Chapter 7: More programming in R

Contents

1 The family of `apply` functions in R

The family of `apply` functions in R allows to repetitively perform an action on multiple slices of data. The use of the `apply` function or one of its variants avoids the explicit use of loop constructs.

Each function belonging to the `apply` family requires another function as one of the arguments. This specified function will be applied on the input data.

There are so many different `apply` functions because they are meant to operate on different types of data. Not all the variants of the `apply` function will be discussed in this section.

1.1 The apply function in R

Syntax:

```
apply(X, MARGIN, FUN)
```

- `X` is an array, including a matrix
- `MARGIN` specifies how the function will be applied:
  - `MARGIN = 1`: rows
  - `MARGIN = 2`: columns
  - `MARGIN = c(1,2)`: rows and columns
- `FUN` is the function to be applied

The function `apply` returns a vector, an array or a list of values.

Example *boiler*

Use the `boiler` data (`qcc` package). It reports temperature readings from eight burners on a boiler. There are 25 observations and 8 variables

```
data(boiler)
head(boiler)
```

```
##    t1  t2  t3  t4  t5  t6  t7  t8
## 1 507 516 527 516 499 512 472 477
## 2 512 513 533 518 502 510 476 475
## 3 520 512 537 518 503 512 480 477
## 4 520 514 538 516 504 517 480 479
## 5 530 515 542 525 504 512 481 477
## 6 528 516 541 524 505 514 482 480
```

- Assume that we want to work with the log values instead of the values itself

```
ln_boiler <- apply(boiler, c(1,2), log)
head(ln_boiler)
```

```
##            t1       t2       t3       t4       t5       t6       t7       t8
## [1,] 6.228511 6.246107 6.267201 6.246107 6.212606 6.238325 6.156979 6.167516
## [2,] 6.238325 6.240276 6.278521 6.249975 6.218600 6.234411 6.165418 6.163315
## [3,] 6.253829 6.238325 6.285998 6.249975 6.220590 6.238325 6.173786 6.167516
## [4,] 6.253829 6.242223 6.287859 6.246107 6.222576 6.248043 6.173786 6.171701
## [5,] 6.272877 6.244167 6.295266 6.263398 6.222576 6.238325 6.175867 6.167516
## [6,] 6.269096 6.246107 6.293419 6.261492 6.224558 6.242223 6.177944 6.173786
```

- Compute the average temperature per row (for every observation)

```
mean_per_row <- apply(boiler, 1, mean)
mean_per_row
```

```
##  [1] 503.250 504.875 507.375 508.500 510.750 511.250 507.625 507.000 510.375
## [10] 510.250 510.500 509.875 510.750 506.250 512.125 511.250 512.625 507.125
## [19] 503.000 511.750 507.250 507.500 510.750 509.000 512.000
```

- Compute the average per column

```
mean_per_col <- apply(boiler, 2, mean)
mean_per_col
```

```
##     t1     t2     t3     t4     t5     t6     t7     t8
## 525.00 513.56 538.92 521.68 503.80 512.44 478.72 477.24
```

1.2 The tapply function in R

The function tapply is helpful while dealing with categorical variables. It allows to apply a function to numeric data distributed across various categories.

Syntax:

```
tapply(X, INDEX, FUN)
```

- X is an R object, usually a vector
- INDEX is a list of one or more factors
- FUN is the function to be applied

Example *tips*
We use the tips data (package reshape)
A waiter collected information about the amount of tip he received over a period of a few months.

```
head(tips)
```

```
##   total_bill  tip    sex smoker day   time size
## 1      16.99 1.01 Female     No Sun Dinner    2
## 2      10.34 1.66   Male     No Sun Dinner    3
## 3      21.01 3.50   Male     No Sun Dinner    3
## 4      23.68 3.31   Male     No Sun Dinner    2
## 5      24.59 3.61 Female     No Sun Dinner    4
## 6      25.29 4.71   Male     No Sun Dinner    4
```

1. Compute the average amount of tip, based on the day of the week

```
?tapply
```

```
tapply(tips$tip, tips$day, mean)
```

```
##      Fri      Sat      Sun     Thur
## 2.734737 2.993103 3.255132 2.771452
```

Remark:
In case you work with missing data, you can use tapply(tips$tip, tips$day, mean, na.rm = TRUE).

2. Compute the average amount of tip, based on the day of the week and the time of the day.

```
tapply(tips$tip, list(tips$day, tips$time), mean)
```

```
##        Dinner    Lunch
## Fri  2.940000 2.382857
## Sat  2.993103       NA
## Sun  3.255132       NA
## Thur 3.000000 2.767705
```

Remark:
The tapply function does the same as the by and the aggregate function in R.

1.3 The lapply function in R

Syntax:

```
lapply(X, FUN)
```

- X is the input data which can be a list, a vector or a data frame.
- FUN is the function to be applied.

The functionlapply returns a list of the same length as X. The specified function FUN is only applicable through columns.

Example *tips*

Because `lapply` works on a list, we first create a list of the `tips` data frame by using the `split` function. By using the `split` function, we divide a vector into groups defined by a factor.
Here we divide the total amount of the tip by day of the week

```
list_tip <- split(tips$tip, tips$day)
list_tip
```

```
## $Fri
##  [1] 3.00 3.50 1.00 4.30 3.25 4.73 4.00 1.50 3.00 1.50 2.50 3.00 2.20 3.48 1.92
## [16] 3.00 1.58 2.50 2.00
##
## $Sat
##  [1]  3.35  4.08  2.75  2.23  7.58  3.18  2.34  2.00  2.00  4.30  3.00  1.45
## [13]  2.50  3.00  2.45  3.27  3.60  2.00  3.07  2.31  5.00  2.24  3.00  1.50
## [25]  1.76  6.73  3.21  2.00  1.98  3.76  2.64  3.15  2.47  1.00  2.01  2.09
## [37]  1.97  3.00  3.14  5.00  2.20  1.25  3.08  2.50  3.48  4.08  1.64  4.06
## [49]  4.29  3.76  4.00  3.00  1.00  1.61  2.00 10.00  3.16  3.41  3.00  2.03
## [61]  2.23  2.00  5.16  9.00  2.50  6.50  1.10  3.00  1.50  1.44  3.09  3.00
## [73]  2.72  2.88  2.00  3.00  3.39  1.47  3.00  1.25  1.00  1.17  4.67  5.92
## [85]  2.00  2.00  1.75
##
## $Sun
##  [1] 1.01 1.66 3.50 3.31 3.61 4.71 2.00 3.12 1.96 3.23 1.71 5.00 1.57 3.00 3.02
## [16] 3.92 1.67 3.71 3.50 2.54 3.06 1.32 5.60 3.00 5.00 6.00 2.05 3.00 2.50 2.60
## [31] 5.20 1.56 4.34 3.51 4.00 2.55 4.00 3.50 5.07 2.50 2.00 2.74 2.00 2.00 5.14
## [46] 5.00 3.75 2.61 2.00 3.50 2.50 2.00 2.00 3.00 3.48 2.24 4.50 5.15 3.18 4.00
## [61] 3.11 2.00 2.00 4.00 3.55 3.68 5.65 3.50 6.50 3.00 5.00 3.50 2.00 3.50 4.00
## [76] 1.50
##
## $Thur
##  [1] 4.00 3.00 2.71 3.00 3.40 1.83 5.00 2.03 5.17 2.00 4.00 5.85 3.00 1.50 1.80
## [16] 2.92 2.31 1.68 2.50 2.00 2.52 4.20 1.48 2.00 2.00 2.18 1.50 2.83 1.50 2.00
## [31] 3.25 1.25 2.00 2.00 2.00 2.75 3.50 6.70 5.00 5.00 2.30 1.50 1.36 1.63 1.73
## [46] 2.00 4.19 2.56 2.02 4.00 1.44 2.00 5.00 2.00 2.00 4.00 2.01 2.00 2.50 4.00
## [61] 3.23 3.00
```

Now we want to compute the average tip per day of the week on this list.

```
lapply(list_tip, mean)
```

```
## $Fri
## [1] 2.734737
##
## $Sat
## [1] 2.993103
##
## $Sun
## [1] 3.255132
##
## $Thur
## [1] 2.771452
```

Remark:

We have the same result as with the `tapply` function earlier.

2 Loops in R

Remark:
If you can omit an R loop, do not use loops!

2.1 For loop in R

Syntax:

```
for(val in sequence){statement}
```

Example *tips* 1
In this example, `tips` data is used.

```
head(tips, n=3)
```

```
##   total_bill  tip    sex smoker day   time size
## 1      16.99 1.01 Female     No Sun Dinner    2
## 2      10.34 1.66   Male     No Sun Dinner    3
## 3      21.01 3.50   Male     No Sun Dinner    3
```

Count the number of reservations with even number of people at the table. Use the variable `size`.

```
sum_even <- 0
for (val in tips$size) {
  if (val %% 2 == 0) sum_even = sum_even + 1
}
# %% is modulo division in R
sum_even
```

```
## [1] 197
```

Example *tips* 2
In this example, `tips` data is used.
Assume that we want to create a new variable which is the total of the bill and the tip.

The best way to do this in R is

```
tips$total <- tips$total_bill + tips$tip
head(tips$total)
```

```
## [1] 18.00 12.00 24.51 26.99 28.20 30.00
```

Another way to do it with a for loop is

```
tips$total <- 0
for (i in 1:length(tips$total_bill)) {
  tips$total[i] <- tips$total_bill[i] + tips$tip[i]
}
head(tips)
```

```
##   total_bill  tip    sex smoker day   time size total
## 1      16.99 1.01 Female     No Sun Dinner    2 18.00
## 2      10.34 1.66   Male     No Sun Dinner    3 12.00
## 3      21.01 3.50   Male     No Sun Dinner    3 24.51
## 4      23.68 3.31   Male     No Sun Dinner    2 26.99
## 5      24.59 3.61 Female     No Sun Dinner    4 28.20
## 6      25.29 4.71   Male     No Sun Dinner    4 30.00
```

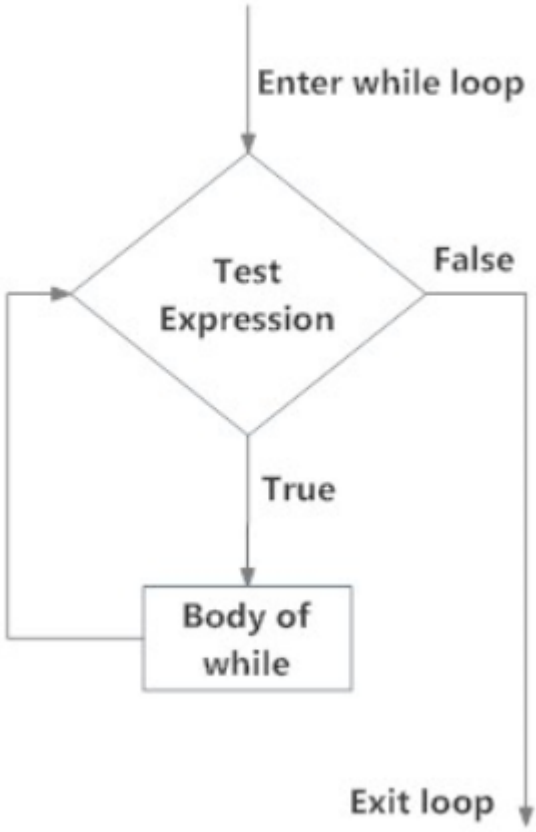
Remark:

1. The number of iterations in a for loop is fixed and known in advance.
2. If you can avoid loops, then do not use loops.

2.2 While loop in R

Syntax:

```
while(test_expression){statement}
```



Example 1

We want to print the values from 2 to 6

```
i <- 1
while(i<6){
  i = i+1
  print(i)
}
```

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

Example 2

`break` will end the loop abruptly.

```r
i <- 1
while(i<6){
  i = i+1
  if (i==4) break
  print(i)
}
```

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

Example 3

`next` can skip one step of the loop

```r
i <- 1
while(i<6){
  i = i+1
  if (i==4) next
  print(i)
}
```

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

2.3 Repeat loop in R

Syntax:

```
repeat{statement}
```

Example

```r
i <- 1
repeat{
  print(i)
  i = i+1
  if (i==6){break}
}
```

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

Remark:

Be careful with the `repeat` function, ensure that there is a termination of the loop otherwise you might have an infinite loop.

3 *If then else* in R

Syntax:

```
if(test_expression){statement}
```

```
if(test_expression){statement1} else {statement2}
```

Example 1

```
x <- -5
if(x>0){
  print("Non-negative number")
} else {
  print("Negative number")
}
```

```
## [1] "Negative number"
```

4 Examples of writing functions in R

4.1 General example

Write a function `Fn(vec)` which computes a vector of moving averages of width 3. $\frac{x_1+x_2+x_3}{3}$, $\frac{x_2+x_3+x_4}{3}$, ..., $\frac{x_{n-2}+x_{n-1}+x_n}{3}$.

When your original vector has length n, then the vector of moving averages will have length $n-2$.
Apply your function on the vector `1:6`.

```
Fn <- function(vec)
{
  for(i in 3:length(vec))
  {x[i] <- (vec[i-2] + vec[i-1] + vec[i])/3
  print(x[i])}
}
z <- 1:6
Fn(z)
```

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

4.2 Illustration of the Central Limit Theorem (use of `for` loop and `apply` functions)

4.2.1 Description of the CLT illustration

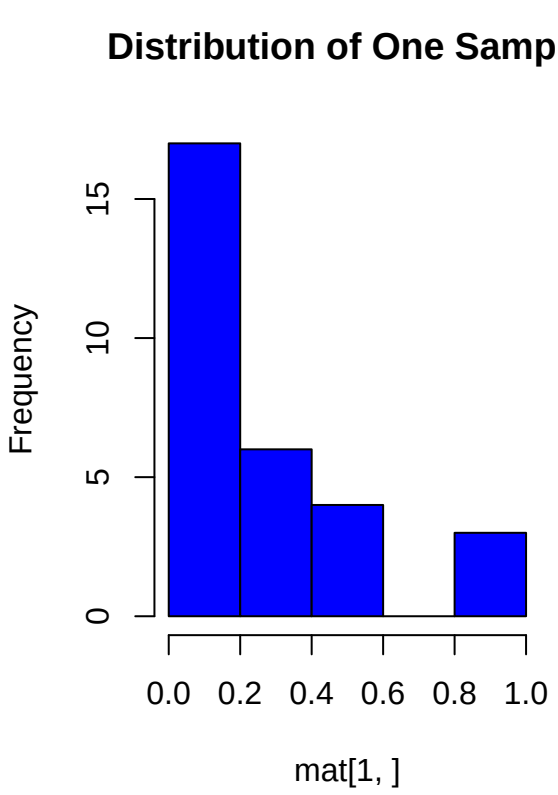
Make a visualization of the central limit theorem (CLT).
The **CLT** says that if you take samples from a distribution and compute the average. Then these sample averages have a normal distribution, no matter the distribution of the original population as long as the sample size is large enough.

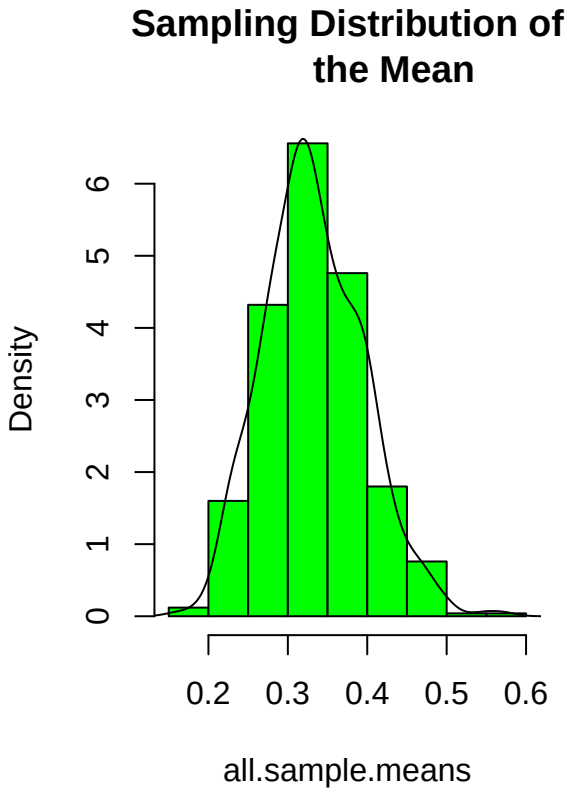
- Step 1: Generate 30 (=`n`) data points from an *exponential distribution* with rate 3. (hint: use the function `rexp`).
- Step 2: Do this now 5 times. Consider every sample as one new line in a data matrix `mat`. Hence `mat` will be a 5×30 matrix.
- Step 3: Compute a vector `all.samples.means` which has the averages for every sample. This vector has a length of 5.
- Step 4: Make two histograms next to each other (in one and the same graphical window).
  a) The first histogram is a frequency histogram of the first sample (the data from step 1).
  b) The second histogram is a relative frequency histogram of the sample averages, overlayed with the corresponding density curve.
- Step 5: Create now a function which is producing the previous steps (2-4) and has as parameters: `n` (number of data points, default 30), `rpt` (number of samples to take, default 5).
- Step 6: Apply this function when `n = 30` and `rpt = 500`.

4.2.2 Solution of the CLT illustration

Generate a matrix with rows and columns each row is one sample. One sample consists of 30 data points generated from an exponential distribution with rate 3.

```
# Step 1
rexp(30, rate=3)

# Step 2
mat <- matrix(rep(0,150), nrow=5)
for (i in (1:5))
{
mat[i,] <- rexp(30, rate=3)
}

# Step 3
# compute the average for every sample
all.sample.means <- apply(mat,1,mean)

# Step 4
# create a histogram with the original data of 1st row
#and another histogram with the averages
par(mfrow=c(1,2))
hist(mat[1,],col="blue", main="Distribution of One Sample")
hist(all.sample.means, col="green", main="Sampling Distribution of
     the Mean", prob=T)
lines(density(all.sample.means))


# Step 5
# create a function out of this
clt_fun <- function(rpt = 5, n=30)
{
  mat <- matrix(rep(0,n*rpt), nrow=rpt)
    for (i in (1:rpt))
        {
      mat[i,] <- rexp(n, rate=3)
        }

  # compute the average for every sample
  all.sample.means <- apply(mat,1,mean)

  # create a histogram with the original data for 1st row
  #and another histogram with the averages
  par(mfrow=c(1,2))
  hist(mat[1,],col="blue", main="Distribution of One Sample")
  hist(all.sample.means, col="green", main="Sampling Distribution of
         the Mean", prob=T)
  lines(density(all.sample.means))
}

# Step 6
clt_fun(rpt=500,n=30)
```





4.3 Reuse tidyverse code

Example 1:

```
# example 1: compute grouped average delay for flights
 flights %>%
  group_by( month) %>%
  summarise(n=n(), Avg = mean(dep_delay, na.rm = T)) %>%
  arrange(Avg)


# example 2: compute average tip by day
tips %>%
  group_by(day) %>%
  summarise(n=n(), Avg = mean(tip)) %>%
  arrange(Avg)
```

A possible function can be written as :

```
group_mean <- function(data, var, by) {
  data %>%
    group_by(by) %>%
    summarise(average = mean(var, na.rm = TRUE)) %>%
    arrange(average)
}
group_mean(data=tips, var=tip, by=day)
```

The statement above is not working. We need to make some adaptations in the function. One possibility is to make use of the embracing operator (package rlang).

```
# we need to make some adaptations
library(rlang)

# use embracing operator {{ }}
group_mean2 <- function(data, var, by) {
  data %>%
    group_by({{ by }}) %>%
    summarise(average = mean({{ var }}, na.rm = TRUE))%>%
    arrange(average)
}
# now we can reuse dplyr code
group_mean2(data=tips, var=tip, by=day)
```

```
## # A tibble: 4 x 2
##   day   average
##   <fct>   <dbl>
## 1 Fri      2.73
## 2 Thur     2.77
## 3 Sat      2.99
## 4 Sun      3.26
```

5 Dates

Here, we use the functions from the `lubridate` package

```
library(lubridate)
library(nycflights13)
library(ggplot2)
```

5.1 Create date from strings

Current date:

```
today()
```

R starts counting from 1 January 1970
Internally, `Date` objects are stored as the number of days since January 1, 1970, using negative numbers for earlier dates.

This is a vector:

```
vec <- c("1970-01-01", "2020-01-31", "2020-02-01") # This is a vector
```

This is a date:

```
dates <- ymd(c("1970-01-01", "2020-01-31", "2020-02-01")) # This is a date
as.numeric(dates)
```

```
## [1]     0 18292 18293
```

5.2 Create date from individual components

```
head(flights)
```

```
## # A tibble: 6 x 19
##    year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
## 1  2013     1     1      517            515         2      830            819
## 2  2013     1     1      533            529         4      850            830
## 3  2013     1     1      542            540         2      923            850
## 4  2013     1     1      544            545        -1     1004           1022
## 5  2013     1     1      554            600        -6      812            837
## 6  2013     1     1      554            558        -4      740            728
## # ... with 11 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
## #   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
## #   hour <dbl>, minute <dbl>, time_hour <dttm>
```

To create a date from year, month and day:

```
flights_date <- flights %>%
  select(year, month, day) %>%
  mutate(dep_date = make_date(year, month, day))
head(flights_date)
```

```
## # A tibble: 6 x 4
##    year month   day dep_date
##   <int> <int> <int> <date>
## 1  2013     1     1 2013-01-01
## 2  2013     1     1 2013-01-01
## 3  2013     1     1 2013-01-01
## 4  2013     1     1 2013-01-01
## 5  2013     1     1 2013-01-01
## 6  2013     1     1 2013-01-01
```

`make_date` produces objects of class `Date`.

5.3 How to compare to a fixed date?

We now want e.g. to select the flights with departure date on January 22, 2013.

```
sub <- flights_date %>%
  filter(dep_date == ymd(20130122))
head(sub)
```

```
## # A tibble: 6 x 4
##    year month   day dep_date
##   <int> <int> <int> <date>
## 1  2013     1    22 2013-01-22
## 2  2013     1    22 2013-01-22
## 3  2013     1    22 2013-01-22
## 4  2013     1    22 2013-01-22
## 5  2013     1    22 2013-01-22
## 6  2013     1    22 2013-01-22
```

5.4 Once you have a date, you can get components

```
head(economics, n=3)
```

```
## # A tibble: 3 x 6
##   date         pce    pop psavert uempmed unemploy
##   <date>     <dbl>  <dbl>   <dbl>   <dbl>    <dbl>
```

```
## 1 1967-07-01  507. 198712    12.6     4.5     2944
## 2 1967-08-01  510. 198911    12.6     4.7     2945
## 3 1967-09-01  516. 199113    11.9     4.6     2958
```

- `year()` and `month()` returns respectively the years and month component of a date-time as a decimal number.
- `mday()` returns the day of the month.
- `wday()` returns the day of the week as a decimal number or an ordered factor if label is `TRUE`.

```
econ <- economics
econ2 <- econ %>%
  select(date) %>%
  mutate(date_Y = year(date), date_M = month(date), date_D = mday(date),
         date_wkd = wday(date, label = TRUE))
head(econ2)
```

```
## # A tibble: 6 x 5
##   date       date_Y date_M date_D date_wkd
##   <date>      <dbl>  <dbl>  <int> <ord>
## 1 1967-07-01   1967      7      1 za
## 2 1967-08-01   1967      8      1 di
## 3 1967-09-01   1967      9      1 vr
## 4 1967-10-01   1967     10      1 zo
## 5 1967-11-01   1967     11      1 wo
## 6 1967-12-01   1967     12      1 vr
```

6 Spreading and gathering

6.1 Gathering: make a long table

The function `pivot_longer()` "lengthens" data, increasing the number of rows and decreasing the number of columns.

```
# Usage of pivot_longer (see help page):
pivot_longer(
  data,
  cols,
  names_to = "name",
  values_to = "value",
    ...
)
```

```
library(tidyr)
table4a
```

```
## # A tibble: 3 x 3
##   country     `1999` `2000`
## * <chr>        <int>  <int>
## 1 Afghanistan    745   2666
## 2 Brazil       37737  80488
## 3 China       212258 213766
```

```
table_long <- pivot_longer(table4a, 2:3, names_to = "year", values_to = "cases")
table_long
```

```
## # A tibble: 6 x 3
##   country     year   cases
##   <chr>       <chr>  <int>
## 1 Afghanistan 1999     745
## 2 Afghanistan 2000    2666
## 3 Brazil      1999   37737
## 4 Brazil      2000   80488
## 5 China       1999  212258
## 6 China       2000  213766
```

6.2 Spreading: make a wide table

`pivot_wider()` "widens" data, increasing the number of columns and decreasing the number of rows.

```
# Usage of pivot_wider (see help page):
pivot_wider(
  data,
  id_cols = NULL,
  names_from = name,
  values_from = value,
    ...
)
```

```
table2
```

```
## # A tibble: 12 x 4
##    country      year type            count
##    <chr>       <int> <chr>           <int>
##  1 Afghanistan  1999 cases             745
##  2 Afghanistan  1999 population   19987071
##  3 Afghanistan  2000 cases            2666
##  4 Afghanistan  2000 population   20595360
##  5 Brazil       1999 cases           37737
##  6 Brazil       1999 population  172006362
##  7 Brazil       2000 cases           80488
##  8 Brazil       2000 population  174504898
##  9 China        1999 cases          212258
## 10 China        1999 population 1272915272
## 11 China        2000 cases          213766
## 12 China        2000 population 1280428583
```

```
table_wide <- pivot_wider(table2, id_cols=1,names_from=c(year,type),  values_from=count)
table_wide
```

```
## # A tibble: 3 x 5
##   country     `1999_cases` `1999_population` `2000_cases` `2000_population`
##   <chr>              <int>             <int>        <int>             <int>
## 1 Afghanistan          745          19987071         2666          20595360
## 2 Brazil             37737         172006362        80488         174504898
## 3 China             212258        1272915272       213766        1280428583
```

7 Exercises

7.1 Exercise 1

Write a method with a **while loop** to print `nr 1` through `nr n-1`. For example if `n = 6`, we have:

```
## [1] "nr 1"
## [1] "nr 2"
## [1] "nr 3"
## [1] "nr 4"
## [1] "nr 5"
```

7.2 Exercise 2

Write a method with a **while loop** that computes the sum of first n positive integers:
sum = 1 + 2 + 3 + ... + n
Example: for n = 5, sum = 15

7.3 Exercise 3

Use a **nested while loop** to produce the following output

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

7.4 Exercise 4

(1) Use the **data `billboard`** from package `tidyr` for this exercise. This data set contains the song rankings for billboard top 100 in the year 2000.

- `date.enter` is the date the song entered the top 100
- `wk1`, `wk2`, ..., `wk76` is the rank of the song in each week after it entered the top 100
- `artist` and `track` are respectively the artist name and song name.

a. Create the variables `Year`, `Month` and `Day_nr` that correspond to the year, month and day of the month of the entering date. Select from `billboard_date` only the created variables and the variables `artist`, `wk1`, `wk2`, `wk3` and `wk4`. The name of the new data set is `billboard_date`.

```
head(billboard_date, n=8)
```

```
## # A tibble: 8 x 8
##   artist         wk1   wk2   wk3   wk4  Year Month Day_nr
##   <chr>        <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>  <int>
## 1 2 Pac           87    82    72    77  2000     2     26
## 2 2Ge+her         91    87    92    NA  2000     9      2
## 3 3 Doors Down    81    70    68    67  2000     4      8
## 4 3 Doors Down    76    76    72    69  2000    10     21
## 5 504 Boyz        57    34    25    17  2000     4     15
## 6 98^0            51    39    34    26  2000     8     19
## 7 A*Teens         97    97    96    95  2000     7      8
## 8 Aaliyah         84    62    51    41  2000     1     29
```

b. Create from `billboard_date` a data set `b_billboard`that looks as follows. Compare the dimensions of `billboard_date` and `b_billboard`.

```
head(b_billboard, n=10)
```

```
## # A tibble: 10 x 6
##    artist        Year Month Day_nr Week   Rank
```

```
##    <chr>        <dbl> <dbl>  <int> <chr> <dbl>
##  1 2 Pac         2000     2     26 wk1      87
##  2 2 Pac         2000     2     26 wk2      82
##  3 2 Pac         2000     2     26 wk3      72
##  4 2 Pac         2000     2     26 wk4      77
##  5 2Ge+her       2000     9      2 wk1      91
##  6 2Ge+her       2000     9      2 wk2      87
##  7 2Ge+her       2000     9      2 wk3      92
##  8 2Ge+her       2000     9      2 wk4      NA
##  9 3 Doors Down  2000     4      8 wk1      81
## 10 3 Doors Down  2000     4      8 wk2      70
```

(2) Use the **data `us_rent_income`** from package `tidyr`. Make this data set more wide by increasing the number of columns. Both the column `estimate` and the column `moe` should have a separate column for each possible level of the column `variable`. The new data set should look like this:

```
## # A tibble: 10 x 6
##    GEOID NAME                 estimate_income estimate_rent moe_income moe_rent
##    <chr> <chr>                          <dbl>         <dbl>      <dbl>    <dbl>
##  1 01    Alabama                        24476           747        136        3
##  2 02    Alaska                         32940          1200        508       13
##  3 04    Arizona                        27517           972        148        4
##  4 05    Arkansas                       23789           709        165        5
##  5 06    California                     29454          1358        109        3
##  6 08    Colorado                       32401          1125        109        5
##  7 09    Connecticut                    35326          1123        195        5
##  8 10    Delaware                       31560          1076        247       10
##  9 11    District of Columbia           43198          1424        681       17
## 10 12    Florida                        25952          1077         70        3
```

7.5 Exercise 5

Use the `boiler` data frame of the `qcc` package. We have 25 time points ($i = 1, 2, ..., 25$) and at every time point we observe one measurement `t1` (this is `xi`). (We do not use the other variables `t2`, `t3`, ... `t8`). We are going to construct a moving range and individual chart for this data in the following way.
Here some background information:

Moving range chart

- What is plotted? $mr_i = |x_i - x_{i-1}|, for\ i = 2, 3, ...$
- What is center line? $\overline{mr}$ = average of the mr_i for the first 20 time points
- Control limits: $LCL = D_3 \cdot \overline{mr}$ and $UCL = D_4 \cdot \overline{mr}$, with $D_3 = 0$ and $D_4 = 3.267$

Individual chart

- What is plotted? x_i = measurement at time point i
- What is center line? $\overline{x}$ = sample mean of the measurements for the 20 first time points.
- Control limits: $LCL = \overline{x} - E_2 \cdot \overline{mr}$ and $UCL = \overline{x} + E_2 \cdot \overline{mr}$ with $E_2 = 2.66$

Questions

a) Create `total` data frame with three columns as is given below: time point, measurement x and moving range mr.

```
head(total, n=6)
```

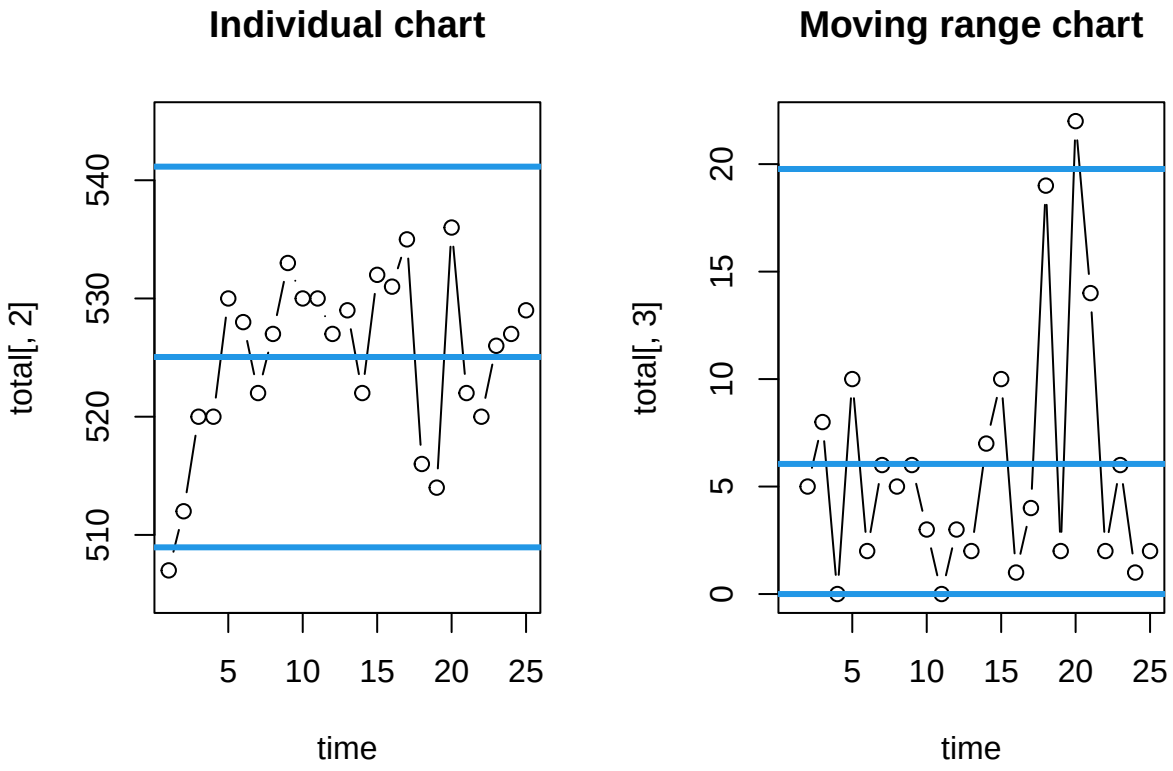
```
##      time   x mr
## [1,]    1 507 NA
## [2,]    2 512  5
```

```
## [3,]    3 520  8
## [4,]    4 520  0
## [5,]    5 530 10
## [6,]    6 528  2
```

b) Compute $\overline{mr}$, the average of the mr for the first 20 time points, and $\overline{x}$, the average measurement for the first 20 time points.

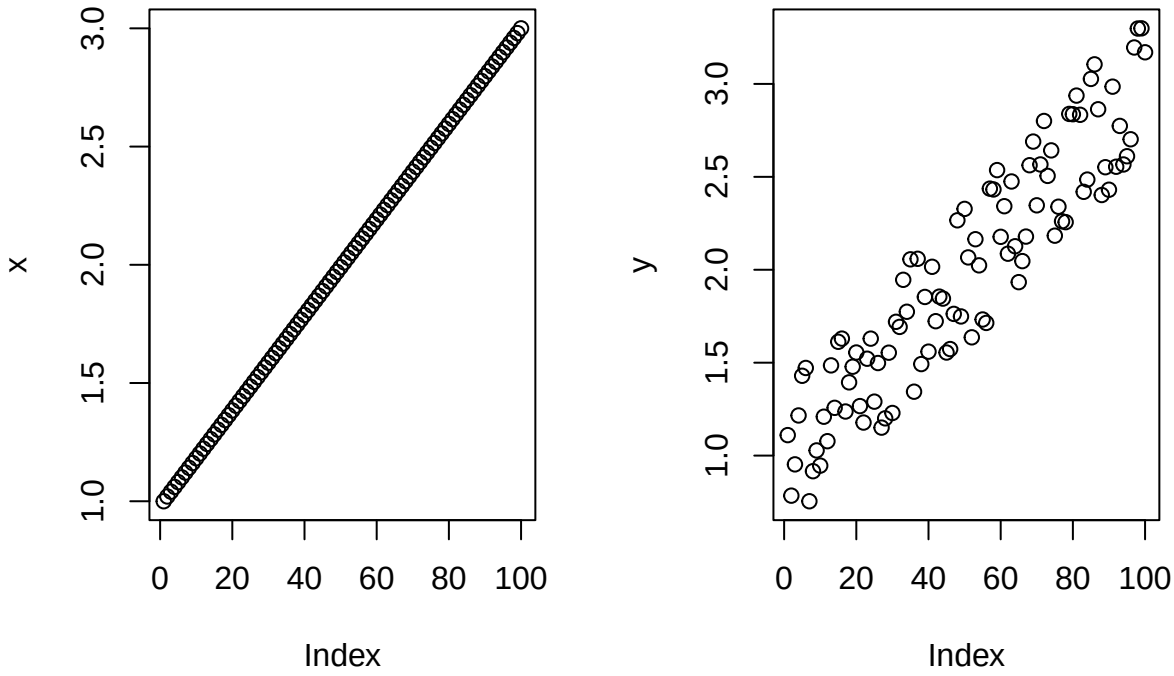
c) Compute the corresponding control limits by using the above formulas.

d) Make the moving range plot and the individual measurement plot as given below:

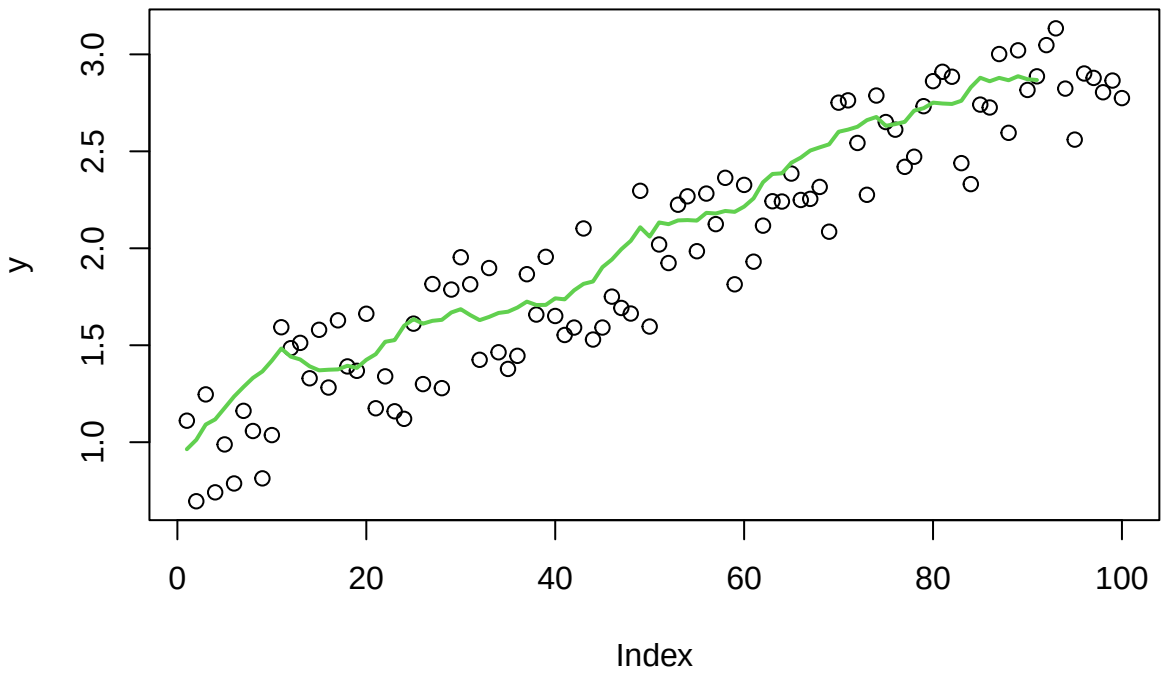


7.6 Exercise on rolling average to detect trends in your data

- Step 1: Generate sequence of data between 1 and 3 of total length 100. Use the `jitter` function (with a large factor) to add noise to your data. This is the vector y.



- Step 2: Compute the vector of rolling averages `roll.mean` with the average of 5 consecutive points. This vector has only 96 averages.
- Step 3: Add the vector of these averages to your plot.
- Step 4: Generalize step 2 and step 3 by making a function with parameters `consec` (default = 5) and `y`.
- Step 5: Apply your function to rolling averages of 10 consecutive points.


y
1.0
1.5
2.0
2.5
3.0
0
20
40
60
80
100
Index