DATA HANDLING IN HISTORICAL DEMOGRAPHY USING R

Riccardo Omenti

Lesson 3: June 30th

Affiliation: Department of Statistical Sciences, University of Bologna

E-mail: riccardo.omenti2@unibo.it

Contents

Introduction

This document provides an introduction about how to employ RStudio to carry out simple data analysis tasks in Historical Demography.

As you learned in the previous classes, RStudio is a very flexible programming language characterized by an enormous amount of build-in functions. This facilitates its use for researchers that are not experienced in Computer Programming.

Furthermore, in R it is possible to find an immense collection of packages. A R package may be defined as a collection of functions, documentation and data that are bundled together for a specific domain or purpose. Overall, employing packages in R makes it easier for users to carry out specific data analysis tasks. This may include producing fancier graphs (*ggplot*), implementing complex statistical models without entering into the detail of their mathematics (*lme4*). For each package available in R, a pdf document, which contains all the details surrounding its functionalities, is available.

Throughout this document, we will be using data sets that are available through ad hoc R packages or well-established data sources. The first part of the document is devoted to the use of dates in R. Time variables are essential for demographers, especially for researchers interested in Historical Demography. Following this introduction to dates in R, the document goes through the implementation of simple demographic and statistical methods in R that are essential for any demographer.

By providing code examples, step-by-step instructions, and practical explanations of key concepts, this guide equips demographers with the necessary skills to harness time variables and perform robust demographic analyses using R. Whether exploring historical trends, projecting future populations, or analyzing specific events, researchers will gain a comprehensive toolkit for extracting meaningful insights from demographic data.

With this introduction, the document sets the stage for a comprehensive and accessible guide to using R for Historical Demography.

Dates and Times

R has developed a special representation for dates and times. Within R, dates are represented by the *Date* class, whereas times are represented by the *POSIXct* or *POSIXlt* classes.

Dates are stored internally as the number of days since January 1st 1970, while times are recorded as the number of seconds since that same date.

It is not relevant to know the internal representation of dates in R. However, I believe it is a funny story to tell.

Dates in R

Dates are represented by the class *Date* and can be easily transformed from a character string to *Date* object using the build-in function *as.Date()*

```
# date as a character string

mydate = "2023-06-30"

# convert to Date object

mydate = as.Date(mydate)
```

You can see the internal representation of a date using the *unclass* function.

Times in R

Times are represented by the *POSIXct* or the *POSIXlt* classes. They are part of the POSIX (Portable Operating System Interface) standard for handling date and time in a consistent and platform-independent manner. Nonetheless, these classes come with their differences.

POSIXct has the following features:

- it represents date and time as the number of seconds since January 1, 1970, UTC (Coordinated Universal Time).
- it is stored as numeric value; hence, it is efficient for calculations and comparisons.
- it is created using the function *as.POSIXct*

POSIXlt has the following features:

- it represents date and time as a list of individual components (year, month, week, day, hours, minutes, seconds)
- each individual component is stored separately. This allows for more flexibility in accessing specific parts of a date and time.
- it is created using the function *as.POSIXlt*

Overall, when a researcher is faced with the decision to choose one of the two classes, he/she should take into account the following factors:

1. **PRECISION**: *POSIXct* is more precise as it stores dates and times in memory as numbers. On the other hand, *POSIXlt*, by treating each component separately, tends to consume more memory.
2. **FLEXIBILITY**: *POSIXlt* is more flexible as it allows an easier manipulation of the components of a date.
3. **EFFICIENCY**: *POSIXct* is more efficient for calculations that involve dates and times.

```
# object with current date and time

current_day_time = Sys.time()

# check the class

class(current_day_time)
```

```
## [1] "POSIXct" "POSIXt"
```

```
# POSIXct by default
```

By default, the output of the function *Sys.time()* is of class *POSIXct*. Using the *unclass* function on the output of *Sys.time()*, we obtain its internal numerical representation.

```
# internal numerical representation
unclass(current_day_time)
```

```
## [1] 1688029116
```

In order to change the class of the object, we can apply the function *as.POSIXlt()*. From this object, we are able to access the various components of a date and time.

```
# create POSIXlt object
current_day_time_new = as.POSIXlt(current_day_time)

# access the names of its components
```

```r
names(unclass(current_day_time_new))
```

```
##  [1] "sec"    "min"    "hour"   "mday"   "mon"    "year"   "wday"   "yday"
##  [9] "isdst"  "zone"   "gmtoff"
```

```r
# extract components
current_day_time_new$yday
```

```
## [1] 179
```

In order to convert a string into an object of *POSIXct* class, you can use the function *strptime*

```r
# create POSIXlt object
end_of_the_world = "December 21, 2012 00:00"

strptime(end_of_the_world,"%B %d, %Y %H:%M")
```

```
## [1] "2012-12-21 CET"
```

The symbols *%* are employed to format the strings. There is a multitude of formats out there. You can simply run the command *?strptime* to have a look at the extensive list of formats that this function has to offer.

The *lubridate* package

The *lubridate* package facilitates the use of dates in R that may be quite counter intuitive.

```r
#install lubridate package

#install.packages("lubridate")

# upload it

library("lubridate")


# alternatively

if(!require(lubridate)) {
  install.packages("lubridate"); require(lubridate)}
```

This packages handle dates written in distinct classes (*numeric*, *character*, *Date*).

```r
# numeric input
numeric = 20230630
ymd(numeric)
```

```
## [1] "2023-06-30"
```

```r
# character input
character = "2023/06/30"
ymd(character)
```

```
## [1] "2023-06-30"
```

It contains simple functions to extract the component of a date. Each function takes as input the *Date* object and a logical condition (*label*) that tells us whether the output must be numeric (label=F) or character (label=T)

```r
# date
today = ymd("2023/06/30")

# create date

today = make_date(year = 2023, month = 6, day = 30)


wday(today,label = T)
```

```
## [1] Fri
## Levels: Sun < Mon < Tue < Wed < Thu < Fri < Sat
```

```r
wday(today, label = F)
```

```
## [1] 6
```

If we are considering also times, we can easily handle different time zones.

```r
# date
today_time = Sys.time()

today_time = make_datetime(year = 2023, month = 6, day = 30,
               hour = 10, min = 30, sec = 45, tz = "Europe/Rome")


# Change the printing

with_tz(today_time,"America/Chicago")
```

```
## [1] "2023-06-30 03:30:45 CDT"
```

```r
# Change time

force_tz(today_time,"America/Chicago")
```

```
## [1] "2023-06-30 10:30:45 CDT"
```

```r
# format your output

format(today_time, "%T %Z")
```

```
## [1] "10:30:45 CEST"
```

Operations with dates

In R, the time difference between two dates can be measured through the build-in function *difftime()*. It automatically provides with the number of days between two dates. In the input, you can set this difference in terms of other time units, including seconds and weeks. However, this function does not include time measured in years.

```r
# date
today = today()

date_911 = make_date(year = 2001, month = 9, day = 11)


time_diff_days = difftime(today,date_911,units = "days")
```

```
time_diff_weeks = difftime(today,date_911 ,units = "weeks")
```

Within the package *lubridate*, you can store the time difference between two dates in a *duration* object.

```
# dates
today = today()

date_911 = make_date(year = 2001, month = 9, day = 11)

# time difference in seconds
time_diff = as.duration(today-date_911)

time_diff
```

```
## [1] "687830400s (~21.8 years)"
```

In order to change the time unit, we can simply divide by the number of seconds in a day, week, month etc. depending on our time unit of interest

```
# time difference in seconds
time_diff = as.duration(today-date_911)

# number of days
time_diff_days = time_diff/ddays(1)

# number of months
time_diff_weeks = time_diff/dmonths(1)

# number of years
time_diff_years = time_diff/dyears(1)
```

lubridate provides also functions to exact components of a date, including *year()*, *week()*, *day()*.

```
# extract year

year(date_911)
```

```
## [1] 2001
```

```
# extract month

month(date_911)
```

```
## [1] 9
```

```
# extract day

day(date_911)
```

```
## [1] 11
```

Data Example

In order to work with dates in R, we will use a famous historical data set, that is present in the package *HistData*, Florence Nightingale's Crimean War dataset. The data set provides the recorded number of deaths by cause (*Disease*, *Wound*, *Other*) and the corresponding rates during the Crimean War for the historical period 1854-1855. You can load the data set in R by installing the package *HistData* and by running the following command data("Nightingale").

The *HistData* package was created by Michael Friendly et al. (2022). It is characterized by 36 small data sets that are interesting and important in the history of statistics and data visualization. You can have a look at the data sets within the package by running *HistData::* and a description of the data set by issuing the command *help(NameOftheDataSet)*.

```
# install and upload the library
if(!require(HistData)) {
  install.packages("HistData"); require(HistData)}

# upload the data set

data("Nightingale")
```

Using this data set, we may calculate the overall number of days, weeks and years in which Florence Nightingale kept track of the deaths.

```
# overall number of weeks
print(as.duration(Nightingale$Date[nrow(Nightingale)]-Nightingale$Date[1])/dweeks(1))
```

```
## [1] 100
```

```
# overall number of days
print(as.duration(Nightingale$Date[nrow(Nightingale)]-Nightingale$Date[1])/ddays(1))
```

```
## [1] 700
```

```
# overall number of days
print(as.duration(Nightingale$Date[nrow(Nightingale)]-Nightingale$Date[1])/dyears(1))
```

```
## [1] 1.916496
```

As a simple exploratory analysis, we may check how cause-specific death rates changed across the observation period.

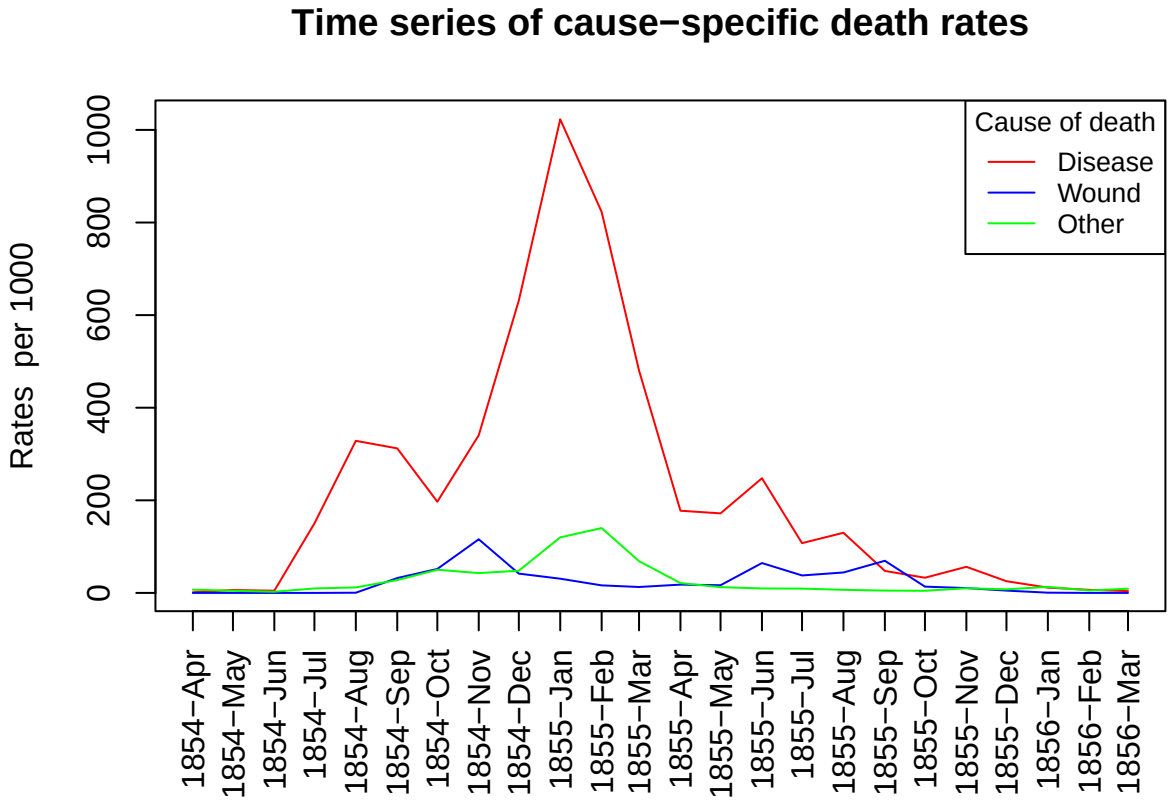
By means of the function *axis.Date()*, we can easily customize the appearance of the labels of the. x- or y-axis when dealing with dates.

This function requires the following:

- *x*: to specify the axis to which the date belongs, namely 1 for x-axis and 2 for y-axis;
- *at*: to customize the appearance of the dates, the displayed time intervals;
- *format*: to customize the desire format for the dates;
- *las*: to control the orientation of the labels.

In the *las* option, if we set it equal to 0 or 1 (default option), the labels will be perpendicular to the axis. On the other hand, if we set it equal to. 2, they will be parallel to the axis

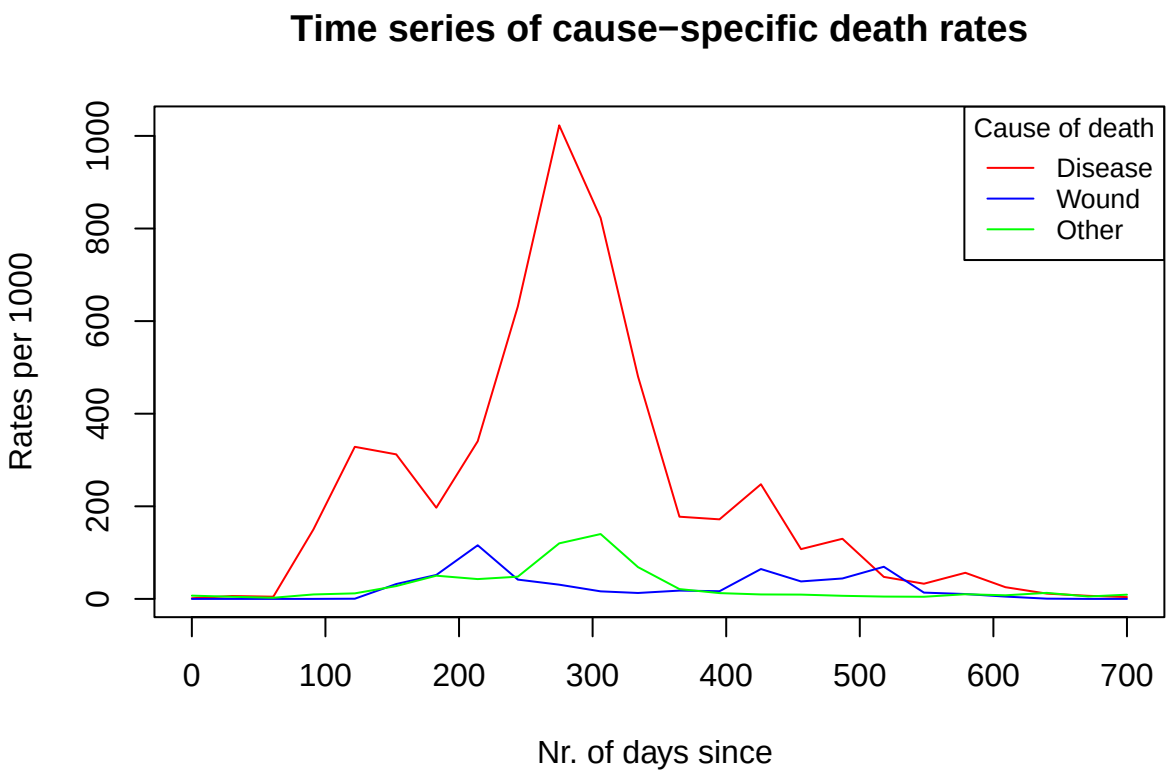
```
plot(Nightingale$Date,Nightingale$Disease.rate,xaxt="n",xlab = "",
     ylab = "Rates  per 1000",type="l",pch = 18, col = "red")
axis.Date(1, at = seq(Nightingale$Date[1], Nightingale$Date[nrow(Nightingale)], by = "month"),
          format = '%Y-%b', las = 2)
lines(Nightingale$Date,Nightingale$Wounds.rate, pch = 18, col = "blue", type = "l")
lines(Nightingale$Date,Nightingale$Other.rate, pch = 18, col = "green", type = "l")
legend("topright", legend=c("Disease", "Wound", "Other"),
       col=c("red", "blue","green"), lty = 1, cex=0.8, title = "Cause of death")
title("Time series of cause-specific death rates") # add title
```



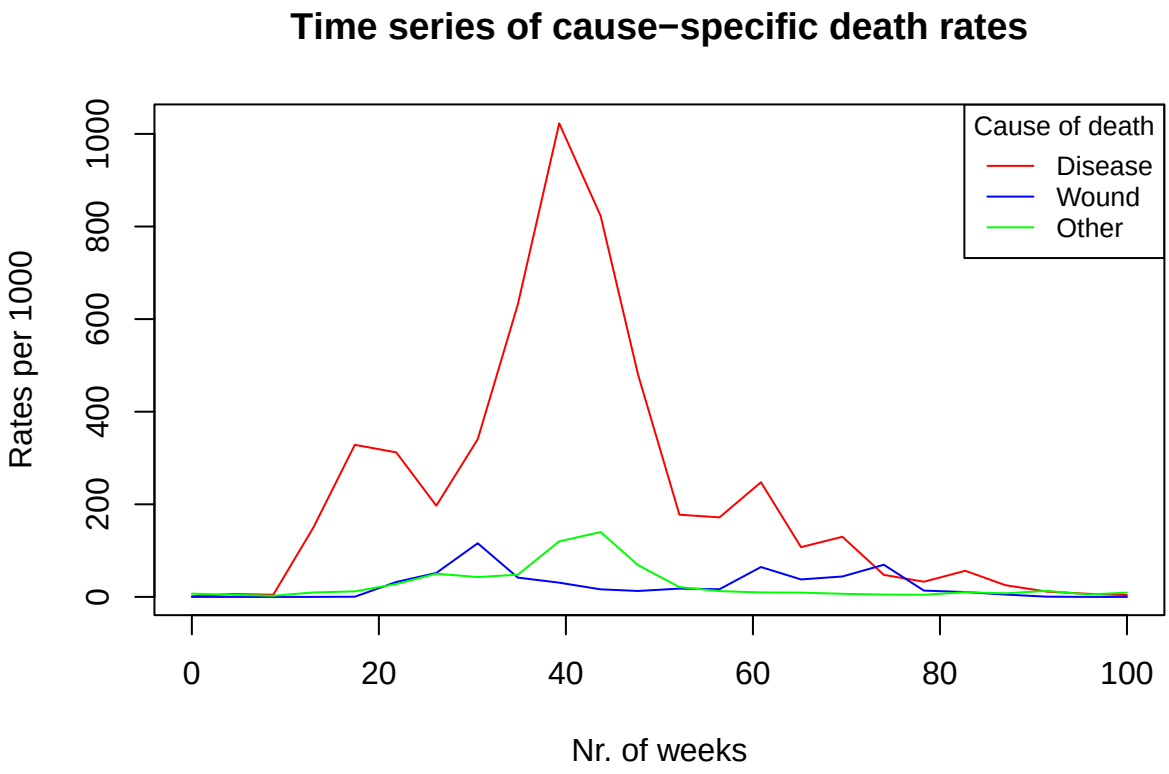
A useful exercise is to change the temporal scale of the previous plot. We may plot cause-specific death rates as function of the number of days or weeks since Florence Nightingale had started recording the data.

For this, we need to create two additional variables, namely the number of days (*nr_days*) and the number of weeks (*nr_weeks*) since the first recording.

```
Nightingale$nr_days = as.duration(Nightingale$Date-Nightingale$Date[1])/ddays(1)
plot(Nightingale$nr_days,Nightingale$Disease.rate,xlab = "Nr. of days since ",
     ylab = "Rates per 1000",type="l",pch = 18, col = "red")
lines(Nightingale$nr_days,Nightingale$Wounds.rate, pch = 18, col = "blue", type = "l")
lines(Nightingale$nr_days,Nightingale$Other.rate, pch = 18, col = "green", type = "l")
legend("topright", legend=c("Disease", "Wound", "Other"),
       col=c("red", "blue","green"), lty = 1, cex=0.8, title = "Cause of death")
title("Time series of cause-specific death rates")
```




```
Nightingale$nr_weeks = as.duration(Nightingale$Date-Nightingale$Date[1])/dweeks(1)
plot(Nightingale$nr_weeks,Nightingale$Disease.rate,xlab = "Nr. of weeks",
     ylab = "Rates per 1000",type="l",pch = 18, col = "red")
lines(Nightingale$nr_weeks,Nightingale$Wounds.rate, pch = 18, col = "blue", type = "l")
lines(Nightingale$nr_weeks,Nightingale$Other.rate, pch = 18, col = "green", type = "l")
legend("topright", legend=c("Disease", "Wound", "Other"),
       col=c("red", "blue","green"), lty = 1, cex=0.8, title = "Cause of death")
title("Time series of cause-specific death rates")
```



Calculation of Demographic Indicators in R: the case of Italy

In this section, we will implement the calculation of some simple demographic indicators in R. In particular, the following indicators will be considered: Crude Birth Rate (CBR), Crude Death Rate (CDR), Rate of Natural Increase (R), Doubling Time (T_{double}).

As an example, the Italian population from 1872 up to 2019 will be examined. The Human Mortality Data provides us with population, death and birth counts for the entire observation period 1872-2019.

In order to carry out the analysis, we will consider four data sets.

- births : yearly number of births stratified by sex in Italy during the historical period 1872-2019
- exposure : mid-year population (or person-years) classified by sex in Italy during the historical period 1872-2019
- deaths : yearly number of deaths classified by sex in Italy during the historical period 1872-2019
- pop_size : yearly population counts classified by sex in Italy during the historical period 1872-2019
- mid_year_pop : yearly population counts classified by sex and age in Italy during the historical period 1872-2019

All these data sets are contained in the .RData file "Italy_Data.RData".

An RData file is used in R for saving and loading R objects, such as data frames, matrices, lists, functions, or any other R objects. It enables you to save your R workspace through the function *save.image()* or specific R objects to a file for later use or sharing with others by means of the function *save()*.

The calculation of the indicators for the demographic analysis require us to link the previous data sources.

Crude Birth Rates

Let's start by calculating the crude birth rates (CBR) for Italy across the historical period 1872-2019.

The Crude Birth Rate for some country c during the year t ($CBR_{t,c}$) is given by the following equation, namely ratio of the number of births to the mid-year population in calendar year t for a country of interest c.

$$CBR_{t,c} = \frac{B_{t,c}}{E_{t,c}} \times 1000$$

where $B_{t,c}$ and $E_{t,c}$ are the number of births for some country c during the calendar year t and the mid-year population for the year t in country c.

In R, it is fairly simple to compute demographic indicators. All we need is:

- the mathematical formula.
- simple knowledge about data structure in R.

For our case, we create a new data set *birth_analysis* by linking the data sets *births* and *exposure* using the variable *Year* as a key.

```
# load all the data sets
load("Italy_Data.RData")

# data set for the analysis of birth dynamics

birth_analysis = merge(births[,c("Year","total_births")],
                       exposure[,c("Year","total_exp")],by="Year")
```

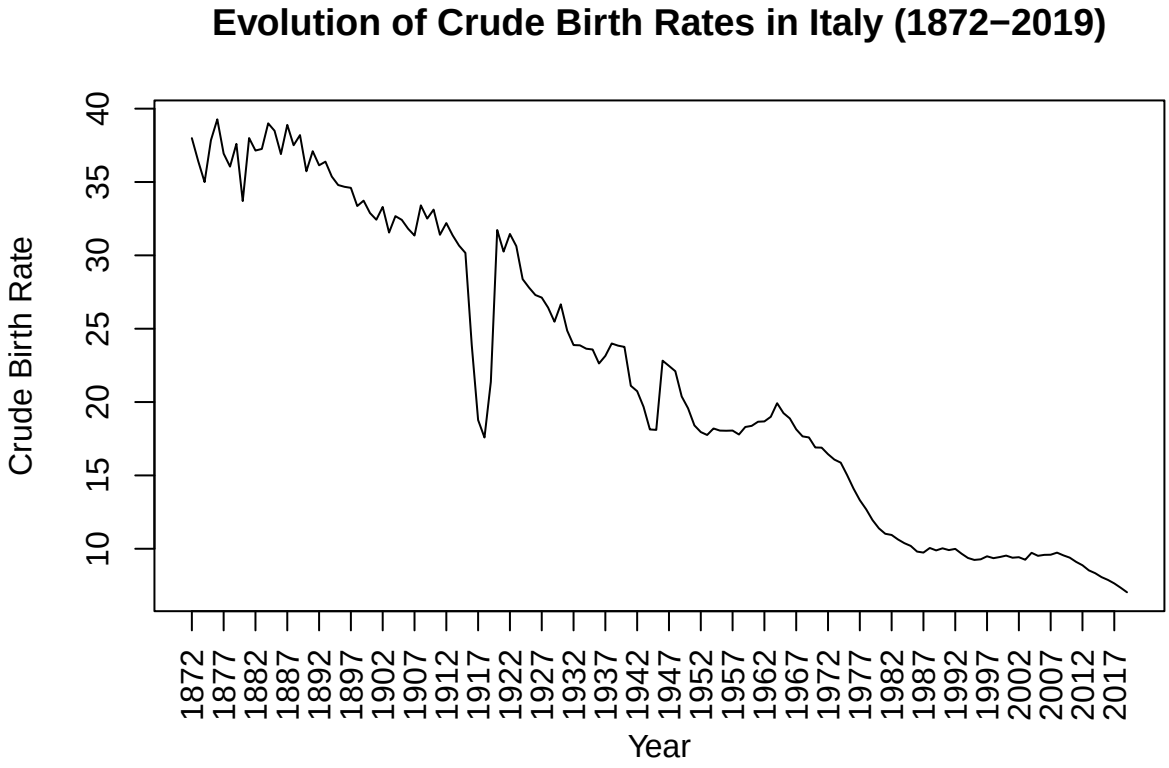
We can add the crude birth rate as an additional variable to the data set. In this case, we divide the column referring to the birth counts by the one for the population exposure. R will perform the division element by element. Note that to calculate the ratio of two vectors in R, you can simply divide the elements of one vector by the corresponding elements of the other vector.

```
birth_analysis$birth_rates = birth_analysis$total_births/birth_analysis$total_exp*1000
```

We may plot the time series of the crude birth rates for the historical period 1872-2019.

We can customize the appearance of the axis labels by means of the *axis()* command. This function works very similarly to *axis.Date()*, but it is used for numerical and categorical variables. If we wish to use it for a given axis, we must make sure that the plot() function does not print the original labels. Otherwise, the customize labels will be overwritten right above the original ones. This error can be prevented by setting xaxt="n" for the x-axis or yaxt="n" for the y-axis depending on which label we would like to customize.

```
# plot for CBR during the period 1872-2019

plot(birth_analysis$Year,birth_analysis$birth_rates,type = "l",xaxt="n",
     xlab = "Year",ylab = "Crude Birth Rate")
axis(1,at=seq(1872,2019,5),srt = 60,las=3)
title("Evolution of Crude Birth Rates in Italy (1872-2019)")
```



###

Relative Variation in Crude Birth Rates

In order to gain more information about the change in crude birth rates across the entire observation period, we may calculate the relative variation in the number of crude birth rates between consecutive years expressed in %.

The formula is given by

$$\Delta \text{CBR}_c(t, t+1) = \frac{CBR_{t+1,c} - CBR_{t,c}}{CBR_{t,c}} \times 100$$

The calculation of such indicator is fairly straightforward. It is sufficient to calculate the difference between the crude birth rates recorded at two adjacent years and to divide this difference by the crude birth rate at the earlier year.

This operation can be done in R using the function *diff()*. The latter takes as input a vector of numbers and gives as output a vector of the differences between consecutive elements of the input vector. However, we must be careful since the size of the vector in the output is 1-unit smaller than the number of elements of the input vector.

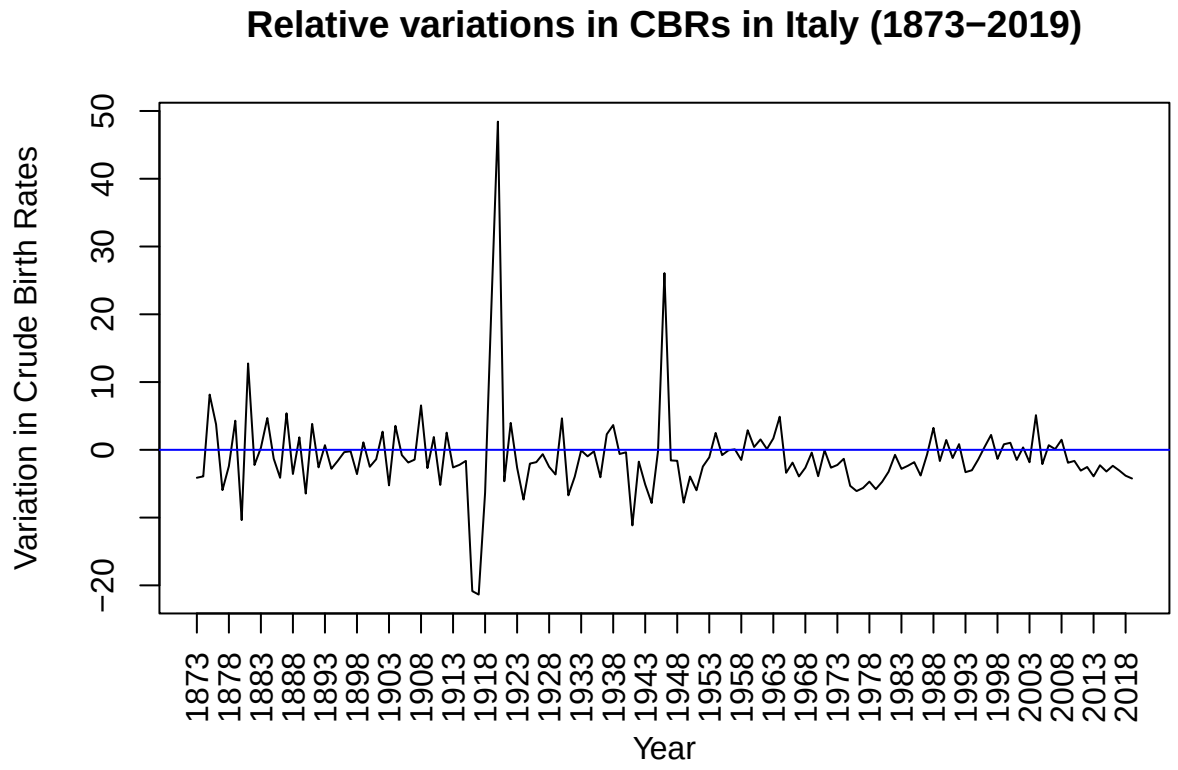
In our example, the function will provide the user with the differences in crude birth rates from 1873 (rather than 1872) onwards. Of course, when we divide by the rate at the earlier year, we must not consider the rate for the last year of the observation window.

```
#relative variations in CBRs
var_italy_birth_rates= diff(birth_analysis$birth_rates)/birth_analysis$birth_rates[-nrow(birth_analysis
```

When we plot the annual relative variations over the historical period 1873-2019. In R, we can add an horizontal line parallel to the x-axis to present the "baseline" case of zero growth.

```
# plot for relative variations in CBRs during the period 1872-2019

plot(birth_analysis$Year[-1],var_italy_birth_rates,xlab="Year",ylab="Variation in Crude Birth Rates",ty
```

```r
axis(1,at=seq(1873,2019,5),srt = 60,las=3)
abline(h=0,col="blue")
title("Relative variations in CBRs in Italy (1873-2019)")
```



Crude Death Rates

We continue with the calculation of the crude death rates.

The Crude Death Rate for some country c during the year t ($CDR_{t,c}$) is given by the following equation, namely ratio of the number of deaths to the mid-year population in calendar year t for a country of interest c.

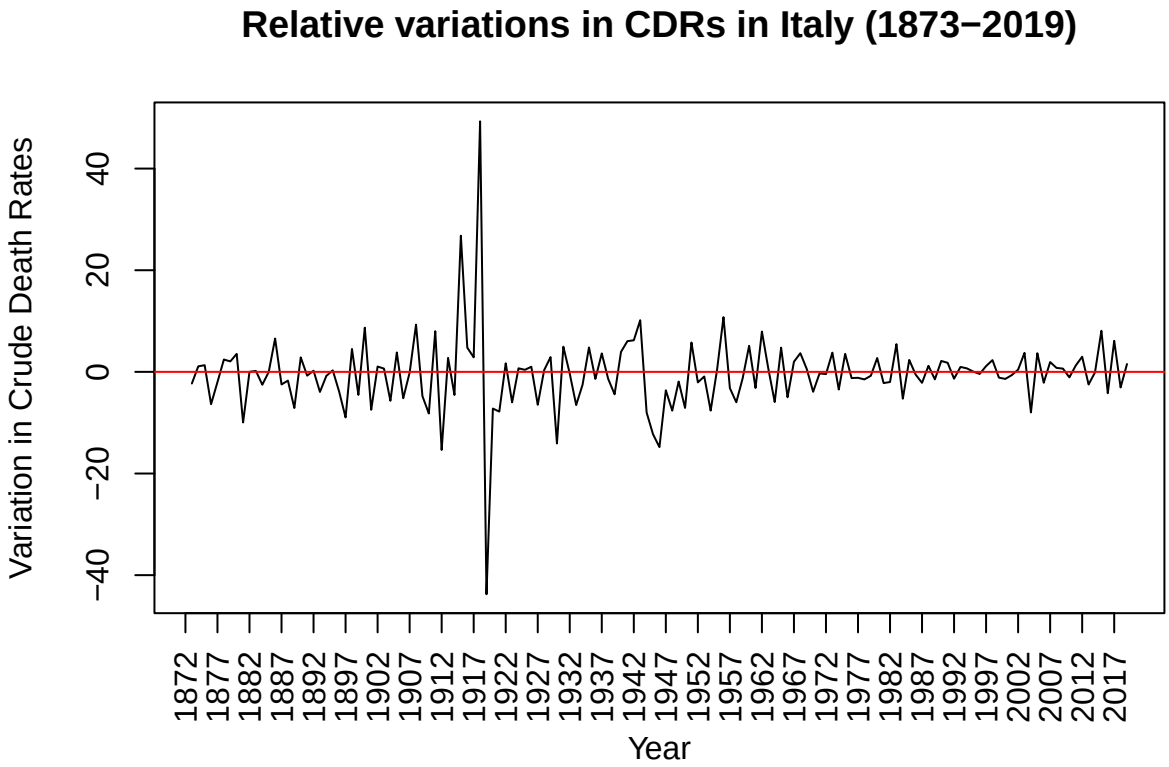
$$CDR_{t,c} = \frac{D_{t,c}}{E_{t,c}} \times 1000$$

where $D_{t,c}$ is the number of deaths in year t for some country c.

```r
# data set for the analysis of deaths
death_analysis = merge(deaths[,c("Year","total_deaths")],
                       exposure[,c("Year","total_exp")],by="Year")
# CDR calculation
death_analysis$death_rates = death_analysis$total_deaths/death_analysis$total_exp*1000

# plot for CDR during the period 1872-2019
plot(death_analysis$Year,death_analysis$death_rates,type = "l",xaxt="n",
     xlab = "Year",ylab = "Crude death Rate")
axis(1,at=seq(1872,2019,5),srt = 60,las=3)
title("Evolution of Death Rates in Italy (1872-2019)")
```

In a similar fashion, we can compute the relative variations in the crude death rates.

```
# relative variations in CDRs
var_italy_death_rates = diff(death_analysis$death_rates)/death_analysis$death_rates[-nrow(death_analysis
# plot for relative variations in CDRs during the period 1872-2019
plot(death_analysis$Year[-1],var_italy_death_rates,type = "l",xaxt="n",
     xlab = "Year",ylab = "Variation in Crude Death Rates")
axis(1,at=seq(1872,2019,5),srt = 60,las=3)
abline(h=0,col="red")
title("Relative variations in CDRs in Italy (1873-2019)")
```

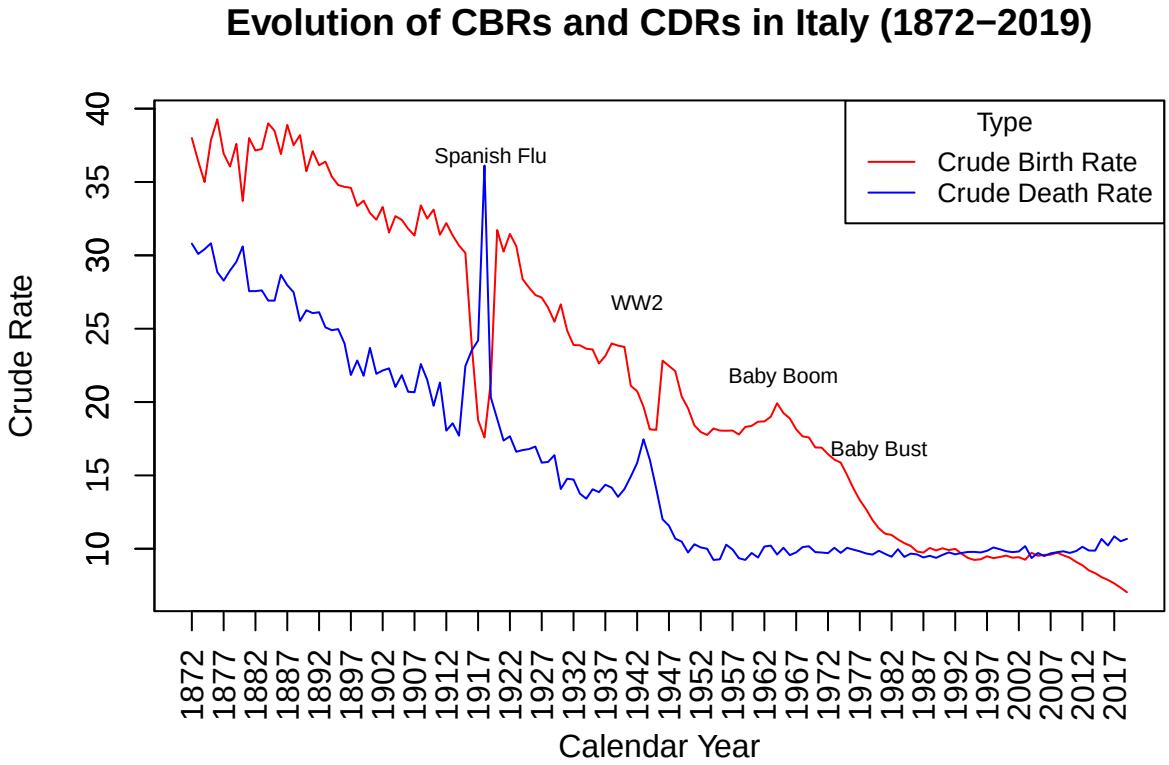


In R, we can represent both series of crude rates in a single plot. We can add the command *lines()* after *plot()* for each series of rates we would like to consider. It is advisable to set for each line a different color. The choice of the color is managed by the *col* argument. We must declare the name of the color in lower-case letters enclosed in quotation mark.

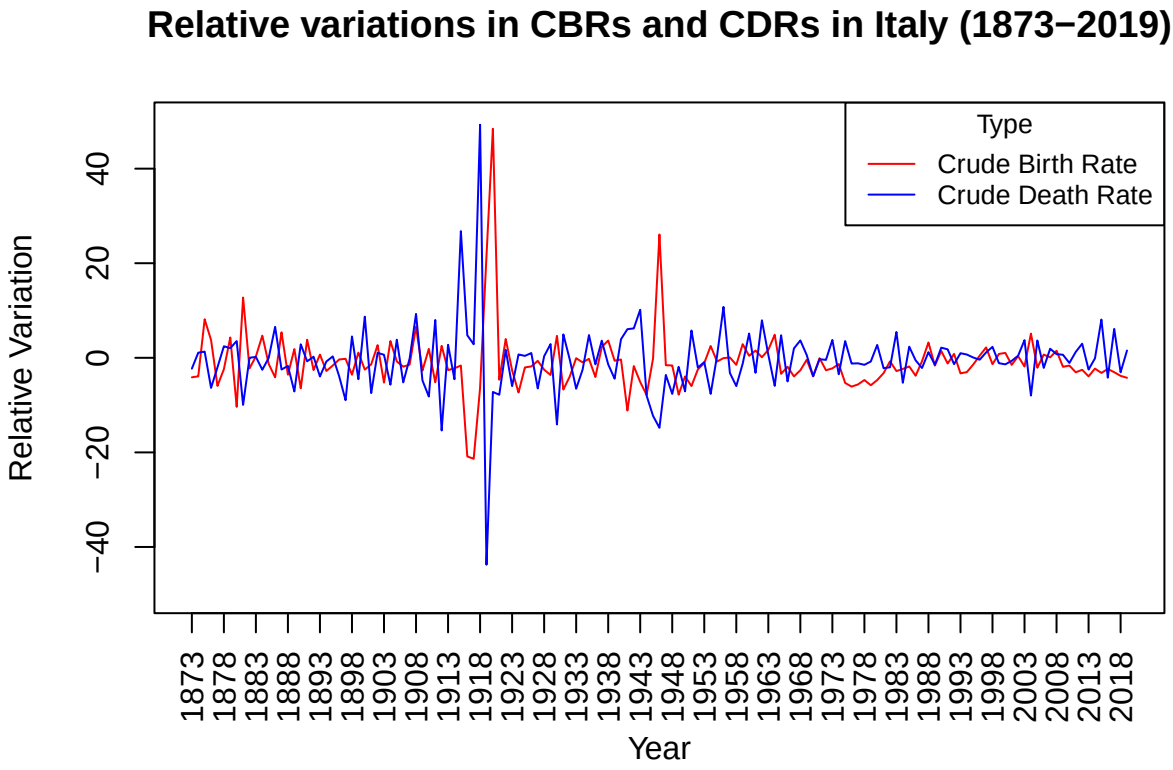
We can also add pieces of text on the plot with the command *text()*. The latter requires as input: coordinates where text is placed (x, y), the actual text (*label*), text size (*cex*), the position of the text relative to the coordinates (*pos*).

In addition, it is possible to add a legend by means of *legend()*. This function takes as input: the position of the legend on the plot (*position*), the names of the elements in the legend (*legend*), the corresponding colors (*col*), the line type (*lty* : 1 (solid line), 2 (dashed line), and 3 (dotted line)), the size of the names of the elements in the legend (*cex*), the title of the legend.

```
# both CDRs and CBRs on the same plot
plot(birth_analysis$Year,birth_analysis$birth_rates,xaxt="n",type = "l",
     xlab = "Calendar Year",ylab = "Crude Rate",col="red")
lines(death_analysis$Year,death_analysis$death_rates,
      xaxt="n",col="blue")
legend("topright",
       legend=c("Crude Birth Rate", "Crude Death Rate"),
       col=c("red", "blue"), lty = 1, cex=0.8, title = "Type")
axis(1,at=seq(1872,2019,5),srt = 60,las=3)
axis(2,at=seq(0,60,10))
text(x=1919,y=35,cex=0.65, pos=3,label="Spanish Flu")
text(x=1965,y=20,cex=0.65, pos=3,label="Baby Boom")
text(x=1942,y=25,cex=0.65, pos=3,label="WW2")
text(x=1980,y=15,cex=0.65, pos=3,label="Baby Bust")
title("Evolution of CBRs and CDRs in Italy (1872-2019)")
```




```r
# relative variations in CDRs and in CBRs on the same plot
plot(birth_analysis$Year[-1],var_italy_birth_rates,xlab="Year",
     ylab="Relative Variation",type = "l",xaxt="n",col="red",
     ylim=c(-50,50))
lines(death_analysis$Year[-1],var_italy_death_rates,xaxt="n",col="blue")
legend("topright", legend=c("Crude Birth Rate", "Crude Death Rate"),
       col=c("red", "blue"), lty = 1, cex=0.8, title = "Type")
axis(1,at=seq(1873,2019,5),srt = 60,las=3)
title("Relative variations in CBRs and CDRs in Italy (1873-2019)")
```



Rates of Natural Increase

The annual Rate of Natural Increase may be calculated by taking the difference between the annual crude birth rates and the annual crude death rates.

The formula is given by

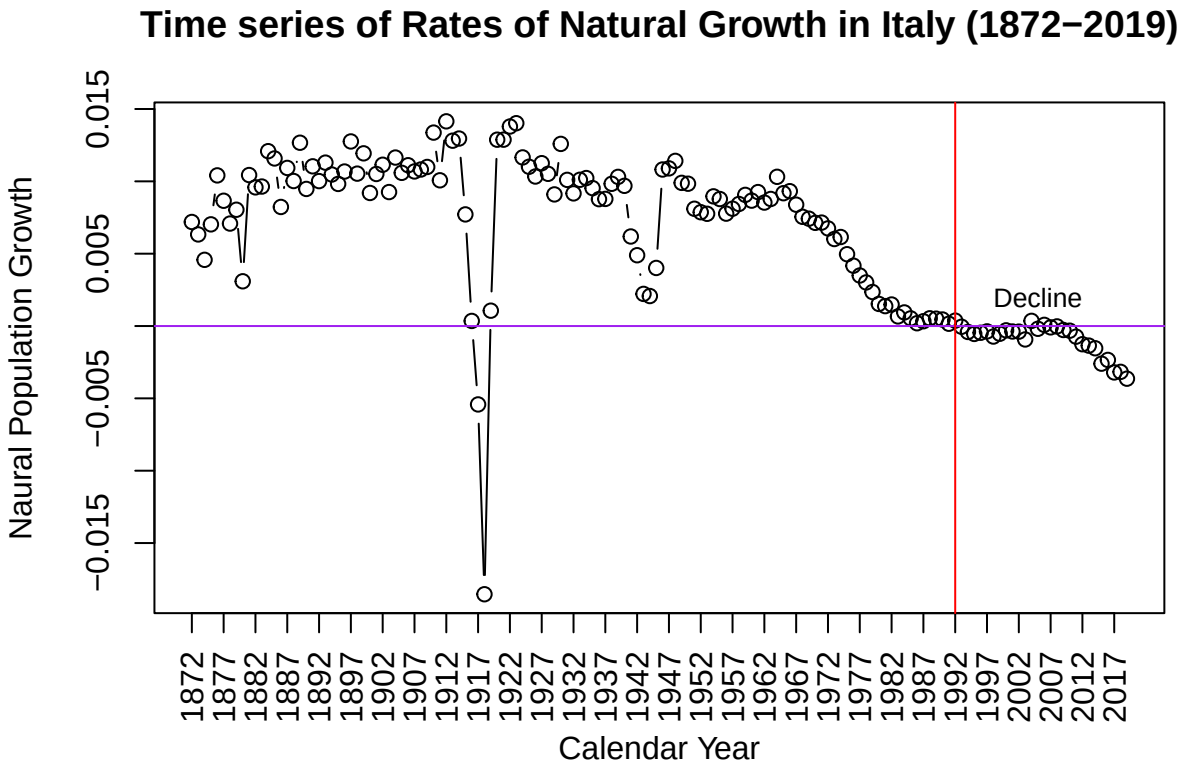
$$R_{t,c} = \frac{CBR_{t,c} - CDR_{t,c}}{1000}$$

In R, it is sufficient to take the different between the vectors of the annual crude birth rates and the annual crude death rates. As a result, we will have a vector of element-wise differences.

```
# Rate of Natural Increase calculation
natural_increase_rates = (birth_analysis$birth_rates-death_analysis$death_rates)/1000
```

We can plot the time series of the annual Rates of Natural Increase from 1973 up to 2019.

```
# plot of Rates of Natural Increase for the period 1872-2019
plot(birth_analysis$Year,natural_increase_rates,xaxt="n",type = "b",
     xlab = "Calendar Year",ylab="Naural Population Growth")
axis(1,at=seq(1872,2019,5),srt = 60,las=3)
abline(h=0,col="purple")
abline(v=1992,col="red")
text(2005,0,cex=0.80,pos=3,"Decline")
title("Time series of Rates of Natural Growth in Italy (1872-2019)")
```



Exponential Growth Model

Suppose that the Italian Population can be modeled through an exponential growth model. We also assume that the natural increase of such population is equal to the Italian Rate of Natural Increase from 1972 to 1973, namely 0.007.

An interesting exercise is to examine how the Italian population sizes would have been under the exponential growth model.

The implementation is fairly simple. All we need is:

- an initial population size (the population size in 1872)
- a Rate of Natural Increase (the Rate of Natural Increase in 1872)
- a functional form for modeling the relationship between the initial population size at some year (t_0) and the population sizes in subsequent years ($t_0 < t$)

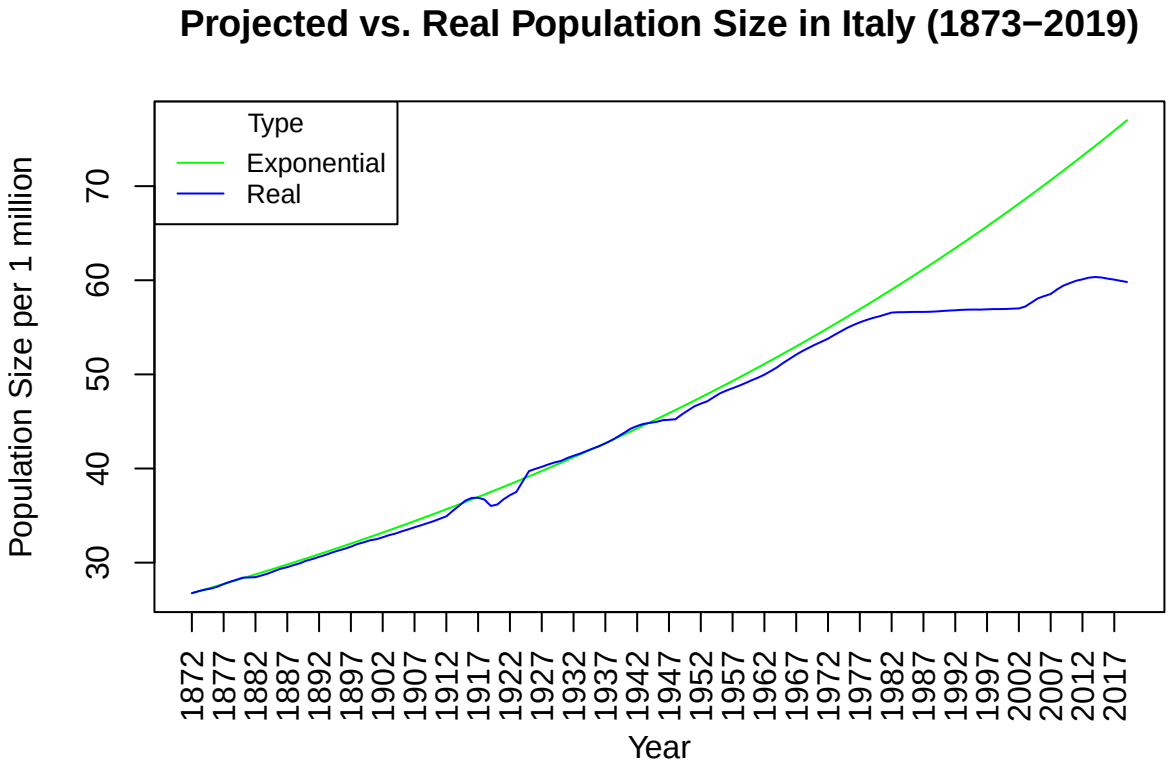
$$P_t = P_{t_0} e^{R(t-t_0)}$$

In R, we store the numerical value of each input parameter in a variable. We apply the previous model to the observation window 1873-2019.

```
# Initial Population Size (= Pop. size in 1872)
P_init = pop_size$Total_pop[1]
# Fixed Rate of Natural Increase (= Rate of Natural Increase in 1872)
R = natural_increase_rates[1]
# Time horizon for the simulation
time_horizon = 1873:2019
# Projected Population Sizes
P_exponential = P_init*exp(R*(0:length(time_horizon)))
```

We can create a plot that compare the true trajectory of the population size against the trajectory based upon the Exponential Growth Model.

```
# Plot projected and real population sizes
plot(1872:2019,P_exponential/1e06,xlab="Year",ylab="Population Size per 1 million",
     xaxt="n",type = "l",col="green")
lines(pop_size$Year,pop_size$Total/1e06,col="blue")
axis(1,at=seq(1872,2019,5),srt = 60,las=3)
legend("topleft", legend=c("Exponential","Real"),
       col=c("green","blue"), lty = 1, cex=0.8, title = "Type")
title("Projected vs. Real Population Size in Italy (1873-2019)")
```



Doubling Time

Assuming that the previous model holds true, how long would it take the Italian population to double at the proposed Rate of Natural Increase. We may check how much it differs from the true doubling time.

There is a simple formula for which only the Rate of Natural Increase is needed.

$$T_{\text{double}} = \frac{\log(2)}{R}$$

In order to compute the true doubling time, it suffices to find the first year in which the Italian population is bigger than the italian population in 1872

```
# Model-based doubling time
T_double_model = log(2)/R
print(T_double_model)
```

```
## [1] 96.39883
```

```r
# True doubling time
T_double_true = min(pop_size$Year[pop_size$Total_pop>2*P_init])-1872
print(T_double_true)
```

```
## [1] 100
```

```r
# Difference in the two
T_double_true-T_double_model
```

```
## [1] 3.601165
```

Halving Time

Now, let us assume that the 2019 Rate of Natural Increase remains constant from 2019 on wards, Since this rate is negative, it may be an interesting exercise to check how long it would take the Italian Population to halve.

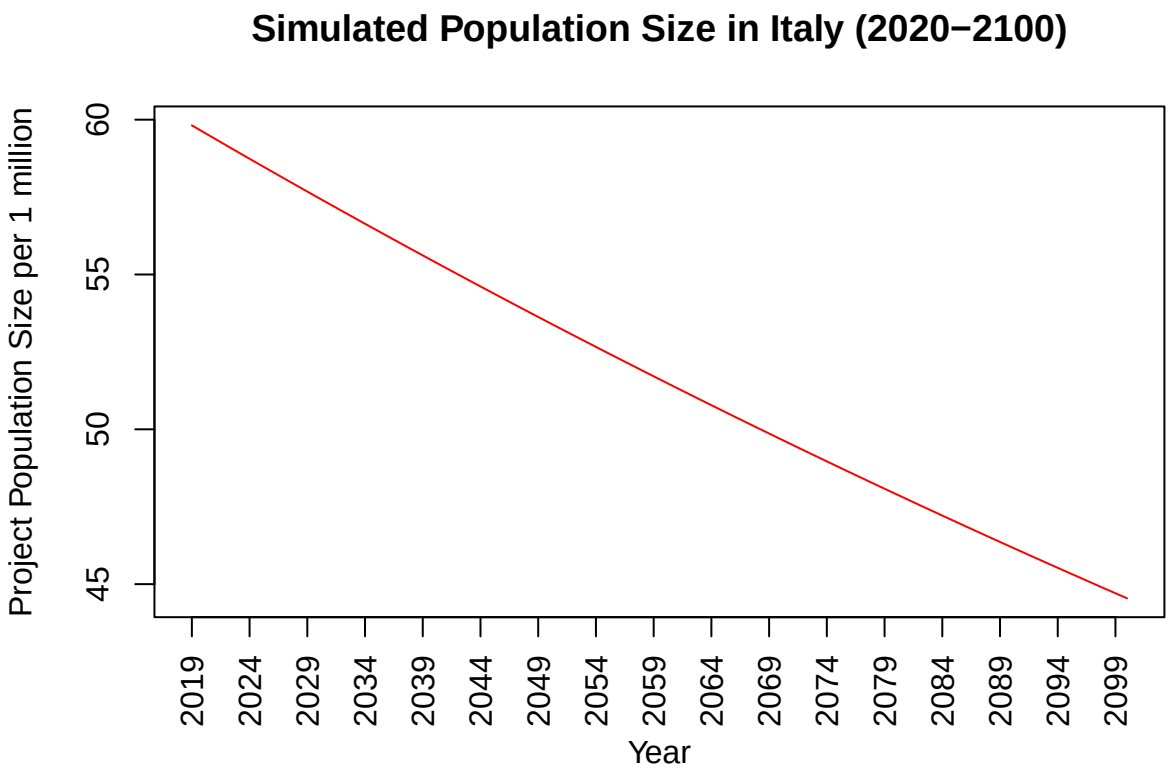
The formula for the halving time is given by

$$T_{\text{halve}} = -\frac{log(2)}{R}$$

```r
# Fixed Rate of Natural Increase (=Rate of Natural Increase in 2019)
R_new = natural_increase_rates[length(natural_increase_rates)]
# Model-based Halving time
T_halve = (-1)*log(2)/R_new
print(T_halve)
```

```
## [1] 190.4466
```

```r
# Projection horizion
projection_horizon = 2020:2100
# Initial Population Size (= pop. size in 2019)
P_init = pop_size$Total_pop[pop_size$Year==2019]
# Projected Population Sizes
P_projected = P_init*exp(R_new*0:length(projection_horizon))
#Plot projected population sizes
plot(2019:2100,P_projected/1e06,xlab="Year",ylab="Project Population Size per 1 million",
     type="l",xaxt="n",col="red")
axis(1,at=seq(2019,2100,5),srt = 60,las=3)
title("Simulated Population Size in Italy (2020-2100)")
```


Simulated Population Size in Italy (2020−2100)
Project Population Size per 1 million
45
50
55
60
2019
2024
2029
2034
2039
2044
2049
2054
2059
2064
2069
2074
2079
2084
2089
2094
2099
Year

Calculation of structural ratios

The demographic composition of Italy can also be assessed using simple structural indexes. In particular, one may interested in uncovering certain aspects inherent in the age structure of the Italian Population.

Consider the data set *pop_age* that contains the population counts by age and sex from 1872 up to 2019.

Dependency Ratio

The Dependency Ratio can be calculate by taking the ratio of the dependent population (0-14 and 65+) to the population in working age (15-64) for some calendar year t in a country c. This ratio can also be decomposed as the sum of the Child Dependency Ratio and the Aged Dependency Ratio.

$$I^d_{t,c} = \frac{P_{65+,t,c} + P_{0-14,t,c}}{P_{15-64,t,c}} = \underbrace{\frac{P_{0-14,t,c}}{P_{15-64,t,c}}}_{\text{Child Dependency Ratio}} + \underbrace{\frac{P_{65+,t,c}}{P_{15-64,t,c}}}_{\text{Aged Dependency Ratio}}$$

The calculation in R is fairly simple. Starting from the data set *pop_age*, we note that age is of type character and should be transformed to numeric. We cannot simply apply the function as.numeric() since the value "110+" will transformed to NA. Hence, an *ifelse* command is also necessary to take care of this.

After transforming Age into a numerical variables, we select Year, Age, Total. Then, we can create a new variable that identifies the three age categories ($0-14$,$15-64$ and $65+$). In order to calculate the index, we use the function *aggregate()* to calculate the total group sizes by year.

```
# Make sure Age is numeric
pop_age$Age = ifelse(pop_age$Age == "110+", 110, as.numeric(pop_age$Age))
```

```
## Warning in ifelse(pop_age$Age == "110+", 110, as.numeric(pop_age$Age)): NAs
## introduced by coercion
```

```
# Make sure Year is numeric
pop_age$Year = as.numeric(pop_age$Year)
```

```
## Warning: NAs introduced by coercion
```

```
# Select only Year, Age and Total
Italy_dep = pop_age[,c("Year","Age","Total")]
# Create a categorical variable for the three age groups
Italy_dep$Age_group = ifelse(Italy_dep$Age<15,"young",ifelse(Italy_dep$Age>64,"old","working"))
# Calculate the total population counts by year and age group
Italy_dep = aggregate(Italy_dep$Total,by=list(Italy_dep$Year,Italy_dep$Age_group),FUN=sum)
# rename the columns
colnames(Italy_dep) = c("Year","Age_group","Pop")
```

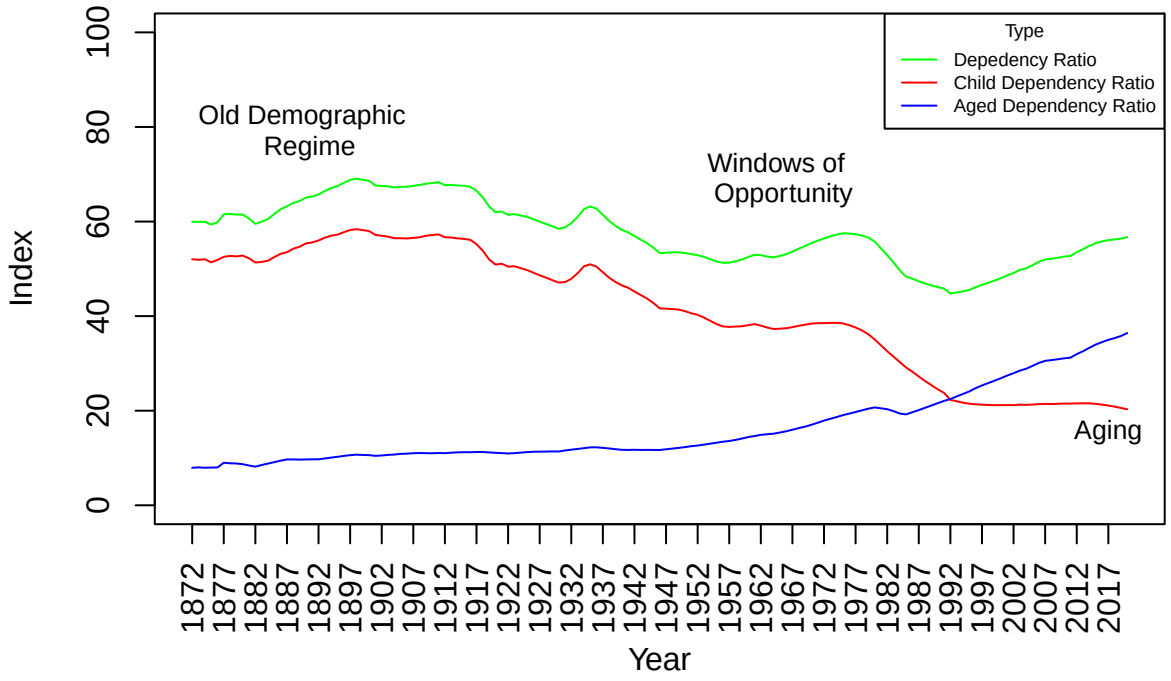
We can transform the data set from long to wide format. This is achieved by means of the command *reshape()*.

The *reshape()* function allows to restructure your data based on specified identifiers, time periods, or categories. The first argument is the data to be restructured. The second *idvar* is the variable that uniquely identifies each record. Then, *timevar* represents the different time periods or categories. Finally, setting *direction* to "wide" implies that we want to convert the data set from long format to wide format.

```
# write the data set in wide format
Italy_dep  = reshape(Italy_dep, idvar = "Year",timevar = "Age_group",direction = "wide")

# Dependency Ratio
Italy_dep$Dep_Index = (Italy_dep$Pop.old+Italy_dep$Pop.young)/Italy_dep$Pop.working*100
# Young Dependency Ratio
Italy_dep$Young_Index = (Italy_dep$Pop.young)/Italy_dep$Pop.working*100
```

```r
# Aged Dependency Ratio
Italy_dep$Old_Index = (Italy_dep$Pop.old)/Italy_dep$Pop.working*100
```

```r
# Generate a plot with the time series for three ratios
plot(Italy_dep$Year,Italy_dep$Dep_Index,xlab="Year",ylab="Index",
     xaxt="n",type = "l",col="green",ylim=c(0,100))
lines(Italy_dep$Year,Italy_dep$Young_Index,col="red")
lines(Italy_dep$Year,Italy_dep$Old_Index,col="blue")
axis(1,at=seq(1872,2019,5),srt = 60,las=3)
legend("topright", legend=c("Depedency Ratio","Child Dependency Ratio","Aged Dependency Ratio"),
       col=c("green","red","blue"), lty = 1, cex=0.6, title = "Type")
text(1890,70,cex=0.80,pos=3,"Old Demographic \n Regime")
text(1965,60,cex=0.80,pos=3,"Windows of \n Opportunity")
text(2017,10,cex=0.80,pos=3,"Aging")
```

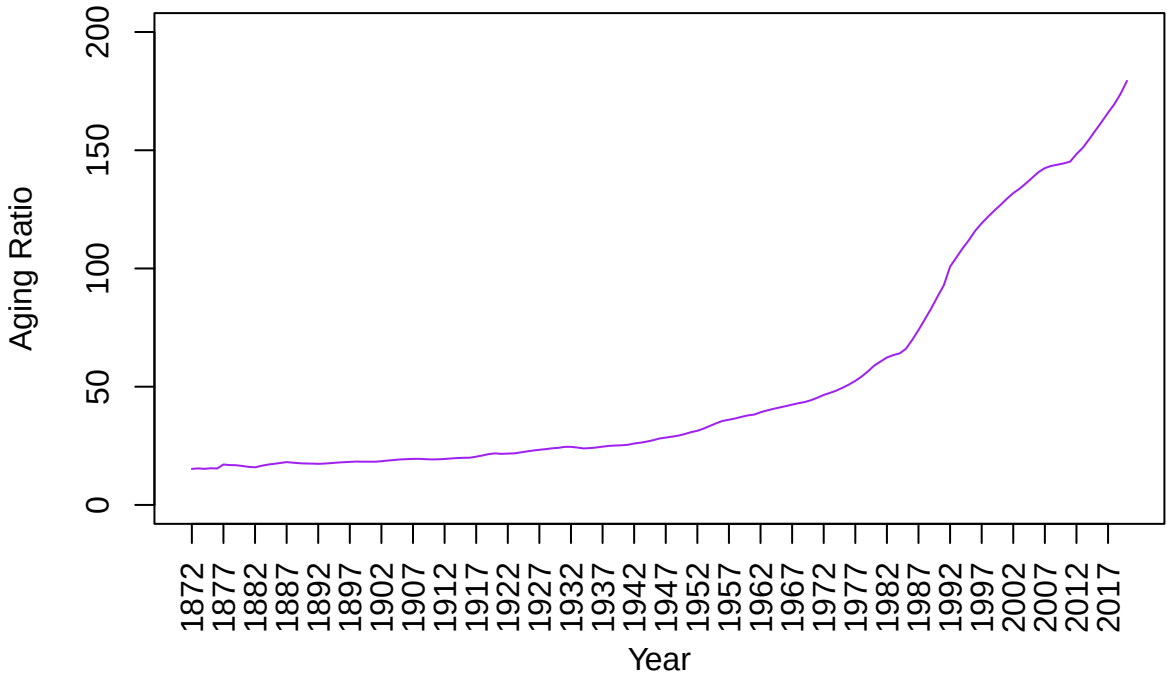


Aging Ratio

It is ratio of population aged 65 or more to the total population aged 14 or less for a country c in a calendar year t.

$$I^a_{t,c} = \frac{P_{t,65+}}{P_{t,0-14}}$$

```r
# Aging Ratio
Italy_dep$Aging_Ratio = Italy_dep$Pop.old/Italy_dep$Pop.young*100

# Plot for Aging Ratio
plot(Italy_dep$Year,Italy_dep$Aging_Ratio,xlab="Year",ylab="Aging Ratio",
     xaxt="n",type = "l",col="purple",ylim=c(0,200))
axis(1,at=seq(1872,2019,5),srt = 60,las=3)
```



Linear model in R

In this section, we will illustrate how to implement a linear model in R and how to interpret its output.

We will be working with the *gapminder* data coming from the homonym package. This data set contains for 142 countries the following variables:

- continent where the country is locatate (*continent*)
- country name (*country*)
- year of measurement (*year*)
- life expectancy at birth (*lifeExp*)
- population size (*pop*)
- GDP per capita (*gdpPercap*)

Premise

As any statistical model in the world, the linear regression model is far from being perfect and requires the researcher to make multiple assumptions on the data generating process.

The examples I am going to illustrate in the upcoming pages of the document are mainly employed for teaching purposes.

There are several issues we are not accounting for, including possible correlations among model residuals and reverse causation. Furthermore, in order to avoid dependencies among observations arising from the same country, the model is fit to a sub-sample of observations with the variable *Year* being equal to 1952. In case we wish to consider multiple observations per country, we should have opted for more complex models in the field of Longitudinal Data Analysis.

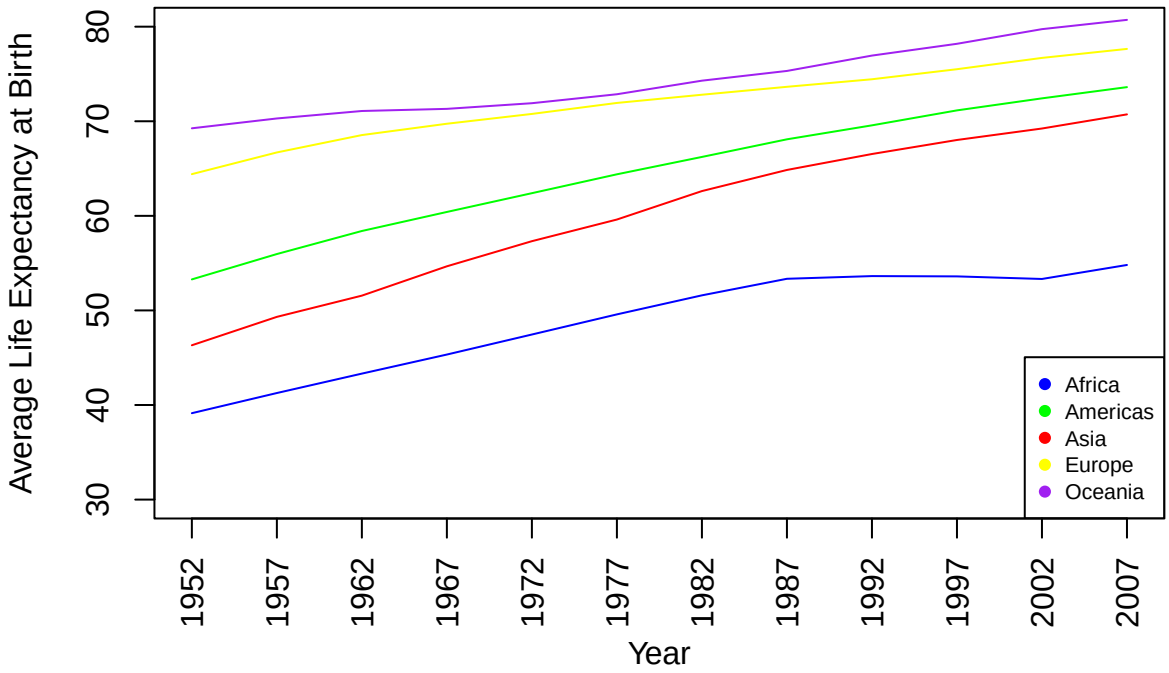
A bit of exploratory analysis

Some simple visual inspections can be carried out. In our case, we may calculate the average life expectancy and the total population size by continent and year.

```r
# install gapminder
if(!require(gapminder)) {
  install.packages("gapminder"); require(gapmindera)}
```

```
## Loading required package: gapminder
```

```r
# call the data set
data("gapminder")
# calculate the average life exp. at birth
# by continent and year
average_life_Exp = aggregate(gapminder$lifeExp,by=list(gapminder$continent,gapminder$year),FUN=mean)
# rename the variables
colnames(average_life_Exp) = c("Continent","Year","Average_life_exp")

# plot the average life exp.
# one line for each continent
plot(average_life_Exp$Year[average_life_Exp$Continent=="Americas"],
     average_life_Exp$Average_life_exp[average_life_Exp$Continent=="Americas"],
     type="l",col="green",xlab="Year",ylab="Average Life Expectancy at Birth",ylim=c(30,80),xaxt="n")
axis(1,at=unique(average_life_Exp$Year),srt = 60,las=3)
lines(average_life_Exp$Year[average_life_Exp$Continent=="Europe"],
     average_life_Exp$Average_life_exp[average_life_Exp$Continent=="Europe"],col="yellow")
lines(average_life_Exp$Year[average_life_Exp$Continent=="Asia"],
     average_life_Exp$Average_life_exp[average_life_Exp$Continent=="Asia"],col="red")
lines(average_life_Exp$Year[average_life_Exp$Continent=="Africa"],
     average_life_Exp$Average_life_exp[average_life_Exp$Continent=="Africa"],col="blue")
lines(average_life_Exp$Year[average_life_Exp$Continent=="Oceania"],
     average_life_Exp$Average_life_exp[average_life_Exp$Continent=="Oceania"],col="purple")
legend(
  x ="bottomright",
  legend = levels(average_life_Exp$Continent), # for readability of legend
  col = c("blue","green","red","yellow","purple"),
  pch = 19, # same as pch=20, just smaller
  cex = .7 # scale the legend to look attractively sized
)
```



For our analysis, we focus our attention on the year 1952. We can start by creating a reduced data set with records having the variable year being equal to 1952.

```
# consider only the year 1952
gapminder_subset = gapminder[gapminder$year==1952,]
```

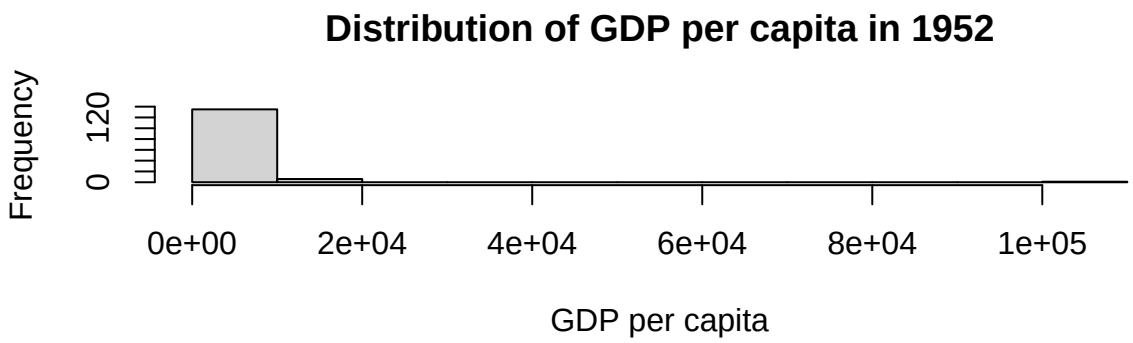
We can start with a simple exploratory analysis.

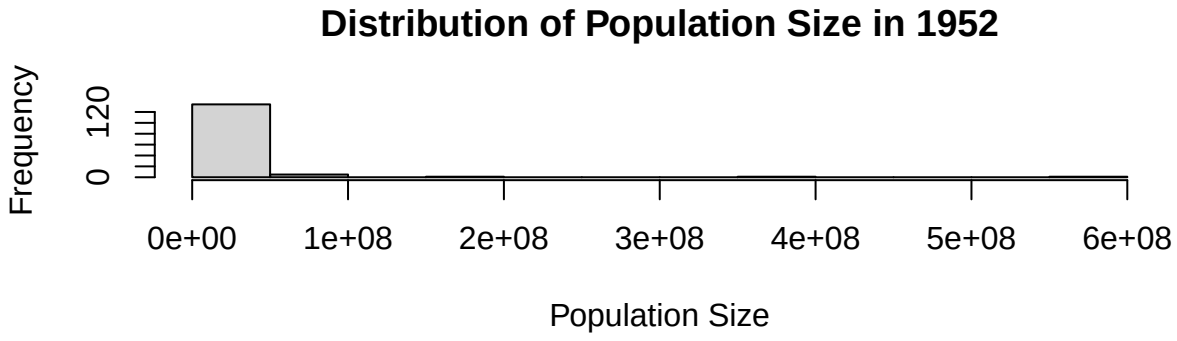
We can create a scatter plots to explore the following relationship:

- life expectancy at birth vs GDP per capita
- life expectancy at birth vs Population size

If we visually represent the distribution of GDP per Capita and Population Size, we can note that both tend to be heavily skewed. Hence, it may be adviseable to perform a logarithmic transformation.

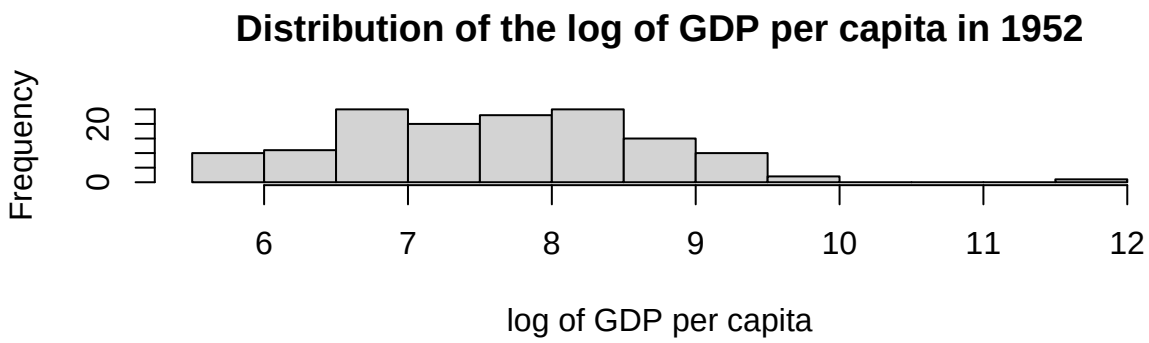
```
# plot histograms for GDP per capita and. Population Size in 1952
par(mfrow = c(2,1))
hist(gapminder_subset$gdpPercap,
     xlab = "GDP per capita",
     main = "Distribution of GDP per capita in 1952")

hist(gapminder_subset$pop,
     xlab = "Population Size",
     main = "Distribution of Population Size in 1952")
```

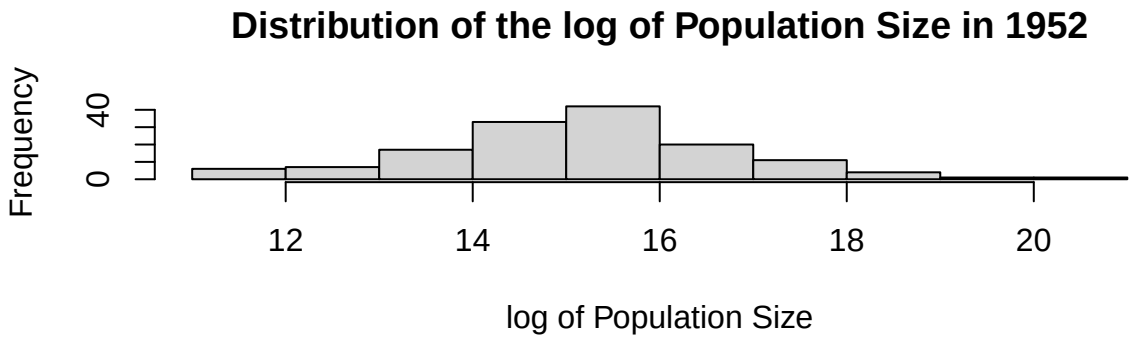




We can inspect the relationship between life expectancy and the two log-transformed variables. We can also label points on the catersian plane according the country's continent.

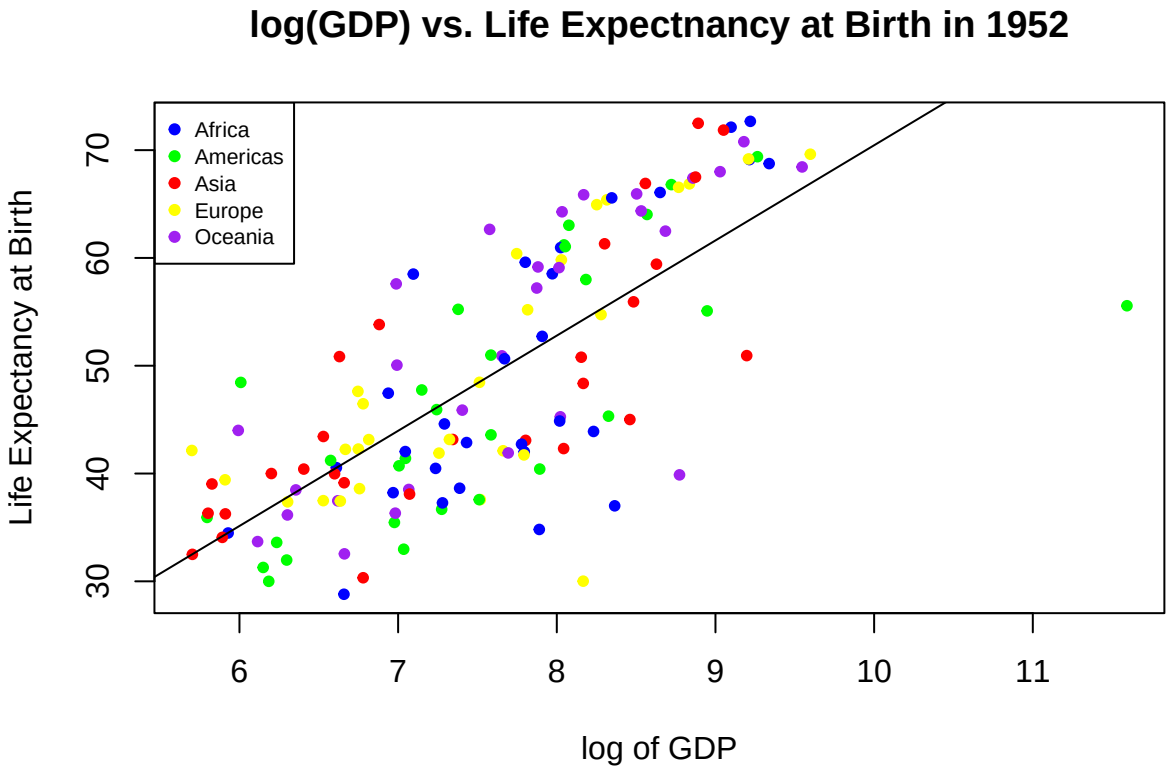
```
# plot histograms for log of GDP per capita and log of Population Size in 1952
par(mfrow = c(2,1))
hist(log(gapminder_subset$gdpPercap),
     xlab = "log of GDP per capita",
     main = "Distribution of the log of GDP per capita in 1952")

hist(log(gapminder_subset$pop),
     xlab = "log of Population Size",
     main = "Distribution of the log of Population Size in 1952")
```





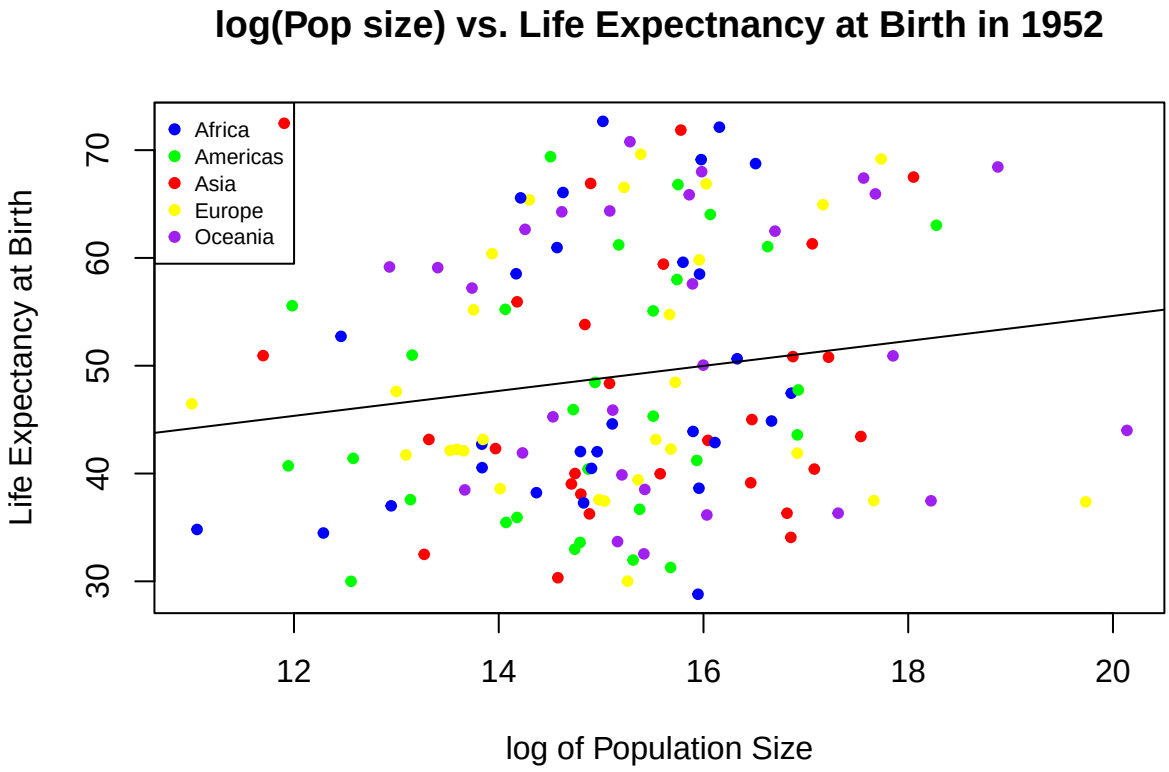
Life expectancy at birth vs log of GDP per capita

```
plot(log(gapminder_subset$gdpPercap),
     gapminder_subset$lifeExp,
     xlab="log of GDP",ylab="Life Expectancy at Birth",
     col=c("blue","green","red","yellow","purple"),pch = 20)
legend(
  x ="topleft",
  legend = levels(gapminder_subset$continent), # for readability of legend
  col = c("blue","green","red","yellow","purple"),
  pch = 19, # same as pch=20, just smaller
  cex = .7 # scale the legend to look attractively sized
)
abline(lm(lifeExp~I(log(gdpPercap)),gapminder_subset))
title("log(GDP) vs. Life Expectnancy at Birth in 1952")
```



Life expectancy at birth vs log of Population size

```
plot(log(gapminder_subset$pop),
     gapminder_subset$lifeExp,
     xlab="log of Population Size",
     ylab="Life Expectancy at Birth",
     col=c("blue","green","red","yellow","purple"),pch = 20 )
legend(
  x ="topleft",
  legend = levels(gapminder_subset$continent), # for readability of legend
  col = c("blue","green","red","yellow","purple"),
  pch = 19, # same as pch=20, just smaller
  cex = .7 # scale the legend to look attractively sized
)
abline(lm(lifeExp~I(log(pop)),gapminder_subset))
title("log(Pop size) vs. Life Expectnancy at Birth in 1952")
```



Simple Linear Model

A simple linear regression model can be fitted using the *lm()* function.

You need to specify the *formula* in the form y~x1 where y is the dependent variable and x1 is the independent one. By means of the *data* argument, you also must specify the name of the data where the variables involved in the regression are stored.

You can store the result of your model in the R environment. Then, you can apply the *summary* function to the saved object to look at the model results.

```
# run simple linear model
simple_regression = lm(lifeExp~I(log(gdpPercap)),gapminder_subset)
# show the results
summary(simple_regression)
```

```
##
## Call:
## lm(formula = lifeExp ~ I(log(gdpPercap)), data = gapminder_subset)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -28.9571  -5.7319   0.7517   6.5770  13.7361
##
## Coefficients:
##                   Estimate Std. Error t value Pr(>|t|)
## (Intercept)       -17.8457     5.0668  -3.522 0.000578 ***
## I(log(gdpPercap))   8.8298     0.6626  13.326  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
```

```
## Residual standard error: 8.146 on 140 degrees of freedom
## Multiple R-squared:  0.5592, Adjusted R-squared:  0.556
## F-statistic: 177.6 on 1 and 140 DF,  p-value: < 2.2e-16
```

The output of the *summary()* command provides the user with the model results, including the estimated regression coefficient under the *estimate* column, the standard error of such estimates under the *Std. Error* column, the t-test for testing the significance of the regression coefficients under the column *t value*, the p-value of the t-test under the column *p_value*. The asterisks close to the p-value column inform the user about the statistical significant of the regression coefficients.

- '***' means that the p-value of the test is less than 0.001
- '**' means that the p-value of the test is less than 0.01.
- '*' means that the p-value is less than 0.05.
- '.' means that the p-value is less than 0.1.

The model also include other quantities that can be calculated from the model, including the residual standard error, the R-squared statistics and its adjusted version, the results of the F-test. This test checks the overall significance of the fitted model.

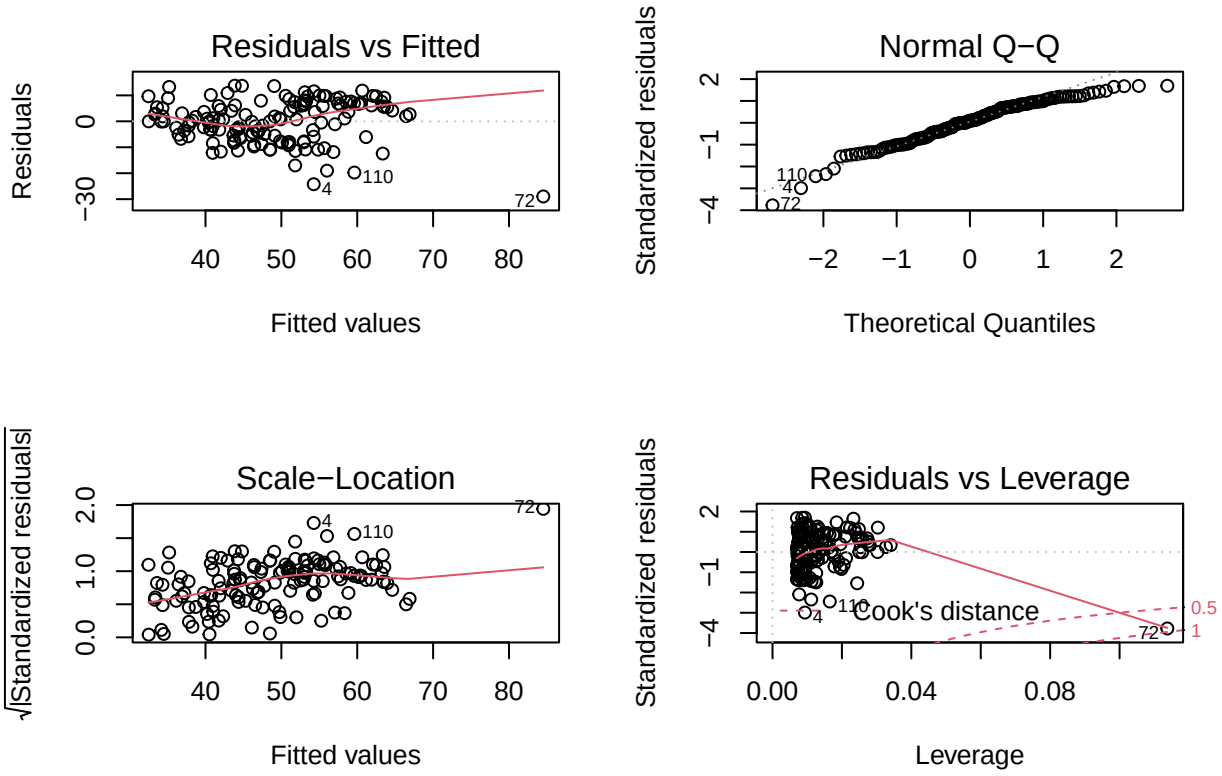
If we apply the function *coef()* to the summary of the model, we will get as a result the output of the model in a matrix form.

```
# look at the coefficients
coef(summary(simple_regression))
```

```
##                     Estimate Std. Error   t value     Pr(>|t|)
## (Intercept)       -17.845675  5.0667704 -3.522101 5.784846e-04
## I(log(gdpPercap))   8.829813  0.6625915 13.326180 1.123341e-26
```

By applying the function *plot* to the model, you can generate multiple plots to carry out model diagnostics

```
# generate plot for diagnostics
par(mfrow = c(2, 2))
plot(simple_regression)
```



- Residuals vs Fitted Plot to check the linearity assumption
- Normal Q-Q plot to check the normality assumption
- Scale-Location Plot to check the costant variance assumption
- Residuals vs Leverage for handling with outliers

Multiple Linear Model

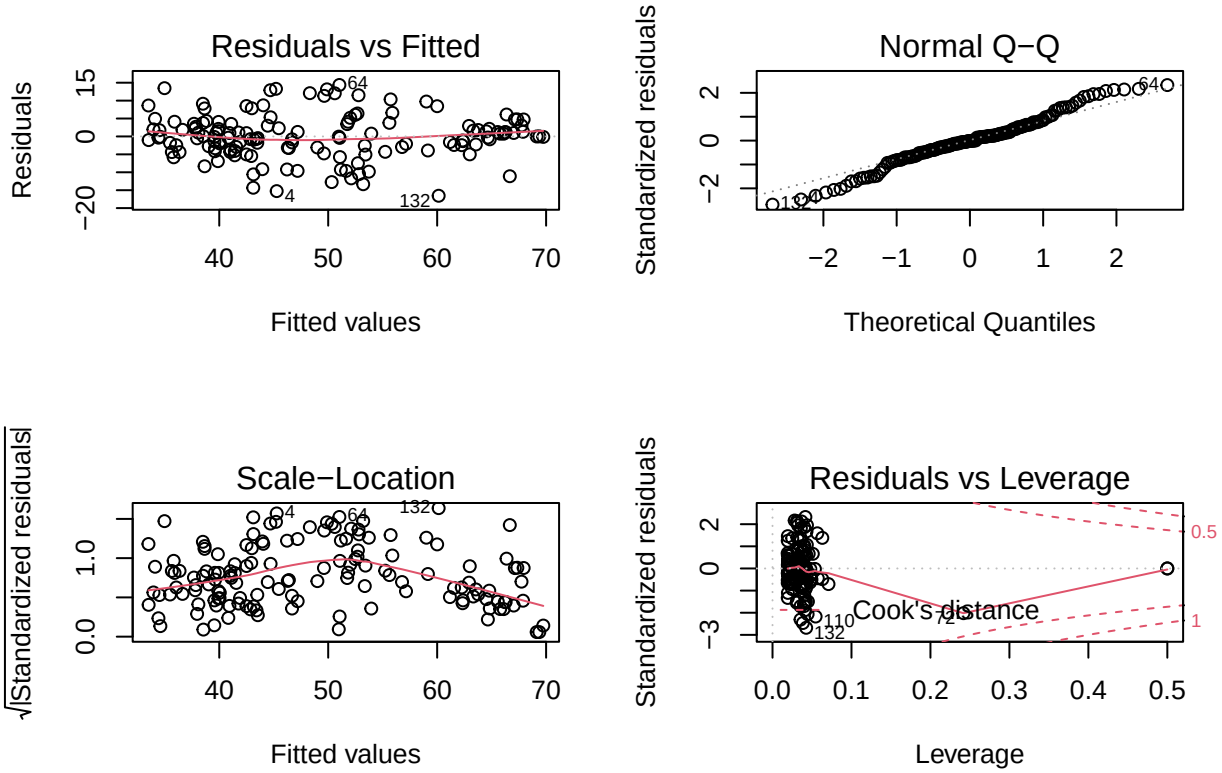
We can increase the number of regressors by adding variables on the right hand side of the *formula* argument. In this specific example, we add continent as a regressor. Please note that continent is a categorical variable. In R, if we wish to run a regression with a categorical variable as covariate, the categorical variable must be a factor. In our example, this action is not necessary as continent is of factor type. However, if continent would have not been a factor, we should have transformed it into a factor object by means of the function as.factor().

```
# run a multiple linear regression model
multiple_regression = lm(lifeExp~I(log(gdpPercap))+continent,gapminder_subset)
# show the results
summary(multiple_regression)
```

```
##
## Call:
## lm(formula = lifeExp ~ I(log(gdpPercap)) + continent, data = gapminder_subset)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -16.5537  -3.2079  -0.1173   3.4868  14.3635
##
## Coefficients:
##                    Estimate Std. Error t value Pr(>|t|)
```

```
## (Intercept)          6.3024     4.7636   1.323 0.188040
## I(log(gdpPercap))    4.7720     0.6806   7.012  1.0e-10 ***
## continentAmericas    8.1439     1.7584   4.631  8.4e-06 ***
## continentAsia        5.0429     1.4374   3.508 0.000612 ***
## continentEurope     17.6392     1.8109   9.740  < 2e-16 ***
## continentOceania    18.8623     4.8233   3.911 0.000145 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 6.312 on 136 degrees of freedom
## Multiple R-squared:  0.7429, Adjusted R-squared:  0.7335
## F-statistic:  78.6 on 5 and 136 DF,  p-value: < 2.2e-16
```

```
# diagnotistics
par(mfrow = c(2, 2))
plot(multiple_regression)
```



Dealing with categorical variables in regression models

If we wish to change the baseline category, we need to apply the function *relevel()* to the categorical variable.

First of all, we must recall that a factor variable implicitly assigns to each category a number. This number ranges from 1 up to the total number of categories of the variable. In our example, if we wish to find about the order of the discrete categories of a variable, the function *levels()* can be employed. The output of this function provides the user with the unique categories of the variable in the correct order

```
# factor
levels(gapminder_subset$continent)
```

```
## [1] "Africa"   "Americas" "Asia"     "Europe"   "Oceania"
```

In a regression model, the first level appearing in the output is employed as baseline category. If we wish to

change the reference level, we can apply the *relevel()* function to the categorical variable. This function takes two input arguments: the categorical variable itself and the new reference level (*ref*).

Suppose we would like to use "Asia" as reference level, which set to be the third level, we can proceed as follows.

```
# run a multiple linear regression model
multiple_regression_newref = lm(lifeExp~I(log(gdpPercap))+relevel(continent,ref = 3),gapminder_subset)
# show the results
summary(multiple_regression_newref)
```

```
##
## Call:
## lm(formula = lifeExp ~ I(log(gdpPercap)) + relevel(continent,
##     ref = 3), data = gapminder_subset)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -16.5537  -3.2079  -0.1173   3.4868  14.3635
##
## Coefficients:
##                                     Estimate Std. Error t value Pr(>|t|)
## (Intercept)                          11.3453     5.1067   2.222 0.027960 *
## I(log(gdpPercap))                     4.7720     0.6806   7.012 1.00e-10 ***
## relevel(continent, ref = 3)Africa    -5.0429     1.4374  -3.508 0.000612 ***
## relevel(continent, ref = 3)Americas   3.1010     1.7620   1.760 0.080668 .
## relevel(continent, ref = 3)Europe    12.5964     1.7748   7.097 6.39e-11 ***
## relevel(continent, ref = 3)Oceania   13.8194     4.7770   2.893 0.004446 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 6.312 on 136 degrees of freedom
## Multiple R-squared:  0.7429, Adjusted R-squared:  0.7335
## F-statistic:  78.6 on 5 and 136 DF,  p-value: < 2.2e-16
```

We can also change the order of the levels of the factor variables according to our preferences. This task can be achieved using the function *factor()*. The latter requires the specification of two main arguments: the factor variable to be re-leveled and a vector with categories arranged in the desired order.

```
# relevel

continents = factor(gapminder$continent,
                    levels=c("Asia","Africa","Oceania","Americas","Europe"))
levels(continents)
```

```
## [1] "Asia"     "Africa"   "Oceania"  "Americas" "Europe"
```

Model Comparison

If we wish to compare two nested models, we can use the function *anova()*. This function requires as arguments the objects in which the results of the two models are stored. The first argument has to be the model with the lowest number of indepedent variables, whereas the second must be the model with the largest number of regressors. However, the regressor(s) of the first model must be contained also in the second model.

In our example, the model *simple_regression* is nested within the model *multiple_regression* as the former does not contain the variable *continent*. We select the best model using the *anova()* function.

```r
# model comparison
anova(simple_regression,multiple_regression)
```

```
## Analysis of Variance Table
##
## Model 1: lifeExp ~ I(log(gdpPercap))
## Model 2: lifeExp ~ I(log(gdpPercap)) + continent
##   Res.Df    RSS Df Sum of Sq      F   Pr(>F)
## 1    140 9290.7
## 2    136 5418.3  4    3872.4 24.299 3.49e-15 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

To choose the best model, we should look at the p-value of the test, which is reported in the column *Pr(>F)*.

Another way to perform model selection in R is by using the AIC and BIC criteria. If we wish to compare a set of models fitted on the same sample but not necesserily nested, we should pick the model with either the lowest AIC or BIC.

In R, there are the functions *AIC()* and *BIC()* to measure the AIC or BIC of the fitted model. These functions take as main argument the fitted model in R.

```r
AIC_models = data.frame(type=c("simple","multiple"),
                        AIC = c(AIC(simple_regression),AIC(multiple_regression)))


BIC_models = data.frame(type=c("simple","multiple"),
                        BIC = c(BIC(simple_regression),BIC(multiple_regression)))


print(AIC_models)
```

```
##       type       AIC
## 1   simple 1002.6728
## 2 multiple  934.1018
```

```r
print(BIC_models)
```

```
##       type       BIC
## 1   simple 1011.5403
## 2 multiple  954.7926
```

Beyond the linear model: Logistic Regression in R

In R, we can also fit more complex statistical models. In particular, through the function *glm()* we are able to fit a wide range of models belonging to the class of the generalized linear models. Well-known models belong to this class, including the logistic regression model, the Poisson regression model as well as the multinomial regression model.

The function is quite similar to *lm()* function with some additional arguments that tell R which family of generalized linear model has to be fitted (*family*).

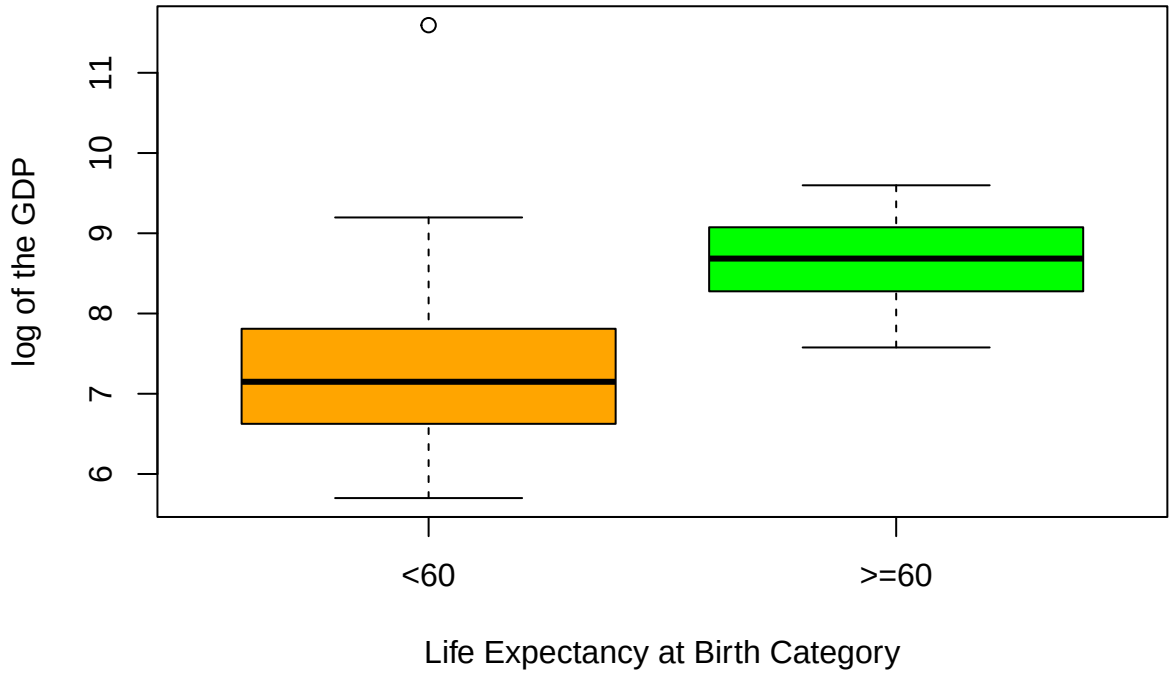
Simple Logistic Regression

In our example, we are again interested in exploring the relation ship between the life expectancy and the GDP per capita for multiple countries in 1952. For the sake of interpretation, we would like to create a binary variable based upon the life expectancy variable. In practice, we assume that countries with life expectancy

greater than or equal to 60 take the value 1. On the other hand, countries with a life expectancy lower than 60 take a value equal to 0.

As a descriptive check, we can check the distribution of the log of the GDP per Capita stratified by the categories of the new variable (< 60,≥ 60), This can be achieved visually by using box plots.

```
# create a new binary variable
gapminder_subset$lifeExp_cat = ifelse(gapminder_subset$lifeExp>=60,1,0)
# boxplots of log of GDP per capita
boxplot(log(gapminder_subset$gdpPercap) ~
          gapminder_subset$lifeExp_cat,
        col=c("orange","green"),
        names=c("<60",">=60"),
        xlab="Life Expectancy at Birth Category",
        ylab="log of the GDP")
```



We run a logistic regression model, in which the response is the binary variable created out of the life expectancy variable and the independent variable is the log of the GDP per capita. The family of this regression model is "binomial".

```
# run a simple logistic regression model
simple_logistic_reg = glm(lifeExp_cat~I(log(gdpPercap)),gapminder_subset,family="binomial")
# show the results
summary(simple_logistic_reg)
```

```
##
## Call:
## glm(formula = lifeExp_cat ~ I(log(gdpPercap)), family = "binomial",
##     data = gapminder_subset)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -3.8396  -0.5217  -0.1936  -0.0555   2.0913
##
```

```
## Coefficients:
##                   Estimate Std. Error z value Pr(>|z|)
## (Intercept)        -19.871      3.560  -5.581 2.39e-08 ***
## I(log(gdpPercap))    2.350      0.433   5.427 5.73e-08 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 158.60  on 141  degrees of freedom
## Residual deviance:  94.31  on 140  degrees of freedom
## AIC: 98.31
##
## Number of Fisher Scoring iterations: 6
```

The output of the model is similar to the output of the linear regression model. Instead of the F-test results, it reports the null deviance (variability in the response not explained by the independent variables) and the residual deviance (remaining variability in the response after accounting for the effect of the log of the GDP per capita).

It is important to stress that the coeffiecients are reported on the log-odds scale. In order to interpret the coefficients on the odds scale, we need to exponentiate them.

This is possible by applying the *exp()* function to the *coef()* object.

```
# coefficients in odds scale
exp(coef(simple_logistic_reg))
```

```
##       (Intercept) I(log(gdpPercap))
##      2.344269e-09      1.048315e+01
```

Multiple Logistic Regression

We can add other varibles

```
# run a multiple linear regression model
multiple_logistic_reg = glm(lifeExp_cat~I(log(gdpPercap))+continent,gapminder_subset,family="binomial")
# show the results
summary(multiple_logistic_reg)
```

```
##
## Call:
## glm(formula = lifeExp_cat ~ I(log(gdpPercap)) + continent, family = "binomial",
##     data = gapminder_subset)
##
## Deviance Residuals:
##      Min        1Q    Median        3Q       Max
## -2.70617  -0.36533  -0.00009  -0.00003   2.11642
##
## Coefficients:
##                   Estimate Std. Error z value Pr(>|z|)
## (Intercept)        -30.179   1390.278  -0.022  0.98268
## I(log(gdpPercap))    1.483      0.471   3.148  0.00164 **
## continentAmericas   16.816   1390.274   0.012  0.99035
## continentAsia       16.622   1390.274   0.012  0.99046
## continentEurope     18.975   1390.274   0.014  0.98911
## continentOceania    36.044   7729.586   0.005  0.99628
```

```
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 158.595  on 141  degrees of freedom
## Residual deviance:  69.417  on 136  degrees of freedom
## AIC: 81.417
##
## Number of Fisher Scoring iterations: 18
```

If we wish to compare the two models, we can issue the command *anova()* and additionally set the test to "Chi" to specify that we want to use the chi-squared test for comparing the deviances.

```
# perform model comparison
anova(simple_logistic_reg,multiple_logistic_reg,test="Chi")
```

```
## Analysis of Deviance Table
##
## Model 1: lifeExp_cat ~ I(log(gdpPercap))
## Model 2: lifeExp_cat ~ I(log(gdpPercap)) + continent
##   Resid. Df Resid. Dev Df Deviance  Pr(>Chi)
## 1       140     94.310
## 2       136     69.417  4   24.893 5.286e-05 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

References

Wickham, Hadley, Mine Çetinkaya-Rundel, and Garrett Grolemund (2016). *R for data science*. O'Reilly Media, Inc.

Wickham, H., & Bryan, J. (2023). *R packages*. O'Reilly Media, Inc.