R for Beginners

Draft II

Massimiliano Porto*

October 26, 2017

*Kobe University

Contents

1 PART 1

1.1 R and RStudio

R is a free software environment for statistical coumputing ang graphics.
R Studio is an integrated development environment that makes R easier to use.

1.2 RStudio: Basics

1.2.1 RStudio Interface

If you open **Rstudio** you will see a screen like in figure 1.1. It is divided in 4 panes.

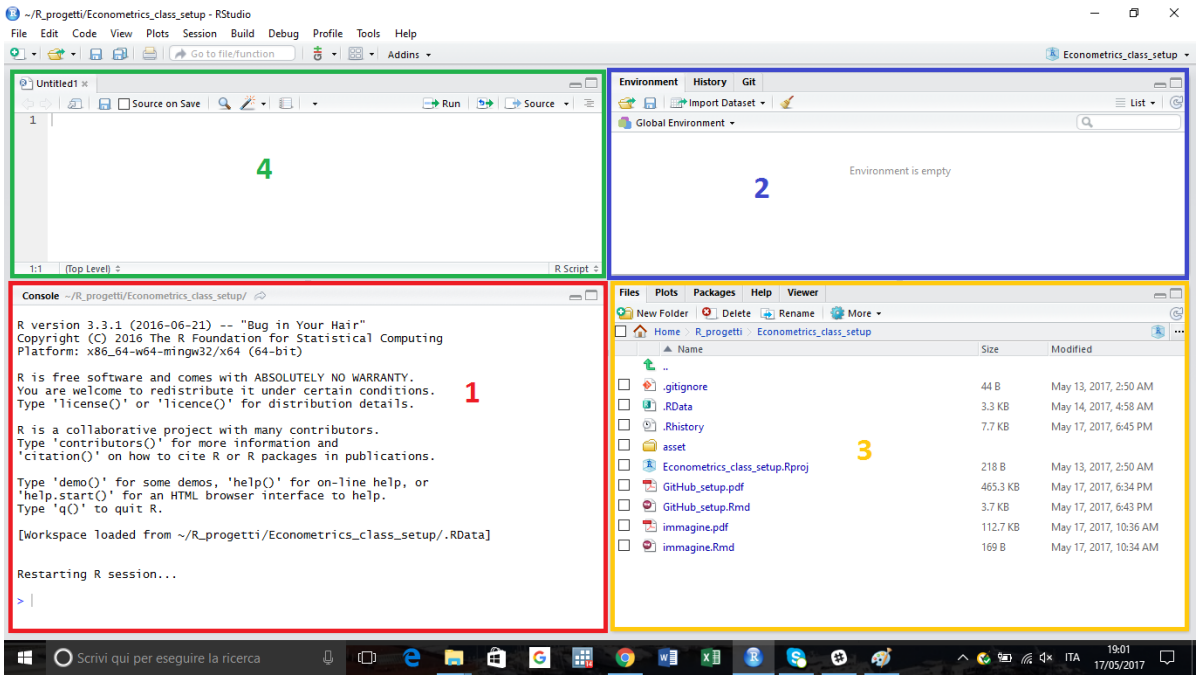


Figure 1.1: RStudio panes

Console pane: the **console** is the pane we numbered 1 in the figure 1.1. Here you write your code, called *command* in *R language* because you command your computer something to do.

Environment pane: the **environment** pane (2 in figure 1.1) is the place where that stores all the objects you create in R.[1] We will discuss more about objects later.

Files, plots, packages,.. pane: the pane number 3 in figure 1.1 is where you find your files, the packages you can install to improve the capabilities of R, where you can visualize the plots you create etc.

Source pane: **source** pane (4 in figure 1.1) provides you different ways to write and save your code. Later we will cover **R Script** and **R Markdown** in detail.

However, if you are not satisfied with this visualization you can modify it by clicking on the **toolbar** View > Panes > Pane Layout.

1.2.2 Launch a new project

A project is a place to store your work on a particular topic (or project). Create a project called**"Econometrics"** for all of your work in this course.

Follow the procedure as in figures 1.2, 1.3, 1.4.

Click on the R symbol in the top hand right corner, click New Directory > Empty Project and then write the directory name and click create project.

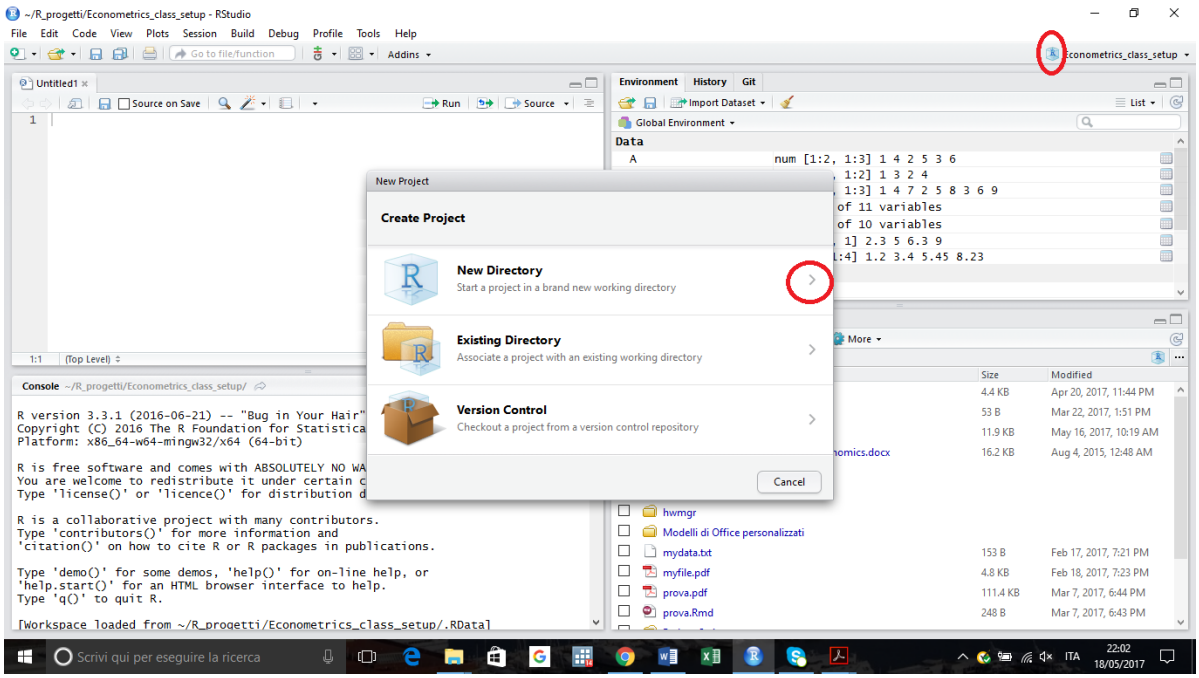


Figure 1.2: Launch a new project

If you want to check if you are in the right directory enter the following in the console pane.

[1] To be precise, the environment pane doesn't show hidden objects. (it's a quite technical issue.)

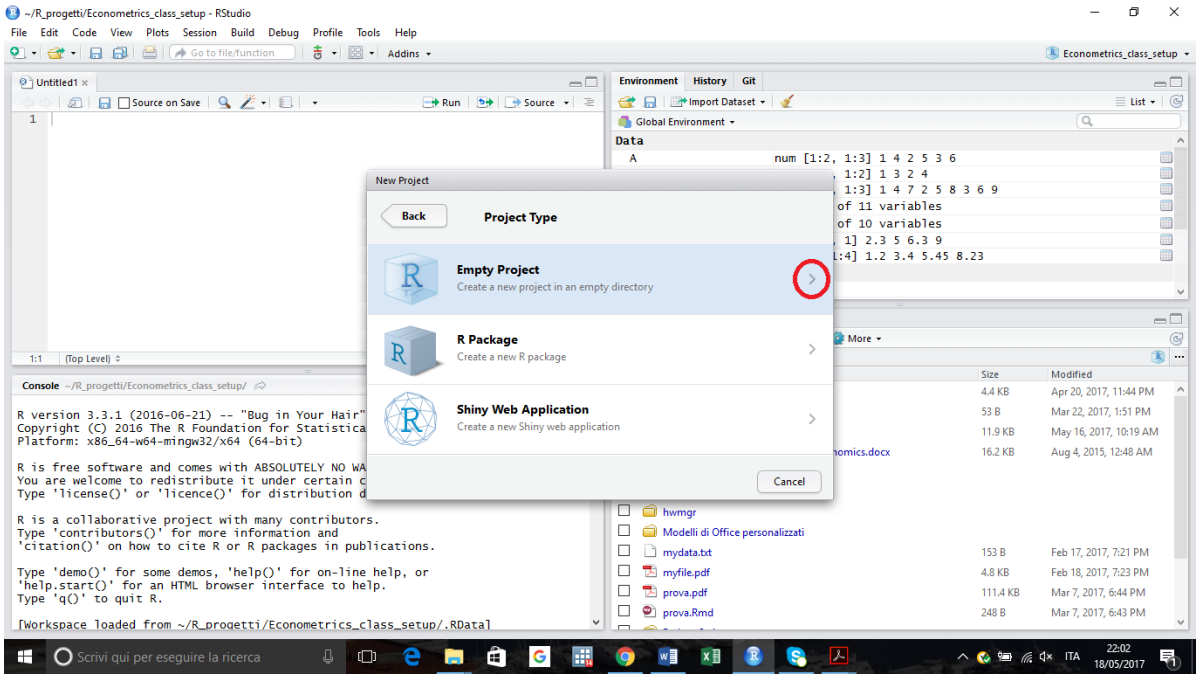


Figure 1.3: Launch a new project

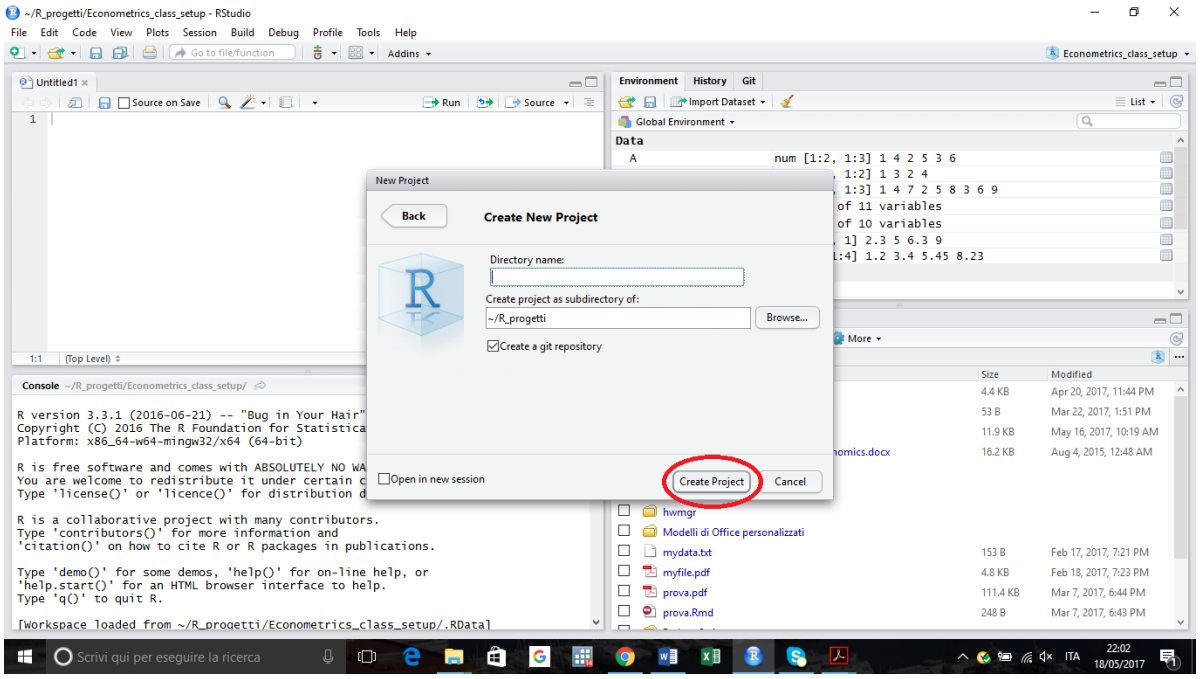


Figure 1.4: Launch a new project

```
getwd()
```

```
## [1] "C:/Users/porto/OneDrive/Documenti/R_progetti/Econometrics_class_setup"
```

If you want to change it

```
setwd()
```

write the new directory path in the brackets or from **Files, plots, packages...** pane by clicking on **more** as in figure **??**.

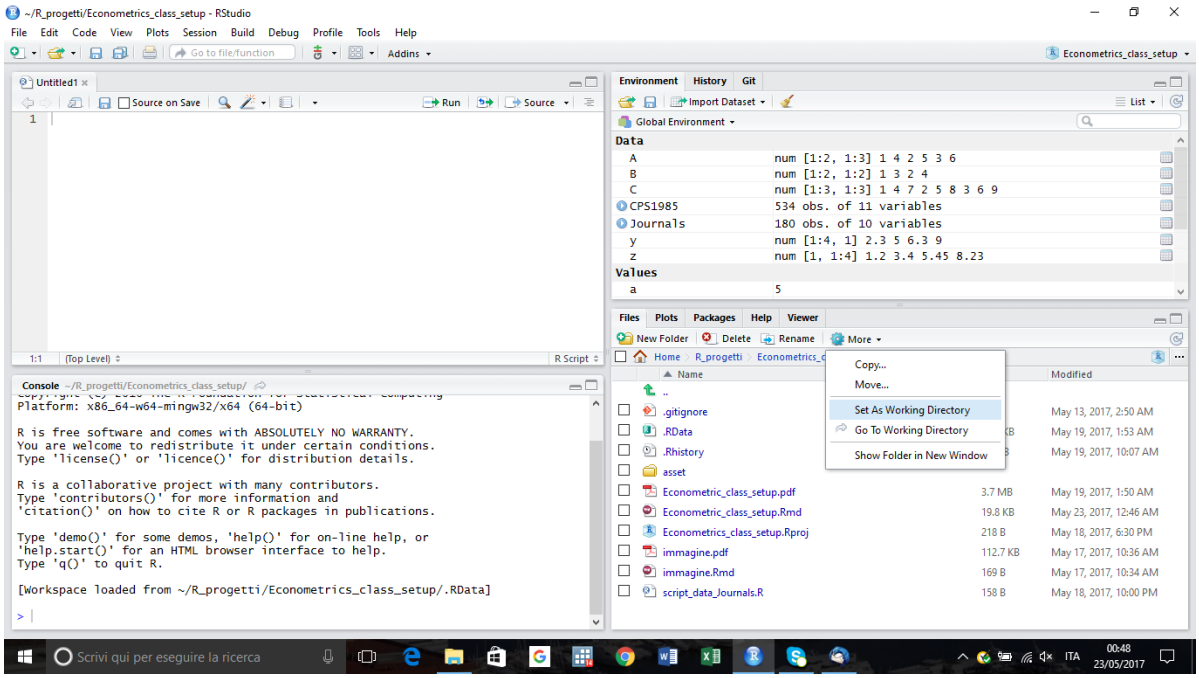


Figure 1.5: Set working directory **??**

1.2.3 Basic operations

Files in your working directory include:

- **.RData**: Holds the objects etc in your environment;
- **.RHistory**: Holds the history of what you typed in the console;
- Save on exit;
- **.RProfile**: Holds specific setup information for the working directory you are in.
- Did you create it? Check by writing in the console pane `file.exists("~/.Rprofile")`. If not, you can create it by writing in the console pane `file.edit("~/.Rprofile");`
- **Setting up R profile**: you can customize your environment for a project. We will discuss how to set up .RProfile in class.

1.2.3.1 Objects

Let's start giving some basic commands to R in the **console** pane. Let's treat R as a basic calculator making it compute `1 + 1`.

We see that R provides the answer *2* preceeded by `[1]`. Let's skip the meaning or these brackets for the moment.

Let's say that you think to use this operation `1 + 1` recursively during your work. R provides a fast and simple way to use this information without the need to give the same inputs again and again. How? By creating an **object**. Objects are values with structure. The name you assign to the **object** stores a value.

For example, let's call the object that stores our former operation *a*.[2]

Did you see what happened? The **object** that we created was just added to the **environment pane**. Let's sum `a + 2`. What will be the solution? (figure 1.6)

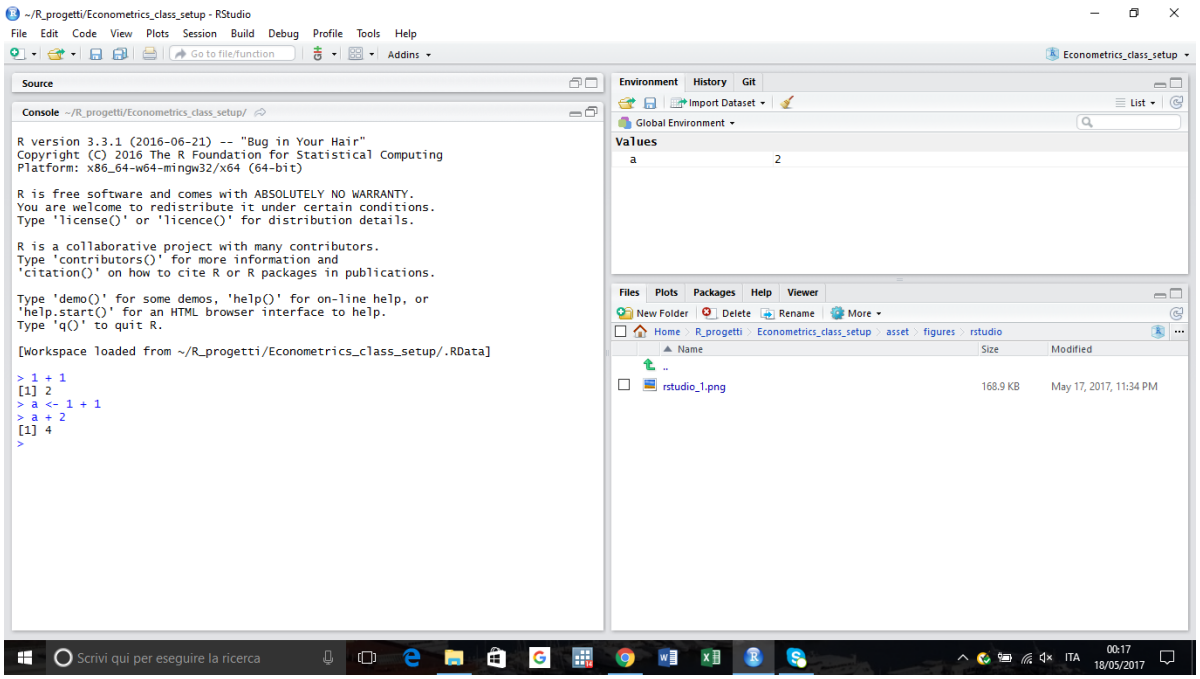


Figure 1.6: Objects

Be careful with **object** names. If you call another **object** *a* (for example `a = 5`), R overwrites your previous object called *a*. So in the environment pane you will find again only one **object** called *a*, but with the new assigned value. (figure 1.7)

Let's examine the meaning of the brackets. Create a new object, *b*, that stores all the integers from 0 *to* 100. We do this using the operator `:`. We see that a new object is added at the environment pane. Let's visualize it in the **console** typing the input *b*. The command returns the numbers in object *b* preceded by bracketed numbers `[]`.

[2]The less-than symbol, <, followed by minus symbol, -, is the assign operator. In most cases = has the same meaninig but the former symbol that looks like an arrow is what is used in programming language

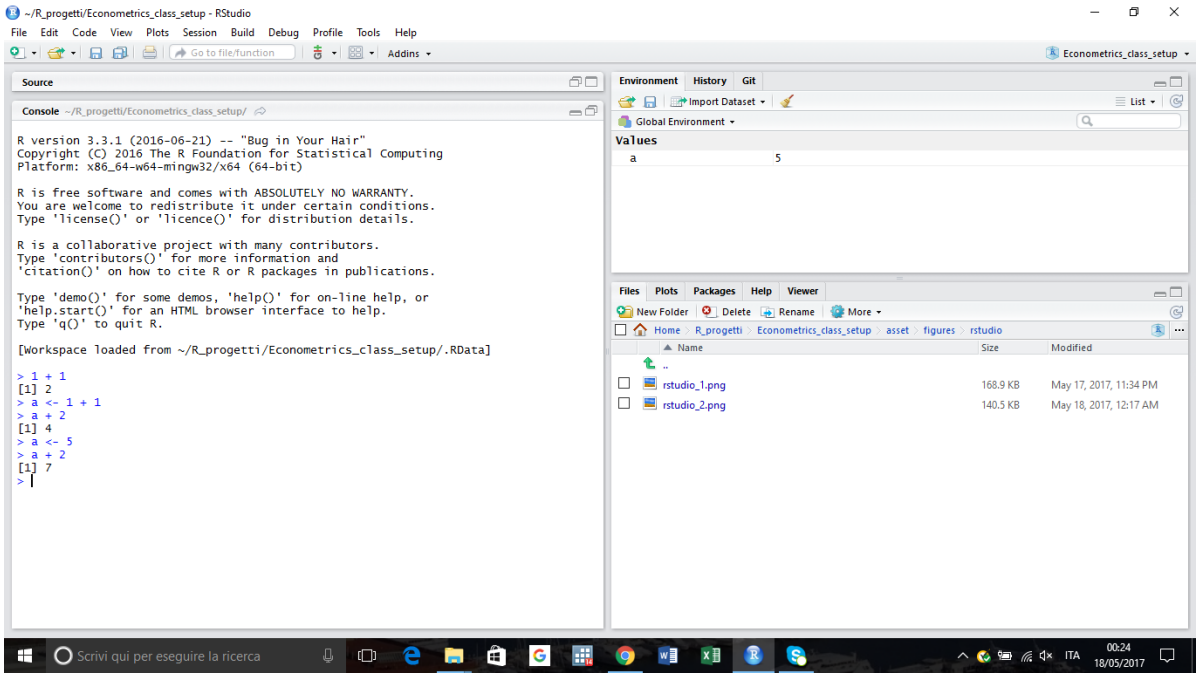


Figure 1.7: Overwriting objects

The bracketed numbers indicate the position of the first number in the row, within the object. This means that *0* is the first number of the series, while *18* is in the ninteenth position and so on (Fig. 1.8).

Now let's compute another operation. Let's multiply `5 * b` or we can also compute `a * b`. (figure 1.8)

1.2.3.2 Vector and Matrix

The former operation is a *scalar by vector* multiplication. Understanding vectors is key to understanding R. For details read **Applied Econometrics with R, Chapter 2**.[3]

Let's tries some basics. We compute a vector using the function `c()`.

```
x <- c(0.5, 3, 4.44, 7)
```

```
## [1] 0.50 3.00 4.44 7.00
```

Other functions to create vectors are `rbind()` and `cbind()`.

```
y <- rbind(2.3, 5, 6.3, 9)
```

```
##      [,1]
## [1,]  2.3
## [2,]  5.0
## [3,]  6.3
```

[3]Since now it is clear how to give command and what it causes we will use less figures in the rest of this document.

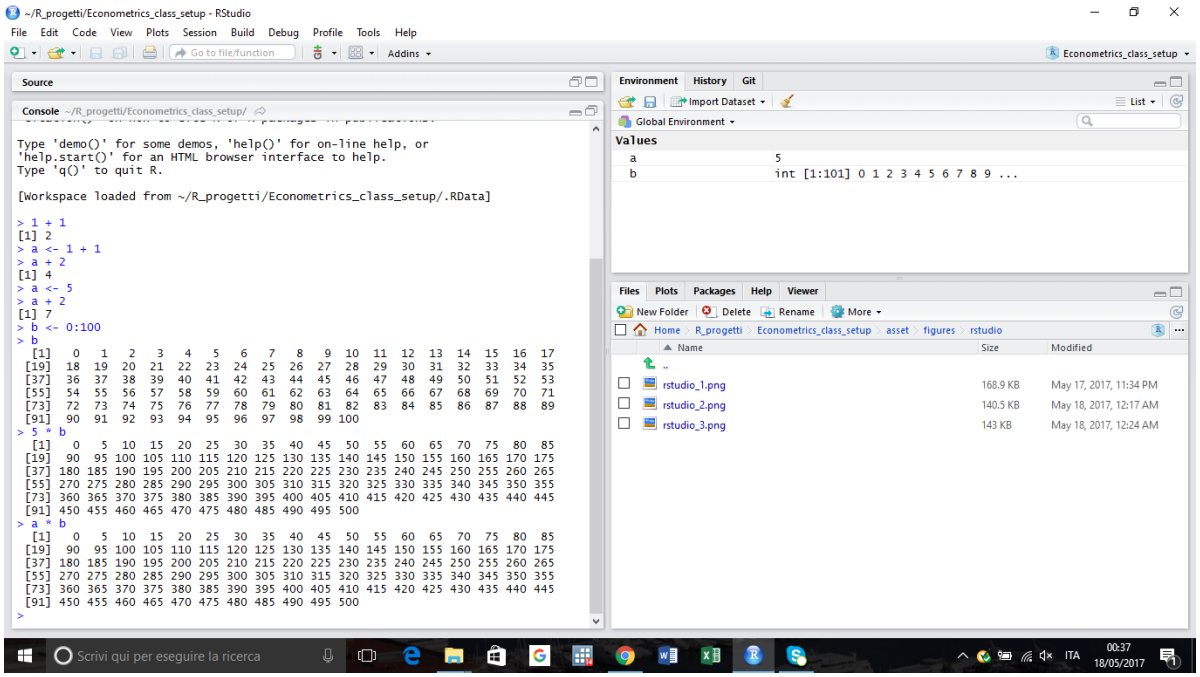


Figure 1.8: Scalar by vector

```
## [4,]  9.0
```

```
z <- cbind(1.2, 3.4, 5.45, 8.23)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]  1.2  3.4 5.45 8.23
```

The difference with the function `c()` is that this one does not compute a vector in mathematical sense but just creates a list of number.[4] We can see the difference by checking the class of the vectors we created.

```
class(x)
```

```
## [1] "numeric"
```

```
class(y)
```

```
## [1] "matrix"
```

```
class(z)
```

```
## [1] "matrix"
```

We can compute various operations. For example,

```
a + x
```

```
## [1]  5.50  8.00  9.44 12.00
```

[4]Not to be confused with function `list()`. For more info write `?list` in the console pane

```
a * y
```

```
##      [,1]
## [1,] 11.5
## [2,] 25.0
## [3,] 31.5
## [4,] 45.0
```

```
log(z)
```

```
##           [,1]     [,2]     [,3]     [,4]
## [1,] 0.1823216 1.223775 1.695616 2.107786
```

where the first operation returns a sum between a scalar and a vector, the second a multiplication between a scalar and a vector and the last operation is the logarithm of the vector.

Now let's build a matrix.

```
A <- matrix(c(
  1, 2, 3,
  4, 5, 6
))
```

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

How come we didn't get a matrix? Because if we don't indicate the number of row `nrow =` (and/or the number of column `ncol =`) we get a column vector.

```
A <- matrix(c(
  1, 2, 3,
  4, 5, 6
), nrow = 2)
```

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

Still note that the order is not the one we want. How to adjust? Adding *byrow = TRUE*

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

```
##      [,1] [,2] [,3]
```

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

Let's compute B as a square matrix.

```
B <- matrix(c(
  1, 2,
  3, 4
), nrow = 2, ncol = 2, byrow = TRUE)
```

```
##      [,1] [,2]
## [1,]    1    2
## [2,]    3    4
```

Let's now multiply `a * A`

```
a * A
```

```
##      [,1] [,2] [,3]
## [1,]    5   10   15
## [2,]   20   25   30
```

Do you remember what we said at beginning about objects? If we assign the same name R overwrites it. So, why now we can perform our multiplication? Let's have a look at the environment pane (figure 1.9)

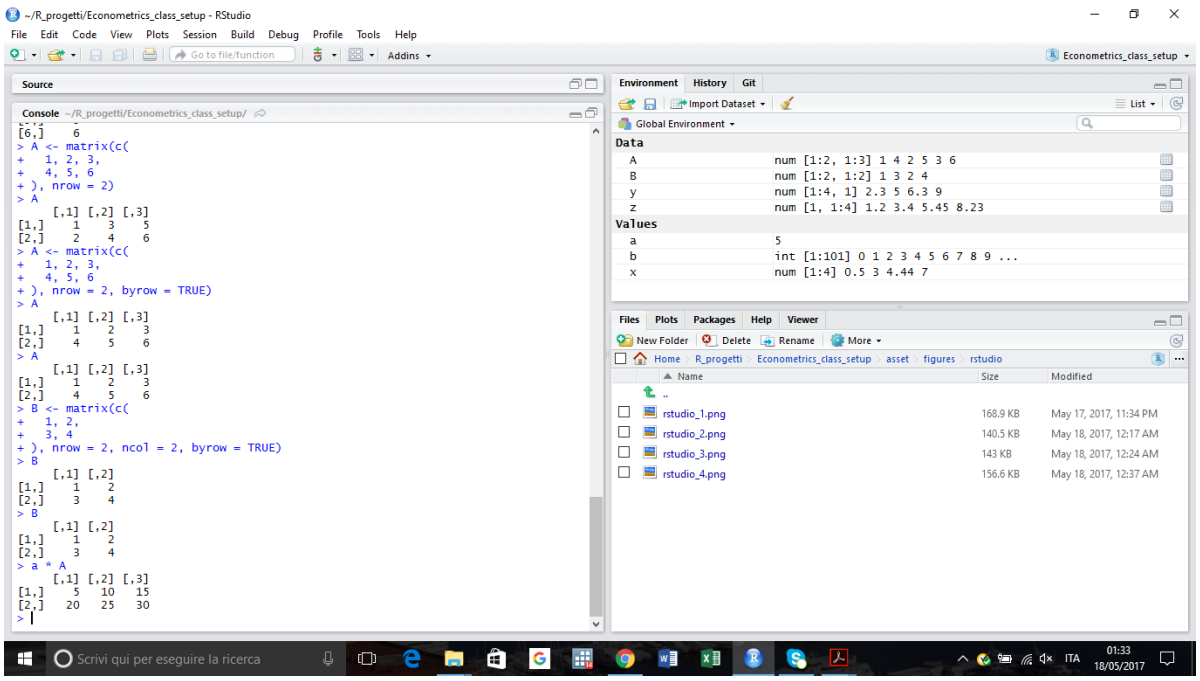


Figure 1.9: Case sensitive: a & A

You can see that both values a and A (but also b and B) are stored. This because R is case sensitive. This means that $a \neq A$ and $b \neq B$. Be careful to this when writing codes.

Have a look at the console pane. If you see how matrices were written you note a + symbol. This symbol means that the code you are writing continues on the following line. However, I anticipate that I didn't write the code in the console pane but in **R Markdown**. We will examine **R Script** and **R Markdown** later in this document.

Let's continue with matrix to see some operations. For example, the matrix multiplication is implemented by the following operator %*%.

```
B %*% A
```

```
##      [,1] [,2] [,3]
## [1,]    9   12   15
## [2,]   19   26   33
```

We can also compute the determinant

```
det(B)
```

```
## [1] -2
```

transpose

```
t(B)
```

```
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
```

inverse

```
solve(B)
```

```
##      [,1] [,2]
## [1,] -2.0  1.0
## [2,]  1.5 -0.5
```

eigenvalues and eigenvectors

```
eigen(B)
```

```
## $values
## [1]  5.3722813 -0.3722813
##
## $vectors
##            [,1]       [,2]
## [1,] -0.4159736 -0.8245648
## [2,] -0.9093767  0.5657675
```

and diagonal

```
diag(6)
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    1    0    0    0    0    0
## [2,]    0    1    0    0    0    0
## [3,]    0    0    1    0    0    0
## [4,]    0    0    0    1    0    0
```

```
## [5,]    0    0    0    0    1    0
## [6,]    0    0    0    0    0    1
```

Note that if you want 5 on the main diagonal

```
diag(5, 6)
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    5    0    0    0    0    0
## [2,]    0    5    0    0    0    0
## [3,]    0    0    5    0    0    0
## [4,]    0    0    0    5    0    0
## [5,]    0    0    0    0    5    0
## [6,]    0    0    0    0    0    5
```

Note that `rep()` replicates a number in first position how many times you write in the second position.

```
diag(c(1, rep(2, 3), 1))
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    0    0    0    0
## [2,]    0    2    0    0    0
## [3,]    0    0    2    0    0
## [4,]    0    0    0    2    0
## [5,]    0    0    0    0    1
```

1.2.3.3 Logical operators

R includes logical operators such as `<, >, <=, >=, == (for exact equality) and != (for inequality)`.

Pay attention to == operator.

Let's see an example.

```
3 == 1 + 1 + 1
```

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

But what about `0.3 == 0.1 + 0.1 + 0.1`? Let's try

```
0.3 == 0.1 + 0.1 + 0.1
```

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

Use function `all.equal()` to compare decimal numbers

```
all.equal(0.3, 0.1 + 0.1 + 0.1)
```

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

Moreover, if you want to tie two logical expression you can use *&* for logical "and" and | for logical "or".

Some examples

```
2 > 1
```

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

Let's take again our x vector and check if

```
x > 3 & x <= 7
```

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

1.2.3.4 Flow control

R allows to implement some actions only if some conditions are met or allows to implement the same action for a certain number of times. It provides control structures such as `if/else`, `for` and `while` loops. Read **Applied Econometrics with R, Chapter 2** for explanation.

Let's see here a simple example based on the previous calculation and compute

```
if(all.equal(0.3, 0.1 +0.1 +0.1)){
  print("you are right")
} else{
  print("you are wrong")
}
```

```
## [1] "you are right"
```

Let's print the even number of our vector b

```
for(i in b){
  if(i %% 2 == 0)
  print(i)
}
```

```
## [1] 0
## [1] 2
## [1] 4
## [1] 6
## [1] 8
## [1] 10
## [1] 12
## [1] 14
## [1] 16
## [1] 18
## [1] 20
## [1] 22
## [1] 24
## [1] 26
## [1] 28
## [1] 30
## [1] 32
## [1] 34
## [1] 36
## [1] 38
```

```
## [1] 40
## [1] 42
## [1] 44
## [1] 46
## [1] 48
## [1] 50
## [1] 52
## [1] 54
## [1] 56
## [1] 58
## [1] 60
## [1] 62
## [1] 64
## [1] 66
## [1] 68
## [1] 70
## [1] 72
## [1] 74
## [1] 76
## [1] 78
## [1] 80
## [1] 82
## [1] 84
## [1] 86
## [1] 88
## [1] 90
## [1] 92
## [1] 94
## [1] 96
## [1] 98
## [1] 100
```

Let's countdown 10, 9, 8, ... with `while`

```
d <- 10
while(d >= 0){
  cat(d, "\n")
  d <- d - 1
}
```

```
## 10
## 9
## 8
## 7
## 6
## 5
```

```
## 4
## 3
## 2
## 1
## 0
```

1.2.3.5 Functions

R allows to write functions that encode different commands avoiding to repeat the same commands.

R also provides a number of built-up functions. A diffuse family of R functions are the `apply()` family tha includes `sapply`, `tapply()`, `lapply()`.

For details, read **Applied Econometrics with R, Chapter 2** or, for example, type in the **console** pane `?apply`.["?" let to get information about a command in R (you can also use `help()`)]

Let's write our own function for example. Let's say we have a list of matrices but we want only to work with square matrices. We will implement an easy function, that we call `is_square_matrix`, that will give us the result *TRUE* if the matrix is square, *FALSE* otherwise.

```
is_square_matrix <- function(x){
  if(nrow(x) == ncol(x)){
    TRUE
  } else {
    FALSE
  }
}
```

Let's test it with our matrices A, B.

```
is_square_matrix(A)
```

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

```
is_square_matrix(B)
```

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

Given that the building a function is a more advanced topic, make reference to more advanced materials.

1.3 Getting help for R

- Function
- help, example, demo, help.search
- Internet
- www.r-project.org
- www.cran.r-project.org

- www.rseek.org
- Google
  - filetype: R regression - rebol
  - CRAN regression

1.4 Resources for R

1.4.1 CRAN Task Views

CRAN Task Views provide information about packages that can be used with R. Packages extend the functionality of R. They provide routines for various types of data manipulation, econometric models, etc. Packages are constantly being developed and update by R users. Three Task Views are relevant to this course:

- Econometrics: https://cran.r-project.org/web/views/Econometrics.html
- Time series: https://cran.r-project.org/web/views/TimeSeries.html
- Empirical finance: https://cran.r-project.org/web/views/Finance.html

1.4.2 Books

Several books have been published on using R for applied econometrics. We will use parts of the following books (provided as handouts in class).

- Heiss, F., "Using R for Introductory Econometrics". Companion site for "Using R for Introductory Econometrics"" (includes access to the book online for free) http://www.urfie.net/index.html
- Kleiber, C. and A. Zeileis, Applied Econometrics with R, Springer, 2008.

1.4.3 Online Guides for R

A comprehensive introduction to R can be found here: https://cran.r-project.org/doc/manuals/r-release/R-intro.pdf

A brief and easy introduction to R for econometrics can be found here: https://mondo.su.se/access/content/user/ma@su.se/Public/R_intro.pdf

Other R documentation and guides for various applications contributed by R users can be found here: https://cran.r-project.org/other-docs.html#english

1.4.4 R-Bloggers

A blog site containing many useful posts on using R. http://www.r-bloggers.com/

1.4.5 Search

Lots of information is posted online by R users including questions and answers. If you are stuck, search for information that will help you answer your queston.

2 PART 2

2.1 Packages

```
# to import database
library("utils")
library("readr")
library("readxl")

# to plot
library("ggplot2")
library("easyGgplot2")

# for Time Series
library("fBasics")
library("tseries")
library("zoo")
library("xts")
library("timeSeries")

# Data management
library("tidyr")

# for Date
library("lubridate")

# for creating tables
library("kableExtra")

# for editing and making websites in R Markdown
library("bookdown")
```

2.2 R Script & R Markdown

The problem in writing directly the commands in the console pane is that it is difficult to retrieve all the commands we wrote.

So if we want to have a trace of all the commands we write during our work is

better to use **R Script**. We open it clicking on the white square with a cross below File as in figure 2.1.

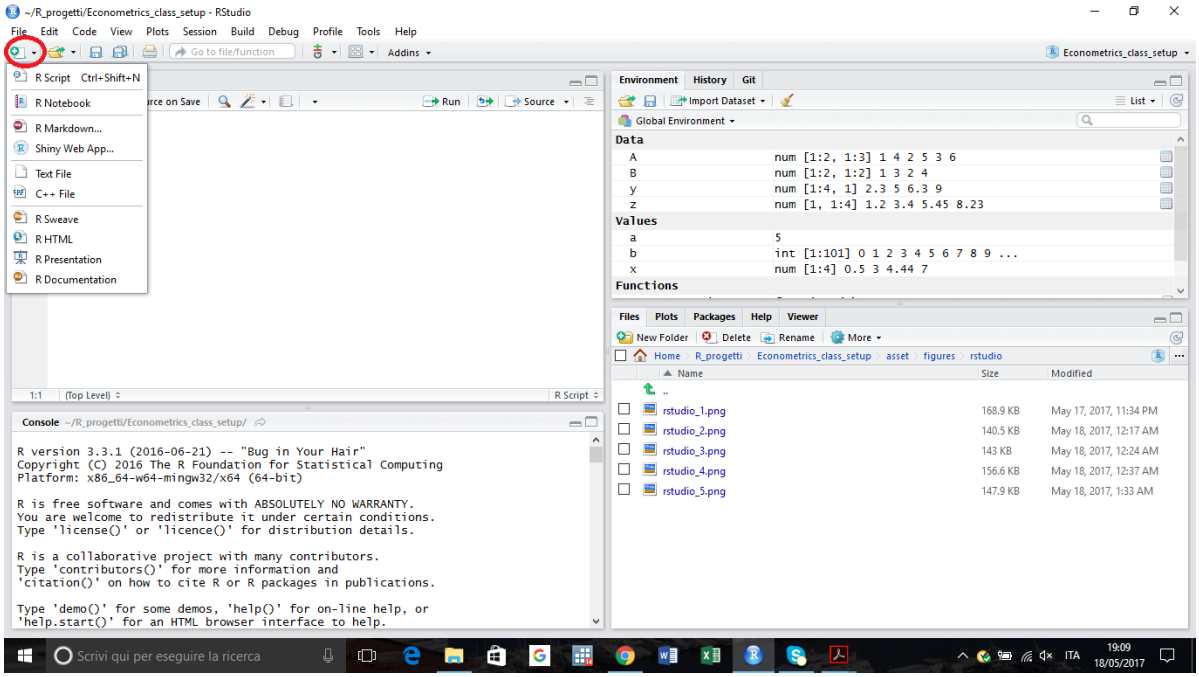


Figure 2.1: Open R Script & R Markdown

As we noted earlier in this section, in the console pane we saw commands written not on the same line but in different lines linked with a +. It was the case of *A* and *B* matrices (see figure 18).

Even if we can write in the console pane in the same way adding + to tell R that our command continues, we can write easily that command in R Script without the need to add + between lines. You can compare the two different writing styles in figure 2.2.[1]

Let's make some examples based on the resources of the textbook **Applied Econometrics with R, by Kleiber & Zeileis**.

To do this we need to load the package with the data.[2]

```
data("CPS1985", package = "AER")
```

Let's use a linear regression model, using the `lm()` function, to estimate the impact of experience on wage in the dataset *CPS1985*.

```
cps_lm <- lm(log(wage) ~ experience, data = CPS1985)

summary(cps_lm)
```

[1] Note that even if you can write all the code on the same line, it is convention to split the code in different lines. In this way it is clearer to the reader the main commands of the code

[2] You can install a package by `install.packages()`

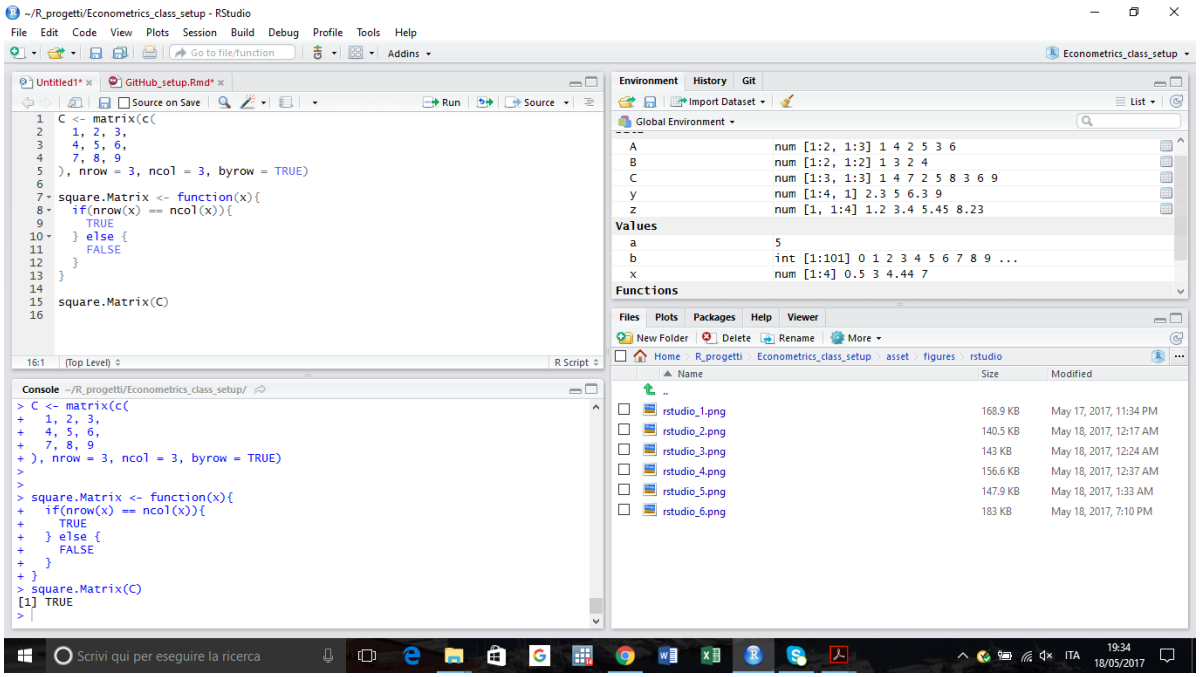


Figure 2.2: Different input in Console and R Script

The command `summary()` returns the usual summary of the coefficients (with estimates, standard errors, test statistics, and p values) as well as the associated R2, along with further information (figure 2.3).

Let's say now that we think that wage is influenced not only by experience but also by education.

So we modify our regression as follows

```
cps_lm <- lm(log(wage) ~ experience + education, data = CPS1985)

summary(cps_lm)
```

If we were probably editing our paper, thesis in Word, perhaps we found easily to copy the commands and the outcome and paste in Word. But now that we modified it, we should delete what we pasted and copy and paste the new outcome in Word.

Let's plot log-wage versus experience (figure 2.4).

Again we think to put it in our paper by copying or saving the image. But now we think perhaps that it would be better to make it fancier and add a title as follows.

```
plot(log(wage) ~ experience, data = CPS1985, pch = 1, col = "blue", main = "Scatterplot l
```

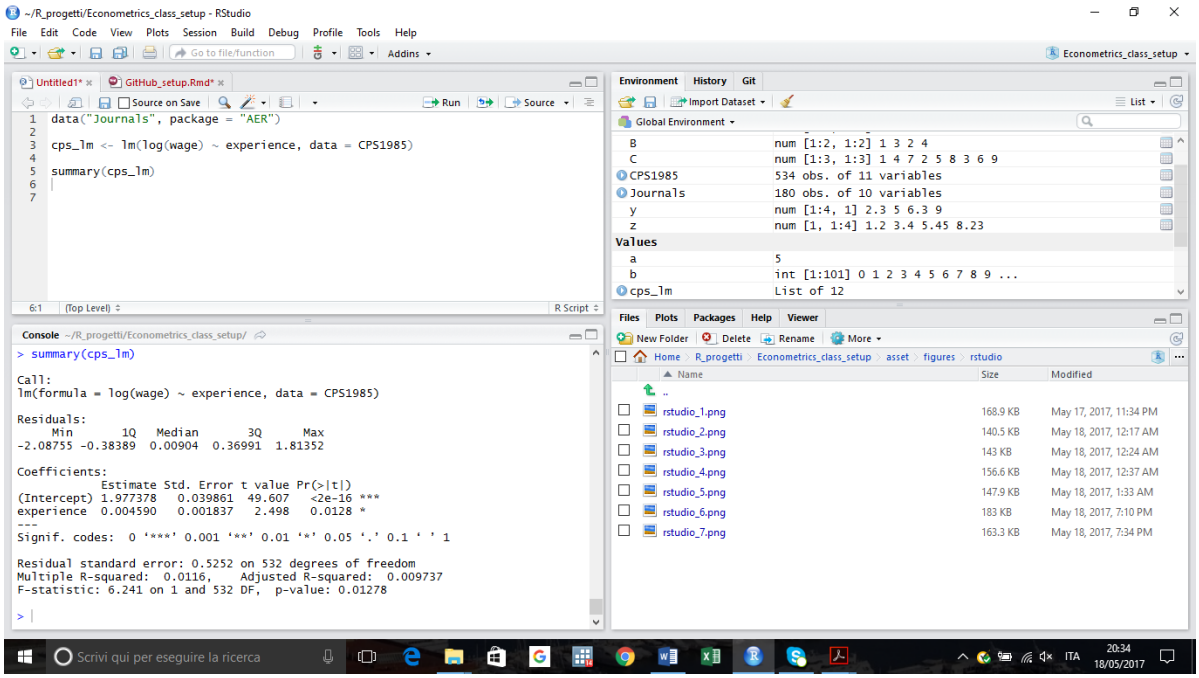


Figure 2.3: Summary of regression

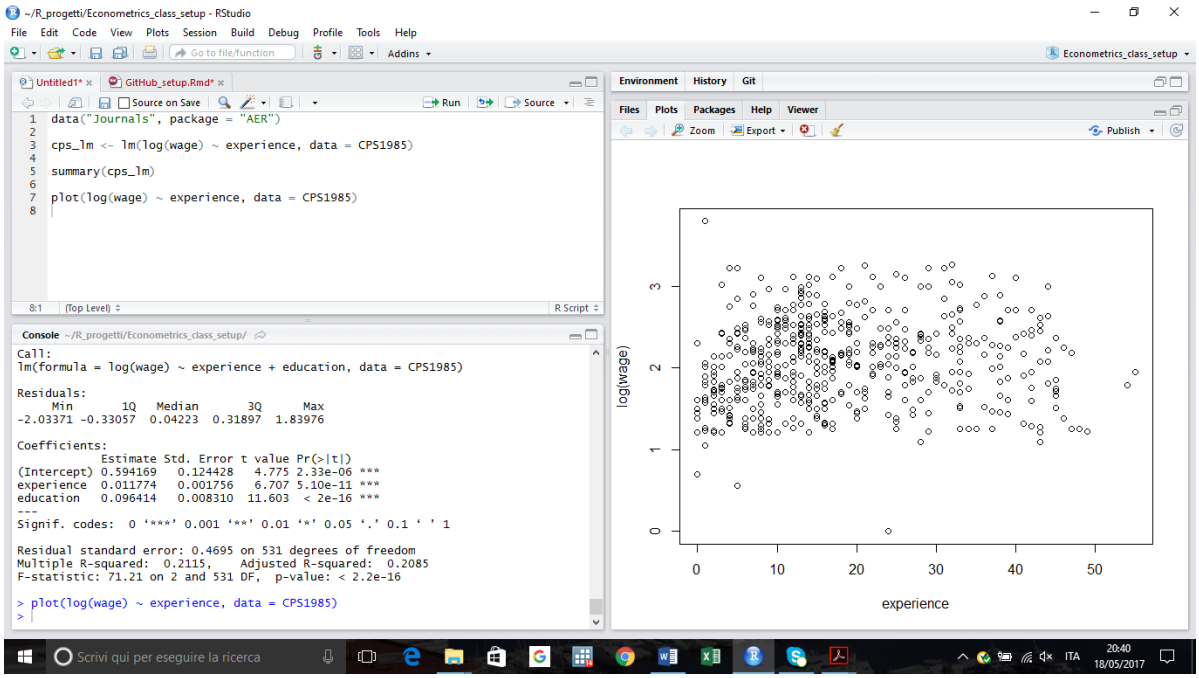
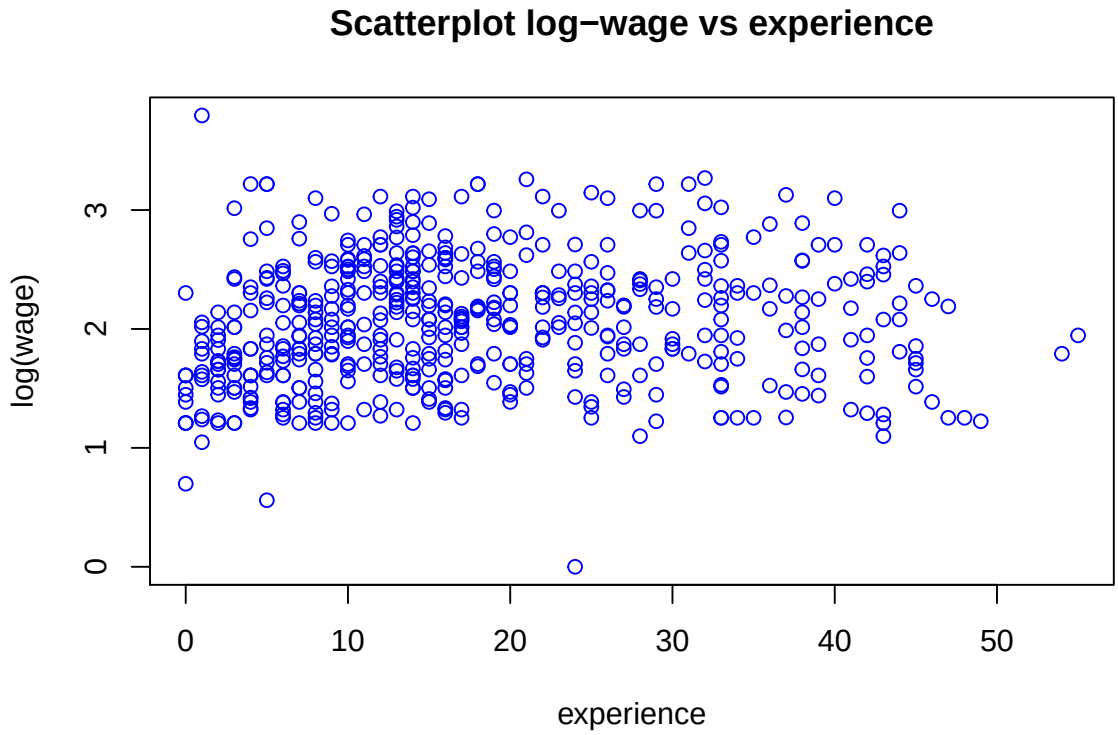


Figure 2.4: Scatterplot of log-wage versus experience



Again, we should change our output again.

In this case, **R Markdown** comes in handy. We open it as showed in figure 19. When we open it, R asks us what output we want (figure 2.5). Anyway this can be changed later.

A window like in figure 2.6 is opened.

After we opened **R Markdown**, it explains its basic characteristics. As you can see there is a grey space in the document

This is the chunk where you can write your code and run it by clicking on the arrow. If you click the arrow in the **{r cars}** example you will see something like this (figure 2.7)

As it is clear, the output is already in your document. What is the good thing? If you change mind about your code you need only to modify it in the chunk and run again. No need to rewrite, copy and paste somewhere else.

R Markdown provides various options for displaying the output. For example, you can prevent what you write in the code from being included in your paper.

To start working in **R Markdown** you can delete the examples. To insert by yourself the code chunk just click on *Insert* (figure 2.8). If you click the gear symbol in the code chunk you can have an idea of some options.

This document was edit in **R Markdown**. Let's see some examples of how we edited (figures 2.9 & 2.10).

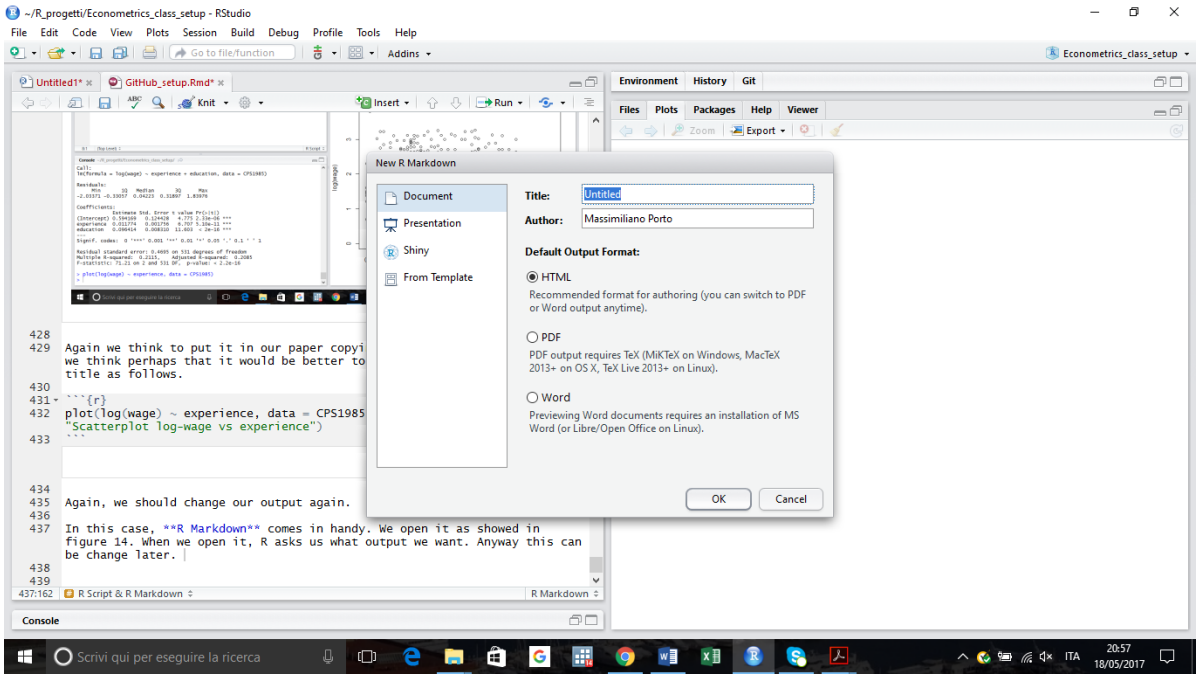


Figure 2.5: Open R Markdown

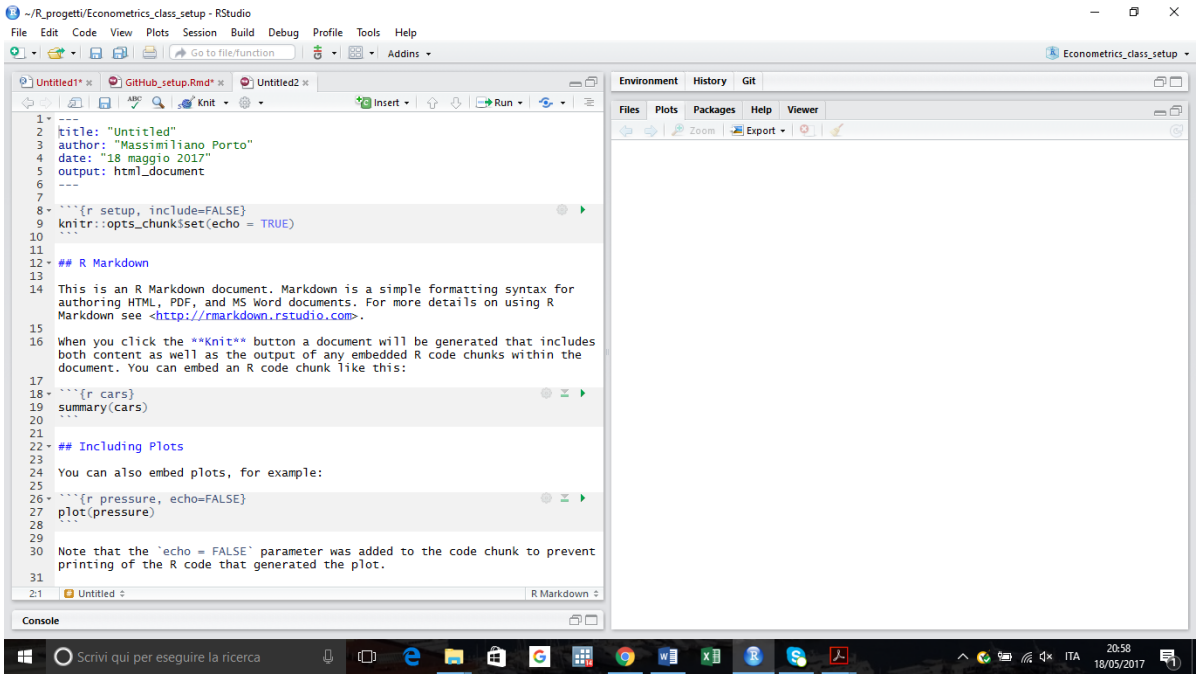


Figure 2.6: Markdown chunk

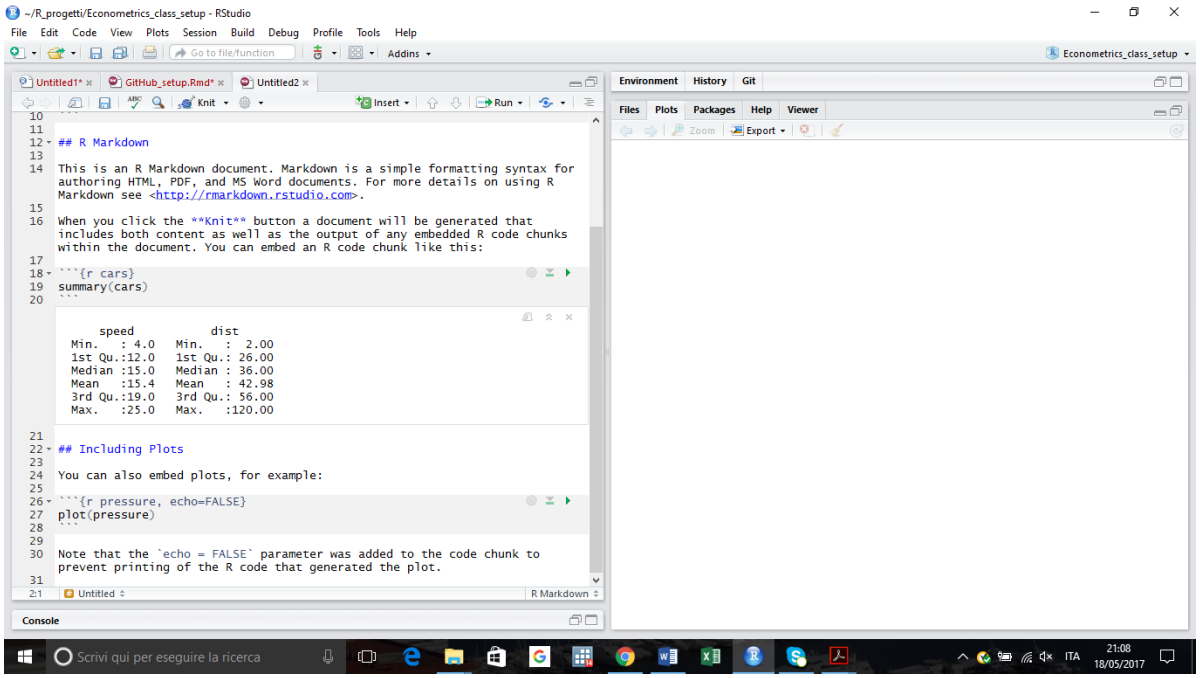


Figure 2.7: Run a chunk

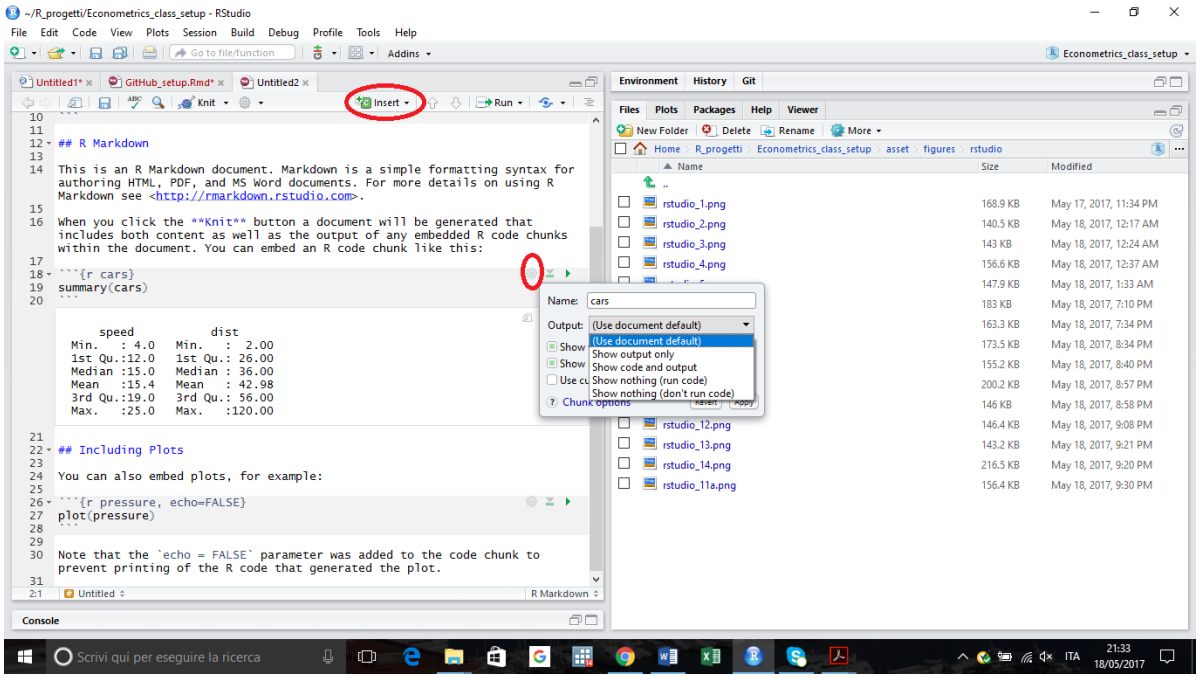


Figure 2.8: Example R Markdown 1

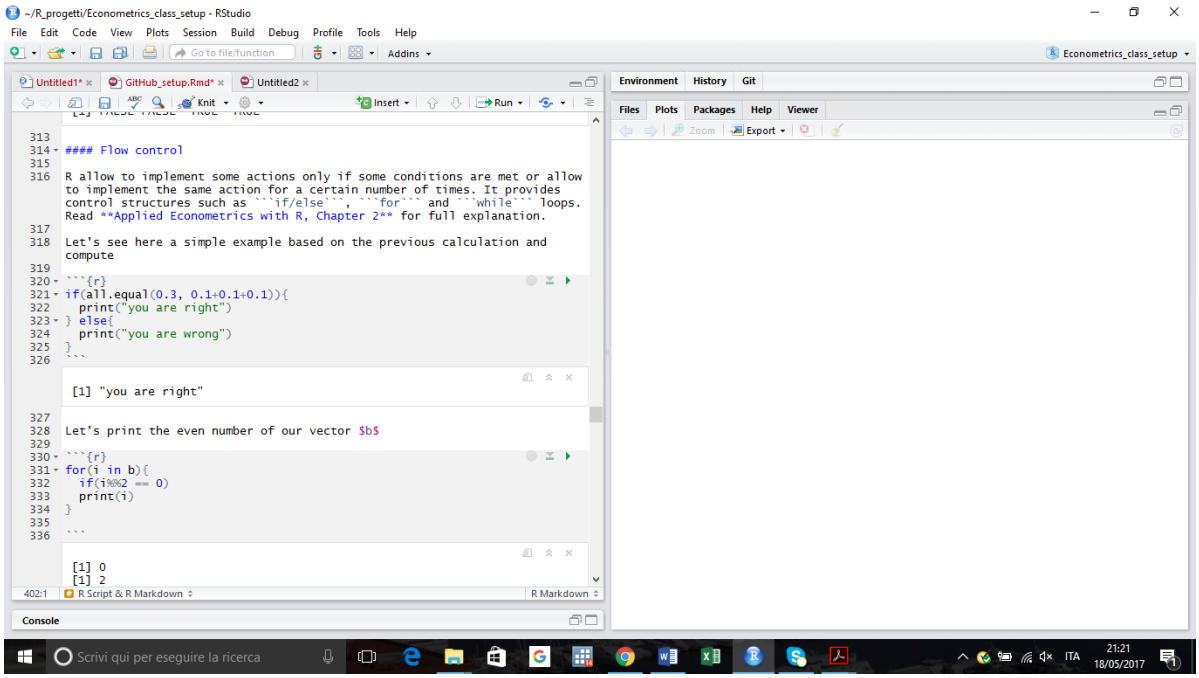


Figure 2.9: Example R Markdown 2

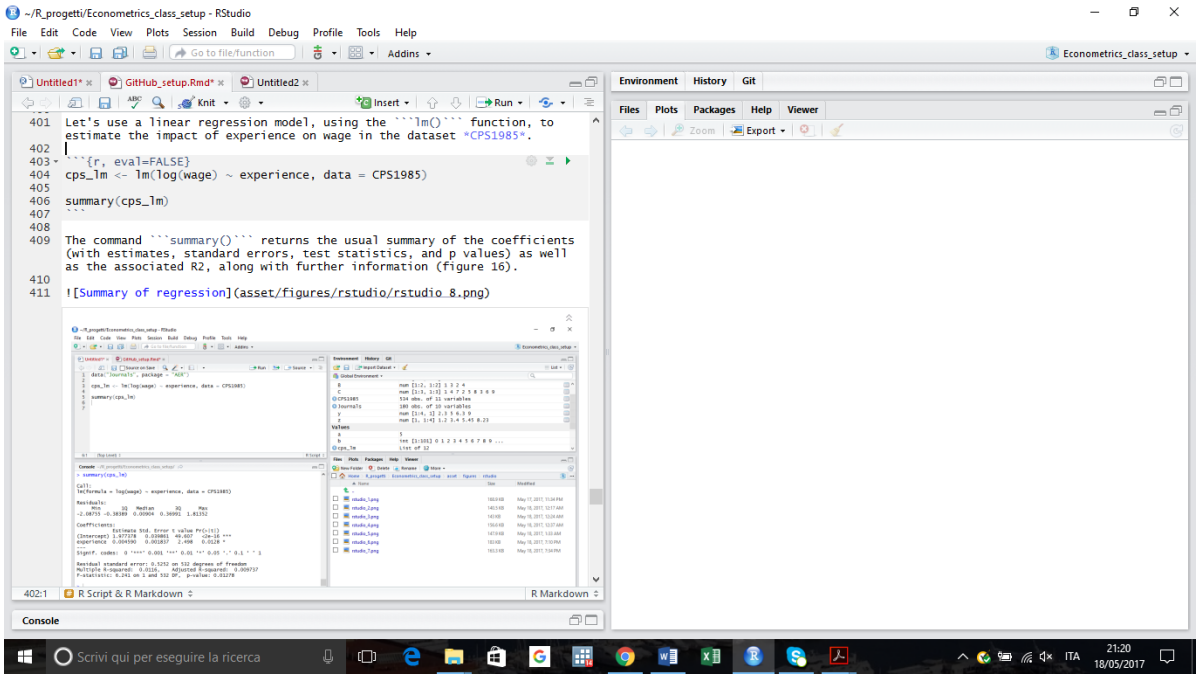


Figure 2.10: Example R Markdown 3

2.2.1 LaTeX

Now that we saw how to work in **R Markdown** let's convert the Rmd file in pdf by clicking on **knit** as in figure 2.11.

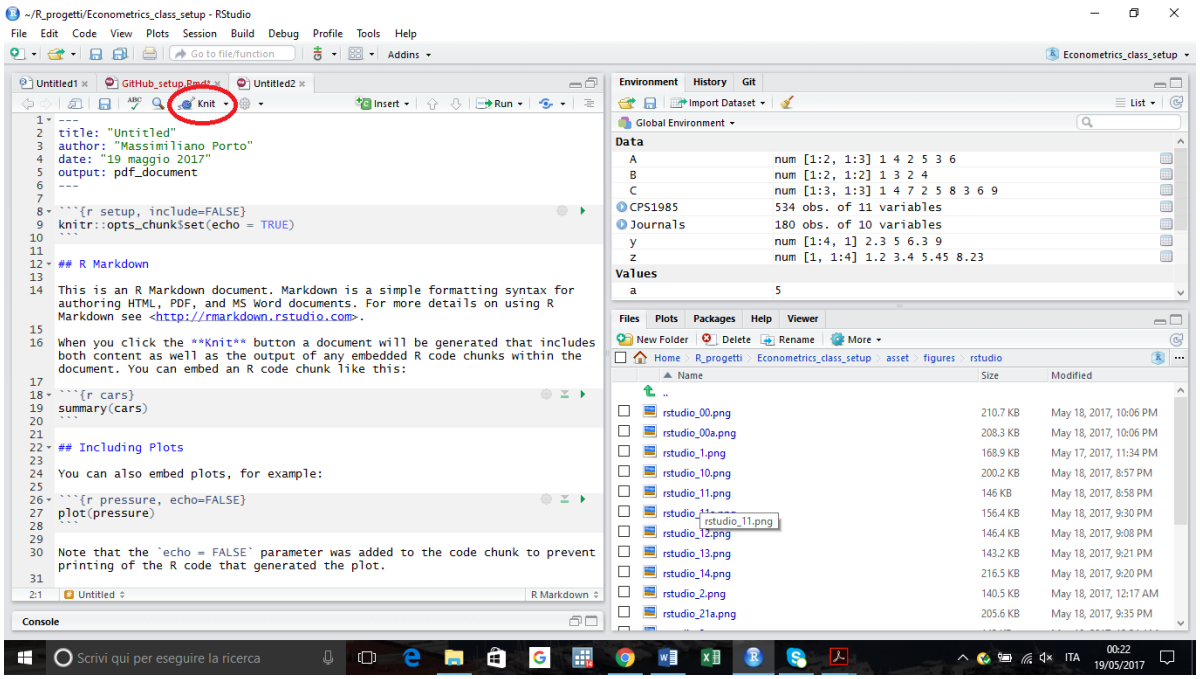


Figure 2.11: Knit

However, before successfully creating your pdf you need to install some packages:

- bookdown `install.packages("bookdown")`
- knitr `install.packages("knitr")`

Moreover, you need to install LaTeX system. There are different applications, for example:

- TeX Maker (http://www.xm1math.net/texmaker/)
- MiKTeX (https://miktex.org/download)
- TeX Live (https://www.tug.org/texlive/)
- MacTeX for macOS (https://www.tug.org/mactex/)

2.3 Data management

In this section we will see how to import a dataset in R and simple functions to manage it.

Before starting to work on a dataset, it is important to have a *double look* to the data. You can see your dataset with the function `View()`. You can use `head()` or `tail()` if you want to see only the first or the last entries.

Let's see an example. We will have a look to the dataset *Journals* in the package *AER*.

```
data("Journals", package = "AER")
View(Journals)
head(Journals)
```

```
##                                                    title
## APEL                     Asian-Pacific Economic Literature
## SAJoEH           South African Journal of Economic History
## CE                                 Computational Economics
## MEPiTE MOCT-MOST Economic Policy in Transitional Economics
## JoSE                             Journal of Socio-Economics
## LabEc                                     Labour Economics
##                     publisher society price pages charpp citations
## APEL                Blackwell      no   123   440   3822        21
## SAJoEH So Afr ec history assn      no    20   309   1782        22
## CE                     Kluwer      no   443   567   2924        22
## MEPiTE                 Kluwer      no   276   520   3234        22
## JoSE                  Elsevier      no   295   791   3024        24
## LabEc                 Elsevier      no   344   609   2967        24
##        foundingyear subs             field
## APEL           1986   14           General
## SAJoEH         1986   59  Economic History
## CE             1987   17       Specialized
## MEPiTE         1991    2      Area Studies
## JoSE           1972   96 Interdisciplinary
## LabEc          1994   15             Labor
```

```
tail(Journals)
```

```
##                                                title
## JASA    Journal of the American Statistical Association
## JoFi                                 Journal of Finance
## QJoE                     Quarterly Journal of Economics
## JoPolEc                    Journal of Political Economy
## Ecnmt                                     Econometrica
## AER                           American Economic Review
##                     publisher society price pages charpp citations
## JASA     Am. Statistical Assn     yes   310  1260   5664      2800
## JoFi          Am. Finance Assn     yes   226  2272   3036      3791
## QJoE                MIT press      no   148  1467   2184      4138
## JoPolEc Univ of Chicago Press      no   159  1669   2640      6697
## Ecnmt               Blackwell     yes   178  1482   2992      7943
## AER                Am Ec Assn     yes    47  1867   3900      8999
##         foundingyear subs        field
```

```
## JASA            1971  487 Econometrics
## JoFi            1945  799      Finance
## QJoE            1886  660      General
## JoPolEc         1892  737      General
## Ecnmt           1932  346      General
## AER             1911 1098      General
```

2.3.1 Class and structure

The second look we were talking about is an introspective look to the data through functions `class()` and `str()`.

Through these two functions we can have a better understanding of the the type of object we are working with. This is very important because some functions simply don't work with some objects. In this case we have to change their class. We will see some examples later.

Now let's see some examples of `class()`. This functions can be applied to any object. For example, let's check the class of some objects we created earlier

```
class(a)
```

```
## [1] "numeric"
```

```
class(A)
```

```
## [1] "matrix"
```

We have a *numeric* and *matrix* class. To put it easily, we have a "number" and a "matrix" object.

But observe the following

```
c <- "3"
class(c)
```

```
## [1] "character"
```

It has a *character* class. We see that it is a number, a 3. Let's try what happen if we add `a + c`. We could expext a 8. But

```
a + c
```

we get an error. Again to put it simple, even if the object c stores a 3 it has no *numeric* class, i.e. it is not a number.

Since we work with dozens of variables and thousands of observations it is clear that it is fundamental to know what objects we are working with.

Let's go back to dataset. Let's check class for Journals.

```
class(Journals)
```

```
## [1] "data.frame"
```

We have a `data.frame` objects that share many of the properties of matrices and lists.

Let's see what information provide the `str()` function.

```
str(Journals)
```

```
## 'data.frame':    180 obs. of  10 variables:
##  $ title       : chr  "Asian-Pacific Economic Literature" "South African Journ
##  $ publisher   : Factor w/ 52 levels "ANU Press","Academic Press",..: 11 45 28
##  $ society     : Factor w/ 2 levels "no","yes": 1 1 1 1 1 1 1 1 1 1 ...
##  $ price       : int  123 20 443 276 295 344 90 242 226 262 ...
##  $ pages       : int  440 309 567 520 791 609 602 665 243 386 ...
##  $ charpp      : int  3822 1782 2924 3234 3024 2967 3185 2688 3010 2501 ...
##  $ citations   : int  21 22 22 22 24 24 24 27 28 30 ...
##  $ foundingyear: int  1986 1986 1987 1991 1972 1994 1995 1968 1987 1949 ...
##  $ subs        : int  14 59 17 2 96 15 14 202 46 46 ...
##  $ field       : Factor w/ 24 levels "General","Economic History",..: 1 2 3 4
```

It provides the complete structure of the object. We can also select only one variable of the object. For example

```
str(Journals$price)
```

```
##  int [1:180] 123 20 443 276 295 344 90 242 226 262 ...
```

```
str(Journals$society)
```

```
##  Factor w/ 2 levels "no","yes": 1 1 1 1 1 1 1 1 1 1 ...
```

We see that price is recorded as `int`, i.e. integer, while society as `factor`, i.e. a category type.

Let's see some other examples with database available on R.

```
data("AirPassengers")
class(AirPassengers)
```

```
## [1] "ts"
```

```
data("EuStockMarkets")
class(EuStockMarkets)
```

```
## [1] "mts"    "ts"     "matrix"
```

`AirPassengers` returns a `ts` object, i.e. a time series object, while the class for `EuStockMarkets` says that the object is a multiple time series object.

You can encounter other class types. We will see others later.

For the moment we checked the class of objects that already exist and are available on R. What about the class of the dataset we built?

First let's see how to import it in R.

2.3.2 Import dataset

I downloaded Purchasing Power Parity (PPP) data from OECD database. To replicate download from OECD-ilibrary the PPP for GDP, year frequency from 1999, for Argentina, Australia, Brazil, Canada, Chile, China, Colombia, Czech Rep., Denmark, EU19, Great Britain, Hungary, India, Indonesia, Israel, Japan, Korea, Norway, New

Zealand, Poland, Russia, South Africa, Sweden, Switzerland, Turkey or just take it from my GitHub page https://github.com/massimilianoj/R_beginners.

```
ppp <- read.table("~/R_progetti/Portfolio_management_pmwr/currency_model_dataset/ppp.csv"
    header = TRUE, sep = ",")
```

I used `read.table()` to import it (package "utils"). Note that

- `header = TRUE` means that the first row of the database contains the title of our columns;
- because my file is a csv file (comma separated value) I need to indicate the kind of separator of the file as `sep = ","`.

There are other functions you can use to import a dataset. For example

```
ppp2 <- read.csv("~/R_progetti/Portfolio_management_pmwr/currency_model_dataset/ppp.csv",
    header = TRUE, sep = ",")
```

Another way is to click on **Import Dataset** button as in figure 2.12

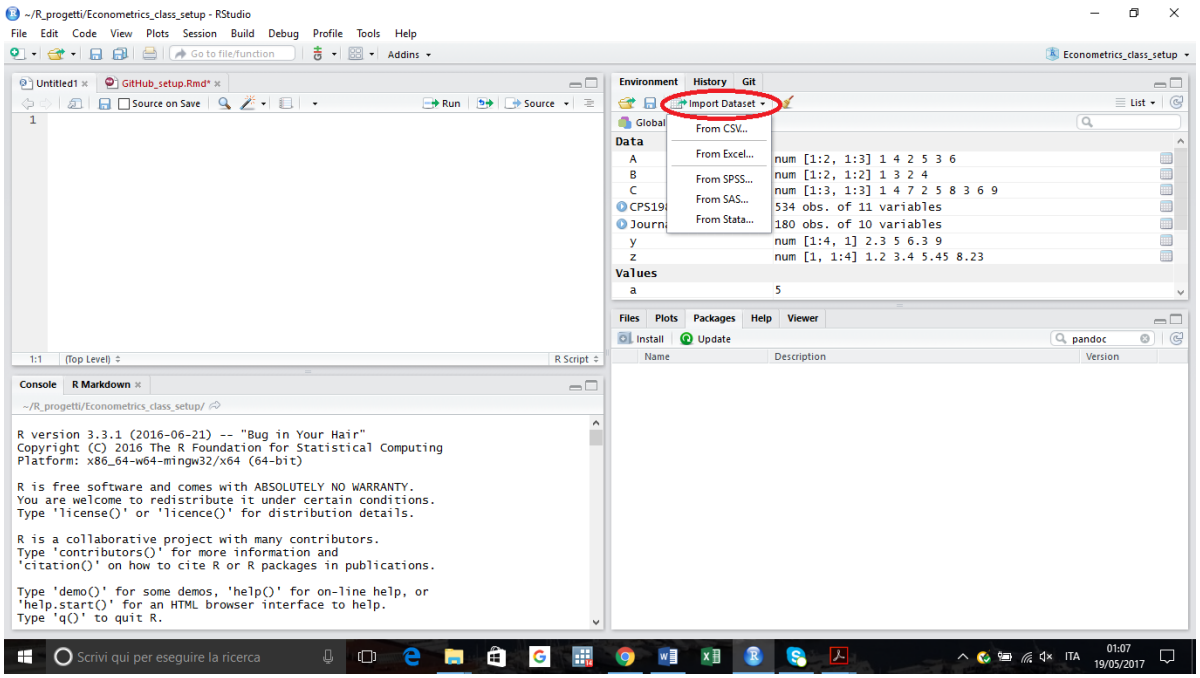


Figure 2.12: Import Dataset

Let's see how to import from a **CVS** file (figures 2.13 & 2.14)

You can see from figure 2.14 that I used another library, `readr` and another function, `read_csv()`. I named this dataset ppp3.

The three dataset, at first sight, look the same. Let's check class and structure.

```
class(ppp)
```

```
## [1] "data.frame"
```

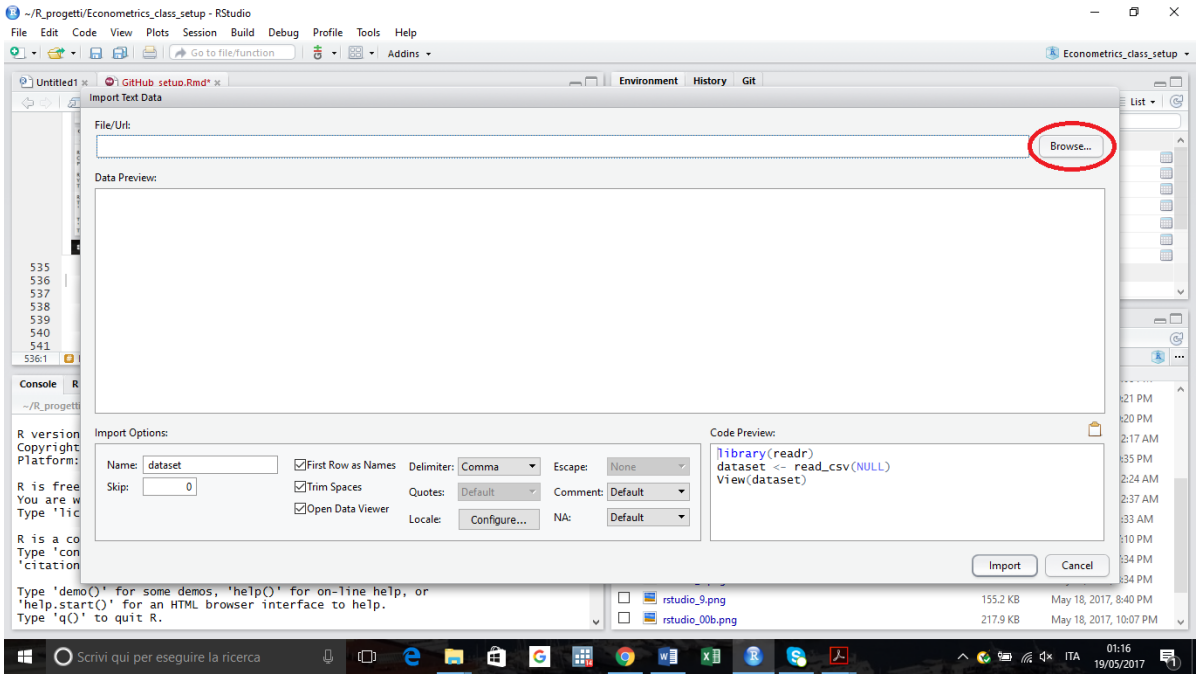


Figure 2.13: Import from CVS file (1)

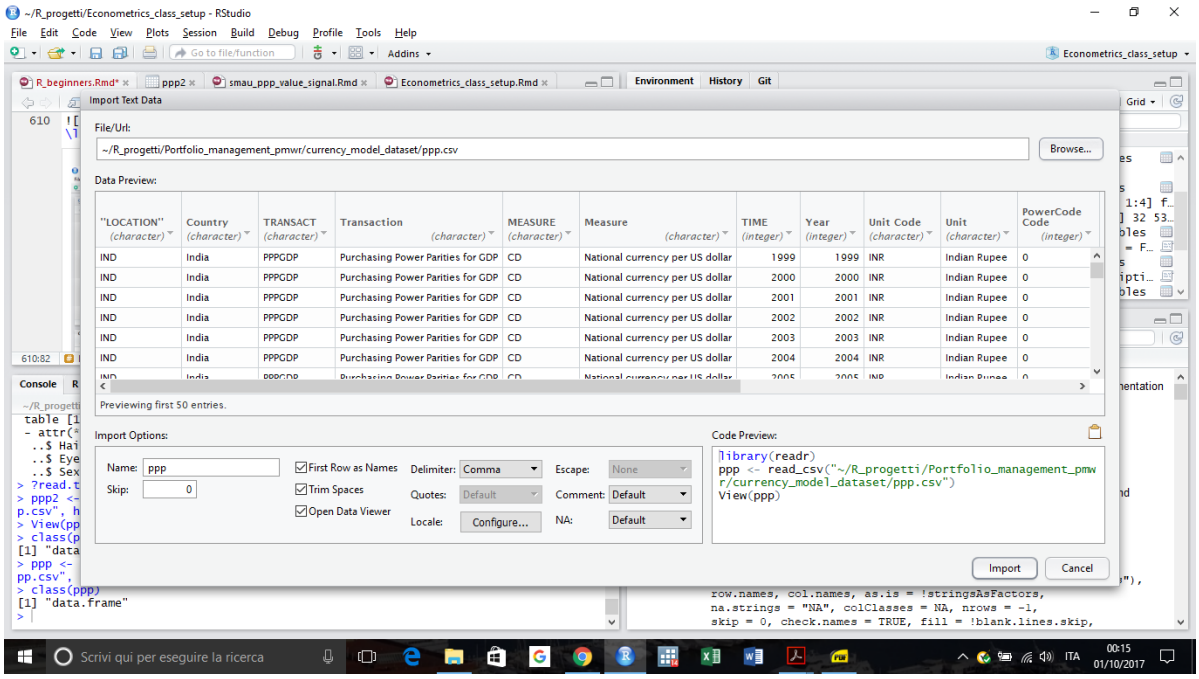


Figure 2.14: Import from a CVS file (2)

```
class(ppp2)
```

```
## [1] "data.frame"
```

```
class(ppp3)
```

```
## [1] "tbl_df"     "tbl"        "data.frame"
```

The first two databases return a `data.frame` object, while the third one returns `"tbl_df"  "tbl"  "data.frame"` that is a tibble data.frame. For more info about tibble, write `?tibble` in the console pane.

From here onwards the example will be provided with ppp database.

2.3.3 Simple operations on dataset

2.3.3.1 Select and rename columns

First, let's select the columns of interest and rename them.

```
# select column of interest
ppp <- ppp[, c(1, 8, 15)]
colnames(ppp) <- c("iso_code", "year", "ppp")
```

In the first expressione we are saying to R: "select all the rows and columns 1, 8 and 15".[3] In the second expression we renamed the titles of the columns of our database.

2.3.3.2 Transform database long-wide

Have a look again at the database (Fig. 2.15).

You see that we have a "long" format, while we want it wide, with each country on a different column with its ppp value per year. I will use `spread()` function (`tidyr` package) to rearrange it.[4]

```
# transform the dataset wide
ppp_w <- spread(ppp, iso_code, ppp)
```

Let's see the *new* database.

It is what we wanted. But what did we do? In the function `spread` - the first input is the database we use; - the second input is the key, i.e. the column that will be spread as column title; - the third input is the value, i.e. what we want to put as value under the new columns we created.

In general, a good way to see briefly the structure of a function is to use the function `args()`. For example

[3] If we had written `ppp <- ppp[c(1, 8, 15), ]` we would have said to R to select only rows 1, 8 and 15 and all the columns.

[4] `#` is used to write comments in the chunck; what follows will not be run by R.

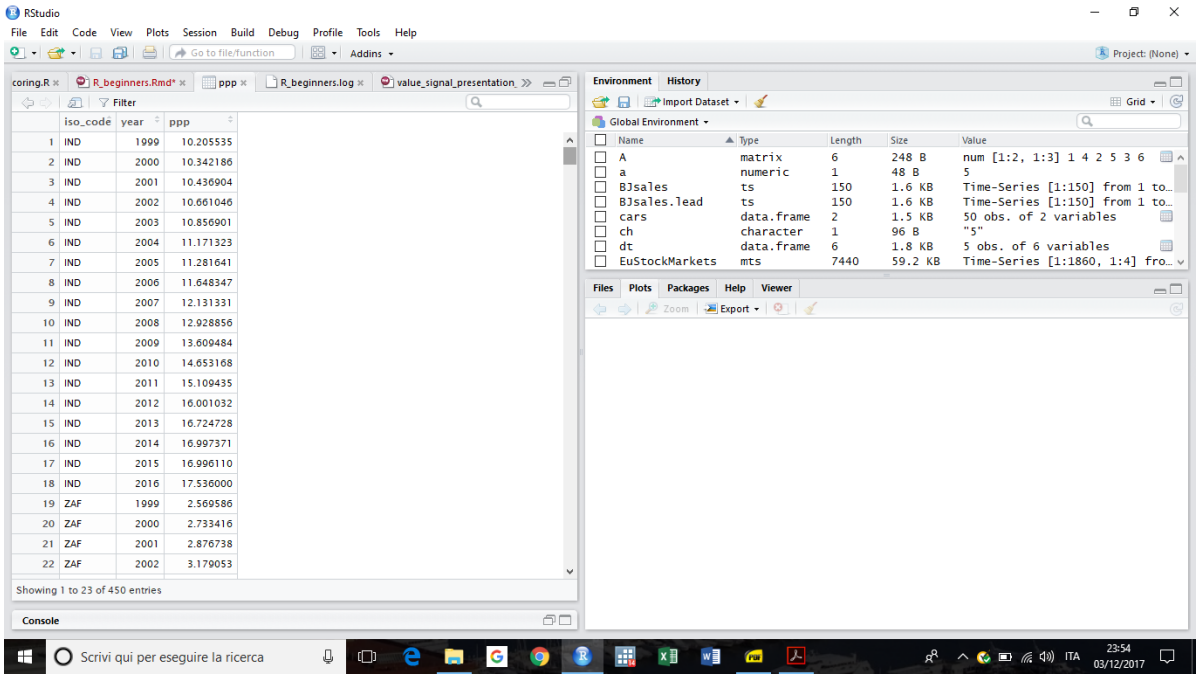


Figure 2.15: Long database

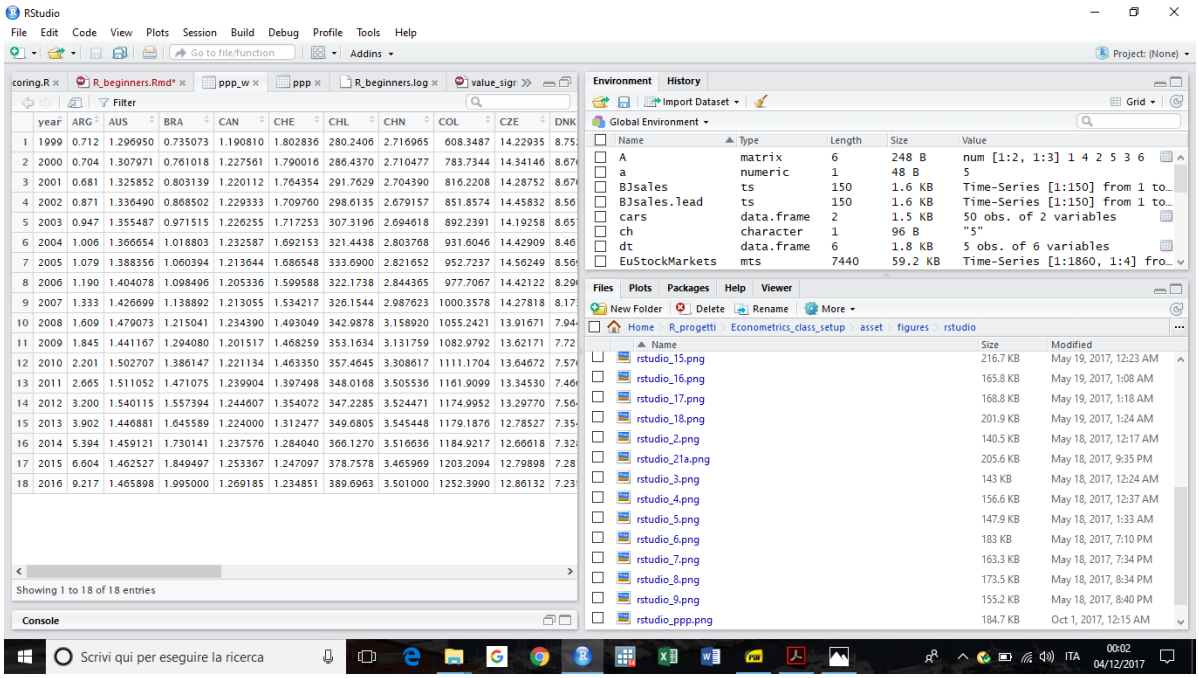


Figure 2.16: Wide database

```
args(spread)
```

```
## function (data, key, value, fill = NA, convert = FALSE, drop = TRUE,
##     sep = NULL)
## NULL
```

2.3.3.3 Rename single columns and drop variables

Next simple operations are to rename a single column and drop some columns. For example, we want to rename only the column for the eurozone and drop ARG, AUS and NZL from the database. Note that we used a - before the corresponding column number to drop it.

```
# rename column for the eurozone EMU
colnames(ppp_w)[which(colnames(ppp_w) == "EA19")] <- "EMU"

# drop ARG, AUS, NZL
ppp_w <- ppp_w[c(-2, -3, -21)]
```

What about if we want to go from wide to long? Let's do it

```
ppp_l <- gather(ppp_w, iso_code, ppp, BRA:ZAF)
```

Intuitively, to go back we need a column that will contain all the countries, `iso_code` and another column, `ppp`, that will contain all the ppp values per country. `BRA:ZAF` is the range of column that could be selected also one by one by using `c()`.

2.3.3.4 Operations only on some columns of the database

Suppose that we want to make our databse relative to EMU ppp. We want to divide all the columns by EMU but not year.

```
# create a new object ppp_to_EMU equal to ppp_w We implement this step
# because we don't want the following modification to affect our ppp_w for
# following examples
ppp_to_EMU <- ppp_w
ppp_to_EMU[, c(2:23)] <- ppp_to_EMU[, c(2:23)]/ppp_to_EMU$EMU
```

You can see that the year column is not affected by the operation (Fig. 2.17)

2.3.3.5 Subset the database

Still working on `ppp_to_EMU`. Let's suppose we are interested only in the period between 2007 and 2012. We subset the database using `subset()` (Fig. 2.18). Note that you can subset a database using other functions such as `window()` or `fiter()`

```
ppp_EMU_subset <- subset(ppp_to_EMU, year >= "2007" & year <= "2012")
```

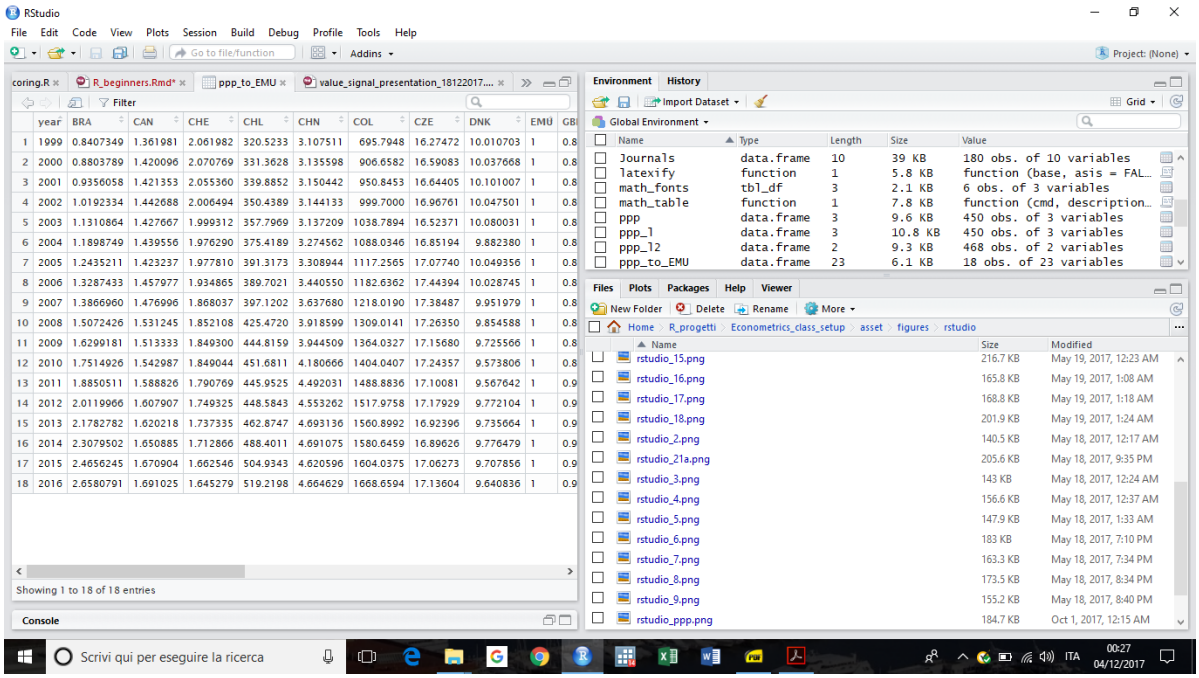


Figure 2.17: Wide database

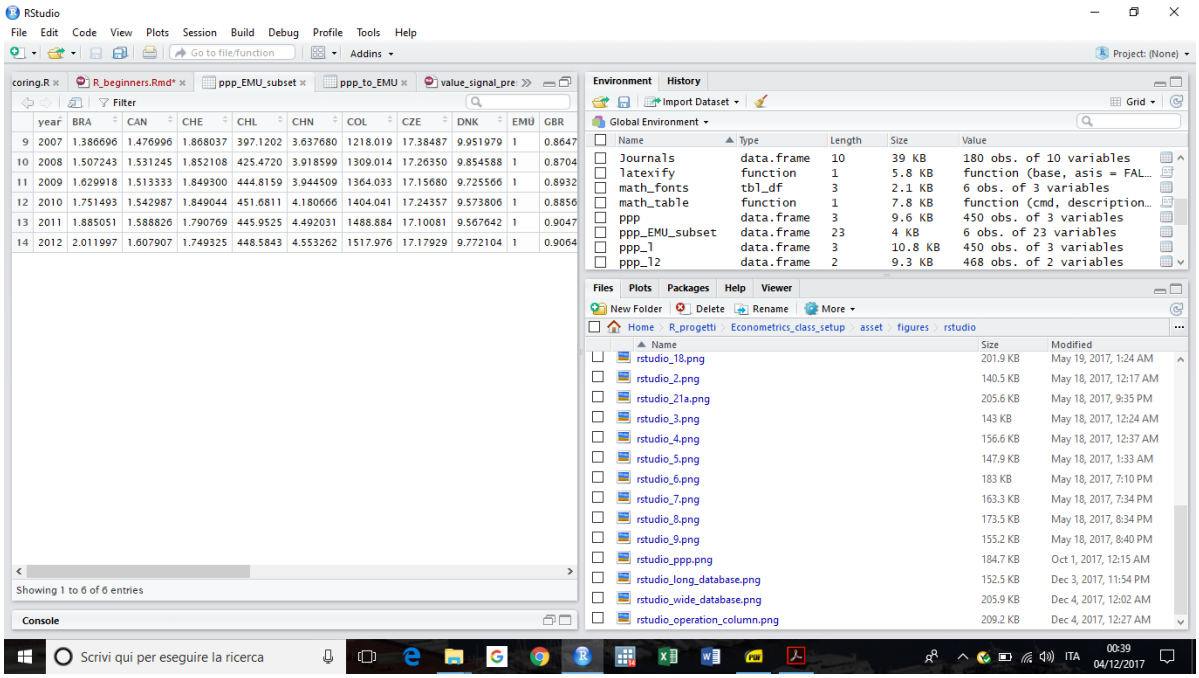


Figure 2.18: Wide database

2.3.3.6 Merge two databases

First let's import gdp from OECD and following the previous steps to arrange it.

```
gdp <- read.table("~/R_progetti/Portfolio_management_pmwr/currency_model_dataset/gdp.csv"
    header = TRUE, sep = ",")
```

```
gdp <- gdp[, c(1, 8, 15)]
colnames(gdp) <- c("iso_code", "year", "gdp")
gdp_w <- spread(gdp, iso_code, gdp)
colnames(gdp_w)[which(colnames(gdp_w) == "EA19")] <- "EMU"
gdp_w <- gdp_w[, c(1, 5, 6, 7, 8, 9, 10, 11, 13, 14, 19, 21, 22, 23, 26, 28,
    29, 34, 36, 38, 41, 42, 44)]
gdp_w <- subset(gdp_w, year >= "1999" & year <= "2016")
gdp_l <- gather(gdp_w, iso_code, gdp, BRA:ZAF)
```

Now we have two databases, `ppp_l` and `gdp_l`, that share same year and iso_code. We can merge them as follows (Fig. 2.19:

```
merge_data <- merge(ppp_l, gdp_l, by = c("year", "iso_code"))
```

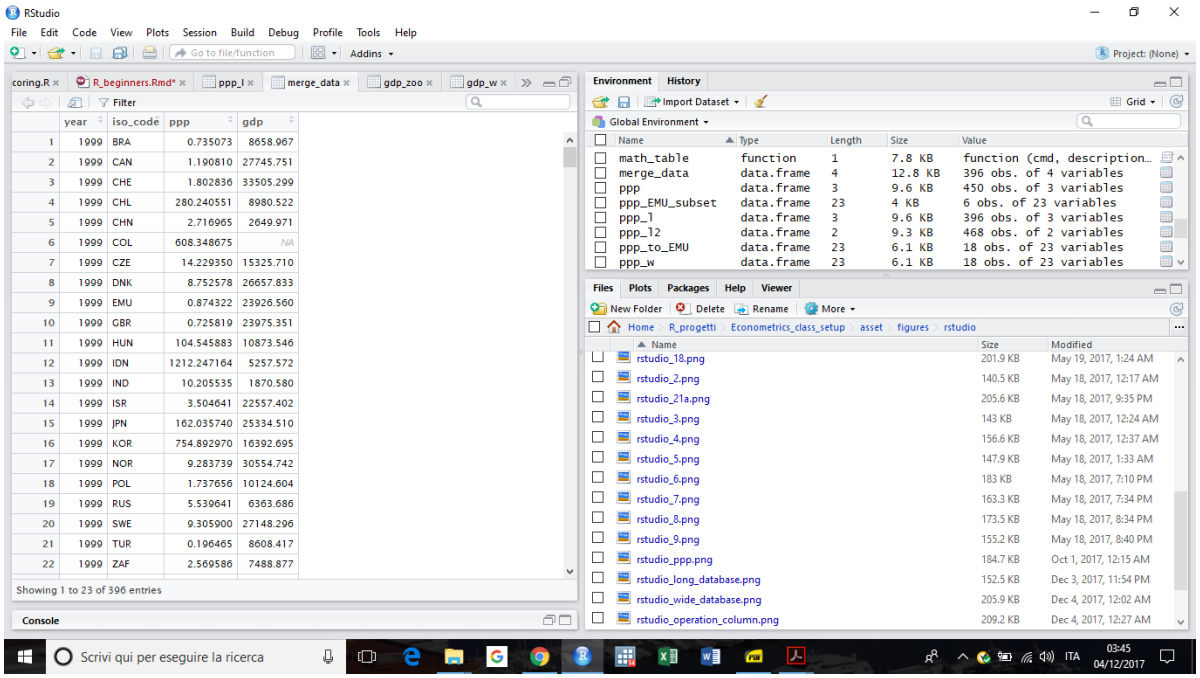


Figure 2.19: Merged database

With this example we conclude this section on how to build a database.

2.4 Dates in R

Let's say few words about *dates* in R. Let's use `ppp_w` as example. Its frequency is annual. If we check the class for `year`

```
class(ppp_w$year)
```

```
## [1] "integer"
```

we observe that it is an integer and not a date class. To put it simple, for R a date is composed of year, month and day. We can extract a date from our `ppp_w$year` as follows using `zoo` package:

```
# create a new column named date
ppp_w$date <- as.Date.yearmon(ppp_w$year)
class(ppp_w$date)
```

```
## [1] "Date"
```

Let's make another example with a databse with a monthly frequency. Let's import the cpi dataset from the OECD with monthly frequency.

```
cpi <- read.table("~/R_progetti/Portfolio_management_pmwr/currency_model_dataset/
    header = TRUE, sep = ",")
```

```
cpi <- cpi[, c(3,9,17)]
colnames(cpi) <- c("iso_code", "month", "cpi")
class(cpi$month)
```

```
## [1] "factor"
```

In this case we have a factor class for month. Let's make it date and then extract also the quarter.

```
# make month as a date class
# create a new column named quarter
cpi$date <- cpi$month
cpi$month <-as.Date(as.yearmon(cpi$month))
class(cpi$month)
```

```
## [1] "Date"
```

```
cpi$quarter <- as.yearqtr(cpi$month)
```

The `yearmon()` returns a date in the default format year-month-day, where day is the beginning of the month. We could decide we want a specific day of the month instead. We can make it manually as follows

```
cpi$date <- paste(cpi$date, "15", sep ="-")
```

Note that in this case we have a character class

```
class(cpi$date)
```

```
## [1] "character"
```

that we can easily convert into date having the year-month-day format

```
cpi$date <- as.Date(cpi$date)
class(cpi$date)
```

```
## [1] "Date"
```

An advance package to manage dates is `lubridate()`. You can read more about `lubridate()` here https://cran.r-project.org/web/packages/lubridate/vignettes/lubridate.html

We conclude this section with a remark on the format of the date:

1. Four digit year is ‘
2. two digit year is ‘
3. numeric month is ‘
4. alphabetic (abbreviated) month is ‘
5. alphabetic (full) month is ‘
6. day is ‘

2.5 Plot

A trick to plot all the columns of a database is to make it as zoo class.

For example, let’s plot the GDP rom `gdp_w`. First let’s make it as zoo. Notice that zoo put the column with the dates at the place of the rows numbers (Fig. 2.20)

```
gdp_zoo <- read.zoo(gdp_w)
```

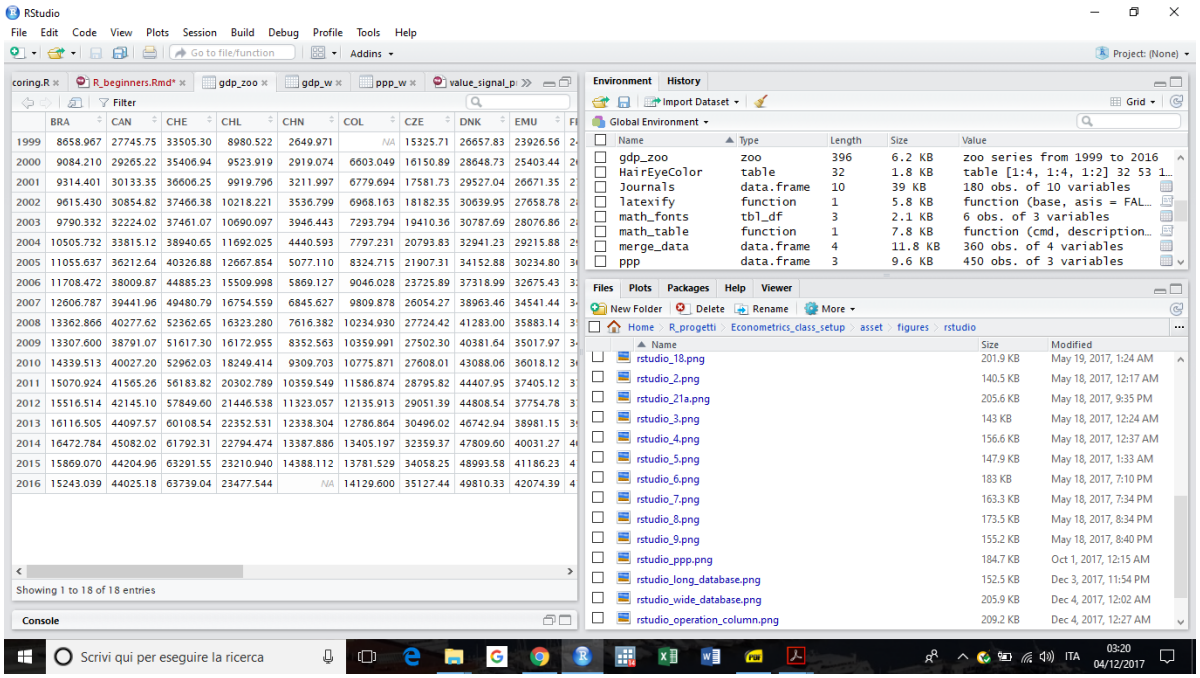


Figure 2.20: Zoo database

Remark: given that we have an annual frequency we didn’t transform the year in

Date class because it can be plotted as integer (a constant 1 integer difference).

Second, just plot the database as follows (because we have many countries we just selected the first 10):

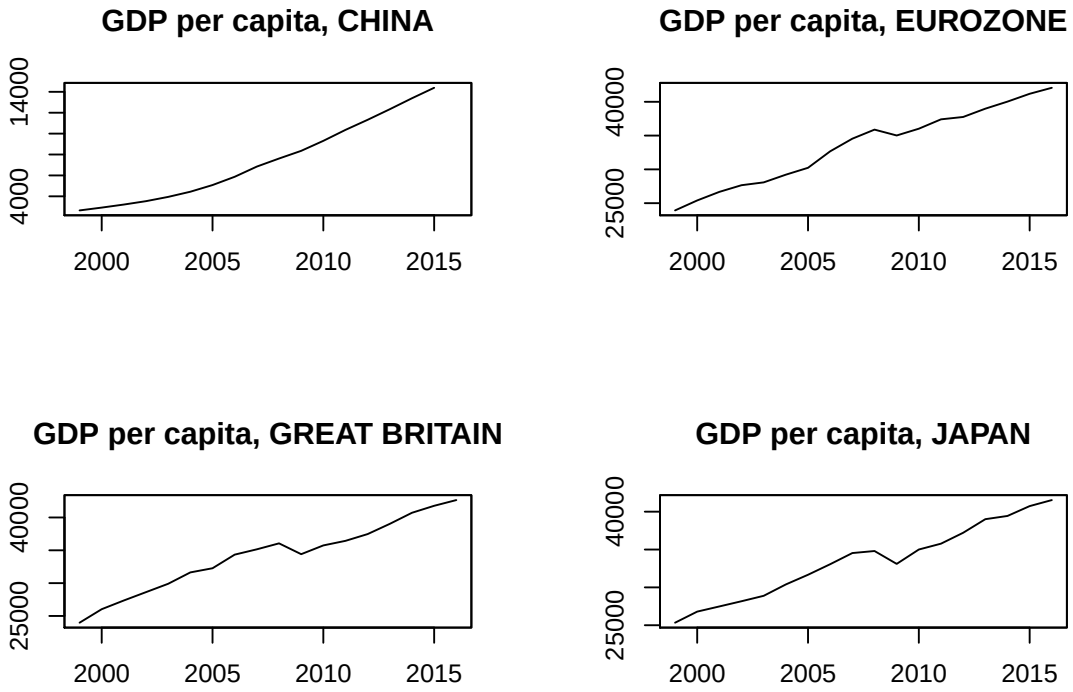
```
plot(gdp_zoo[, c(1:10)])
```

This is a quick way to have a look to the plot. Now let's see how to make nice plots.

2.5.1 Simple plot

Let's plot CHN, EMU, GBR and JPN.

```
par(mfrow = c(2, 2))
CHN_plot <- plot(gdp_zoo$CHN, main = "GDP per capita, CHINA", xlab = "", ylab = "
EMU_plot <- plot(gdp_zoo$EMU, main = "GDP per capita, EUROZONE", xlab = "",
    ylab = "")
GBR_plot <- plot(gdp_zoo$GBR, main = "GDP per capita, GREAT BRITAIN", xlab = "",
    ylab = "")
JPN_plot <- plot(gdp_zoo$JPN, main = "GDP per capita, JAPAN", xlab = "", ylab = "
```

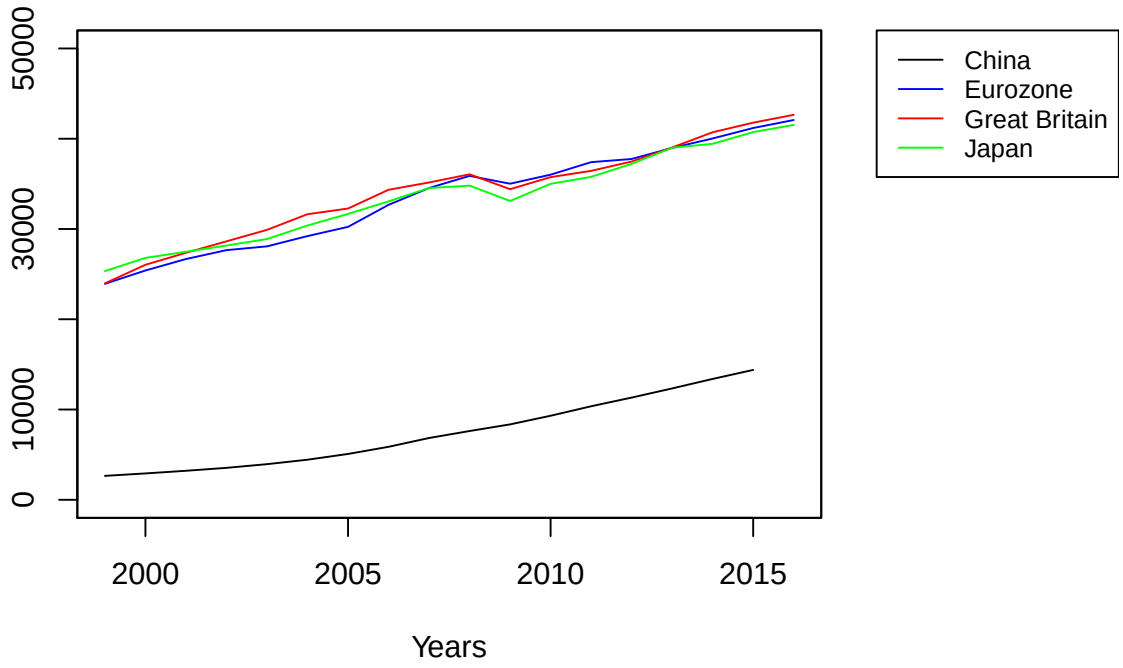



```
par(mfrow = c(1, 1))
```

Note that `par(mfrow = c())` sets the display of the plots. In this case, we said to R to display the plots on two rows and two columns. At the end, we set again the default, one plot; in `main =` we write the tile of the plot; we set `xlab = "", ylab = ""` equal to `""` to prevent R from creating default title for the axis.

Now let's plot all together.

```
par(xpd = T, mar = par()$mar + c(0, 0, 0, 6))
plot(CHN_plot, type = "l", col = "black", lty = 1, ylim = c(0, 50000), ylab = "",
    xlab = "Years")
lines(EMU_plot, type = "l", col = "blue", lty = 1)
lines(GBR_plot, type = "l", col = "red", lty = 1)
lines(JPN_plot, type = "l", col = "green", lty = 1)
legend("topright", inset = c(-0.4, 0), legend = c("China", "Eurozone", "Great Britain",
    "Japan"), col = c("black", "blue", "red", "green"), cex = 0.8, xpd = TRUE,
    lty = 1)
```




```
par(mar = c(5, 4, 4, 2) + 0.1)
```

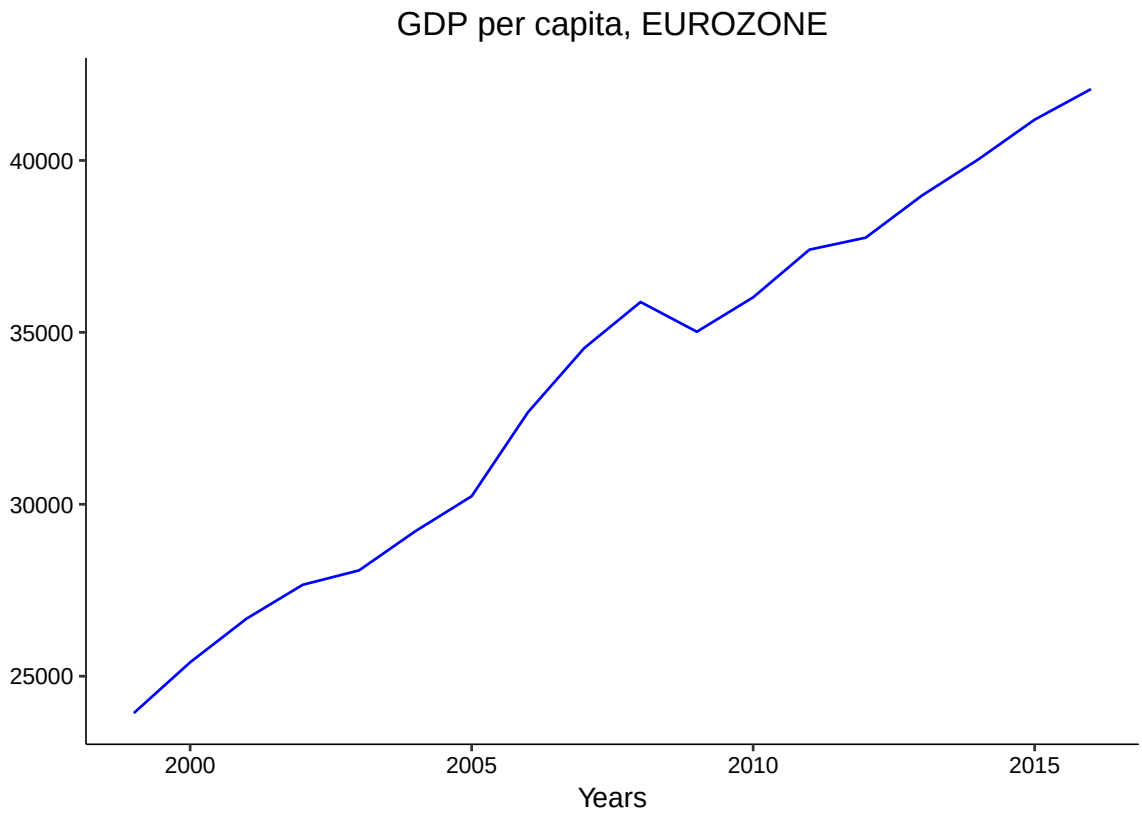
Note that the first and last expression sets the margin to put the legend out of the plot area. `type, lty, cex` are arguments for making the plot fancier. For details read **Keliber & Zeiles, p. 43**.

2.5.2 GGPLOT

Following two examples with GGPLOT.

In the first example we plot the GDP per capita for the Eurozone. First, in the `ggplot` function we set the database, the value for x axis (in this case years) and for y axis (in this case GDP per capita values) in `aes()`. Second, we decide for the the kind of plot, in this case line. All other options make the plot fancier. Try to include and exclude them to see the difference.

```
ggplot(gdp_w, aes(x = gdp_w$year, y = gdp_w$EMU)) + geom_line(size = 0.5, col = '
    labs(x = "Years", y = "", title = "GDP per capita, EUROZONE") + theme(plot.ti
    panel.background = element_rect(fill = "white"), axis.line = element_line(si
        colour = "black"), axis.text = element_text(colour = "black"), panel.gric
```



In this second example we plot together the GDP per capita for China, Eurozone, Great Britain and Japan as a bar plot. First we make a new dataframe, `gdp_df`, that contains `year_gdp`, a vector that replicates the year 4 times (because of the number of countries) and `country_gdp`, a vector of GDP per capita for each country. `type` replicates each country for the number of years. Second, we put these new data into the ggplot function. Now our database is `gdp_df` and values for x axis and y axis are, respectively, `year_gdp` and `country_gdp`. Third, we choose the kind of plot, bar, and add a number of options to make it fancier. Again, try to include and exclude them to see what changes.

```
year_gdp <- rep(gdp_w$year, 4)
CHN_gdp <- gdp_w$CHN
EMU_gdp <- gdp_w$EMU
GBR_gdp <- gdp_w$GBR
JPN_gdp <- gdp_w$JPN
country_gdp <- c(CHN_gdp, EMU_gdp, GBR_gdp, JPN_gdp)
type <- c(rep("China", 18), rep("EUROZONE", 18), rep("Great Britain", 18), rep("Japan",
    18))
gdp_df <- data.frame(year_gdp, country_gdp)
graph_gdp <- ggplot(gdp_df, aes(x = year_gdp, y = country_gdp)) + geom_bar(stat = "identi
    aes(fill = type), position = position_dodge()) + scale_fill_brewer(palette = "Paired
    theme_classic() + xlab("") + ylab("US dollars") + ggtitle("GDP per capita") +
```

```
    theme(plot.title = element_text(hjust = 0.5)) + theme(legend.position = "bot
    theme(legend.title = element_blank()) + scale_x_continuous(breaks = seq(1999
    2016, 2))
```

```
graph_gdp
```