Intro to R

Willem Vervoort and Dasapta Erwin Irawan

09 December 2017

Contents

1 Introduction

This is an introduction to R written for the "Open data workshop in Bandung 5-9 Feb 2018", jointly organised by the University of Sydney and Institut Teknologi Bandung.

This work is based on earlier events from the authors and it also builds on many of the introduction to R literature on the internet.

This course is not a complete introduction, and more in depth knowledge on R and the use of R can be gained from many courses on-line and by basic practice.

This course covers simple R, basic statistics, data frame operations, reading in files and a plotting.

We hope that this course offers sufficient depth to help you engage in the rest of the course, which uses R for analysis of the output.

2 R as a modelling environment

The origins of R are in statistics, so this is hat R does best. However, over time, it has proven to be a flexible language that can also be used quite effectively for programming and data science.

2.1 R and R Studio

2.1.1 Base R vs IDE

If R is the machine under the hood, then R Studio would be the dashboard, steering wheel, as well as the gas and brake paddles. People frequently mention R as `base R` and R Studio is an Integrated Development Environment (IDE).

Is there another IDE other than R Studio? The answer is Yes. You could check out R Commander, Revolution R.

2.1.2 Running R online

Can we run R online? The answer is also Yes. R Studio offers a paid cloud service. You could try R fiddle for a limited range of code of package installation, CoCalc/Sage Math Cloud, Jupyter, and the New Comer Code Ocean.

2.1.3 R is cross platform

R and R Studio are cross platform. So you could use R on these major OS', Windows, Mac or Linux, so it's OK if you work with another person who don't use the same OS as you do. You just have to make sure that all parties have the same data and the same package installed in the system, and the same codes to run.

2.1.4 How to install R and R Studio

We recommend to install R first followed by R Studio. Install R from CRAN and R Studio from its official site.

2.1.5 R components

In R programming, as also in other programming language, the two main components are the data and the codes. Using both, you could start an analysis and produce plots and tables as outputs. So in order to do the analysis, we will need `package`.

The good thing about R is, there are `base commands`, that is commands that included in the base R installation. This commands are progressing as you install newer version of R. It's getting better and easier through time. But users are allowed to make their own commands that consists of a function or sets of functions. Sets of function can be grouped as a `package`. So you would need to install the package first and load the package, before using the command or function inside that package. You would only need to install the package once.

You could run this line to install a package from CRAN server.

```
install.packages("packageName") # case sensitive
library(packageName) # to load the package
```

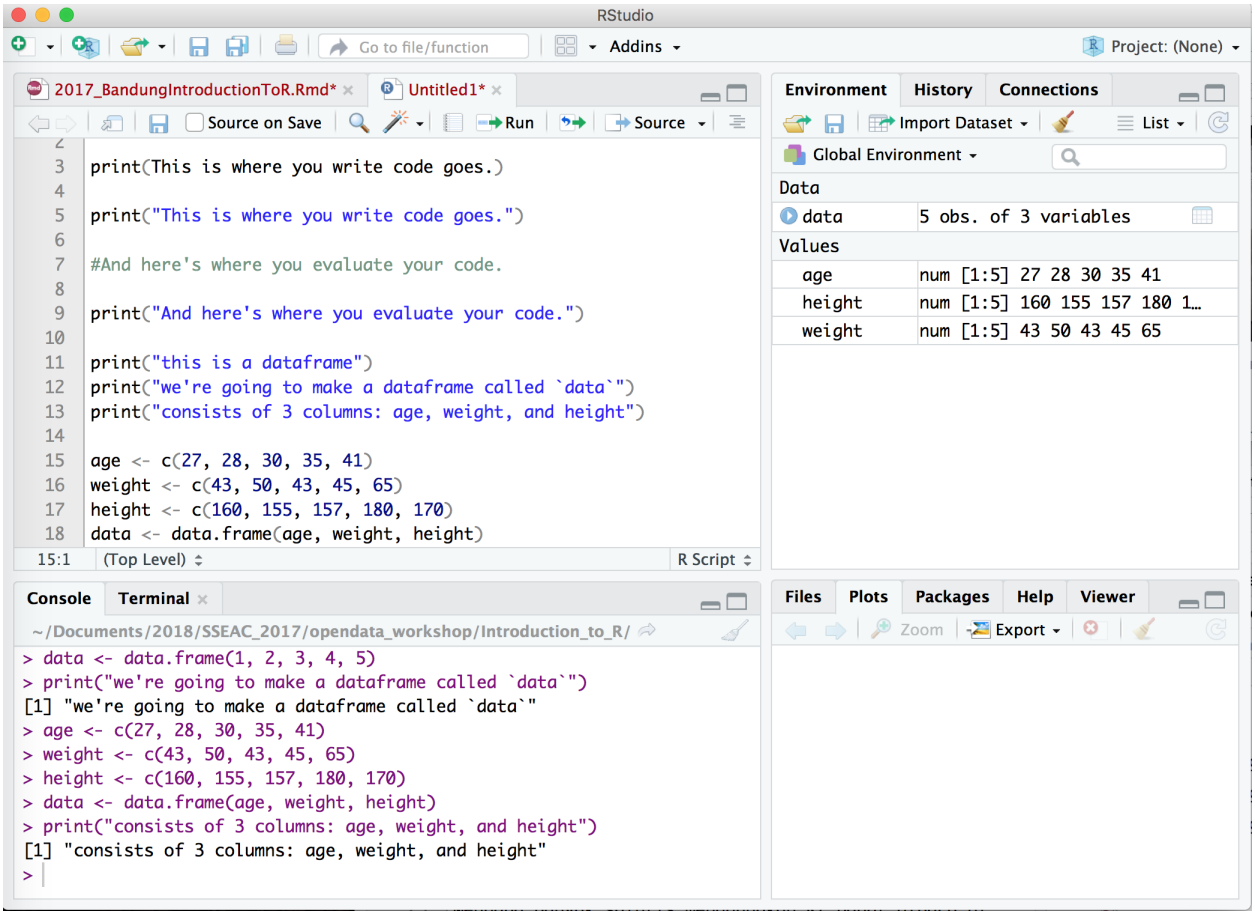


Figure 1: Four panels in R Studio

Other than CRAN, you may find packages that are still in development stage on GitHub, a repository of codes. You could install a package on GitHub using `install_github` command from `devtools` package.

```
install.packages("devtools")
library(devtools)
install_github(repoAddress)
```

Now you could run a command from a package without loading the package. But off course you still need to install the package in to your system

```
install.packages("devtools")
devtools::install_github(repoAddress)
```

2.1.6 Navigation

In R Studio, you would see four panels (clock-wise): Codes on top right, Environment, Files/folder/plots/packages, and console. You write your lines of code in the code panel then observe the process of you code in the console panel. Find out in console, if your codes are running well or have a problem (error messages), or just a warning. Then you could see all components that related to the process in the Environment panel.

2.1.7 Working folder structure

In R and in any other command line-based application, you would need to tell the app your current folder location and the location of the data. Usually we use the following folder structure:

- main project folder
- data: put your data here
- code: put your code here
- output: put your plots and tables here
- text: put your report here

But in practice, we usually work with code, data, outputs in one folder, but use it as a process or intermediate folder. We usually sort out the components at the final stage of our work. But when is "final stage" be? So do the file sorting several times.

2.1.7.1 Exercise

- Can you check your working folder/directory and what's inside it?

3 BASIC R

3.1 R as a calculator

In its most basic form, R is a calculator

```
3*5
```

```
## [1] 15
```

```
50/100 + 0.1
```

```
## [1] 0.6
```

```
10 - 20
```

```
## [1] -10
```

3.2 Objects in R

The basic structure of R is based on objects, which are named. R is case sensitive, so keep this in mind. The main object we will use here is a *dataframe* or its modern variant the *tibble*.

All objects will be loaded on to the memory. So if you have a datafile, the first thing to do is to load it on your memory as an object that can be seen in the Environment panel. Thus, whatever you do with the object will not change your file, unless you save the object as a file.

R uses "<-" to assign a value (or another object) to an object. You may find "=" means the same, but we don't recommend it, because you also use "=" with different meaning in the command and parameter.

```
# assign
x <- 5
y <- 2
```

You can call up what is stored in the object (inspect) again by just typing its name:

```
x
```

```
## [1] 5
```

These objects will show up in the "Environment" window in Rstudio, or you can use `ls()` in the console to list the objects. The function `c()` can be used to stick things together into a vector. Redo the below commands in your own script.

```
# a vector
x = c(1,2,5,7,8,15,3,12,11,19)
# another vector
y = 1:10
# you have now two objects
ls()
```

```
## [1] "x" "y"
```

```
# you can add, multiply or subtract
z = x + y
z
```

```
##  [1]  2  4  8 11 13 21 10 20 20 29
```

```
zz = x * y
zz
```

```
##  [1]   1   4  15  28  40  90  21  96  99 190
```

```
zzz = x - y
zzz
```

```
##  [1]  0  0  2  3  3  9 -4  4  2  9
```

```
foo = 0.5*x^2 - 3*x + 2
foo
```

```
##  [1]  -0.5  -2.0  -0.5   5.5  10.0  69.5  -2.5  38.0  29.5 125.5
```

3.3 A dataframe

A dataframe is a bit more complex, and here is a simple demonstration of its power.

```
Rainfall <- data.frame(City = c("Montevideo","New York", "Amsterdam","Sydney", "Moscow", "Hong Kong"),
Rain_mm = c(950, 1174, 838, 1215, 707, 2400))
Rainfall
```

```
##         City Rain_mm
## 1 Montevideo     950
## 2   New York    1174
## 3  Amsterdam     838
## 4     Sydney    1215
## 5     Moscow     707
## 6  Hong Kong    2400
```

As you can see a data.frame can mix character columns (City) and numeric columns (Rain_mm). Here I used `c()` to generate vectors which I put in the columns. In addition, the columnns have names, which you can access using `colnames()`: City, Rain_mm

Once you have a dataframe, you can access parts of the dataframe or manipulate the dataframe.

```
# call a column
Rainfall$City
```

```
## [1] Montevideo New York   Amsterdam  Sydney     Moscow     Hong Kong
## Levels: Amsterdam Hong Kong Montevideo Moscow New York Sydney
```

```
# or
Rainfall["City"]
```

```
##         City
## 1 Montevideo
## 2   New York
## 3  Amsterdam
## 4     Sydney
## 5     Moscow
## 6  Hong Kong
```

```
# or
Rainfall[,1]
```

```
## [1] Montevideo New York   Amsterdam  Sydney     Moscow     Hong Kong
## Levels: Amsterdam Hong Kong Montevideo Moscow New York Sydney
```

```
# find a row
Rainfall[Rainfall["City"]=="Montevideo"]
```

```
## [1] "Montevideo" " 950"
```

```
# see the first two rows
Rainfall[1:2,]
```

```
##         City Rain_mm
## 1 Montevideo     950
## 2   New York    1174
```

```
# find a subset
lots <- Rainfall[Rainfall["Rain_mm"] > 1000,]
lots
```

```
##        City Rain_mm
## 2  New York    1174
## 4    Sydney    1215
## 6 Hong Kong    2400
```

3.3.1 Exercise

Using the above examples, can you do the following?

- Extract the column with the rainfall values?
- Extract the row with the annual rainfall at Amsterdam?
- Which cities have rainfall below 1500 mm?

3.4 The working directory

Generally R works from a "working directory". This is the directory on disk where it expects to find files or write files to. You can set this in Rstudio via the menu item "Session" –> "Set working directory", but you can also set this in code. Setting the working directory is useful when you want to access data in files on your computer or the network.

The basic function to use is `setwd("path/to/file")`. The thing to note is that in the path description you have to use "forward /" rather than the standard windows "backward".

```
# set the working directory
setwd("~/Documents/2018/SSEAC_2017/opendata_workshop/Introduction_to_R")

# see some of the files
dir()[1:10]
```

```
##  [1] "2017_BandungIntroductionToR_files"
##  [2] "2017_BandungIntroductionToR.log"
##  [3] "2017_BandungIntroductionToR.pdf"
##  [4] "2017_BandungIntroductionToR.Rmd"
##  [5] "2017_BandungIntroductionToR.tex"
##  [6] "2017_UruguayIntroductionToR.PDF"
##  [7] "2017_UruguayIntroductionToR.RMD"
##  [8] "four_panels.png"
##  [9] "Parana_CorrientesSt.csv"
## [10] "rest_rmd_willem.R"
```

3.5 Reading data from different sources

There are a multitude of functions to read data from the disk into the R memory, I will demonstrate only a few here.

Because a lot of data is stored in comma delimited txt files (such as Excel exports), using `read.csv()` is a good standard option.

Here I am reading in some monthly data from the Concordia station in the Uruguay river in Argentina. This data was originally downloaded from the Global River Discharge Database

```
UR_flow <- read.csv("UruguayRiver_ConcordiaSt.csv")
# check the first few lines (6 by default)
head(UR_flow)
```

```
##   Year Month Flow
## 1 1969     1 7888
## 2 1969     2 5951
## 3 1969     3 4296
## 4 1969     4 4173
## 5 1969     5 4539
## 6 1969     6 4857
```

Previously you would have to save a binary data file, say in `xls` in to `csv` or `txt`. As R progresses, now you could load a dataset directly from its binary format. There are many packages to do such task, readxl package is one of them. You could google your way of the most convenient package to use.

3.5.1 Exercise

- Can you read in the file: "Parana_CorrientesSt.csv"?

4 STATISTICAL ANALYSIS AND DATA MANIPULATION

Now it's time to dig ourselves in to a more technical bits. How to manipulate data so we can perform some analyses on it to answer our research problem. There are, offcourse, base R commands to do the job, but find it easier for us to use `tidyverse` package. This package is actually a combo of several packages written by the same author.

4.1 Packages to use

Much of the power in R comes from the fact that it is open source and this means many people write new code and share this code. The formal way to do this is via "packages", which, once checked and endorsed by the R community, appear in the CRAN repository as a **package**.

Here we might want to use some of the features in the package tidyverse. The other package we will use later is the package zoo

There are two components to using packages. The first is to make sure that the package is installed, for which we can use the function `install.packages()`. Note that the name of the package is a *string* so needs to between quotes `""`.

```
#install.packages("tidyverse")
```

If the package is installed in your personal library, you will need to load the package in R using `require()` or `library()`. There are subtle differences between these two functions, but they are currently not that important. Check the help files.

```
require(tidyverse)
```

4.1.1 Exercise

- Can you load the package zoo?

4.2 Statistical analysis

4.2.1 Summarising data

It is often important to summarise data, for example we might want to know the average monthly flow or the standard deviation of flow. R of course have several functions to deal with this.

4.2.2 Standard statistical functions

Here are some simple examples of standard statistical functions `mean`, `sd` and `cor` (and of course there are many more).

```
# average monthly flow
mean(UR_flow$Flow)
```

```
## [1] 5456.553
```

```
# st dev average flow
sd(UR_flow$Flow)
```

```
## [1] 3491.968
```

```
# subset two years and correlate
flow1969 <- UR_flow[UR_flow$Year==1969,]
flow1970 <- UR_flow[UR_flow$Year==1970,]

cor(flow1969$Flow,flow1970$Flow)
```

```
## [1] -0.3164468
```

4.2.3 Using aggregate()

Another useful function is `aggregate()`, which allows you to apply a function over data frame and particular across different factors. Here is an example of summing the Uruguay river flow by year.

```
# aggregate to annual flow
(annual_flow <- aggregate(UR_flow,list(Year=UR_flow$Year),sum))
```

```
##    Year  Year Month   Flow
## 1  1969 23628    78  53753
## 2  1970 23640    78  52130
## 3  1971 23652    78  66648
## 4  1972 23664    78  99562
## 5  1973 23676    78 103070
## 6  1974 23688    78  47130
```

```
## 7  1975 23700    78  68075
## 8  1976 23712    78  53500
## 9  1977 23724    78  73650
## 10 1978 23736    78  41700
## 11 1979 23748    78  61047
```

Note that the parentheses around the statement means that the result of the statement is printed.

4.2.3.1 Exercise

- Can you calculate the standard deviation of the monthly flow by year?

4.3 Data manipulation (using `tidyverse`)

Make sure you've done this.

```
install.packages("tidyverse")
library(tidyverse)
```

4.4 Important commands

The following list is the important commands to remember:

- `select()` select columns
- `filter()` filter rows
- `arrange()` re-order or arrange rows
- `mutate()` create new columns
- `summarise()` summarise values
- `group_by()` allows for group operations in the "split-apply-combine" concept

Let's open this dataset. It's a water quality data in `csv` format.

```
chemdata <- read.csv("semarang_chem.csv", header = TRUE)
head(chemdata)
```

```
##         ID              Area Year      Lat     Long UTM_east UTM_north
## 1 SB_185     PT. Ny.Meneer-1 1992 -6.95854 110.4562   439936   9230800
## 2 SB_273            PT. INAN 1992 -6.98295 110.4432   438500   9228100
## 3 SB_283    Obs. SD Kuningan 1992 -6.96437 110.4161   435500   9230150
## 4 SB_271   PT. Sango Keramik 1992 -6.98006 110.3133   424150   9228400
## 5 SB_270      Dolog Mangkang 1992 -6.97099 110.2934   421950   9229400
## 6 SB_278       Hotel Santika 1992 -6.99333 110.4292   436950   9226950
##   UTM_zone Depth    WL Elev TDS   ph   EC    K   Ca   Mg  Na  SO4    Cl
## 1      49M    96 23.37    2 424 7.85  704  6.5 10.0  6.0 120 92.5  35.7
## 2      49M    94 14.40   20 964 7.31 1372  5.0  8.7  8.0 150 65.5 136.4
## 3      49M   150 15.25    2 531 7.24  759 10.0  3.7 29.0 148 71.7  37.2
## 4      49M    65 31.49   25 279 6.94  408  6.0 41.2 12.0  30 10.9  13.9
## 5      49M    NA 19.80   20 381 7.23  557  9.0 45.0 16.4  50 38.3  57.0
## 6      49M    86  7.86    5 901 7.15 1341 10.0 53.7 29.4 160 14.9 224.0
##    HCO3  Bal               Aq                Fac
## 1 222.0 -1.5           Garang  sodium bicarbonate
## 2 171.4 -1.7 Quaternary marine     sodium chloride
## 3 379.1  2.8           Garang  sodium bicarbonate
## 4 261.7 -4.3            Damar calcium bicarbonate
## 5 231.9 -1.6            Damar calcium bicarbonate
```

```
## 6 355.9 -0.6 Quaternary marine  sodium bicarbonate
```

4.4.1 select()

Suppose you want certain columns for your analysis. Use `select()`. In `tidyverse` package, we could use pipe operator `%>%` to give series of command. Instead of using many brackets.

```
chemdata %>%
  select(Lat, Long) %>%
  View("chemdata")
```

Or you want multiple columns `Lat`, `Long` until `Depth`. Use `select()` function.

```
chemdata %>%
  select(Lat, Long:Depth) %>%
  View("chemdata")
```

Or you want multiple columns Lat, Long until Depth, but you don't want UTM_zone. Use `select()` function.

```
chemdata %>%
  select(Lat, Long:Depth, -UTM_zone) %>%
  View("chemdata")
```

4.4.2 filter()

You want to select all data from Damar Formation. Use `filter()` function.

```
chemdata %>%
  filter(Aq == "Damar") %>%
  View("chemdata")
```

4.4.3 arrange()

Sorting out data by Aq and Fac. Use `arrange()` function.

```
chemdata %>%
  arrange(Aq, Fac) %>%
  View("chemdata")
```

4.4.4 mutate()

Making new columns, for instance, calculating the ration between Ca and Na. Use `mutate() function`

```
chemdata %>%
  mutate(ratio_Cana = Ca / Na) %>%
  View("chemdata")
```

4.4.5 summarise()

Making a summary from your data. Use `summarise()` function.

```
chemdata %>%
  summarise(mean_TDS = mean(TDS),
            max_Cl = max(Cl),
            min_Cl = min(Cl),
            total = n())
```

4.4.6 group_by()

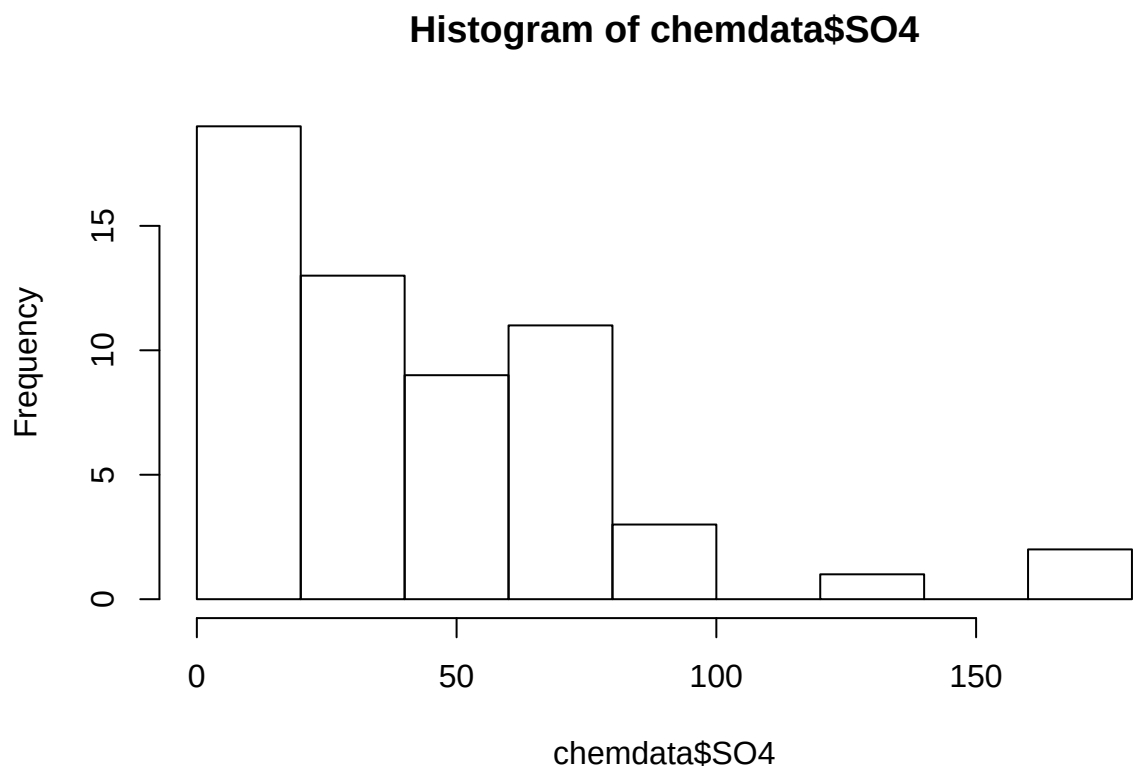
Sorting out the data based on certain order. Use `group_by()` function.

```
chemdata%>%
  group_by(Aq) %>%
  summarise(mean_TDS = mean(TDS),
            max_Cl = max(Cl),
            min_Cl = min(Cl),
            total = n())
```

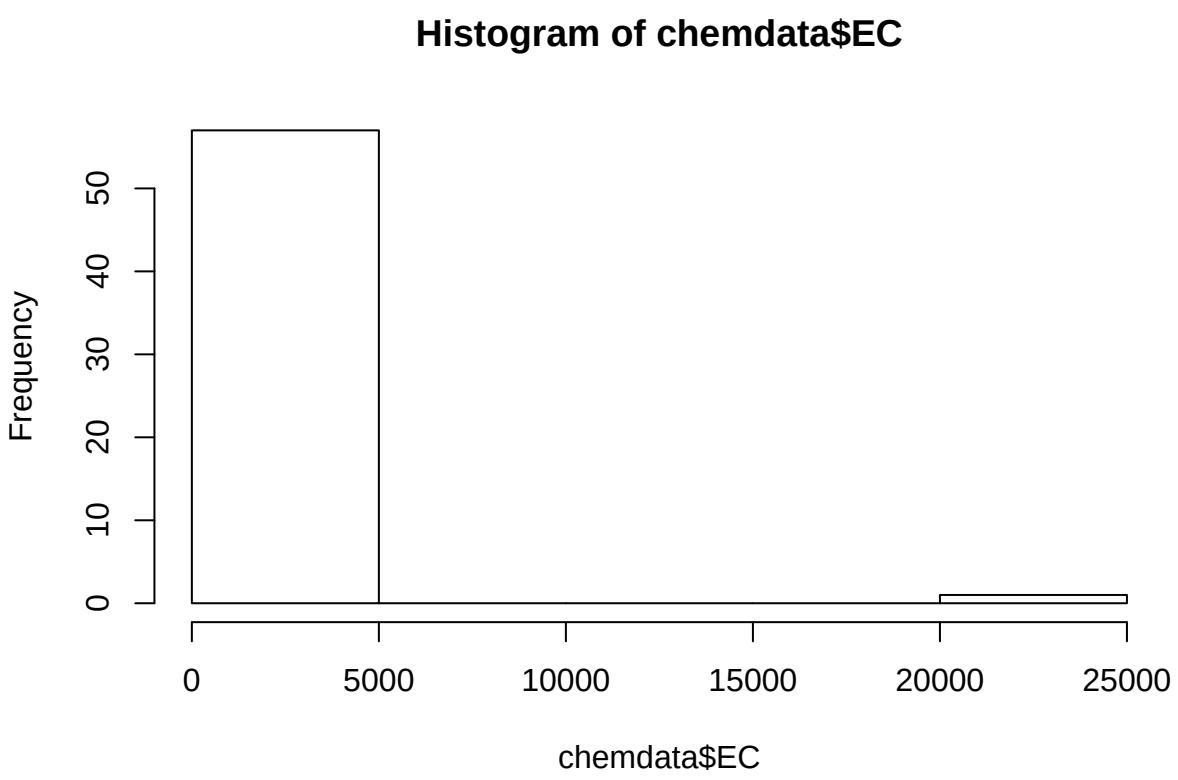
#PLOTTING

R is good at plotting. There are many ways to create a plot. So you just have to choose which one is the easiest for you. One way is using base R plotting engine. Like these plots.

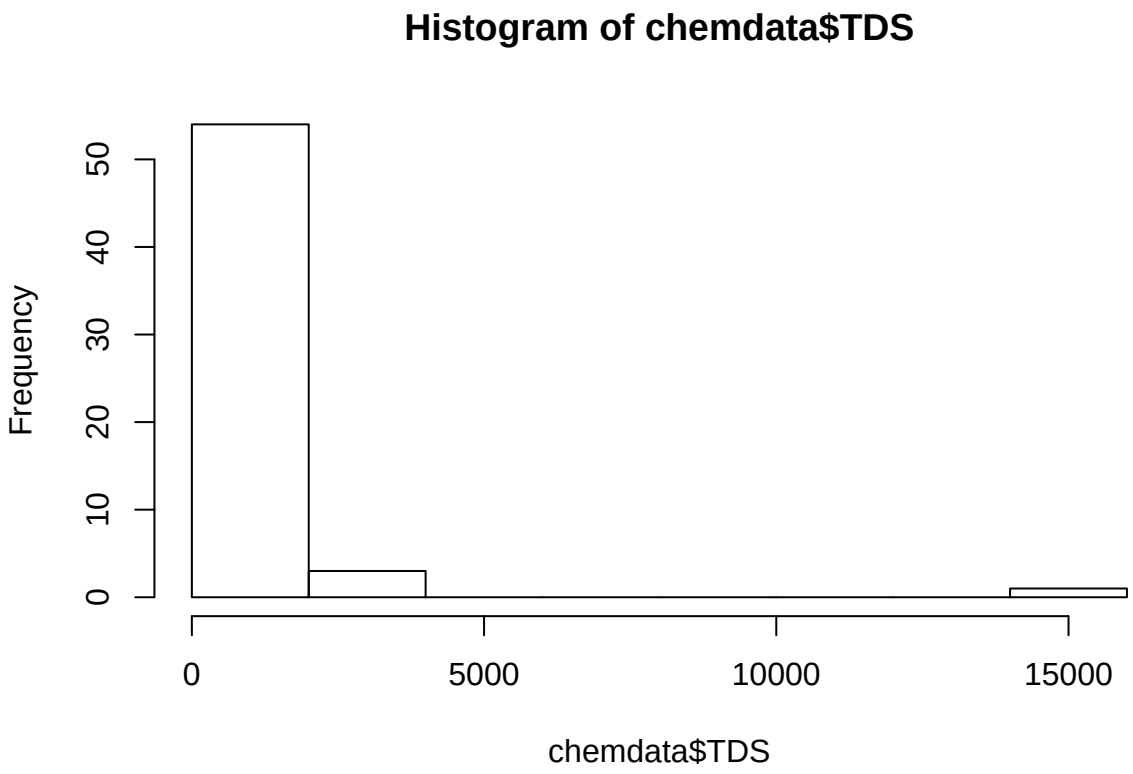
```
hist(chemdata$SO4)
```



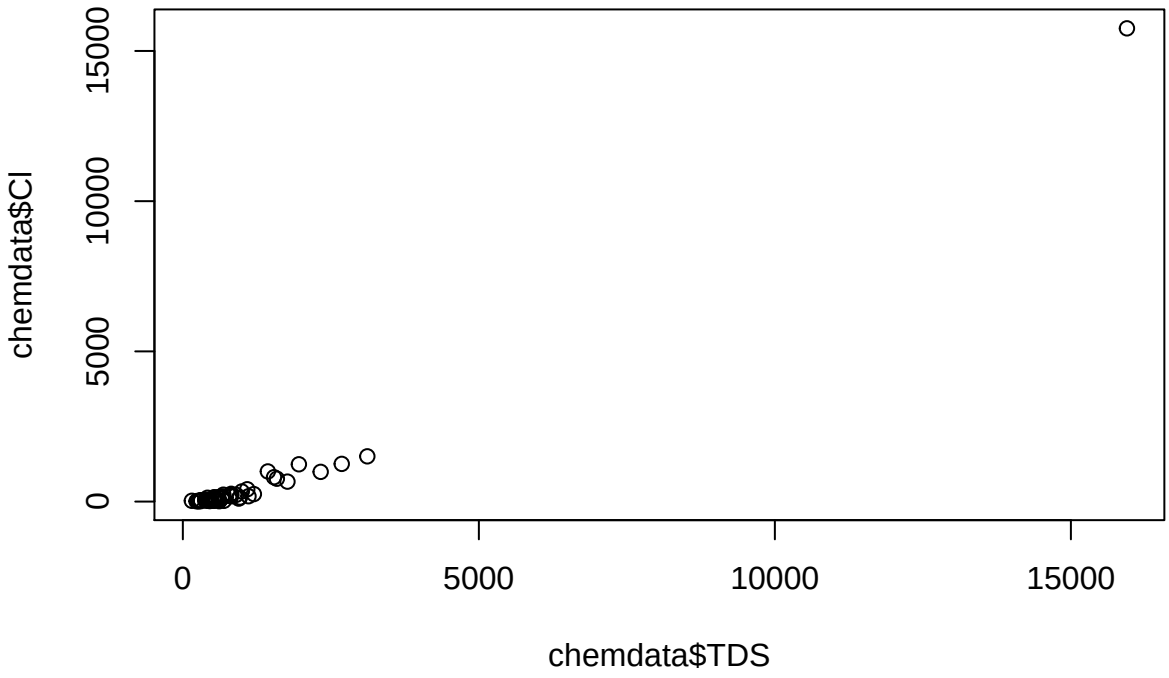

```
hist(chemdata$EC)
```




```
hist(chemdata$TDS)
```

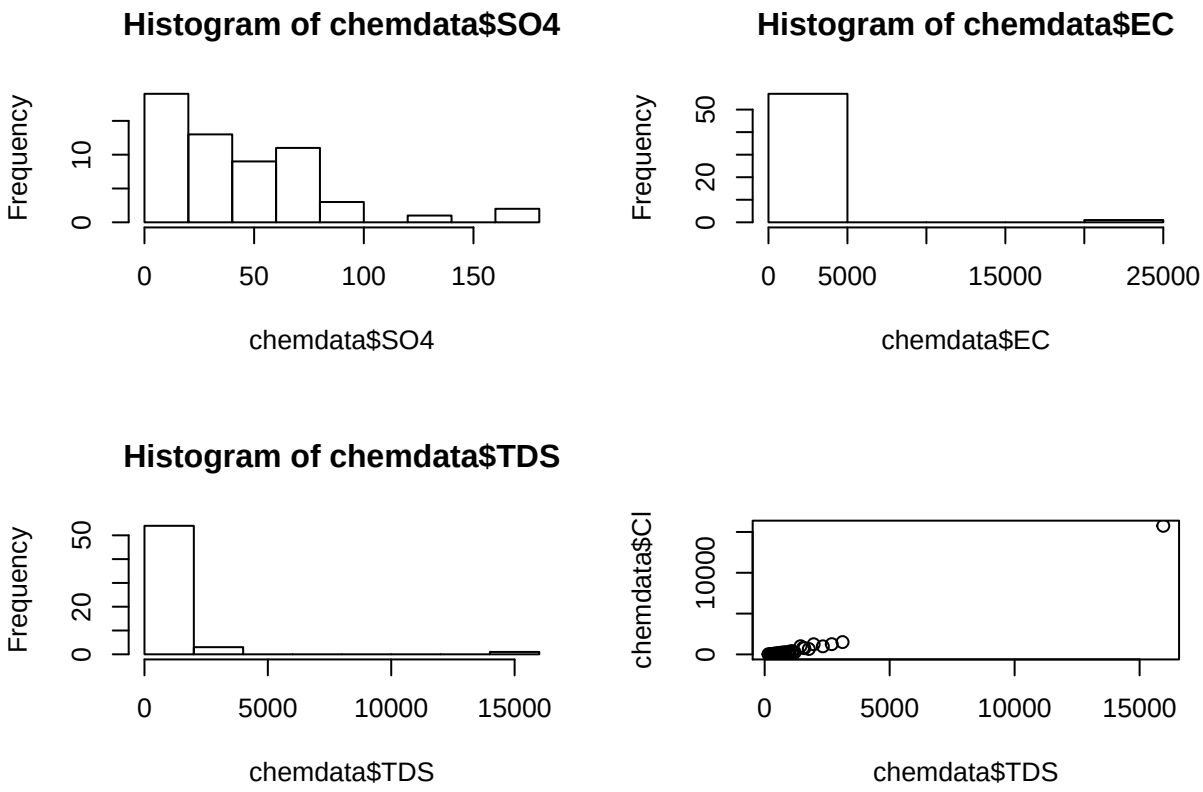



```
plot(chemdata$TDS, chemdata$Cl)
```



Maybe you want to look at them in one panel.

```
par(mfrow=c(2,2))
hist(chemdata$SO4)
hist(chemdata$EC)
hist(chemdata$TDS)
plot(chemdata$TDS, chemdata$Cl)
```



You could always tweak the plot to suits your needs. There are many resources about plotting in R, like:

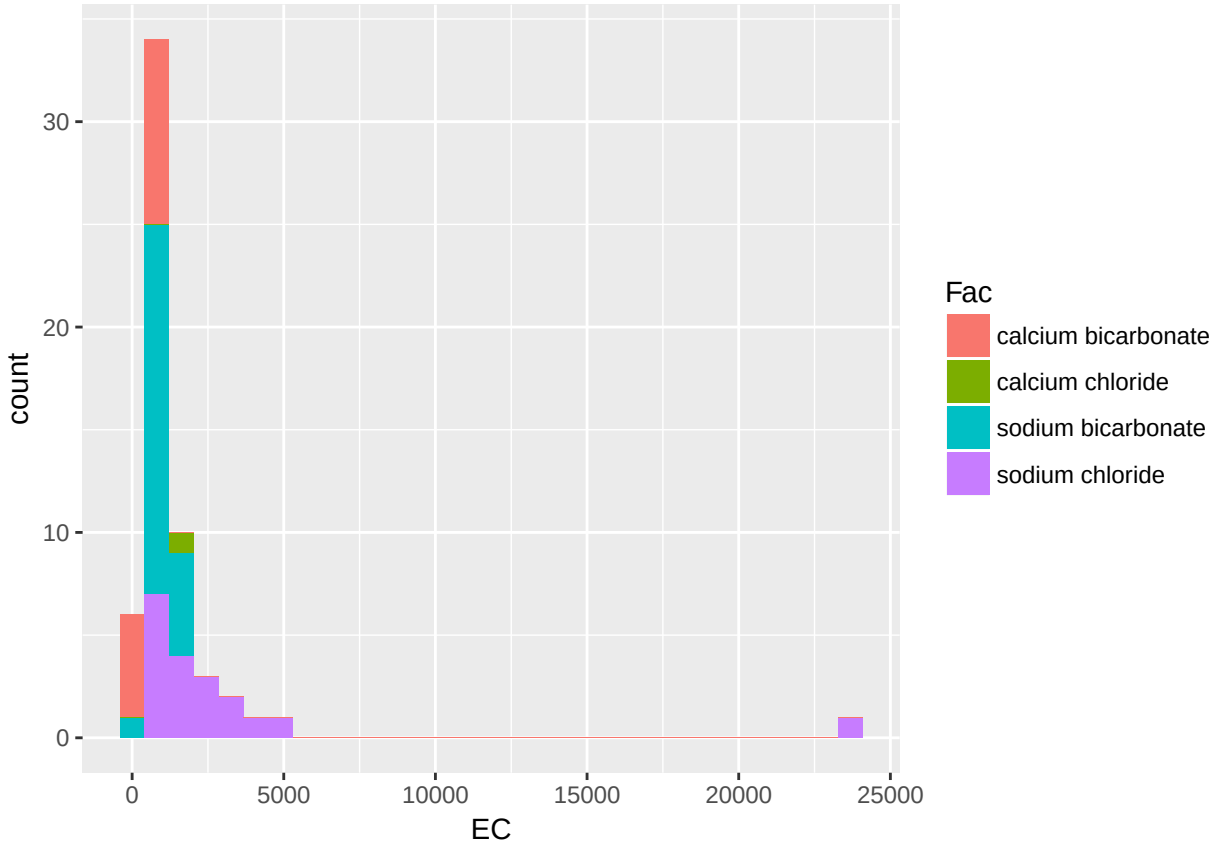
- Producing Simple Graphs with R,
- Quick R.
- and more.

Or you could you `ggplot2` plotting engine from `tidyverse`.

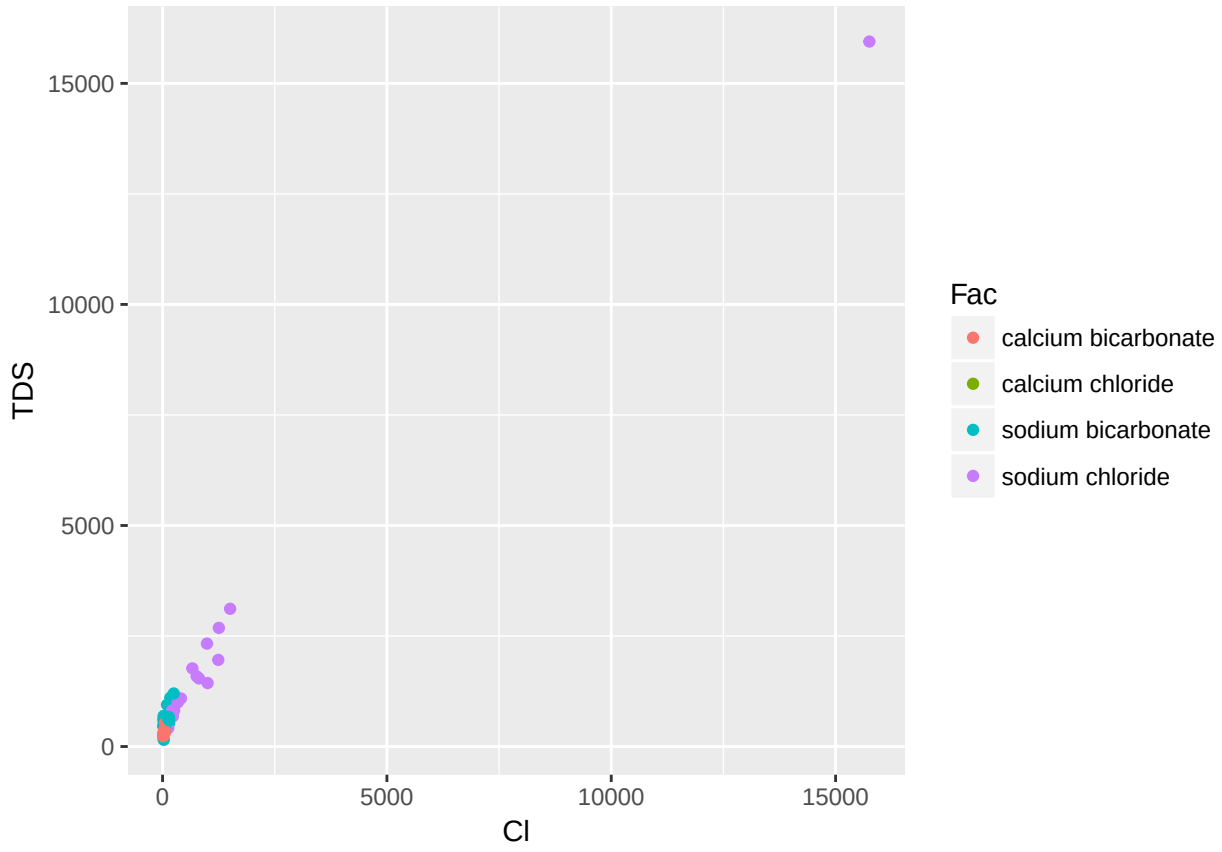
```
library(ggplot2)
```

```
ggplot(chemdata, aes(EC, fill = Fac)) +
  geom_histogram()
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```




```
ggplot(chemdata, aes(Cl, TDS, colour = Fac)) +
  geom_point()
```


15000
10000
5000
0
TDS
0
5000
10000
15000
Cl
Fac
calcium bicarbonate
calcium chloride
sodium bicarbonate
sodium chloride

END OF DOCUMENT