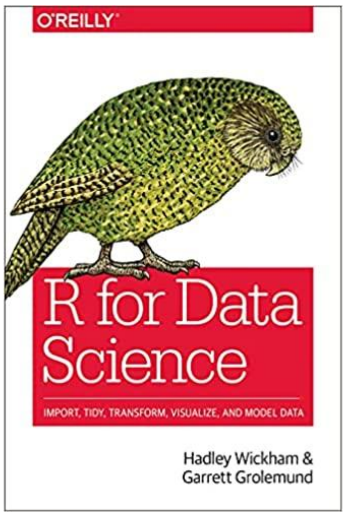
Crash course in R for data science

Version GW.20200617

Lecturer: Dr. Guilherme Maia de Oliveira Wood

Appointments:

Date	Contents
April 7th **9:00-12:00**	Part 1: 9:00 – 10:00 Introduction to the course • Literature, Moodle, course requirements, evaluation Creation of batch files • Good practices writing batch files Part 2: 10:00-11:00 R for text processing • Loading text data • Cleaning text data (REGEX) • Harvesting internet data • Part 3: 11:00-12:00 Sentiment analysis • VadeR • SentimentAnalysis • Syuzhet • Sentiment.ai


O'REILLY
R for Data Science
IMPORT, TIDY, TRANSFORM, VISUALIZE, AND MODEL DATA
Hadley Wickham & Garrett Grolemund

Course description

The R-software is well-suited for work in data science as it combines high flexibility regarding statistical data analysis with good capacity to process large amounts of data. The following book is the main reference for our course.

This book presents the most important uses of R in the realm of data science. Particularly, data organization techniques are described.

When empirical data are collected and analyzed, the requirements for interpreting statistical tests are not always met. Statistical tests of these requirements are often incorrect or non-existent. Under these conditions, one is forced to look at the data itself to look for measures such as confidence intervals. This is the core of the application of bootstrapping methods.

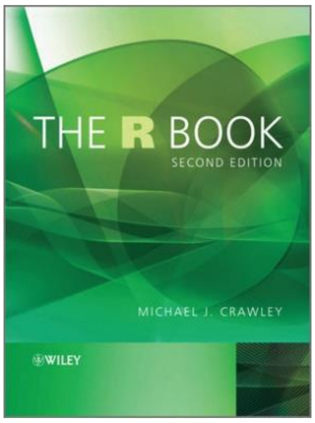
Mastering these skills gives you a much deeper understanding of empirical data as well as tools for assessing the level of evidence and increasing the replicability of psychological examinations.

Some important objects and functions in R

Guilherme Maia de Oliveira Wood

Institute of Psychology, University of Graz

For a general overview of R commands, read chapter 2 of the book The R Book, by Michael J. Crawley. Particularly chapters 1 to 6 are essential to consider yourself literate in R. A good understanding of the notes in this document is conditional upon a read of at least chapters 2 to 6 of this book.



The CRAN project: It is the repository for the whole R project. There are specific web pages for each R package (the many thousands of them). On each page there is a reference manual containing the list of command lines in each package as well as a short description of contents and sometimes useful examples of use. Some packages also offer the so-called vignettes. Vignettes contain very useful examples of use of the different packages. It is always a good idea to consult and read vignettes.

Search Machine for R: There is a very important website called R-seek (https://rseek.org/), which is a search machine only for R. Every aspect of R, language, functions, new packages, etc., are in there. Use that frequently to learn more about R.

Blog with a Facebook channel: There is also the blog R-Bloggers to help you to keep up with every R related topic (https://www.r-bloggers.com/). In the blog new creative uses of libraries is exemplified, there are also launchings of new libraries with nice examples, many contributions show how to use R to solve problems in data analysis. The diversity of contributions makes it always a fascinating read and an inexhaustible source of nice code examples.

Publications on new R packages: There is an internet journal called The R Journal, which publishes exclusively on new R packages (https://journal.r-project.org/)

Reference manual: Every R library is created and stored online following the same structure. Every library you can install is accompanied by a reference manual, a description of each and every one of the functions included in that library, the compatibilities and dependencies with other libraries, and (sometimes) useful examples of the use of the different functions. Compatibility and inheritance of object structure work really well in R. Many of the troubles one meets using Python do not occur in R. Nevertheless, be warned that many statistical procedures are implemented with partially overlapping contents in several libraries. Therefore, loading several libraries with similar functions may generate confusion and unexpected outcomes.

To work with R, you need its latest version and I can recommend to install R-Studio as well to make writing code easier and prettier. Because simulations usually rely upon parallel computing, under windows one should always install the R version for 64-bit. Every six months there is a new R version. From time to time you should get a newer version (once every two years is fine).

Most libraries are kept by independent researchers or researcher groups. Some libraries get discontinued for this reason, but this is an exception. In my career I only faced this problem once.

##

General aspects of R programming

A general schema of the software structure of R is depicted in Figure 1. R is run by means of scripts (batch files), which are basic text documents containing R code. You can write batches in R or in R-Studio, which is a nicer GUI for R. R-Studio can be downloaded at the following site for free: https://rstudio.com/products/rstudio/download/

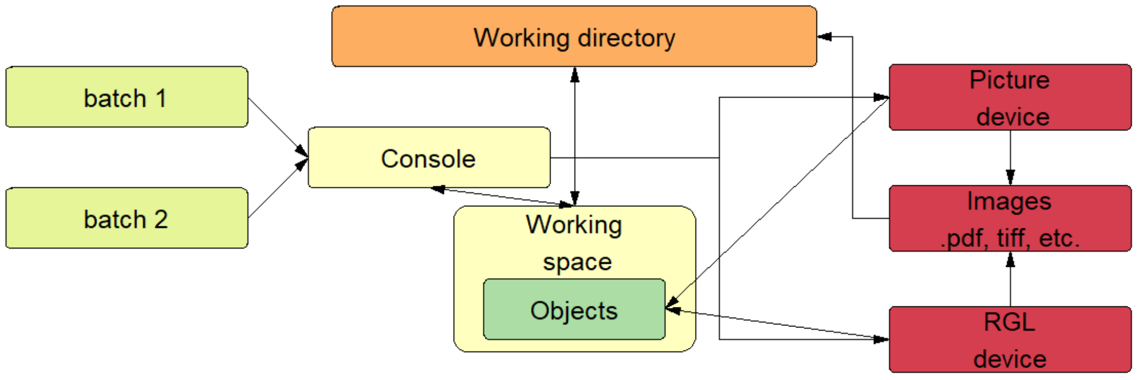


Figure 1 illustrates the general structure of R. Batch files provide instructions to the console. The console creates and manipulates objects defined in the active working space. All objects are stored in the working space, which can stay active for just the current session or be saved for posterior utilization. The console can open further devices such as those employed to generate plots or RGL graphic objects. Those graphic objects can also be stored in the working directory (unless other directory is chosen to save the images).

is the sign for a comment in the R language. Commands or text typed after a **#** are ignored by R when executing lines from an R script. Use it to comment your scripts in a clear, objective, and systematic manner. When marking your work, I will consider the quality of your commenting as well, so do not forget to make it good!

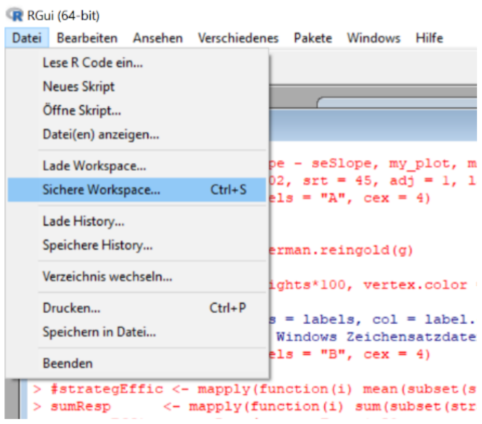
In R, the key combination **Ctrl + R** executes the current script line or those you selected in the script. When you want to ask for help regarding some function or command line, just type a question mark before the function or command line. In R it will direct you to a web page containing the description of the function. In R studio, it will open a side window with exactly the same content as the web page.

To install a new package (there are several thousands of them) you may use **install.packages**("name of the package"). When a package has been installed, before using it in a given R session, you have to activate it using the command **require**(name of the package) or **library**(name of the package). Many commands of different packages have exactly the same name and some packages overrule others so that one should be careful to understand which package is driving functions such as ANOVA() or plot() at that specific context. Package compatibility is one of the strengths of R, it works wonderfully in almost every case so that one really has to be careful to understand who is doing what in an R script loading several libraries simultaneously (this is the usual case!).

Hint: Even as an experienced R programmer, I ALWAYS have R-seek open at all times to search for examples or explanations of specific pieces of code or specific functions. I recommend you to do the same.

R is an object-based language. This means that whatever you read or create in R gets a name and belongs automatically to one category of objects, which in R language is called a **class**. Each class has its properties: character objects are non-numerical, so that you cannot apply numeric functions to objects from this category. This means, you are not allowed to divide or multiply character objects. Integer is a class of numeric objects containing only integer numbers. You cannot assign a number with decimals to integer (because R is a highly interpretative language it will under different circumstances do one of two things: change the properties of the object and make it suitable to receive decimals, or it will round down your number with decimals). Beyond integer, you also have other numeric objects with different degrees of precision (which of course consume much more memory to be stored) and are suitable to store decimal, irrational, and complex numbers. Furthermore, there are also **data.frames**, **matrices**, **arrays**, **lists**, and **vectors,** which may contain numeric information with different degrees of precision (integer to float) as well as non-numeric information (i.e. **character**, **factors**, etc.).

Every time an R session is started, a workspace is activated, which is described by the total amount of memory reserved for R calculations and is filled with R objects. As long as a session is active, all objects created in that session will remain available for further calculations, etc. When that session is closed, all those objects are deleted. After calculating long simulation studies and obtaining a large amount of numerical data, it is always a good idea to **save the workspace** containing relevant objects such as effortfully computed data. When one saves a work space, all objects in that workspace can be saved and reloaded in a posterior time, or in another computer. It is not necessary to save a workspace, if computations can be executed again easily.



```
pe - seSlope, my_plot, m
02, srt = 45, adj = 1, 1
els = "A", cex = 4)

erman.reingold(g)

ights*100, vertex.color

s = labels, col = label.
 Windows Zeichensatzdate
els = "B", cex = 4)
> #strategEffic <- mapply(function(i) mean(subset(s
> sumResp       <- mapply(function(i) sum(subset(str
```

R is a highly interpretative language. This means, R makes guesses about what you want to have as an output to avoid that the program crashes because of trivial definitions. This makes the language much more economic and easier to read and learn and most of the time it is a truly innocuous feature. But this feature also may produce outputs quite different from what was originally desired because R may convert objects from one class to another without telling you. This can obviously make the program crash. One common reason why R code does not work properly is because objects change their properties with the change of class. Still, R is very useful because when it crashes it is quite easy to figure out the reason and to fix it. The errors and warnings one sees at the console when something does not work are almost always very informative.

How to write good batch files

In the following lines I try to give you some hints based on my own experience as a non-professional programmer on how to organize your work and write intelligible scripts you may be able to understand and reuse even years after writing them.

Naming objects and promoting generality: Give names to the entities that are part of your script. If you define sample size, create an object with an understandable name such as

sampleSize <- c(200)   # c(seq(200, 2000, 200))

From this simple line of code, we can learn many things: First, give your objects the most precise names you can. If the variable is an index for file names, one possible name for the object is for instance “fileNameIndex”. If it is the whole list of file names, you may call it for instance “fileNameList”. “sampleSize” explains already which kind of information this object conveys. This is a very good practice you should incorporate to your programming habits in general. The operator “<-“ assigns the value on the right to the object on the left and is more general than “=” so that it is preferable to use the <- operator most of the time. Finally, we come to the c() command. This c means “concatenate” what means as much as “combine” in this context. c() can have a single element as in the example above as well as a vector of many different values, such as the input after the # above. If you start a simulation with a certain sample size, and late you want to expand it, adding other values to an existing object sampleSize using the c() command makes the structure of the script more *general*. If sampleSize contains more than one single value, to access single values one at a time you will have to specify their *index* (see indexable objects, below).

Length: Do not let batch files grow indefinitely, split them into smaller files when they grow larger than a given number of lines you will perceive as too long. Organize the different jobs so that the output of one batch file fits the required input for the next one. Think modular: this means to develop pieces of R code you can reuse. Give batch files understandable names you or someone else may be able to understand even a couple of years later. The more advanced your research questions, the more important is keeping everything as understandable as possible.

Use of comments: Start a batch file always with a description of what it does. Use comment marks # to do that. Use comments to describe relevant parts of your programming code. Particularly more complex programming lines have to be well commented. Think of others reading your code when writing comments. When working with vectorial computing some lines of code can become really long and completely incomprehensible. In these cases, you should employ as much text as you need in many lines around this very complex line of code to explain what is going on there. Remember to be as precise as possible when describing the computation operations being performed by each line of code.

List of libraries: List all the libraries you will employ at the top of your batch file. At some point you have needed so many of them that you will not remember their name anymore, so put their names there where you can find them easily. You may also want to create a list of all libraries you have ever used for documentation. Some libraries are really useful for very specific purposes. We use them once and later it is very hard to find their names again. Under these circumstances, such list of all libraries may be useful. You can for instance download the reference manual of each library use have ever used to your reference manager program to be able to localize them later, in case you forget their name. Some libraries get discontinued (this is rare, I only experienced it once in the last 15 years), so that sometimes only older R versions will be able to perform some specific calculations.

Working directory: Set the working directory explicitly, and consciously in each batch file.

Indentation: Indentation is not a putative aspect of the R language as it is for Python. Nevertheless, indentation can help understanding better the structure of batch files. Use it!

Plots: Plots with professional quality require a considerable number of code lines. It is a good idea to have different scripts only for the production of high-quality images, since even relatively simple plots with text elements, annotations, interactive elements, customized plot elements, interesting attractive colors, etc. consume forcefully many, many lines. Sometimes we need some plots to check data quality at some point of the analysis. Keep these plots simple, create other ones in separate files for the purpose of publication. There are many libraries to generate plots in R: lattice, ggplot, ggplot2 and many more for more specific purposes. They follow their own internal logic. It is a large time investment, creating nice plots in R, think of that when organizing your time. Save good examples in your private script folder to spare time the next time you want to create those plots again in the future. Plots can be saved as images with a high resolution. It is always a good idea to learn how to create them for your thesis or publications.

R object classes (types)

Indexable objects: Indexable objects have one or more dimensions and may contain more than one element. To access single elements in such an object, it is necessary to specify their index within the object. Please consider the two-dimensional object myMatrix, which is a matrix with 2 rows and 2 columns.

myMatrix <- matrix(c("apple", "banana", "grape", "peach"), nrow = 2, ncol = 2, byrow = T)

by typing myMatrix on the console one will get the output

"apple" "banana"

"grape" "peach"

By typing *myMatrix[1,]* you obtain

"apple" "banana"

By typing *myMatrix[, 1]* you obtain

"apple"

"grape"

By typing *myMatrix[2, 1]* you obtain

"grape"

Several times you will extract parts of larger indexable objects to make calculations and insert the outcomes as part of another large indexable object. Being acquainted to how this works in R is very important to use this programming language for statistical programming. The following objects are all indexable.

character -> All types of text. This is useful for the organization of file names, some variables are also coded as characters, when they contain text (e.g. different strategies reported by different participants).

factor -> A factor is a vector of different categories. This variable is very useful to calculate general linear models (ANOVAs, mixed-effect models, etc.), because it indicates a "factor", a categorial variable with different levels. Numerical variable can be converted easily to factors with the command **factor**(yourNumericalVariable).

data.frame -> Data frames behave much like SPSS .sav files. They are objects with columns and rows. Each column may belong to a different type, one can be character, the next one numerical, the next one can be factor. This is quite practical when different types of data should be grouped in the same data structure. To know how big a **data.frame** is, one uses **dim**(**data.frame**).

tibble -> Are an improvement of data.frame retaining some positive properties of data.frame and abolishing some undesirable ones. For data mining tibbles are the preferred data format.

Matrices and arrays -> arrays are matrices with more than 2 dimensions. All elements of matrices or arrays have to belong to the same class: either numerical only, or character only, or factor only, etc.

Matrices are the base for every calculation with several sets of data. Arrays are very useful to store MRI data. To know how big a **matrix** or an **array** is, one uses **dim**(**matrix**) or **dim**(**array**).

Hint: For most computations in R, it is particularly advantageous to deal with objects with as few columns as possible. If your matrix is much larger in one dimension, define this as your rows. When filling a matrix with values, there is an instruction called byrow = T/F, which determines whether values are filled row by row or column by column into a matrix. When making large calculations one should always pay attention to the dimensions of matrices and arrays. If one can transpose a matrix for it to have only a few columns and many, many rows, this will usually speed up considerably calculations.

list -> List is a very flexible format. Lists can contain anything: matrices, integers, character, arrays, other lists as well as combinations of all these elements. One can store very complex objects containing many parts using lists. The output of most statistics functions in R are lists. One can give labels to specific elements of a list and use this label to retrieve the contents of that list element. Lists are also very important for parallel computing. To know how big a **list** is, one uses **length**(**list**).

There are hundreds of other object types in the R language, which may be based on those mentioned above or have other characteristics, such as images, graphs, etc. Most of these objects are related with specific libraries and their properties, elements, structure, and characteristics can be examined by using the command line **str**().

Commands and functions

setwd("computer address") -> defines the address "computer address" as the working directory. It works much the same as in other programming languages and is the address where the program searches for and stores data -unless explicitly stated otherwise.

list.files() -> lists all files and folders in the specified directory. If none is specified, it reads the working directory.

grep("some text", object) -> Works the same as in other programming languages: it searches for a character sequence with specified properties (which can be very sophisticated!) in the object "object", which is a list of characters (e.g. a list of files from the working directory).

read.table() -> is a very useful command line to read data in the R environment. It works best when input data is organized in a nice matrix and the file is some type of text file (e.g. tab-delimited). Small amounts of data, let's say a data matrix with less than 500 columns and as many row as you want can be read easily from a file using **read.table**. To read other types of data into R there are other commands as well (e.g. to read text inputs one may use **read.lines**(), to read MRI data into R one can use command lines from libraries tailored to analyze such data). There are also hundreds of other possibilities, R is a very, very flexible tool to read data. One can load images from photos or stream data from a website, there is almost no limit for which kind of data can be read into R.

One can also easily write output data from R. Specific libraries can help creating data readable in Matlab, SPSS, etc. The command line most useful to write simple readable files in ASCII is **write.table**(). One important hint to obtain clean outputs when using **write.table**() is to suppress the writing of row numbers by writing row.names = F in the command line (e.g. **write.table**(dataObjectToBeSaved, "myFile.txt", row.names = F)).

c() -> means concatenate. It combines different elements. Related and useful commands are **rbind**() and **cbind**(), which combine inputs by adding them to different rows or columns. Do not abuse it!

```
A <- c(2, 3, 5, 4, 7)

>A

2, 3, 5, 4, 7

B <- cbind(c(2, 3), c(4,5), c(6, 4))

>B

2       4       6

3       5       4
```

Hint: When using **c**(), **cbind**(), **rbind**() to combine inputs with different formats, such as a numerical vector (i.e. a numerical variable) and a list of participant codes (i.e. a **character** or a **factor** variable), R will convert all the inputs to the lowest level of computability among all object formats -if- the object receiving the **cbind** operation is a **matrix**. Character objects, for instance, do not allow all operations allowed to numerical inputs. Therefore, the variable containing numbers will be converted to character. If it is a **data.frame**, you may concatenate different objects without lost of any of their specific properties.

nchar("character string") -> counts the number of characters a string contains. When applied to a list of character strings, such as a list of file addresses, it will count the number of characters in each element of the list.

substr() -> extracts a part of a character string. It requires as input the character string, at which character it should start the extraction and at which it should stop. It is very useful to extract specific information from file names and file addresses among many other operations on text.

```
A <- c("C:/downloads/myStats/test.txt") # is a character string with nchar = 29

B <- substr(A, nchar(A)- 8, nchar(A)-5)   # B = "test"
```

A **for** loop -> one very practical way to achieve recursive programming in R is a for loop. A loop executes a series of commands a give number of times and stops after the last step in the interaction was achieved. It works as the c++ loops. The syntax is as follows: you define a iteraction variable i, an interval of values for the variable I, and define some commands to be executed a couple of times, enclose that with { } and you have a for loop!

```
nIterations = 1000 # variable nIterations is a number with the value 1000

sampleSize = 100   # variable nIterations is a number with the value 100

myCorrelations <- matrix(NA, ncols = 1, nrow = nIterations) # is a (originally empty) matrix with one

                                                 # column and nIteration rows

for(i in 1:nIterations){                         # begins the for loop (our first bootstrapping!)

a <- rnorm(sampleSize, 0, 1) # defines a random variable a with the size sampleSize, mean = 0, sd = 1

b <- rnorm(sampleSize, 0, 1) # defines a random variable b with the size sampleSize, mean = 0, sd = 1
```

myCorrelations[i,] <- cor(a,b)} # calculates the correlation between a and b and sends outcome to

Hint: one should never let objects “grow” during the execution of **for** loops because it is very time-consuming. This means for instance, adding new rows or columns to a matrix. Instead, one should define each object with exactly the same size as it is supposed to have at the end and step-by-step when the corresponding elements of these objects are calculated, they replace the “empty” parts of the object. Proceeding in this way computations are much more efficient than otherwise adding new rows or columns to the output object at every new iteration in a **for** loop.

apply(data, dimension, function) -> executes some function for all rows or all columns of a matrix or data.frame at once. Usually they are more efficient than a **for** loop exactly because the apply command line defines the size of the output object at the begin of computation and is the basis for vectorial computing in R. Extensions of that to lists are called **lapply**. A multivariate version of apply is called **mapply** and has the same effect of a for loop for a function. The output of a **mapply** command line can be a matrix or a list, depending on the contents of the function it executes.

iRepeats <- 1000   # variable iRepeats with the value 1000

Sample1 <- matrix(rnorm(100, 0, 1), ncols = 20, nrows = iRepeats)

Sample2 <- matrix(rnorm(100, 0.5, 1), ncols = 20, nrows = iRepeats)

myPs <- mapply(function(j) t.test(Sample1[j,], Sample2[j,])$p.value, 1: iRepeats)

Object myPs contains 1000 p-values resulting from the comparison of two samples of normally distributed variables (each row of objects Sample 1 and Sample 2 contains 20 observations).

colMeans(), **colSums**() -> calculate the means or sums od matrix columns and speed up computations, since these command lines are vectorized.

str(“object”) -> shows the structure of an object. Most useful commands in R produce quite complex objects with several elements, sometimes they are long lists of output parameters. To extract information from those complex objects it is necessary to understand their structure. The command str will show us the names, localization, type, size and other properties of a complex object.

class(“object”) -> informs us to which class an object belongs. R interprets our inputs A LOT, this means, several times R will change the class of a given object without telling us that. To check the properties of an object, one can ask class whether an object is a matrix, a data.frame or something else. This command is important to check when something went wrong. Many times R changes the class of an object and some computation stops working.

dim(“object”) -> informs us the dimensions of an object. It only works for objects with more than one dimension, matrices, data.frames, arrays. Lists and vectors can be checked with the command length(“object”).

by(“dependent variable”, list(“independent variables”), function) -> Executes a function on a dependent variable separately for each level of one or more independent variables. It is very practical to calculate means for each participant in each experimental condition.

na.rm -> removes missing data from calculations. It is an argument passable to different functions (e.g. the **by**() function), that means it will be part of different command lines and is in itself not a command line. Some functions only work properly when missing data are removed from the data set, while other ones will produce different results depending on the status of **na.rm na.rm** = T or **na.rm** = F**.**

function() -> Helps making your code more compact. A function is any set of command lines wrapped together. Functions transform some input into some output using the operations described in the function. To make a function produce some output data, one defines an object to be returned **return**(objectToBeReturned) in the last line of the definition of every function.

Example: How to calculate the mean RT of trimmed data.

```
trim <- function(inputRT, trim) {   # inputRT is a numeric vector containing a munch of reaction times
                                     # trim is an integer describing how extreme (in standard deviations)
                                     # have to be to be removed from the data set
myMax   <- mean(inputRT) + sd(inputRT)*trim   # finds mean + trim*sd
myMean <- mean(inputRT) - sd(inputRT)*trim   # finds mean - trim*sd
trimmedDataIndex <- which(inputRT< myMax & inputRT> myMin, arr.ind = T) # finds the positions of
                                                          # elements fulfilling the conditions in which
return(trimmedDataIndex)  # produces the output of the function
}
```

Usage:

```
indexCoolRTData    <- trim(someData, 3)                    # function(inputRT, trim)
```

rep("element to be replicated", "number of times it should be replicated") -> This function generates replications of elements. "element to be replicated" can be a single integer, character, a numerical vector, and so on.

gl("number of levels", "number of replications") -> is similar to rep, but its output is always a **factor** with the specified number of levels and number of replications.

seq("begin", "end", "step") -> this function generates a sequence of numbers from the begin to the end, which can be positive or negative, with "begin" > "end" and a negative "step" or "begin" < "end" and a positive "step".

scale("numeric values", center = TRUE, scale = TRUE) -> calculates the z-standardization of variables.

round("input", "number of decimals") -> Rounds numerical input down to the specified number of decimal places.

Rescale (psych package)

t(matrix) -> transposes a matrix.

unlist(list) -> transforms a list of whichever complexity in a vector of single elements.

sample() -> Samples a data set according to specifications (sample size, probability that specific observations will be picked, etc.)

set.seed("some number") -> When generating data with specific distributions, R picks up some entry at its library of well sampled distributions and uses that to generate new data. This means, one never knows exactly which data will be generated unless one fixates one entry at the library to make results perfectly reproducible. If someone tried the same simulation you run on your computer without

set.seed(), results would be always slightly different. By setting **set.seed**() to the same number, results will be exactly the same, which ever computer is used to run the simulations.

Sys.time() -> reads the internal clock of the computer. This command line is very useful to measure how long calculation of large simulations take, to improve calculation plan.

if() {}**else**{} -> If else conditions are very useful when specifying the circumstances under which some computation should be executed and those in which other computations or no computation at all should be executed.

Hint: When dealing with exceptions in your scripts, the use of if else commands is absolutely necessary to avoid code crashes. One should implement control instances in the script in which some conditions have to be fulfilled before some other command lines, that can cause a crash when fed with wrong input. In simulations and bootstrapping analyzes it is sometimes the case, that samples may not have the necessary characteristics to be fed to an analysis. For instance, when a random sample of questionnaire data only contains one specific value, the variance in this sample will be 0! Feeding this to statistical calculations will produce crashes or at least warnings. For this reason, one always checks whether data fulfill specific assumptions before plugging them into something else. Particularly when dealing with massive computations, one such error can compromise many days of work!

try() -> this command line always embeds another function, which will be executed. In case the function produces an error, this will not lead to a crash of other routines as occurs when functions are used without **try**(). **try**() is very useful for simulations as well as every time one cannot be sure that a given function will get typical inputs all the time. When large simulations are calculated, it may take many hours, even days. When a function crashes, the calculations will be interrupted. Sometimes it is certain, that a function cannot be evaluated and the number of times a certain function crashes is even an active part of the simulation. In these cases, **try**() is very useful, because it prevents a loop or some vectorial calculation to be stopped and generates an output containing each crash as valid input to be analyzed.

Distributions

Generating large number of observations with specific distributions is one of the most useful tools available in R. In R all distributions relevant for statistics are available and help a lot calculating simulation studies. A gaussian distribution with sample size n = 100, mean mu = 0 and standard deviation sigma = 1 can be generated as follows:

n = 100

mu = 0

sigma = 1

myGaussian <- **rnorm**(n, mu, sigma) # generates observations from a Gaussian distribution

Instead of observations, one may be interested in the density of the Gaussian distribution at a given quantile. **dnorm**() gives the density, **pnorm**() gives the distribution function, **qnorm**() gives the quantile function of a Gaussian distribution. The same values can be calculated for a t-distribution, an F-distribution, gamma, beta, poisson, binomial, log-normal, etc. In most cases, the function names are constructed in exact the same way as the normal distribution.

For instance, the following commands give us value of the F-test corresponding to p=0.05 when degrees of freedom are 5 and 2. The quantile function **qf**() of the F-distribution against the decimal value 0.95.

```
> qf(.95, df1=5, df2=2)  # this function returns the F value associated with the 95th percentile
[1] 19.296
```

Another useful distribution is the uniform distribution. When modelling processes, which depend on variable values, which assume values in a given interval, the use of **runif**() to generate these data is advisable. Examples are for instance a set of probabilities in the interval (0, 1) or ages of participants in two age intervals young = 20 to 40 and 60 to 80.

Parallel computing in R

Parallel computing speeds up calculations when computation problems are really big. When calculating Monte Carlo simulations, even a moderate number of replications can already generate really large computational problems. When the same mathematical operations have to be executed over and over again in exactly the same way, it is mostly a good idea to parallelize computations. The amount time spared when parallelization is well placed is surprisingly large. One small computer with eight CPUs working under parallelization can finish calculations 10, 30, even 100 times faster than without parallelization. Parallelization distributes parts of a big task to the different CPUs and collects the results at the end, when the last CPU is ready with its part of the calculation. Usually all processors start and stop working well synchronized. That it works so well is one of the main tasks of the parallelization algorithm.

The Figure below show schematically how parallelization works. The parallelization algorithm distributes the function to be computed “function” as well as parts of the input data “input data” to the different processors of the computer. These execute the function “function” to process the part of the “input data” it has been assigned to and when all processors finish their work, the output is stored in the output object “Output object”. In the parallelization algorithm we will employ, the “Output object” is always of the type **list**.

The speed-up factor one can achieve with a specific parallelization algorithm depends on how much input data one has, how complex are the computations to be performed and how much RAN memory the computer has. When the size of input data approaches the size of RAN, parallelization will not work as well as with smaller amounts of data. Moreover, when computations can become lengthy (e.g. a structural equation model which is just identified and will need a lot of interactions to converge in many instances), parallelization will be less efficient.

When parallel computing is running, one should not spend RAN resources with anything else. Close all other non-essential applications such as Youtube, films, music, as well as all other non-essential software that consumes a lot of RAN and slows down computations. Parallel computing is an optimal occupation for the computer at the office overnight or on weekends. One should calculate beforehand the size of expected outputs before starting parallelization, because running out of memory will shut the computer down and can damage your hardware.

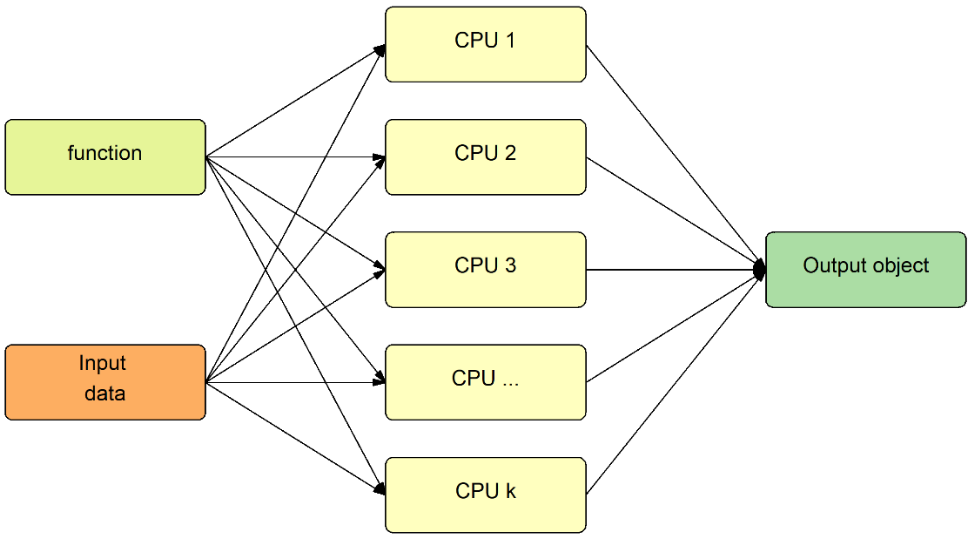


Figure 1: Function and Input data are distributed to all available CPUs, which process parts of the data using the function to produce an output object after finishing all computations.

There are many ways to implement parallel computing in R using different packages. Here I will show only one way how to do that but I encourage you to try alternative ways. I have used the library "snowfall" to parallelize computations.

One of the main challenges when programming simulations lays not on executing the calculations themselves but on dealing with the NUMERICAL and LOGICAL EXCEPTIONS that can emerge when trying to execute specific calculations. When trying to calculate a regression or a mixed-effects model and the does not converge, the structure of the output of calculations may differ from the structure one expects from calculations that can be carried out successfully. It is enough that one such exception happens a single time in the whole bunch of replications to stop all the calculations and collapse. Therefore, one has to implement a safety exit for the software to continue calculations even if at some point the indented calculations cannot and want not to work.

Task 1: Generating correlations

Correlations are very popular in psychological research, particularly in the psychology of individual differences and psychometry. Correlations fulfill many different functions in psychological research, they are employed (i) to determine similarities between different items of a test and investigate aspects of validity and reliability of psychometric instruments, (ii) to characterize undesirable confounding effects in clinical designs, which are due to comorbidities, sampling artifacts, etc, (iii) to show dependencies between brain activity and cognitive performance, (iv) to analyze graph theoretical properties of brain activation or of social interactions, and so on.

When two variables correlate with each other, they show a specific form of linear dependence. When variables correlate, they share information about each other. Knowing one of them reveals to some degree proportional to the strength of the correlation something about the other one. One can also say, these two correlated variables share a proportion of their variance. Pearson product-moment correlations are proportional to the variance shared by two variables. The square of the correlation describes exactly the proportion of variance shared by two variables.

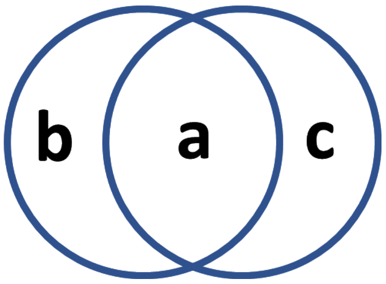
$\sigma^2_{shared\ variance} / (\sigma^2_{shared\ variance} + \sigma^2_{non\text{-}shared\ variance})$

When total variance equals 1, the shared and non-shared variances are scaled as values between 0 and 1 and are easy to understand as proportions of total variance. This is useful when generating artificial data with a specific covariance structure.

The variance shared by two variables can be called *a*, which is a random variable with a specific mean μ and a standard deviation of σ.

When we add two non-correlated variables, the standard deviation of the resulting variable is just the sum of the standard deviations of the two original variables we added. When summing sampled data, results will not be exact in each single replication but they will come very close to the correct results when averaging over a certain number of replications. The relative size of $\sigma_{shared\ variance}$ *in relation to* $\sigma_{total\ variance}$ gives us the strength of the correlation between two variables. For the sake of simplicity, we let all variables to be correlated have exactly the same variance, so that the covariances will all be in the same scale. With this simple fact, we are already able to generate two variables, which share a proportion of their variances. Let's create some correlations:

a <- rnorm(100, 0, sqrt(0.5)) # $\sigma_{shared\ variance}$
b <- rnorm(100, 0, sqrt(0.5)) # $\sigma_{non\text{-}shared\ variance}$
c <- rnorm(100, 0, sqrt(0.5)) # $\sigma_{non\text{-}shared\ variance}$
var1 <- a + b # $\sigma_{shared\ variance} + \sigma_{non\text{-}shared\ variance\ b}$
var2 <- a + c # $\sigma_{shared\ variance} + \sigma_{non\text{-}shared\ variance\ c}$
cor(var1, var2)
> 0.5 # a correlation of 0.5



The Venn diagram on the right underscores that the variance component a is common to both variables var1 and var2, which are represented as the two circles.

But how to generate variables that correlate to a specific degree and present a given covariance structure? One very important fact to acknowledge is that variances are the square of standard deviations. If the aim is to generate variables with – let's say – a variance of 1 and shared variance of 90%, the standard deviation of the shared and non-shared components of both variables will be:

a <- rnorm(100, 0, sqrt(0.9)) # $\sigma_{shared\ component}$

b <- rnorm(100, 0, sqrt(0.1)) # $\sigma_{non\text{-}shared\ component}$

c <- rnorm(100, 0, sqrt(0.1)) # $\sigma_{non\text{-}shared\ component}$

Please note, that 0.9 + 0.1 = 1 to produce the unity variance. It is obvious that the sum of the standard deviations will yield a sum larger than 1. Remember, that you can also use Pythagoras theorem to calculate the variances and standard deviations of the shared and the two non-shared variance components. This will be particularly useful when calculating covariances when using other scales than var = 1.

$\sigma^2_{total} = \sigma^2_{shared\ variance} + \sigma^2_{non\text{-}shared}$

In more complex cases, when the correlations are generated by more than one shared component, one can still determine exactly the strength of these components when using variances = 1.

For instance: $\sigma^2_{total} = \sigma^2_{shared\ component\ 1} + \sigma^2_{shared\ component\ 2} + \sigma^2_{non\text{-}shared}$

Variance of shared component 1 can for instance be equal to 0.5, the variance of shared component 2 equal to 0.4 and the non-shared variance is 0.1 to add to a total variance of 1.

Task 2 Generating a covariance structure

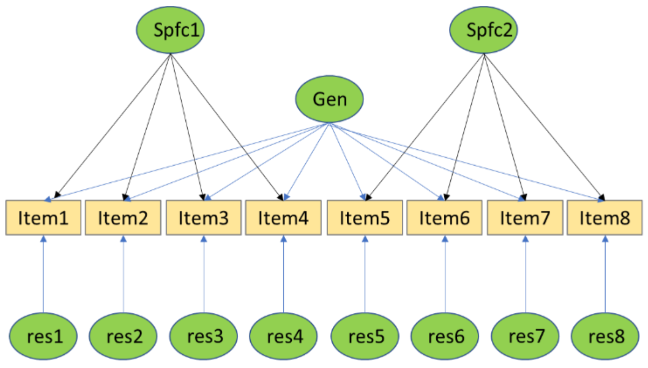
In this task we will learn to generate not a single correlation but a correlation matrix with a specific structure. In this task we will combine different sources of shared and non-shared variance, which resamble realistic data sets.

The following script generates a data matrix with a specified latent structure. Namely, one general factor onto which all observable variables load and two specific factors, onto which four indicators, respectively, load.

```
myBigN    <- 400 # Sample size for data generation
varFaktor <- 10   # Variance of specific factors expressed as % of total variance
varGen    <- 0     # Variance of the general factor expressed as % of total variance
varRes    <- 90    # Variance of the residuals expressed as % of total variance
nVar      <- 8       # Number of variables

factor1 <- rnorm(myBigN, 0, sqrt(varFaktor/100)) #
factor2 <- rnorm(myBigN, 0, sqrt(varFaktor/100)) #
genFact <- rnorm(myBigN, 0, sqrt(varGen/100)) #
factors <- c(rep(factor1, nVar/2), rep(factor2, nVar/2)) #
general <- rep(genFact, nVar) #
residua <- rnorm(myBigN*nVar, 0, sqrt(varRes/100)) #

data <- matrix(c(factors + general + residua), ncol = nVar, byrow = F)
corData              <- cor(data)
```



Covariance structure generated by the script of Task2. Green ellipses are latent variables, which generate the covariance structure we usually estimate by measuring observable variables (orange rectangles).

As can be seen in the figure, the latent structure of the data contains one general factor and two specific factors. Please note that the latent variables as well as the residuals are all uncorrelated. Moreover, in the above version of the script, the general factor has a variance of 0. This means, it does not contribute to generate correlations in this version of the script. To increase its role in the generation of the covariance structure, its variance has to be increased to a number > 0.

Let us have a look at the correlation values generated by this script. To increase readability, we reduce the number of decimals when looking at the correlation matrix.

```
round(corData, 3)
```

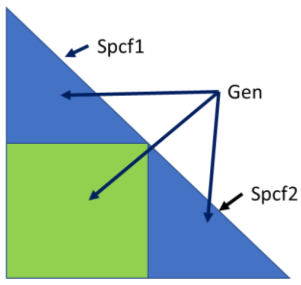
```
[Item1] [Item2] [Item3] [Item4] [Item5] [Item6] [Item7] [Item 8]
[Item1]  1.000  0.089 0.078  0.125 -0.003 -0.091  0.046  0.050
[Item 2]  0.089  1.000 0.102  0.111 -0.039 -0.009 -0.041 -0.007
[Item 3]  0.078  0.102 1.000  0.102  0.014  0.044  0.029  0.088
[Item 4]  0.125  0.111 0.102  1.000 -0.031  0.029  0.054 -0.031
[Item 5] -0.003 -0.039 0.014 -0.031  1.000  0.079  0.117  0.146
[Item 6] -0.091 -0.009 0.044  0.029  0.079  1.000  0.070  0.085
[Item 7]  0.046 -0.041 0.029  0.054  0.117  0.070  1.000  0.135
[Item 8]  0.050 -0.007 0.088 -0.031  0.146  0.085  0.135  1.000
```

Now let us increase the role of the general factor and let it explain 50 % of the total variance, while the specific factors now will explain only 20% of the variance each.

```
varFaktor <- 20   # Variance of specific factors expressed as % of total variance
varGen    <- 50    # Variance of the general factor expressed as % of total variance
varRes    <- 30    # Variance of the residuals expressed as % of total variance
```

Now, the correlation values look like this:

```
             [It1]   [It2]   [It3]   [It4]   [It5]   [It6]   [It7]   [It8]
[Item1] 1.000 0.714 0.695 0.714 0.540 0.503 0.522 0.472
[Item2] 0.714 1.000 0.669 0.691 0.481 0.445 0.462 0.427
[Item3] 0.695 0.669 1.000 0.682 0.529 0.437 0.465 0.457
[Item4] 0.714 0.691 0.682 1.000 0.515 0.461 0.495 0.459
[Item5] 0.540 0.481 0.529 0.515 1.000 0.714 0.684 0.729
[Item6] 0.503 0.445 0.437 0.461 0.714 1.000 0.679 0.701
[Item7] 0.522 0.462 0.465 0.495 0.684 0.679 1.000 0.699
[Item8] 0.472 0.427 0.457 0.459 0.729 0.701 0.699 1.000
```



It is easy to see that the correlation matrix has a general structure similar to this diagram. The regions colored in blue have correlations higher than the region colored green. Correlations observed in the blue regions are generated by specific and general sources. Green one are generated by the general factor alone.

Interestingly, the covariance structure depicted in the matrix above can be expressed as a structural equation model in two different ways, as depicted in the figure below.

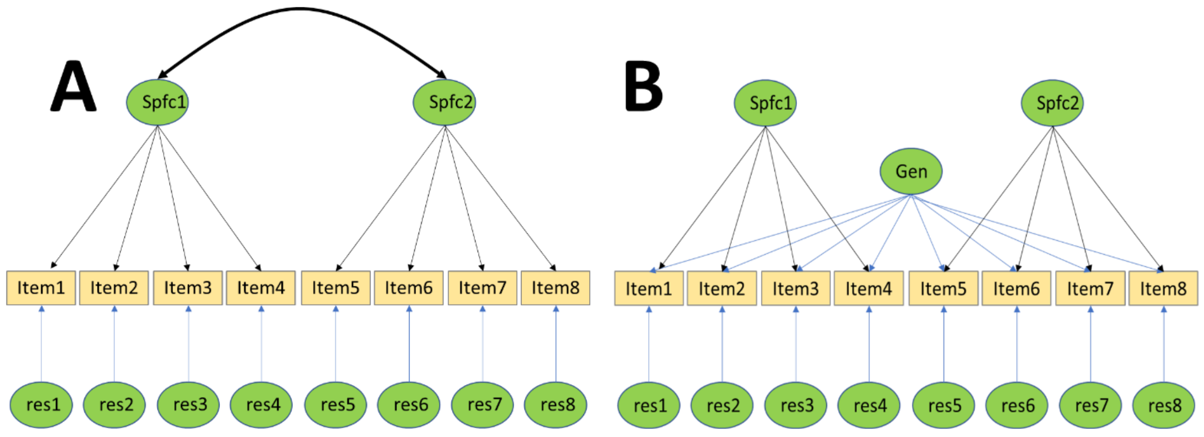


Figure: The latent structure of the correlation matrix depicted above can be expressed as the outcome of two correlated latent factors or as the presence of a general factor and two specific and uncorrelated ones.

In the first case, the theory predicts the existence of two correlated but distinguishable constructs. That correlations belonging to the green region of the diagram above are relatively smaller than those in the blue regions of the diagram are usually interpreted as the divergent validity of these two constructs. In the second case, however, theory makes substantially different predictions. Namely, the two specific factors are uncorrelated -i.e. they are independent from each other- and the correlations one observes in the empirical data are generated by some unspecific factor, which can absorb many different sources of shared variance between items, such as the test situation or the format of the items 1 to 8.

Task 3: The stability and equivalence of psychometric instruments

The comparability of test scores is an important psychometric property of psychological instruments. Particularly when the effects of an intervention are evaluated, plain familiarity with the testing situation should not be able to jeopardize the interpretation of intervention outcomes. Parallel forms of tests are key to exclude trivial explanations for higher test scores in the post-test in comparison to the pre-test. Many test manuals speak of parallel forms of tests but a more rigorous test of parallel forms is necessary to ascertain full interpretability of intervention outcomes. One distinguishes between genuine parallel forms, tau-equivalent tests, essentially tau-equivalent tests and congeneric measurements.

Classical test theory teaches us that test scores are generated by our (platonic) true expression of a given ability and under normal testing situations is blurred by the presence of measurement error, which is uncorrelated with itself across measurements and with the true scores. Under these assumptions, the following additive equation is true:

X = T + E with X = observed scores, T = true scores and E = error measurement

When different versions of a test measure a given construct T, their intercorrelations will reflect extent to which they are equivalent to each other. Parallel tests have the same true score variance and the same error variance. This is the strongest degree of equivalence between test scores.

Let us generate some data for four parallel test versions.

```
nParallel <- 4
varTrue   <- 30
varResid  <- 70
myBigN    <- 100
set.seed(sample(1:100000, 1))
TrueScore <- matrix(rep(rnorm(myBigN, 0, sqrt(varTrue/100)),nParallel), ncol = nParallel, nrow =
myBigN, byrow = F)
Residuals  <- matrix(NA, ncol = nParallel, nrow = myBigN)
for(i in 1:nParallel){set.seed(sample(1:100000, 1))
Residuals[, i]<- rnorm(myBigN, 0, sqrt(varResid/100))}
Data <- TrueScore + Residuals
corData <- cor(Data)
```

In turn, tau-equivalent tests have the same true score variance but different residual variance.

```
varTrue    <- 70
varResid  <- c(30, 40, 50, 35)
myBigN    <- 100
nParallel <- 4
TrueScore <- matrix(rep(rnorm(myBigN, 0, sqrt(varTrue/100))), ncol = nParallel, nrow = myBigN,
byrow = F)
Residuals  <- matrix(NA, ncol = nParallel, nrow = myBigN)
for(I in 1:nParallel){set.seed(sample(1:100000, 1))
Residuals[, i]<- rnorm(myBigN, 0, sqrt(varResid[i]/100))}
Data <- TrueScore + Residuals
corData <- cor(Data)
```

Finally, congeneric measurements are tests correlated positively, which according to theory load on the same latent construct of interest.

```
True    <- 70
varTrue <- c(1, 1.1, 0.85, 0.3)
varResid  <- c(30, 40, 50, 35)
myBigN   <- 100
nParallel <- 4
TrueScore <- matrix(rep(rnorm(myBigN, 0, sqrt(True/100))), ncol=nParallel, nrow=myBigN, byrow=F)
TrueScore <- mapply(function(i) TrueScore[, i]*varTrue[i], 1:nParallel)
Residuals  <- matrix(NA, ncol = nParallel, nrow = myBigN)
for(i in 1:nParallel){set.seed(sample(1:100000, 1))
Residuals[, i]<- rnorm(myBigN, 0, sqrt(varResid[i]/100))}
Data <- TrueScore + Residuals
corData <- cor(Data)
```

Parallel forms of tests are defined Cronbach's alpha, test-retest reliability, split-half reliability are measurements of the internal consistency and stability of psychometric tests. Let us employ a psychometric property of tests as an opportunity to understand better covariance structures.

The significance of a correlation coefficient is given by the following t-test with n-2 degrees of freedom.

$$T_0(r) = \frac{r\sqrt{n-2}}{\sqrt{1-r^2}} \sim t(n-2)$$

Correlations are very sensitive to deviations from the assumptions their interpretability rely upon. Sample size is usually too small to allow trustful interpretation. Many researchers have difficulties understanding how severe the problems with the misinterpretation of correlations can be. Assuming that variables are normally distributed, one can use R to generate correlated data.

set.seed(123456)

a <- rnorm(100, 0, 1) # $\sigma_{shared\ variance}$

b <- rnorm(100, 0, 0.65) # $\sigma_{non\text{-}shared\ variance}$

c <- rnorm(100, 0, 0.65) # $\sigma_{non\text{-}shared\ variance}$

As can be seen in the Venn diagram, a is the portion of the variance of both variables var1 and var2

>0.683

The formula of a Pearson product-moment correlation is depicted below.

$$\rho_{X,Y} = \frac{\mathrm{E}(XY) - \mathrm{E}(X)\mathrm{E}(Y)}{\sqrt{\mathrm{E}(X^2) - \mathrm{E}(X)^2} \cdot \sqrt{\mathrm{E}(Y^2) - \mathrm{E}(Y)^2}}$$

Building regression models

y = b1*x1 + b2*x2 + b3*x3 + resid

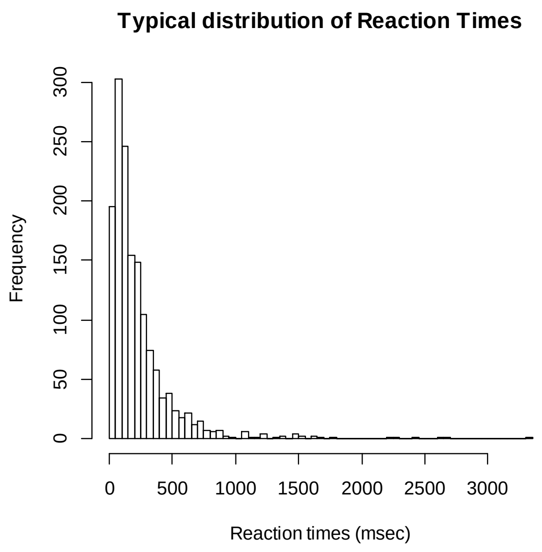
Task 4 Generating group differences

The most common type of statistical analysis involves the comparison of two or more groups for mean differences. This can be translated into a simple t-test for a comparison of two samples, more complex experimental designs with repeated measurements and several different experimental factors, hierarchical designs, mixed effects models with linear or even non-linear terms.

```
individuals <- 25
effectSize <- 0.8
mu <- 100
sigma <- 15
group1 <- rnorm(individuals, mu, sigma)
group2 <- rnorm(individuals, mu+ sigma*effectSize, sigma)
t.test(group1, group2)
```

Task 5 Modelling Reaction Times data

Reaction times usually have a skewed distribution such as shown in the figure below. It is easy to generate these distributions in R, for most of the times individual reaction times follow a log-normal distribution. With the library **stats** one can generate RT-like data using the function **rlnorm**().

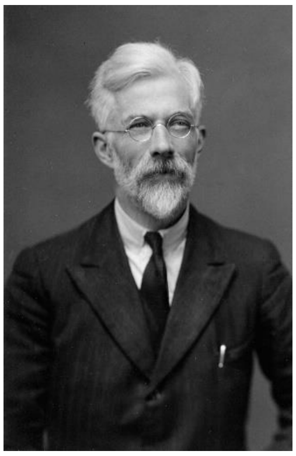


Although data presented in this figure are artificial, they resemble quite closely real RT data. This histogram shows how most observations are relatively small numbers (RT < 500 msec), what is typical of simple reaction times tasks. The command line behind these data is simply **rlnorm**(1500, meanlog = 5, sdlog = 1). With simple adjustment of values for meanlog and sdlog one can generate RTs very close to all possible cognitive experiments. Noteworthy is the fact that several observations are much larger than the mean, what is characteristic of the skewed distribution of RT data. How to analyze these data? Should one remove the observations from the right side? Which is a suitable cut-off point, since individuals may vary in the degree of logarithmic compression of their RT reactions?

Hint: Convince yourself that reaction times are typically log-normal distributed! You can use the Kolmogorov-Smirnoff test to test that the distribution is log normal.

```
series <- c(341, 291, 283, 155, 271, 270, 250, 272, 209, 236, 295, 214, 443, 632, 310, 334, 376, 305,
216, 339)
fit = fitdistr(series, "lognormal")$estimate  # require(MASS)
fit
meanlog
5.66611754205579
sdlog
0.290617205700481
ks.test(series, "plnorm", meanlog=fit[1], sdlog=fit[2], exact=TRUE)
One-sample Kolmogorov-Smirnov test
data:  series
D = 0.13421, p-value = 0.8181
alternative hypothesis: two-sided
```

Task 6: Analysis of variance



Analysis of variance is one of the most popular statistical tools to evaluate the results of experiments. It is based on the principle of variance partitioning, in which total variance is decomposed into its elements: variability across conditions also called systematic variance and variability within conditions, also called residuals. Ronald Aylmer Fisher (1890-1962) was a brilliant statistician admired by many of his most eminent colleagues such as Karl Pearson and William Sealy Gosset (a.k.a. the “Student”). He developed several techniques to compare different crops and fertilizers in completely randomized experiments. In this kind of experiment, each observation is fully independent from all others. Later on, analysis of variance was expanded to deal with dependencies across observations. It was noted, that sometimes it is reasonable to pair observations across conditions, to remove some of the variability usually assigned to the residuals. For instance, if I have three lawyers and two medical doctors in each group, the factor “profession” can be inserted in the design and possibly reduce the residuals. The advantage of block-designs is an increase in power to detect more subtle effects. The same logic can be applied to experiments with repeated measurements, which are quite typical of psychology. In this case, the same participant is tested repeatedly under many or even all experimental conditions. Here, it would be really unfair to treat each observation belonging to each experimental condition as fully independent, for they were obtained by testing several times the same person (i.e. they are pseudoreplications).

The statistics of analysis of variance (henceforth ANOVAs) is equivalent to size comparison of different portions of the variance. For instance, the variance across conditions observed in Factor A can be compared with the appropriate residuals. If the effect of A is much larger than the size of the residuals, one speaks of a significant effect. The F-statistics (named after Fisher by Fischer to honor himself) reflects the relative size of variance components: when F = 10 this means the variance of the factor is 10 times larger than the variance of the residuals, this is a substantial effect. To know how big an effect is, that means, calculating effect sizes, follows a similar logic under ANOVA: one compares the variance related to one specific main- or interaction effect with the variance of the other effects in the design.

One fundamental point about the F-statistics is the scaling of different variance components. This scaling is performed with the help of the number of degrees of freedom associated with both systematic effects and with the residuals.

Sum of squares of the effect / degrees of freedom of the effect /
Sum of squares of the residuals / degrees of freedom of the residuals

This is the reason why F-distributions are associated with two degrees of freedom: one number for the effect, another for the residuals. The way the number of degrees of freedom is calculated determines how large the different variance components will be when being compared. It is a priority to allocate as many degrees of freedom to the residuals as possible, because they reduce the size of residuals and let experimental effects become larger. Particularly when effects DO exist in nature, this is a very good thing! However, there is a huge caveat in this scaling process: It only works well, if the unities being scaled are measured with comparable precision, that means, when the variability of residuals when considered level by level has to be comparable. If it is not, we have a problem with heteroscedasticity.

Systematic sources of variance, the so-called effects can assume many forms. Some are called fixed-effects, because their levels represent all the possible states (or at least all the possible states that are somehow important) of a given variable. Literate vs. analphabets, male vs. female sex, sick vs healthy are examples of fixed effects. There are also other effects with a potentially infinite number of different levels. They are called random-effects because the levels entering a specific experiment are sampled from a much larger population of levels. The factor “individual” or a list of stimuli taken from a large list of possible candidates (e.g. lists of words) are typical random-effects.

When models contain only fixed-effects, they are called ANOVA models, when they contain both fixed and random effects, they are called mixed-effects models. Every repeated-measurements ANOVA is a simple case of mixed-effects model. Together with regression models, all these statistical models are subsumed under the General Linear Models, which also include SEM.

Fixed-effects work well for a small number of levels. In the case of random effects, the number of degrees of freedom can become very large and some interactions between random effects cannot be easily dealt with using ANOVA models (in the past one calculated *f1* and *f2* effects, see Nuerk et al. Cortex, 2002 for an example).

Differently from SPSS, R likes to have all the data from the dependent variable in one single column and not split over more columns.

```
nParticipants <- 20
Factor1      <- gl(2, 40)
Factor2      <- factor(rep(c(1,2, 1, 2), nParticipants))
Subj        <- factor(rep(gl(nParticipants, 2),2))
EffF1        <- c(rnorm(nParticipants, 50, 10), rnorm(nParticipants, 60, 10))
EffF2                  <- rep(0, 80)
EffF2[seq(1, 79, 2)]  <- rnorm(nParticipants, 60, 10)
EffF2[seq(2, 80, 2)]  <- rnorm(nParticipants, 40, 10)
EffSub       <- sort(rep(rnorm(40, 0, 20), 2))
dv <- EffF1 + EffF2 + EffSub
hist(dv)
model1 <- aov(dv ~ (Factor1*Factor2) + Error(Subj/( Factor1*Factor2)))
```

Task 7 Generating correlated predictors for a regression model

```
library(MASS)
N <- 200 # Number of random samples
#set.seed(123)
# Target parameters for univariate normal distributions
rho <- 0.6
mu1 <- 0; s1 <- 1
mu2 <- 0; s2 <- 1
# Parameters for bivariate normal distribution
mu <- c(mu1,mu2) # Mean
sigma <- matrix(c(s1^2, s1*s2*rho, s1*s2*rho, s2^2), 2) # Covariance matrix
bvn1 <- mvrnorm(N, mu = mu, Sigma = sigma) # from MASS package
colnames(bvn1) <- c("bvn1_X1","bvn1_X2")
mySD <- 0.44
beta = 0.3
y <- beta*bvn1[, 1]+ beta*bvn1[, 2] + rnorm(N, 0, mySD)
summary(lm(y ~ bvn1))
```

Your individual projects

Doris Größinger

Box-Cox transformation of reaction times data in repeated measures ANOVA

Reaction times (RT) are a common measure in psychological experiments. Usually multiple RTs are taken for each participant and each experimental condition. Analysis is often done with ANOVA by calculating the sample mean (Whelan, 2008). However, in practice RT data are not normally distributed, which can influence the representability of the mean. Another problem are possible outliers generated by processes that are not of importance (Ratcliff, 1993). Different methods have been studied to reduce the effects of outliers and the distribution for between-subjects ANOVA (Ratcliff, 1993) or a simple two-time-point repeated measures design (Bush, Hess, & Wolford, 1993). Some of these methods include trimming, data transformation or calculation of indicators other than the mean, like the median, with some working better than others (Bush et al., 1993; Ratcliff, 1993). Another method, which has not yet been analyzed in this context is the Box-Cox transformation (Box & Cox, 1964). With the Box-Cox transformation a λ-exponent is determined which is then used to transform the data closer to a normal distribution.
The aim of this study is to analyze the impact of the Box-Cox transformation on the power of different repeated measures ANOVA models, which are typical in psychological experiments. I will not analyze different trimming methods for outliers as these have been discussed thoroughly by Ratcliff (1993). This study also showed that the inverse transformation used without other trimming methods displays acceptable results, depending on the type of effect introduced to the data.
The definition of the repeated measures ANOVA designs is based on typical psychological designs, is similar to Krutchkoff (1988) and can be seen in table 1. Unequal numbers of participants for the

different time-points simulate dropout over the course of the measurements. Each of these models will be analyzed with 10, 20 or 50 observations per participant, with and without outliers and with and without effect. Outlier definition is the same as in Ratcliff (1993), where each value had a 0.9 chance to be drawn from the standard distribution and a 0.1 chance to be drawn from a different distribution consisting of the standard distribution added by a random value between 0ms and 2000ms. Therefor each condition could have a different number of outliers. A main effect is included in one group for each design by increasing the distribution mean of 400ms once by 30 and 40 ms.

Table 1 Designs used in the study. n(k) means n groups with k observations each.

Design nr.	n(k)				
1	2(10)	6	3(10)	13	5(10)
2	2(20)	7	3(20)	14	5(20)
3	2(50)	8	3(50)	15	5(50)
4	1(20), 1(10)	9	2(50), 1(40)	16	3(50), 2(40)
5	1(50), 1(40)	10	2(20), 1(10)	17	3(20), 2(10)
		11	1(50), 1(40), 1(30)	18	2(50), 2(40), 1(30)
		12	1(30), 1(20), 1(10)	19	2(30), 2(20), 1(10)
				20	1(50), 1(40), 1(30), 1(20), 1(10)

Ref:

Box, G. E., & Cox, D. R. (1964). An analysis of transformations. Journal of the Royal Statistical Society: Series B (Methodological), 26(2), 211-243.

Bush, L. K., Hess, U., & Wolford, G. (1993). Transformations for within-subject designs: a Monte Carlo investigation. Psychological bulletin, 113(3), 566.

Krutchkoff, R. G. (1988). One—way fixed effects analysis of variance when the error variances may be unequal. Journal of Statistical Computation and Simulation, 30(4), 259-271.

Ratcliff, R. (1993). Methods for dealing with reaction time outliers. Psychological bulletin, 114(3), 510.

Whelan, R. (2008). Effective analysis of reaction time data. The Psychological Record, 58(3), 475-482.

Saša Zorjan
Median split
I was thinking of simulating continuous variables that would be normally distributed and correlated with each other (small to medium correlation) - I would then dichotomize one of the variables and perform a regression analysis to see what happens when we enter the variable as a continuous vs. dichotomous variable. I can look at different options –

1) splitting it "lower" and "higher" 50%, 2) removing the middle 50% to end up with "extreme" groups (lower and higher 25%) etc.
2) I can also look at different correlations (to see what happens if the variables are more or less correlated).

"In order to examine this topic, the present project will conduct Monte Carlo simulations where we will vary 1) sample sizes (N = 50, 100, 200), 2) correlations between the predictor X1 and the outcome (0.0, 0.3, 0.5 and 0.7) and 3) correlations between the two predictors (X1, X2) (0.0, 0.3, 0.5 and 0.7). Additionally, we will use 3 different methods of categorizing continuous variables (a) median split, b) 3 groups (33.3%) and c) extreme groups (lowest and highest 25% of scores). A sample will be drawn from a multivariate normal distribution. A multiple regression model will be used to analyze the beta estimates for the two continuous predictors and their interaction (β1, β2, β3). The equation for the model that we will run is as follows: $Y = \beta 0 + \beta 1\ X1 + \beta 2\ X2 + \beta 3\ X1\ X2$.
In study 1, one of the predictors (X1) will be categorized and the beta estimates will again be obtained with a multiple regression model. The estimates will then be compared and plotted.
In study 2, both predictors (X1, X2) will be categorized and the beta estimates will again be obtained and compared."

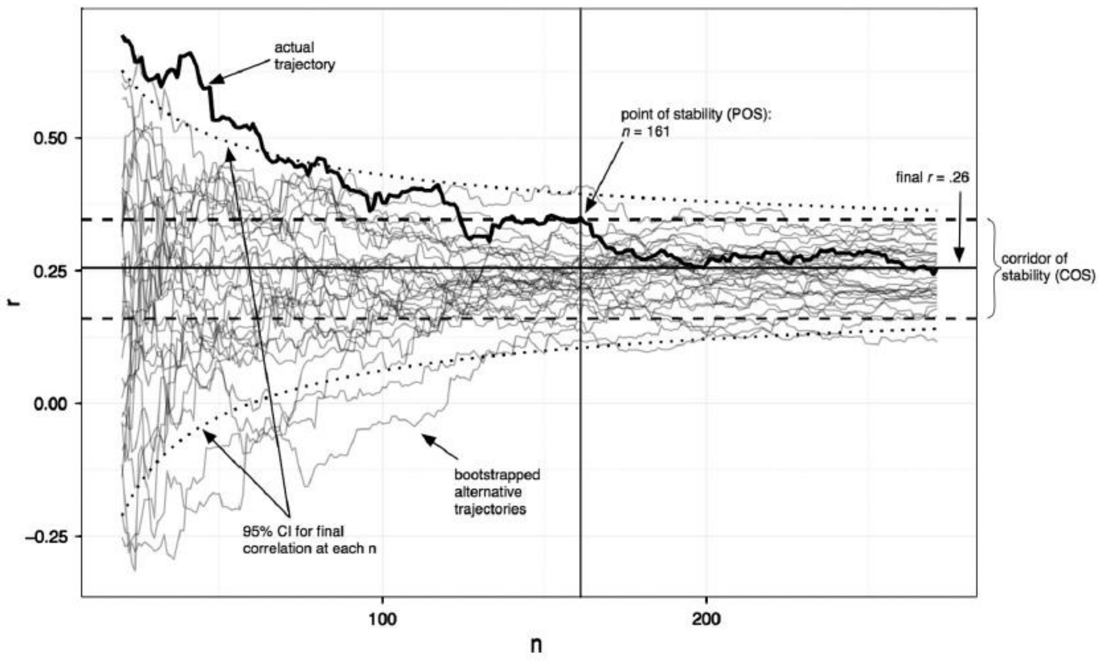
Viktoria Jobstl

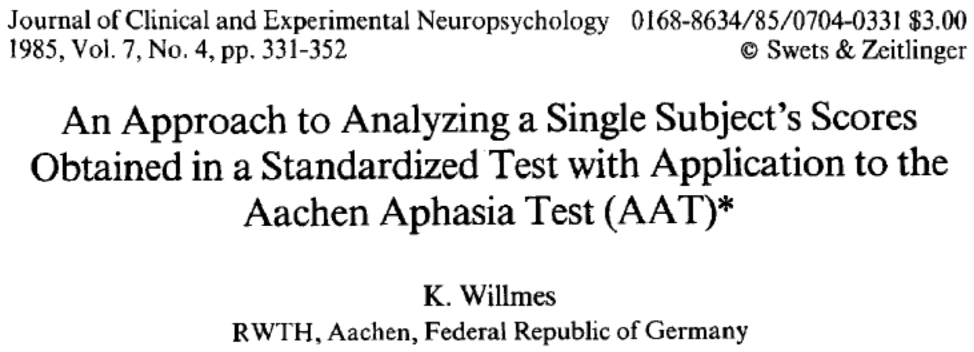
In spring 2019 around 600 kindergartners completed the screening and then again the same group of children + some additional in autumn 2019.
In spring 2020 a different group of kindergartners completed the same screening (N > 2000).
(The screening changed slightly between the different time points but the variables stayed the same.) In the screening phonological awareness, number and letter knowledge, rapid automatized naming ... were assessed.

I thought I could analyse the relationship of certain variables and then use the found parameters to simulate certain results with different sample sizes etc. and check whether the predicted data represents the collected.

Gabriela Hofer
Stability of correlations
If possible, I would like to replicate the attached paper for my final report in our course. It is a paper that focuses on the sample size that is necessary for correlations to stabilize. Since I am doing a lot of correlational work (mostly looking at correlations between self-estimates and more objective measures), this paper has been a very relevant resource for me and I am sure that replicating it would strengthen my understanding of the subject.

Corridor of stability is given by *w* and can be obtained for each sample size by means of the effect size *q*, which assumes different values and is added/subtracted from fisher's *z*-transform. With increasing sample sizes, one eventually reaches a point of stability POS.





Journal of Clinical and Experimental Neuropsychology 0168-8634/85/0704-0331 $3.00
1985, Vol. 7, No. 4, pp. 331-352 © Swets & Zeitlinger

An Approach to Analyzing a Single Subject's Scores Obtained in a Standardized Test with Application to the Aachen Aphasia Test (AAT)*

K. Willmes
RWTH, Aachen, Federal Republic of Germany

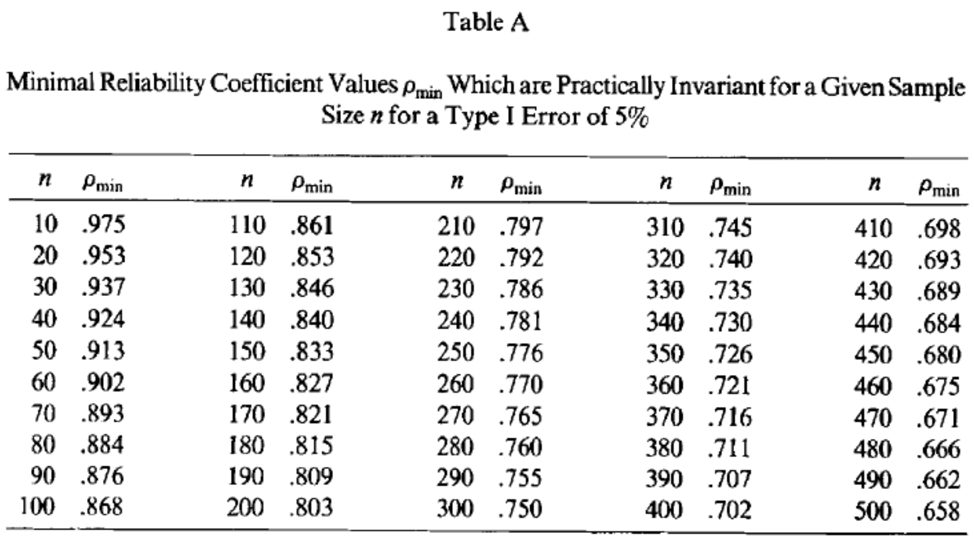


Table A

Minimal Reliability Coefficient Values ρ_{min} Which are Practically Invariant for a Given Sample Size n for a Type I Error of 5%

n	ρ_{min}	n	ρ_{min}	n	ρ_{min}	n	ρ_{min}	n	ρ_{min}
10	.975	110	.861	210	.797	310	.745	410	.698
20	.953	120	.853	220	.792	320	.740	420	.693
30	.937	130	.846	230	.786	330	.735	430	.689
40	.924	140	.840	240	.781	340	.730	440	.684
50	.913	150	.833	250	.776	350	.726	450	.680
60	.902	160	.827	260	.770	360	.721	460	.675
70	.893	170	.821	270	.765	370	.716	470	.671
80	.884	180	.815	280	.760	380	.711	480	.666
90	.876	190	.809	290	.755	390	.707	490	.662
100	.868	200	.803	300	.750	400	.702	500	.658

Viktoria Frühwirth

Variance inflation coefficient

How do beta coefficients in multiple linear regression change with increasing variance inflation factor?

I had this problem with multicollinearity in my own research and found multiple articles suggesting VIFs varying between 3 and 10 as okay.

Homework discussion

Take commenting seriously

Do not neglect indentation

```
        Install.packages("myFavouritePackage")

        #Require(myFavouritePackage)

        myFavouritePackage::plot(myObject)
```

Vertical format: As many columns as independent variables plus one column for each dependent variable.

Unlist()

Monte Carlo and Bootstrapping imply data generation/sampling and data analysis

The format of data generation should follow the optimal format for analysis

Repeated parts of script -> are they necessary? Can it be simplified?

Trellis

Homework 3
This time we will run a simulation of a regression model with two informative predictors. We set the (unstandardized) regression weights at a value of 0.2 or 0.4 and generate predictors with two levels of correlation, low = 0.2 and high = 0.6. There are many methods to generate correlated predictors. Five different methods are described in the following website:

https://blog.revolutionanalytics.com/2016/08/simulating-form-the-bivariate-normal-distribution-in-r-1.html

I encourage you to employ the second method to generate data.

Moreover, we also have to think about the weight of residuals on the model. They determine the proportion of explained variance since Var(y) = COV(y, x) + Var(residuals). Here, two levels of residuals should be modelled = 40% and 80%. Sample sizes will also vary from very small to large n = c(20, 50, 100, 200, 500). Each condition should be iterated 1000 times.

The dependent variable for further analysis is the standard error for the first predictor. In an ANOVA model, the main- and interaction effects of the regression weight, predictor intercorrelations, proportion of residuals and sample sizes should be evaluated. Since all effects will probably be significant, use the partial eta^2 to evaluate the role of each main- and interation effect. The web-page R-seek can be very helpful to suggest ways to obtain the partial eta^2.