type = "l")
```



Alternatively, you could also use the command

³⁰Analogously, this syntax is also used to drop a row or column of a data frame by using, for example, `data[-2,]` or `data[, -c(3,8)]`, respectively.

```
plot.ts(y.ts)
```



which is a variant of the `plot` function that was pre-specified for time series data.

8 Estimating econometric models

Once you understand the basic syntax of functions and how to read their documentation the estimation of econometric models in R is straightforward. The only remaining challenges are to prepare your data and to find the right packages and functions for the task at hand.

For the examples presented in this section it is not necessary to install or load any further packages, because the required functions are contained in the `stats` package. It comes with the basic installation of R and is loaded automatically whenever a new session is started.

8.1 Linear regression with cross sectional data

The standard function for the estimation of *linear regression models* in R is `lm`. For basic applications it requires two inputs: the formula of the model and the data. The model formula consists of the name of the endogenous variable, here y , followed by a tilde `~` and the predictors of y separated by a plus sign. The data input can be a data frame, which should contain columns with the same name as the variables used in the formula. The resulting code is

```
lm.cs <- lm(y ~ a + b, data = data_cs) # Estimate model
```

Note that, alternatively, it would be possible to execute

```
lm.cs <- lm(data_cs$y ~ data_cs$a + data_cs$b) # Estimate model
```

which would yield the same results. The only difference is that the data argument was dropped and the data enters the function directly via the formula. It would also be possible to save the columns y , a and b of `data_cs` as separate vectors and execute `lm.cs <- lm(y ~ a + b)`. However, I personally recommend to use the approach that was presented first, because it keeps the data together at one place.

The result of the `lm` function is an object of class *list*. Beside classes like vectors, matrices or data frames, they are another way to structure data. Looking at the object `lm.cs` in the upper right window and extending it – with a click on the blue button with the white arrow – reveals that a list can consist of many other objects. It is similar to data frames, where each of the columns can contain vectors with different data formats. However, lists are a bit more flexible, because each of its elements can be a vector, matrix, data frame or even a list again. So it is also possible, for example, to create a list, which contains a list, which contains another list, which contains a data frame, which consists of one or more vectors.

Similar to data frames you can also use the function `names` to see the elements of the first layer of a list:

```
names(lm.cs)
```

```
## [1] "coefficients" "residuals"      "effects"        "rank"
## [5] "fitted.values" "assign"         "qr"            "df.residual"
```

```
## [9] "xlevels"      "call"          "terms"         "model"
```

We see that the output of the `lm` function consists of twelve elements.

Similarly to data frames, we can access each element of a list with the operator `$`. For example, to obtain the coefficients of the estimated model, we can use

```
lm.cs$coefficients
```

```
## (Intercept)      a      b
## 302.677770    3.008121 -3.032553
```

Alternatively, if you know the index of the element, you could also access this element by using double brackets. Given we want to extract the estimated coefficient values and know from the execution of the `names` function that they are contained in the first element of `lm.cs`, we can use

```
lm.cs[[1]]
```

```
## (Intercept)      a      b
## 302.677770    3.008121 -3.032553
```

to get the estimates. It is also possible to enter the element name as text between the brackets to get the coefficient values:

```
lm.cs[["coefficients"]]
```

```
## (Intercept)      a      b
## 302.677770    3.008121 -3.032553
```

Note that the syntax for lists is very similar to the syntax of vectors and data frames as presented above. In a nutshell, vectors are accessed with single brackets and no comma sign – `[]` – and matrices and data frames are accessed with single brackets and row and column indices separated by a comma – `[,]` – and lists are accessed by double brackets without comma – `[[]]`.

After extending the `lm.cs` object in the upper right window and scrolling down a bit, we notice that there is an element called `model`, which is a data frame with the three variables in the model.³¹ This provides a good example for how to access the multiple layers of a list object. If the task were to extract the 10th value of variable `b`, we could execute the following line

```
lm.cs$model$b[10]
```

Alternatively, we could also use `lm.cs[["model"]][,"b"][10]` or `lm.cs[[12]][10,3]` or `lm.cs[[12]][,3][10]`. All approaches give the same result. The only difference is the syntax used to access the elements of the respective object. But as already recommended above, I personally suggest to avoid the use of numeric indices when accessing the elements of an object whenever possible.

If we print the output of the `lm` function, not the whole content of the object will be displayed in the console

```
lm.cs
```

```
##
## Call:
## lm(formula = y ~ a + b, data = data_cs)
##
## Coefficients:
## (Intercept)      a      b
##    302.678      3.008    -3.033
```

Futhermore, the output of `lm` only contains the value of the estimated parameter values, but does not provide any further information such as the significance levels of the estimated coefficients or a general goodness-of-fit measure like R-squared. Such test statistics can be obtained with the `summary` function. It takes the output of an estimator function like `lm` as input and processes its content to obtain further indicators such as R-squared and the residual standard error, which is used to calculate the t values of the variables in the model.

³¹Not that the data for the intercept term would just be a vector of ones, which need not be included in this data frame.

```
s.cs <- summary(lm.cs)
s.cs

##
## Call:
## lm(formula = y ~ a + b, data = data_cs)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -19.2184  -4.0176  -0.1972   4.1295  19.0714
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept) 302.67777    2.20770   137.1  <2e-16 ***
## a             3.00812    0.02296   131.0  <2e-16 ***
## b            -3.03255    0.01957  -155.0  <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 6.296 on 997 degrees of freedom
## Multiple R-squared:  0.9761, Adjusted R-squared:  0.976
## F-statistic: 2.033e+04 on 2 and 997 DF,  p-value: < 2.2e-16
```

Again, the `summary` function produces a list with elements that can be extracted with the `$` operator.

Last, note that `lm` automatically includes an intercept term unless `-1` is added to the formula. For example, forcing R to estimate the same model without an intercept would result in

```
lm.no.intercept <- lm(y ~ -1 + a + b, data = data_cs) # Estimate
summary(lm.no.intercept) # Show summary
```

For more information on the implementation of different estimators in R you can visit my website [R-Econometrics](#). Among other introductions, it contains a list of packages and functions that can be used to estimate the examples in the main text of the Wooldridge (2013) econometrics textbook. Alternatively, you can consult Heiss (2016), which is an introductory econometrics textbook that works with R and is also based on the content of Wooldridge (2013). Whereas Heiss (2016) is a standalone textbook, the collection of examples on my website is some kind of companion for Wooldridge (2013).

8.2 ARIMA models

ARIMA models are among the standard approaches in the analysis of time series data and we can use the `arima` function to estimate them in R. The user only has to provide the vector of the time series and a vector of length three, which contains the order of the autoregressive part, the degree of differencing required to make the time series stationary and the order of the moving average part, respectively. Since we already now the true model, we set the AR order to 1 and the other values to zero, because the series is stationary and does not contain an MA process by construction.

```
ar.ts <- arima(y.ts, order = c(1,0,0)) # Estimate
ar.ts # Print the result in the console

##
## Call:
## arima(x = y.ts, order = c(1, 0, 0))
##
## Coefficients:
##      ar1  intercept
##    0.7381    0.0132
## s.e. 0.0213    0.1221
##
## sigma^2 estimated as 1.027:  log likelihood = -1432.75,  aic = 2871.5
```

After executing the code and looking at the results we notice that R automatically included an intercept term. But since we know that the data generating process does not include a deterministic term, we omit it by setting `include.mean = FALSE`.

```
ar.no.intercept <- arima(y.ts, order = c(1,0,0), include.mean = FALSE) # Estimate
ar.no.intercept # Print the result in the console
```

```
##
## Call:
## arima(x = y.ts, order = c(1, 0, 0), include.mean = FALSE)
##
## Coefficients:
##          ar1
##      0.7382
## s.e.  0.0213
##
## sigma^2 estimated as 1.027:  log likelihood = -1432.76,  aic = 2869.51
```

Similar to the `lm` function, `arima` produces a list with several elements and further functions can use these elements to calculate useful numbers like forecasts. However, note that there is no `summary` function for the output of the `arima` function. This is hardly necessary, because the output of the function already contains goodness-of-fit indicators and alike variables.

For a more extensive treatment of time series analysis in R see, for example, Hyndman & Athanasopoulos (2014), Hyndman & Athanasopoulos (2018) or the respective chapters in Wooldridge (2013). [R-Econometrics](#) also contains a small, but growing section on time series topics.

9 References

9.1 Literature

- Butts, C. T. (2016) [sna: Tools for Social Network Analysis](#). R package version 2.4.
- Dowle, M., & Srinivasan, A. (2017) [data.table: Extension of data.frame](#). R package version 1.10.4-3.
- Heiss, F. (2016) Using R for Introductory Econometrics. CreateSpace Independent Publishing Platform. <http://urfie.net/>.
- Hyndman, R. J., & Athanasopoulos, G. (2014) Forecasting: Principles and Practice. otexts. <https://www.otexts.org/fpp>.
- Hyndman, R. J., & Athanasopoulos, G. (2018) Forecasting: Principles and Practice. otexts. <http://otexts.org/fpp2/>.
- R Core Team (2017) [R: A language and environment for statistical computing](#). R Foundation for Statistical Computing, Vienna.
- Walker, A. (2017) [openxlsx: Read, Write and Edit XLSX Files](#). R package version 4.0.17.
- Wickham, H. (2007) [Reshaping Data with the reshape Package](#). Journal of Statistical Software, 21(12), 1-20.
- Wickham, H. (2009). [ggplot2: Elegant Graphics for Data Analysis](#). Springer New York.
- Wickham, H., & Miller, E. (2017) [haven: Import and Export 'SPSS', 'Stata' and 'SAS' Files](#). R package version 1.1.0.
- Wickham, H., Francois, R., Henry, L., & Miller, K. (2017) [dplyr: A Grammar of Data Manipulation](#). R package version 0.7.4.
- Wooldridge, J. M. (2013) Introductory Econometrics: A Modern Approach. (5th ed.). Mason, OH: South-Western, Cengage Learning.
- Xie, Y. (2017) [bookdown: Authoring Books and Technical Documents with R Markdown](#). R package version 0.5.

Zeileis, A., & Grothendieck, G. (2005) zoo: S3 Infrastructure for Regular and Irregular Time Series. *Journal of Statistical Software*, 14(6), 1-27. [doi:10.18637/jss.v014.i06](https://doi.org/10.18637/jss.v014.i06)

9.2 Further resources

[Cookbook for R by Winston Chang](#): For a more intensive treatment of the topics presented in this introduction.

[DataCamp](#): Online learning platform for R and other programming languages.

[ggplot2](#): Documentation for the ggplot2 package.

[R-Bloggers](#): Website referring to blog entries on all kinds of R examples.

[R-Econometrics](#): Website with short introductions into R and links to further resources.

[RStudio](#): The RStudio Website with further resources and useful *Cheatsheets*.