# PRACTICAL: USING R FOR SIMULATION ANALYSES

K Degeling, M van de Ven, H Koffijberg

---

## INTRODUCTION

This introductory practical session will provide an introduction to RStudio, which provides a more user-friendly interface to R. The subset of R functionalities that are essential to performing simulation analyses will be introduced. **It is essential to read all the reading materials carefully and make sure you fully understand these functionalities, as they will all be used in further practical sessions.**

Although all further practical sessions are custom made for this course, this introductory practical mostly uses publicly available sources, utilizing the wealth of information and tutorials on R and RStudio available. One excellent resource is the book R for Data Science by Hadley Wickham and Garrett Grolemund, which is free in a digital format. Another great resource is the R Programming for Data Science book by Roger Peng, which is freely available and wihch you can consult for information on basically everything in the list below.

This practical session covers the following aspects of R and RStudio:

1. R Studio and its different windows
2. Loading and saving data files
3. Installing and loading packages andlibraries
4. Defining and working with variables, vectors, matrices, data.frames, and lists
5. Sub-setting/selecting from vectors, matrices, data.frames, and lists
6. Basic calculations and transformations on variables, vectors, matrices, data.frames, and lists
7. Working with built-in functions
8. Defining custom functions
9. Making basic plots
10. Efficient vector and matrix-based calculations using the “apply” functions
11. Parallelization of computationally demanding tasks
12. Defining and working with formulas
13. Fitting (logistic) regression models
14. Fitting parametric survival models
15. Simulating random numbers from parametric distributions

## 1: R Studio and its different windows

### 1.1: RStudio and its windows

A short introduction on how to install R and RStudio, as well as an overview of the different windows in RStudio, is provided by datascience+ at this link. Use this webpage to familiarize yourself with RStudio.

NB: if you are unable to install RStudio on your laptop, you can also use RStudio Cloud.

### 1.2: Executing code from the Console

The Console can be used to evaluate code using R directly. Read and work through Sections 4.1 and 4.2 of the R for Data Science book to familiarize yourself with running code from the Console directly: 4 Workflow: basics.

### 1.3: Writing scripts and running code from the Source Editor

Although running code from the Console can be convenient for small tasks, it becomes cubersome for more complex and multiple lines of code. Most of the time, you will be writing code in the RStudio Source Editor and running it from there. Read and work through Sections 6.1 and 6.2 of the R for Data Science book to familiarize yourself with writing and running code from the Source Editor directly: 6 Workflow: scripts. Some further information is available from the RStudio website: Editing and Executing Code.

NB: The example in the R for Data Science book uses the `dplyr` and `nycflights13` packages, which you can install by running the code below in your Console. This will be explained in more detail later.

```
install.packages(pkgs = c("dplyr", "nycflights13"));
```

## 2: Loading and saving data files

RStudio provides an excellent introduction to loading data files into the Environment: Importing Data with RStudio. Notice that once you have imported a data file, the code that RStudio used to do so is visible in the Console. Next time, you can run this code to load the dataset (from the Console or the Source Editor). Alternatively, you can modify the code to load different data or adjust how the current data is being read.

Once you have created one or multiple objects in RStudio, these can either be saved to, for example, `.csv` and `.txt` files using approaches described above, or they can be stored as R data `.rda` files. The latter option is very convenient if you want to use objects in other or later sessions, but don't want to regenerate them. The R-bloggers Load, save, and .rda files tutorial provides a good introduction to the `load()` and `save()` functions in R, and it also illustrates how you can set your working directory. The working directory is a file path on your computer that sets the default location of any files you read into R, or save out of R.

NB: In the example, the `Orange` dataset is used. Use the code below to load this dataset into the Environment. To print the dataset, simply run the code `Orange`. You can also click on the Orange dataset in the Environment to open it in the Viewer (equivalent to running the code `View(Orange)`).

```
# Load the Orange dataset into the Environment
data("Orange");

# Print the Orange dataset to the Console
Orange;

# View Orange
View(Orange);
```

## 3: Installing and loading packages/libraries

A wealth of packages is available for R. Each of these packages includes a set of functions that can be used. Generally speaking, if you want to do something in R, there is a package out there that does it for you. It is up to you, however, to ensure the package does what you want it to do. There are no guarantees. Therefore, it is good practice to inspect the package documentation before using the package.

An overview of all packages installed on your machine, and which of those are currently loaded into the Environment, is provided under the "Packages" tab in the Files/Plots/Packages/Help/Viewer window.

In Step 1 you already installed the `dplyr` and `nycflights13` packages, but perhaps without an understanding of how this works. Have a look at the r-bloggers Installing R Packages tutorial and/or run `?install.packages` to inspect the function itself to get a better understanding.

Note that you can also install packages without running any code using the "Install" button withing the "Packages" tab.

If you want to use a package, you will need to load it into the Environment using the `library()` function, as was illustrated earlier. You will need to do that every time you open RStudio.

## 4: Defining and working with variables, vectors, matrices, data.frames, and lists

Different types of objects can be used when working with R. The most common are: variables, vectors, matrices, data.frames and lists. The cyclismo.org Basic Data Types tutorial provides a nice introduction to these object types. However, there is a bit more to it to perform simulation analyses in R. Therefore, use the following tutorials by datamentor.io to get a better understanding of working with these data objects:

- vectors
- matrices
- data.frames
- lists

After going through these tutorials, it should be relatively forward to do the following:

1. Define a variable `n_samples` to be 10, representing the number of samples
2. Define a data.frame `df_data` with column `x1` that represents a sequence from 1 to `n_sample`
3. Add a column `x2` to `df_data` which represents two times the value of `x1`
4. Change the 4th number in `x1` to `20` and update `x2` accordingly
5. Add a column `y` to `df_data` calculated as `y = x1 + (x2 / 3)`
6. Save objects `n_sample` and `df_data` in a list called `list_analysis`

You will find answers to these exercises in the answer file.

## 5: Sub-setting/selecting from vectors, matrices, data.frames, and lists

The tutorials in the previous step already included some basic operations with data objects. However, selecting from the objects was done in a rather simple way. When performing data analysis in the real world, it is more likely that you will use a condition to select a subset of the data in an object. This operation is also referred to as "logical subsetting", and is what this step will address. Two ways of subsetting data using logical statements will be discussed, the "base R" way using standard R functions as discussed in the previous tutorials, and the "Tidyverse" way that uses a set of functions developed specifically for data cleaning and manipulation in R.

On a side note, the "Tidyverse" comprises a collection of packages and can almost be considered a data-science movement. The book R for Data Science is part of the "Tidyverse". If you are interested, more information can be found on the Tidyverse website.

It is up to you which way you prefer most and you will use. Generally speaking, you should be good starting off with the "Tidyverse" way, as code written using the "Tidyverse" functions is easier to read. These functions are convenient to use and fast, and would often require some heavy coding in "base R" (especially to get the same efficiency). However, if some advanced data processing needs to be done, there might not be a function within the "Tidyverse" to do so and you might need to return to "base R". Of course, there other approaches towards handling data in R, but that is beyond the scope of this practical session.

However, it is good to have an understanding of the "base R" way, so you know what is going on behind the convenient "Tidyverse" functions and so you can check results. Afterwards you can benefit from the convenient functions the "Tidyverse" provides.

### 5.1: Base R

This Subsetting Data tutorial provides a short introduction to subsetting using logical statements.

Using the informatino from that tutorial and the `df_data` data.frame you defined earlier, print the values for `y` corresponding to observations for which `x1 > 3` and `x2 < 18`. The answer to this you will find in the answer file.

### 5.2: Tidyverse

The package `dplyr` is part of the "Tidyverse" (and hence also part of the `Tidyverse` package) and provides the very useful functions `select()` and `filter()` to select variables and filter observations from data frames. The functions do not work on matrices, so if you want to use it on a matrix you will first need to transform a matrix to a data.frame. This is very easy using the `as.data.frame()` function.

Chapter 5 Data transformations of the R for Data Science book provides a great introduction to some of the basic functions within the "Tidyverse", including the `filter()` and `select()` functions. Work through Section 5.2, which discusses the `filter()` function and conditions. Especially the discussion of conditions and the impact of missing values (`NA`) is very helpful. Also work through Section 5.4, which covers the `select()` function. Section 5.3 covers the `arrange()` function, which is helpful but not crucial. The `mutate()` function that is discussed in Section 5.5, however, is essential and provides a usefull alternative to do the basic operations you performed in Step 4. Although we will get to that in the next step, you might as well give this section a go already.

Using this knowledge and the `df_data` data.frame you defined earlier, print the values for `y` corresponding to observations for which `x1 > 3` and `x2 < 18` using the `filter()` and `select()` functions. Please be reminded that you will need to have the `tidyverse` package installed and to have it loaded using the `library()` function. The answer to this you will find in the answer file.

## 6: Basic calculations and transformations

You already did quite some calculations on and using matrices and data.frames so far. However, those have all been according to the "base R" way. Given the information you read about the `mutate()` function, adapt the code you wrote in Step 4 according to the "Tidyverse" way.

## 7 & 8: Working with and defining functions

R provides many useful functions. We have used many already, for example for loading data and packages. R packages also provide useful functions, like the ones you have used from the `tidyverse` package. This datamentor.io tutorial provides a useful introduction to using built-in functions, with a focus on how to provide values for function arguments. Although the functions that R and its packages provide might get you a long way, you will find yourself writing code that you will repeat throughout one project, or across multiple projects. If that is the case, it might be worthwhile defining your own custom function. As you will experience during the course, especially when building a simulation model, defining functions will make your simulation code easier to read, debug, and update. Hence, "If you need to do it more than once, define a function to do it for you!"

The following tutorial provides a short introduction to defining your own functions. The use of custom functions is not complicated, but it is important to understand for your progress throughout the tutorials. You can easily define your own custom function. Functions can use arguments, but they do not need to. Functions can also return a value, but they do not need to. However, all functions have the same code structure:

```
print_function <- function() { # Naming and defining the function

  #Body of code, i.e., the set of instructions

  print("Hello World")

  # End of body of code

}
# Executing the function
print_function()
```

Thus, all functions start by calling `function()`, which is followed by a set of instructions, i.e., the code that the function should evaluate when called. In this case, executing `print_function()` just prints out "Hello World" to the console. Note that `print_function()` in this example does not use any arguments. That is, it does not use any input parameter(s) in the evaluation of the set of instructions encapsulated within the function. If you want to use an argument in your custom function, you need to define it within `function()` and you can name the argument however you like. Let's say you want to create a function that divides two numbers. You can hard-code the numbers you want to divide directly into the function:

```
divide_function <- function() { # Using no arguments

  #Body of code, i.e., the set of instructions

  10/2

  # End of body of code

}
```

```r
# Executing the function
divide_function()
```

However, hard-coding it like this offers very limited flexibility and it would not make much sense to create a function for this operation. If you would specify function arguments for the numerator and denominator, you would be able to divide any number you'd like with the same function:

```r
divide_with_args_function <- function(numerator, denominator) { # Specifying arguments

  #Body of code, i.e., the set of instructions

  # Referencing the specified arguments
  numerator/denominator

  # End of body of code

}
# Executing the function while providing values for the function arguments
divide_with_args_function(10, 2)
divide_with_args_function(numerator = 10, denominator = 2) # Is equal to the above line
```

It is good practice to test your custom functions before using them in other parts of your code. You can do that by executing the functions in the console and inspecting whether the output is as expected. For more experienced (R) programmers, you can write unit tests and can include code within the function body that evaluates whether the function can work with the provided arguments.

### Create your own

Based on the skills and knowledge you have acquired so far on the use of functions in `R`, define a function called `my_integer_sequence` that has two arguments: `n_sample` and `from`. The `n_sample` argument represent the number of integers in the sequence and has no default value, i.e. should always be provided when calling the function. The `from` argument represents the value at which the sequence should start and should have a default value of `1`. Since you can use standard R functions and custom functions within functions, use the standard R `seq()` function to generate the sequence. You will find the answer in the answer file.

Now you will write a function `select_greater_than()` that will use your `my_integer_sequence()` to define an integer sequence, but then only return the subset of that factor greater than a prespecified number. This prespecified number is defined by the third argument of the `select_greater_than()` function, which should be called `threshold` and has a default value of `0`. You will find the answer in the answer file.

## 9: Making basic plots

Making graphics in R has turned into real art and there are many approaches to it. Again, the two most commonly used approaches are either the "base R" approach and the "Tidyverse" approach using `ggplot()`. If you are new to plotting in R, it might be worthwhile immediately learning `ggplot()` as this function/package contains a range of tools you can use to plot almost everything. If you are already used to the `plot()` and associated plot functions in "base R", there is no need to learn `ggplot()` specifically for this course. The focus here will be on a quick introduction to `ggplot()`.

Section 3 Data visualisation of the R for Data Science book provides a great introduction to `ggplot()`. Some of the information may be a bit to detailed at this point, so there is no need to understand everything. What you do want to understand is how to make a basic scatter plot and histogram.

You may consult the help function to learn more about the following functions:

- `geom_line` for plotting a simple line
- `labs()` to change the titles for the plot in general, axes, and legend
- `scale_x_continuous()` and `scale_y_continuous()` to modify the axes (including the limits)

If you are keen to become a `ggplot()` expert, this tutorial provides a detailed discussion and demonstration of different types of plots.

Use your already existing "base R" plotting skills or newly acquired `ggplot()` skills to make the following three plots using the `df_data` data frame you defined earlier (our go on this is available in the answer file):

1. Scatter plot of `x1` on the horizontal axis and `x2` on the vertical axis
2. Line plot of `x1` on the horizontal axis and `y` on the vertical axis
3. Line plot of `x2` on the horizontal axis and `y` on the vertical axis

## 10: Efficient vector and matrix-based calculations using the "apply" functions

So far you have performed operations to specific values within a matrix or data.frame, or a specific row or column/variable of an object. However, in practice we often want to apply a specific function to each row or column in a matrix or data.frame. Although something like that could be implemented using a `for` loop, for example, R provides a set of functions that perform such operations typically much faster. This set of functions is called the apply-family and is part of "base R".

This guru99 tutorial provides a very general introduction to the apply functions. This nicercode tutorial provides a good demonstration of the `lapply()` and `sapply()` functions (there is no need to read the "tapply and aggregate" section or beyond).

Besides performing operations to matrices and data.frames, the apply-family can be used to replace a loop that is used to perform something multiple times. For example, you can use it to run a simulation model multiple times and record the output without specifying what the structure of that output is. Most importantly, there are "multi-threaded" counterparts of the apply functions, which allow you to run code in parallel very easily. How you can parallelize your code will be discussed later.

To practice using the apply-family functions, perform the following steps (answers are provided in the answer file):

1. Use the `apply()` function to calculate the mean values for all columns in your `df_data` data.frame
2. Define a vector `x_values` as an integer sequence from `1` to `5`, and use both `sapply()` and `lapply()` so that:
    - For every value of `x_values` an integer sequence is generated using your own `my_integer_sequence()` function, with `n_sample = 100` and `from` set by the respective value of `x_value`
    - The mean of the integer sequence is returned

## 11: Parallelization of computationally demanding tasks

When analyses become increasingly complex, typically the computational burden of these analyses also increases. There are different ways of dealing with computational burden, one of which is to parallelize your analysis so that you can run multiple iterations at the same time, each using one core of our CPU. In such a multi-threaded approach, multiple instances of the analysis run independently and, hence, not all types of analyses can be run in parallel. However, running a simulation model multiple times is a perfect example of an analysis that is very suitable for parallelization.

A convenient way to make your code scalable to multiple CPU cores is by using one of the parallel versions of the apply-family, e.g. `parSapply()` or `parLapply()`. So if your code is written using an apply-function in which a computationally demanding analysis is performed (such as a simulation run), it can be easily adapted to run in parallel. Although documentation about parallel programming in R is less well developed compared to other aspect, this tutorial provides a good introduction, and together with consulting the help function using `?parallel::clusterApply` (the `clusterApply()` function within the `parallel` package), it should provide you with sufficient information to write some code that runs in parallel.

Adapt the code below in such a way that the "bootstrapping" and "analysis" are performed in parallel.

```
# We will use the convenient sample_frac() and summarize() functions from dplyr
library(tidyverse);

# First we will simulate a data.set of participants
n_participants <- 45;
df_participants <- data.frame(
  ID = 1:n_participants,
  age = rnorm(n = n_participants, mean = 65, sd = 8),
  weight = rnorm(n = n_participants, mean = 70, sd = 2)
);

# We will bootstrap the data 10,000 times to get the distribution about the means
# - this is the part you want to run in parallel
n_bootstrap <- 10000;
system.time({
  mat_analysis <- sapply(X = 1:n_bootstrap, FUN = function(i_bootstap) {

    # Get a bootstrap sample
    df_bootstrap <- df_participants %>%
      sample_frac(size = 1, replace = TRUE);

    # Get the summary statistics
    output <- df_bootstrap %>%
      summarize(
        mean_age = mean(age),
        mean_weight = mean(weight)
      ) %>%
      unlist();

    # Return the output
    output;

  })
})

# The rows and columns are swapped
```

```r
dim(mat_analysis);

# So let's swap them back and make it a data.frame
df_analysis <- as.data.frame(t(mat_analysis));

# Now we can use an apply function to get the mean and empirical confidence intervals
apply(df_analysis, 2, function(col) {

  c(CI = quantile(x = col, probs = 0.025),
    mean = mean(x = col),
    CI = quantile(x = col, probs = 0.975))

});
```

## 12 & 13: Defining formulas and fitting (logistic) regression models

So far, the analyses that have been performed in this practical session are rather simple. However, R is a strong tool for more sophisticated analyses, such as regression modeling. You will use such statistical models in the discrete event simulation you will develop over the next few days. Most regression-type models in R are to be defined using formulas that describe the relationship between the dependent and independent variables.

This tutorialspoint tutorial provides a short introduction to linear regression. Try to replicate the analysis yourself.

This r-statistics.co tutorial provides a more in-depth example and explanation should you want to practice more.

Concerning logistic regression models, this R-bloggers How to perform a Logistic Regression in R tutorial provides a good introduction on how to use the R `glm()` function to fit logistic regressions by specifying the argument `family = binomial(link = 'logit')`. The data that is used in this tutorial is included in the files you have downloaded from Dropbox. Thus, there is no need to register for an account to download the data. Try to replicate the analysis yourself. Please be aware that the approach they use to impute missing data is not the approach we would recommend.

## 14: Fitting parametric survival models

Modeling time-to-event data is an important part of developing discrete event simulations. This topic will be discussed in detail during the coming days, but it would be helpful to familiarize yourself with two functions that can be used to fit so-called "parametric distributions" to time-to-event data. Our two favorite functions to fit such distributions are the `fitdist()` function of the `fitdistrplus` package for simple analyses, and the `survreg()` function of the `survival` package for the somewhat more complex analysis.

This tutorial helps you to become acquainted with the `fitdistrplus` package.

After the `fitdistrplus` tutorial, consulting the help function for information regarding the `survreg()` function (and corresponding `Surv()` function to write a survival model formula) should give a good understanding of how this function works.

## 15: Simulating random numbers from parametric distributions

Since this course is about simulation, there is an addutional step after fitting parameteric distributions to time-to-event data. That is, simulating data from these distributions. Standard R functions, such as

`runif()`, `rnorm()`, `rweibull()`, etc., can be used to do so, similar to the use of `predict()` to predict from `survreg` objects directly. More detail about the concept of drawing random samples from distributions will be provided throughout the course, but it is useful to familiarize yourself with the concept already.

This datamentor.io tutorial provides a brief introduction to simulating random numbers using standard R functions.