NTNU

Introduction to R, a programming language for Statistical Computing

Erik Svendsmark
9-16-2021

Table of Contents

1 Why use R

R is a commonly used programming language in Computer Science. While it is possible to do much of the same computation in other programming languages (like Python), R has some great benefits.

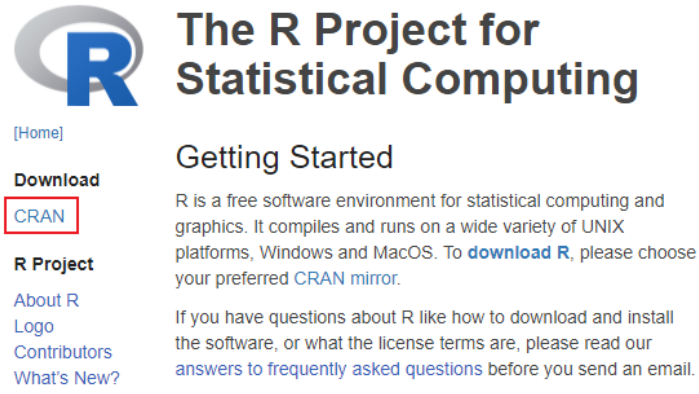
The benefits of R include:

- Free of charge and open source
- Highly flexible and provides many statistical and graphical techniques
- Optimized for vector operations
    - Great for going through a row or table
- Many resources online to find help
- Many available packages
    - Packages add new functions to R
        - **Base-packages:**
          Installed with R, but not loaded by default
        - **Contributed (3rd party)-packages:**
          Need to be downloaded, installed, and loaded separately

2 How to download R and RStudio

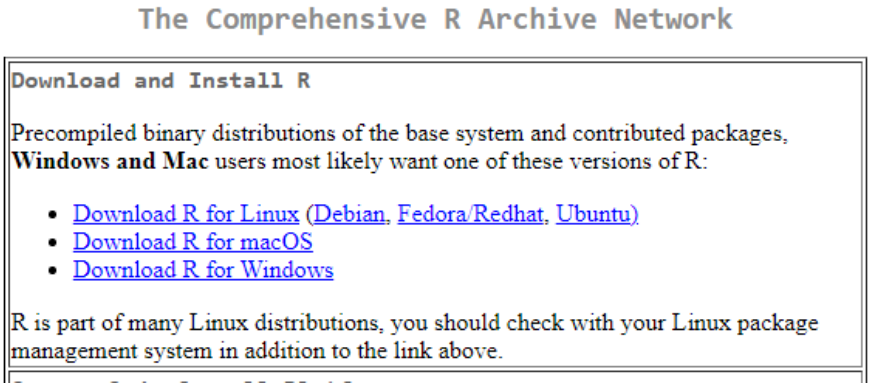
To get started with R, follow the steps below:

1. Go to the website: https://www.r-project.org/
2. Under the Download menu on the left, click CRAN.

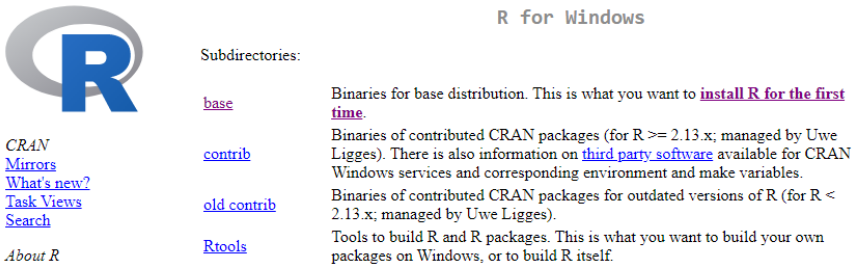


3. Click the link under "0-Cloud" as shown on the screenshot below.

4. Click the link that suits your computer's operating system.



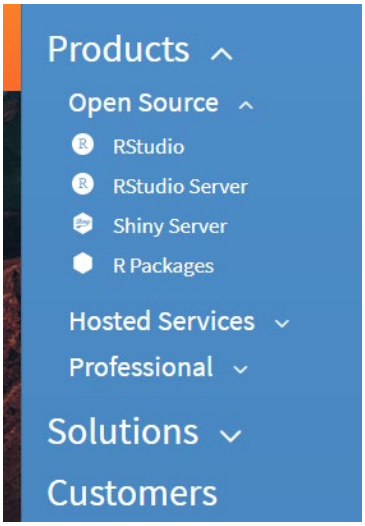
5. Start the download:
   a. **Mac**
      Download the newest release.
   b. **Windows**
      Click "base" and download the newest release.



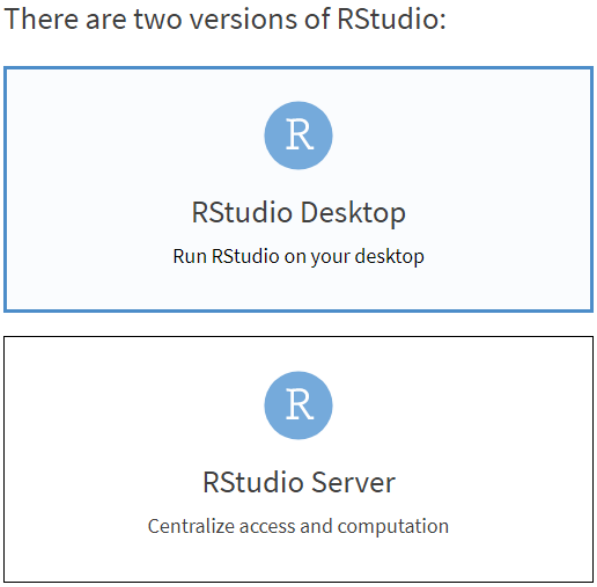
6. Follow the standard downloading steps, the download should only take a minute.

The next thing you should do, is to download an integrated development environment (IDE) for R, called **RStudio**.

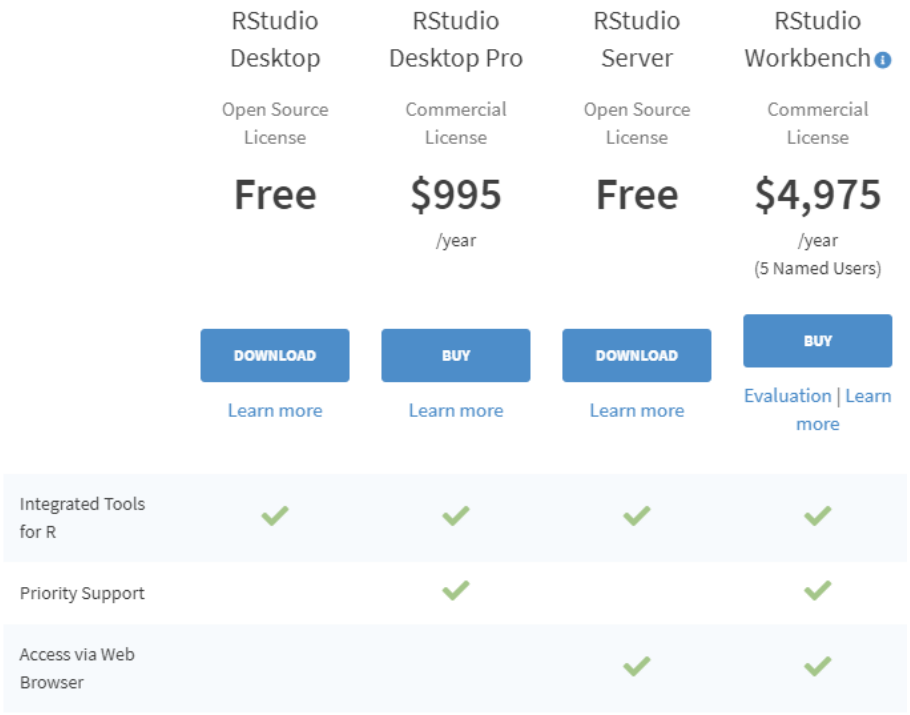
1. Go to the website: https://www.rstudio.com/
2. Navigate to the download page via: Products -> Open Source -> RStudio



3. Chose the desktop version.

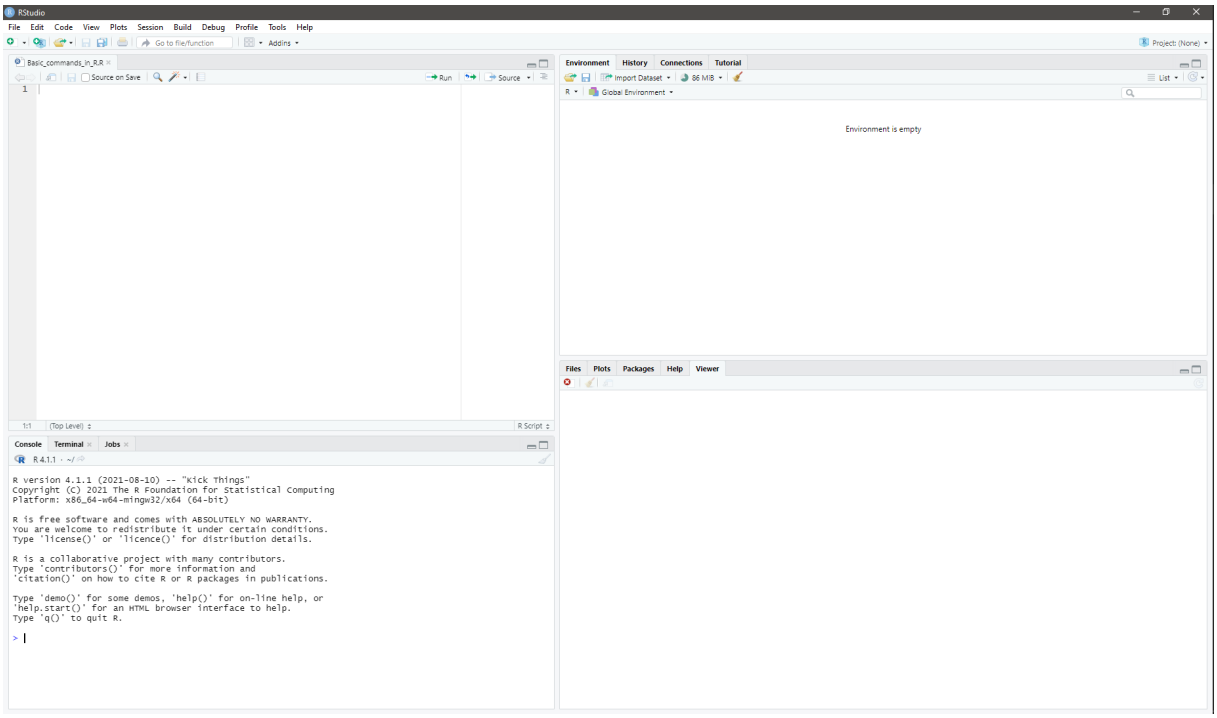


4. Download the free version.



5. Follow the standard downloading steps.

3 How to use RStudio



After opening the RStudio program, we get this window. As you can see, the main window is divided into four smaller windows.

Upper left window: "Source"

This window allows us to open, edit and run R-scripts. When writing a script, remember to save it with File -> Save.

It also displays data sets that are imported to R, like an Excel file.

Lower left window: "Console"

The console allows us to run R code one line at a time and execute our command when we hit Enter. The output (if any) will be displayed in the next line, every row of the output indexed by [1], [2], …

Tip: To clear the console, use the shortcut Ctrl + L.

You can also find this option in Edit -> Clear Console.

Upper right window: "Environment, History, Connections"

The *Environment* tab shows the objects (data and variables) that have been loaded or created in the workspace you are working on.

The *History* tab is a log of executed commands.

Lower right window: "Files, Plots, Packages, Help, Viewer"

The *Files* tab shows your current directory. It can be useful to create a dedicated folder that you do your work in. To change directory, use Session -> Set Working Directory -> Choose Directory…

This can also be done directly in the console with the command: **setwd("<name of the directory>")**, for instance:

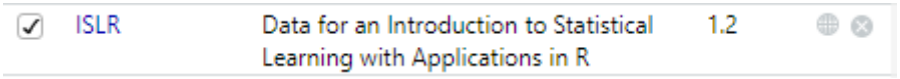
```
> setwd("C:/Users/User123")
```

If you want to change the default working directory, use Tools -> Global Options… and change the directory from there.

The *Plots* tab are where the plots will be shown when you create them.

The *Packages* tab shows what libraries are installed and loaded into the current session memory. You

can load packaged by ticking off the box to the left.



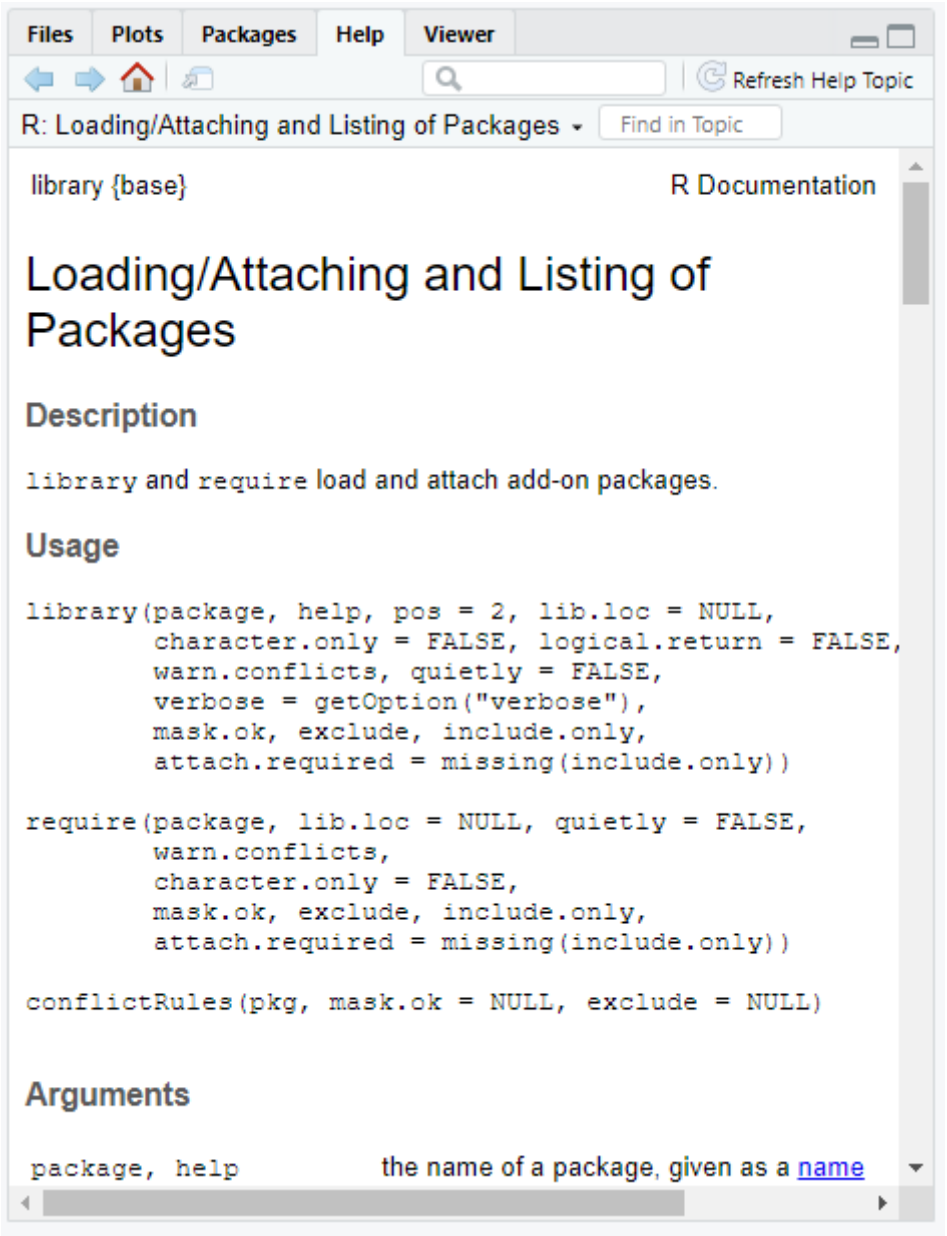
Alternatively, you can load them in the console with the command:

```
> library(ISLR)
```

The *Help* tab is used for instance when you want to look up the documentation of a function, which you can always do from the console with the function **?<function name>**.

For instance:

```
> ?library
```



4 Basic commands

4.1 How to use packages

Libraries are groups of functions and data sets, and they must be loaded before you can use them. Some libraries already got downloaded when you installed R and RStudio, and can be loaded with the function **library()**:

```
> library(MASS)
```

The library "MASS" contains a lot of data sets and useful functions.

Other libraries must first be installed (the first time you ever use them), and then loaded when you want to use them. Here I want to load the ISLR library, which contains data sets associated with the book "An introduction to statistical learning".

```
> library(ISLR)
Error in library(ISLR) : there is no package called 'ISLR'
```

The error message implies that the package ISLR is not downloaded, so let's do that:

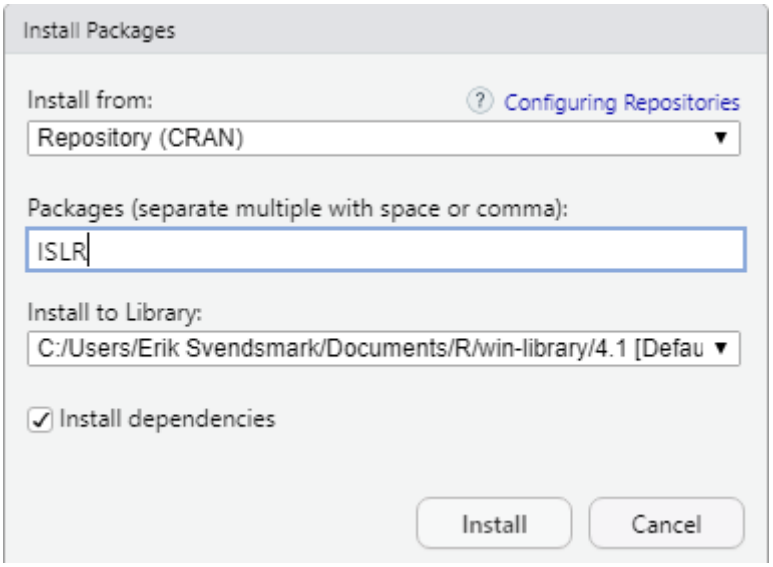
```
> install.packages("ISLR")
Installing package into 'C:/Users/Erik Svendsmark/Documents/R/win-library/4.1'
(as 'lib' is unspecified)
trying URL 'https://cran.rstudio.com/bin/windows/contrib/4.1/ISLR_1.2.zip'
Content type 'application/zip' length 2924364 bytes (2.8 MB)
downloaded 2.8 MB

package 'ISLR' successfully unpacked and MD5 sums checked

The downloaded binary packages are in
        C:\Users\Erik Svendsmark\AppData\Local\Temp\RtmpMnVP5b\downloaded_packages
> library(ISLR)
```

Now you can load the library as usual.

Alternatively, you can use Tools -> Install Packages and enter the packages you want to install.



Since you likely will be using Excel together with R, you should try to install the packages that contains functions to read data from Excel sheets ('.xls' and '.xlsx' files).

```
> install.packages("readxl")
```

NB: The **library()** function must be executed every time you want to use a given package. You can check if a package is loaded in the *Packages* tab in the lower right window.

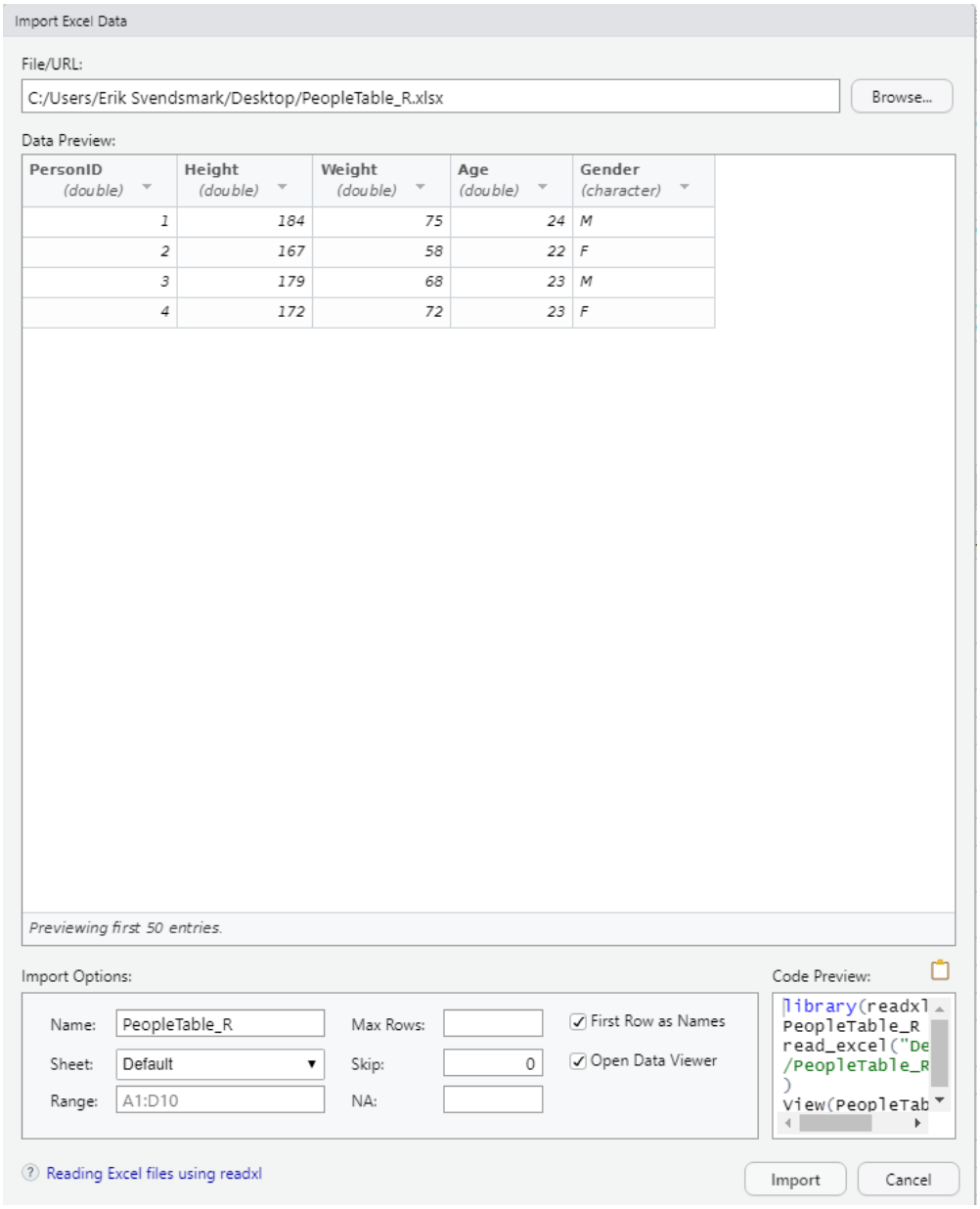
When you are done with R, you can detach the packages with the function **detach()**, or by unchecking the package in the *Packages* tab.

```
> detach(package:ISLR, unload=TRUE)
```

4.2 Import data from Excel

The easiest way to import a dataset from Excel, is by File -> Import Dataset -> From Excel…

You will get this window, where you click Browse… and find the file you want to import. Then click Import at the lower right to execute the import.



Note that the datatypes of the variables (for instance the values in the Height column are interpreted as Double) can be changed from this window, using the drop-down menu.

The lower right sub window (Code Preview) shows the corresponding R code to import the data set, and you can note that the actions we take simply corresponds to code that could be directly written in the console:

```
> library(readxl)
> PeopleTable_R <- read_excel("Desktop/PeopleTable_R.xlsx")
> View(PeopleTable_R)
```

NB: Note that on line two the operator '<-' is used. This has the same effect as using '='.

4.3 Data description

After loading the data in section 4.2, you can find a description of the dataset in the *Environment* tab.

PeopleTable_R	4 obs. of 5 variables
$ PersonID:	num [1:4] 1 2 3 4
$ Height :	num [1:4] 184 167 179 172
$ Weight :	num [1:4] 75 58 68 72
$ Age :	num [1:4] 24 22 23 23
$ Gender :	chr [1:4] "M" "F" "M" "F"

To find the dimensions of the table, use the function **dim()**:

```
> dim(PeopleTable_R)
[1] 4 5
```

The table has 4 rows and 5 columns.

To get some basic summary statistics of the dataset, we can run the function **summary()** with the dataset as it's parameter (input variable).

```
> summary(PeopleTable_R)
    PersonID        Height          Weight          Age           Gender
 Min.   :1.00   Min.   :167.0   Min.   :58.00   Min.   :22.00   Length:4
 1st Qu.:1.75   1st Qu.:170.8   1st Qu.:65.50   1st Qu.:22.75   Class :character
 Median :2.50   Median :175.5   Median :70.00   Median :23.00   Mode  :character
 Mean   :2.50   Mean   :175.5   Mean   :68.25   Mean   :23.00
 3rd Qu.:3.25   3rd Qu.:180.2   3rd Qu.:72.75   3rd Qu.:23.25
 Max.   :4.00   Max.   :184.0   Max.   :75.00   Max.   :24.00
```

Instead of a summary, we could use the specific functions: **mean()**, **median()**, **quantile**, **min()**, **max()** and **sd()**. Other useful functions are **cor()** for correlation between two vectors, and **var()** to find the variance between to vectors.

To target only one of the variables in the dataset, we can use **<dataset>$<variable>**.

```
> mean(PeopleTable_R$Height)
[1] 175.5
```

Tip: If you are only using one dataset, you can attach it with the function **attach(<dataset>)**. When this is done, you can simply write the variables without the need for **<dataset>$** in front.

A list of all the variables in the dataset is obtained by the function **names()**.

```
> names(PeopleTable_R)
[1] "PersonID" "Height"   "Weight"   "Age"      "Gender"
```

A quick way to check that the data is loaded properly, is with the function **head()**. This displays the first 6 rows of the dataset in the console.

```
> head(PeopleTable_R)
# A tibble: 4 x 5
  PersonID Height Weight   Age Gender
     <dbl>  <dbl>  <dbl> <dbl> <chr>
1        1    184     75    24 M
2        2    167     58    22 F
3        3    179     68    23 M
4        4    172     72    23 F
```

4.4 Changing and creating data

NB: Even though R allows for white spaces to be part of the variable name, this is bad practice and should be avoided for future problems.

To address a specific entry in a vector or matrix, we use square brackets '[]'.

If I want to address the second entry of the variable names in PeopleTable_R, this can be done by:

```
> names(PeopleTable_R)[2]
[1] "Height"
```

Note: Indexing in R starts at 1 and not 0.

Changing names of variable names is simple, for instance:

```
> names(PeopleTable_R)[2] = "Length"
> names(PeopleTable_R)
[1] "PersonID" "Length"   "Weight"   "Age"      "Gender"
```

The second line is just to illustrate that the variable has changed name.

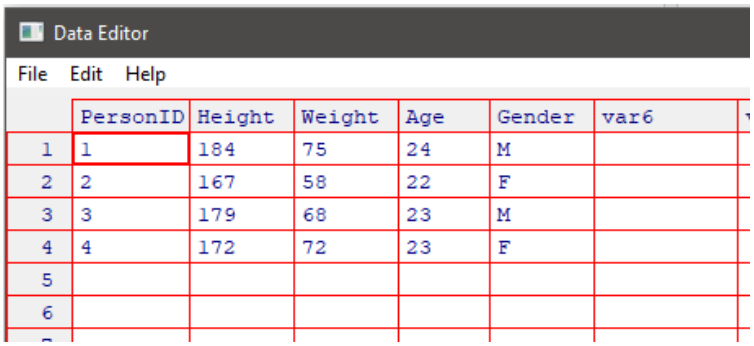
In some cases, you want to add another column in the dataset, and this can be done in R.

```
> PeopleTable_R["BMI"] = c(PeopleTable_R$Weight / (PeopleTable_R$Height/100)^2)
```

	PersonID	Height	Weight	Age	Gender	BMI
1	1	184	75	24	M	22.15265
2	2	167	58	22	F	20.79673
3	3	179	68	23	M	21.22281
4	4	172	72	23	F	24.33748

Another way to edit the dataset is with the function **fix()**.

```
> fix(PeopleTable_R)
```



When the fix-function is executed, a new window opens (as you can see above). From this window it is possible to edit the data set.

For an overview of commonly used operators, you can refer to the following table:

Operation	Explanation
a / b	a divided by b
a * b	a multiplied by b
a^b	a to the power of b
log(a)	Logarithm of a
exp(a)	Taking the exponential
sqrt(a)	Square root of a
abs(a)	Absolute value of a
==	(exactly) equal
!=	Not equal
>	Larger than
<	Smaller than
>=	Larger or equal
<=	Smaller or equal
&	Logical AND
|	Logical OR
!	Logical NOT

4.5 Saving to workspace

It is useful to save the work you have done, so you can open it later. To do this, you can save the session by clicking Session -> Save Workspace As…

Also useful is to save the instructions you have used in an R script, and this can be saved from File -> Save. These scripts can be created by File -> New File -> R Script. To open them, click File -> Open Files… and locate your script. To run a line in the script, press Ctrl + Enter, with the cursor in the respective line. To run the entire script, mark everything and press Ctrl + Enter.

Tip: Comments in R are created by the starting cue '#'. It is highly recommended to comment code that is not self-explanatory.

To shut down R, we can use the function **q()**. Then we will be given the choice to save the current workspace (all objects created will be saved and can be used later).

```
> q()
Save workspace image to ~/.RData? [y/n]:
```

After typing "y" or "n", the window closes down.

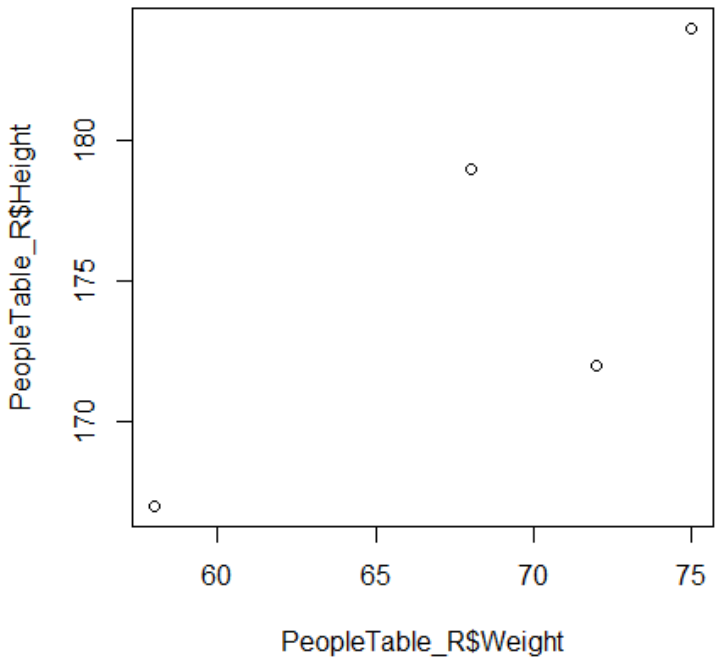
Before we exit, we can also save the history from the console (what lines we have typed in) with the function **savehistory()**. To load, we use the function **loadhistory()** next time.

5 Graphics (and plots)

The graphics in R have some standard parameters, so they look the same every time. To change these, you can use the **par()** function. For more information, type **?par** in the console.

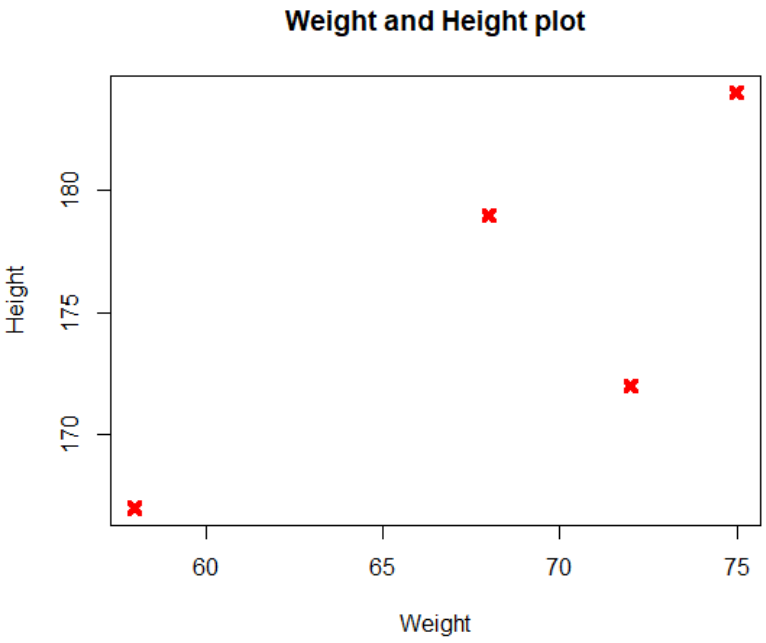
Plotting the Weight on the x-axis, and Height on the y-axis can be done like this:

```
> plot(PeopleTable_R$Weight, PeopleTable_R$Height)
```



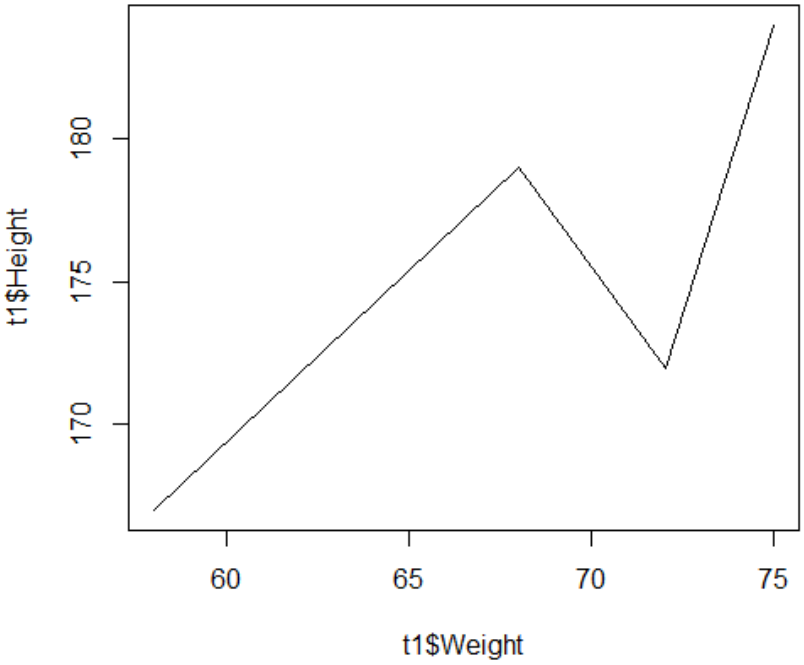
To show some of the possible improvements for this simple plot, this is an example:

```
> plot(PeopleTable_R$Weight, PeopleTable_R$Height, xlab="Weight",
 ylab="Height", main="Weight and Height plot", col="red", pch=4,
 lwd=4)
```



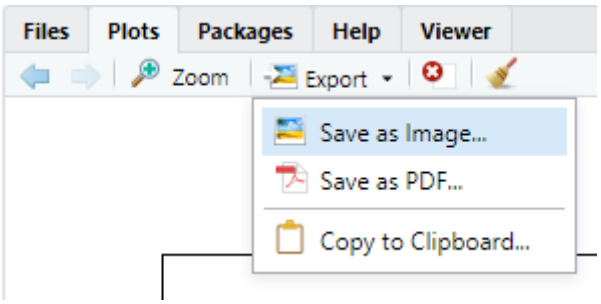
We can also create a line plot:
(Note that I create a new dataset 't1' where the data is ordered by weight)

```
> t1 <- PeopleTable_R[order(PeopleTable_R$Weight),]
> plot(t1$Weight, t1$Height, type="l")
```



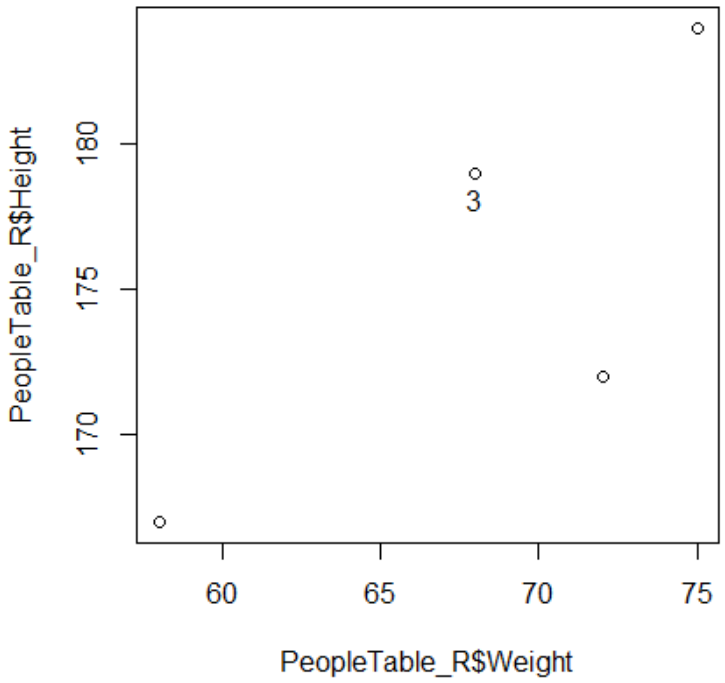
Use **?plot** for more information on how to use this function.

There are numerous ways to save the plots, for instance in the *Plots* tab:



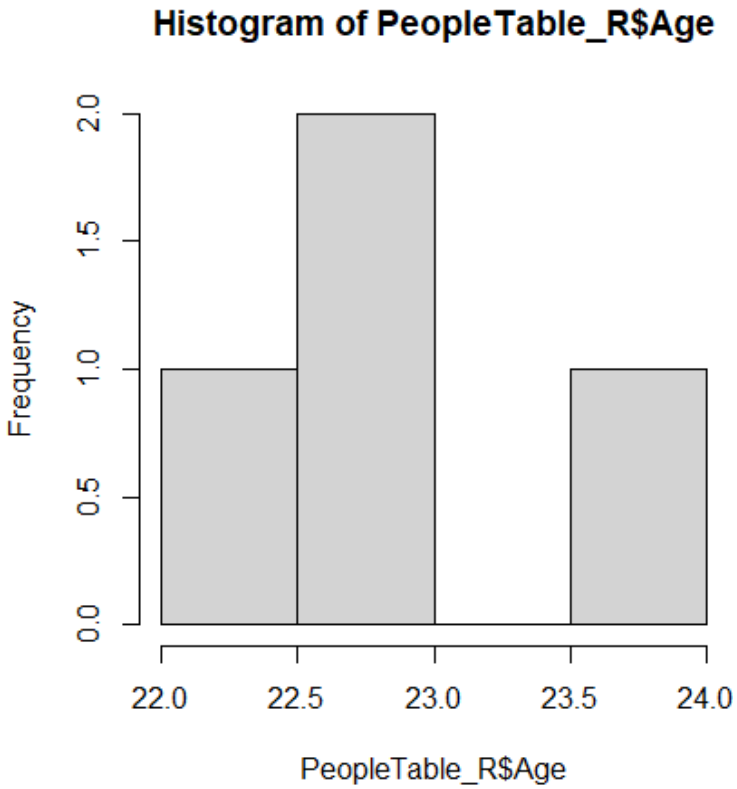
An interactive way of identifying points on a plot, is by the function **identify()**. The first two arguments are the variables for the x-axis and y-axis, and thirdly we chose what variable we want to know about a given point. After the **identify()** line is executed, simply click on the points in the plot you want to identify, and when you are done, press Esc. The values will be printed in the console, as well as displayed in the plot.

```
> plot(PeopleTable_R$Weight, PeopleTable_R$Height)
> identify(PeopleTable_R$Weight, PeopleTable_R$Height, PeopleTable_R$PersonID)
[1] 3
```



Histograms can also be a simple but very useful plot and is made by the function **hist()**.

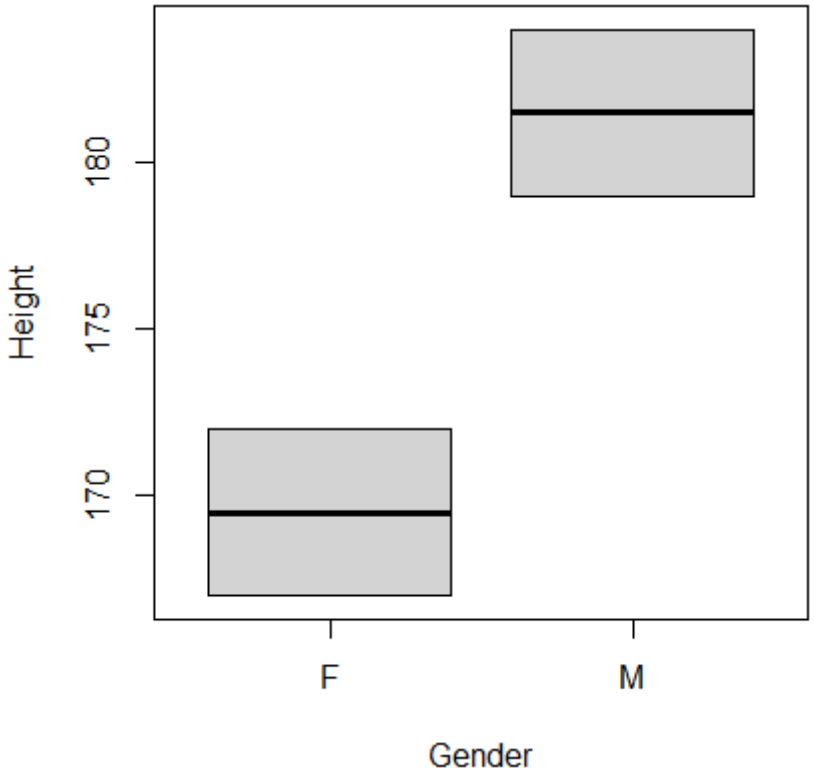
```
> hist(PeopleTable_R$Age)
```



These can also be refined by some arguments, see **?hist** for more.

It also possible to create boxplots in R with the function **boxplot()**, where the first input is the relation between the variables Height and Gender in this case, both from the dataset PeopleTable_R.

```
> boxplot(Height ~ Gender, data=PeopleTable_R)
```



To get more control we can use parameters.

Tip: To clear the *Plot* tab, you can use the function **dev.off()** in the console.

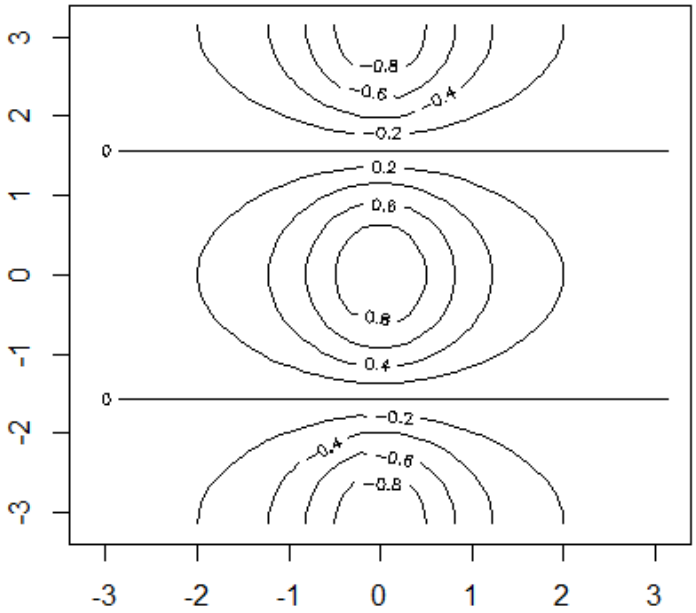
5.1 More advanced plots

To produce a contour plot (used to represent three-dimensional data; somewhat like a topographical map) in R, we use the function **contour()**.

The most important input variables (arguments) to the contour() function, are the grid lines (usually x and y) and a matrix (z) containing the values to be plotted. Here I will illustrate an example based on the code in p.46 in "An introduction to statistical learning".
Note that the function **outer()** takes the arguments vector x, vector y, and a function. The outer function applies this function to the two vectors.

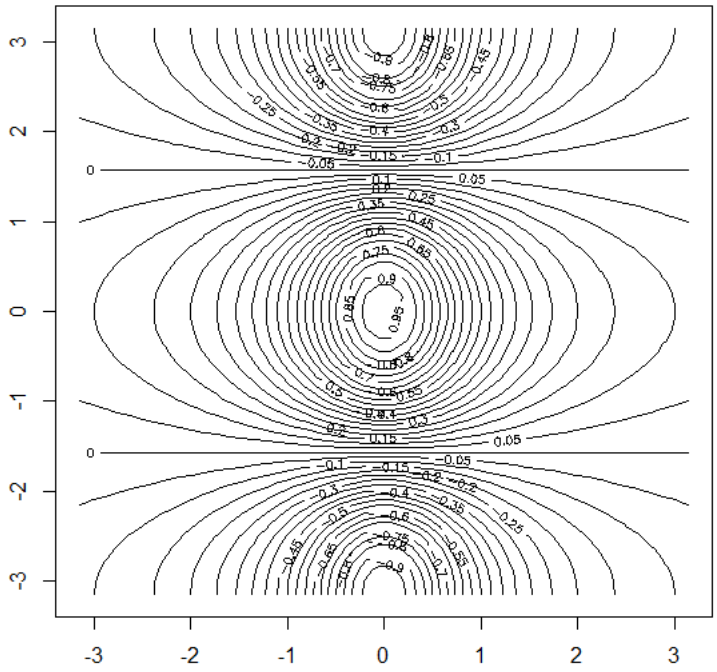
```
> x = seq(-pi, pi, length=50)
> y = x
> f = outer(x, y, function(x,y)cos(y)/(1+x^2))
> contour(x, y, f)
```

When the fourth line is executed, we get the plot:



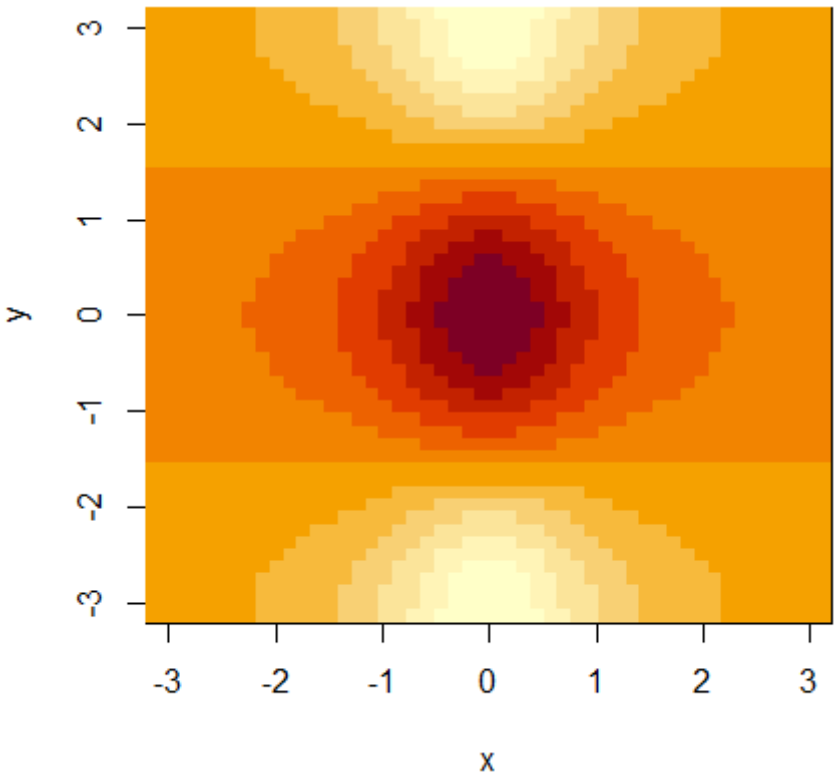
If we prefer to decide the number of levels in the plot, we can do this by explicit changing the **nlevels**-argument.

```
> contour(x, y, f, nlevels=45, add=T)
```



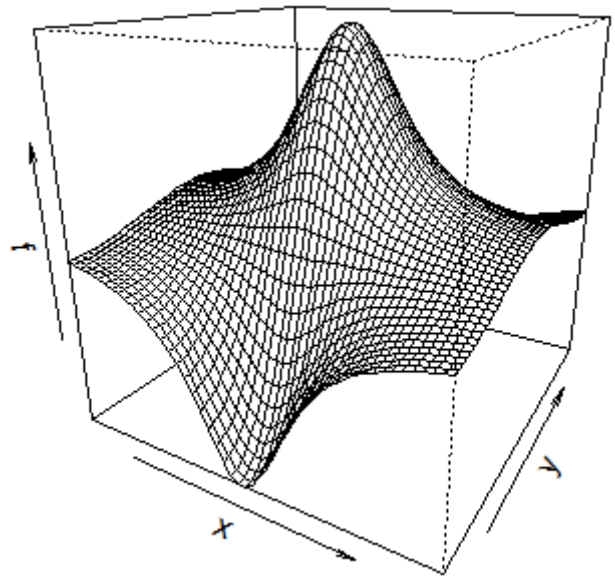
The function **image()** is very similar to **contour()**, but gives us a color-coded plot (heatmap):

```
> image(x, y, f)
```



For a three-dimensional plot, we can use the function **persp()**. Theta and phi are used to control at what angle we view the plot.

```
> persp(x, y, f, theta=30, phi=20)
```



For more information about these different plots, the help sites (**?contour**, **?image**, **?persp**) are recommended.

6 Sequences, vectors, matrices, and random numbers

6.1 Create a sequence of numbers

It can be useful to create a sequence of numbers, and this is done with the function **seq()**.

To create a vector of integers between 1 and 10 (both inclusive):

```
> x = seq(1, 10)
> x
 [1]  1  2  3  4  5  6  7  8  9 10
```

In short, we could equally write:

```
> x = 1:10
> x
 [1]  1  2  3  4  5  6  7  8  9 10
```

To create a vector of equally spaced numbers from 0 to 1 with length 10:

```
> x = seq(0, 1, length=10)
> x
 [1] 0.0000000 0.1111111 0.2222222 0.3333333
 0.4444444 0.5555556 0.6666667 0.7777778
 [9] 0.8888889 1.0000000
```

6.2 Vectors in R

To create a vector in R, we use the function **c()**, for concatenate.
The vector $[1, 2, 7]$ is created by typing: **c(1, 3, 7)**

To save the vector to a variable **x** (you can call it whatever you want), we can use the syntax **<-** or **=**, both have the same effect. Typing the variable **x** in the console would now display the vector.

```
> x <- c(1,3,7)
> x
[1] 1 3 7
> y = c(2,6,1)
> y
[1] 2 6 1
```

We can use mathematical operations on vectors, such as addition (+):

```
> x+y
[1] 3 9 8
```

To find the length of a vector, we use the function length().

```
> length(x)
[1] 3
```

6.3 Objects stored

It might be useful to get an overview of what objects (such as data and functions) we have saved so far. We can get a list of such objects with the function **ls()** [starting with a small L, ls() is short for list].

```
> ls()
[1] "x" "y"
```

To remove objects, we use **rm()**. In the parenthesis we put the objects we want to remove.

rm(x) to remove the object x.

```
> rm(x)
> ls()
[1] "y"
```

rm(list=ls()) to remove all objects.

```
> rm(list=ls())
> ls()
character(0)
```

6.4 Matrices

To create a matrix in R, we use the function **matrix()**. It takes multiple inputs, the first three are: data (the actual numbers in the matrix), number of rows, number of columns.

To create the matrix $A = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}$ we write:

```
> x = matrix(data=c(1,2,3,4,5,6), nrow=3, ncol=2)
> x
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
```

You can also split up on multiple lines:

```
> x = matrix(
+     data=c(1,2,3,4,5,6),
+     nrow=3,
+     ncol=2
+ )
> x
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
```

(The +-signs are created automatically. To shift line without executing the line, use Shift + Enter.)

Or in short write:

```
> x = matrix(c(1,2,3,4,5,6), 3, 2)
> x
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
```

All of the above will give the same result. For the last one, it is important to not mix up the move order of the inputs, check with **?matrix** to find out more.

The standard way of distributing the data in the matrix in R, is to use the data to fill up first column, then second column, and so on. It is possible to have the program start filling up first row, the second row, and so on -by using the additional input:

```
> x = matrix(c(1,2,3,4,5,6), 3, 2, TRUE)
> x
     [,1] [,2]
[1,]    1    2
[2,]    3    4
[3,]    5    6
```

Equally correct you could write:

```
> x = matrix(data=c(1,2,3,4,5,6), nrow=3, ncol=2, byrow=TRUE)
> x
     [,1] [,2]
[1,]    1    2
[2,]    3    4
[3,]    5    6
```

Multiple commands can be used on matrices, for instance we can raise every element in the matrix to the power of p, $for\ any\ p \in \mathbb{R}$, with the command **x^p**. For $x^{0.5}$ we can use the syntax **sqrt(x)**.

```
> x
     [,1] [,2]
[1,]    1    2
[2,]    3    4
[3,]    5    6
> x^2
     [,1] [,2]
[1,]    1    4
[2,]    9   16
[3,]   25   36
> sqrt(x)
         [,1]     [,2]
[1,] 1.000000 1.414214
[2,] 1.732051 2.000000
[3,] 2.236068 2.449490
```

To index the matrix, simply write:

```
> x[2,1]
[1] 3
```

We can also select multiple rows and columns:

```
> x[c(1,3), c(1,2)]
     [,1] [,2]
[1,]    1    2
[2,]    5    6
```

The 1 is found in x[1,1], 2 in x[1,2], 5 in x[3,1] and 6 in x[3,2].

Also possible is:

```
> x[1:3, 2]
[1] 2 4 6
```

Or:

```
> x[ ,2]
[1] 2 4 6
```

(in this case, the blank will index all rows)

To index every row except the listed, we use minus (-):

```
> x[-3, 1:2]
     [,1] [,2]
[1,]    1    2
[2,]    3    4
```

To find the dimension of a matrix, use the function **dim()**:

```
> dim(x)
[1] 3 2
```

x has 3 rows and 2 columns

6.5 How to deal with missing values

Missing values are represented by 'NA', for Not Available.

Impossible values (for instance: $\frac{1}{0}$) are represented by 'NaN', for Not a Number.

This section will look at how to deal with missing values in a vector, matrix, or a dataset.

6.5.1 Missing values in a vector

To check for missing values, we can use the function **is.na()**.

```
> x = c(1, 2, NA)
> x
[1]  1  2 NA
> is.na(x)
[1] FALSE FALSE  TRUE
> is.na(2)
[1] FALSE
> is.na(NA)
[1] TRUE
```

If we want to recode a certain value to missing value, this can be done with by:

```
> y = c(1,4,3,-1,2,3,-1)
> y
[1]  1  4  3 -1  2  3 -1
> y[y==-1] = NA
> y
[1]  1  4  3 NA  2  3 NA
```

When doing analysis of vectors, we should exclude the missing values from the calculation.

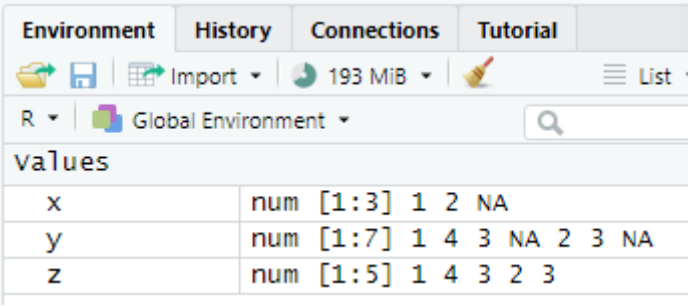
```
> mean(y)
[1] NA
> mean(y, na.rm=TRUE)
[1] 2.6
> y
[1]  1  4  3 NA  2  3 NA
```

Note that the vector y is not changed, only in the calculation of the mean is the NA-values removed.

To omit the missing values and save the new vector in another variable, we can use **na.omit()**.

```
> z = na.omit(y)
> z
[1] 1 4 3 2 3
attr(,"na.action")
[1] 4 7
attr(,"class")
[1] "omit"
```

In the *Environment* tab to our upper right, we can get an overview of our variables.



A quick way to count the number of missing values in a dataset, is by **sum(is.na())**.

```
> sum(is.na(y))
[1] 2
```

As we can see, the vector y clearly has two missing values.

6.5.2 Missing values in a matrix

The logic and functions around missing values for vectors can be extended to matrices.

```
> A = matrix(c(1,NA,3,NA,5,6), nrow=3, ncol=2)
> A
     [,1] [,2]
[1,]    1   NA
[2,]   NA    5
[3,]    3    6
```

Finding the missing values in the matrix A:

```
> is.na(A)
      [,1]  [,2]
[1,] FALSE  TRUE
[2,]  TRUE FALSE
[3,] FALSE FALSE
```

Replace a value (here: -1) with 'NA' in the matrix B:

```
> B = matrix(c(1,-1,0,0,-1,1), nrow=3, ncol=2)
> B
     [,1] [,2]
[1,]    1    0
[2,]   -1   -1
[3,]    0    1
> B[B==-1] = NA
> B
     [,1] [,2]
[1,]    1    0
[2,]   NA   NA
[3,]    0    1
```

To find the mean, we must first exclude the missing values:

```
> mean(A)
[1] NA
> mean(A, na.rm=TRUE)
[1] 3.75
```

Using the **na.omit()** function on matrices, R removes all rows that contains missing values.

```
> C = na.omit(A)
> C
     [,1] [,2]
[1,]    3    6
attr(,"na.action")
[1] 2 1
attr(,"class")
[1] "omit"
```

Counting the number of missing values works the same way as for vectors.

```
> sum(is.na(A))
[1] 2
```

6.6 Random variables

To generate a vector of n numbers of random normal variables in R we use the function **rnorm(n)**.

```
> rnorm(10)
 [1] -0.67859445  1.06736136  1.33289019 -0.04068704  0.71950229
 [6]  1.89003242  0.04429754 -1.13726634  0.71368754 -0.24584483
```

By default, it creates n number of random variables with mean of 0 and standard deviation of 1. This can be altered, for instance by using the inputs: **rnorm(10, mean=50, sd=0.1)**. (Here n=10)

```
> rnorm(10, mean=50, sd=0.1)
 [1] 50.05436 50.05122 50.11055 50.00676 49.86885 49.96544
 [7] 49.87812 50.21248 49.86971 50.00061
```

Every time the function is called, we get a different answer. The reproduce the same set of random numbers every time, we can use the function **set.seed()** before we use the **rnorm(n)** function. The input should be an (arbitrary) integer. We can now do:

```
> set.seed(1303)
> rnorm(10)
 [1] -1.1439763145  1.3421293656  2.1853904757  0.5363925179
 [5]  0.0631929665  0.5022344825 -0.0004167247  0.5658198405
 [9] -0.5725226890 -1.1102250073
> rnorm(10)
 [1] -0.04868712 -0.69565622  0.82891748  0.20665286 -0.23567451
 [6] -0.55631049 -0.36475436  0.86235503 -0.63077154  0.31360213
> set.seed(1303)
> rnorm(10)
 [1] -1.1439763145  1.3421293656  2.1853904757  0.5363925179
 [5]  0.0631929665  0.5022344825 -0.0004167247  0.5658198405
 [9] -0.5725226890 -1.1102250073
```

Every time we use the same seed, we get the same randomly generated vector.

7 Functions, loops, and conditional execution

7.1 How to create a function

Creating functions are intuitive in R. We must use **function()**, and first specify the input variables (for instance x and y). Then we specify how the function will map these inputs to an output (for instance $f(x, y) = 2x + y^2$). At last, we write what the function will return -this is completely up to us (for instance we can return the input values and the output value in the form $c(x, y, value)$).

```
> f1 <- function(x, y) {
+     value <- 2*x + y^2
+     return(c(x, y, value))
+ }
> f1(2, 3)
[1]  2  3 13
```

Note that the '+' signs are automatically added by R. To use multiple lines, press Shit + Enter to jump to next line without executing any code yet.

7.2 How to see the code inside a function

For simple functions, the source code of the function will not give you much extra information. However, for more complicated functions, this can be useful -and it's very easy to do.

For functions in R that comes from the base package, simply write the function name in the console:

```
> rowMeans
function (x, na.rm = FALSE, dims = 1L)
{
    if (is.data.frame(x))
        x <- as.matrix(x)
    if (!is.array(x) || length(dn <- dim(x)) < 2L)
        stop("'x' must be an array of at least two dimensions")
    if (dims < 1L || dims > length(dn) - 1L)
        stop("invalid 'dims'")
    p <- prod(dn[-(id <- seq_len(dims))])
    dn <- dn[id]
    z <- if (is.complex(x))
        .Internal(rowMeans(Re(x), prod(dn), p, na.rm)) + (0+1i) *
            .Internal(rowMeans(Im(x), prod(dn), p, na.rm))
    else .Internal(rowMeans(x, prod(dn), p, na.rm))
    if (length(dn) > 1L) {
        dim(z) <- dn
        dimnames(z) <- dimnames(x)[id]
    }
    else names(z) <- dimnames(x)[[1L]]
    z
}
<bytecode: 0x0000024bc0052158>
<environment: namespace:base>
```

In R there are different type of classes, some stricter than others. For instance, the S4 class requires a formal definition and a uniform way to create objects, unlike the S3 class.

I will use the S4 function **show()** in this example. If you try to type the function name, you will get:

```
> show
standardGeneric for "show" defined from package "methods"

function (object)
standardGeneric("show")
<bytecode: 0x0000024bbf30bb20>
<environment: 0x0000024bb7e56080>
Methods may be defined for arguments: object
Use  showMethods(show)  for currently available ones.
(This generic function excludes non-simple inheritance; see ?setIs)
```

To see that this is a S4 function, we can use **isS4()**.

```
> isS4(show)
[1] TRUE
```

The proper way to see its source code is first to find the object it belongs to.

```
> showMethods(show)
Function: show (package methods)
object="ANY"
object="C++Class"
object="C++Function"
object="C++Object"
object="classGeneratorFunction"
object="classRepresentation"
object="envRefClass"
object="externalRefMethod"
object="function"
    (inherited from: object="ANY")
object="genericFunction"
object="genericFunctionWithTrace"
object="MethodDefinition"
object="MethodDefinitionWithTrace"
object="MethodSelectionReport"
object="MethodWithNext"
object="MethodWithNextWithTrace"
object="missing"
    (inherited from: object="ANY")
object="Module"
object="namedList"
object="ObjectsWithPackage"
object="oldClass"
object="refClassRepresentation"
object="refMethodDef"
object="refObjectGenerator"
object="signature"
object="sourceEnvironment"
object="standardGeneric"
    (inherited from: object="genericFunction")
object="traceable"
```

Then we check for the method we are interested in.

```
> getMethod('show', 'signature')
Method Definition:

function (object)
{
    message(gettextf("An object of class %s", dQuote(class(object)))
        domain = NA)
    val <- object@.Data
    names(val) <- object@names
    callNextMethod(val)
}
<bytecode: 0x0000024bb7f173e8>
<environment: 0x0000024bb7e80570>

Signatures:
        object
target  "signature"
defined "signature"
```

7.3 How to create loops

Loops can be created by: **for (variable in vector) {…}**

```
> v = c()
> for (i in 1:10) {
+     v = c(v, i)
+ }
> v
 [1]  1  2  3  4  5  6  7  8  9 10
```

The variable ‘v’ is first assigned an empty vector. For every iteration in the loop, the vector expands by adding the value of ‘i’ at the end of the vector.

7.4 How to do conditional execution

If-else statements can be created by: **if (condition) {…} else {…}**

```
> i = 0
> if (i == 0){
+     a = 10
+ } else {
+     a = 20
+ }
> a
[1] 10
```

The variable ‘i’ is assigned the value 0. Our if-statements checks whether ‘i’ is equal to ‘0’. Since it is, the line ‘a = 10’ is executed. The else part is not executed in this case.

8 Linear Regression

To illustrate how to do simple statistical calculations and linear regression in R, I will use the Excel sheet 'PeopleTable_R' as shown below:

	PersonID	Height	Weight	Age	Gender
1	1	184	75	24	M
2	2	167	58	22	F
3	3	179	68	23	M
4	4	172	72	23	F

To import files from Excel, see section 4.2.

To acquire some basic statistical values from the dataset, we can use **summary()** as mentioned in section 4.3.

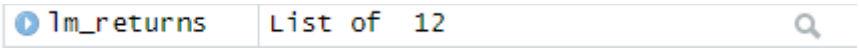
```
> summary(PeopleTable_R[c("Weight", "Height")])
     Weight          Height
 Min.   :58.00   Min.   :167.0
 1st Qu.:65.50   1st Qu.:170.8
 Median :70.00   Median :175.5
 Mean   :68.25   Mean   :175.5
 3rd Qu.:72.75   3rd Qu.:180.2
 Max.   :75.00   Max.   :184.0
```

To estimate linear models, we use the function **lm()**. The first argument should be on the form $y \sim x$ for variables y and x, where y is the dependent variable and x the independent variable. As a second argument, we must specify the dataset these variables are attained from.

Say we want to predict the height of a person with a given weight. To do this, we fit a simple linear regression model to these variables by using the function **lm()**.

```
> lm_returns = lm(Height ~ Weight, data = PeopleTable_R)
```

This does not print any output, but we can observe in the *Environment* tab that the variable lm_returns has been created and contains some data.



To see some basic information about the model, we can call our variable lm_returns.

```
> lm_returns

Call:
lm(formula = Height ~ Weight, data = PeopleTable_R)

Coefficients:
(Intercept)       Weight
   121.4385       0.7921
```

We observe that the parameter estimates for the intercept $(\hat{\alpha})$ is 121.4385 and the slope $(\hat{\beta})$ is 0.7921. That is, the function $f(x) = 121.4385 + 0.7921x$ is the best fitting linear line of our four points of $(< Weight >, < Height >)$.

To get a summary of all the data created in lm_returns, we can use the **summary()** function.

```
> summary(lm_returns)

Call:
lm(formula = Height ~ Weight, data = PeopleTable_R)

Residuals:
      1       2       3       4
 3.1533 -0.3809  3.6980 -6.4704

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 121.4385    30.5940   3.969    0.058 .
Weight        0.7921     0.4463   1.775    0.218
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 5.728 on 2 degrees of freedom
Multiple R-squared:  0.6117,    Adjusted R-squared:  0.4175
F-statistic:  3.15 on 1 and 2 DF,  p-value: 0.2179
```

This gives us: p-values, standard errors for the coefficients, R^2 statistic and F-statistic for the model.

We can run **names(lm_returns)** to find more information stored in the variable:

```
> names(lm_returns)
 [1] "coefficients"  "residuals"     "effects"
 [4] "rank"          "fitted.values" "assign"
 [7] "qr"            "df.residual"   "xlevels"
[10] "call"          "terms"         "model"
```

To extract the quantities, we can use for instance:

```
> lm_returns$coefficients
(Intercept)      Weight
121.4385432   0.7921093
```

A safer way to do this is by using extractor functions like **coef()**:

```
> coef(lm_returns)
(Intercept)      Weight
121.4385432   0.7921093
```

To find the confidence interval for the coefficient estimates, we use the function **confint()**. As we see below, the 95% confidence interval is the standard unless otherwise specified.

```
> confint(lm_returns)
                2.5 %     97.5 %
(Intercept) -10.197012 253.074098
Weight       -1.128146   2.712364
> confint(lm_returns, level=0.90)
                 5 %       95 %
(Intercept) 32.1043703 210.772716
Weight      -0.5110676   2.095286
```

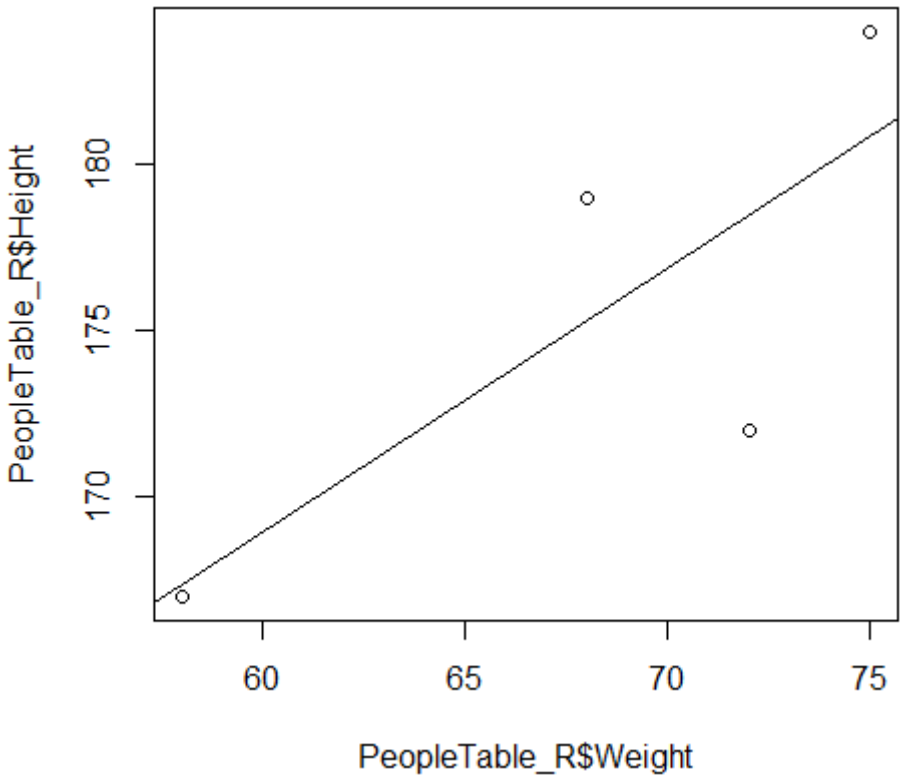
To produce confidence intervals and prediction intervals for the prediction of height for a given value of weight, we use the function **predict()**.

```
> predict(lm_returns, data.frame(weight=c(60, 65, 70, 75)), interval="confidence")
       fit      lwr      upr
1 168.9651 148.8941 189.0361
2 172.9256 159.1118 186.7395
3 176.8862 164.1125 189.6599
4 180.8467 162.9615 198.7319
> predict(lm_returns, data.frame(weight=c(60, 65, 70, 75)), interval="predict")
       fit      lwr      upr
1 168.9651 137.1792 200.7510
2 172.9256 144.6711 201.1802
3 176.8862 149.1254 204.6470
4 180.8467 150.3939 211.2996
```

In this case the 95% confidence interval associated with weight=65 is $(159.11, 186.74)$, while the 95% prediction interval is $(144.67, 201.18)$. Naturally they both have the same predicted value (172.93 when weight is 65), where the predict interval is bigger than the confidence interval.

To plot the regression line, we can use **abline()**:

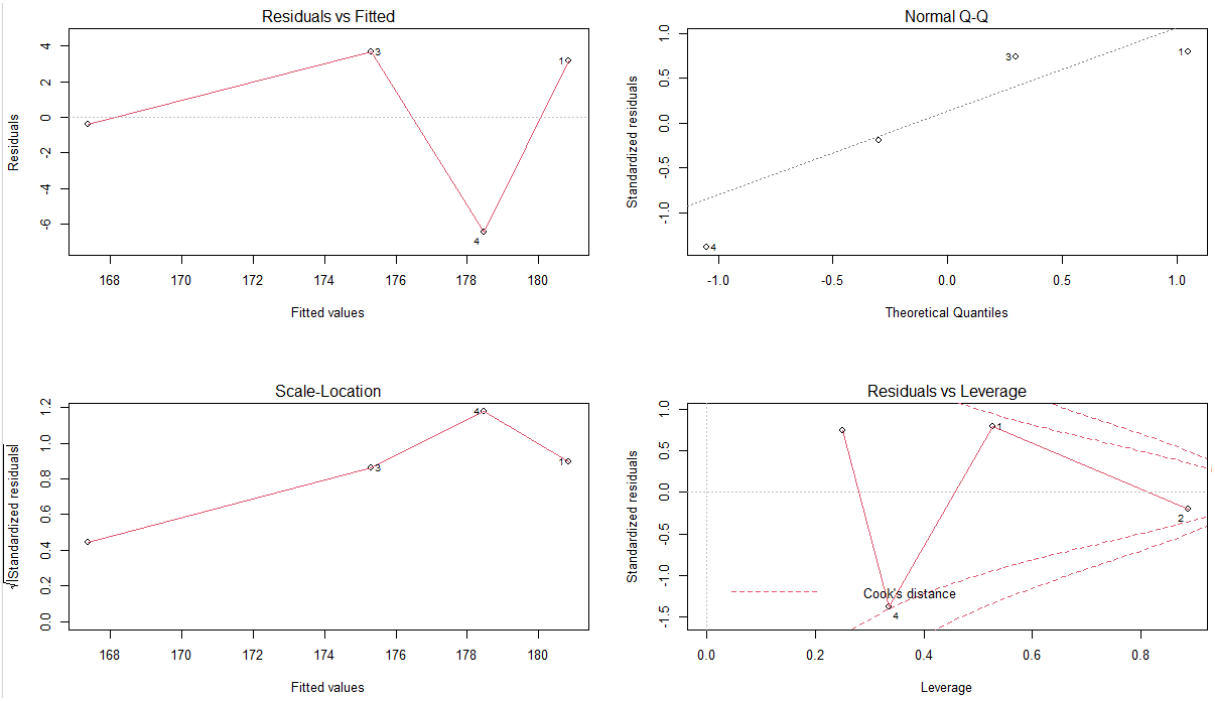
```
> plot(PeopleTable_R$Weight, PeopleTable_R$Height)
> abline(lm_returns)
```



8.1 More statistical plots connected to the linear model

There are four diagnostics plots we can view by applying **plot()** function to the output from **lm()**. To view all of the four plots at once, we can use:

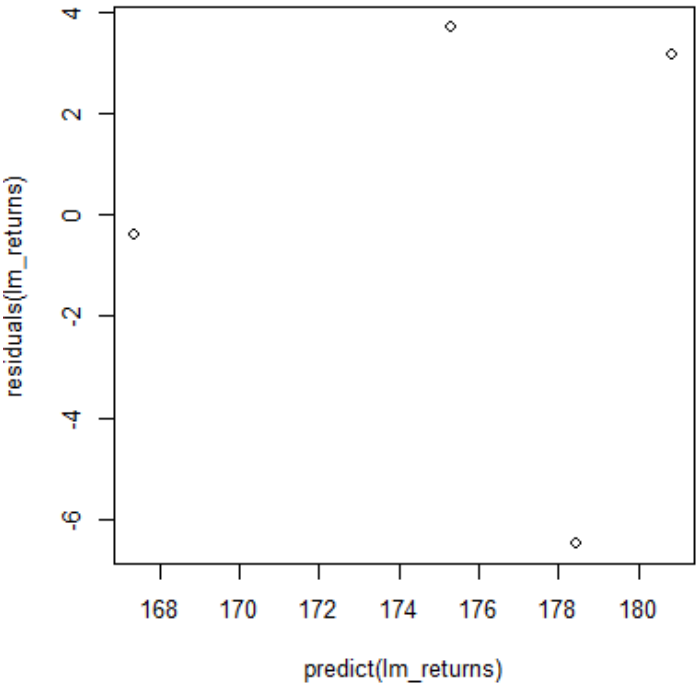
```
> par(mfrow=c(2,2))
> plot(lm_returns)
```



If you do not use the first line, you will be able to see one plot at a time, using Enter to switch to the next plot.

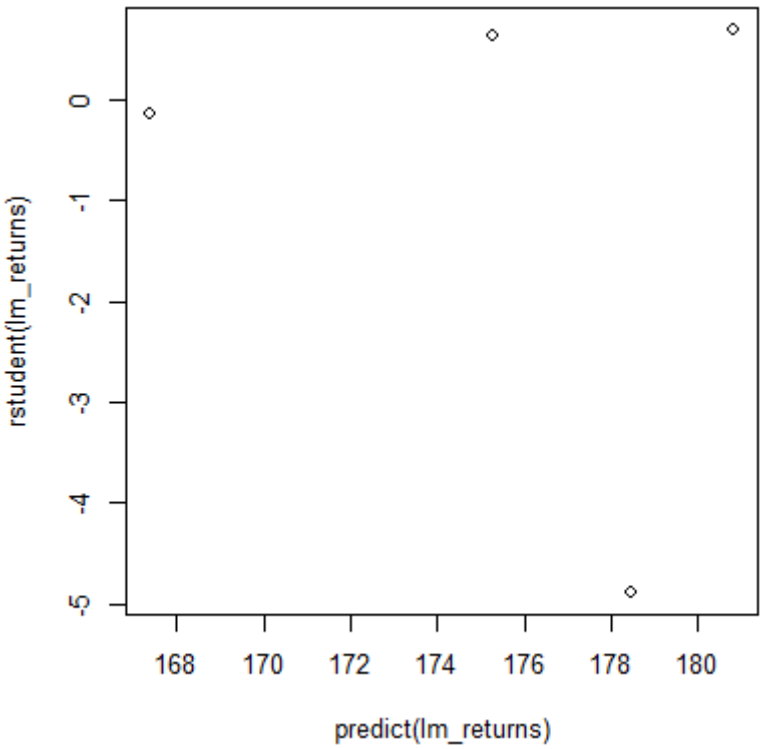
The residuals from the linear regression fit can also be computed with **residuals()**.

```
> plot(predict(lm_returns), residuals(lm_returns))
```



The function **rstudent()** returns the studentized residuals, which can be plotted against the fitted values.

```
> plot(predict(lm_returns), rstudent(lm_returns))
```



As a last example, the **hatvalues()** function can compute leverage statistics, which we can plot:

```
> plot(hatvalues((lm_returns)))
```