SCR lecture 6: consolidating the basics

Exercises

1 A Plenary Exercise on Share of Women Researchers by sectors of performance.

This is a (slightly modified) task that once was part of an assignment for grade (SCR 2016). The goal here is to download and clean Eurostat Official Statistics data on the EU women' employment such that it can be used for further data analysis and visualization. Thus, the original data is not very tidy, at the end of this task it should have the following structure:

```
load("0_data/tidy_dat.RData")
str(tidy_dat)
```

```
## 'data.frame':    60 obs. of  26 variables:
##  $ sector: Factor w/ 2 levels "BES","GOV": 1 1 1 1 1 1 1 1 1 1 ...
##  $ geo   : Factor w/ 41 levels "AT","BA","BE",..: 1 2 3 4 5 6 7 8 9 10 ...
##  $ X2006 : num  13.6 NA 21.5 36.7 NA 22 12.9 NA NA NA ...
##  $ X2007 : num  14.2 NA 21.7 38.5 NA 22.4 13 12.2 24.6 18.3 ...
##  $ X2008 : num  NA NA 23.8 41.4 18.7 25.2 13.6 NA NA NA ...
##  $ X2009 : num  16.3 NA 24 43.4 NA 27.3 13.8 12.7 23.9 18.6 ...
##  $ X2010 : num  NA NA 25.5 43.7 NA 26.3 13.6 NA 24.8 NA ...
##  $ X2011 : num  16.3 NA 26.1 45 NA 27 13.8 14.2 27 19.1 ...
##  $ X2012 : num  NA 33.3 NA 42.8 23.3 29.6 13.7 NA 27.9 NA ...
##  $ X2013 : num  17.6 58.6 24.9 42.8 NA 31.3 13.9 14.1 26.6 19.3 ...
##  $ X2014 : num  NA 54.1 NA 39.6 NA 32.9 13.9 NA NA NA ...
##  $ X2015 : num  17.1 NA 27.1 37.5 23.3 32.4 12.8 14.7 24.5 19.8 ...
##  $ X2016 : num  NA NA NA 39.3 NA 29 12.6 NA NA NA ...
##  $ X2017 : num  NA NA NA NA NA NA 12.5 NA 28 NA ...
##  $ C2006 : Factor w/ 3 levels "b","d","e": NA NA NA NA NA NA NA NA NA NA ...
##  $ C2007 : Factor w/ 3 levels "b","d","e": NA NA NA NA NA NA NA NA 1 NA ...
##  $ C2008 : Factor w/ 3 levels "b","d","e": NA NA NA NA NA NA NA NA NA NA ...
##  $ C2009 : Factor w/ 2 levels "d","e": NA NA NA NA NA NA NA NA NA 2 ...
##  $ C2010 : Factor w/ 3 levels "be","d","e": NA NA NA NA NA NA NA NA 3 NA ...
##  $ C2011 : Factor w/ 3 levels "b","d","e": NA NA NA NA NA NA NA NA NA NA ...
##  $ C2012 : Factor w/ 3 levels "b","d","e": NA NA NA NA NA NA NA NA NA NA ...
##  $ C2013 : Factor w/ 3 levels "b","d","e": NA NA NA NA NA NA NA NA NA NA ...
##  $ C2014 : Factor w/ 3 levels "b","d","e": NA NA NA NA NA NA NA NA NA NA ...
##  $ C2015 : Factor w/ 2 levels "d","e": NA NA NA NA NA NA NA NA NA 2 ...
##  $ C2016 : Factor w/ 3 levels "b","d","e": NA NA NA NA NA NA NA NA NA NA ...
##  $ C2017 : Factor w/ 2 levels "d","p": NA NA NA NA NA NA NA NA 2 NA ...
```

For a descirption and its format see

> http://ec.europa.eu/eurostat/tgm/refreshTableAction.do?tab=table&plugin=1&pcode=tsc00005&language=en
>
> http://ec.europa.eu/eurostat/estat-navtree-portlet-prod/BulkDownloadListing?file=data/tsc00005.tsv.gz&unzip=true

We have created a tinyurl address that links to Eurostat data: https://tinyurl.com/w6-data-191018

You will need at least one function, i.e. `strsplit()`. We have decomposed the whole task into sub-tasks. In case you come up with a different way to tackle the whole problem, you are welcome to do so! Then, just provide a cleary code written answer such that we can easily follow each of the steps you have chosen to take.

a

Download the data from the website into `R`.

Answer:

```
url <- "https://tinyurl.com/w6-data-191018"
mydat <- read.delim(url)
```

b

Focus on the first column of the data which is called `unit.sectperf.geo.time`. Each one looks something like `PC_HC,BES,AT` where each abbreviation is separated by a column. We want the sector which in this case is `BES` and the geo location which in this case is `AT`.

Use `strsplit` to extract these values for each element in the `unit.sectperf.geo.time` column of the data, and create two new variables called `sector` and `geo`. (hint, the data is currently a `factor` and `regular expressions` work on `characters`)

Answer:

```
# answer
first_col <- mydat[, 1]
first_col <- strsplit(as.character(first_col), split = ",", fixed = TRUE)
three_cols <- do.call("rbind", first_col)
sector <- three_cols[,2]
geo <- three_cols[,3]
```

c

Program your own function that you can apply to one of the remaining columns of the data. You may want to use the hint on how to create your function:

Hint:

```
GiveNameThatTellsWhatThisFunctionNeedsToDo <- function(column){
  # function takes a single column as argument,
  # and returns a list with two elements.
  # The percentages of the column are stored in
  # the first element and the annotated letters
  # are stored in the second element
}
```

Please bear in mind that this can be a frustrating task. As you will most probably experience in your later career: cleaning data is very frustrating. . . .

Answer:

```
CleanYearPercentageColumn <- function(column){
  # column <- mydat[,10]
  column <- as.character(column)

  col_splitted <- strsplit(column, split = " ") # split on whitspace, returns a list

  prct_col <- sapply(col_splitted, "[", 1) # percentages, but still character
  prct_col <- suppressWarnings(as.numeric(prct_col)) # percentage column
  annt_col <- sapply(col_splitted, "[", 2) # annotation column with letter(s)

  return(list(prct_col, annt_col))
}
```

d

Apply your function of c in an explicit or implicit loop at each column. Combine the results into two data frames, one that holds the `numbers`, and one that holds the `letters`.

Answer:

```
my_list <- lapply(
  X = mydat[2:ncol(mydat)],
  FUN = CleanYearPercentageColumn
) # apply function and store results in a list
percentages <- do.call("cbind", lapply(my_list, '[[' , 1))
percentages <- data.frame(percentages)
annotation <- do.call("cbind", lapply(my_list, '[[' , 2))
annotation <- data.frame(annotation)
```

e

Finally, combine the `sector`, `geo`, `percentages` and `annotation` into one data frame and make sure everything has the correct class, column names and that the variables match up with the output from above. Show that your cleaned data set is the same as `tidy_dat`.

Answer:

```
names(annotation) <- gsub("X", "C", names(annotation))
sector <- factor(sector)
geo <- factor(geo)
my_clean_dat <- data.frame(sector, geo, percentages, annotation)
```

```
load("0_data/tidy_dat.RData")
all.equal(my_clean_dat, tidy_dat)
```

```
## [1] TRUE
```

Self-Study: Exercises for Consolidation of your R Skills

2 Hashtable

Primer: this exercise will require that you know how to use:

- Dataframes
- Indexing
- Write a function
- `for` loops
- `if-else` statements
- How to use help files

In this exercise we will take a look at a (simple) hashtable! A hashtable is a lookup table (for example a phone book) where we have a key, which we need to match to some other datum, e.g. a name, for which the phonebook contains a corresponding phonenumber.

A simple way to create a look up table, is to created a sorted list of names, and an equally sorted list of phone numbers so that the position of a name, corresponds to the position of the phonenumber that belongs to that name. Finding the right phonenumber could then be done by looping through the sorted list of names, and stop once we find a match for the name whose number we are looking for. We will do this first.

For simplicity we'll look at a phone book that is already sorted, and which has no duplicate entries!

a

Read in `phone_book.txt`.

Answer:

```
phone_book <- read.table("0_data/phone_book.txt", header = T, stringsAsFactors = F)
```

b

Write a function that takes a phonebook and a name argument, and that goes through the sorted list of names in the phone book and returns the phone number of the name provided in the argument.

Hint: Use `for` and `break`

Answer:

```
LookupNumber <- function(name, phone_book){

  for (i in 1:nrow(phone_book)){

    if (phone_book[i, 1]==name){
      break
    }

  }

  return(phone_book[i, 2])
}
```

c

Apply your function to the following names:

- Brilan

- Dicey
- Imre
- Kaytelyn
- Makia
- Patrica
- Shivaun
- Vihana
- Wilmetta

And record for each name the time it takes using the function `system.time`. If it takes a long time, please just do this exercise for the first few names.

Answer:

```
LookupNumber("Brilan", phone_book)
```

```
## [1] 31693576527
```

```
system.time(LookupNumber("Brilan", phone_book))
```

```
##    user  system elapsed
##   0.121   0.001   0.122
```

```
system.time(LookupNumber("Wilmetta", phone_book))
```

```
##    user  system elapsed
##   0.934   0.005   0.940
```

d

As you may have noticed, looking up a name in this way takes quite some time. It would be nice if we knew, directly from the name, the position of that name, in the phone book. This is where a hashtable comes in.

This will require a hashfunction: one that converts the name into an index, and a hasthable that has the names in that position.

We'll start with making a hashfunction. The hashfunction should take as input a name, and give as output a value of 6 digits (possibly including leading zero's). These 6 digits should be a product of the particular letters in the name. We will apply a very important skill in programming. We'll divide the task up into smaller pieces. For example: we will start by creating all the components we need for the hashfunction, by just focusing on applying these components on the first name. There are many ways in which we could make this hashfunction, so we will just program one example application.

We will break up this big task into smaller pieces.

i

Split the first name, using `strsplit()` into a vector of letters.

Answer:

```r
names <- as.character(phone_book$names)
split_name <- strsplit(names[1], split="")[[1]]
```

ii

Create an equally long vector, with each entry equal to the smallest digit of the position of the letter in the alphabet. E.g. 'a' would be '1', 'm' would be '3' (you can use the modulo operator function `%%` for this (see `?'%%'`)). To see the position of the letters of the name you can use the operator function `%in%` and the function `which()`. *Hint: R has built-in vectors containing `letters` (and capital `LETTERS`).*

Answer:

```r
B <- length(split_name)
name_numbers <- numeric(B)
name_numbers[1] <- which(LETTERS %in% split_name[1]) %% 10
for (i in 2:B) {
  name_numbers[i] <- which(letters %in% split_name[i]) %% 10
}
```

iii

Compress the first 6 numbers of this vector of numbers into a single number using `paste()`. Use only the first 6 numbers (or less if no more are available). Add 1. You may have an idea already why we would need to add 1.

Answer:

```r
minl <- 1:min(length(name_numbers), 6) # get minimum needed length
index <- as.numeric(paste(
  name_numbers[minl], # to avoid NA's
  collapse=""
)) + 1
```

Because of the modulo operator it is possible that the hashvalue of a name is 0. That is not a valid index.

iv

Make a function called `HashFunction` that performs the actions programmed in **1 e**, using as argument a single name, pasting to a single number and returns that number.

Answer:

```r
HashFunction <- function(name){
  split_name <- strsplit(name, split="")[[1]]
  B <- length(split_name)
  name_numbers <- numeric(B)
  name_numbers[1] <- which(LETTERS%in%split_name[1])%%10
  for (i in 2:B){
    name_numbers[i] <- which(letters%in%split_name[i])%%10
  }
  minl <- 1:min(length(name_numbers), 6) # get minimum length
  index <- paste(name_numbers[minl], collapse="") # to avoid NA's
  index <- as.numeric(index) + 1
  return(index)
}
```

e

Make a `list` object in `R` of length 1000000. Loop through `phone.book`, using `HashFunction` to create an index number for each name.

Answer:

```
hash_table <- vector(mode ="list", length = 1000000)
indices <- sapply(phone_book$names, HashFunction)
```

不同于lapply, sapply会留下名字作为Index

f

Hopefully, all names have a unique index. Probably, this is not true. Check this

Answer:

```
length(indices)
```

```
## [1] 95025
```

```
length(unique(indices))
```

```
## [1] 35719
```

It seems there are only 35719 unique indices! For example these names have the same hashvalue.

```
indices[indices == 981836]
```

```
##    Shahmeer    Sharmeen    Sharmeka     Sharmel    Sharmell     Sharmen
##      981836      981836      981836      981836      981836      981836
##    Sharmeta     Sharmon Sharmonique     Sharmyn
##      981836      981836      981836      981836
```

g

This means that our `HashFunction` cannot map each name, to a unique phone number. In general, something like this will happen with every hashtable, no matter how many possible indices we allow for.

The way to solve this, is to make sure that we are able to solve these local conflicts. We can do this, by making a very small phone book, at the indices where conflicts occur: once at this position, we go through all the names that were mapped to this position, and check which corresponds to the name we were looking for.

First things first. Create the hashtable by looping through all the names, and the indices, and by putting an entry of the phone number at the position of the index. If a second name is entered at the index, append it to the name and phone number already there.

Hint: This time you are allowed to create a growing container in a `for` loop!

Answer:

```r
N <- nrow(phone_book)

for (i in 1:N) {
  # i <- 1
  j <- indices[i]

  if (is.null(hash_table[[j]])) {
    hash_table[[j]] <- phone_book[i, ]
  } else {
    hash_table[[j]] <- rbind(hash_table[[j]], phone_book[i, ])
  }
}
```

h

Write a function that finds the phone number of the name provided by using as arguments only a name, and the hashtable.

Answer:

```r
LookupNumberHash <- function(name, table){

  index <- HashFunction(name)

  entry <- table[[index]]

  if (nrow(entry) > 1) {
    for (j in 1:nrow(entry)) {
      if (entry[j, 1] == name) {
        number <- entry[j, 2]
        break
      }
    }
  } else {
    number <- entry[1, 2]
  }

  return(number)
}
```

i

Use the hashtable you've made, and the lookup function you've made using this particular hashfunction to look up the names mentioned in *1c*. Measure the amount of time it takes to lookup these names.

Hint: you can use the positions `c(12586, 23775, 35043, 46271, 57512, 68749, 79861, 90045, 91170)` *to check your answers!*

Answer:

```r
lookup_names <- c(
  "Brilan",
  "Dicey",
```

```
  "Imre",
  "Kaytelyn",
  "Makia",
  "Patrica",
  "Shivaun",
  "Vihana",
  "Wilmetta"
)
sapply(lookup_names, function(name) LookupNumberHash(name, hash_table))
```

```
##      Brilan       Dicey        Imre    Kaytelyn       Makia     Patrica
## 31693576527 31683776520 31658329939 31629220976 31642713309 31624008191
##     Shivaun      Vihana    Wilmetta
## 31675509022 31611018225 31693564692
```

```
system.time(sapply(lookup_names, function(name) LookupNumberHash(name, hash_table)))
```

```
##    user  system elapsed
##   0.002   0.000   0.002
```

```
phone_book[c(12345, 23456, 34567, 45678, 56789, 67890, 78901, 89012, 90123), ]
```

```
##          names     numbers
## 12345   Breton 31679341276
## 23456   Dewand 31672366111
## 34567   Ianthe 31650224287
## 45678 Kateryna 31607878669
## 56789 Madielyn 31683902112
## 67890     Oram 31635190307
## 78901  Shawnte 31693531193
## 89012     Uyen 31640526802
## 90123   Vincil 31684511321
```

3 Cleaning a Genetics Data Set

You are hired as the only biostatistician in a genetics research institute. You just started working there, and a colleague comes to you with the file `0_data/RawCompareSNPTable1.scsv`. The colleague gives you the following explanation of the file:

"On the first line we see persons labeled as C180_v1, C229_v1 ... C433_v1, or not labeled at all. Every person has it's own column. A new column starts after ;. Hence, ; is a delimiter indicating a new field (= colum in our case).

The second line gives in the first four columns nothing, and then suddenly "chr1:52840454". This indicates that for person "C249_v1" we have the genetic information about his/her base pair 52840454 on chromosome 1."

Your colleague actually already started working (manually) on the file. What your colleague wanted to create was a sort of incidence matrix from you can see which persons are genotyped for a specific base-pair on a chromosome. Could you help your colleague?

In this case, it is not necessary to understand the data you are working with. However, in case you are interested to understand a bit more of the data. For an explanation of the chromosome see: https://ghr.

nlm.nih.gov/primer/basics/chromosome. An explanation of DNA and a basepair check out https://ghr.nlm.nih.gov/primer/basics/dna.

The Task:

Create a function that has as input the unstructured semi-colon separated values (`.scsv`) file and outputs the file in a structured way. In the structured file the rownames are the chromosomes that are ordered ascendingly from 1:19, on to the last chromosome X. Also, the basepair numbers are ordered ascendingly within the chromosome too. Hence, the row with "chr1:52840454" comes before "chr1:52859391" since 52840454 is smaller then 52859391. For every column (sample / person) indicate TRUE or FALSE whether their genotyped information is present (TRUE) or not (FALSE). Finally, create a new column called `sum`, which counts the number of TRUE in each row i.e. the number of persons for whom that specific base pair is known.

The final file should eventually look like the "FinalCompareSNPTable1.scsv" file (see the data folder).

Note: "In this latter file a 1 is given for TRUE, and NA is representing FALSE. You can choose yourself whether to stick with TRUE/FALSE values, or 1/NA values. Either of these solutions will obtain full points.

Hint: the function gsub() is used in de the model answers.

Answer:

The function that sorts the chromosomes could have been done in a vectorized way too, that is, without for loops, or apply functions:

```
SortGenes <- function(strg) {

  chrs <- gsub(":.*", "", strg)
  chrs <- gsub("chr", "", chrs)
  chrs <- as.numeric(gsub("X", 99, chrs))

  bp <- as.numeric(gsub(".*:", "", strg))

  return(strg[order(chrs, bps)])
}
```

```
CleanGenFile <- function(file_addr, outfile = NULL) {
  # A function that cleans the file of the genetics department

  # Args:
    # file_addr: address of the file that needs to be cleaned (in char)
    # outfile: address where to write the clean file (in char). When NULL, then it
    #   does NOT write the cleaned data to a file.

  d <- read.delim(file_addr, sep = ";", stringsAsFactors = F)
  d <- d[, grep("v1", colnames(d))] # remove unnamed columns (read.delim calls them X.<n>)
                                    Keep only columns with v1
  v <- unlist(d)
  v <- unique(v)
  v <- v[v != ""]

  SortGenes = function(strg) {
    spl  <- strsplit(strg, ":")

    chrs <- sapply(spl, function(s) s[1])
    chrs <- sub("chr", "", chrs)
```

```
    chrs <- sub("X", "90", chrs)
    chrs <- as.numeric(chrs)

    bp <- sapply(spl, function(s) s[2])
    bp <- as.numeric(bp)

    strg[order(chrs, bp)]
  }

  v <- SortGenes(v)

  clean <- lapply(d, function(clm) {
    c(NA, 1) [v %in% clm + 1]
  })

  clean <- data.frame(clean, row.names = v)
  clean$sum <- rowSums(clean, na.rm = TRUE)

  if( !is.null(outfile)) write.table(clean, outfile, sep = "\t")
  return(clean)
}
```

```
cleaned <- CleanGenFile(
  file_addr = "0_data/RawCompareSNPTable1.scsv",
  outfile = "0_data/CleanedSNPTable1.scsv"
)
final_dat <- read.delim(
  "0_data/FinalCompareSNPTable1.scsv",
  sep = "\t",
  stringsAsFactors = F
)
all.equal(cleaned, final_dat)
```

```
## [1] TRUE
```

4 Looking for primes

See how far you can get with this exercise in 1 hour.

In this exercise we'll use most programming structures we've covered to find some prime numbers. You probabably know: a prime number (or a prime) is a natural number greater than 1 that has no positive divisors other than 1 and itself.

We'll look at two different implementations and find that something we've discussed before (and we told you to avoid) is sometimes unavoidable: 'growing objects'.

a

Let's start off basic. Create an object called `my_primes` with the value `2`. Check if the value `3` is divisibly by `my_primes`, and if not, append `my_primes` with the entry `3`. Now check if the value `4` is divisble by any of the elements in `my_primes`, if not add it to `my_primes` otherwise do not, and continue to check `5`.

Answer:

```
my_primes <- 2
if (all(3 %% my_primes != 0)){
  my_primes <- c(my_primes, 3)
}
if (all(4 %% my_primes != 0)){
  my_primes <- c(my_primes, 4)
}
if (all(5 %% my_primes != 0)){
  my_primes <- c(my_primes, 5)
}
```

b

Suppose our goal is to find the smallest 100 prime numbers. You don't want to be doing this by hand right? We don't yet know however at what natural number we will have found 100 primes. We thus have to use something like `repeat` or `while` to repeatedly keep performing the operation described above.

Use `repeat` to repeatedly check if some value `i` is a prime or not. Start with `my_primes` equal to `2`, and with `i` equal to `3`. Perform the check you've performed in **a**. Remember that most operators in `R`, can be be done in a vectorized manner. E.g. try `4 %% c(2, 3)` and see if you can use this to check if `4` is divisble by any of the entries in the vector.

We already know that we want to find 100 primes. This means we can tell `R` to make `my_primes` a vector of length 100, fill the first entry with `2`, and as long as we keep track of how many primes we've found so far, we can fill the entire vector nicely. This way we can avoid growing the `my_primes` object.

Finally, write a conditional inside your `repeat` loop to `break` out of the loop, if you've written a prime in the 100th element of the vector `my_primes`.

Answer:

```
n_primes <- 1
my_primes <- numeric(100)
my_primes[1] <- 2
i <- 3
repeat{

  if (!any(i %% my_primes == 0, na.rm=TRUE)){
    n_primes <- n_primes + 1
    my_primes[n_primes] <- i
  }

  i <- i + 1

  if (n_primes == 100){
    break
  }

}
```

5 Visualization

We can use a Q-Q-plot to compare (the Quantiles of) a theoretical distribution, with the (Quantiles of the) distribution of an observed variable. We are going to use it to compare some observed variables with a normal distribution.

In the first part of this exercise we will compare a variable, sampled from a t distribution with degrees of freedom equal to 10, to a normal distribution. This first part was also presented as an exercise in the course material of last week.

See how far you can get with this whole exercise 30 minutes hour. You may want to look back to the lecture slides and exercises of some of the previous lectures.

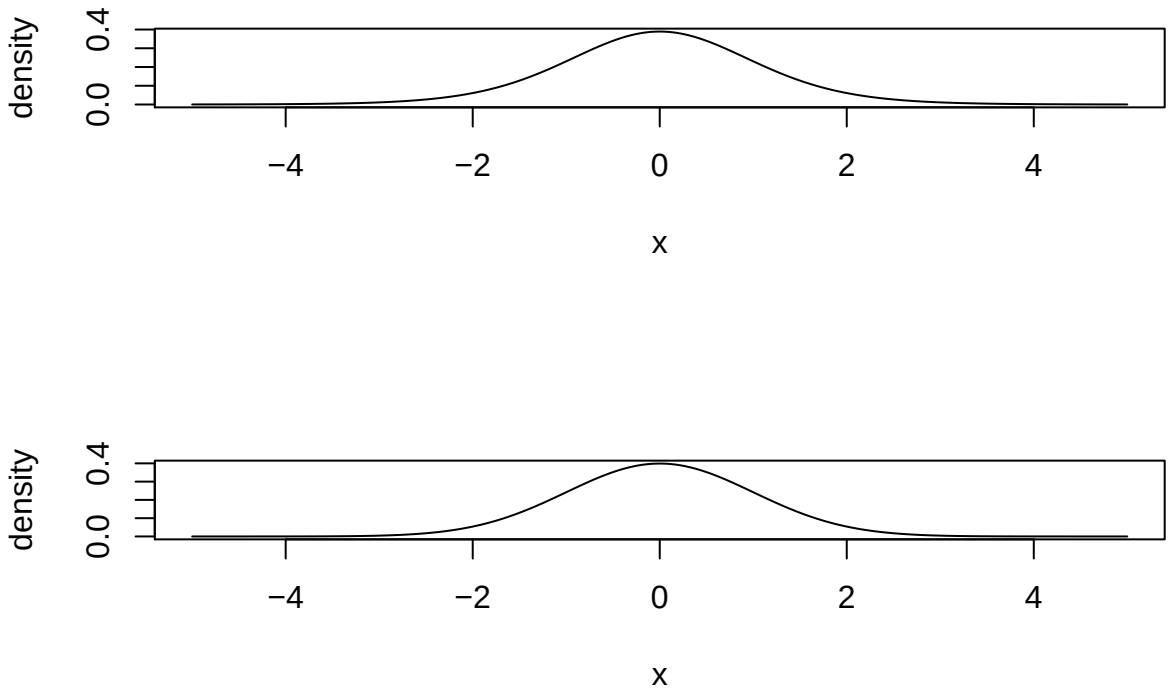
5a

For investigation purposes, first make a plot of the density function of a t-distribution with degrees of freedom equal to 10, evaluated over the interval $[-5, 5]$. Also make a plot of the density function of the standard normal distribution, evaluated over the interval $[-5, 5]$. Set the plotting parameters such that both plots are given in the same plotting window, one on top of the other. Give the axes of both plots nice labels. *Hint: you may need to use your searchengine muscles here.*

Answer:

```
x <- seq(-5, 5, by=0.01)

par(mfrow=c(2, 1))
plot(x=x, y=dt(x=x, df=10), type='l', ylab="density", xlab="x")
plot(x=x, dnorm(x=x), type='l', ylab="density", xlab="x")
```




```
par(mfrow=c(1, 1))
```

5b

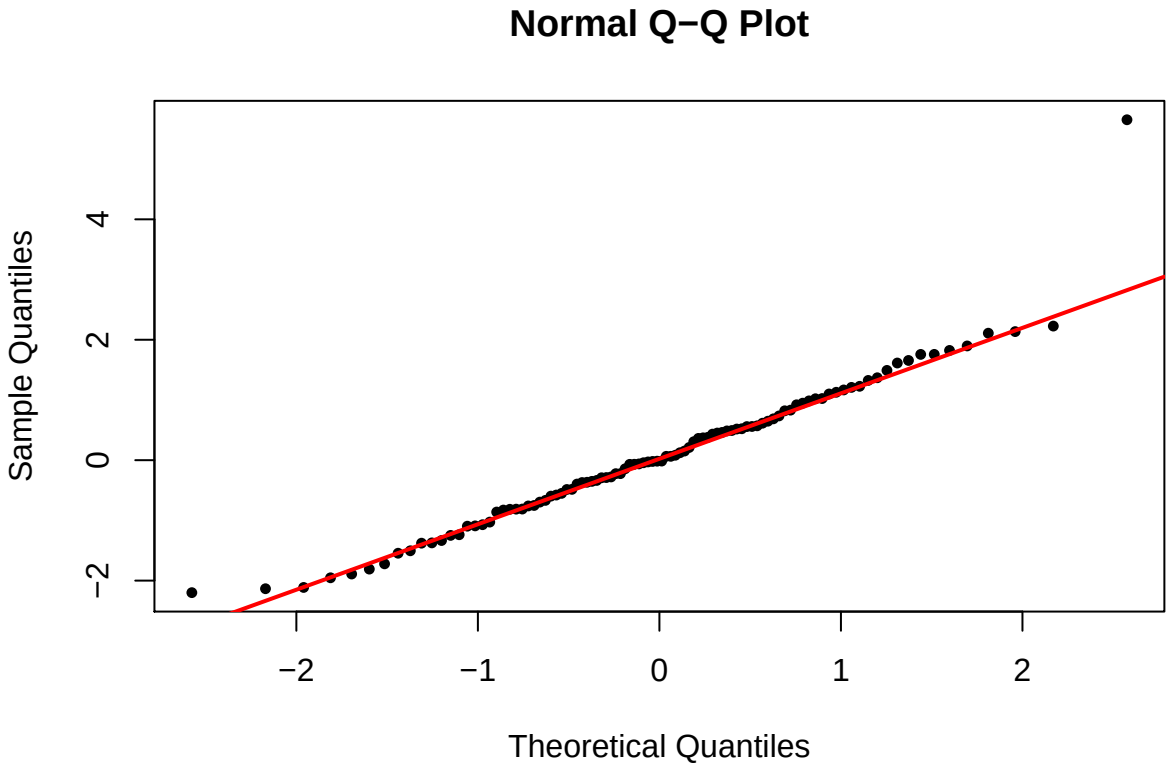
Although this comparison gives us some idea if we look *really* carefully, it's not a very great way to compare two distributions directly. For this, we will use a Q-Q-plot. Look at the helpfiles of `qqnorm`, especially at the definition of `qqline`. Use `qqnorm` to compare the quantiles of a variable that is randomly sampled (100

samples) from a t distribution with degrees of freedom equal to 10, to a normal distribution. Before you sample, set a seed. Make sure that you also plot a Q-Q-line. Make the line red and a little wider. Make the plot points solid, and a little smaller. *Hint: you may need to use your searchengine muscles here.*

Answer:

```
set.seed(6112016)

N <- 100
x <- rt(N, df=10)
qqnorm(x, pch=16, cex=0.75)
qqline(x, lwd=2, col='red')
```



5c

Interpret the Q-Q-plot.

Answer: Clearly the t distribution has wider tails.

6 Outer

`outer` is a very convenient function when it comes to creating data fit for `contourplot`.

a

Use `outer` to evaluate the function $f(x, y) = sin(x) * sin(y)$ for all pairwise combinations of a sequence of a 100 equally spaced values of `x` and `y`, both in the interval $[-\pi, \pi]$

Answer:

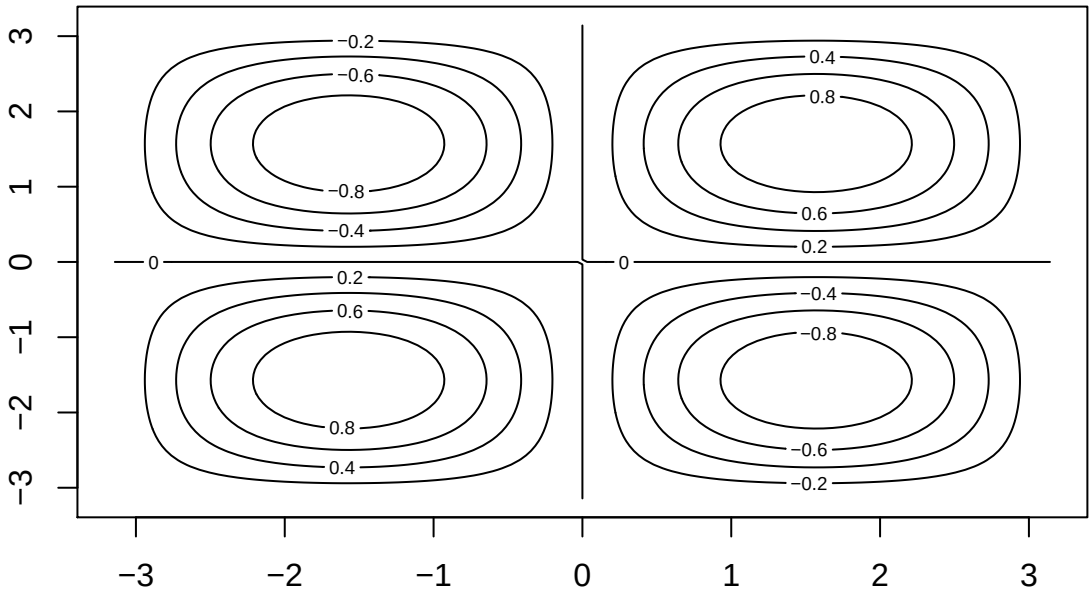
```
x <- y <- seq(-pi, pi, length.out=100)
f_x_y <- outer(x, y, function(i, j){sin(i)*sin(j)})
```

b

Use the result from **a** to make a contourplot using `contour`. What is the shape of the surface? Is each quadrant similar?

Answer:

```
contour(x, y, f_x_y)
```

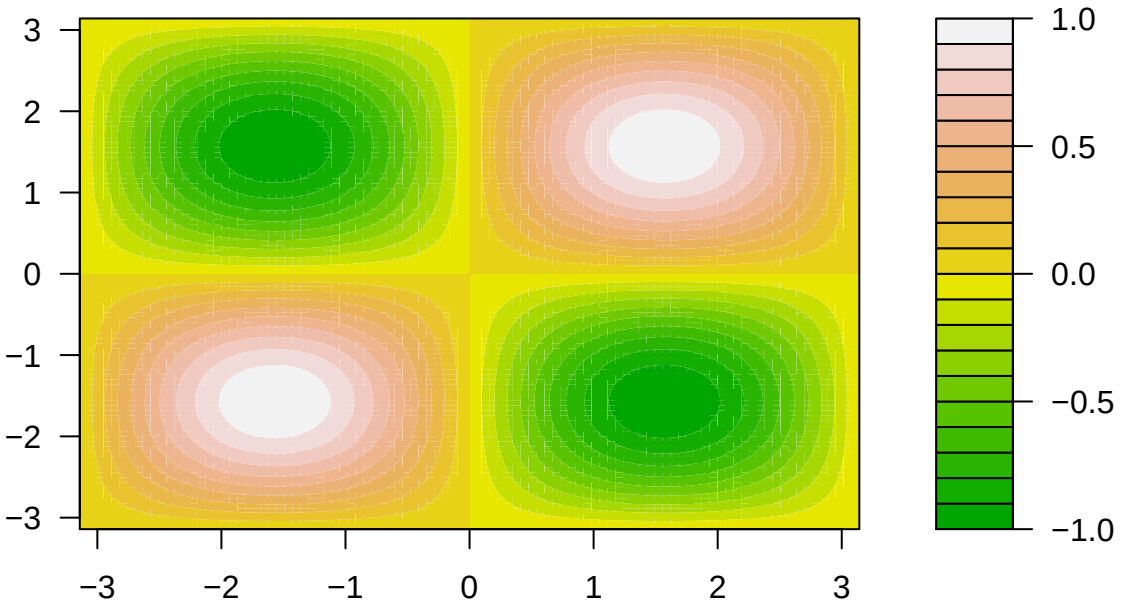


c

Now use `filled.contour` to make a contourplot of the data from **a**. Change the colourscheme to make use of `terrain.colors`.

Answer:

```
filled.contour(x, y, f_x_y, color.palette = terrain.colors)
```



d

Instead of using `outer`, write some code to perform the task you performed using `outer` yourself. Create an appropriately sized matrix, and use two nested `for` loops (i.e. a `for` loop *inside* a `for` loop), to fill this matrix. See if the results are the same as when you used `outer`.

Answer:

```
results_matrix <- matrix(0, ncol=length(y), nrow=length(x))

for (i in 1:length(x)){
  for (j in 1:length(y)){
    results_matrix[i, j] <- sin(x[i]) * sin(y[j])
  }
}

identical(results_matrix, f_x_y)
```

```
## [1] TRUE
```

7 Reproducing the Mona Lisa... *cough*

Before you start the exercises, install the package `png`. We've talked during the lectures about how `plot` functions are basically using the graphics device as a canvas, and drawing every point, or line, on this canvas by simply telling which 'pixels' should be on or off (or of a specific colour). In this exercise we'll look at this a bit more, by drawing our own images.

a

Colours on most modern LCD or OLED screens are created by combining the colours red, green and blue. This 'coding' scheme is called 'rgb'. Look at the helpfile of `writePNG`. This `rgb` coding scheme is one way to provide data to `writePNG` to write a `png` file (an image). It takes as argument a three dimensional array,

with the first two dimensions equal to the width and the height of the picture, and the third dimensions as 'layers' for the colours red, green and blue. Go online to find the 'rgb' coding of your favorite colour. Create an array that writes a 400 by 400 pixel image of that colour. Note that most 'rgb' codings use a range of 0 through 255 to indicate the 'intensity' of a particular colour, while sometimes 0 through 1 is used: appropriately scale the colour code you find online!

```
library(png)

colour_array <- array(0, dim=c(400, 400, 3))
my_rgb <- c(144, 238, 144)/255
for (i in 1:3){
  colour_array[, , i] <- my_rgb[i]
}

writePNG(colour_array, "0_images/my_image.png")
```

b

Open the `png` file on your computer, does it look as intended?

c

Let's create a colour gradient. Create another 'empty' array of dimensions 400 by 400 by 3. Create a sequence called `x`, of 400 equally spaced values between 0 and 1. Use this sequence to fill up the rows of the first layer of your array with these values. E.g. the entire first row of the first layer should be equal to `x[1]`, the second should be equal to `x[2]` etc. Use the same sequence and another for loop to do the same for the second layer of your array, but instead of looping over the rows, loop over the columns. In the end, write your array to a `png` file and look at the result. Is the result a colour gradient?

Answer:

```
x <- seq(0, 1, length.out = 400)

gradient_array <- array(0, dim = c(400, 400, 3))

for (i in 1:length(x)) {
  gradient_array[i, , 1] <- x[i]
}

for (j in 1:length(x)) {
  gradient_array[, j, 2] <- x[j]
}

writePNG(gradient_array, "0_images/my_gradient.png")
```

Sortof, only two colours of course.

d

The third colour we're going to add by increasing it in a 'mixed' way: the intensity should increase over the rows *and* the columns. That is: it's intensity should be 0 at the [1, 1] pixel, and 1 at the [400, 400] pixel. This also means that the first column, looks exactly the same as the first row!

Use a for loop to go through the columns and the rows of the third layer of the array you used for the gradient in **c** to fill each cell appropriately.

Hint, make the other layers of the gradient 0 for now, so you can see the individual contribution of the third layer when you write the `png` file. Add back in the first two layers once you've got the correct result!

Answer:

```
for (i in 1:length(x)) {
  for (j in 1:length(x)) {
    gradient_array[i, j, 3] <- x[ceiling(i + j) / 2]
  }
}
writePNG(gradient_array, "0_images/my_gradient.png")
```