Notes for 3611: Programming in Quantitative Economics

C. Dolleris

2025-05-29

Table of contents

Preface

This is a collection of notes for the course **“Programming in Quantitative Economics”**.

Part I

Lectures

1 Introduction to the Course

1.1 Standard functions in R

R comes with a lot of standard functions. We can print stuff to the screen, invert matrices, perform OLS, generate random numbers, etc., so there is no need to reinvent the wheel every time!

These functions are also available in R packages. However, these are 'special' packages that are included by default in R and don't need to be loaded. Examples are 'stats', 'base', 'graphics',...

These functions generally have sensible names:

- `print` to print something to the screen.
- `plot` to plot figures.
- `solve` to invert a matrix.
- `lm` to perform OLS ('linear model' $\rightarrow$ 'lm')
- `sum`, `median`, `mean`,...

Obviously, there are too many to list here. R manuals are freely avaiable online https://cran.r-project.org/

Small Exercise 1

- *Open and run the* `.R` *file called* `Becoming_a_useR`*.*

Click to view “Becoming_a_useR.R”

```r
 1 # 1. Add and remove elements from the environment
 2 # Clear the environment: Type ctrl+L for clearing the console
 3 rm(list = ls())
 4 Sys.setenv(lang = "en_US")
 5 # Define an integer
 6 iN <- 8
 7
 8 # Remove the object from the environment
 9 rm(iN)
10
11 # Reintroduce the integer
12 iN <- 8
13
14 # 2. There is difference between "=" and "<-" in R. Let's discuss an example
15 # A matrix with zero elements can be generated by a syntax: matrix(1, ncol = Nc, nrow = Nr
16
17 mE <- matrix(0, ncol =  2)  #Saying to R the number of columns is 2
18 ncol                        #Comment
19 mR <- matrix(0, ncol <- 2)  #Saying to R the number of ?rowss is 2: 2 is assigned to ncol a
20 ncol                        #Comment
21 mE # Print matrix E
22 mR # Print matrix R
23
24 # 3. Some error messages
25 # changing the value of iN
26 iN <- 10
27 cat("... but now it is ", iN, "\n")
28 cat("What am I printing now? ", iN==10, "\n") # Comment
29 cat("Now I get an error... ", iS, "\n")       # Why?
```

- *Can you understand what each line of the code does?*
- *What is the output?*
- *What are the differences in the matrices defined in line 17-19?*
  - The matrix `mE` is defined properly with 2 columns. The matrix `mR` has 2 rows.
- *Try to run lines 27-29. Can you comment the output?*
  - The error is received since the variable `iS` is not defined.

Small Exercise 2

- *Open and run the* `.R` *file called* `Exercise_error`.

Click to view "Exercise_error.R"

```r
# 0. Clear the environment:  CTRL + L to clear the Console
rm(list=ls())
Sys.setenv(lang = "en_US")
# Define vector X
vX <- c(-1,2)

# Why did the following output produce NAN output?
cat("This is square root of X: ", sqrt(vX), "\n")
     # Why?
```

- *Why do you get an error message?*
  - There is no error message. There is a warning message.
- *Which output do you get from* $\sqrt{-1}$*?*
  - Naturally, you get an error as the square root of a negative number isn't possible.

1.2 External packages in R

More than 12,000 external packages are avaiable on CRAN: https://cran.r-project.org/web/packages/.

Knowing the name of a package, say `("rugarch")`, this can be installed in R with `install.packages("rugarch")`.

After a package has been installed, in order to load it into the R environment, we need to run: `library("rugarch")`.

1.3 Types of variables

The function `class()` allows you to find the class of an R object. For instance:

```r
cfoo = "sun"
class(cfoo)
#> [1] "character"
```

```
bfoo = TRUE
class(bfoo)
#> [1] "logical"
```

```
dfoo = 1.785632458
class(dfoo)
#> [1] "numeric"
```

The "`foo`" terminology stands for a variable which is not important, i.e., a variable where intermediate results are stored. We will later see that "c", "b" , and "d" before the "foo" actually mean something!

1.3.1 Vectors, Matrices, Arrays, Data frames

Data can be organized in different ways in R depending on the particular needs. If we plan to do linear algebra, we want to use:

```
vY = c(1, 2, 5.48652) # vectors
mY = matrix(c(1, 2, 3, 4, 5, 6, 7, 8, 9.485), ncol = 3) # matrices
aY = array(1:27, dim = c(3, 3, 3)) # arrays
```

If we want to organize data (numeric/character or mixed), we use:

```
d = data.frame(x = 1, y = 1:10, fac = LETTERS[1:10]) # data frame
```

Vectors, matrices and ararays can be: "numeric" or "character" i.e., they can contain only one of the two types of variables (we cannot have a vector with some elements "numeric" and otheres "character"). Data.frames can contain both.

1.3.2 Access elements

To access elements of vectors, matrices and arrays, we use the square brackets:

```
vY[1]
#> [1] 1
vY[1:2]
#> [1] 1 2
vY[c(3,1)]
#> [1] 5.48652 1.00000
```

```r
mY[1, 1]
#> [1] 1
mY[1, ]
#> [1] 1 4 7
mY[1, c(1,3)]
#> [1] 1 7
aY[1, 1, 1]
#> [1] 1
aY[1, 1, ]
#> [1]  1 10 19
```

1.4 How to get help

When you are in trouble the `help()` function is your friend! Whenever you want to understand the functioning of any function, say `list`, you type:

```r
help(list)
```

This is equivalent to:

```r
?list
```

If you don't remember the full function name, but only a part of it (or something related to it), you can use the "`??`" operator, for example:

```r
??download
```

Small Exercise 3

- *Open and run the* `.R` *file called* `Get_2_know_the_helpeR.R`.

Click to view “Get_2_know_the_helpeR.R”

```r
#Clear the environment
rm(list = ls())
Sys.setenv(lang = "en_US")

# Define a vector A
vA <- c(1, 2, 3, 4, 5, 6)

# Define a 2x3 matrix A
mA <- matrix(vA, nrow = 2, ncol = 3, byrow = TRUE)

# Define a 2x2 matrix B
mB <- matrix(c(9, 8, 7, 6), ncol = 2, byrow = TRUE, nrow = 2)

# Print matrix B
mB

# Find out what the function crossprod (from the "base" package) does via the help() functi
mC <- crossprod(mA,mB)
mC

# The below matrix will give you an error. Try to reproduce the matrix mC via linear algebr
mD <- mA %*% mB
# Hint: Try ??transpose and look for Matrix Transpose.
mD <- t(mA) %*% mB
mD # Print matrix D

# Use one of the below to test if mC and mD are equal
all.equal(mC,mD) # Check if all mC and mD are equal, returns "TRUE" if this is mC = mD.
vCheck <- c(mC) == c(mD) # Form a vector of booleans, look up ?'=='
all(vCheck) # Check if all mC and mD are equal, returns "TRUE" if mC = mD.
```

- *Run the code line by line. Do you understand it?*
- *Try to look up some of the functions with the* `help`*-function.*
- *Discover how to transpose* `mA` *to obtain* `mD`*?*

```r
md <- t(mA) %*% mB
```

- *Evaluate whether your* `mD` *is equal to* `mC`.

```r
all.equal(mC,mD) # Check if all mC and mD are equal, returns "TRUE" if this is mC = mD.
vCheck <- c(mC) == c(mD) # Form a vector of booleans, look up ?'=='
all(vCheck) # Check if all mC and mD are equal, returns "TRUE" if mC = mD.
```

Small Exercise 4

- *Try to install the package "Rfast". (Hint: Go back a couple of slides to find the relevant function).*

```r
install.packages("Rfast")
```

- *Load the package and try the following code:*

```r
library("Rfast") # Loads the package
mA <- matrix(1:100,10,10)
rowVars(mA)
```

- *There are only a finite amount of names to call functions (especially if it has to make sense). What happens if you have two packages that have a similar name? E.g.,* `rowVars` *from* `Rfast` *and* `matrixStats`*?*

Either use:

```r
detach("package:Rfast", unload=TRUE)
```

Or by the use of namespaces, i.e., `Rfast::rowVars(mA)` to run the function from the package we want. (This syntax also carries over to other programming languagues, e.g., directly to C++)

1.5 Additional code in R

Click to view “Week1_lecture1.R”

2 Getting Started with R

2.1 Working directory

```r
getwd()
sDir <- "G:/foo" # this path needs to exist
setwd(sDir)
getwd()
```

```r
# Assign a new path to create the new directory
sPath <- getwd()
sDir <- file.path(sPath, "new_dir2")
dir.create(sDir) # Creates a new directory
setwd(sDir) # set WD to sDir
```

Small Exercise 1

```r
source("foo.r") # tell R to Source/Run the .R file/script
```

Click to view “Example_1.R”

```r
rm(list = ls()) # clear the environment
cat("\014")     # Clear console:equivalent to ctrl + L


getwd() # Check working directory is correct

dir() # Check that the directory contain the desired files.

# Source the functions contained in the script "example1_fun.r"
source("example1_fun.r")

squaring(-3)  # Calling the function squaring with input -3

squaring(3)   # Calling the function squaring with input 3

squaring("a") # Calling the function squaring with input "a"

clc() # Use the newly defined clc() function to clear console
```

Click to view “example1_fun.R”

```r
# This file contains two functions.
# At this stage it is not necessary that you completely understand the functions, they wil

## Function 1: squaring
# input numeric types: dX
# Output the square of the input
squaring <- function(dX) {
  dX^2
}


## Function 2: clc
# non-input function to clear console
clc <- function() {
cat("\014") # "\014" is equivalent to ctrl + L (Windows) or cmd + L (MacOS)
}
```

2.2 Arithmetic

```r
(1 + 1/100)^100
#> [1] 2.704814
```

2.3 Variable types

2.3.1 Defining variables

```
iX <- 100
iX = 100
# These are equivalent
```

Becoming_a_useR.R

```
iX
#> [1] 100
(1 + 1/iX)^iX
#> [1] 2.704814
iX <- iX + 1 # is allowed
iX
#> [1] 101
```

2.3.2 Variable types and syntax

-
-
  -
  -
  -
  -
  -
-

2.3.3 Variable names and Hungarian notation

-

-
-

2.3.4 Integers

2.3.5 Doubles

2.3.6 Precision

2.4 Built-in

2.4.1 Character/strings

```r
sText <- "Hello world"
substr(sText, start = 6, stop = nchar(sText))
#> [1] " world"
```

2.4.2 Built-in functions

```r
exp(1)
#> [1] 2.718282
options(digits = 16)
exp(1)
#> [1] 2.718281828459045
pi
#> [1] 3.141592653589793
sin(pi/6)
#> [1] 0.4999999999999999
```

2.4.3 Functions

```
seq(from = 1, to = 9, by = 2)
#> [1] 1 3 5 7 9
```

```
seq(from = 1, to = 3)
#> [1] 1 2 3
```

```
1:4
#> [1] 1 2 3 4
```

2.4.4 Arguments of functions

```
seq(1, 9, 2)
#> [1] 1 3 5 7 9
```

```
seq(by = 2, to = 9, from = 1)
#> [1] 1 3 5 7 9
# Is different from
seq(2, 9, 1)
#> [1] 2 3 4 5 6 7 8 9
```

2.5 Vectors

```
vX = c(1, 3, 4)
rep(vX, 3)
#> [1] 1 3 4 1 3 4 1 3 4
```

```
vX <- seq(1, 20, by = 2)
vY <- rep(3, 4)
vZ <- c(vY, vX)
vZ
#>  [1]  3  3  3  3  1  3  5  7  9 11 13 15 17 19
```

2.5.1 Vector operations

```
vX <- c(1, 2, 3)
vY <- c(4, 5, 6)
vX * vY
#> [1]  4 10 18
vX + vY
#> [1] 5 7 9
vY ^ vX
#> [1]   4  25 216
```

```
c(1, 2, 3) + c(1, 2)
#> Warning in c(1, 2, 3) + c(1, 2): longer object length is not a multiple of
#> shorter object length
#> [1] 2 4 4
```

```
c(1, 2, 3, 4) + c(1, 2)
#> [1] 2 4 4 6
(1:10) ^ c(1, 2)
#>  [1]   1   4   3  16   5  36   7  64   9 100
```

```
2 + c(1, 2, 3)
#> [1] 3 4 5
2 * c(1, 2, 3)
#> [1] 2 4 6
```

```
1:20 %% 3
#>  [1] 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0 1 2
```

2.6 Missing values

2.6.1 Missing date: NA

```
vA <- c(11, NA, 13)
vA
#> [1] 11 NA 13
```

```
mean(vA) # NAs can propagate
#> [1] NA
```

```r
mean(vA, na.rm = TRUE) # NAs can be removed
#> [1] 12
```

2.6.2 Searching for NAs

```r
vA <- c(11, NA, 13)
is.na(vA) # identify missing elements
#> [1] FALSE  TRUE FALSE
```

```r
any(is.na(vA)) # are any missing?
#> [1] TRUE
```

```r
na.omit(vA)
#> [1] 11 13
#> attr(,"na.action")
#> [1] 2
#> attr(,"class")
#> [1] "omit"
```

2.7 Small Exercise

-
    -
-
    -
-

  –

•

•

  –

•

  –

  –

Click to view the complete solution to the exercise

```r
rm(list = ls()) # clear environment
cat("\014")     # Clear console:equivalent to ctrl + L

# Small exercise with vectors and missing values
# Exercises 1
s <- c(5.01,"hello",FALSE,10L)
typeof(s) # Note the vector is of mode "character"
s <- c(5.01,FALSE,10L)
typeof(s) # Without the character element the vector is now of mode "double"
s <- c(FALSE,10L)
typeof(s) # ... Now of mode integer
s <- c(FALSE)
typeof(s) # ... Now of mode logical

# Exercise 2
vY <- c(NaN,NA,6L) # define vector
vY
is.na(vY)  # look for NA-value
is.nan(vY) # look for NaN-value
# NaN: "Not a Number", used for things that cannot be calculated
# NA: "Not Available", values that are unknown or missing

# NaN examples:
0/0       # is NaN
sqrt(-1) # is NaN
Inf-Inf  # is NaN
NaN/NA   # is NaN

## Extra question..
# Can NA matrices be used for more memory efficient
# usage as predefined matrix/vectors prior to i.e. a loop?
object.size(matrix(NA,1000,1000)) # Size: 4000216 bytes
object.size(matrix(0,1000,1000))  # Size: 8000216 bytes
```

2.8 Logical

2.8.1 Logical expression

-
-
-
-
-
-
-
-
-

```
vX = c(1, 2, 3, 4)
vX == 2
#> [1] FALSE  TRUE FALSE FALSE
vX != 2
#> [1]  TRUE FALSE  TRUE  TRUE
```

```
vX <- c(1, 3, 4, 18)
vX > 2
#> [1] FALSE  TRUE  TRUE  TRUE
vX[vX > 2 & vX < 10]
#> [1] 3 4
```

```
subset(vX, subset = vX > 2)
#> [1]  3  4 18
```

2.8.2 Logical expressions: & and |

```
vX = 4
vY = vX > 2
vY
#> [1] TRUE
vZ = vX < 3
vZ
#> [1] FALSE
```

```
vY & vZ
#> [1] FALSE
```

```
vY | vZ
#> [1] TRUE
```

2.8.3 Sequential && and ||

```
dX <- 0
dX * sin(1/dX) == 0
#> Warning in sin(1/dX): NaNs produced
#> [1] NA
(dX == 0) | (sin(1/dX) == 0)
```

```
#> Warning in sin(1/dX): NaNs produced
#> [1] TRUE
(dX == 0) || (sin(1/dX) == 0)
#> [1] TRUE
```

```
vX <- c(1:10) # Note: c() is not necessary here.
vX
#>  [1]  1  2  3  4  5  6  7  8  9 10
bTest <- FALSE
(vX == 1 & bTest == TRUE)
#>  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
(vX == 1 && bTest == TRUE) # Not even possible to run
#> Error in vX == 1 && bTest == TRUE: 'length = 10' in coercion to 'logical(1)'
(vX == 2 && bTest == FALSE)
#> Error in vX == 2 && bTest == FALSE: 'length = 10' in coercion to 'logical(1)'
(vX == 1 && bTest == FALSE)
#> Error in vX == 1 && bTest == FALSE: 'length = 10' in coercion to 'logical(1)'
```

2.9 Matrices

```
mA <- matrix(1:6, nrow = 2, ncol= 3, byrow = TRUE)
```

```
dim(mA)
#> [1] 2 3
```

-
-
-

-
-
-

2.9.1 Multiplication with matrices

```
mA <- matrix(c(3, 5, 2, 3), nrow = 2, ncol = 2)
mB <- matrix(c(9, 4, 1, 2), nrow = 2, ncol = 2)
mA
#>      [,1] [,2]
#> [1,]    3    2
#> [2,]    5    3
mB
#>      [,1] [,2]
#> [1,]    9    1
#> [2,]    4    2
```

```
mA * mB
#>      [,1] [,2]
#> [1,]   27    2
#> [2,]   20    6
```

```
mA %*% mB
#>      [,1] [,2]
#> [1,]   35    7
#> [2,]   57   11
```

```
kronecker(mA, mB)
#>      [,1] [,2] [,3] [,4]
#> [1,]   27    3   18    2
#> [2,]   12    6    8    4
#> [3,]   45    5   27    3
#> [4,]   20   10   12    6
```

2.10 Factors

-
-

-

-

```
phys.act <- c("L", "H", "H", "L", "M", "M")
phys.act[2] > phys.act[1]
#> [1] FALSE
phys.act <- factor(phys.act, levels = c("L", "M", "H"), ordered = TRUE)
is.ordered(phys.act)
#> [1] TRUE
phys.act[2] > phys.act[1]
#> [1] TRUE
```

2.11 Lists

```
my.list <- list("one", TRUE, 3, c("f", "o", "u", "r"))
my.list[[1]]
#> [1] "one"
mode(my.list[[1]])
#> [1] "character"
```

```
my.list[1]
#> [[1]]
#> [1] "one"
mode(my.list[1])
#> [1] "list"
```

```
my.list <- list(first = "one", second = TRUE, third = 3, fourth = c("f", "o", "u", "r"))
names(my.list)
#> [1] "first"  "second" "third"  "fourth"
my.list$second
#> [1] TRUE
```

```
my.list <- list("one", TRUE, 3, c("f", "O", "u", "r"))
names(my.list) <- c("first", "second", "third", "fourth")
```

```
x <- list(1, c(2, 3), c(4, 5, 6))
unlist(x)
#> [1] 1 2 3 4 5 6
```

2.12 Dataframes

```r
mData <- data.frame(Plot = c(1, 2, 2, 5, 8, 8),
                    Tree = c(2, 2, 3, 3, 3, 2),
                    Species = c("DF", "WL", "GF", "WC", "WC", "GF"),
                    Diameter = c(39, 48, 52, 35, 37, 30),
                    Height = c(20.5, 33.0, 30.0, 20.7, 22.5, 20.1))
```

2.12.1 Dataframes: Extract

```
mData$Diameter
#> [1] 39 48 52 35 37 30
mData[["Diameter"]]
#> [1] 39 48 52 35 37 30
mData[[4]]
#> [1] 39 48 52 35 37 30
mData[, 4]
#> [1] 39 48 52 35 37 30
mData[, "Diameter"]
#> [1] 39 48 52 35 37 30
```

2.12.2 Dataframes: Assign

```
mData$newdata <- c(1, 2, 3, 4, 5, 6)
```

```
mData$newdata2 <- TRUE
```

```
mData$newdata3 <- c("one", "two")
mData
#>   Plot Tree Species Diameter Height newdata newdata2 newdata3
#> 1    1    2      DF       39   20.5       1     TRUE      one
#> 2    2    2      WL       48   33.0       2     TRUE      two
#> 3    2    3      GF       52   30.0       3     TRUE      one
#> 4    5    3      WC       35   20.7       4     TRUE      two
#> 5    8    3      WC       37   22.5       5     TRUE      one
#> 6    8    2      GF       30   20.1       6     TRUE      two
```

2.13 Small exercises with logical operators, matrices, lists and dataframes

-

-
-
-

-

-

-

```
mA <- matrix(1:40, 10, 4)
mA # it's filled by columns
#>       [,1] [,2] [,3] [,4]
#>  [1,]    1   11   21   31
#>  [2,]    2   12   22   32
#>  [3,]    3   13   23   33
#>  [4,]    4   14   24   34
#>  [5,]    5   15   25   35
#>  [6,]    6   16   26   36
#>  [7,]    7   17   27   37
#>  [8,]    8   18   28   38
#>  [9,]    9   19   29   39
#> [10,]   10   20   30   40
dfA <- as.data.frame(mA) # convert to dataframe
colnames(dfA) <- c("one", "two", "three", "four") # add headers
dfA # view dataframe
#>    one two three four
#> 1    1  11    21   31
#> 2    2  12    22   32
#> 3    3  13    23   33
#> 4    4  14    24   34
#> 5    5  15    25   35
#> 6    6  16    26   36
#> 7    7  17    27   37
#> 8    8  18    28   38
#> 9    9  19    29   39
#> 10  10  20    30   40
dfA$five <- FALSE
dfA # view the new column
#>    one two three four  five
#> 1     1  11    21   31 FALSE
#> 2     2  12    22   32 FALSE
#> 3     3  13    23   33 FALSE
#> 4     4  14    24   34 FALSE
#> 5     5  15    25   35 FALSE
#> 6     6  16    26   36 FALSE
#> 7     7  17    27   37 FALSE
#> 8     8  18    28   38 FALSE
#> 9     9  19    29   39 FALSE
#> 10  10  20    30   40 FALSE
is.list(dfA) # check if it's a list
#> [1] TRUE
is.data.frame(dfA) # check if it's a dataframe
#> [1] TRUE
dfA[, !(names(dfA) == "two")] # remove the column with the name two
#>    one three four  five
#> 1    1    21   31 FALSE
#> 2    2    22   32 FALSE
#> 3    3    23   33 FALSE
#> 4    4    24   34 FALSE
#> 5    5    25   35 FALSE
#> 6    6    26   36 FALSE
#> 7    7    27   37 FALSE
#> 8    8    28   38 FALSE
#> 9    9    29   39 FALSE
#> 10  10    30   40 FALSE
```

2.14 Additional code in R

Click to view “Week1_lecture1.R”

3 Basic Programming

-
-
-

```r
source("prog.r")
```

3.1 The `if` statement

```r
if (logical_expression) {
  expression_1 #code to run if logical_expression is TRUE
  ...
}
```

```
if (logical_expression) {
  expression_1
  ...
} else {
  expression_2
  ...
}
```

```
dX <- rnorm(1) # random number from standard normal distribution
if (dX > 0) {
  print("x is positive")
} else {
  print("x is non-positive")
}
#> [1] "x is positive"
```

3.1.1 Small Exercise

-
-
-

```
dX <- rnorm(1) # Generates 1 observation, mean 0 and standard deviation 1
if (dX < 0) {
  cat("The value of x is ", dX, " and its square is ", dX^2)
} else {
  cat("The value of x is ", dX, " and its cube is ", dX^3)
}
#> The value of x is  0.6551145  and its cube is  0.2811587
```

3.1.2 Nested if statements

```
if (logical_expression_1) {
  expression_1 # code to run if logical_expression_1 is TRUE
  ...
} else if (logical_expression_2) {
  expression_2 # code to run if logical_expression_1 is FALSE and logical_expression_2 is TRU
  ...
} else if (logical_expression_2) {
  expression_3 # code to run if logical_expression_1 is FALSE and logical_expression_2 is FAI
  ...
} else {
  expression_4 # code to run if all logicals are FALSE
}
```

```
dX - runif(1) # Random number from the uniform distribution
#> [1] -0.328898
if (dX < 0.5) {
  print("x is less than 0.5")
} else if (dX < 0.75) {
  print("x is less than 0.75")
} else {
  print ("x is greather than or equal to 0.75")
}
#> [1] "x is less than 0.75"
```

3.1.3 An extension to the exercise solution above

```
dX <- rnorm(1) # random number from the Gaussian distribution
if (dX < 0) {
  cat("the value of x is ", dX, " and its square is ", dX^2)
} else if (dX >= 0) {
  cat("x is non-negative, taking the value ", dX, " and its square is ", dX^2)
} # Note the tautological nature of the logical expression in the else if part...
#> x is non-negative, taking the value  0.7273106  and its square is  0.5289807
```

3.2 The `for` loop

```
for (x in vector) {
  expression_1
  ...
}
```

```
for (iX in 1:50) {
  print(iX)
}
```

```
vX <- seq(1, 9, by = 2)
dSum_x <- 0
for (iX in vX) {
  dSum_x <- dSum_x + iX
  cat("The current loop element is ", iX, "\n")
  cat("The cumulative total is ", dSum_x, "\n")
}
#> The current loop element is  1
#> The cumulative total is  1
#> The current loop element is  3
#> The cumulative total is  4
#> The current loop element is  5
#> The cumulative total is  9
#> The current loop element is  7
#> The cumulative total is  16
#> The current loop element is  9
#> The cumulative total is  25
sum(vX)
#> [1] 25
cumsum(vX)
#> [1]  1  4  9 16 25
```

3.2.1 Exercise: a combination of for loops and if conditions

-
-
-
-

```r
iPos <- 0
iNeg <- 0 # This integer is technically redundant, as the knowledge of the positive values ar
vSimVals <- vector()
for (iX in 1:100) {
  dX <- rnorm(1)
  if (dX > 0) {
    iPos <- iPos + 1
  } else {
    iNeg <- iNeg + 1
  }
  vSimVals[iX] <- dX
}
cat("The fraction of positive realizations is: ", iPos, "/", (iNeg + iPos), " or ", iPos / (i
#> The fraction of positive realizations is:  50 / 100  or  0.5
cat("The first of the simulated values are: ", vSimVals[1:5], "...")
#> The first of the simulated values are:  -0.7997055 -0.02644413 -1.237805 -0.07337126 1.483
```

3.3 The `while` loop

```r
while (logical_expression) {
  # This should have an impact on logical expression
  expression_1
  ...
}
```

```
iN <- 1
while (iN <= 6) {
  print(iN ^ 2)
  iN <- iN + 1
}
#> [1] 1
#> [1] 4
#> [1] 9
#> [1] 16
#> [1] 25
#> [1] 36
```

3.3.1 Exercise with while loops

-
-
-
-
-

```
dCurrent <- rnorm(1, mean = 100, sd = sqrt(100))
dIte <- 0
while (dCurrent >= 1) {
  dCurrent <- dCurrent / 2
  dIte <- dIte + 1
}
print(dIte)
#> [1] 7
```

Click to view an example of nested while and if statements

```r
Want.2.bake.cake <- FALSE
bowl <- matrix(0,1,1)

while(Want.2.bake.cake == FALSE) {

  Want.2.bake.cake <- (runif(1) < 0.5)

  if (Want.2.bake.cake == TRUE) {

    print("You start baking, find the appropriate bowl")
    Sys.sleep(2)

    print("looking for bowl...")

    while ((prod(dim(bowl)) != 100 | nrow(bowl) != ncol(bowl))) {

      bowl <- matrix(0,sample(1:100,1), sample(1:100,1))

      if (!(prod(dim(bowl)) != 100 | nrow(bowl) != ncol(bowl))){
        Sys.sleep(2)

        print("Wrong bowl... look for another one..")
        print("looking for another...")

      }
    }
  }
}
```

Please note that the code is actually wrong! The logical expression in the `if` condition after sampling of a bowl is only true if the correct bowl is found, not when the wrong bowl is found.

3.4 Vector-based programming

```r
vX <- (1:6) ^ 2
vX
#> [1]  1  4  9 16 25 36
```

3.4.1 Example

```r
iN <- 100
dS <- 100
for (i in 1:iN) {
  dS <- dS + i ^ 2
}
dS
#> [1] 338450
```

```r
sum((1:iN)^2)
#> [1] 338350
```

```r
iN <- 1e8
system.time({
  dS <- 0
  for (i in 1:iN) {
    dS <- dS + i ^2
  }
  dS
})
#>   bruger   system forløbet
#>     2.28     0.00     2.28
system.time({sum((1:iN) ^ 2)})
#>   bruger   system forløbet
#>     0.29     0.11     0.40
```

3.5 Load data in R

```
read.table(file, header = FALSE, sep = "", dec = ".", row.names, col.names, na.strings = "NA
```

3.5.1 Load MAERSK historical prices

```
mData <- read.table(file = "data/MAERSK-B.CO.csv", sep = ",", dec = ".", header = TRUE, row.n
head(mData)
#>               Open    High     Low   Close Adj.Close Volume
#> 2000-02-01 7640.00 7666.67 7333.33 7466.67  2267.703   4590
#> 2000-02-02 7466.67 7666.67 7400.00 7566.67  2298.074   5640
#> 2000-02-03 7741.07 7800.00 7659.00 7800.00  2368.939   3090
#> 2000-02-04 7800.00 7800.00 7525.80 7600.00  2308.197   1185
#> 2000-02-07 7660.00 7666.67 7566.67 7600.00  2308.197   1035
#> 2000-02-08 7600.00 8200.00 7600.00 8110.80  2463.332  12990
dim(mData)
#> [1] 4571    6
```

3.5.2 Descriptive statistics of the MAERSK log-returns

```
any(is.na(mData[, "Adj.Close"])) # Search for NAs
#> [1] TRUE
length(which(is.na(mData[, "Adj.Close"]))) # How many NAs?
#> [1] 59
# Compute financial log-returns omitting the NAs:
vY <- diff(log(na.omit(mData[, "Adj.Close"])))
# Compute descriptive statistics
c("mean" = mean(vY), "sd" = sd(vY), "median" = median(vY))
#>         mean           sd       median
#> 0.0003439318 0.0220065696 0.0000000000
```

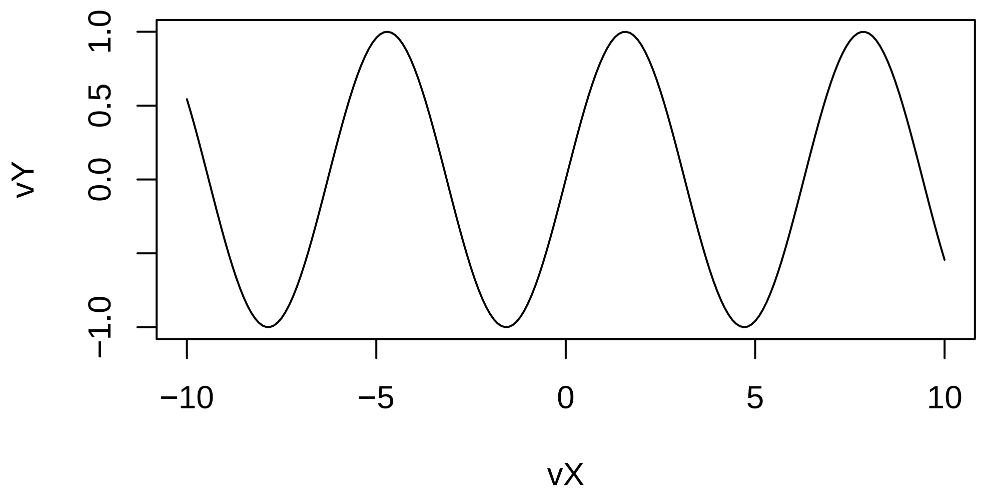
3.5.3 Output to a file

```
write.table(x, file = "", quote = TRUE, sep = " ", na = "NA", dec = ".", row.names = TRUE, co
write.table(vY, file = "MAERSK_returns.txt", row.names = FALSE, col.names = FALSE, dec = "."
```

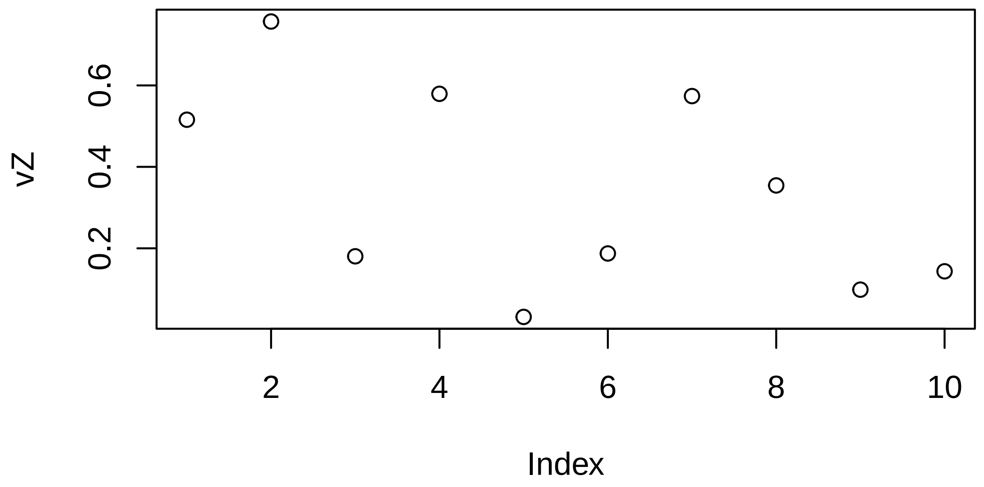
3.6 Plotting

http://ggplot2.org/

```
vX <- seq(-10, 10, 0.1)
vY <- sin(vX)
plot(vX, vY, type = "l")
```



```
vZ <- runif(10)
plot(vZ)
```

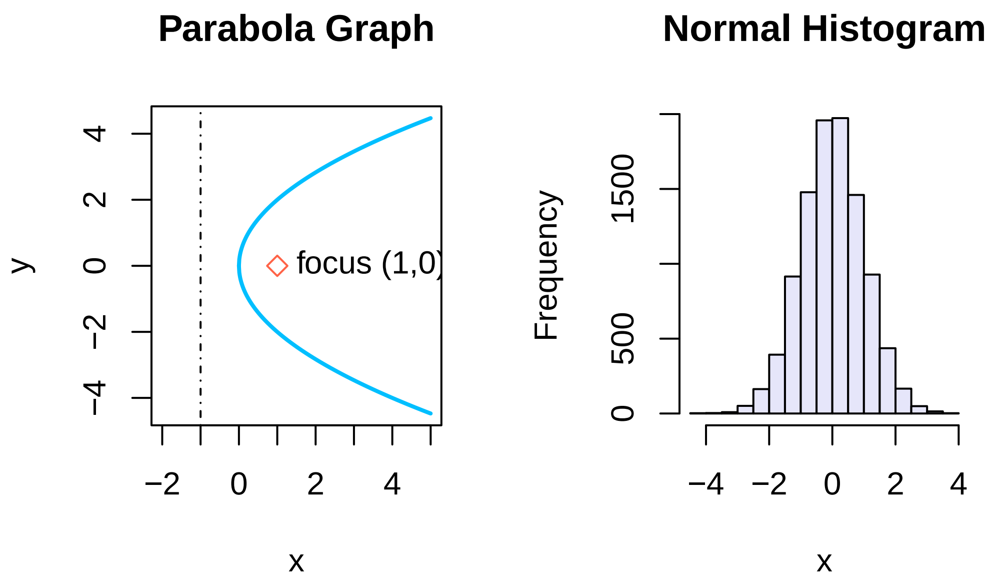


3.6.1 Basic arguments for plot

-
-
-
-
-
-
-
-
-
-
-
-
-
-
-

3.6.2 Example:

```r
par(mfrow = c(1, 2))
x <- seq(0, 5, by = 0.01)
y.upper <- 2 * sqrt(x)
y.lower <- -2 * sqrt(x)
y.max <- max(y.upper)
y.min <- min(y.lower)
plot(c(-2, 5), c(y.min, y.max), type = "n", xlab = "x", ylab = "y", main = "Parabola Graph")
lines(x, y.upper, lwd = 2, col = "deepskyblue")
lines(x, y.lower, lwd = 2, col = "deepskyblue")
abline(v = -1, lty = 4)
points(1, 0, col = "tomato", pch = 5)
text(1, 0, "focus (1,0)", pos = 4)

x = rnorm(10000)
hist(x, col = "lavender", main = "Normal Histogram")
```



3.6.3 Exercise

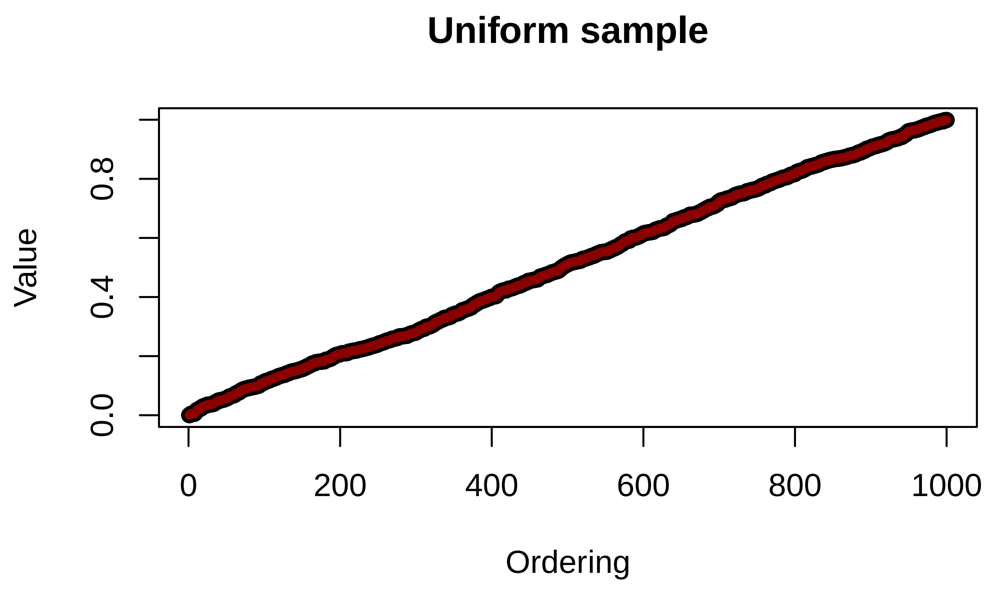
-

-

-

```
pdf("./myplot.pdf") # Your plot function with input
dev.off()
```

```
x <- sort(runif(1000), decreasing = FALSE)
plot(x, main = "Uniform sample", xlab = "Ordering", ylab = "Value") # A simple scatterplot
points(x, cex = .5, col = "dark red") # Colors
```



3.7 Additional code in R

Click to view “Week1_lecture1.R”

```
#0.1. clear the workspace
rm(list = ls())

#0.2. set the working directory
setwd("...Exercises")

#1. If statement exercise
# Simulate a standard random normal variable
dX <- rnorm(1)

# Plain if
if (dX < 0) {
dX_2 <- dX^2
print(dX_2)
} else {
dX_3 <- dX^3
print(dX_3)
}
```

4 Programming with Functions

-
-
-
-
-

4.1 Definition of functions in R

```r
name <- function(argument_1, argument_2, ...) {
  expression_1
  expression_2
  #<some other expression>
  return(output)
}
```

-
-
-

4.1.1 Running a Function

```
name(x1, x2)
```

4.1.2 Examples

```
power_fct <- function(vX, power) {
  return(vX ^ power)
}
```

```
power_fct(vX = 2, power = 3)
#> [1] 8
```

```
swap <- function(vZ) {
  dTemp <- vZ[2]
  vZ[2] <- vZ[1]
  vZ[1] <- dTemp
  return(vZ)
}
```

```r
swap(vZ = c(7, 8, 9))
#> [1] 8 7 9
```

4.1.3 Notes on Functions

4.1.4 Small Exercises

-

```r
## There are multiple solutions to this:
# The efficient
fEfficientSum <- function(iN) {
  # Input validation: Check if n is a non-negative integer
  if (!is.numeric(iN) || iN < 0 || iN %% 1 != 0) {
    stop("Input must be a non-negative integer.")
  }
  return(iN * (iN + 1) / 2)
}
# The for loop
fForSum <- function(iN) {
  sum <- 0
  for (iIndex in 1:iN) {
    sum = sum + iIndex
  }
  return(sum)
}
# The recursive
fRecursiveSum <- function(iN) {
  if (iN <= 0) {
    return(iN)
  } else {
    return(iN + fRecursiveSum(iN - 1))
  }
}
fEfficientSum(5)
```

```r
#> [1] 15
fForSum(5)
#> [1] 15
fRecursiveSum(5)
#> [1] 15
```

-

```r
fRemoveMissing <- function(vInput) {
  if (any(is.na(vInput)) == TRUE) {
    return(na.omit(vInput))
  } else {
    return(vInput)
  }
}
vVector <- c(1, NA, 2)
vVector <- fRemoveMissing(vVector)
#> 1 2
```

4.1.5 Example: roots of a quadratic function

```r
rootfinder <- function(dA, dB, dC) {
  vOut = numeric(2)
  vOut[1] = (-dB + sqrt(dB ^ 2 - 4 * dA * dC)) / (2 * dA)
  vOut[2] = (-dB - sqrt(dB ^ 2 - 4 * dA * dC)) / (2 * dA)
  return(vOut)
}
```

-
-

-
-
-

```
rootfinder <- function(dA, dB, dC) {
  if (dA == 0 && dB == 0 && dC == 0) {
    vRoots <- Inf
  } else if (dA == 0 && dB == 0) {
    vRoots <- NULL
  } else if (dA == 0) {
    vRoots <- -dC / dB
  } else {
    # calculate the discriminant
    dDelta <- dB^2 - 4 * dA * dC
    if (dDelta > 0) {
      vRoots <- (-dB + c(1, -1) * sqrt(dDelta)) / (2 * dA)
    } else if (dDelta == 0) {
      vRoots <- rep(-dB / (2 * dA), 2)
    } else {
      di <- complex(1, 0, 1)
      vRoots <- (-dB + c(1, -1) * sqrt(-dDelta) * di) / (2 * dA)
    }
  }
  return(vRoots)
}
```

```r
# source("./scripts/rootfinder.r)
rootfinder(dA = 1, dB = 1, dC = 0)
#> [1]  0 -1
rootfinder(dA = 1, dB = 0, dC = -1)
#> [1]  1 -1
rootfinder(dA = 1, dB = -2, dC = 1)
#> [1] 1 1
rootfinder(dA = 1, dB = 1, dC = 1)
#> [1] -0.5+0.8660254i -0.5-0.8660254i
```

4.1.6 Advantages of coding with functions

4.1.7 Variables inside a function

```r
test <- function(x) {
  y <- x + 1
  return(y)
}
test(1)
x # This will return an error
y # This will return an error
```

4.1.8 The scope of a variable

```r
test2 <- function(x) {
  y <- x + z
  return(y)
}
z <- 1
test2(1)
#> [1] 2
z <- 2
test2(1)
#> [1] 3
```

```r
test2 <- function(x, z) {
  y <- x + z
  return(y)
}
```

4.2 Arguments of a function

```
formals(rootfinder)
#> $dA
#>
#>
#> $dB
#>
#>
#> $dC
```

4.2.1 Default arguments

```
test3 <- function(x = 1) {
  return(x)
}
test3(2)
#> [1] 2
test3()
#> [1] 1
```

```
formals(matrix)
#> $data
#> [1] NA
#>
#> $nrow
#> [1] 1
#>
#> $ncol
#> [1] 1
#>
#> $byrow
#> [1] FALSE
#>
#> $dimnames
#> NULL
matrix()
#>      [,1]
#> [1,]   NA
```

```
test4 <- function(x = 1, y = 1, z = 1) {
  return(x * 100 + y * 10 + z)
}
test4(2, 2)
#> [1] 221
test4(y = 2, z = 2)
#> [1] 122
```

4.2.2 Default arguments: Partial matching

```
MyFunc <- function(vX, method = c("add", "multiply")) {
  method <- match.arg(method)
  if (method == "add") {
    return(sum(vX))
  } else {
    return(prod(vX))
  }
}
MyFunc(c(1, 2, 3, 4), method = "multiply")
#> [1] 24
MyFunc(c(1, 2, 3, 4), method = "add")
#> [1] 10
```

```r
method = c("yule-walker", "burg", "ols", "mle", "yw")
```

4.3 Ellipsis

```r
SquareMean <- function(vX, ...) {
  dSM <- mean(vX, ...)^2
  return(dSM)
}
SquareMean(c(1, 3, 5, NA))
#> [1] NA
SquareMean(c(1, 3, 5, NA), na.rm = TRUE)
#> [1] 9
```

4.4 Vector-based programming using functions

```r
vX <- c(1, NA, 3, 8)
is.na(vX)
#> [1] FALSE  TRUE FALSE FALSE
```

4.4.1 sapply

```r
f <- function(dX) {
  if (dX < 0) {
    stop("dX needs to be positive.")
  }
  dSum = 0.0
  iC = 0
  while (iC <= dX) {
    dSum = dSum + iC
    iC = iC + 1
  }
  return(dSum)
}
```

```r
vX <- c(1, 4, 9.478, 6, 75, 0.48)
vSum <- numeric(length(vX))
for (i in 1:length(vSum)) {
  vSum[i] <- f(vX[i])
}
vSum
#> [1]    1   10   45   21 2850    0
```

```r
vSum <- sapply(vX, f)
vSum
#> [1]    1   10   45   21 2850    0
```

```
g <- function(dX, iN) {
  if (dX < 0) {
    stop("dX needs to be positive.")
  }
  dSum <- 0.0
  iC <- 0
  while (iC <= dX) {
    dSum <- dSum + iC^iN
    iC <- iC + 1
  }
  return(dSum)
}
sapply(vX, g, iN = 2)
#> [1]      1     30    285     91  143450      0
```

4.4.2 `sapply` and `lapply`

4.4.3 `mapply`

```
mapply(rep, times = c(4, 3), x = c(2, 4))
#> [[1]]
#> [1] 2 2 2 2
#>
#> [[2]]
#> [1] 4 4 4
```

```
mapply(power_fct, vX = 1:4, power = c(2, 2, 3, 3))
#> [1]  1  4 27 64
```

4.4.4 apply

```
apply(X, MARGIN, FUN, ...)
```

-
-
-
-

```
mX <- matrix(1:16, 4, 4)
mX
#>      [,1] [,2] [,3] [,4]
#> [1,]    1    5    9   13
#> [2,]    2    6   10   14
#> [3,]    3    7   11   15
#> [4,]    4    8   12   16
```

```
apply(mX, 2, cumsum)
#>      [,1] [,2] [,3] [,4]
#> [1,]    1    5    9   13
#> [2,]    3   11   19   27
#> [3,]    6   18   30   42
#> [4,]   10   26   42   58
```

4.4.5 Exercise

-
-
-
-

```
mA <- matrix(c(1, 10, 7, 4, 8, 5, 2, 11, 6, 3, 9, 12), 3, 4, byrow = TRUE)
mA
#>      [,1] [,2] [,3] [,4]
#> [1,]    1   10    7    4
#> [2,]    8    5    2   11
#> [3,]    6    3    9   12
t(apply(mA, 1, cumsum)) # compute the row sums
#>      [,1] [,2] [,3] [,4]
#> [1,]    1   11   18   22
#> [2,]    8   13   15   26
#> [3,]    6    9   18   30
apply(mA, 2, cumsum) # compute the column sums
#>      [,1] [,2] [,3] [,4]
#> [1,]    1   10    7    4
#> [2,]    9   15    9   15
#> [3,]   15   18   18   27
apply(mA, 2, sort, decreasing = TRUE) # sort columns in decreasing order
```

```
#>      [,1] [,2] [,3] [,4]
#> [1,]    8   10    9   12
#> [2,]    6    5    7   11
#> [3,]    1    3    2    4
t(apply(mA, 1, sort, decreasing = FALSE)) # sort rows in increasing order
#>      [,1] [,2] [,3] [,4]
#> [1,]    1    4    7   10
#> [2,]    2    5    8   11
#> [3,]    3    6    9   12
sapply(mA[1, ], sin) # sine of first row using sapply
#> [1]  0.8414710 -0.5440211  0.6569866 -0.7568025
sapply(mA[, 2], cos) # # cosine of second column using sapply
#> [1] -0.8390715  0.2836622 -0.9899925
# the easier way is to do 'sin(mA[, 1])'
```

4.4.6 Recursive programming

```
myfactorial <- function(iN) {
  if ((iN == 0) | (iN == 1)) {
    return(1)
  } else {
    return(iN * myfactorial(iN - 1))
  }
}
myfactorial(5)
#> [1] 120
```

4.4.7 Multiple outputs

```r
lOut = list(Par = vParam, SE = vStdErr, LLK = dLogLik)
return(lOut)
```

4.5 Comments

```r
vBeta <- c(0, 1, -5) # Initial values
```

```r
##
## FunctionName(Inputs)
##
## Purpose:
##  Description of what the function does
##
## Input:
##  List of inputs, describing what they represent
##
## Output:
##  List of outputs, describing what they represent
##
## Return value:
##  List of return values, describing what they represent
##
```

4.5.1 Exercise

-

  1.
  2.

-
-

-

```r
f <- function(iN) {
  if (iN < 2) { # stopping criterion
    return(iN)
  } else {
    return(f(iN - 1) + f(iN - 2)) # recursive part
  }
}
sapply(1:20, f) # apply function f to a vector
#>  [1]    1    1    2    3    5    8   13   21   34   55   89  144  233  377  610
#> [16]  987 1597 2584 4181 6765
# regarding the output: it's the fibonacci series
```

5 Program Efficiency and Parallel Computing

5.1 Good practice for programming

5.1.1 Programming in theory - plan ahead

•
•
•
•
  –
  –
•

5.1.2 Closer to practice:

•
  –
  –
  –
  –
  –
  –
•
  –

5.1.3 Use of Functions and computational efficiency

-

-
-
-

-
-
-
-

https://CRAN.R-project.org/view=HighPerformanceComputing

5.2 Code execution time

```
system.time({
  mA <- matrix(rnorm(1000^2), 1000, 1000)
  solve(mA)
})
#>    user  system elapsed
#>    0.94    0.00    0.97
```

```r
vX <- 0
for (i in 1:1e5) {
  vX <- c(vX, i)
}
```

```r
vX <- numeric(1e5 + 1)
for (i in 1:1e5) {
  vX[i + 1] = i
}
```

```r
system.time({
  vX <- 0
  for (i in 1:1e5) {
    vX <- c(vX, i)
  }
})
#>    user  system elapsed
#>    8.68    1.73   10.42

system.time({
  vX <- numeric(1e5 + 1)
  for (i in 1:1e5) {
   vX[i + 1] = i
  }
})
#>    user  system elapsed
#>       0       0       0
```

5.3 Loops versus vectors

```r
for (i in 1:length(vX)) vX[i] <- vX[i]^2
```

5.3.1 But... If you have to use a loop in R...

-
-

5.3.2 The flexibility vs speed trade-off

```r
library("microbenchmark")
#> Warning: package 'microbenchmark' was built under R version 4.3.3
vY <- runif(1E7)
microbenchmark(sum(vY) / length(vY), mean(vY))
#> Unit: milliseconds
#>                 expr    min      lq     mean   median     uq    max neval
#>  sum(vY)/length(vY) 7.6384 7.66925 7.734769 7.72605 7.7641 8.3797   100
#>            mean(vY) 15.3724 15.43435 15.501740 15.46510 15.5127 16.6341   100
all.equal(sum(vY) / length(vY), mean(vY))
#> [1] TRUE
```

5.3.3 Arithmetic is not always the fastest

```
library("microbenchmark")
vY <- runif(1E5)
mX <- replicate(10, runif(1E5))
microbenchmark(vY^(1/2), sqrt(vY))
#> Unit: microseconds
#>      expr    min      lq     mean  median     uq    max neval
#>  vY^(1/2) 2469.9 2516.95 2661.301 2714.85 2744.1 2891.5   100
#>  sqrt(vY)  345.6  351.75  545.903  572.20  584.6 4335.4   100
all.equal(vY^(1/2), sqrt(vY))
#> [1] TRUE
microbenchmark(t(mX) %*% vY, crossprod(mX, vY))
#> Unit: milliseconds
#>               expr    min      lq     mean  median      uq     max neval
#>       t(mX) %*% vY 2.9688 3.10205 3.756373 3.15805 3.27405 10.6380   100
#>  crossprod(mX, vY) 1.6745 1.70385 1.733298 1.71670 1.74195  1.9167   100
all.equal(t(mX) %*% vY, crossprod(mX, vY))
#> [1] TRUE
```

5.3.4 Vectorization: Tips and Tricks for Speed

-
-
-
-

5.3.5 Summing the columns of a matrix

```
mA <- matrix(rnorm(1e8), nrow = 10000)
```

1.

```
system.time({
  vColSum <- rep(NA, ncol(mA))
  for (i in 1:ncol(mA)) {
    s <- 0
    for (j in 1:nrow(mA)) {
      s <- s + mA[j, i]
    }
    vColSum[i] <- s
  }
})
#>    user  system elapsed
#>    3.37    0.01    3.39
```

2.

```
system.time(
  vColSum <- apply(mA, 2, sum)
)
#>    user  system elapsed
#>    0.70    0.38    1.11
```

3.

```
system.time({
  vColSum <- numeric(ncol(mA))
  for (i in 1:ncol(mA)) {
    vColSum[i] <- sum(mA[, i])
  }
})
#>    user  system elapsed
#>    0.38    0.24    0.64
```

4.

```
vU <- rep(1, ncol(mA))
system.time({
  vColSum <- vU %*% mA
})
#>    user  system elapsed
#>    0.17    0.00    0.17
```

5.

```
system.time({
  vColSum <- colSums(mA)
})
#>    user  system elapsed
#>    0.07    0.00    0.07
```

5.3.6 Comparison

5.3.7 Vectorized if/else statement

```r
fun <- function(vY) {
  iN <- length(vY) # Number of elements
  vD <- numeric(iN) # Pre-allocate vector
  for (i in 1:iN) {
    if(vY[i] < 0.5) vD[i] <- -1 else vD[i] <- 1
  }
  vD
}
```

```r
ifelse(vY < 0.5, -1, 1)
```

```r
library("microbenchmark")
vY <- runif(1E5)
microbenchmark(fun(vY), ifelse(vY < 0.5, -1, 1))
#> Unit: milliseconds
#>                      expr    min     lq     mean  median      uq     max neval
#>                   fun(vY) 6.6506 6.7240 6.856817 6.78885 6.88315 10.6997   100
#>  ifelse(vY < 0.5, -1, 1) 3.1478 3.2287 3.357350 3.33385 3.44685  4.5541   100
all.equal(fun(vY), ifelse(vY < 0.5, -1, 1))
#> [1] TRUE
```

```r
mY <- matrix(4E2, 200, 200)
```

```r
fun3 <- function(mY) {
  iN <- nrow(mY) # Number of rows of mY
  iK <- ncol(mY) # Number of columns of mY
  mD <- matrix(0, iN, iK) # Pre-allocate to avoid growing objects!
  for (i in 1:iN) { # Outer loop
    for (j in 1:iK) { # Nested loop
      if (mY[i, j] < 0.5) mD[i, j] <- -1 # If true
      else mD[i, j] <- 1 # Otherwise
    }
  }
  mD
}
```

```
microbenchmark(ifelse(mY < 0.5, -1, 1), fun3(mY))
#> Unit: microseconds
#>                      expr    min     lq     mean  median      uq    max neval
#>  ifelse(mY < 0.5, -1, 1)  404.2  596.2  849.110  932.50  986.50 1161.1   100
#>                  fun3(mY) 3008.7 3073.0 3181.797 3117.15 3174.25 8091.1   100
all.equal(ifelse(mY < 0.5, -1, 1), fun3(mY))
#> [1] TRUE
```

5.3.8 Subsetting/subscripting: One of the most powerful tools!

```
fun2 <- function(mY) {
  mBool <- (mY < 0.5) # Matrix of booleans
  mY[mBool] <- -1 # If true
  mY[!mBool] <- 1 # If false
  mY # output
}
microbenchmark(ifelse(mY < 0.5, -1, 1), fun2(mY), fun3(mY))
#> Unit: microseconds
#>                      expr    min      lq     mean  median      uq    max neval
#>  ifelse(mY < 0.5, -1, 1)  406.2  917.00  873.091  943.65  966.05 1098.6   100
#>                  fun2(mY)  225.0  515.15  511.020  536.20  551.00 2944.2   100
#>                  fun3(mY) 2988.7 3081.05 3141.008 3123.55 3171.05 3784.9   100
all.equal(fun2(mY), fun3(mY))
#> [1] TRUE
```

5.3.9 Exercises in vectorization

-

-

-

```r
# Using ifelse()
slowSolution <- function(vInput) {
  # ifelse(vInput <= 0, vInput <- - vInput ^ 3, ifelse(vInput <= 1, vInput <- vInput ^ 2, sq

  for (i in 1:length(vInput)) {
    ifelse(vInput[i] <= 0, vInput[i] <- - vInput[i] ^ 3, vInput[i])
    ifelse(vInput[i] <= 1, vInput[i] <- vInput[i] ^ 2, vInput[i])
    ifelse(vInput[i] > 1, vInput[i] <- sqrt(vInput[i]), vInput[i])
  }
  return(vInput)
}

# Fast solution using indexing
fastSolution <- function(vInput) {
  vInput[vInput <= 0] <- - vInput[vInput <= 0] ^ 3
  vInput[vInput <= 1] <- vInput[vInput <= 1] ^ 2
  vInput[vInput > 1] <- sqrt(vInput[vInput > 1])
  return(vInput)
}

set.seed(1)
vX <- rnorm(10)

microbenchmark(slowSolution(vX), fastSolution(vX))
#> Unit: microseconds
#>              expr  min   lq    mean median    uq    max neval
#>  slowSolution(vX) 38.1 42.9 119.559   43.3 43.75 7665.5   100
#>  fastSolution(vX)  2.1  2.5  45.091    2.8  3.20 4204.8   100
all.equal(slowSolution(vX), fastSolution(vX))
#> [1] TRUE
```

```
# Vectorization of rowMeans
vX <- rnorm(10000)
mX <- matrix(vX, 100, 100)
fRowMeans <- function(mInput) {
  return(rowSums(mInput) / ncol(mInput))
}

microbenchmark(rowMeans(mX), fRowMeans(mX))
#> Unit: microseconds
#>           expr  min   lq   mean median   uq    max neval
#>   rowMeans(mX) 20.9 21.1 21.913  21.20 21.3   38.5   100
#>  fRowMeans(mX) 21.7 21.9 40.044  22.05 22.2 1701.6   100
all.equal(rowMeans(mX), fRowMeans(mX))
#> [1] TRUE
```

5.4 Parallel processing

5.4.1 Warning before parallellization

1.
2.
3.

5.4.2 Parallel Processing

```
library(parallel)
detectCores()
#> [1] 6
```

5.4.3 Creating a cluster

```
cluster <- makeCluster(2)
```

```
clusterCall(cluster, function(x) print("Pick me!"))
#> [[1]]
#> [1] "Pick me!"
#>
#> [[2]]
#> [1] "Pick me!"
```

5.4.4 Running jobs in parallel

-
-
-

```r
waste.of.time <- function(x) for (i in 1:10000) i
system.time(lapply(1:10000, waste.of.time))
#>    user  system elapsed
#>    0.83    0.00    0.84
system.time(parLapply(cl = cluster, 1:10000, waste.of.time))
#>    user  system elapsed
#>    0.02    0.00    0.42
```

5.4.5 Kill your cluster

```r
stopCluster(cluster)
```

5.4.6 Small exercises on the parallel universe

-
-
-
-
-
-
-
-

```r
# Install if necessary and load the library quietly
if(!require("parallel")) install.packages("parallel")
suppressMessages(library(parallel))

# Make cluster with more cores than 1 if possible
if (detectCores() > 1) cluster <- makeCluster(detectCores() - 1)

waste.of.time <- function(x) for (i in 1:10000) i
fibo <- function(n) {
  if (n < 2) return(n)
  return(fibo(n - 1) + fibo(n - 2))
}

# `fibo` is a recursive function and thus needs to be exported to the cluster
clusterExport(cluster, "fibo")

# Without a cluster
system.time(sapply(1:30, waste.of.time))
system.time(sapply(1:30, fibo))

# Using the cluster
system.time(parSapply(cl = cluster, 1:30, waste.of.time))
system.time(parSapply(cl = cluster, 1:30, fibo))

# Kill the cluster
stopCluster(cluster)
```

5.5 Memory issues

```
dA <- 5
ls()
#> [1] "dA"
rm("dA")
```

5.6 Small hints

-
-
-
-
-

6 Introduction to Simulation

6.1 Seeding

-
-
-

```
set.seed(10086)
runif(5)
#> [1] 0.708768032 0.384298612 0.003025926 0.277903935 0.714458605
```

```
runif(5)
#> [1] 0.3166092 0.6863317 0.4725650 0.2655531 0.6193426
```

```
set.seed(10086)
runif(5)
#> [1] 0.708768032 0.384298612 0.003025926 0.277903935 0.714458605
```

```r
RNG.state <- .Random.seed
runif(5)
#> [1] 0.3166092 0.6863317 0.4725650 0.2655531 0.6193426
```

```r
.Random.seed <- RNG.state
runif(5)
#> [1] 0.3166092 0.6863317 0.4725650 0.2655531 0.6193426
```

6.2 Discrete Random variables

6.2.1 Simulate Discrete Random Variables

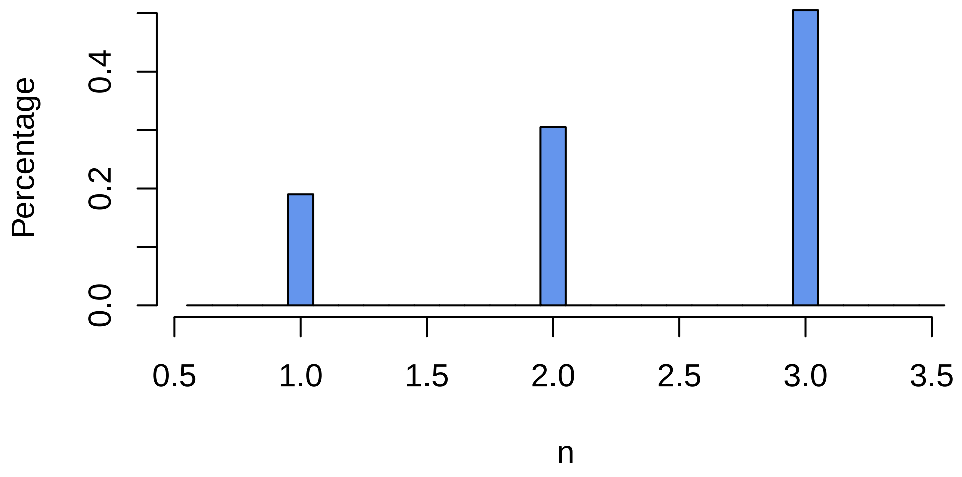
-
-

1.
2.
3.

```r
Discrete.Simulate <- function(F, size, ...) {

  # F - Cumulative distribution function
  # size - number of i.i.d. random variables to be simulated

  m <- 0
  U <- runif(size)
  X <- rep(NA, size)
  X[F(0, ...) >= U] <- 0

  while (any(F(m, ...) < U)) {
    m <- m + 1
    X[(F(m, ...) >= U) & (F(m - 1, ...) < U)] <- m
  }

  return(X)
}
```

```r
# cumulative distribution function

F <- function(x) {
  return(0.2 * ((x >= 1) && (x < 2)) + 0.5 * ((x >= 2) && (x < 3)) + (x >= 3))
}
```

```r
# draw 1000 i.i.d. random variables whose CDF is F

set.seed(10086)
X <- Discrete.Simulate(F, 1000)

# histogram
h <- hist(X, breaks = seq(0.55, 3.55, 0.1), plot = FALSE)
h$density <- h$counts / 1000
plot(h, freq = FALSE, col = "cornflowerblue",
     main = "", xlab = "n", ylab = "Percentage")
```



-

-

```r
# cumulative distribution function of binomial distribution

F <- function(x, n, p) {
  Index <- 0:floor(x)
  Density <- choose(n, Index) * p^(Index) * (1-p)^(n-Index)
  return(sum(Density))
}

# draw 1000 i.i.d. binomial random variables

set.seed(10086)
X <- Discrete.Simulate(F, 1000, 10, 0.5)
```

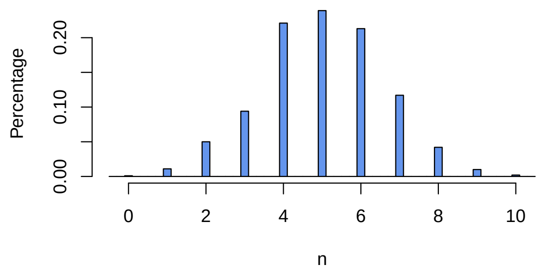
```r
Y <- rbinom(1000, 10, 0.5)
```

```r
h1 <- hist(X, breaks = seq(-0.5, 10.5, 0.2), plot = FALSE)
h1$density <- h1$counts / 1000
plot(h1, freq = FALSE, col = "cornflowerblue",
     main = "", xlab = "n", ylab = "Percentage")
h2 <- hist(Y, breaks = seq(-0.5, 10.5, 0.2), plot = FALSE)
h2$density <- h2$counts / 1000
plot(h2, freq = FALSE, col = "cornflowerblue",
     main = "", xlab = "n", ylab = "Percentage")
```

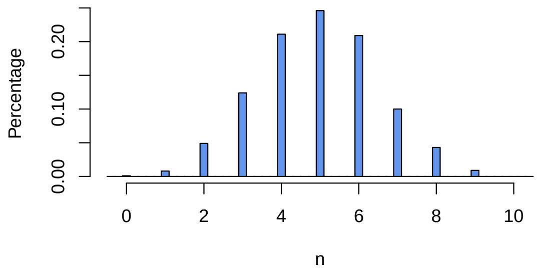
6.3 Inversion

6.3.1 Inversion Method

-



(a) Use Discrete.Simulate

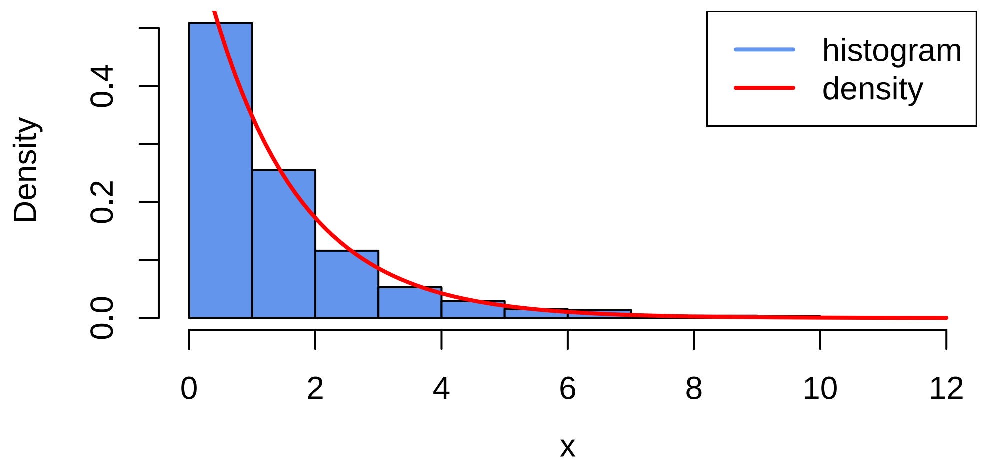


(b) Use rbinom

-
-

```
Exponential.Simulate <- function(lambda, size) {

  # lambda - intensity of the exponential distribution
  # size - number of i.i.d. random variables to be drawn
  U <- runif(size)
  return(-1/lambda * log(U))

}

# A test of the function
set.seed(10086)
X <- Exponential.Simulate(0.7, 1000)
```

```
# 1. Create the Histogram
hist(X,
     freq = FALSE,        # Use density instead of frequency on y-axis
     breaks = 15,         # Adjust number of breaks for desired resolution
     col = "cornflowerblue",
     xlab = "",             # Remove default x-label (add it later with mtext)
     ylab = "Density",
     main = "",
     xlim = c(0, 12))

# 2. Superimpose the Theoretical Density Curve
curve(dexp(x, rate = 0.7),  # Theoretical exponential density function
      from = 0,             # Start of the curve
      to = 12, # End of the curve
      col = "red",
      lwd = 2,            # Adjust line thickness
      add = TRUE)         # Add to the existing plot (histogram)

# 3. Add Labels and Title (using mtext for better control)
mtext("x", side = 1, line = 2.5)  # Add x-axis label below the axis
mtext("Histogram of Exponential Sample and Theoretical Density",
      side = 3, line = 1, outer = TRUE) # Title at the top margin

# 4. Add a Legend
legend("topright",                    # Position of the legend
       legend = c("histogram", "density"), # Text for the legend
       col = c("cornflowerblue", "red"),   # Colors of the legend elements
       lwd = 2)                          # Line width for the legend
```



6.3.2 Small Exercise

-
-
  -
  -
  -

```r
Gumbel.Simulate <- function(min, max, size) {
  U <- runif(size, min, max)
  return(-log(-log(U)))
}

set.seed(1234)

X <- Gumbel.Simulate(0, 1, 100)
X[1:4]
#> [1] -0.7766432  0.7458437  0.7022162  0.7495061
```

6.4 Acceptance-Rejection

-

-
-

1.

2.

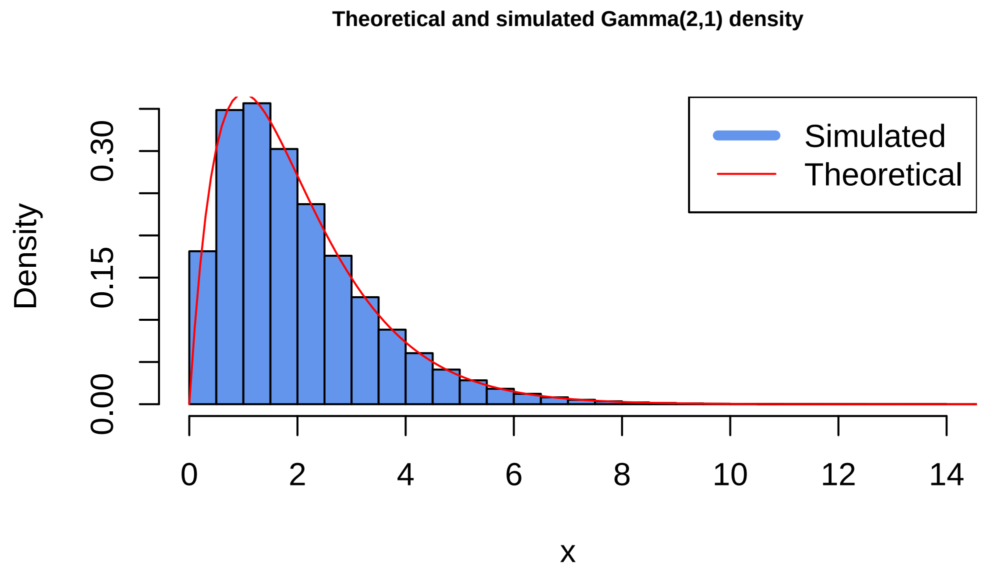
3.

[1]

[1]Nelder, J. A. and Mead, R. (1965) A simplex algorithm for function minimization. Computer Journal 7, 308-313.

```
### Auxiliary functions ###

gamma.pdf <- function(x, k, theta) {
  return(theta^k * x^(k-1) * exp(-theta*x)/gamma(k))
}

exponential.pdf <- function(x, lambda) {
  return(lambda * exp(-lambda*x))
}

Exponential.Simulate <- function(lambda, size = 1) {
  V <- runif(size)
  return(-1/lambda * log(V))
}

Gamma.Simulate <- function(k, theta, size = 1) {

  lambda <- theta/k
  c <- k^k * exp(-k+1) / gamma(k)

  U <- rep(NA, size)
  Y <- rep(NA, size)
  X <- rep(NA, size)
  Unaccepted <- rep(TRUE, size)

  while (any(Unaccepted)) {

    UnacceptedCount <- sum(Unaccepted)

    U <- runif(UnacceptedCount)
    Y <- Exponential.Simulate(lambda, UnacceptedCount)
    Accepted_ThisTime <- Unaccepted[Unaccepted] &
      ( U <= ( gamma.pdf(Y, k, theta) / exponential.pdf(Y, lambda)/c ) )

    X[Unaccepted][Accepted_ThisTime] <- Y[Accepted_ThisTime]
    Unaccepted[Unaccepted] <- !Accepted_ThisTime

  }

  return(X)

}
```

```
set.seed(10086)

X <- Gamma.Simulate(2, 1, 10^6)

# histogram

hist(X, breaks = 30, freq = FALSE,
     main = "Theoretical and simulated Gamma(2,1) density",
     col = "cornflowerblue",
     xlim = c(0, 14), ylim = c(0, 0.35),
     xlab = "x",
     cex.main = 0.7)

xticks = seq(0, max(X), 0.1)
lines(xticks, dgamma(xticks, 2, 1), col = "red")
legend("topright", legend = c("Simulated", "Theoretical"), lty = c(1, 1), lwd = c(5, 1), col
```



6.5 Simulate Normals

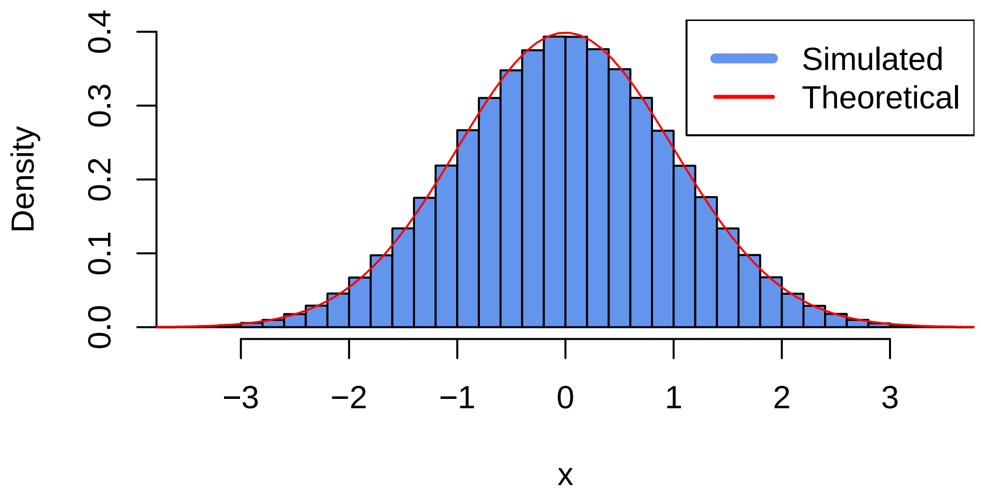
-
-
-

6.5.1 Simulate Normals: Using uniformly distributed random variable

```r
### Simulate standard normal random variable ###

Normal.Simulate <- function(size = 1) {

  Z <- rep(NA, size)

  for (i in 1:size) {
    Z[i] <- sum(runif(12)) - 6
  }
  return(Z)
}
```

```r
set.seed(10086)

X <- Normal.Simulate(10^6)

# histogram

hist(X, breaks = 35, freq = FALSE,
     main = "Theoretical and simulated N(0,1) density",
     col = "cornflowerblue",
     xlim = c(-3.5, 3.5), ylim = c(0, 0.4),
     xlab = "x",
     cex.main = 0.7)

xticks = seq(min(X), max(X), 0.1)
lines(xticks, dnorm(xticks, 0, 1), col = "red")
legend("topright", legend = c("Simulated", "Theoretical"), lty = c(1, 1), lwd = c(5, 2), col
```

Theoretical and simulated N(0,1) density



6.5.2 Simulate Normals: Acceptance-Rejection with Exponential Envelope

1.
2.
3.

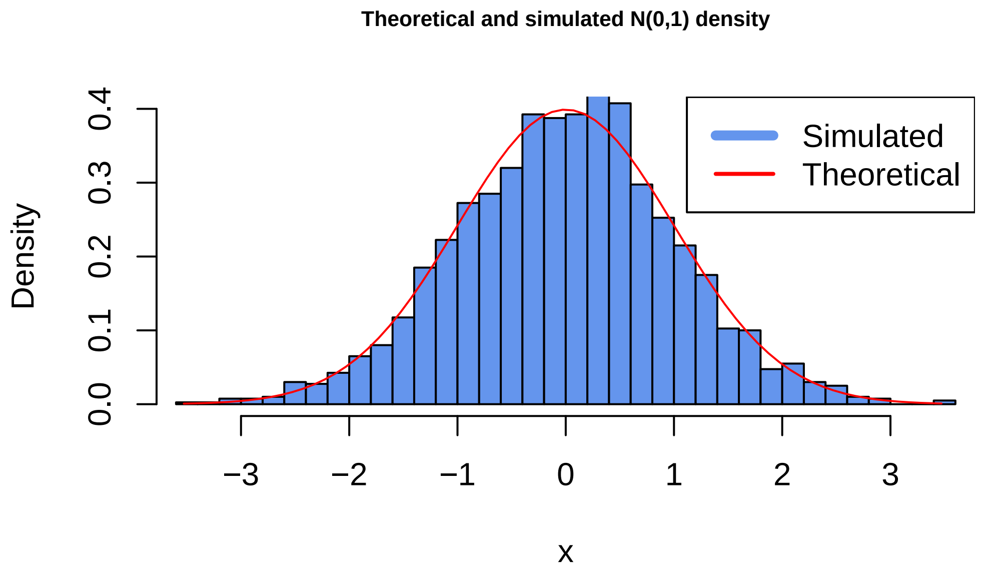
6.5.3 Simulate Normals: Box-Muller Algorithm

Theorem 6.1 (Box-Muller transform)**.**

1.
2.
3.

```r
BoxMuller <- function(size = 1) {

  # size: number of standard bivariate normal random variables to simulate

  U <- runif(size)
  V <- runif(size)

  X <- sqrt(-2*log(U)) * cos(2*pi*V)
  Y <- sqrt(-2*log(U)) * sin(2*pi*V)

  return(c(X,Y))

}
```

```r
set.seed(10086)

X <- BoxMuller(1000)

# histogram

hist(X, breaks = 35, freq = FALSE,
     main = "Theoretical and simulated N(0,1) density",
     col = "cornflowerblue",
     xlim = c(-3.5, 3.5), ylim = c(0, 0.4),
     xlab = "x",
     cex.main = 0.7)

xticks = seq(min(X), max(X), 0.1)
lines(xticks, dnorm(xticks, 0, 1), col = "red")
legend("topright", legend = c("Simulated", "Theoretical"), lty = c(1, 1), lwd = c(5, 2), col
```



6.6 Monte Carlo Integration

6.6.1 Monte Carlo Integration - a brief introduction

6.6.2 Monte Carlo Integration - the code

```r
MonteCarlo.Integration <- function(f, n, a, b) {

  U <- runif(n, min = a, max = b)
  return( (b-a)*mean(f(U)) )

}
```

6.6.3 Monte Carlo Integration - examples

```r
set.seed(10086)
MonteCarlo.Integration(function(x) 1/(1+x^2), 100, -1, 1)
#> [1] 1.589363
```

```r
set.seed(10086)
MonteCarlo.Integration(function(x) atan(x), 100, 0, 1)
#> [1] 0.445995
```

6.6.4 Small Exercise

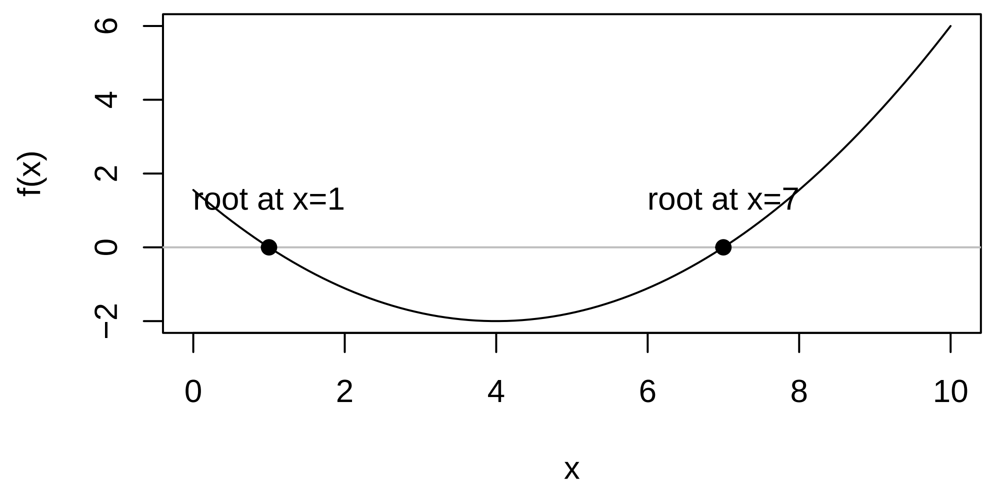
-
-
-

-
-

```r
vU <- runif(10000)
vV <- runif(10000)

fLessThanOne <- function(vInput) {
  return(vInput < 1)
}

dResult <- (1 - 0) * mean(fLessThanOne(vU^2 + vV^2))
dResult * 4
#> [1] 3.1716
```

7 Root-finding

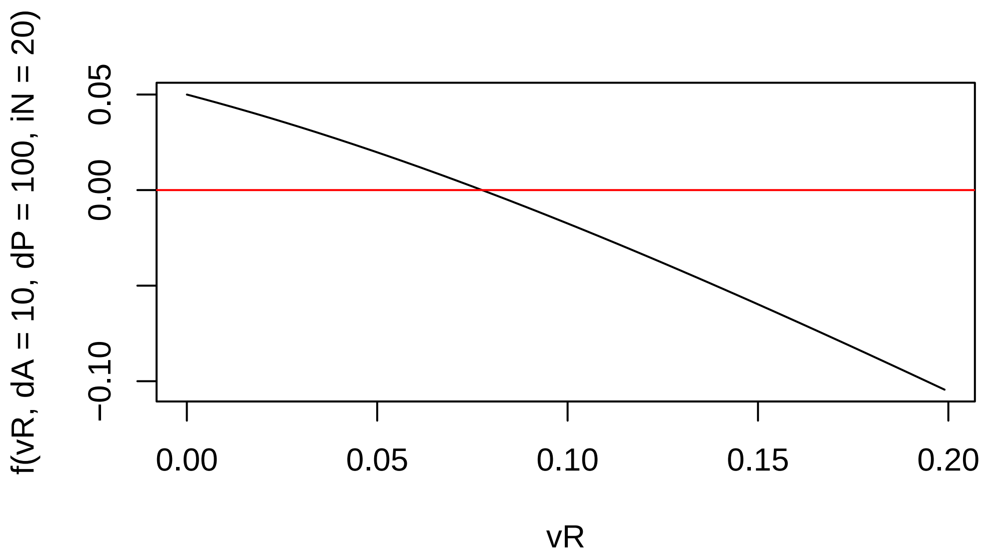
7.1 Main problem



7.1.1 The Roots of a Function

7.1.2 Example: Loan repayments

```r
f <- function(dR, dA, dP, iN) {
  dOut = dA/dP - (dR * (1.0 + dR)^iN)/((1.0 + dR)^iN - 1)
  return(dOut)
}

vR <- seq(1e-5, 0.2, 0.001)

plot(vR, f(vR, dA = 10, dP = 100, iN = 20), type = "l")
abline(h = 0, col = "red")
```



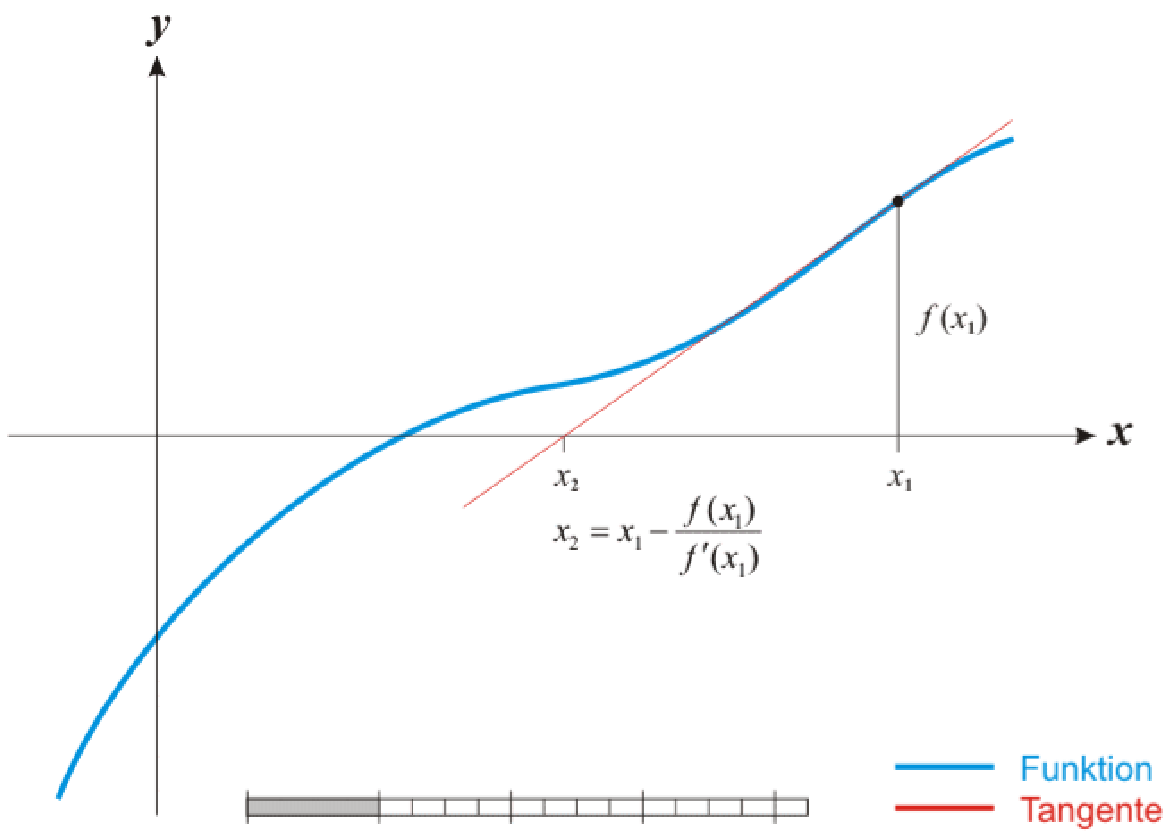
-
-
-
-

7.2 Numerical Methods for Root-finding

7.3 The Newton-Raphson Method

-
-
-
-
-
-

•



7.3.1 Notes for the Newton-Raphson method

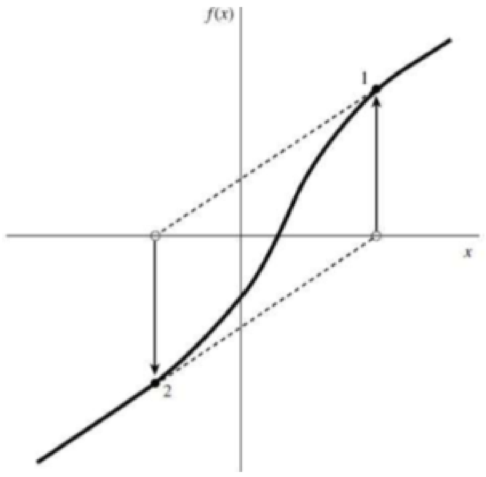
•

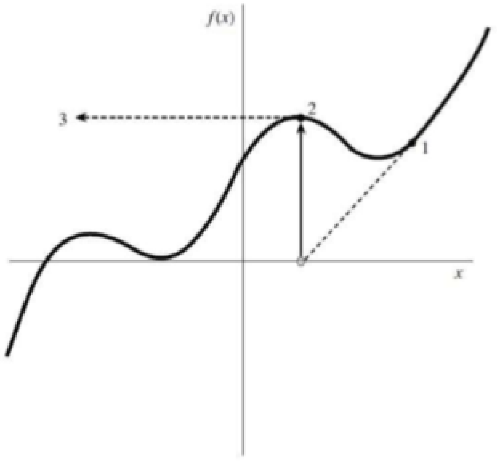
7.3.2 The Newton-Raphson algorithm

-
-
-
-
  -
  -

7.3.3 Drawbacks of the Newton-Raphson method

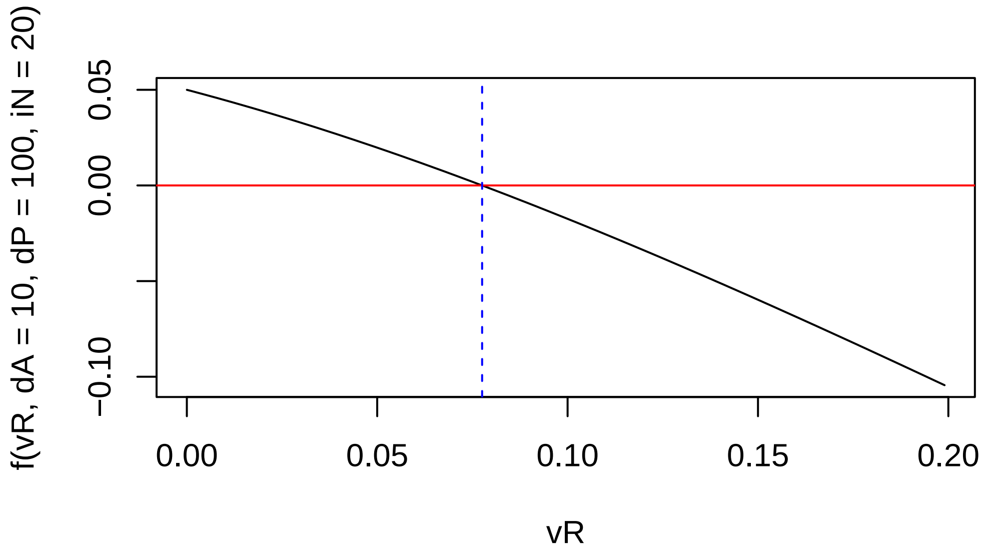
-





7.3.4 Loan example

```r
f <- function(dX, dA, dP, iN) {
  dOut = dA/dP - (dX * (1.0 + dX)^iN)/((1.0 + dX)^iN - 1)
  return(dOut)
}

f_prime <- function(dX, dA, dP, iN) {
  dNum1 = ((1.0 + dX)^iN + iN * dX * (1.0 + dX)^(iN - 1)) * ((1.0 + dX)^iN - 1)
  dNum2 = iN * dX * (1.0 + dX)^(2 * iN - 1)
  dDen = ((1.0 + dX)^iN - 1)^2
  dOut = -(dNum1 - dNum2)/dDen
  return(dOut)
}

NR <- function(f, f_prime, dX0, dTol = 1e-9, max.iter = 1000, ...) {
  dX <- dX0
  fx <- f(dX, ...)
  iter <- 0
  while ((abs(fx) > dTol) && (iter < max.iter)) {
    dX <- dX - f(dX, ...)/f_prime(dX, ...)
    fx <- f(dX, ...)
    iter <- iter + 1
    cat("At iteration ", iter, "value of x is: ", dX, "\n")
  }
  if (abs(fx) > dTol) {
    cat("Algorithm failed to converge\n")
    return(NULL)
  } else {
    cat("Algorithm converged\n")
    return(dX)
  }
}

NR(f, f_prime, dX0 = 0.15, dA = 10, dP = 100, iN = 20)
#> At iteration  1 value of x is:  0.08241672
#> At iteration  2 value of x is:  0.07758319
#> At iteration  3 value of x is:  0.0775469
#> At iteration  4 value of x is:  0.0775469
#> Algorithm converged
#> [1] 0.0775469
```



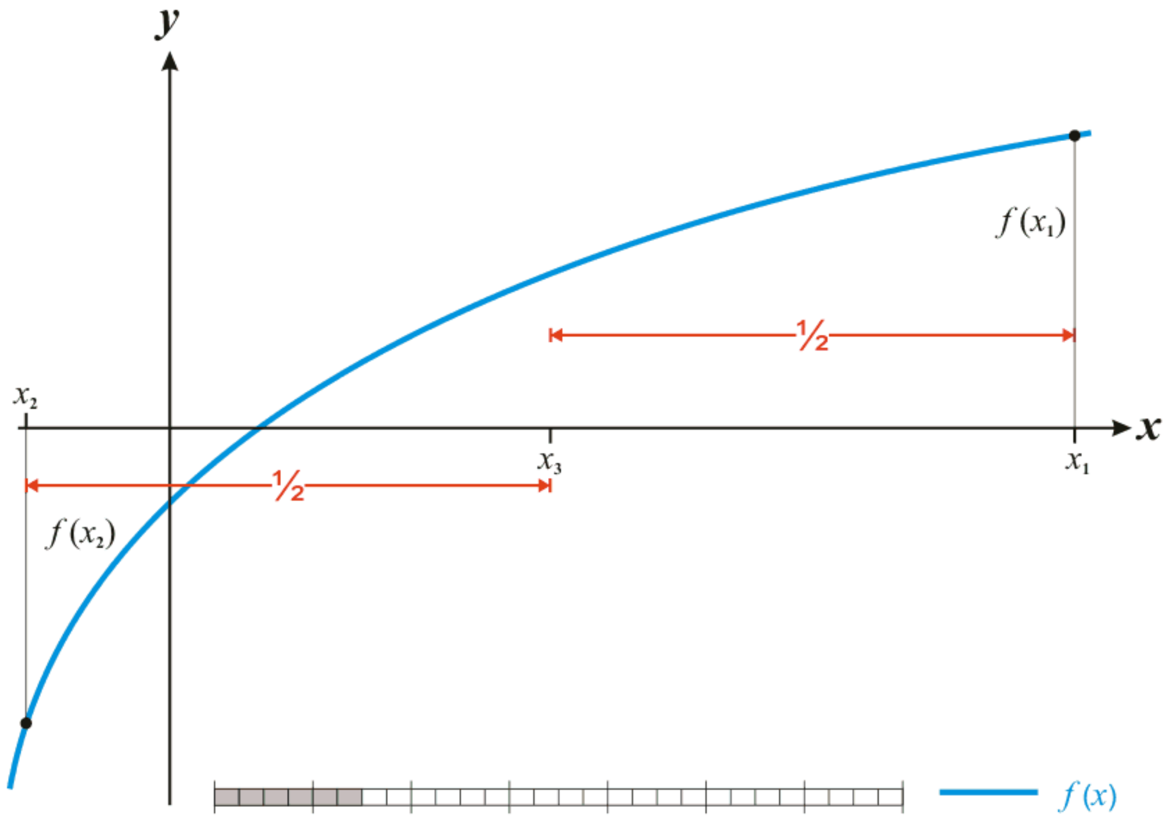
7.4 The Secant method

-

-
-

7.5 The bisection method

0.
1.
2.
3.
4.



7.5.1 The bisection method: Implementation in R

```r
bisection <- function(f, dX.l, dX.r, dTol = 1e-9, max.iter = 1000, ...) {

  #check inputs
  if (dX.l >= dX.r) {
    cat("error: x.l >= x.r \n")
    return(NULL)
  }
  f.l <- f(dX.l, ...)
  f.r <- f(dX.r, ...)
  if (f.l == 0) {
    return(dX.l)
  } else if (f.r == 0) {
    return(dX.r)
  } else if (f.l*f.r > 0) {
    cat("error: f(x.l)*f(x.r) > 0 \n")
    return(NULL)
  }

  # successively refine x.l and x.r
  iter <- 0
  while ((dX.r - dX.l) > dTol && (iter < max.iter)) {
    dX.m <- (dX.l + dX.r)/2
    f.m <- f(dX.m, ...)
    if (f.m == 0) {
      return(dX.m)
    } else if (f.l*f.m < 0) {
      dX.r <- dX.m
      f.r <- f.m
    } else {
      dX.l <- dX.m
      f.l <- f.m
    }
    iter <- iter + 1
    cat("at iteration", iter, "the root lies between", dX.l, "and", dX.r, "\n")
  }

  # return approximate root
  return((dX.l + dX.r)/2)
}
```

```
bisection(f, dX.l = 1e-6 , dX.r = 0.2, dTol = 1e-5, dA = 10, dP = 100, iN = 20)
#> at iteration 1 the root lies between 1e-06 and 0.1000005
#> at iteration 2 the root lies between 0.05000075 and 0.1000005
#> at iteration 3 the root lies between 0.07500063 and 0.1000005
#> at iteration 4 the root lies between 0.07500063 and 0.08750056
#> at iteration 5 the root lies between 0.07500063 and 0.08125059
#> at iteration 6 the root lies between 0.07500063 and 0.07812561
#> at iteration 7 the root lies between 0.07656312 and 0.07812561
#> at iteration 8 the root lies between 0.07734436 and 0.07812561
#> at iteration 9 the root lies between 0.07734436 and 0.07773499
#> at iteration 10 the root lies between 0.07753967 and 0.07773499
#> at iteration 11 the root lies between 0.07753967 and 0.07763733
#> at iteration 12 the root lies between 0.07753967 and 0.0775885
#> at iteration 13 the root lies between 0.07753967 and 0.07756409
#> at iteration 14 the root lies between 0.07753967 and 0.07755188
#> at iteration 15 the root lies between 0.07754578 and 0.07755188
#> [1] 0.07754883
```

7.6 The `uniroot`function

[1]

[1]Nelder, J. A. and Mead, R. (1965) A simplex algorithm for function minimization. Computer Journal 7, 308-313.

```
uniroot(f, interval, ..., lower = min(interval), upper = max(interval), tol = .Machine$doubl
```

7.6.1 Example for the loan repayment function

```
uniroot(f, lower = 1e-6, upper = 0.2, dA = 10, dP = 100, iN = 20)
#> $root
#> [1] 0.07752483
#>
#> $f.root
#> [1] 1.659286e-05
#>
#> $iter
#> [1] 4
#>
#> $init.it
#> [1] NA
#>
#> $estim.prec
#> [1] 6.103516e-05
```

7.7 Numerical Derivatives in R

[2]

```
grad(func, x, method = "Richardson", side = NULL, method.args=list(), ...)
```

[2]Byrd, R. H., Lu, P., Nocedal, J. and Zhu, C. (1995) A limited memory algorithm for bound constrained optimization. SIAM J. Scientific Computing, 16, 1190-1208.

-
-
-

```
grad(f, 0.2, dA = 10, dP = 100, iN = 20)
#> [1] -0.9351161

f_prime(0.2, dA = 10, dP = 100, iN = 20)
#> [1] -0.9351161

abs(grad(f, 0.2, dA = 10, dP = 100, iN = 20) - f_prime(0.2, dA = 10, dP = 100, iN = 20))
#> [1] 1.055467e-11
```

7.7.1 Warnings

-
-

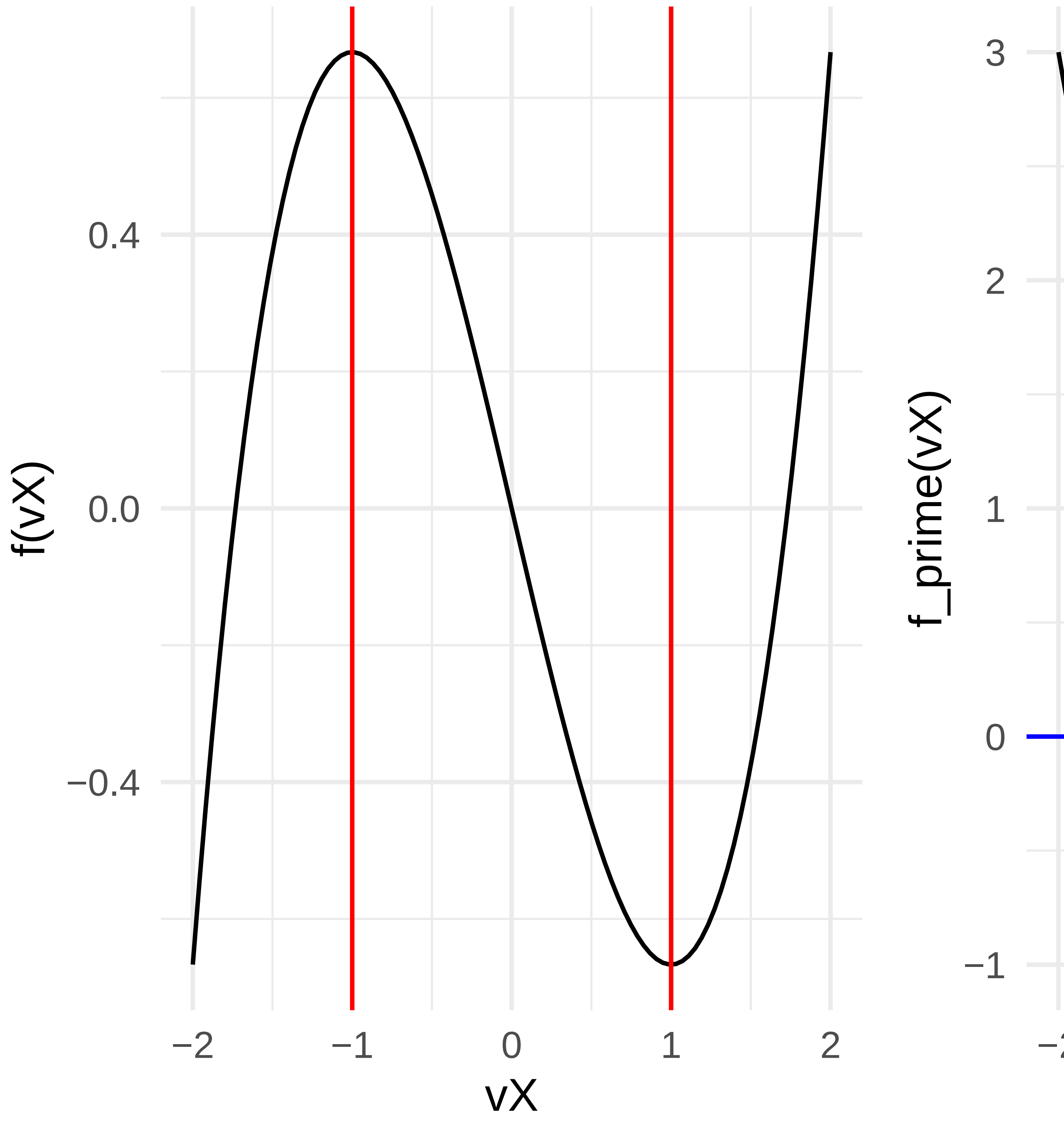
```
library(microbenchmark)
#> Warning: pakke 'microbenchmark' blev bygget under R version 4.3.3
microbenchmark(
  grad(f, 0.2, dA = 10, dP = 100, iN = 20),
  f_prime(0.2, dA = 10, dP = 100, iN = 20)
)
#> Unit: microseconds
#>                                         expr  min   lq   mean median   uq   max
#>   grad(f, 0.2, dA = 10, dP = 100, iN = 20) 36.6 43.2 63.788   46.4 54.7 528.1
#>   f_prime(0.2, dA = 10, dP = 100, iN = 20)  1.0  1.4  2.422    1.7  2.2  37.8
#>  neval
#>    100
#>    100
```

8 Numerical Optimization 1

8.1 Numerical vs Analytical Optimization

-
-
-
-

8.1.1 Optimization and Root-finding



0.4
0.0
−0.4
f(vX)
−2
−1
0
1
2
vX
3
2
1
0
−1
f_prime(vX)

8.1.2 Maximum: Definition

8.1.3 Numerical optimization: local search techniques

8.1.4 Numerical optimization: general structure

1.
2.
3.
4.
5.

8.1.5 Stopping criteria

-
-

1.
2.
3.
4.

8.2 The Newton-Raphson Method - and Method for Optimization.

-
-
-
-
  -
  -

8.2.1 Newton's Method for Optimization

8.2.2 Newton's algorithm convergence

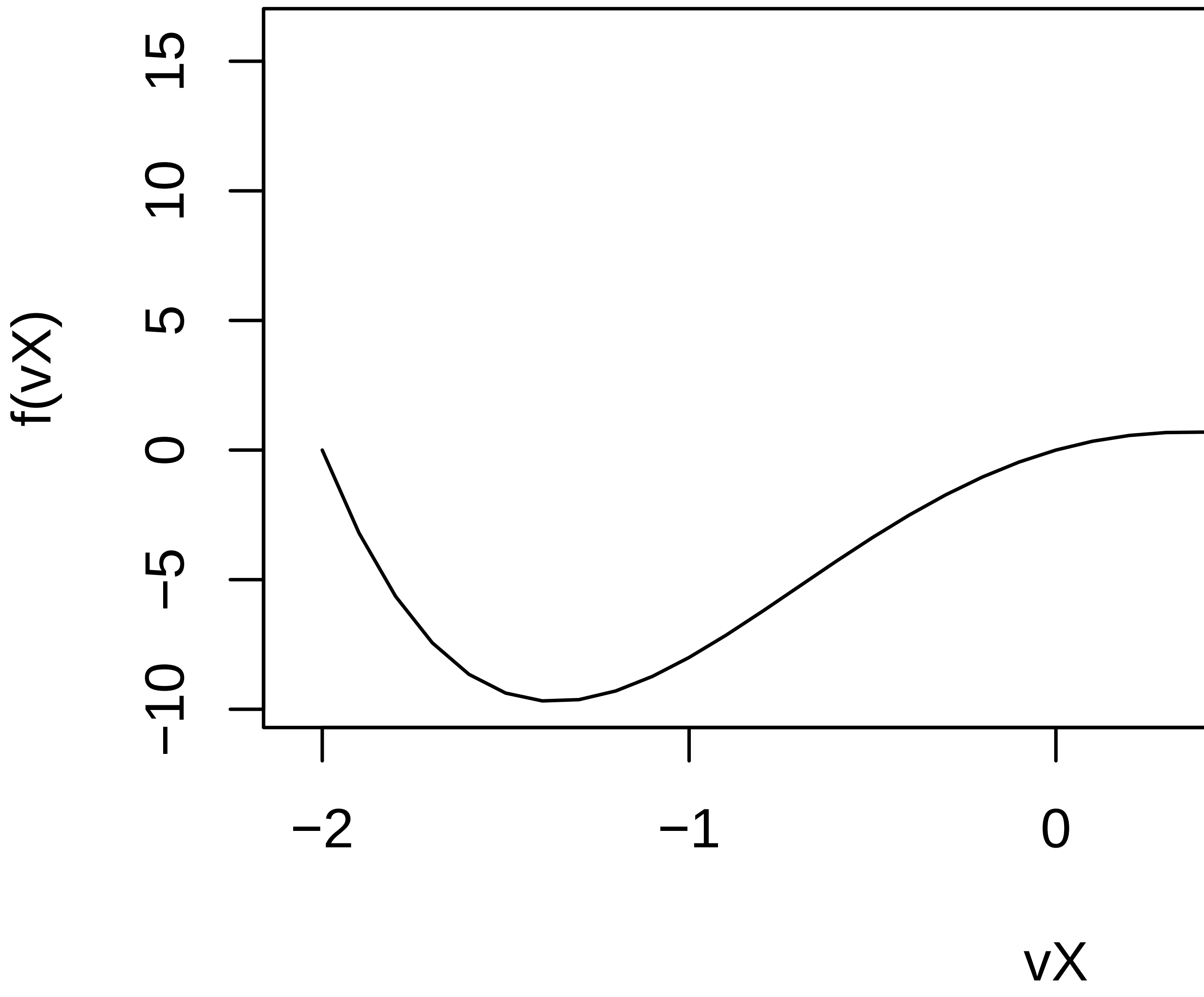
-
-
-
-

8.2.3 Newton's method: Example

```r
f <- function(dX) {
  dOut <- 2 * dX * (dX - 1)^2 * (dX + 2)
  return(dOut)
}
```

```r
f_prime <- function(dX) {
  dOut <- 4 - 12 * dX + 8 * dX^3
  return(dOut)
}
```

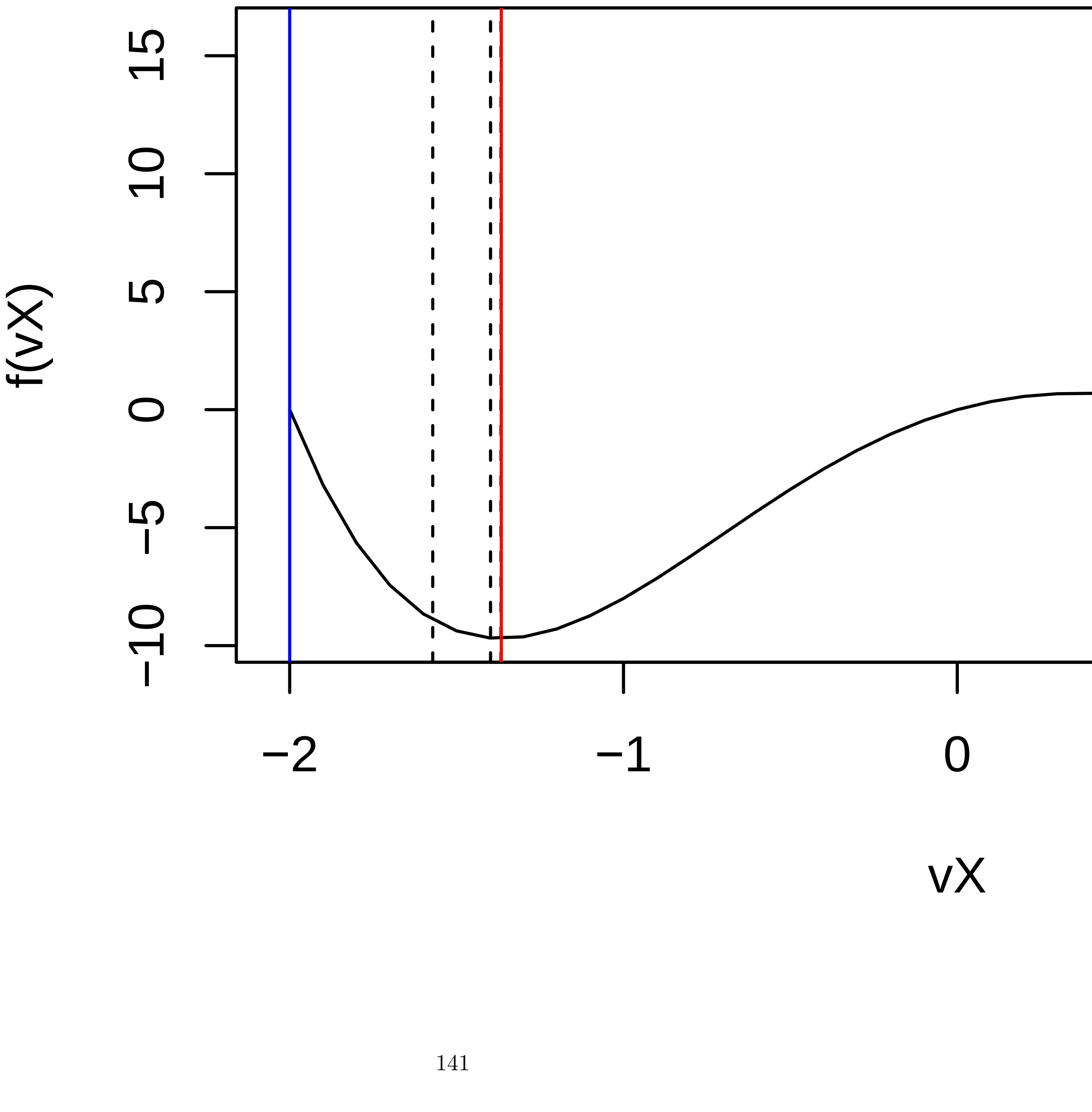
```r
f_second <- function(dX) {
  dOut <- 24 * dX^2 - 12
  return(dOut)
}
vX <- seq(-2, 2, 0.1)
plot(vX, f(vX), type = "l")
```

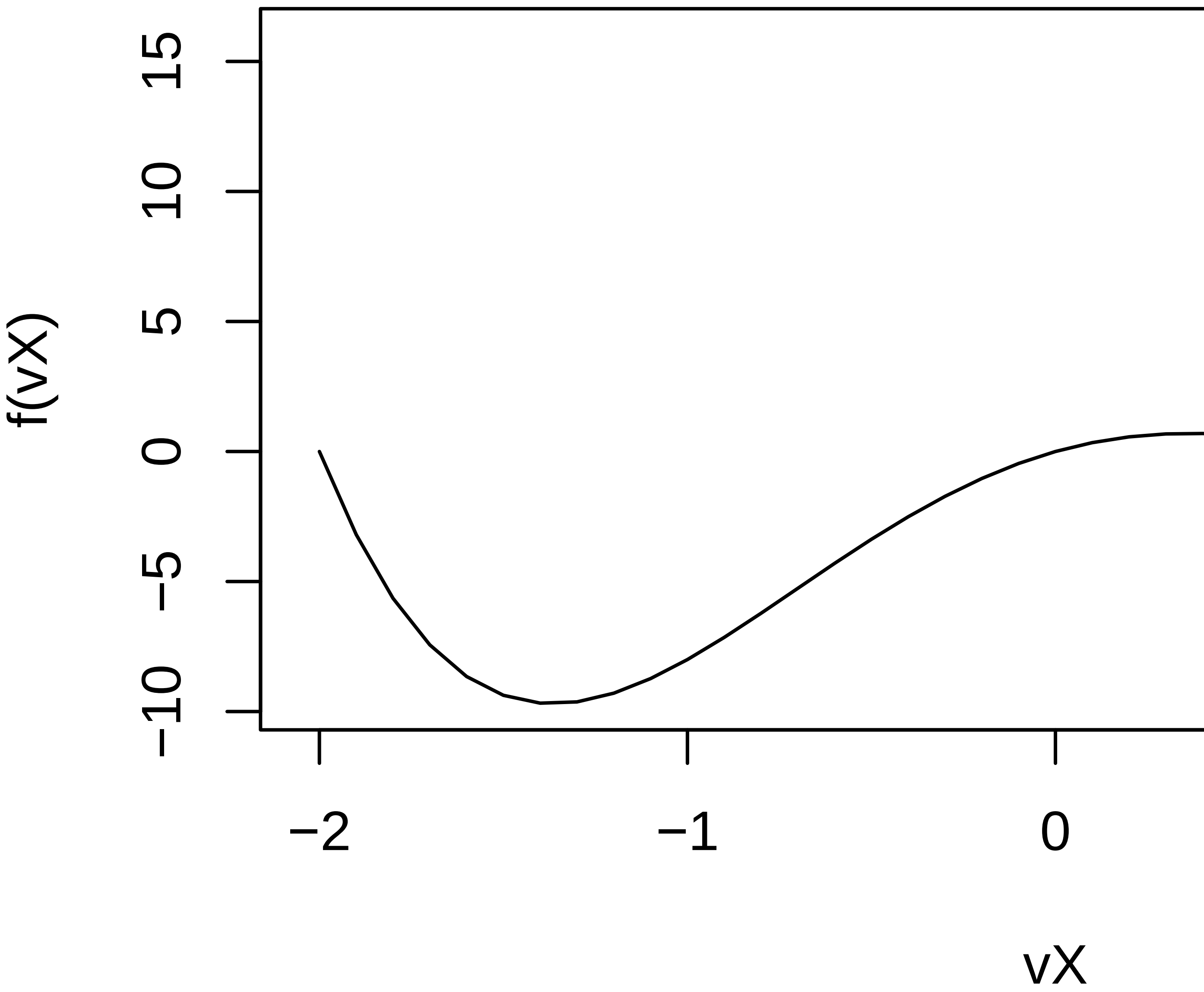

15
10
5
0
−5
−10
f(vX)
−2
−1
0
vX

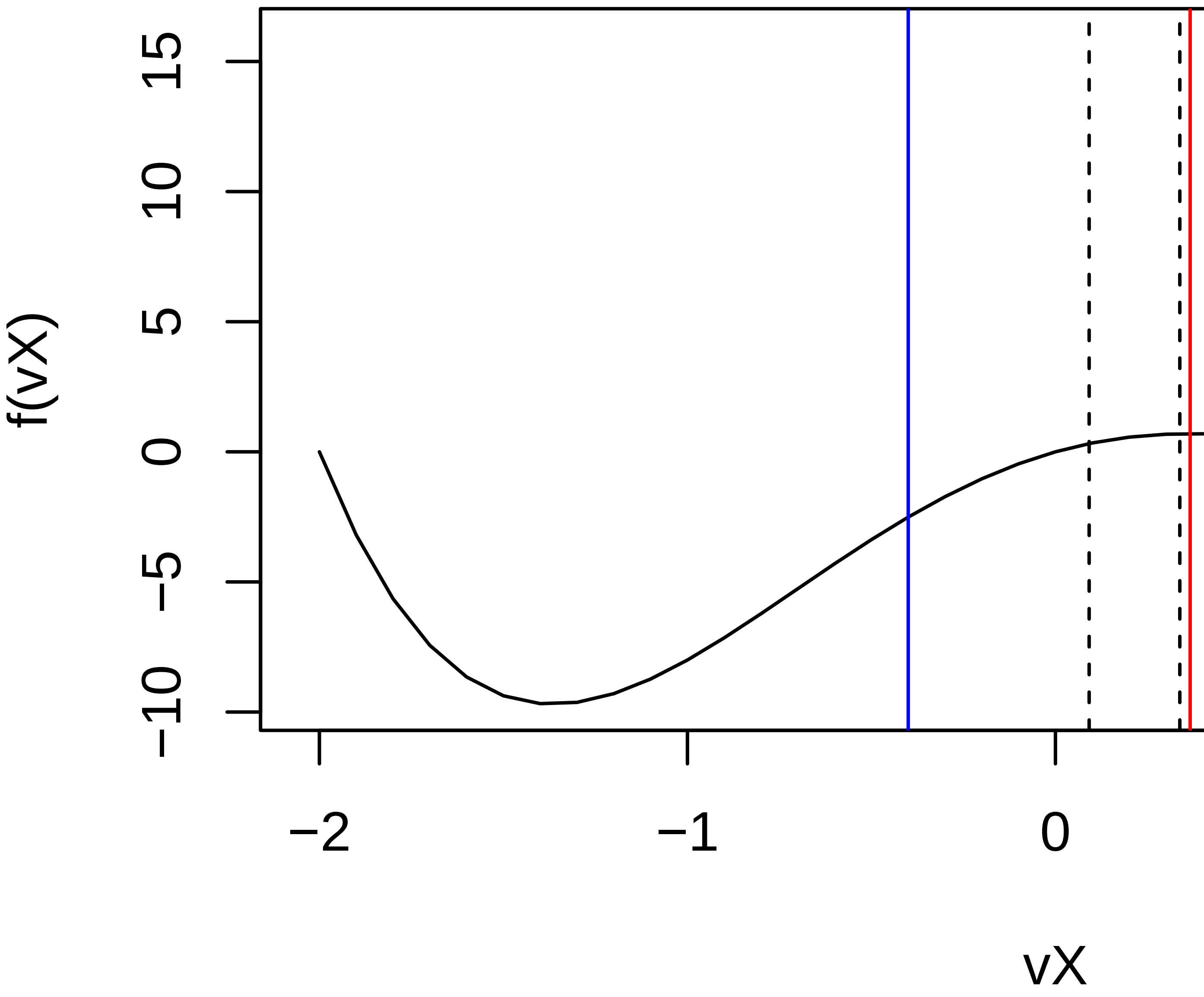
8.2.4 Newton's method: implementation in R

```r
NM <- function(f, f_prime, f_sec, dX0, dTol = 1e-9, n.max = 1000){
  dX <- dX0
  fx <- f(dX)
  fpx <- f_prime(dX)
  fsx <- f_sec(dX)
  n <- 0
  while ((abs(fpx) > dTol) && (n < n.max)) {
    dX <- dX - fpx/fsx
    fx <- f(dX)
    fpx <- f_prime(dX)
    fsx <- f_sec(dX)
    n <- n + 1
    cat("At iteration", n, "the value of x is:", dX, "\n")
  }
  if (n == n.max) {
    cat('newton failed to converge\n')
  } else {
    return(dX)
  }
}
```

```r
x1 <- NM(f, f_prime, f_second, dX0 = -2)
#> At iteration 1 the value of x is: -1.571429
#> At iteration 2 the value of x is: -1.398224
#> At iteration 3 the value of x is: -1.367014
#> At iteration 4 the value of x is: -1.366026
#> At iteration 5 the value of x is: -1.366025
vX <- seq(-2, 2, 0.1)
plot(vX, f(vX), type = "l")
abline(v=-2, col="blue") # initial guess
abline(v=-1.571429, lty=2) # first iteration
abline(v=-1.398224, lty=2) # second iteration
abline(v=-1.367014 , lty=2) # third iteration
abline(v=x1, col="red") # solution
```


f(vX)
15
10
5
0
−5
−10
−2
−1
0
vX

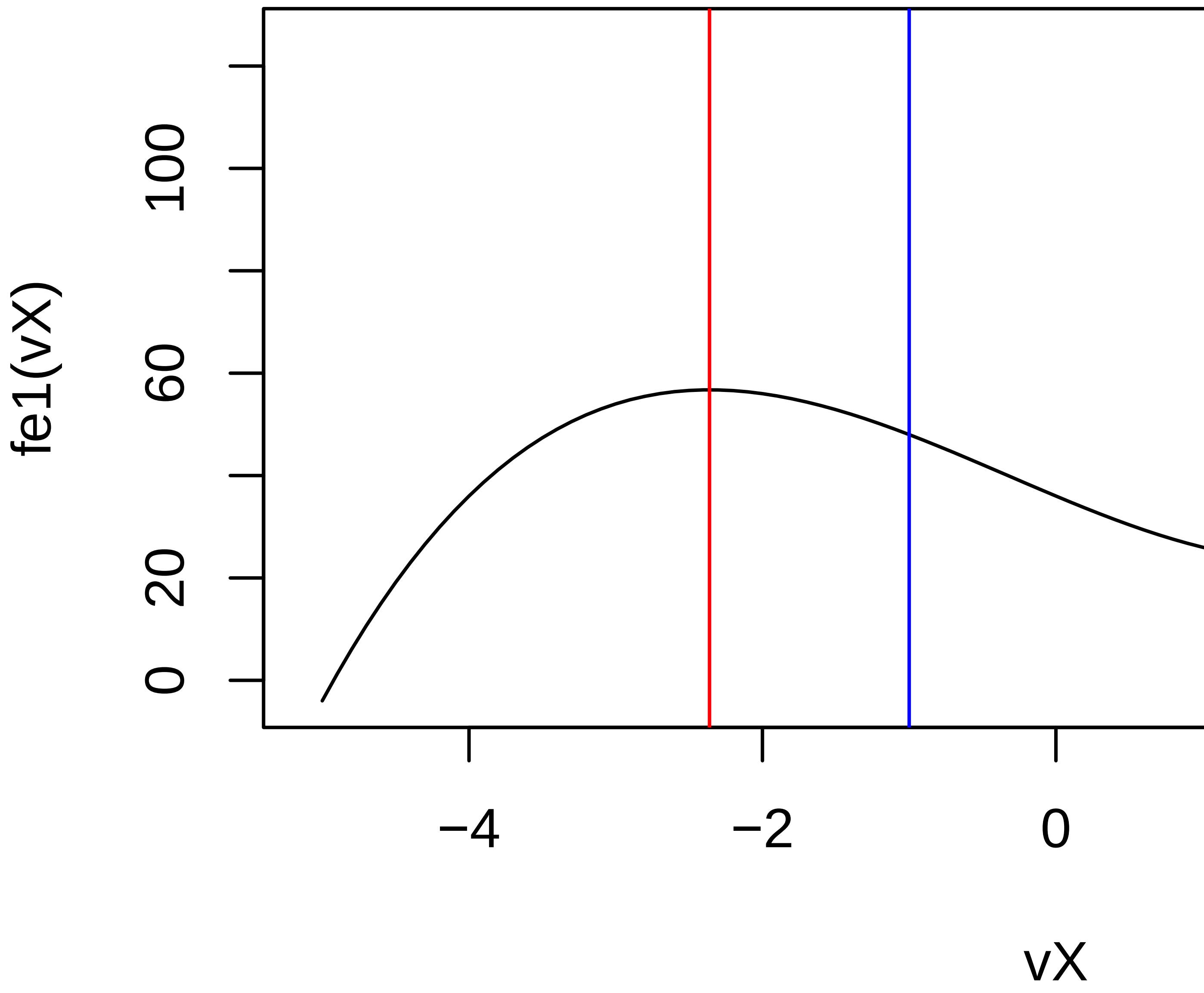

15
10
5
0
−5
−10
f(vX)
−2
−1
0
vX

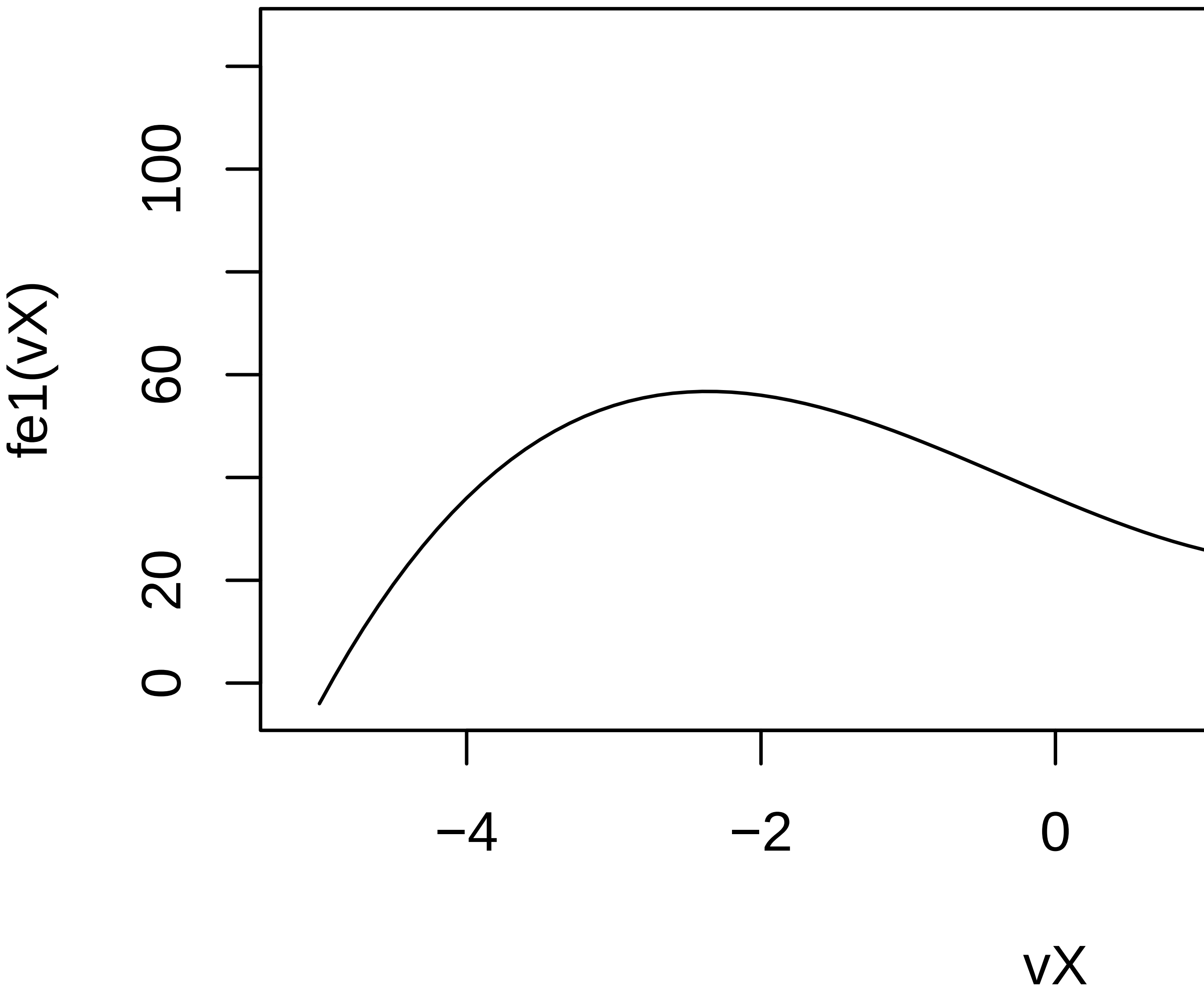


f(vX)
15
10
5
0
−5
−10
−2
−1
0
vX

8.2.5 Exercise

```r
# Functions
fe1 <- function(dX) {
  dOut <- dX^3 + (6-dX)^2
  return(dOut)
}
feprime <- function(dX) {
  dOut <- 3 * dX^2 - 2*(6-dX)
  return(dOut)
}
fesecond <- function(dX) {
  dOut <- 6 * dX + 2
  return(dOut)
}
#Solution
xe1 <- NM(fe1, feprime, fesecond, dX0 = -1) # minimum
vX <- seq(-5, 5, 0.1)
plot(vX, fe1(vX), type = "l")
abline(v=-1, col="blue") # initial guess
abline(v=xe1, col="red") # solution

xe2 <- NM(fe1, feprime, fesecond, dX0 = 4) # maximum
vX <- seq(-5, 5, 0.1)
plot(vX, fe1(vX), type = "l")
abline(v=4, col="blue") # initial guess
abline(v=xe2, col="red") # solution
```


fe1(vX)
0
20
60
100
−4
−2
0
vX



fe1(vX)
0
20
60
100
−4
−2
0
vX

8.3 The Golden-section method

8.3.1 The idea of the Golden-section method

0)

1)

2)

-

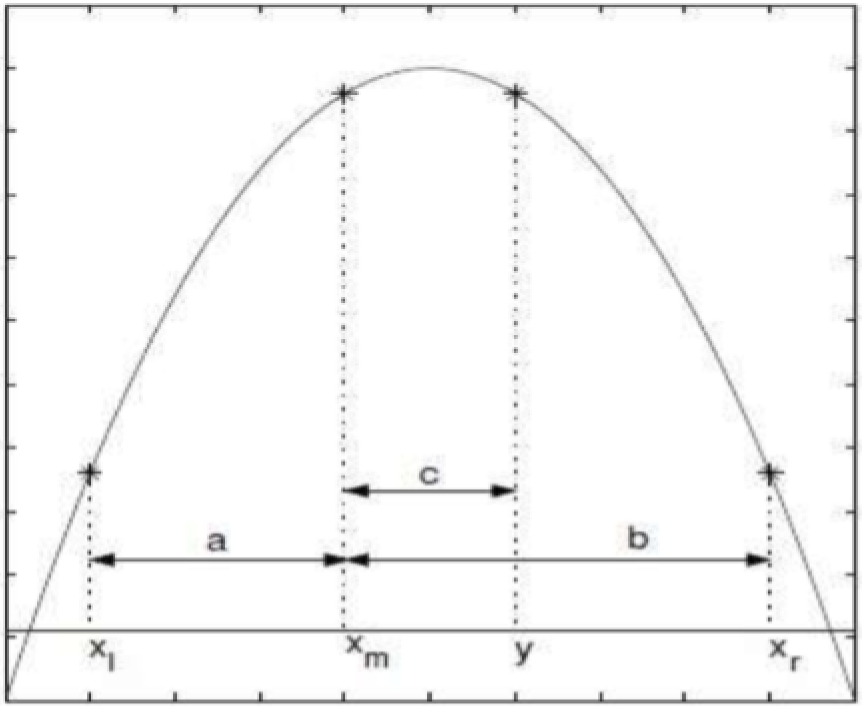
-

-

-

3)

8.3.2 The Golden ratio



8.4 New algorithm

0.
1.
2.

i)

ii)

3.

8.4.1 Notes

8.4.2 The golden-section method: R implementation

```r
gsection <- function(f, dX.l, dX.r, dX.m, dTol = 1e-9) {

  # golden ratio plus one
  dGR1 <- 1 + (1 + sqrt(5))/2

  # successively refine x.l, x.r, and x.m
  f.l <- f(dX.l)
  f.r <- f(dX.r)
  f.m <- f(dX.m)
  while ((dX.r - dX.l) > dTol) {
    if ((dX.r - dX.m) > (dX.m - dX.l)) { # if the right segment is wider than the left
      dY <- dX.m + (dX.r - dX.m)/dGR1 # put Y into the right segment according to the golden
      f.y <- f(dY)
      if (f.y >= f.m) {
        dX.l <- dX.m
        f.l <- f.m
        dX.m <- dY
        f.m <- f.y
      } else {
        dX.r <- dY
        f.r <- f.y
      }
    } else { #if the left segment is wider than the right
      dY <- dX.m - (dX.m - dX.l)/dGR1 # put Y into the left segment according to the golden
      f.y <- f(dY)
      if (f.y >= f.m) {
        dX.r <- dX.m
        f.r <- f.m
        dX.m <- dY
        f.m <- f.y
      } else {
        dX.l <- dY
        f.l <- f.y
      }
    }
  }
  return(dX.m)
}
gsection(f, dX.l = -1, dX.r = 1, dX.m = 0.5)
#> [1] 0.3660254
```

8.5 Optimization in R: univariate

```
optimize(f, interval, ..., lower = min(interval), upper = max(interval), maximum = FALSE, tol
```

```
optimize(f, lower = -0.4, upper = 0.8, maximum = TRUE)
#> $maximum
#> [1] 0.3660215
#>
#> $objective
#> [1] 0.6961524
```

```
# Using built-in optimizer
optimize(fe1, lower = -5, upper = 5, maximum = TRUE) # maximum
#> $maximum
#> [1] -2.360919
#>
#> $objective
#> [1] 56.74535

optimize(fe1, lower = -5, upper = 5) # minimum
#> $minimum
#> [1] 1.694255
#>
#> $objective
#> [1] 23.4028

f2 <- function(x){-1*fe1(x)}
optimize(f2, lower = -5, upper = 5)
#> $minimum
#> [1] -2.360919
#>
#> $objective
#> [1] -56.74535
```

9 Numerical Optimization 2

9.1 Multivariate Optimization

-
-
-
-

9.2 Steepest ascent method

9.2.1 Steepest ascent method: Implementation in R

```r
ascent <- function(f, grad.f, vX0, dTol = 1e-9, n.max = 100) {
  vX.old <- vX0
  vX <- line.search(f, vX0, grad.f(vX0))
  n <- 1
  while ((f(vX) - f(vX.old) > dTol) & (n < n.max)) {
    vX.old <- vX
    vX <- line.search(f, vX, grad.f(vX))
    cat("at iteration", n, "the coordinates of x are", vX, "\n")
    n <- n + 1
  }
  return(vX)
}
```

9.2.2 Line search

9.2.3 Line-search: Implementation in R

```
line.search <- function(f, vX, vG, dTol = 1e-9, dA.max = 2^5) {
  # f is a real function that takes a vector of length d
  # x and y are vectors of length d
  # line.search uses gsection to find a >= 0 such that
  # g(a) = f(x + a*y) has a local maximum at a,
  # within a tolerance of tol
  # if no local max is found then we use 0 or a.max for a
  # the value returned is x + a*y
  if (sum(abs(vG)) == 0){
    return(vX) # +0*vG
```

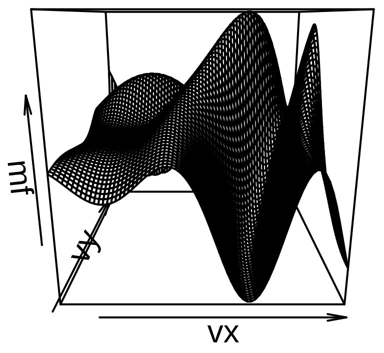
```r
  } # g(a) constant
  g <- function(dA){
    return(f(vX + dA*vG))
  }
  # find a triple a.l < a.m < a.r such that
  # g(a.l) <= g(a.m) and g(a.m) >= g(a.r)

  # choose a.l
  dA.l <- 0
  g.l <- g(dA.l)
  # find a.m
  dA.m <- 1
  g.m <- g(dA.m)
  while ((g.m < g.l) & (dA.m > dTol)) {
    dA.m <- dA.m/2
    g.m <- g(dA.m)
  }
  # if a suitable a.m was not found then use 0 for a, so just return vX as the next step
  if ((dA.m <= dTol) & (g.m < g.l)){
    return(vX)
  }
  # find a.r
  dA.r <- 2*dA.m
  g.r <- g(dA.r)
  while ((g.m < g.r) & (dA.r < dA.max)) {
    dA.m <- dA.r
    g.m <- g.r
    dA.r <- 2*dA.m
    g.r <- g(dA.r)
  }
  # if a suitable a.r was not found then use a.max for a
  if ((dA.r >= dA.max) & (g.m < g.r)){
    return(vX + dA.max*vG)
  }
  # apply golden-section algorithm to g to find a
  dA <- gsection(g, dA.l, dA.r, dA.m)
  return(vX + dA*vG)
}
```

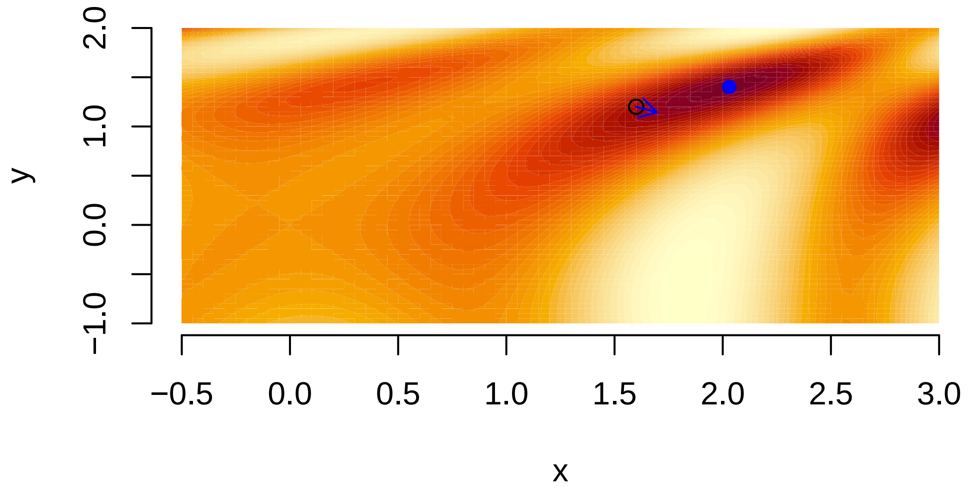
Note: the code depends on the golden section algorithm. Click here to view it.

```r
# golden section algorithm from last week
gsection <- function(f, dX.l, dX.r, dX.m, dTol = 1e-9) {

  # golden ratio plus one
  dGR1 <- 1 + (1 + sqrt(5))/2

  # successively refine x.l, x.r, and x.m
  f.l <- f(dX.l)
  f.r <- f(dX.r)
  f.m <- f(dX.m)
  while ((dX.r - dX.l) > dTol) {
    if ((dX.r - dX.m) > (dX.m - dX.l)) { # if the right segment is wider than the left
      dY <- dX.m + (dX.r - dX.m)/dGR1 # put Y into the right segment according to the golde
      f.y <- f(dY)
      if (f.y >= f.m) {
        dX.l <- dX.m
        f.l <- f.m
        dX.m <- dY
        f.m <- f.y
      } else {
        dX.r <- dY
        f.r <- f.y
      }
    } else { #if the left segment is wider than the right
      dY <- dX.m - (dX.m - dX.l)/dGR1 # put Y into the left segment according to the golden
      f.y <- f(dY)
      if (f.y >= f.m) {
        dX.r <- dX.m
        f.r <- f.m
        dX.m <- dY
        f.m <- f.y
      } else {
        dX.l <- dY
        f.l <- f.y
      }
    }
  }
  return(dX.m)
}
```

9.2.4 Example

```r
# function
f <- function(vX) {
  dOut = sin(vX[1]^2/2 - vX[2]^2/4) * cos(2*vX[1] - exp(vX[2]))
  return(dOut)
}

# gradient
grad.f <- function(vX) {
  dX_prime.1 = cos(vX[1]^2/2 - vX[2]^2/4)*vX[1] * cos(2*vX[1] - exp(vX[2]))
  dX_prime.2 = sin(vX[1]^2/2 - vX[2]^2/4) * (-sin(2*vX[1] - exp(vX[2]))*2)
  dY_prime.1 = cos(vX[1]^2/2 - vX[2]^2/4)*(-vX[2]/2) * cos(2*vX[1] - exp(vX[2]))
  dY_prime.2 = sin(vX[1]^2/2 - vX[2]^2/4) * sin(2*vX[1] - exp(vX[2]))*exp(vX[2])
  vOut = c(dX_prime.1 + dX_prime.2, dY_prime.1 + dY_prime.2)
  return(vOut)
}

ascent(f, grad.f, vX0 = c(0.1, 0.3))
#> [1] 2.030674 1.401513
ascent(f, grad.f, vX0 = c(0, 0.5))
#> [1] 0.3424608 1.4271493
```


mf
vy
vx


2.0
1.0
0.0
−1.0
y
−0.5
0.0
0.5
1.0
1.5
2.0
2.5
3.0
x

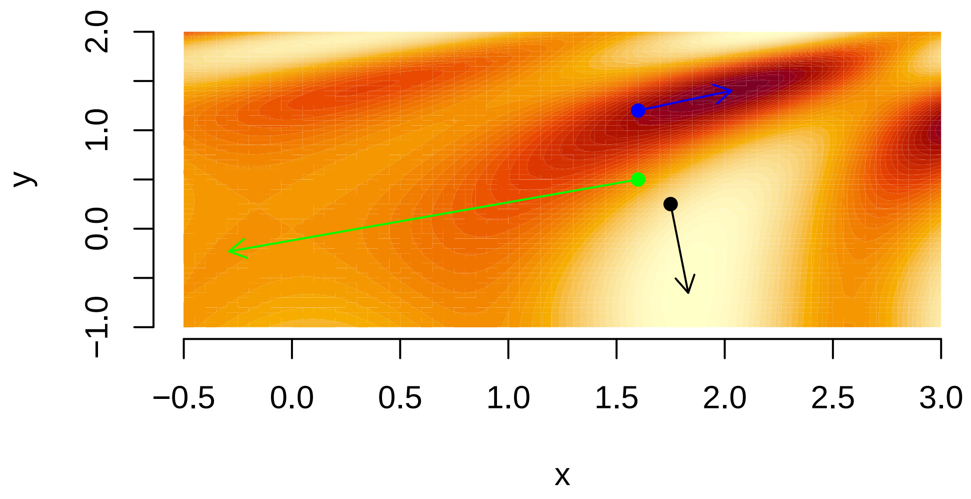
9.3 Newton's method in higher dimensions

9.3.1 Newton's method: Implementation in R

```
newton <- function(f3, vX0, dTol = 1e-9, n.max = 100) {
  # Newton's method for optimisation, starting at x0
  # f3 is a function that given x returns the list
  # {f(x), grad f(x), Hessian f(x)}, for some f
  vX <- vX0
  f3.x <- f3(vX)
  n <- 0
  while ((max(abs(f3.x[[2]])) > dTol) & (n < n.max)) {
    vX <- vX - solve(f3.x[[3]], f3.x[[2]])
    #vX <- vX - solve(f3.x[[3]])%*%f3.x[[2]]
    f3.x <- f3(vX)
    cat("At iteration", n, "the coordinates of x are", vX, "\n")
    n <- n + 1
  }
  if (n == n.max) {
    cat('newton failed to converge\n')
  } else {
    return(vX)
  }
}
```

9.3.2 Example

```
newton(f3, vX0 = c(1.6, 1.2)) # maximum
#> [1] 2.030697 1.401526
newton(f3, vX0 = c(1.6, 0.5)) # saddle point
#> [1] -0.2902213 -0.2304799
newton(f3, vX0 = c(1.75, 0.25)) # minimum
#> [1]  1.8313777 -0.6516928
```



```
eigen(f3(n1)[[3]]) # maximum
#> eigen() decomposition
#> $values
#> [1]  -2.030712 -23.079010
#>
#> $vectors
#>             [,1]       [,2]
#> [1,] -0.8429247 -0.5380316
#> [2,] -0.5380316  0.8429247

eigen(f3(n2)[[3]]) # saddle point
#> eigen() decomposition
#> $values
#> [1] -0.06666478 -1.20252229
#>
#> $vectors
#>            [,1]      [,2]
#> [1,] -0.456103 -0.889927
#> [2,] -0.889927  0.456103

eigen(f3(n3)[[3]]) # minimum
```

```
#> eigen() decomposition
#> $values
#> [1] 7.382289 0.349442
#>
#> $vectors
#>              [,1]        [,2]
#> [1,] -0.99798281 -0.06348473
#> [2,]  0.06348473 -0.99798281
```

Click to view the function f3 and the code for the plot

```r
#function, gradient, and hessian in list
f3 <- function(vX) {
  dA <- vX[1]^2/2 - vX[2]^2/4
  dB <- 2*vX[1] - exp(vX[2])
  f <- sin(dA)*cos(dB)
  f1 <- cos(dA)*cos(dB)*vX[1] - sin(dA)*sin(dB)*2
  f2 <- -cos(dA)*cos(dB)*vX[2]/2 + sin(dA)*sin(dB)*exp(vX[2])
  f11 <- -sin(dA)*cos(dB)*(4 + vX[1]^2) + cos(dA)*cos(dB) -
    cos(dA)*sin(dB)*4*vX[1]
  f12 <- sin(dA)*cos(dB)*(vX[1]*vX[2]/2 + 2*exp(vX[2])) +
    cos(dA)*sin(dB)*(vX[1]*exp(vX[2]) + vX[2])
  f22 <- -sin(dA)*cos(dB)*(vX[2]^2/4 + exp(2*vX[2])) - cos(dA)*cos(dB)/2 -
    cos(dA)*sin(dB)*vX[2]*exp(vX[2]) + sin(dA)*sin(dB)*exp(vX[2])
  return(list(f, c(f1, f2), matrix(c(f11, f12, f12, f22), 2, 2)))
}

n1<-newton(f3, vX0 = c(1.6, 1.2))
n2<-newton(f3, vX0 = c(1.6, 0.5))
n3<-newton(f3, vX0 = c(1.75, 0.25))

{
  plot(NA,xlim=range(vx),
       ylim=range(vy),xlab="x",ylab="y",
       frame=FALSE)
  levels = pretty(range(mf), 50)
  color.palette = function(n) hcl.colors(n, "YlOrRd", rev = TRUE)
  .filled.contour(x=vx, y=vy, z=mf,
                  levels=levels,
                  col=color.palette(length(levels) - 1))
  points(1.6, 1.2, pch=16, col="blue")
  arrows(1.6, 1.2, n1[1], n1[2], length=0.1, col="blue")

  points(1.6, 0.5, pch=16, col="green")
  arrows(1.6, 0.5, n2[1], n2[2], length=0.1, col="green")

  points(1.75, 0.25, pch=16, col="black")
  arrows(1.75, 0.25, n3[1], n3[2], length=0.1, col="black")
}
```

-
-
-

9.4 Disadvantages of steepest ascent and Newton's method

9.5 Gradient and Hessian in R

9.6 Optimization in R: multivariate

- [1]
-
- [2]

9.6.1 Example

```r
# fnscale multiplies the function by -1, such that a maximum is found.
# the default is a minimum
optim(c(1.6, 1.2), f, gr = grad.f, method = "BFGS", control = list(fnscale = -1))
#> $par
#> [1] 2.030697 1.401526
#>
#> $value
#> [1] 1
#>
#> $counts
#> function gradient
#>       22        9
#>
#> $convergence
#> [1] 0
#>
#> $message
#> NULL
```

[1] Nelder, J. A. and Mead, R. (1965) A simplex algorithm for function minimization. Computer Journal 7, 308-313.

[2] Byrd, R. H., Lu, P., Nocedal, J. and Zhu, C. (1995) A limited memory algorithm for bound constrained optimization. SIAM J. Scientific Computing, 16, 1190-1208.

10 Constrained Optimization and Pitfalls in Optimization

10.1 Constrained Optimization

-
-
-

-
-
-

-
-

10.2 Penalization functions

10.3 Barrier functions - (Interior-point)

10.4 Parameter constraints

-
-

1)
2)

10.5 Reparameterization

10.6 Differentiation after reparameterization: Delta method

10.7 Transforming parameters

1.

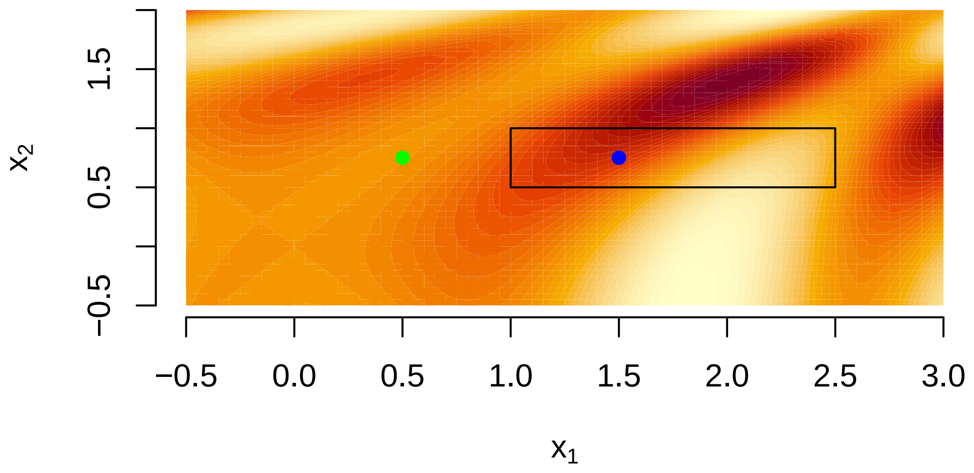
2.

10.8 Transform: Common transformations

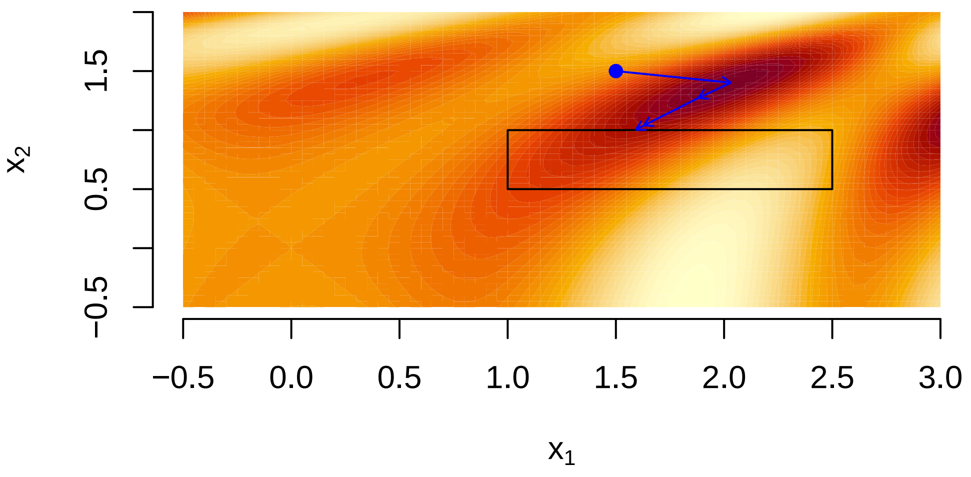
10.9 Code for constrained optimization

Click to view full code associated with the first part

```
# adapted code from last lecture -------------------------------------------

#golden section
gsection <- function(f, dX.l, dX.r, dX.m, dTol = 1e-9, ...) {

  # golden ratio plus one
  dGR1 <- 1 + (1 + sqrt(5))/2

  # successively refine x.l, x.r, and x.m
  f.l <- f(dX.l, ...)
  f.r <- f(dX.r, ...)
  f.m <- f(dX.m, ...)
  while ((dX.r - dX.l) > dTol) {
    if ((dX.r - dX.m) > (dX.m - dX.l)) { # if the right segment is wider than the left
      dY <- dX.m + (dX.r - dX.m)/dGR1 # put Y into the right segment according to the golde
      f.y <- f(dY, ...)
      if (f.y >= f.m) {
        dX.l <- dX.m
        f.l <- f.m
        dX.m <- dY
        f.m <- f.y
      } else {
        dX.r <- dY
        f.r <- f.y
      }
    } else { #if the left segment is wider than the right
      dY <- dX.m - (dX.m - dX.l)/dGR1 # put Y into the left segment according to the golden
      f.y <- f(dY, ...)
      if (f.y >= f.m) {
        dX.r <- dX.m
        f.r <- f.m
        dX.m <- dY
        f.m <- f.y
      } else {
        dX.l <- dY
        f.l <- f.y
      }
    }
  }
  return(dX.m)
}

#line search
line.search <- function(f, vX, vG, dTol = 1e-9, dA.max = 2^5, ...) {
  # f is a real function that takes a vector of length d
  # x and y are vectors of length d
  # line.search uses gsection to find a >= 0 such that
  # g(a) = f(x + a*y) has a local maximum at a,
  # within a tolerance of tol
  # if no local max is found then we use 0 or a.max for a
  # the value returned is x + a*y
  if (sum(abs(vG)) == 0){
    return(vX) # +0*vG
```

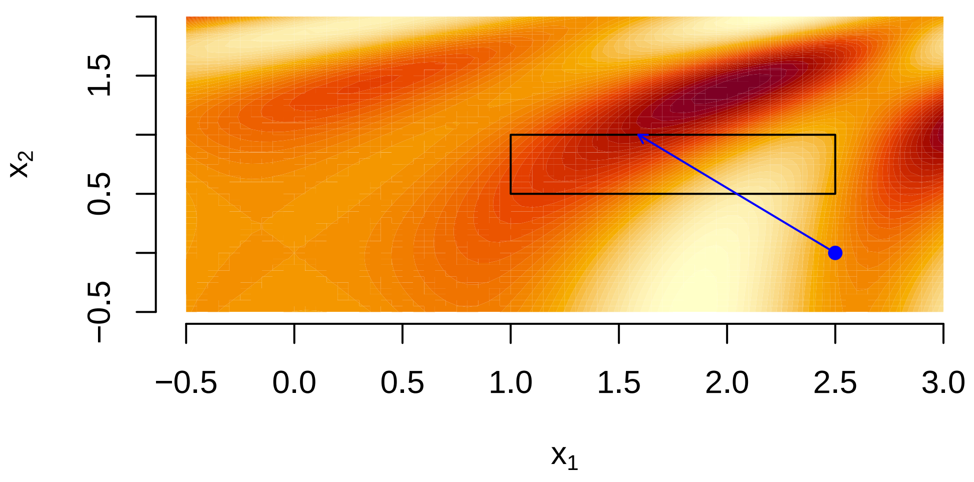

x_2
1.5
0.5
−0.5
−0.5
0.0
0.5
1.0
1.5
2.0
2.5
3.0
x_1

```r
p(vX, va, vb)
#> [1] 0.25



#plotting the solutions for different gamma_h
{
  plot(NA,xlim=range(vx1),
       ylim=range(vx2),xlab=expression("x"[1]),ylab=expression("x"[2]),
       frame=FALSE)
  levels = pretty(range(mf), 50)
  color.palette = function(n) hcl.colors(n, "YlOrRd", rev = TRUE)
  .filled.contour(x=vx1, y=vx2, z=mf,
                  levels=levels,
                  col=color.palette(length(levels) - 1))
  rect(xleft=va[1], ybottom=va[2], xright=vb[1], ytop=vb[2])
}


vX0<-c(1.5,1.5)
points(vX0[1],vX0[2], col="blue", pch=16)


#gamma_h=0, no penalization
vX_star<-ascent(f_p, grad.f_p, vX0, verbose=FALSE, dGamma=0, va=va, vb=vb)
arrows(x0=vX0[1], y0=vX0[2], x1=vX_star[1], y1=vX_star[2], length=0.05, col="blue")
vX_old<-vX_star

#gamma_h=1
vX_star<-ascent(f_p, grad.f_p, vX0=vX_old, verbose=FALSE, dGamma=1, va=va, vb=vb) #hot star
arrows(x0=vX_old[1], y0=vX_old[2], x1=vX_star[1], y1=vX_star[2], length=0.05, col="blue")
vX_old<-vX_star

#gamma_h=10
vX_star<-ascent(f_p, grad.f_p, vX0=vX_old, verbose=FALSE, dGamma=10, va=va, vb=vb)
arrows(x0=vX_old[1], y0=vX_old[2], x1=vX_star[1], y1=vX_star[2], length=0.05, col="blue")
vX_old<-vX_star

#gamma_h=100
vX_star<-ascent(f_p, grad.f_p, vX0, verbose=FALSE, dGamma=100, va=va, vb=vb)
arrows(x0=vX_old[1], y0=vX_old[2], x1=vX_star[1], y1=vX_star[2], length=0.05, col="blue")
```



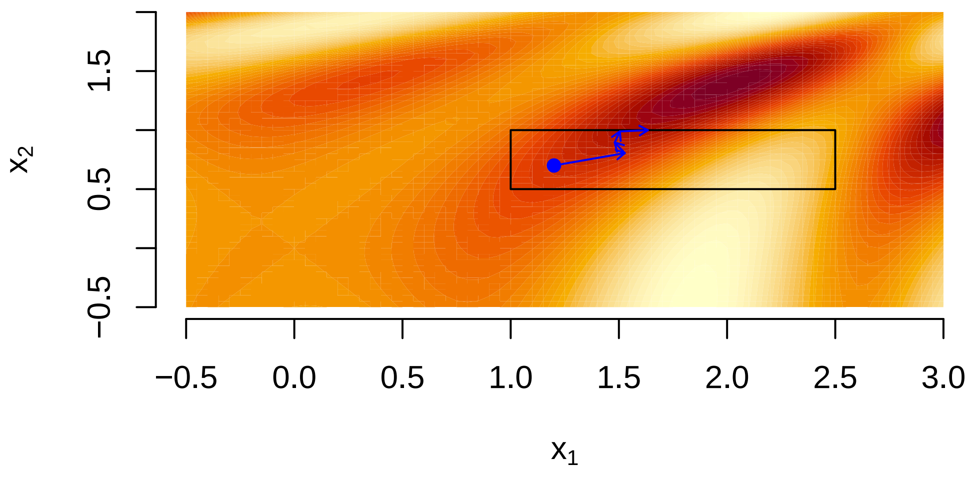
x2
−0.5
0.5
1.5
−0.5
0.0
0.5
1.0
1.5
2.0
2.5
3.0
x1

```r



#making a function that iterates like above
penalized_ascent<-function(f_p, grad.f_p, vX0, epsilon_h = 1e-9, h.max = 100, verbose_ = TR
  #first iteration
  vXh <- ascent(f_p, grad.f_p, vX0, verbose=FALSE, dGamma=0, ...)
  vXh_old <- vX0
  h <- 1
  while( sum(abs(vXh - vXh_old)) > epsilon_h && h < h.max){
    vXh_old <- vXh
    vXh <- ascent(f_p, grad.f_p, vXh_old, verbose=FALSE, dGamma=10^(h-1), ...) #gamma_h get
    if(verbose_){
      cat("at iteration", h, "the coordinates of x are", vXh, "\n")
    }
    h <- h + 1
  }
  return(vXh)
}

#test out how it works for different starting values!
{
  plot(NA,xlim=range(vx1),
       ylim=range(vx2),xlab=expression("x"[1]),ylab=expression("x"[2]),
       frame=FALSE)
  levels = pretty(range(mf), 50)
  color.palette = function(n) hcl.colors(n, "YlOrRd", rev = TRUE)
  .filled.contour(x=vx1, y=vx2, z=mf,
                  levels=levels,
                  col=color.palette(length(levels) - 1))
  rect(xleft=va[1], ybottom=va[2], xright=vb[1], ytop=vb[2])
}


vX0<-c(2.5,0)
points(vX0[1],vX0[2], col="blue", pch=16)
vX<-penalized_ascent(f_p, grad.f_p, vX0, va=va, vb=vb)
#> at iteration 1 the coordinates of x are 3.024968 1.039979
#> at iteration 2 the coordinates of x are 1.629637 1.039826
#> at iteration 3 the coordinates of x are 1.593251 1.003943
#> at iteration 4 the coordinates of x are 1.589693 1.000394
#> at iteration 5 the coordinates of x are 1.589684 1.000039
#> at iteration 6 the coordinates of x are 1.589684 1.000004
#> at iteration 7 the coordinates of x are 1.589684 1
#> at iteration 8 the coordinates of x are 1.589684 1
#> at iteration 9 the coordinates of x are 1.589684 1
arrows(x0=vX0[1], y0=vX0[2], x1=vX[1], y1=vX[2], length=0.05, col="blue")
```



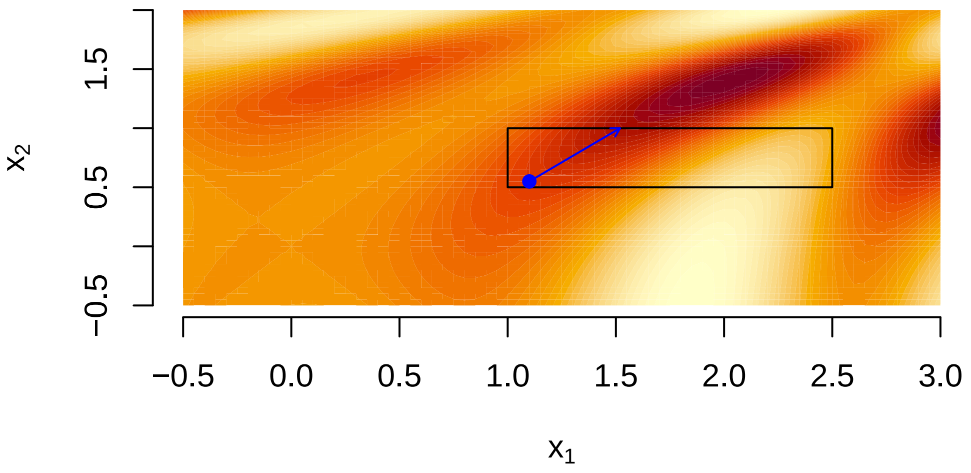
x_2
−0.5
0.5
1.5
−0.5
0.0
0.5
1.0
1.5
2.0
2.5
3.0
x_1

```

# constrained with barrier ------------------------------------------------
#barrier function
b<-function(vX, va, vb){
  gX<-c(vX-vb, va-vX)
  if(all(gX<=0)){
    return(-sum(log(-gX)))
  }else{
    return(Inf)
  }
}
#new objective with a barrier
f_b<-function(vX, dGamma, ...){
  return(f(vX)-dGamma*b(vX, ...)) #subtract the penalty since we're maximizing
}

grad.f_b<-function(vX, dGamma, ...){
  return(grad(func=f_p, x=vX, dGamma=dGamma, ...)) # using numerical derivatives here becau
}


#plotting the solutions for different gamma_h
{
  plot(NA,xlim=range(vx1),
       ylim=range(vx2),xlab=expression("x"[1]),ylab=expression("x"[2]),
       frame=FALSE)
  levels = pretty(range(mf), 50)
  color.palette = function(n) hcl.colors(n, "YlOrRd", rev = TRUE)
  .filled.contour(x=vx1, y=vx2, z=mf,
                  levels=levels,
                  col=color.palette(length(levels) - 1))
  rect(xleft=va[1], ybottom=va[2], xright=vb[1], ytop=vb[2])
}


vX0<-c(1.2,0.7) #start inside the box
points(vX0[1],vX0[2], col="blue", pch=16)


#gamma_h=1
vX_star<-ascent(f_b, grad.f_b, vX0, verbose=FALSE, dGamma=1, va=va, vb=vb)
arrows(x0=vX0[1], y0=vX0[2], x1=vX_star[1], y1=vX_star[2], length=0.05, col="blue")
vX_old<-vX_star

#gamma_h=1/10
vX_star<-ascent(f_b, grad.f_b, vX0=vX_old, verbose=FALSE, dGamma=1/10, va=va, vb=vb) #hot s
arrows(x0=vX_old[1], y0=vX_old[2], x1=vX_star[1], y1=vX_star[2], length=0.05, col="blue")
vX_old<-vX_star

#gamma_h=1/100
vX_star<-ascent(f_b, grad.f_b, vX0=vX_old, verbose=FALSE, dGamma=1/100, va=va, vb=vb)
arrows(x0=vX_old[1], y0=vX_old[2], x1=vX_star[1], y1=vX_star[2], length=0.05, col="blue")
vX_old<-vX_star
```



x2
1.5
0.5
−0.5
−0.5
0.0
0.5
1.0
1.5
2.0
2.5
3.0
x1

```r

barrier_ascent<-function(f_b, grad.f_b, vX0, epsilon_h = 1e-9, h.max = 100, verbose_ = TRUE
  #first iteration
  vXb <- ascent(f_b, grad.f_b, vX0, verbose=FALSE, dGamma=1, ...)
  vXb_old <- vX0
  h <- 1
  while( sum(abs(vXb - vXb_old)) > epsilon_h && h < h.max){
    vXb_old <- vXb
    vXb <- ascent(f_b, grad.f_b, vXb_old, verbose=FALSE, dGamma=10^(-h), ...) #gamma_h gets
    if(verbose_){
      cat("at iteration", h, "the coordinates of x are", vXb, "\n")
    }
    h <- h + 1
  }
  return(vXb)
}

#test out how it works for different starting values!
{
  plot(NA,xlim=range(vx1),
       ylim=range(vx2),xlab=expression("x"[1]),ylab=expression("x"[2]),
       frame=FALSE)
  levels = pretty(range(mf), 50)
  color.palette = function(n) hcl.colors(n, "YlOrRd", rev = TRUE)
  .filled.contour(x=vx1, y=vx2, z=mf,
                  levels=levels,
                  col=color.palette(length(levels) - 1))
  rect(xleft=va[1], ybottom=va[2], xright=vb[1], ytop=vb[2])
}


vX0<-c(1.1,0.55)
points(vX0[1],vX0[2], col="blue", pch=16)
vX<-barrier_ascent(f_b, grad.f_b, vX0, va=va, vb=vb)
#> at iteration 1 the coordinates of x are 1.490837 0.8984811
#> at iteration 2 the coordinates of x are 1.506099 0.9881654
#> at iteration 3 the coordinates of x are 1.517451 0.9987611
#> at iteration 4 the coordinates of x are 1.518635 0.9998752
#> at iteration 5 the coordinates of x are 1.518754 0.9999875
#> at iteration 6 the coordinates of x are 1.518766 0.9999988
#> at iteration 7 the coordinates of x are 1.518767 0.9999999
#> at iteration 8 the coordinates of x are 1.518767 1
#> at iteration 9 the coordinates of x are 1.518767 1
#> at iteration 10 the coordinates of x are 1.518767 1
#> at iteration 11 the coordinates of x are 1.518767 1
arrows(x0=vX0[1], y0=vX0[2], x1=vX[1], y1=vX[2], length=0.05, col="blue")
```



10.10 Pitfalls

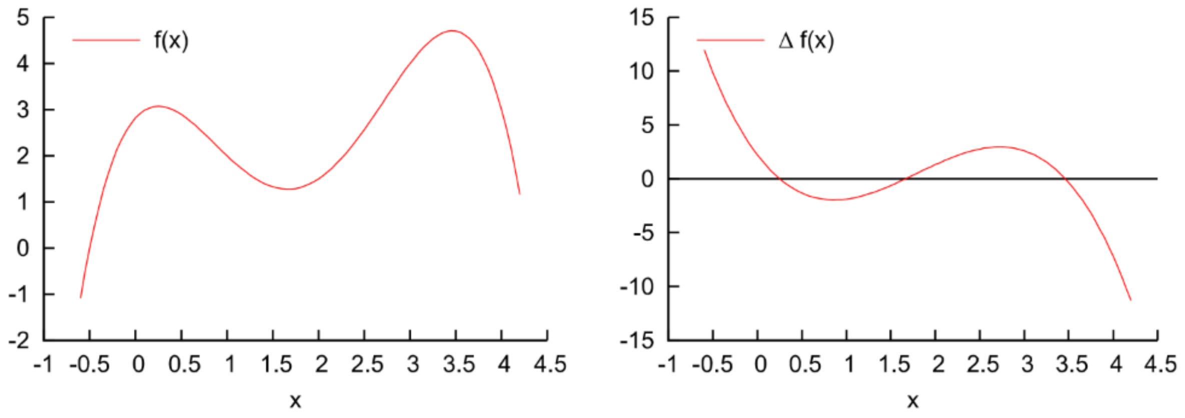


Figure 10.1: Pitfall 1

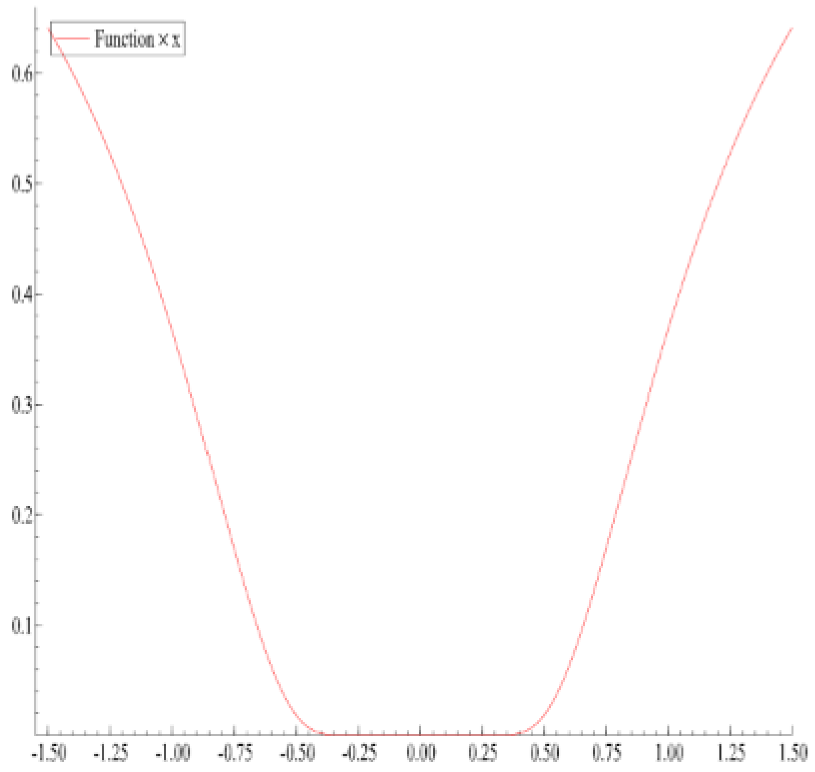


Figure 10.2: Pitfall 2

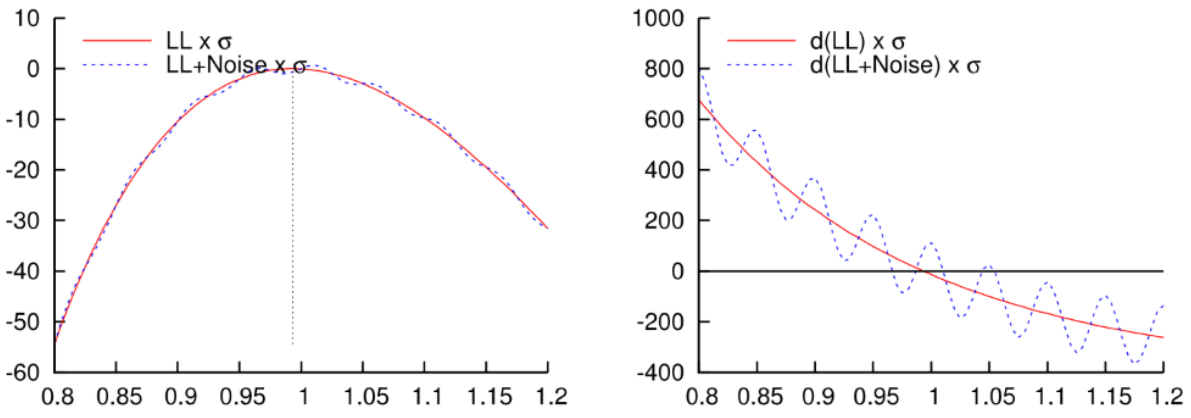


Figure 10.3: Pitfall 3

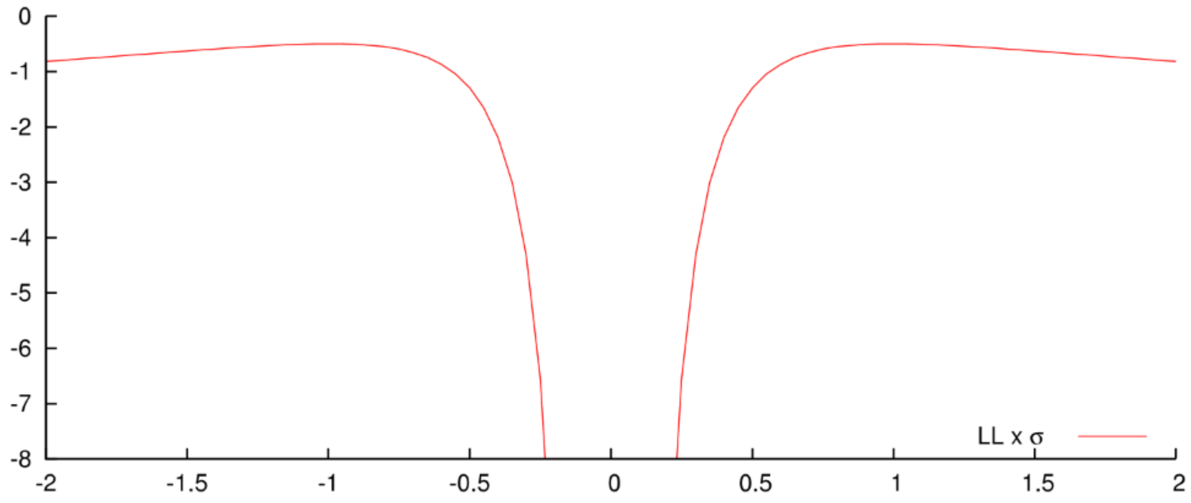


Figure 10.4: Pitfall 4

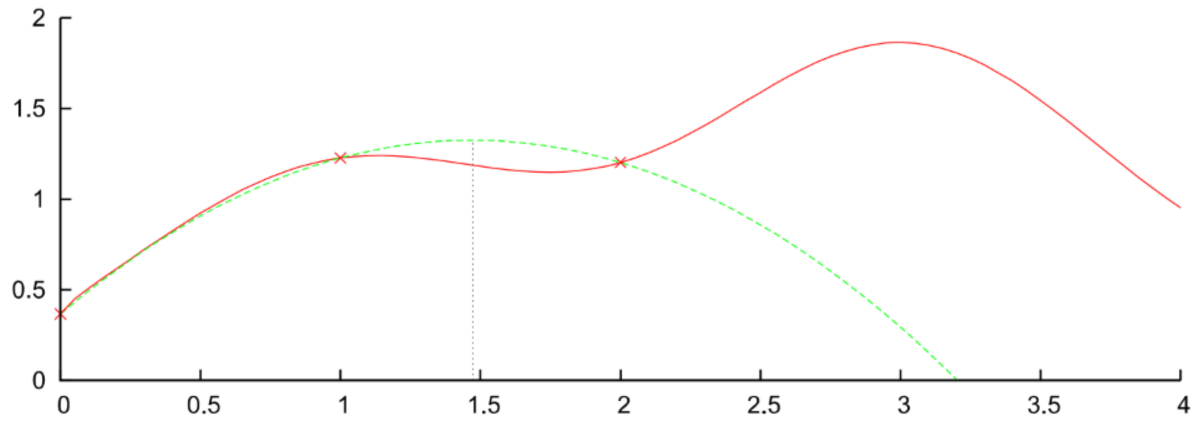


Figure 10.5: Pitfall 5

10.11 Problematic Hessian?

-
-
-

10.12 Recap of Newton's method in higher dimensions

10.13 Broyden, Fletcher, Goldfarb and Shanno (BFGS)

1.
2.
3.

4.

5.

6.

10.14 Example: Regression Likelihood

-
-
-

10.15 `optim` continued

-
-
-

-
-
-
-

-

10.16 Code for pitfalls and BFGS

Click to view full code associated with the first part

```
##  Purpose: Constrained Optimization
```

11 Integration of C++ Code with Rcpp

11.1 Introduction

11.2 Rcpp

11.3 R versus C++

-
-
-

11.4 Basic C++ in R

```r
install.packages("Rcpp")
```

-
-

11.5 C++ functions in R

```r
suppressMessages(library(Rcpp))
#> Warning: pakke 'Rcpp' blev bygget under R version 4.3.3

cppFunction('
int add(int x, int y, int z) {
  int sum = x + y + z;
  return sum;
}')
```

```
add(1, 2, 3)
#> [1] 6
```

11.5.1 Example: No inputs, scalar output

```
oneR <- function() 1L # R function

cppFunction('
int oneCpp() { # C++ function
  return 1,
}')
```

11.6 Variables declation

```
iN <- 5
dK <- 0.156532
bJ <- TRUE
sT <- "Hello World"
```

```
int    iN = 5;              // integer
double dK = 0.156352;       // double
bool   bJ = true;           // boolean
string st = "Hello World"; // string
```

11.6.1 Example: scalar input, scalar output

```
signR <- function(iX) {
  if (iX > 0) {
    1
  } else if (iX == 0) {
    0
  } else {
    -1
  }
}

cppFunction('int SignC(int x) {
  if (x > 0) {
    return 1;
  } else if (x == 0) {
    return 0;
  } else {
    return -1;
  }
}')
```

11.7 For loops in C++

```
for (int i = 0 (init); i < 10 (check); i++ (increment)) {
  // some code here
}
```

11.7.1 Example: vector input, scalar output

```
sumR <- function(vX) {
  dTotal <- 0
  for (i in 1:length(vX)) {
    dTotal <- dTotal + vX[i]
  }
  return(dTotal)
}

cppFunction('
double sumC(NumericVector x) {
  int n = x.size();
  double total = 0;
  for(int i = 0; i < n; i++) {
    total += x(i);
  }
  return total;
}')
```

11.7.2 Efficiency of loops: C++ vs. R

```
suppressMessages(library(microbenchmark))
#> Warning: pakke 'microbenchmark' blev bygget under R version 4.3.3

vX <- runif(1e4)
microbenchmark(
  sum(vX),
  sumC(vX),
  sumR(vX)
)
#> Unit: microseconds
#>      expr   min     lq    mean median     uq    max neval
#>   sum(vX)   7.4   7.60   8.168   7.65   8.05   15.9   100
#>  sumC(vX)  21.8  22.30  33.565  22.65  24.80  945.9   100
#>  sumR(vX) 272.2 274.95 317.612 276.70 286.40 2493.0   100
```

11.7.3 Example: vector input, vector output

```
pdistR <- function(dX, vY) {
  sqrt((dX - vY) ^ 2)
}

cppFunction('
NumericVector pdistC(double x, NumericVector y) {
  int n = y.size();
  NumericVector out(n);

  for (int i = 0; i < n; i++) {
    out(i) = sqrt(pow(y(i) - x, 2.0));
  }
  return out;
}')
```

11.7.4 Example: matrix input, vector output

```
cppFunction('
NumericVector rowSumsC(NumericMatrix x) {
  int nrow = x.nrow(), ncol = x.ncol();
  NumericVector out(nrow);

  for (int i = 0; i < nrow; i++) {
    double total = 0;
    for (int j = 0; j < ncol; j++) {
      total += x(i, j);
    }
    out(i) = total;
  }
  return out;
}')
```

```
set.seed(1014)
mX <- matrix(sample(100), 10)
rowSums(mX)
#>  [1] 446 514 480 514 352 627 525 586 572 434
rowSumsC(mX)
#>  [1] 446 514 480 514 352 627 525 586 572 434
```

11.8 Using sourceCpp

```
/*** R
# This is R code
*/
```

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
double meanC(NumericVector x) {
  int n = x.size();
  double total = 0;
  for (int i = 0; i < n; i++) {
    total += x(i);
  }
  return total / n;
}


/*** R
suppressMessages(library(microbenchmark))
vX <- runif(1e5)
microbenchmark(mean(vX), meanC(vX))
*/
```

```
sourceCpp("CppSource.cpp")
#>
#> > suppressMessages(library(microbenchmark))
#>
#> > vX <- runif(1e+05)
#>
#> > microbenchmark(mean(vX), meanC(vX))
#> Unit: microseconds
#>       expr   min     lq    mean median     uq    max neval
#>   mean(vX) 152.2 154.05 162.447  155.6 162.70  333.2   100
#>  meanC(vX) 221.2 222.35 249.120  224.4 227.75 1502.2   100
suppressMessages(library(microbenchmark))
vX <- runif(1e5)
microbenchmark(
  mean(vX),
  meanC(vX)
)
#> Unit: microseconds
#>       expr   min     lq    mean median     uq   max neval
#>   mean(vX) 151.8 153.85 163.742 156.25 166.00 332.7   100
#>  meanC(vX) 221.1 223.55 235.354 224.90 230.55 429.4   100
```

11.9 The Fibonacci sequence

11.9.1 Fibonacci in R

```
fibR <- function(n) {
  if (n == 0) return(0)
  if (n == 1) return(1)
  return(fibR(n - 1) + fibR(n - 2))
}
```

-
-
-
-

11.9.2 Fibonacci in C++

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
int fibC(int x) {
  if (x == 0) return(0);
  if (x == 1) return(1);
  return(fibC(x - 1) + fibC(x - 2));
}
```

11.9.3 Comparing fibR and fibC

```
suppressMessages(library(Rcpp))
sourceCpp("fibonacci.cpp")
fibC(25)
#> [1] 75025
```

```
suppressMessages(library(microbenchmark))
microbenchmark(fibR(25), fibC(25))
#> Unit: microseconds
#>      expr     min       lq      mean   median       uq      max neval
#>  fibR(25) 92850.9 95145.30 98075.012 96429.10 98875.70 132369.9   100
#>  fibC(25)   119.3   122.95   153.469   134.25   154.05   1380.4   100
```

11.10 Vector, Matrices and Linear Algebra in C++

```r
install.packages("RcppArmadillo")
```

-
-

11.10.1 Example

```cpp
// [[Rcpp::depends(RcppArmadillo)]]
#include <RcppArmadillo.h>
using namespace Rcpp;


// [[Rcpp:export]]
arma::mat matprodC(arma::mat m1, arma::mat m2) {
  if(m1.n_cols != m2.n_rows)
    stop("Incompatible matrix dimensions");

  return m1 * m2;
}
```

```r
suppressMessages(library(RcppArmadillo))
#> Warning: pakke 'RcppArmadillo' blev bygget under R version 4.3.3
sourceCpp("armadilloexample.cpp")
m1 <- matrix(1:9, 3, 3)
m2 <- matrix(1:9, 3, 3)

matprodC(m1, m2)
```

```
#>      [,1] [,2] [,3]
#> [1,]   30   66  102
#> [2,]   36   81  126
#> [3,]   42   96  150

m1 %*% m2
#>      [,1] [,2] [,3]
#> [1,]   30   66  102
#> [2,]   36   81  126
#> [3,]   42   96  150
```

11.11 Inverse of Matrix Product

```
FunR <- function(mX, mY) {
  mZ <- mX %*% mY
  mZInv <- solve(mZ)
  return(mZInv)
}
```

```
// [[Rcpp:depends(RcppArmadillo)]]
#include <RcppArmadillo.h>
using namespace arma;
using namespace Rcpp;

// [[Rcpp::export]]
mat FunC(mat mx, mat mY) {
  mat mZ = mx * mY;
  mat mZInv = mZ.i();
  return mZInv;
}
```

```r
suppressMessages(library(RcppArmadillo))
sourceCpp("mycfunction.cpp")
M <- 300
mX <- matrix(rnorm(M^2), M)
mY <- matrix(rnorm(M^2), M)

microbenchmark(FunR(mX, mY), FunC(mX, mY))
#> Unit: milliseconds
#>          expr     min      lq     mean  median      uq     max neval
#>  FunR(mX, mY) 35.5749 36.2369 36.99473 36.4717 36.6533 61.1553   100
#>  FunC(mX, mY) 28.6633 29.0092 29.65765 29.6037 29.8801 36.2092   100
```

11.12 Calling R Functions from C++

```r
SumOfSquaresR <- function(VX) {
  dOut <- sum(vX^2)
  return(dOut)
}
vX <- rnorm(10)
SumOfSquaresR(vX)
#> [1] 15.98534
```

```cpp
#include <Rcpp.h>
using namespace Rcpp;


// [[Rcpp::export]]
SEXP callRFunction(NumericVector vX, Function f) {
  SEXP res = f(vX);
  return res;
}
```

```r
sourceCpp("callRFunction.cpp")
```

```r
callRFunction(vX, SumOfSquaresR)
#> [1] 15.98534
SumOfSquaresR(vX)
#> [1] 15.98534
```

```r
SumOfAbsoluteValuesR <- function(vX) {
  dOut <- sum(abs(vX))
  return(dOut)
}
```

```r
callRFunction(vX, SumOfAbsoluteValuesR)
#> [1] 10.25095
SumOfAbsoluteValuesR(vX)
#> [1] 10.25095
```

11.12.1 Warning

```r
suppressMessages(library(microbenchmark))
vX <- rnorm(1e6)
microbenchmark(
  callRFunction(vX, SumOfSquaresR),
  SumOfSquaresR(vX)
)
#> Unit: milliseconds
#>                              expr    min     lq     mean  median     uq    max
#>  callRFunction(vX, SumOfSquaresR) 2.1354 2.2131 3.114011 2.26725 2.3750 6.7769
#>                  SumOfSquaresR(vX) 2.1184 2.2105 2.846766 2.24515 2.3945 6.0917
#>  neval
#>    100
#>    100
```

11.13 The Rcpp Family

-
-
-

11.14 A simple model for daily financial returns

11.15 Estimation of the GARCH(1,1) model

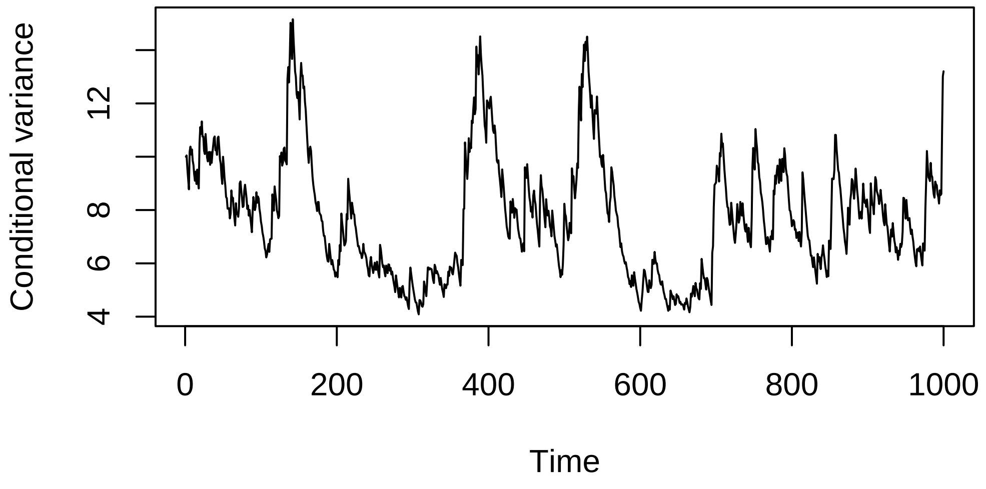
11.15.1 GARCH practical issues

11.16 Simulation of GARCH(1,1) in C++

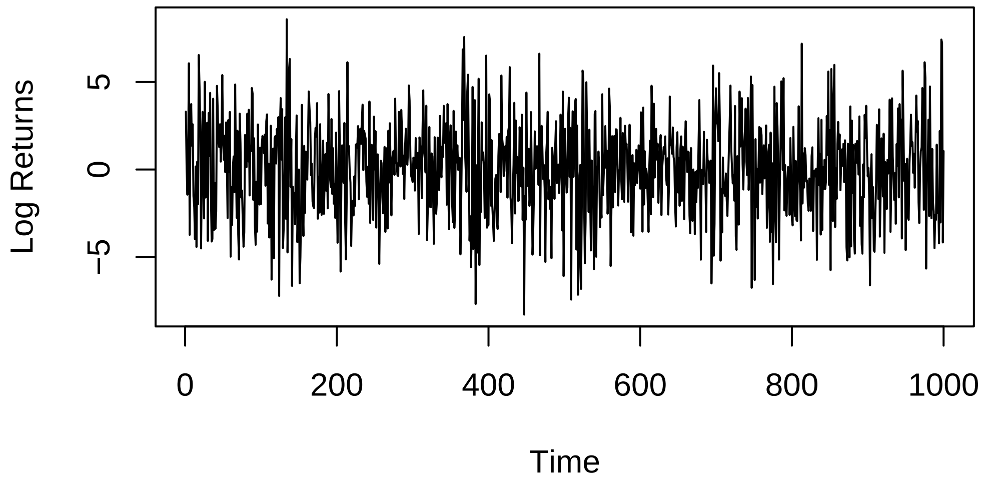
```cpp
// [[Rcpp::depends(RcppArmadillo)]]
#include <RcppArmadillo.h>

using namespace arma;
using namespace Rcpp;

// [[Rcpp::export]]
Rcpp::List GarchSim(int iT, double dOmega, double dAlpha, double dBeta) {
  vec vY(iT); //observations
  vec vSigma2(iT); // conditional variances

  // initialize at the unconditional value
  vSigma2(0) = dOmega/(1.0 - dAlpha - dBeta);

  // sample the first observation
  vY(0) = pow(vSigma2(0), 0.5) * Rf_rnorm(0.0, 1.0);

  for (int t = 1; t < iT; t++) {
    vSigma2(t) = dOmega + dAlpha * pow(vY(t - 1), 2.0) + dBeta * vSigma2(t - 1);
    vY(t) = pow(vSigma2(t), 0.5) * Rf_rnorm(0.0, 1.0);
  }

  List lOut;
  lOut["vSigma2"] = vSigma2;
  lOut["vY"] = vY;

  return lOut;
}
```

-
-
-
-
-

```
sourceCpp("SimGarch.cpp")

iT <- 1000
dOmega <- 0.1
dAlpha <- 0.05
dBeta <- 0.94

lSim <- GarchSim(iT, dOmega, dAlpha, dBeta)

plot(1:1000, lSim[["vSigma2"]], type = "l", lty = 1, xlab = "Time", ylab = "Conditional vari
```



```
plot(1:1000, lSim[["vY"]], type = "l", lty = 1, xlab = "Time", ylab = "Log Returns")
```



12 Creating R packages with RStudio

https://r-pkgs.had.co.nz

12.1 Introduction

12.2 Why a package?

-
-

12.3 Setting up

```
install.packages(c("Rcpp", "RcppArmadillO", "devtools"))
```

https://cran.r-project.org/bin/windows/Rtools/index.html

12.4 Code in Packages

```
MySumFunction <- function(a, b) {
  c <- a + b
  return(c)
}
```

```
MySumFunction(5, 7)
#> [1] 12
```

```
#include <Rcpp.h>
using namespace Rcpp;


// [[Rcpp::export]]
NumericVector timesTwo(NumericVector x) {
  return x * 2;
}


// You can include R code blocks in C++ files processed with sourceCpp
// (useful for testing and development). The R code will be automatically
// run after the compilation.
//

/*** R
timesTwo(42)
*/
```

```
#include <RcppArmadillo.h>

using namespace arma;
using namespace Rcpp;
```

```
#include <RcppArmadillo.h>

using namespace arma;
using namespace Rcpp;

//[[Rcpp::export]]
arma::vec SumOfTwoVec_C(arma::vec vX, arma::vec vY) {
  arma::vec vZ = vX + vY;
  return vZ;
}
```

12.5 Object documentation

-
-
-

12.5.1 Documentation for -package

```
\name{packageexample-package}
\alias{packageexample-package}
\alias{packageexample}
\docType{package}
\title{
\packageTitle{packageexample}
}
\description{
\packageDescription{packageexample}
}
\details{

The DESCRIPTION file:
\packageDESCRIPTION{packageexample}
\packageIndices{packageexample}
~~ An overview of how to use the package, including the most important functions ~~
}
\author{
\packageAuthor{packageexample}

Maintainer: \packageMaintainer{packageexample}
}
\references{
~~ Literature or other references for background information ~~
}
~~ Optionally other standard keywords, one per line, from file KEYWORDS in the R documentati
\keyword{ package }
\seealso{
```

```
~~ Optional links to other man pages, e.g. ~~
~~ \code{\link[<pkg>:<pkg>-package]{<pkg>}} ~~
}
\examples{
~~ simple examples of the most important functions ~~
}
```

```
\name{RcppArmadillo-Functions}
\alias{rcpparma_hello_world}
\alias{rcpparma_innerproduct}
\alias{rcpparma_outerproduct}
\alias{rcpparma_bothproducts}
\title{Set of functions in example RcppArmadillo package}
\description{
  These four functions are created when
  \code{RcppArmadillo.package.skeleton()} is invoked to create a
  skeleton packages.
}
\usage{
rcpparma_hello_world()
rcpparma_outerproduct(x)
rcpparma_innerproduct(x)
rcpparma_bothproducts(x)
}
\arguments{
  \item{x}{a numeric vector}
}
\value{
  \code{rcpparma_hello_world()} does not return a value, but displays a
  message to the console.

  \code{rcpparma_outerproduct()} returns a numeric matrix computed as the
  outer (vector) product of \code{x}.

  \code{rcpparma_innerproduct()} returns a double computer as the inner
  (vector) product of \code{x}.

  \code{rcpparma_bothproducts()} returns a list with both the outer and
  inner products.
}
\details{
  These are example functions which should be largely
  self-explanatory. Their main benefit is to demonstrate how to write a
  function using the Armadillo C++ classes, and to have to such a
  function accessible from R.
}
\references{
  See the documentation for Armadillo, and RcppArmadillo, for more details.
}
\examples{
  x <- sqrt(1:4)
  rcpparma_innerproduct(x)
  rcpparma_outerproduct(x)
}
\author{Dirk Eddelbuettel}
```

12.6 Description

```
Package: packageexample
Type: Package
Title: What the package does (short line)
Version: 1.0
Date: 2025-05-16
Author: Who wrote it
Maintainer: Who to complain to <yourfault@somewhere.net>
Description: More about what it does (maybe more than one line)
License: What license is it under?
Imports: Rcpp (>= 1.0.13-1)
LinkingTo: Rcpp, RcppArmadillo
```

12.7 Different types of R packages

12.8 Beyond the basics

https://github.com/

-
-
-
-
-
- https://happygitwithr.com/

https://cran.r-project.org/

-
-
-
-
-
- https://r-pkgs.org/release.html

12.9 Coding Style and Swirl

- https://adv-r.had.co.nz/Style.html
-

```r
devtools::load_all(".")
install.packages("swirl")
library(swirl)
swirl()
```

Part II

Problem sets

13 Exercise set 1

13.1 (1)

```r
vX <- 1:5 # Numbers 1, 2, 3, 4, 5
vY <- seq(2, 10, 2) # Numbers 2, 4, 6, 8, 10

# Addition
vX + 10
#> [1] 11 12 13 14 15
vY + 1
#> [1]  3  5  7  9 11

# Subtraction
vX - 10
#> [1] -9 -8 -7 -6 -5
vY - 1
#> [1] 1 3 5 7 9

# Multiplication
vX * 10
#> [1] 10 20 30 40 50
vY * -1
#> [1]  -2  -4  -6  -8 -10

# Powers
vX ^ 2
#> [1]  1  4  9 16 25
vY ^ -1
#> [1] 0.5000000 0.2500000 0.1666667 0.1250000 0.1000000
```

```
# Other functions: division (/), exponentiation (^), square root (sqrt())...
```

13.2 (2)

```
v1 <- c(1, 2, 2, 1)
v2 <- c(2, 3, 3, 2)

# Element-wise addition
v1 + v2
#> [1] 3 5 5 3

# Element-wise subtraction
v1 - v2
#> [1] -1 -1 -1 -1

# Element-wise multiplication
v1 * v2
#> [1] 2 6 6 2

# Concatenation into new vector
v3 <- c(v1, v2)
v3
#> [1] 1 2 2 1 2 3 3 2
```

13.3 (3)

```
mA <- matrix(1:6, nrow = 2, ncol = 3, byrow = TRUE)
mA
#>      [,1] [,2] [,3]
#> [1,]    1    2    3
#> [2,]    4    5    6

# Determining the dimensions (number of rows and columns)
dim(mA)
#> [1] 2 3

# Or retrieving either the number of rows or number of columns
nrow(mA) # Rows
#> [1] 2

ncol(mA) # Columns
#> [1] 3
```

13.4 (4)

```
mA <- matrix(seq(12, 1, -1), 3, 4, byrow = TRUE)
mA
#>      [,1] [,2] [,3] [,4]
#> [1,]   12   11   10    9
#> [2,]    8    7    6    5
#> [3,]    4    3    2    1
```

```
# Minimum and maximum of each row
apply(mA, 1, min) # `apply` applies the function `min` to each row (indicated by 1)
#> [1] 9 5 1

apply(mA, 1, max)
#> [1] 12  8  4
```

```
colSums(mA)
#> [1] 24 21 18 15
```

```
mB <- apply(mA, 1, cumsum) # See notes for `apply` above. `cumsum` returns the cumulative su
mB # Column 1 in mB is the cumulative sum of row 1 in mA, and so on....
#>      [,1] [,2] [,3]
#> [1,]   12    8    4
#> [2,]   23   15    7
#> [3,]   33   21    9
#> [4,]   42   26   10
```

```
mC <- apply(mA, 1, cumprod)
mC # Column 1 in mC is the cumulative product of row 1 in mA, and so on....
#>       [,1] [,2] [,3]
#> [1,]    12    8    4
#> [2,]   132   56   12
#> [3,]  1320  336   24
#> [4,] 11880 1680   24
```

```
mA # Unsorted
#>      [,1] [,2] [,3] [,4]
#> [1,]   12   11   10    9
#> [2,]    8    7    6    5
#> [3,]    4    3    2    1

sort(mA[, 1]) # Sort and return only the first column
#> [1]  4  8 12

mA[order(mA[, 1]), ] # All rows sorted by the first column. `order` returns a permutation tha
#>      [,1] [,2] [,3] [,4]
#> [1,]    4    3    2    1
#> [2,]    8    7    6    5
#> [3,]   12   11   10    9
```

13.5 (5)

```r
set.seed(1) # Make the results reproducible by setting a seed
vU <- runif(n = 250, min = 0, max = 1) # The final two arguments are not necessary as they a
```

```r
if (!require("e1071")) install.packages("e1071")
suppressMessages(library(e1071))
```

```r
cat("The mean is: ", mean(vU), "\n")
#> The mean is:  0.5098254

cat("The variance is: ", var(vU), "\n")
#> The variance is:  0.07249006

cat("The skewness is: ", skewness(vU), "\n")
#> The skewness is:  0.04188663

cat("The kurtosis is: ", kurtosis(vU), "\n")
#> The kurtosis is:  -1.121881
```

```r
iN <- length(vU) # Number of observations
dStdU <- sqrt(var(vU)) # Standard deviation of U

dMoment3 <- sum((vU - mean(vU)) ^ 3) / iN # 3rd central moment
dSkewnews <- dMoment3 / dStdU ^ 3
dSkewnews
#> [1] 0.04188663

dMoment4 <- sum((vU - mean(vU)) ^ 4) / iN # 4th central moment
dKurtosis <- dMoment4 / dStdU ^ 4 - 3 # Excess kurtosis
dKurtosis
#> [1] -1.121881
```

13.6 (6)

-

```r
sR <- "RisGreat"
sub("R", "S", sR) # Note, `sub` only replaces the first occurence. `gsub` replaces all.
#> [1] "SisGreat"
```

14 Exercise set 2

14.1 (1)

```
x <- 2
a <- 3
b <- 2
z <- x ^ (a ^ b)
z
#> [1] 512
```

```
z <- (x ^ a) ^ b
z
#> [1] 64
```

```
z <- 3 * x ^ 3 + 2 * x ^ 2 + 6 * x + 1
z
#> [1] 45
```

```
z <- z + 1
z
#> [1] 46
```

14.2 (2)

```
c(1:8, 7:1) # Simple concatenation, but notice how 7:1 returns a decreasing vector of intege
#>  [1] 1 2 3 4 5 6 7 8 7 6 5 4 3 2 1
```

```
rep.int(x = 1:5, times = 1:5) # `rep.int` is faster than `rep` and only requires two argumen
#>  [1] 1 2 2 3 3 3 4 4 4 4 5 5 5 5 5
```

```
1 - diag(3) # We create a matrix of ones and subtract an identity matrix of size 3
#>      [,1] [,2] [,3]
#> [1,]    0    1    1
#> [2,]    1    0    1
#> [3,]    1    1    0
```

```
matrix(c(0, 0, 7, 2, 4, 0, 3, 0, 0), 3, 3) # Manually creating the matrix seems to be the eas
#>      [,1] [,2] [,3]
#> [1,]    0    2    3
#> [2,]    0    4    0
#> [3,]    7    0    0
```

14.3 (3)

```
vX <- 1:100
vX[(vX %% 2 != 0) & (vX %% 3 != 0) & (vX %% 7 != 0)] # We use the modulo operator for indexi
#>  [1]  1  5 11 13 17 19 23 25 29 31 37 41 43 47 53 55 59 61 65 67 71 73 79 83 85
#> [26] 89 95 97
```

14.4 (4)

```
mI <- diag(10) # 10 x 10 identity matrix
mI[mI != 0] <- 5 # Making the non-zero elements 5 by using indexing
mI
#>       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
#>  [1,]    5    0    0    0    0    0    0    0    0     0
#>  [2,]    0    5    0    0    0    0    0    0    0     0
#>  [3,]    0    0    5    0    0    0    0    0    0     0
#>  [4,]    0    0    0    5    0    0    0    0    0     0
#>  [5,]    0    0    0    0    5    0    0    0    0     0
#>  [6,]    0    0    0    0    0    5    0    0    0     0
#>  [7,]    0    0    0    0    0    0    5    0    0     0
#>  [8,]    0    0    0    0    0    0    0    5    0     0
#>  [9,]    0    0    0    0    0    0    0    0    5     0
#> [10,]    0    0    0    0    0    0    0    0    0     5

# Other solutions
mA <- diag(10)
mA <- ifelse(mA != 0, 5, mA) # Using if-else
all.equal(mI, mA)
#> [1] TRUE

mB <- diag(10)
diag(mB) <- 5 # Modify the diagonal directly
all.equal(mI, mB)
#> [1] TRUE
```

14.5 (5)

```
f <- function(iX) {
  if (iX <= 0) {
    return(- iX ^ 3)
  } else if (iX <= 1) {
    return(iX ^ 2)
  } else {
    return(sqrt(iX))
  }
}

set.seed(1)
x <- runif(n = 1, min = -1, max = 2) # Random variable with equal probability of being in the

cat("x is: ", x, " and y is: ", f(x))
#> x is:  -0.203474  and y is:  0.008424164
```

```
# Bonus: the function in one-line: f <- function(iX) ifelse(iX <= 0, -iX ^ 3, ifelse(iX <= 1
```

14.6 (6)

```
# Note, this is just a finite geometric series
iX <- 5 # Example values
iN <- 4
iGeoSum <- 0
for (iIte in 0:iN) {
  iGeoSum <- iGeoSum + iX ^ iIte
}
iGeoSum
#> [1] 781
```

14.7 (7)

```
iX <- 5 # Example values
iN <- 4
iGeoSum <- 0
iIte <- 0
while (iIte <= iN) {
  iGeoSum <- iGeoSum + iX ^ iIte
  iIte <- iIte + 1
}
iGeoSum
#> [1] 781

sum(iX ^ (0:iN)) # Vector-operations. The one-liner uses element-wise exponentiation against
#> [1] 781
```

14.8 (8)

```r
vX <- 1:100
sum(vX[seq.int(3, length(vX), 3)]) # We use some smart indexing of our vector x. `seq.int` i
#> [1] 1683
```

14.9 (9)

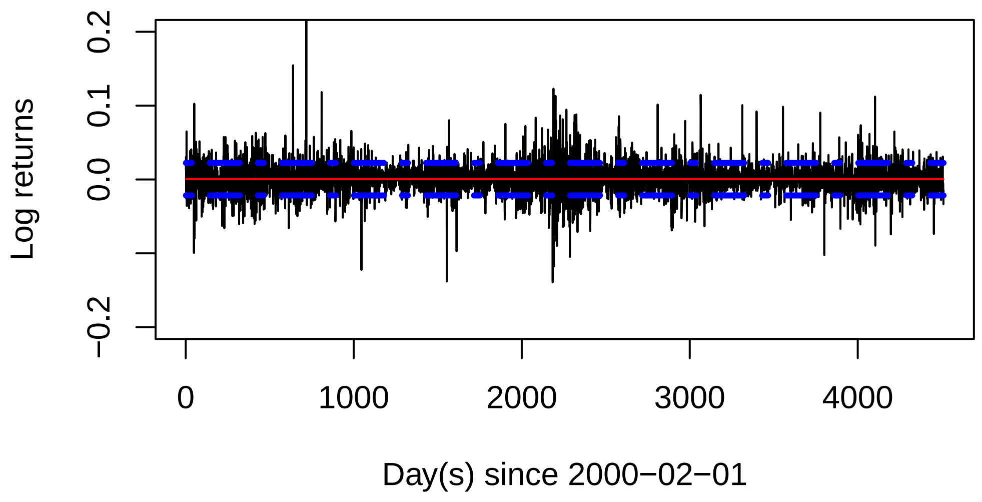
```r
set.seed(1)
vX <- runif(n = 250) # 250 random values
dMin <- vX[1]
for (iN in 1:length(vX)) {
  if (vX[iN] < dMin) dMin <- vX[iN] # We loop over the vector vX and assign the value at the
}
dMin # Print the minimum value
#> [1] 0.01307758
min(vX) # Verifying the result
#> [1] 0.01307758
```

14.10 (10)

```
vX <- rep(0L, 100)

for (i in 1:100) {

  vX[seq_along(vX) %% i == 0] <- bitwXor(vX[seq_along(vX) %% i == 0], 1) # Using bitwise log

  # This also works
  # vX[seq_along(vX) %% i == 0] <- !vX[seq_along(vX) %% i == 0]

}

vX
#>   [1] 1 0 0 1 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0
#>  [38] 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0
#>  [75] 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1

which(vX == 1) # Perfect squares! :-)
#>  [1]   1   4   9  16  25  36  49  64  81 100
```

14.11 (11)

```
mMaerskData <- read.csv("MAERSK-B.CO.csv", header = TRUE, sep = ",", na.strings = "null")

vY <- diff(log(na.omit(mMaerskData[, "Adj.Close"]))) # Log-returns

# Line plot
plot(1:length(vY), vY, type = "l", ylim = c(-0.2, 0.2), ylab = "Log returns", xlab = paste0(

# Mean
lines(1:length(vY), rep(mean(vY), length(vY)), col = "red", lwd = "1")

# Confidence bands
lines(1:length(vY), rep((mean(vY) + sqrt(var(vY))), length(vY)), col = "blue", lwd = "3", lt
lines(1:length(vY), rep((mean(vY) - sqrt(var(vY))), length(vY)), col = "blue", lwd = "3", lt
```


Log returns
0.2
0.1
0.0
−0.2
0
1000
2000
3000
4000
Day(s) since 2000−02−01

15 Exercise set 3

15.1 (1)

```r
euclidean_length <- function(vInput) {
  return(sqrt(sum(vInput^2)))
}

vX <- 1:10
euclidean_length(vX)
#> [1] 19.62142
```

15.2 (2)

-
-
-
-

```r
set.seed(999)
iT <- 420
iN <- 10
A <- matrix(round(runif(iN^2)), iN, iN)

## First solution (using a triple loop)
game_of_life_1 <- function(mInput, iT) {
  iR <- nrow(mInput)
  iC <- ncol(mInput)
  mNew <- matrix(0, iR, iC)
  for (t in 1:iT) {
    # Loop over rows and columns, i.e., each value in the matrix separately
    for (i in 1:iR) {
      for (j in 1:iC) {
        # Use the modulus operator for handling the borders
        # We need (i - 2) due to R indexing beginning at 1
        iNeighbourCount <- mInput[(i - 2) %% iR + 1, (j - 2) %% iC + 1] +
                           mInput[(i - 2) %% iR + 1, (j - 1) %% iC + 1] +
                           mInput[(i - 2) %% iR + 1, j %% iC + 1] +

                           mInput[(i - 1) %% iR + 1, (j - 2) %% iC + 1] +
                           mInput[(i - 1) %% iR + 1, j %% iC + 1] +

                           mInput[i %% iR + 1, (j - 2) %% iC + 1] +
                           mInput[i %% iR + 1, (j - 1) %% iC + 1] +
                           mInput[i %% iR + 1, j %% iC + 1]

        if (mInput[i, j] == 1) { # If alive
          if (iNeighbourCount < 2 || iNeighbourCount > 3) {
            mNew[i, j] <- 0
          } else {
            mNew[i, j] <- 1
          }
        } else { # If dead
          if (iNeighbourCount == 3) {
            mNew[i, j] <- 1
          } else {
            mNew[i, j] <- 0
          }
        }
      }
    }
    mInput <- mNew
  }
  return(mInput)
}

print(game_of_life_1(A, 420))
#>       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
#>  [1,]    0    0    0    0    0    0    0    0    0     0
#>  [2,]    0    0    0    0    0    0    0    0    0     0
#>  [3,]    0    0    0    0    0    0    0    0    0     0
#>  [4,]    0    0    0    0    1    1    0    0    0     0
#>  [5,]    0    0    0    1    0    0    1    0    0     0
#>  [6,]    0    0    0    1    0    0    1    0    0     0
#>  [7,]    0    0    0    0    1    1    0    0    0     0
#>  [8,]    0    0    0    0    0    0    0    0    0     0
#>  [9,]    0    0    0    0    0    0    0    0    0     0
#> [10,]    0    0    0    0    0    0    0    0    0     0


## Second solution (much faster)
game_of_life_2 <- function(mInput, iT) {
  iR <- nrow(mInput)
  iC <- ncol(mInput)
  for (t in 1:iT) {
    mPadded <- matrix(0, iR + 2, iC + 2) # Empty matrix with a border
    mPadded[2:(iR + 1), 2:(iC + 1)] <- mInput # Middle part

    mPadded[1, 2:(iC + 1)] <- mInput[iR, ] # Top row
    mPadded[iR + 2, 2:(iC + 1)] <- mInput[1, ] # Bottom row
    mPadded[2:(iR + 1), 1] <- mInput[, iC] # Left column
    mPadded[2:(iR + 1), iC + 2] <- mInput[, 1] # Right column

    mPadded[1, 1] <- mInput[iR, iC] # Top left value
    mPadded[1, iC + 2] <- mInput[iR, 1] # Top right value
    mPadded[iR + 2, 1] <- mInput[1, iC] # Bottom left value
    mPadded[iR + 2, iC + 2] <- mInput[1, 1] # Bottom right value

    # Now, we can create a neighbour count by shifting the padded matrix and adding the 1's
    mNeighbourCount <- mPadded[1:iR, 1:iC] +
                       mPadded[2:(iR + 1), 1:iC] +
                       mPadded[3:(iR + 2), 1:iC] +
                       mPadded[1:iR, 2:(iC + 1)] +
                       mPadded[3:(iR + 2), 2:(iC + 1)] +
                       mPadded[1:iR, 3:(iC + 2)] +
                       mPadded[2:(iR + 1), 3:(iC + 2)] +
                       mPadded[3:(iR + 2), 3:(iC + 2)]

    # And we can use the neighbour count matrix for indexing
    mInput <- (mInput & (mNeighbourCount == 2 | mNeighbourCount == 3)) | (!mInput & mNeighbou
  }
  return(mInput)
}

# Use the plus-sign to return the matrix as integer booleans
print(+game_of_life_2(A, 420))
#>       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
#>  [1,]    0    0    0    0    0    0    0    0    0     0
#>  [2,]    0    0    0    0    0    0    0    0    0     0
#>  [3,]    0    0    0    0    0    0    0    0    0     0
#>  [4,]    0    0    0    0    1    1    0    0    0     0
#>  [5,]    0    0    0    1    0    0    1    0    0     0
#>  [6,]    0    0    0    1    0    0    1    0    0     0
#>  [7,]    0    0    0    0    1    1    0    0    0     0
#>  [8,]    0    0    0    0    0    0    0    0    0     0
#>  [9,]    0    0    0    0    0    0    0    0    0     0
#> [10,]    0    0    0    0    0    0    0    0    0     0


## A quick speed comparison of the two solutions shows a huge increase in performance
suppressMessages(library(microbenchmark))
#> Warning: pakke 'microbenchmark' blev bygget under R version 4.3.3
microbenchmark(game_of_life_1(A, 420), +game_of_life_2(A, 420))
#> Unit: milliseconds
#>                      expr     min       lq      mean   median      uq      max
#>   game_of_life_1(A, 420) 95.1552 97.57205 99.918607 98.21435 99.5306 139.9134
#>  +game_of_life_2(A, 420)  6.5912  6.80370  7.754339  7.05740  8.0493  34.1505
#>  neval
#>    100
#>    100
```

15.3 (3)

```r
fbinom <- function(n, r) {
  if (n == r | r == 0) {
    return(1)
  } else {
    return(fbinom(n - 1, r - 1) + fbinom(n - 1, r))
  }
}
```

15.4 (4)

```r
fb <- function(n) {
  if (n == 1 || n == 2) {
    return(1)
  } else {
    return(fb(n - 1) + fb(n - 2))
  }
}
fb(8)
```

15.5 (5)

15.6 (6)

```
m1 <- matrix(round(rnorm(100, mean = 20, sd = 10)), 10, 10)
m1
#>       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
#>  [1,]   29   35   18    9   21   29   24   17   12    28
#>  [2,]   18   24   23   11   27   37    2   32   13    13
#>  [3,]   18   25    1   16   25   42    2   19   10    13
#>  [4,]   11   29    1   31    5   29   20   19   28    15
#>  [5,]   25   28   26   -2   28   10    0    8   33    32
#>  [6,]   24   21    8   35    8   23   28   19   21    16
#>  [7,]   10   13   23   35   21   29   11    7   33    -2
#>  [8,]   12   24   13   10    5   12   13   25   35    11
#>  [9,]   21   13   24   32    0   -6   11   15   26    16
#> [10,]    5   36   27   28   14   22   10   28   12     2
m2 <- apply(m1, c(1, 2), function(x) x %% 3)
m3 <- matrix(m1 %% 3, 10, 10) # Not the apply() function, but it also works
m2
```

```
#>       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
#>  [1,]    2    2    0    0    0    2    0    2    0     1
#>  [2,]    0    0    2    2    0    1    2    2    1     1
#>  [3,]    0    1    1    1    1    0    2    1    1     1
#>  [4,]    2    2    1    1    2    2    2    1    1     0
#>  [5,]    1    1    2    1    1    1    0    2    0     2
#>  [6,]    0    0    2    2    2    2    1    1    0     1
#>  [7,]    1    1    2    2    0    2    2    1    0     1
#>  [8,]    0    0    1    1    2    0    1    1    2     2
#>  [9,]    0    1    0    2    0    0    2    0    2     1
#> [10,]    2    0    0    1    2    1    1    1    0     2
```

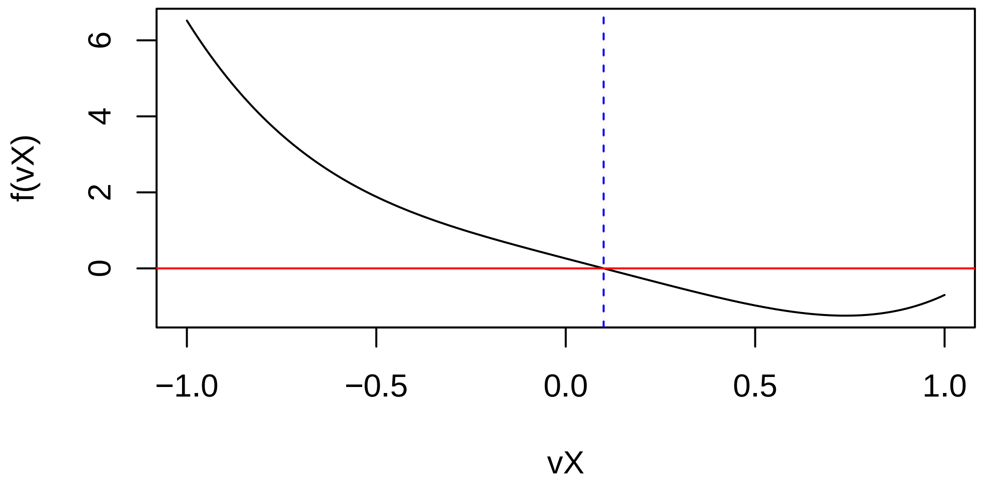
16 Exercise set 4

16.1 (1)

(a)
(b)
(c)
(d)

```r
f <- function(dX) {
  dOut <- -2.6 * (dX - 0.1) + 2.5 * (dX - 0.1)^4
  return(dOut)
}

fSecant <- function(f, dX0, dX1, dTol = 1e-9, max.iter = 1000, ...) {
  iter <- 0
  dX2 <- dX1
  while ((abs(f(dX2, ...)) > dTol) && (iter < max.iter)) {
    dX2 <- dX1 - f(dX1, ...) * ((dX0 - dX1) / (f(dX0, ...) - f(dX1, ...)))
    dX0 <- dX1
    dX1 <- dX2
    iter <- iter + 1
    cat("At iteration ", iter, "value of x is: ", dX1, "\n")
  }
  if (abs(f(dX2, ...)) > dTol) {
    return(list(root = NULL, f.root = NULL, iter = iter, "Algorithm failed to converge. Maxin
  } else {
    return(list(root = dX2, f.root = f(dX2), iterations = iter, "Convergence reached."))
  }
}

root <- fSecant(f, dX0 = -0.5, dX1 = 0.5)
#> At iteration  1 value of x is:  0.1587413
#> At iteration  2 value of x is:  0.09544817
#> At iteration  3 value of x is:  0.1000008
#> At iteration  4 value of x is:  0.1
root
#> $root
#> [1] 0.1
#>
#> $f.root
#> [1] -1.941947e-13
#>
#> $iterations
#> [1] 4
#>
#> [[4]]
#> [1] "Convergence reached."
uniroot(f, interval = c(-1,1))
#> $root
#> [1] 0.1000007
#>
#> $f.root
#> [1] -1.764589e-06
#>
#> $iter
#> [1] 6
#>
#> $init.it
#> [1] NA
#>
#> $estim.prec
#> [1] 6.103516e-05

vX <- seq(-1, 1, 0.01)
plot(vX, f(vX), type = "l")
```

```r
abline(h = 0, col = "red")
abline(v = root["root"], col = "blue", lty = 2)
```



16.2 (2)

(a)
(b)

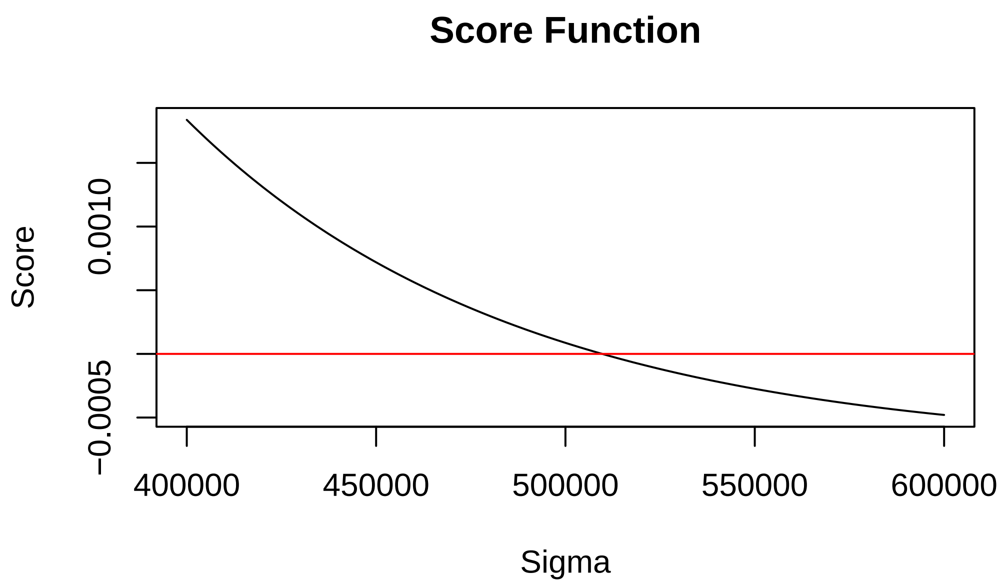
(c)
(d)

(a)
(b)

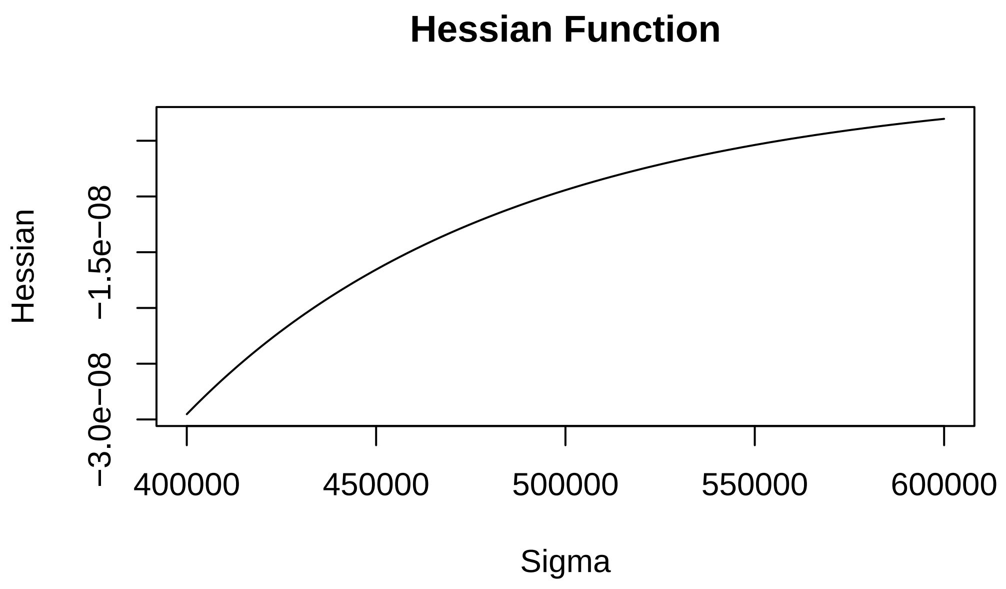
(c)

```
f <- function(vY, dSigma, dC) {
  return((1 / dSigma) * ((dnorm(vY / dSigma)) / (pnorm(dC / dSigma))))
}

fScore <- function(vY, dSigma, f, dC) {
  return(((-1 * length(vY)) / dSigma) + (1 / dSigma^3) * (sum(vY^2)) + ((dC * length(vY)) /
}

fHessian <- function(vY, dSigma, f, dC) {
  return(length(vY) / dSigma^2 - (3 / dSigma^4) * sum(vY^2) + (dC^3 * length(vY) / dSigma^4 
}

fTrunc <- function(f, fScore, fHessian, vY, start.val = sqrt(var(vY)), n.max = 200, dTol = 1e

  n <- 0
  dParam <- start.val

  # Keep updating until stopping criterion or max iterations reached
  while ((abs(fScore(vY, dParam, f, ...)) > dTol) && (n < n.max)) {
    # Newton-Raphson updating
    dParam <- dParam - fScore(vY, dParam, f, ...) / fHessian(vY, dParam, f, ...)
    n <- n + 1
    #cat("At iteration", n, "the value of the parameter is:", dParam, "\n")
  }

  if (n == n.max) {
    return(list(
      param = NULL,
      log.likelihood = NULL,
      score = NULL,
      hessian = NULL,
      iter = n,
      msg = "Algorithm failed to converge. Maximum iterations reached.")
    )
  } else {
    return(list(
      param = dParam,
      log.likelihood = sum(log(f(vY, dParam, ...))),
      score = fScore(vY, dParam, f, ...),
      hessian = fHessian(vY, dParam, f, ...),
      iter = n,
      msg = "Algorithm converged")
    )
  }

}

vX <- readRDS("QPE_income1.R")
results <- fTrunc(f, fScore, fHessian, vX, dC = 1250000)
results
#> $param
#> [1] 509699.8
#>
#> $log.likelihood
#> [1] -19280.04
#>
#> $score
#> [1] -5.692061e-19
#>
#> $hessian
#> [1] -8.458564e-09
#>
#> $iter
#> [1] 4
#>
#> $msg
#> [1] "Algorithm converged"

# Plot Score and Hessian
sigmaVals <- seq(400000, 600000, length.out = 200)
scoreVals <- sapply(sigmaVals, function(sigma) fScore(vX, sigma, f, dC = 1250000))
hessianVals <- sapply(sigmaVals, function(sigma) fHessian(vX, sigma, f, dC = 1250000))

# Plot Score
plot(sigmaVals, scoreVals, type = "l", main = "Score Function", xlab = "Sigma", ylab = "Score
```

```r
abline(h = 0, col = "red")
```



```r
# Plot Hessian
plot(sigmaVals, hessianVals, type = "l", main = "Hessian Function", xlab = "Sigma", ylab = "I
abline(h = 0, col = "red")
```


Hessian Function
Hessian
−3.0e−08
−1.5e−08
400000
450000
500000
550000
600000
Sigma

17 Exercise set 5

17.1 (1)

(a)
(b)
(c)
(d)

(e)
(f)

(a)
(b)
(c)
(d)
(e)

```r
fLogit <- function(vY, mX, add.constant = TRUE, init.vals = NULL, max.iter = 200, dTol = 1e-

  # probability function
  p <- function(vZ) {
    return(1 / (1 + exp(-vZ)))
  }

  # avg log-likelihood function
  f <- function(vY, mX, vB) {
    return(mean(vY * log(p(mX %*% vB)) + (1 - vY) * log(1 - p(mX %*% vB))))
  }

  fScore <- function(vY, mX, vB) {
    return(t(mX) %*% (vY - as.numeric(p(mX %*% vB))) / nrow(mX))
  }

  fHessian <- function(vY, mX, vB) {
    return(-(t(mX) %*% diag(as.numeric(p(mX %*% vB) * (1 - as.numeric(p(mX %*% vB))))) %*% m
  }

  if (add.constant == TRUE) {
    mX <- cbind(1, mX)
  }
  if (is.null(init.vals)) init.vals <- rep(0, ncol(mX))

  i <- 0
  vB <- init.vals

  # Keep updating until stopping criterion or max iterations reached
  while ((max(abs(fScore(vY, mX, vB))) > dTol) && (i < max.iter)) {
    # Newton-Raphson updating
    vB <- vB - solve(fHessian(vY, mX, vB), fScore(vY, mX, vB))
    i <- i + 1
    #cat("At iteration", n, "the value of the parameter is:", dParam, "\n")
  }

  if (i == max.iter) {
    return(list(
      beta_hat = NULL,
      log_likelihood_opt = NULL,
      score_opt = NULL,
      hessian_opt = NULL,
      log_likelihood_null = NULL,
      predicted_probabilities = NULL,
      iterations = i,
      msg = "Algorithm failed to converge. Maximum iterations reached.")
    )
  } else {
    return(list(
      beta_hat = vB,
      log_likelihood_opt = f(vY, mX, vB),
      score_opt = fScore(vY, mX, vB),
      hessian_opt = fHessian(vY, mX, vB),
      log_likelihood_null = f(vY, mX, rep(0, ncol(mX))),
      predicted_probabilities = p(mX %*% vB),
      iterations = i,
      msg = "Algorithm converged")
    )
  }
}
```

17.2 (2)

```
set.seed(123)
n <- 2000
p <- 10
mX <- matrix(rnorm(n*p), n, p)
```

```
vB.actual <- matrix(1:p, p, 1)
vY.star <- mX %*% vB.actual + rnorm(2000, 0, 1)
vY.actual <- vY.star > 0

vResults <- fLogit(vY.actual, mX)
glm_fit <- glm(vY.actual ~ mX, family = binomial)
#> Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred
glm_fit$coefficients
#> (Intercept)         mX1         mX2         mX3         mX4         mX5 
#>  -0.1927213   1.8925010   3.4933457   4.8288383   6.4561863   8.4108110 
#>         mX6         mX7         mX8         mX9        mX10 
#>  10.2085496  11.8431135  13.2432309  14.8829116  16.5606978 
vResults$beta_hat
#>             [,1]
#>  [1,] -0.1927213
#>  [2,]  1.8925010
#>  [3,]  3.4933457
#>  [4,]  4.8288384
#>  [5,]  6.4561864
#>  [6,]  8.4108112
#>  [7,] 10.2085499
#>  [8,] 11.8431138
#>  [9,] 13.2432312
#> [10,] 14.8829119
#> [11,] 16.5606982
```

18 Exercise set 6

(1)
(2)

(3)

```cpp
//[[Rcpp::depends(RcppArmadillo)]]
#include <RcppArmadillo.h>
using namespace Rcpp;
using namespace arma;

//initialize auxiliary functions: prb, LL, score, hessian

// mX should be initialized with mat
vec fP(vec vBeta, mX){
  return(1 / (1 + exp( -mX * vBeta )));
}
// vP should be initialized with vec, and also it should return a double
vec fLnL(vec vBeta, vec vY, mat mX){
  vP = fP(vBeta,mX);
  vec vLnL = mean( vY % log(vP) + (ones<vec>(vY.n_elem) - vY) * log(ones<vec>(vP.n_elem) - vl
  return vLnL;
}
// vScore is not divided by the number of rows in mX
vec fScore(vec vBeta, vec vY, mat mX){
  vec vP = fP(vBeta,mX);
  vec vScore = trans(mX)*(vY - vP);
  return vScore;
}
//
mat fHessian(vec vBeta, vec vY, mat mX){
  int iN = vY.n_elem;
  vec vP = fP(vBeta,mX);
  mat mP = zeros<mat>(iN,iN);

  mP.diag(0) = vP % (ones<vec>(iN) - vP);
  mat mHessian = -trans(mX)*mP*mX / iN;

  return mHessian;
}

//[[Rcpp::export]]
List fLogitCpp(vec vY, mat mX, vec vBeta0, // vBeta should be initialized with default value
               bool constant = true,
               double dTol = 1e-9,
               bool imax = 200){ // this should be an int

  List lOut;
  int iN = vY.n_elem;

  if (constant == true){
    vec vOnes = zeros<vec>(iN); // this should be ones
    vBeta0 = join_cols(zeros<vec>(1),vBeta0);
    mX = join_rows(vOnes,mX);
    //Rcout << "Note: A constant has been added to data matrix mX" << std::endl;
  }

  vec vBeta = vBeta0;
  vec vScore = fScore(vBeta,vY,mX);
  mat mHessian = fHessian(vBeta,vY,mX);

  int i = 0;
  while ((max(abs(vScore)) < dTol) & (i < imax)) { // first sign should be flipped
    //Rcout << i << "\n";
    vBeta = vBeta - solve(fHessian(vBeta,vY,mX),fScore(vBeta,vY,mX));
    vScore = fScore(vBeta,vY,mX);
    mHessian = fHessian(vBeta0,vY,mX); // this should be vBeta

    i++;
  }
  if (i == imax){
    Rcout << "newton failed to converge" << std::endl;
    return lOut;
  } else {
   // Rcout << "Convergence achieved after " << i << " iterations" << std::endl
   //       << "due to max(abs(score)) <" << dTol << std::endl;
    lOut["coefficients"] = vBeta;
    lOut["logLik0"] = fLnL(zeros<vec>(vBeta.n_elem),vY,mX);
    lOut["logLik"] = fLnL(vBeta,vY,mX);
    lOut["score"] = vScore;
    lOut["hessian"] = mHessian;
    lOut["phat"] = fP(vBeta,mX);

    return lOut;
  }
}
```

```
suppressMessages(library(Rcpp))
#> Warning: pakke 'Rcpp' blev bygget under R version 4.3.3
suppressMessages(library(RcppArmadillo))
#> Warning: pakke 'RcppArmadillo' blev bygget under R version 4.3.3
sourceCpp("fLogitCpp.cpp")

set.seed(123)
n <- 2000
p <- 10
mX <- matrix(rnorm(n*p), n, p)
vB.actual <- matrix(1:p, p, 1)
vY.star <- mX %*% vB.actual + rnorm(2000, 0, 1)
vY.actual <- vY.star > 0

vResults <- fLogitCpp(vY.actual, mX, vBeta = rep(0, ncol(mX)))
vResults$coefficients
#>              [,1]
#>  [1,] -0.1927213
#>  [2,]  1.8925004
#>  [3,]  3.4933444
#>  [4,]  4.8288365
#>  [5,]  6.4561838
#>  [6,]  8.4108079
#>  [7,] 10.2085458
#>  [8,] 11.8431091
#>  [9,] 13.2432260
#> [10,] 14.8829059
#> [11,] 16.5606916
```

19 Ordinary exam 2020

19.1 Problem 1:

1.

```r
set.seed(1337)
dfData <- data.frame(
  rvnorm = rnorm(100, 10, sqrt(100)),
  rvunif = runif(100)
)

colMeans(dfData)
#>     rvnorm     rvunif 
#> 12.3705575  0.4927623 
apply(dfData, 2, sd)
#>     rvnorm     rvunif 
#> 10.6541506  0.2865665 
```

2.

```r
mA <- matrix(rnorm(10000), nrow = 100, ncol = 100)
```

```
set.seed(1337)
mA <- matrix(rnorm(10000), nrow = 100, ncol = 100)
sum(mA > 0)
#> [1] 4968

indices <- seq(1, nrow(mA))
mB <- mA[indices %% 2 != 0, indices %% 5 != 0]
mB[, 2]
#>  [1]  0.20997680  1.37178507 -0.20485705 -0.80947218  0.06873847 -0.46916210
#>  [7] -0.73977783  0.37774034 -0.90396001 -0.23914384 -0.31475060 -0.54221576
#> [13] -0.13401439 -0.44703293 -0.58097800 -0.64590441 -0.43772050  1.70627166
#> [19]  1.28239811  3.10367380 -0.25455142  0.11397658  0.28331947 -0.25257320
#> [25] -0.10787613 -1.68283890  0.59339153  1.42397245 -0.16762034 -0.28949467
#> [31] -0.38718739 -0.86600469 -0.16801229 -0.05423047 -0.15457321  1.20115726
#> [37]  0.89901990 -0.81578612  0.56388549 -0.52282158  0.55286246 -0.05125182
#> [43] -0.64952770  1.54842553 -1.13708264 -0.21124735 -1.45882620 -0.13385242
#> [49]  0.31359063 -0.94320880
```

3.

```
set.seed(1337)
MonteCarlo.Integration <- function(f, n, a, b) {

  U <- runif(n, min = a, max = b)
  return( (b-a)*mean(f(U)) )

}
MonteCarlo.Integration(function(x) (x^3 - x^2 + 9), 1e7, 3, 20)
#> [1] 37477.23
```

4.

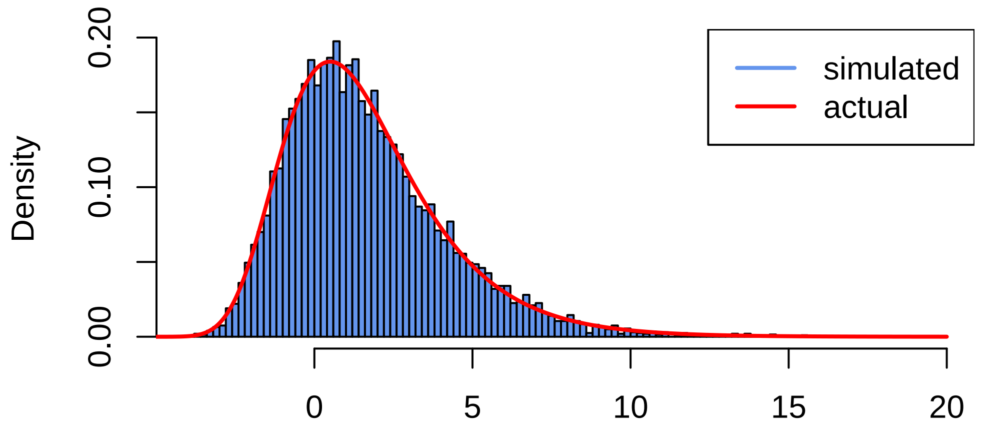
```r
diag.covar <- function(mX, mY) {
      varcov <- 0
      for(j in 1:10) {
        dsum <- 0
        for(i in 1:10) {
          dsum <- dsum + (mX[i,j] - mean(mX[,j]))*(mY[i,j] - mean(mY[,j]))
        }
        varcov[j] <- dsum/(dim(mX)[1]-1)
      }
      return(varcov)
    }
```

```r
mX <- matrix(rnorm(100), nrow = 10, ncol = 10)
mY <- matrix(rnorm(100), nrow = 10, ncol = 10)
```

```r
set.seed(1337)
diag.covar <- function(mX, mY) {
  varcov <- 0
  for(j in 1:10) {
    dsum <- 0
    for(i in 1:10) {
      dsum <- dsum + (mX[i,j] - mean(mX[,j]))*(mY[i,j] - mean(mY[,j]))
    }
    varcov[j] <- dsum/(dim(mX)[1]-1)
  }
  return(varcov)
}

diag.covarVec <- function(mX, mY) {
  mSum <- (t(mX) - colMeans(mX)) * (t(mY) - colMeans(mY))
  varcov <- rowSums(mSum) / (dim(mX)[1]-1)
  return(varcov)
}

mX <- matrix(rnorm(100), nrow = 10, ncol = 10)
mY <- matrix(rnorm(100), nrow = 10, ncol = 10)

diag.covar(mX, mY)
#>  [1] -0.002280477  0.133113794  0.354419701 -0.881358047 -0.134682108
#>  [6] -0.436054671 -0.425000809 -0.119402997 -0.309367075  0.287553344
diag.covarVec(mX, mY)
#>  [1] -0.002280477  0.133113794  0.354419701 -0.881358047 -0.134682108
#>  [6] -0.436054671 -0.425000809 -0.119402997 -0.309367075  0.287553344

suppressMessages(library(microbenchmark))
#> Warning: pakke 'microbenchmark' blev bygget under R version 4.3.3
microbenchmark(diag.covar(mX, mY), diag.covarVec(mX, mY))
#> Unit: microseconds
#>                   expr   min      lq     mean median      uq    max neval
#>     diag.covar(mX, mY) 992.6 1015.60 1077.422 1042.6 1071.95 2682.9   100
#>  diag.covarVec(mX, mY)  18.9   20.25   72.697   23.8   29.90 4513.0   100
```

5.

```r
set.seed(1337)
Gumbel <- function(mu, beta, size) {
  U <- runif(size)
  return(mu - beta * log(-log(U)))
}

vX <- Gumbel(0.5, 2, 10000)
hist(vX,
     freq = FALSE,
     breaks = 141,
     col = "cornflowerblue",
     xlab = "",
     ylab = "Density",
     main = "",
     xlim = c(min(vX), max(vX)))

vInput <- seq(-5, 20, 0.05)

fGumbelPdf <- function(mu, beta, x) {
  return(1/beta * exp(-((x-mu)/beta + exp(-(x-mu)/beta))))
}

lines(vInput, fGumbelPdf(0.5, 2, vInput), type = "l", col = "red", lwd = 2)

legend("topright",
       legend = c("simulated", "actual"),
       col = c("cornflowerblue", "red"),
       lwd = 2)
```

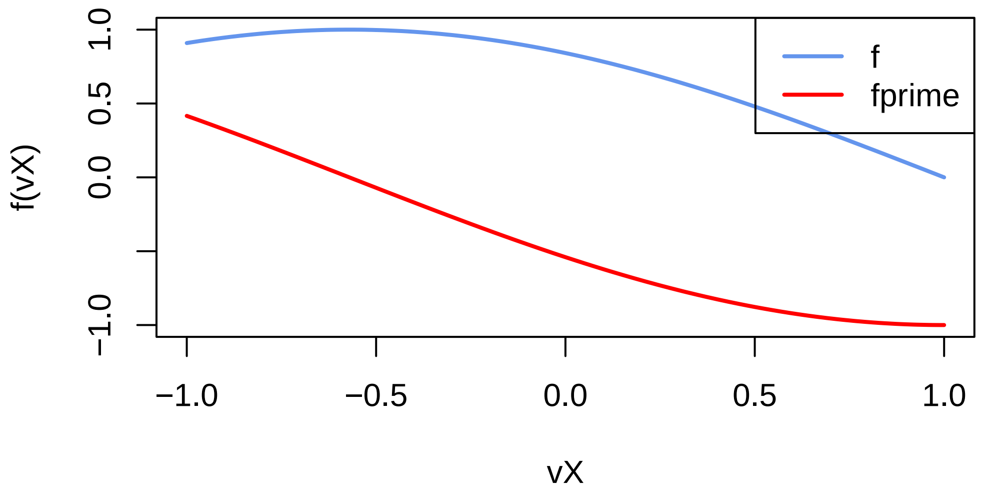


19.2 Problem 2:

1.

```r
f <- function(x) {
  return(sin(x + pi - 1))
}

fPrime <- function(x) {
  return(cos(x + pi - 1))
}
```

```r
fDoublePrime <- function(x) {
  return(-sin(x + pi - 1))
}

vX <- seq(-1, 1, 0.01)
plot(vX, f(vX), type = "l", col = "cornflowerblue", lwd = 2, ylim = c(-1, 1))
lines(vX, fPrime(vX), type = "l", col = "red", lwd = 2)

legend("topright",
       legend = c("f", "fprime"),
       col = c("cornflowerblue", "red"),
       lwd = 2)
```



2.

(a)
(b)
(c)
(d)

```r
NM <- function(f, f_prime, f_sec, dX0, dTol = 1e-9, n.max = 1000){
  dX <- dX0
  fx <- f(dX)
  fpx <- f_prime(dX)
  fsx <- f_sec(dX)
  n <- 0
  while ((abs(fpx) > dTol) && (n < n.max)) {
    dX <- dX - fpx/fsx
    fx <- f(dX)
    fpx <- f_prime(dX)
    fsx <- f_sec(dX)
    n <- n + 1
  }
  if (n == n.max) {
    return(list(
      maximizer = dX,
      function.val = f(dX),
      n.iter = n,
      msg = "Failed to converge. Max number of iterations reached."
    ))
  } else {
    return(list(
      maximizer = dX,
      function.val = f(dX),
      n.iter = n,
      msg = "Convergence reached."
    ))
  }
}
```

3.

(a)
(b)
(c)

```
// [[Rcpp::depends(RcppArmadillo)]]
#include <RcppArmadillo.h>

using namespace arma;
using namespace Rcpp;

// [[Rcpp::export]]
List bisection_cpp(Function f, double dXLeft = -1.0, double dXRight = 1.0, double dTol = 0.00
  if (dXLeft >= dXRight) {
    stop("Starting conditions not met");
  }
  double fLeft = as<double>(f(dXLeft));
  double fRight = as<double>(f(dXRight));
  if (fLeft == 0) {
    List lOut;
    lOut["root"] = dXLeft;
    lOut["func_at_root"] = fLeft;
    lOut["num_ite"] = 0;
    return lOut;
  } else if (fRight == 0) {
    List lOut;
    lOut["root"] = dXRight;
    lOut["func_at_root"] = fRight;
    lOut["num_ite"] = 0;
    return lOut;
  } else if (fLeft * fRight > 0) {
    stop("error: f(x.l)*f(x.r) > 0");
  }

  int iter = 0;
  while ((dXRight - dXLeft) > dTol && (iter < maxIter)) {
    double dXMid = (dXLeft + dXRight)/2;
    double fMid = as<double>(f(dXMid));
    if (fMid == 0) {
      return(dXMid);
    } else if (fLeft * fMid < 0) {
      dXRight = dXMid;
      fRight = fMid;
    } else {
      dXLeft = dXMid;
      fLeft = fMid;
    }
    iter = iter + 1;
  }

  List lOut;
  lOut["root"] = (dXLeft + dXRight)/2;
  lOut["func_at_root"] = as<double>(f((dXLeft + dXRight)/2));
  lOut["num_ite"] = iter;
  return lOut;
}
```

```
suppressMessages(library(Rcpp))
#> Warning: pakke 'Rcpp' blev bygget under R version 4.3.3
suppressMessages(library(RcppArmadillo))
#> Warning: pakke 'RcppArmadillo' blev bygget under R version 4.3.3
sourceCpp('exam2020cpp.cpp')
```

4.

```
NM(f, fPrime, fDoublePrime, 0)$maximizer
#> [1] -0.5707963
bisection_cpp(fPrime, dXLeft = -1, dXRight = 1)$root
#> [1] -0.5707703
optim(0, f, method = "L-BFGS-B", lower = -1, upper = 1, control=list(fnscale=-1))$par
#> [1] -0.5707963
uniroot(fPrime, c(-1, 1))$root
#> [1] -0.5707835

suppressMessages(library(microbenchmark))
microbenchmark(
  NM = NM(f, fPrime, fDoublePrime, 0)$maximizer,
  Bisection = bisection_cpp(fPrime, dXLeft = -1, dXRight = 1)$root,
  Optim = optim(0, f, method = "L-BFGS-B", lower = -1, upper = 1, control=list(fnscale=-1))$p
  UniRoot = uniroot(fPrime, c(-1, 1))$root
)
#> Unit: microseconds
#>       expr  min    lq   mean median    uq    max neval
#>         NM  8.0  9.05 10.605  10.00 10.95   40.3   100
#>  Bisection 20.5 23.70 38.637  25.35 27.35 1215.4   100
#>      Optim 35.2 39.60 46.414  41.60 46.10  214.8   100
#>    UniRoot 33.3 37.35 43.675  40.00 44.90  139.2   100
```

5.

20 Ordinary exam 2021

20.1 Problem 1:

1.

```r
set.seed(134)

fn_slowCauchy <- function(sigma, samples, length) {
  mX <- matrix(0, nrow = length, ncol = samples)
  U <- runif(samples * length)
  mU <- matrix(U, nrow = length, ncol = samples)
  for (i in 1:samples) {
    for (j in 1:length) {
      mX[j, i] <- tan(pi * (mU[j, i] - 1/2)) * sigma
    }
  }
  return(mX)
}

mX <- fn_slowCauchy(2, 5000, 1000)
```

2.

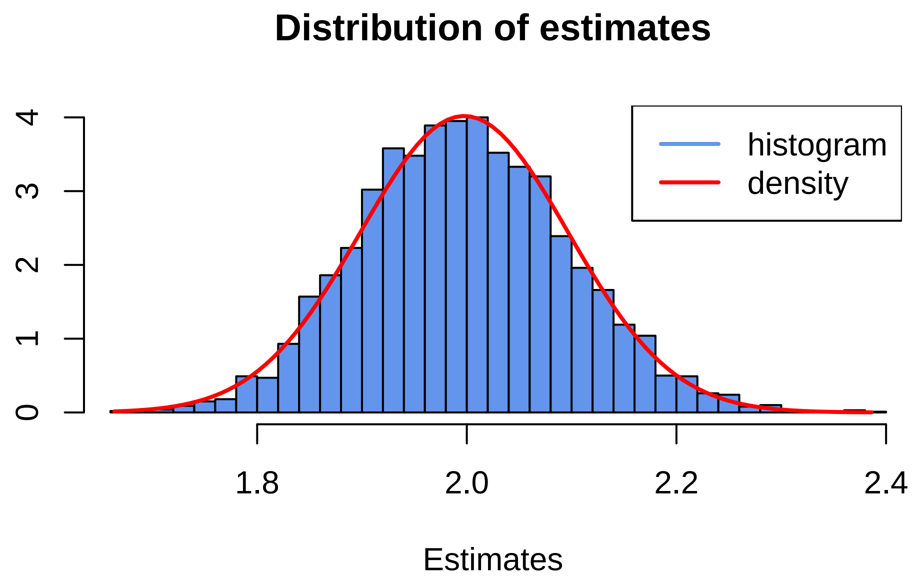
```r
set.seed(134)

fn_fastCauchy <- function(sigma, samples, length) {
  U <- runif(samples * length)
  vCauchy <- tan(pi * (U - 1/2)) * sigma
  mY <- matrix(vCauchy, length, samples, byrow = FALSE)
  return(mY)
}

mY <- fn_fastCauchy(2, 5000, 1000)

all.equal(mX, mY)
#> [1] TRUE
```

3.

```r
fn_estimateScale <- function(mX) {
  calc_scale <- function(vX) {
    vY <- ifelse(vX < 0, vX * -1, vX)
    vY <- sort(vY, decreasing = FALSE)
    return(vY[ceiling(length(vY) / 2)])
  }
  return(apply(mX, 2, calc_scale))
}

vSigma <- fn_estimateScale(mX)
```

4.

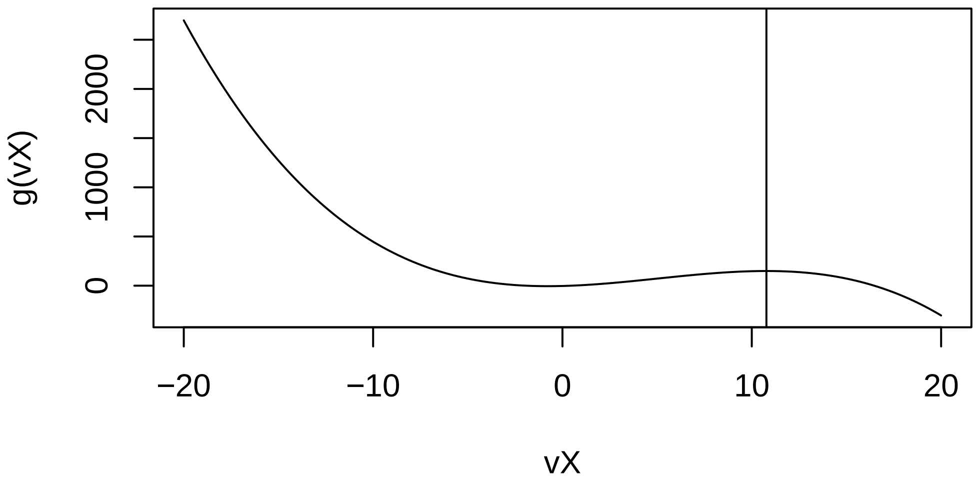
```r
hist(vSigma,
     freq = F,
     breaks = 50,
     col = "cornflowerblue",
     xlab = "Estimates",
     ylab = "",
     main = "Distribution of estimates",
     xlim = c(min(vSigma), max(vSigma)))

curve(dnorm(x, mean(vSigma), sd(vSigma)),
      from = min(vSigma),
      to = max(vSigma),
      col = "red",
      lwd = 2,
      add = TRUE)

legend("topright", legend = c("histogram", "density"), col = c("cornflowerblue", "red"), lwd
```



5.

(a)
(b)
(c)

```r
g <- function(x) {
  return(-0.2 * x^3 + 3 * x^2 + 5 * x - 3)
}

gPrime <- function(x) {
  return(-0.6 * x^2 + 6 * x + 5)
}

fn_findMax <- function(f, gr, dX0 = 0, dTol = 0.00001, max.iter = 200) {
  dXn <- dX0 + sign(gr(dX0)) / sqrt(1)
  n <- 1
  while (abs(gr(dXn)) > dTol & n < max.iter) {
    dXn <- dXn + sign(gr(dXn)) / sqrt(n + 1)
    n <- n + 1
  }

  if (n == max.iter) {
    return(list(
      param = dXn,
      max.val = f(dXn),
      num.iter = n,
      msg = "Maximum number of iterations reached. Stopping..."
    ))
  } else {
    return(list(
      param = dXn,
      max.val = f(dXn),
      num.iter = n,
      msg = "Convergence reached."
    ))
  }
}

lSolution <- fn_findMax(g, gPrime)
lSolution
#> $param
#> [1] 10.7735
#>
#> $max.val
#> [1] 148.98
#>
#> $num.iter
#> [1] 101
#>
#> $msg
#> [1] "Convergence reached."

vX <- seq(-20, 20, by = 0.05)
plot(vX, g(vX), type = "l")
abline(v = lSolution$param, col = "black")
```



20.2 Problem 2: (Constrained) Optimization, C++, and Packaging

1.

```r
set.seed(123)
genData <- function() {
  # --- Generate mDATA.r ---
  set.seed(123) # for reproducibility

# True parameters for data generation
T_obs <- 1200 # Number of observations (approx 4.75 years of daily data)
# GARCH parameters for stock A
omega_A_true <- 0.00001
alpha_A_true <- 0.08
beta_A_true  <- 0.90
# GARCH parameters for stock B
omega_B_true <- 0.000015
alpha_B_true <- 0.07
beta_B_true  <- 0.91
# Constant conditional correlation
rho_true     <- 0.6

# Initialize vectors
r_A <- numeric(T_obs)
r_B <- numeric(T_obs)
sigma2_A_t <- numeric(T_obs)
sigma2_B_t <- numeric(T_obs)

# Initial unconditional variances
sigma2_A_t[1] <- omega_A_true / (1 - alpha_A_true - beta_A_true)
sigma2_B_t[1] <- omega_B_true / (1 - alpha_B_true - beta_B_true)

# Simulate GARCH processes and returns
# Generate correlated normal innovations
innovations <- matrix(rnorm(2 * T_obs), ncol = 2)
chol_R <- matrix(c(1, rho_true, rho_true, 1), nrow = 2)
chol_decomp <- chol(chol_R) # Cholesky decomposition of the correlation matrix
correlated_innovations <- innovations %*% chol_decomp

for (t in 1:T_obs) {
    # Generate returns
    r_A[t] <- sqrt(sigma2_A_t[t]) * correlated_innovations[t, 1]
    r_B[t] <- sqrt(sigma2_B_t[t]) * correlated_innovations[t, 2]

    # Update conditional variances for next period (if not the last observation)
    if (t < T_obs) {
        sigma2_A_t[t+1] <- omega_A_true + alpha_A_true * r_A[t]^2 + beta_A_true * sigma2_A_t
        sigma2_B_t[t+1] <- omega_B_true + alpha_B_true * r_B[t]^2 + beta_B_true * sigma2_B_t
    }
}

# Create the data matrix
simulated_mDATA <- cbind(r_A, r_B)
colnames(simulated_mDATA) <- c("DanskeBank", "Orsted")
return(simulated_mDATA)
}

mDATA <- genData()
```

2.

```r
fAvgNegLogLikGarch <- function(vParams, mInput) {
  if (dim(mInput)[2] != 2) {
    warning("Incorrect number of dimensions in input matrix")
    return(NULL)
  }

  dOmegaA <- vParams[1]
  dAlphaA <- vParams[2]
  dBetaA <- vParams[3]
  dOmegaB <- vParams[4]
  dAlphaB <- vParams[5]
  dBetaB <- vParams[6]
  dRho <- vParams[7]

  dT <- nrow(mInput)

  mSigma2 <- matrix(0, nrow(mInput), 2)
  mSigma2[1, 1] <- dOmegaA / (1 - dAlphaA - dBetaA)
  mSigma2[1, 2] <- dOmegaB / (1 - dAlphaB - dBetaB)

  dOut <- 0

  for (t in 2:dT) {
    mSigma2[t, 1] <- dOmegaA + dAlphaA * mInput[t - 1, 1]^2 + dBetaA * mSigma2[t - 1, 1]
    mSigma2[t, 2] <- dOmegaB + dAlphaB * mInput[t - 1, 2]^2 + dBetaB * mSigma2[t - 1, 2]
    mSigmaCov <- matrix(c(mSigma2[t, 1], rep(dRho * sqrt(mSigma2[t, 1]) * sqrt(mSigma2[t, 2]
    dOut <- dOut - log(sqrt(2 * pi * det(mSigmaCov))) - 0.5 * as.numeric(t(mInput[t, ]) %*%
  }

  return(-dOut / (dT - 1))
}
```

3.

```
vParams <- c(sd(mDATA[, 1]) * 0.05, 0.05, 0.9, sd(mDATA[, 2]) * 0.05, 0.05, 0.9, 0)

optim(vParams, fAvgNegLogLikGarch, mInput = mDATA, method = "BFGS")

# optim cannot converge if it decides to explore negative values, since we cannot take the sc
```

4.

```
fAvgNegLogLikGarchReparam <- function(vParams, mInput) {
  if (dim(mInput)[2] != 2) {
    warning("Incorrect number of dimensions in input matrix")
    return(NULL)
  }

  dOmegaA <- exp(vParams[1])
  dAlphaA <- exp(vParams[2]) / (1 + exp(vParams[2]) + exp(vParams[3]))
  dBetaA <- exp(vParams[3]) / (1 + exp(vParams[2]) + exp(vParams[3]))
  dOmegaB <- exp(vParams[4])
  dAlphaB <- exp(vParams[5]) / (1 + exp(vParams[5]) + exp(vParams[6]))
  dBetaB <- exp(vParams[6]) / (1 + exp(vParams[5]) + exp(vParams[6]))
  dRho <- -1 + 2 * (exp(vParams[7])) / (1 + exp(vParams[7]))

  dT <- nrow(mInput)
  mSigma2 <- matrix(0, nrow(mInput), 2)
  mSigma2[1, 1] <- dOmegaA / (1 - dAlphaA - dBetaA)
  mSigma2[1, 2] <- dOmegaB / (1 - dAlphaB - dBetaB)

  dOut <- 0

  for (t in 2:dT) {
    mSigma2[t, 1] <- dOmegaA + dAlphaA * mInput[t - 1, 1]^2 + dBetaA * mSigma2[t - 1, 1]
    mSigma2[t, 2] <- dOmegaB + dAlphaB * mInput[t - 1, 2]^2 + dBetaB * mSigma2[t - 1, 2]
    mSigmaCov <- matrix(c(mSigma2[t, 1], rep(dRho * sqrt(mSigma2[t, 1]) * sqrt(mSigma2[t, 2]
    dOut <- dOut - log(sqrt(2 * pi * det(mSigmaCov))) - 0.5 * as.numeric(t(mInput[t, ]) %*%
  }

  return(-dOut / (dT - 1))
}
```

5.

```
vParams <- c(sd(mDATA[, 1]) * 0.05, 0.05, 0.9, sd(mDATA[, 2]) * 0.05, 0.05, 0.9, 0)
# transform
vParams[1] <- log(vParams[1])
vParams[2] <- log(vParams[2] / (1 - vParams[2]- vParams[3]))
vParams[3] <- log(vParams[3] / (1 - vParams[2]- vParams[3]))
vParams[4] <- log(vParams[4])
vParams[5] <- log(vParams[5] / (1 - vParams[5]- vParams[6]))
vParams[6] <- log(vParams[6] / (1 - vParams[5]- vParams[6]))
vParams[7] <- log(((vParams[7] + 1) / 2.0) / (1 - ((vParams[7] + 1) / 2.0)))

optim_results <- optim(vParams, fAvgNegLogLikGarchReparam, mInput = mDATA, method = "BFGS")
optim_results
#> $par
#> [1] -10.9045231   0.3523736   3.0242257 -11.6615536   1.7656300   4.3434653
#> [7]   1.3882186
#>
#> $value
#> [1] -5.925866
#>
#> $counts
#> function gradient
#>      100       98
#>
#> $convergence
#> [1] 0
#>
#> $message
#> NULL

vParams <- optim_results$par
dOmegaA <- exp(vParams[1])
dAlphaA <- exp(vParams[2]) / (1 + exp(vParams[2]) + exp(vParams[3]))
dBetaA <- exp(vParams[3]) / (1 + exp(vParams[2]) + exp(vParams[3]))
dOmegaB <- exp(vParams[4])
dAlphaB <- exp(vParams[5]) / (1 + exp(vParams[5]) + exp(vParams[6]))
dBetaB <- exp(vParams[6]) / (1 + exp(vParams[5]) + exp(vParams[6]))
dRho <- -1 + 2 * (exp(vParams[7])) / (1 + exp(vParams[7]))

print(paste0("Est. OmegaA: ", dOmegaA, " vs act.: ", 0.0001))
#> [1] "Est. OmegaA: 1.83749335729341e-05 vs act.: 1e-04"
print(paste0("Est. AlphaA: ", dAlphaA, " vs act.: ", 0.08))
#> [1] "Est. AlphaA: 0.0618438534243581 vs act.: 0.08"
print(paste0("Est. BetaA: ", dBetaA, " vs act.: ", 0.90))
#> [1] "Est. BetaA: 0.894678841062443 vs act.: 0.9"
print(paste0("Est. OmegaB: ", dOmegaB, " vs act.: ", 0.000015))
#> [1] "Est. OmegaB: 8.61889557714679e-06 vs act.: 1.5e-05"
print(paste0("Est. AlphaB: ", dAlphaB, " vs act.: ", 0.07))
#> [1] "Est. AlphaB: 0.0697365650771858 vs act.: 0.07"
print(paste0("Est. BetaB: ", dBetaB, " vs act.: ", 0.91))
#> [1] "Est. BetaB: 0.918332975052663 vs act.: 0.91"
print(paste0("Est. Rho: ", dRho, " vs act.: ", 0.6))
#> [1] "Est. Rho: 0.600615406946657 vs act.: 0.6"
```

6.

```
// [[Rcpp::depends(RcppArmadillo)]]
#include <RcppArmadillo.h>

using namespace arma;
using namespace Rcpp;

// [[Rcpp::export]]
vec transToOrig_cpp(vec vInput) {
  vec vOut = zeros<vec>(vInput.size());
  vOut[0] = exp(vInput[0]);
  vOut[1] = exp(vInput[1]) / (1 + exp(vInput[1]) + exp(vInput[2]));
  vOut[2] = exp(vInput[2]) / (1 + exp(vInput[1]) + exp(vInput[2]));
  vOut[3] = exp(vInput[3]);
  vOut[4] = exp(vInput[4]) / (1 + exp(vInput[4]) + exp(vInput[5]));
  vOut[5] = exp(vInput[5]) / (1 + exp(vInput[4]) + exp(vInput[5]));
  vOut[6] = -1 + 2 * (exp(vInput[6])) / (1 + exp(vInput[6]));

  return vOut;
}
```

```
suppressMessages(library(Rcpp))
#> Warning: pakke 'Rcpp' blev bygget under R version 4.3.3
suppressMessages(library(RcppArmadillo))
#> Warning: pakke 'RcppArmadillo' blev bygget under R version 4.3.3
sourceCpp("exam2021cpp.cpp")

vNew <- transToOrig_cpp(vParams)
vNew[7]
#> [1] 0.6006154
```

7.

21 Retake exam 2021

21.1 Problem 1:

1.

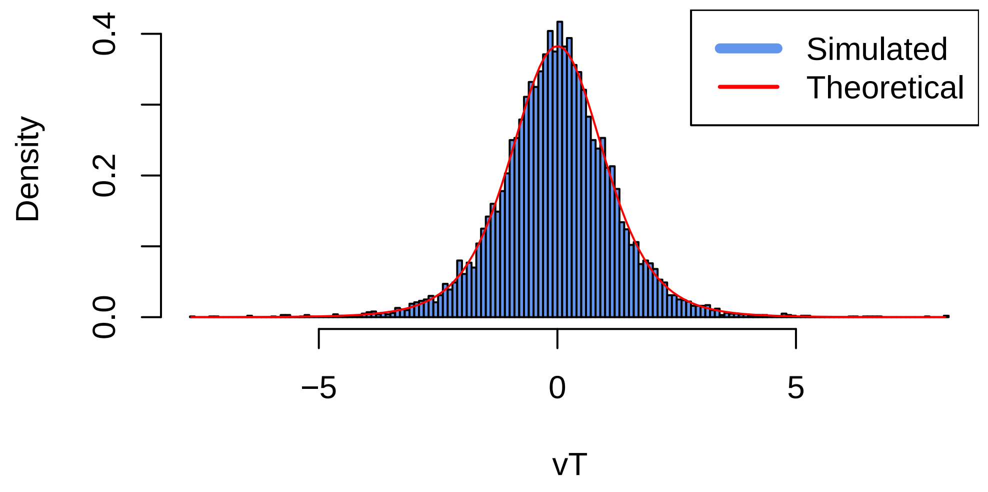
```r
set.seed(134)

### Auxiliary functions ###

gamma.pdf <- function(x, k, theta) {
  return(theta^k * x^(k-1) * exp(-theta*x)/gamma(k))
}

exponential.pdf <- function(x, lambda) {
  return(lambda * exp(-lambda*x))
}

Exponential.Simulate <- function(lambda, size = 1) {
  V <- runif(size)
  return(-1/lambda * log(V))
}

Gamma.Simulate <- function(k, theta, size = 1) {

  lambda <- theta/k
  c <- k^k * exp(-k+1) / gamma(k)

  U <- rep(NA, size)
  Y <- rep(NA, size)
  X <- rep(NA, size)
  Unaccepted <- rep(TRUE, size)

  while (any(Unaccepted)) {

    UnacceptedCount <- sum(Unaccepted)

    U <- runif(UnacceptedCount)
    Y <- Exponential.Simulate(lambda, UnacceptedCount)

    Accepted_ThisTime <- Unaccepted[Unaccepted] &
      ( U <= ( gamma.pdf(Y, k, theta) / exponential.pdf(Y, lambda)/c ) )

    X[Unaccepted][Accepted_ThisTime] <- Y[Accepted_ThisTime]
    Unaccepted[Unaccepted] <- !Accepted_ThisTime

  }

  return(X)

}

vW <- Gamma.Simulate(k = 3, theta = 3, size = 10000)
```

2.

```
BoxMuller <- function(size = 1) {

  U <- runif(size)
  V <- runif(size)

  X <- sqrt(-2*log(U)) * cos(2*pi*V)
  Y <- sqrt(-2*log(U)) * sin(2*pi*V)

  return(c(X,Y))

}

vZ <- BoxMuller(10000/2)
```

3.

```
vT <- 1 / sqrt(vW) * vZ

hist(vT, breaks = 200, freq = FALSE,
     main = "",
     col = "cornflowerblue",
     xlim = c(min(vT), max(vT)))

xticks = seq(min(vT), max(vT), 0.1)
lines(xticks, dt(xticks, 6), col = "red")
legend("topright", legend = c("Simulated", "Theoretical"), lty = c(1, 1), lwd = c(5, 2), col
```



4.

```
f <- function(iN, iA, iB) {
  vX <- seq(1:iN)
  vOut <- vX[vX %% iA == 0 | vX %% iB == 0]
  return(list(
    numbers = vOut,
    sumofnumbers = sum(vOut)
  ))
}

f(iN = 300, iA = 11, iB = 42)
#> $numbers
#>  [1]  11  22  33  42  44  55  66  77  84  88  99 110 121 126 132 143 154 165 168
#> [20] 176 187 198 209 210 220 231 242 252 253 264 275 286 294 297
#>
```

```
#> $sumofnumbers
#> [1] 5334
```

5.

```
f <- function(vX) {
  bFound <- FALSE
  dOut <- NULL
  i <- 1
  while (bFound == FALSE) {
    if (sum(i %% vX) == 0) {
      dOut <- i
      bFound <- TRUE
    }
    i <- i + 1
  }
  return(dOut)
}

f(c(3, 5, 7))
#> [1] 105
f(seq(1, 15))
#> [1] 360360
```

21.2 Problem 2: (Constrained) Optimization, C++, and Packaging

1.

```r
set.seed(123)
genData <- function() {
  # --- Generate mDATA.r ---
  set.seed(123) # for reproducibility

# True parameters for data generation
T_obs <- 1200 # Number of observations (approx 4.75 years of daily data)
# GARCH parameters for stock A
omega_A_true <- 0.000001
alpha_A_true <- 0.08
beta_A_true  <- 0.9
# GARCH parameters for stock B
omega_B_true <- 0.000001
alpha_B_true <- 0.09
beta_B_true  <- 0.91
# Constant conditional correlation
rho_true     <- 0.6

# Initialize vectors
r_A <- numeric(T_obs)
r_B <- numeric(T_obs)
sigma2_A_t <- numeric(T_obs)
sigma2_B_t <- numeric(T_obs)

# Initial unconditional variances
sigma2_A_t[1] <- omega_A_true / (1 - alpha_A_true - beta_A_true)
sigma2_B_t[1] <- omega_B_true / (1 - alpha_B_true - beta_B_true)

# Simulate GARCH processes and returns
# Generate correlated normal innovations
innovations <- matrix(rnorm(2 * T_obs), ncol = 2)
chol_R <- matrix(c(1, rho_true, rho_true, 1), nrow = 2)
chol_decomp <- chol(chol_R) # Cholesky decomposition of the correlation matrix
correlated_innovations <- innovations %*% chol_decomp

for (t in 1:T_obs) {
    # Generate returns
    r_A[t] <- sqrt(sigma2_A_t[t]) * correlated_innovations[t, 1]
    r_B[t] <- sqrt(sigma2_B_t[t]) * correlated_innovations[t, 2]

    # Update conditional variances for next period (if not the last observation)
    if (t < T_obs) {
        sigma2_A_t[t+1] <- omega_A_true + alpha_A_true * r_A[t]^2 + beta_A_true * sigma2_A_t
        sigma2_B_t[t+1] <- omega_B_true + alpha_B_true * r_B[t]^2 + beta_B_true * sigma2_B_t
    }
}

# Create the data matrix
simulated_mDATA <- cbind(r_A, r_B)
colnames(simulated_mDATA) <- c("DanskeBank", "Orsted")
return(simulated_mDATA)
}

mDATA <- genData()
```

2.

(a)

(b)

```r
fAvgNegLogLik <- function(vParams, vInput) {
  dMu <- vParams[1]
  dAlpha <- exp(vParams[2]) / (1 + exp(vParams[2]))
  dBeta <- 1 - dAlpha
  dNu <- 2 + exp(vParams[4])

  dSum <- 0
  dT <- length(vInput)

  vSigma2 <- numeric(dT)
  vSigma2[1] <- var(vInput)

  for (t in 2:dT) {
    vSigma2[t] <- dAlpha * vInput[t - 1]^2 + dBeta * vSigma2[t-1]
    dSum <- dSum + log(dt((vInput[t] - dMu) / sqrt(vSigma2[t]), dNu)) - log(sqrt(vSigma2[t]))
  }

  return(-dSum / (dT - 1))
}
```

3.

```r
vParams <- c(0, log(0.9/(1-0.9)), 1 - log(0.9/(1-0.9)), log(5 - 2))
optim_results <- optim(vParams, fAvgNegLogLik, vInput = mDATA[, 1], method = "BFGS")
optim_results
#> $par
#> [1]  0.0001021925 -2.9909391286 -1.1972245773  6.1836752296
#>
#> $value
#> [1] -3.631868
#>
#> $counts
#> function gradient
#>       54       24
#>
#> $convergence
#> [1] 0
#>
```

```
#> $message
#> NULL
vParams <- optim_results$par

# retransformation
dMu_star <- vParams[1]
dAlpha_star <- exp(vParams[2]) / (1 + exp(vParams[2]))
dBeta_star <- 1 - dAlpha_star
dNu_star <- 2 + exp(vParams[4])

print(paste0("Est. mu: ", dMu_star))
#> [1] "Est. mu: 0.000102192473262923"
print(paste0("Est. alpha: ", dAlpha_star, " vs. true: ", 0.08))
#> [1] "Est. alpha: 0.0478368957544601 vs. true: 0.08"
print(paste0("Est. beta: ", dBeta_star, " vs. true: ", 0.90))
#> [1] "Est. beta: 0.95216310424554 vs. true: 0.9"
print(paste0("Est. nu: ", dNu_star))
#> [1] "Est. nu: 486.770328670692"
```

4.

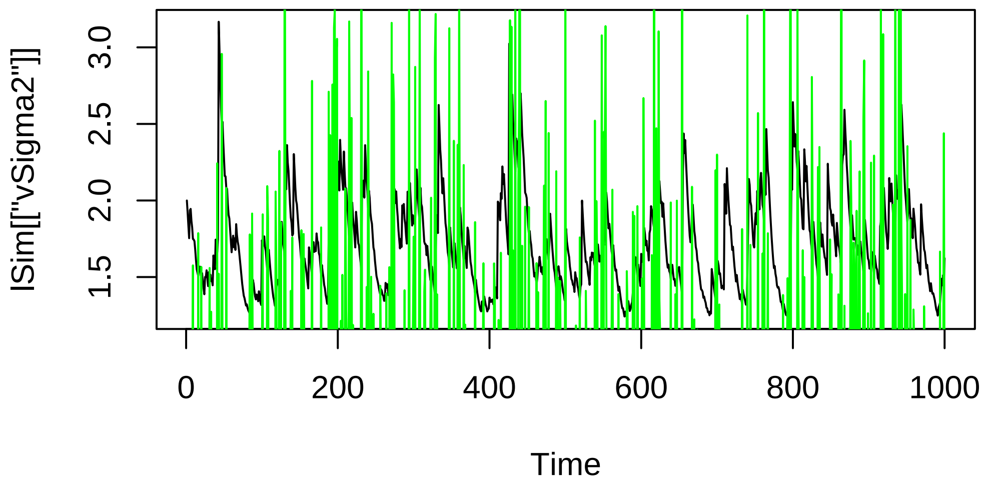
    (a)
    (b)
    (c)

    (d)

```
SEXP a = myFunction(input);
double x = *REAL(a);
double y = 2*x + pow(x,2);
```

```
// [[Rcpp::depends(RcppArmadillo)]]
#include <RcppArmadillo.h>

using namespace arma;
using namespace Rcpp;

// Helper function to get a shock
inline double get_shock(Function f, const vec& vInputs) {
  SEXP s;
  int n_extra_args = vInputs.n_elem;

  if (n_extra_args == 0) { // e.g. f takes only n, like a pre-wrapped user function or rnorm
    s = f(1);
  } else if (n_extra_args == 1) { // e.g. rt(n, df) or rnorm(n, mean)
    s = f(1, vInputs[0]);
  } else if (n_extra_args == 2) { // e.g. rnorm(n, mean, sd)
    s = f(1, vInputs[0], vInputs[1]);
  } else {
    // Fallback or error for more arguments, or could extend this if/else
    Rcpp::stop("vInputs has too many elements for this simplified handler. Max 2 supported.")
  }
  if (TYPEOF(s) != REALSXP || Rf_length(s) != 1) {
    Rcpp::stop("The user-supplied function 'f' did not return a single numeric value.");
  }
  return Rcpp::as<double>(s);
}


// [[Rcpp::export]]
List GarchSim(int iT_in, vec vParams, Function f, vec vInputs) {
  int iT = iT_in;
  if (iT <= 0) { iT = 1; } // Simplified default handling

  double dOmega = vParams[0];
  double dAlpha = vParams[1];
  double dBeta  = vParams[2];

  vec vY(iT, fill::zeros);
  vec vSigma2(iT, fill::zeros);

  if (dOmega <= 0 || dAlpha < 0 || dBeta < 0 || (dAlpha + dBeta) >= 1.0) {
    Rcpp::stop("Invalid GARCH parameters.");
  }

  double unconditional_variance = dOmega / (1.0 - dAlpha - dBeta);
  vSigma2(0) = unconditional_variance;

  double current_eps = get_shock(f, vInputs); // Call helper
  vY(0) = sqrt(vSigma2(0)) * current_eps;

  for (int t = 1; t < iT; t++) {
    vSigma2(t) = dOmega + dAlpha * pow(current_eps, 2) + dBeta * vSigma2(t-1);
    if (vSigma2(t) <= 0) vSigma2(t) = unconditional_variance;

    current_eps = get_shock(f, vInputs); // Call helper
    vY(t) = sqrt(vSigma2(t)) * current_eps;
  }

  return List::create(_["vSigma2"] = vSigma2, _["vY"] = vY);
}
```

```
suppressMessages(library(Rcpp))
#> Warning: pakke 'Rcpp' blev bygget under R version 4.3.3
suppressMessages(library(RcppArmadillo))
#> Warning: pakke 'RcppArmadillo' blev bygget under R version 4.3.3

sourceCpp("exam2021recpp.cpp")

iT <- 1000
dOmega <- 0.1
dAlpha <- 0.05
dBeta <- 0.9
df_t <- 5

lSim <- GarchSim(iT, c(dOmega, dAlpha, dBeta), f = rt, vInputs = df_t)

plot(1:1000, lSim[["vSigma2"]], type = "l", lty = 1, xlab = "Time")
lines(1:1000, lSim[["vY"]],col="green")
```



5.

22 Ordinary exam 2022

22.1 Problem 1:

1.

```r
### Auxiliary functions ###

gamma.pdf <- function(x, k, theta) {
  return(theta^k * x^(k-1) * exp(-theta*x)/gamma(k))
}
```

```r
exponential.pdf <- function(x, lambda) {
  return(lambda * exp(-lambda*x))
}

Exponential.Simulate <- function(lambda, size = 1) {
  V <- runif(size)
  return(-1/lambda * log(V))
}

Gamma.Simulate <- function(k, theta, size = 1) {

  lambda <- theta/k
  c <- k^k * exp(-k+1) / gamma(k)

  U <- rep(NA, size)
  Y <- rep(NA, size)
  X <- rep(NA, size)
  Unaccepted <- rep(TRUE, size)

  while (any(Unaccepted)) {

    UnacceptedCount <- sum(Unaccepted)

    U <- runif(UnacceptedCount)
    Y <- Exponential.Simulate(lambda, UnacceptedCount)

    Accepted_ThisTime <- Unaccepted[Unaccepted] &
      ( U <= ( gamma.pdf(Y, k, theta) / exponential.pdf(Y, lambda)/c ) )

    X[Unaccepted][Accepted_ThisTime] <- Y[Accepted_ThisTime]
    Unaccepted[Unaccepted] <- !Accepted_ThisTime

  }

  return(X)

}

set.seed(123)

iN <- 100
iT <- 1000
```

```r
vW1 <- matrix(Gamma.Simulate(3, 3, iN * iT), iT, iN)
vW2 <- matrix(Gamma.Simulate(1, 3, iN * iT), iT, iN)
```

2.

```r
vB <- matrix(vW1 / (vW1 + vW2), iT, iN)
```

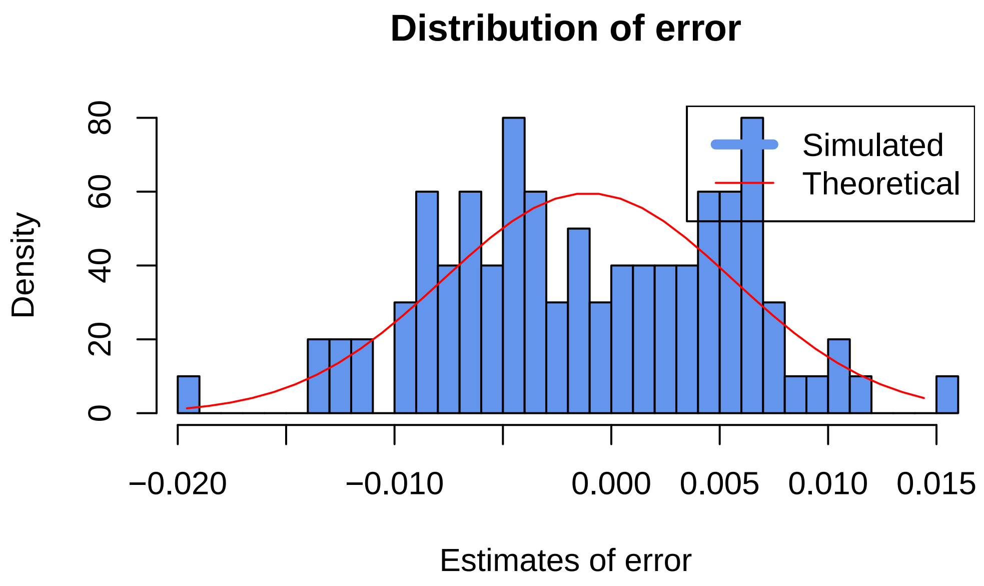
3.

```r
fn_estimateMean <- function(mInput) {
  mOut <- apply(mInput, 2, mean)
  return(mOut)
}

dU <- 3 / (1 + 3)
vDiffMu <- fn_estimateMean(vB) - dU
```

4.

```r
hist(vDiffMu, breaks = 25, freq = FALSE,
     main = "Distribution of error",
     col = "cornflowerblue",
     xlim = c(min(vDiffMu), max(vDiffMu)),
     xlab = "Estimates of error")
```

```r
xticks = seq(min(vDiffMu), max(vDiffMu), 0.001)
lines(xticks, dnorm(xticks, mean(vDiffMu), sd(vDiffMu)), col = "red")
legend("topright", legend = c("Simulated", "Theoretical"), lty = c(1, 1), lwd = c(5, 1), col
```



22.2 Problem 2: (Constrained) Optimization, C++, and Packaging

1.

```r
# Ensure rugarch is installed and loaded
# install.packages("rugarch")
library(rugarch)
#> Warning: pakke 'rugarch' blev bygget under R version 4.3.3
#> Indlæser krævet pakke: parallel
#>
```

```r
#> Vedhæfter pakke: 'rugarch'
#> Det følgende objekt er maskeret fra 'package:stats':
#>
#>     sigma

# --- True parameters for new data generation (MODIFIED FOR PERSISTENCE < 1) ---
set.seed(456) # Use a different seed
T_obs_new <- 1500    # Number of observations for the new series
mu_true_new <- 0.0002 # True mean
omega_true_new <- 0.000015 # True omega

# Modify alphas so their sum is slightly less than 1
alpha1_true_gen <- 0.65
alpha2_true_gen <- 0.349  # Sum = 0.65 + 0.349 = 0.999

cat("Generating data with persistence:", alpha1_true_gen + alpha2_true_gen, "\n")
#> Generating data with persistence: 0.999

# --- Simulate data using rugarch ---
spec_gen <- ugarchspec(
  variance.model = list(model = "sGARCH", garchOrder = c(2, 0)), # ARCH(2)
  mean.model = list(armaOrder = c(0, 0), include.mean = TRUE, archm = FALSE, arfima = FALSE)
  distribution.model = "norm",
  fixed.pars = list(mu = mu_true_new,
                    omega = omega_true_new,
                    alpha1 = alpha1_true_gen, # Use modified alpha
                    alpha2 = alpha2_true_gen) # Use modified alpha
)

# Simulate the path
# n.start is for burn-in; rugarchpath typically handles this well.
# For explicit burn-in to remove: n.sim = T_obs_new + 200, then trim first 200.
# Or use n.start within ugarchpath if it directly supports it for the path output.
# Simpler: simulate longer and trim.
path_sim_obj <- ugarchpath(spec_gen, n.sim = T_obs_new + 200, n.start = 0, m.sim = 1)
vR_new <- as.numeric(fitted(path_sim_obj)[-(1:200)]) # Get returns, remove first 200 as burn

cat("Generated vR_new with length:", length(vR_new), "\n")
#> Generated vR_new with length: 1500
# plot(vR_new, type="l", main="Simulated ARCH(2) Returns (Persistence < 1)")
# acf(vR_new^2, main="ACF of Squared Simulated Returns")
```

2.

(a)
(b)
(c)

i)
ii)

```r
fAvgNegLogLik <- function(vParams, vR, p = 2) {
  dMu <- vParams[1]
  dOmega <- exp(vParams[2])
  vParams[3:length(vParams)] <- vParams[3:length(vParams)] / sum(vParams[3:length(vParams)])

  iT <- length(vR)
  vSigma2 <- numeric(iT)
  for (j in 1:p) {
    vSigma2[j] <- dOmega / (1 - sum(vParams[3:length(vParams)]))
  }
  dSum <- 0

  for (t in (p + 1):iT) {
    vSigma2[t] <- dOmega

    for (j in 1:p) {
      vSigma2[t] <- vSigma2[t] + vParams[j + 2] * vR[t - j]^2
    }

    dSum <- dSum - log(sqrt(vSigma2[t])) + log(dnorm((vR[t] - dMu)/sqrt(vSigma2[t])))
  }

  return(-dSum / (iT - p))
}
```

3.

```
optim_res <- optim(c(0, log(mean(vR_new^2)), 0.7, 0.3), fAvgNegLogLik, vR = vR_new, method =
optim_res
#> $par
#> [1]   0.000669589 -11.162179071   0.691584866   0.332063803
#>
#> $value
#> [1] -3.50962
#>
#> $counts
#> function gradient
#>       24       10
#>
#> $convergence
#> [1] 0
#>
#> $message
#> NULL
```

```
vParams <- optim_res$par
dMu_star <- vParams[1]
dOmega_star <- exp(vParams[2])
vAlphas_star <- vParams[3:length(vParams)] / sum(vParams[3:length(vParams)])

dMu_star
#> [1] 0.000669589
dOmega_star
#> [1] 1.420127e-05
vAlphas_star
#> [1] 0.6756076 0.3243924

spec_fit_rugarch <- ugarchspec(
  variance.model = list(model = "sGARCH", garchOrder = c(2, 0)), # ARCH(2)
  mean.model = list(armaOrder = c(0, 0), include.mean = TRUE), # Estimate mu
  distribution.model = "norm"
)

# Fit the model using rugarch
fit_rugarch <- ugarchfit(spec = spec_fit_rugarch, data = vR_new, solver = "nloptr") # or "nlo

print("--- rugarch Results ---")
#> [1] "--- rugarch Results ---"
print(fit_rugarch)
#>
#> *---------------------------------*
#> *          GARCH Model Fit        *
#> *---------------------------------*
#>
#> Conditional Variance Dynamics
#> -----------------------------------
#> GARCH Model  : sGARCH(2,0)
#> Mean Model   : ARFIMA(0,0,0)
#> Distribution : norm
#>
#> Optimal Parameters
#> ------------------------------------
#>         Estimate  Std. Error  t value Pr(>|t|)
#> mu      0.000494    0.000017  28.8508        0
#> omega   0.000014    0.000000 132.2038        0
#> alpha1  0.670886    0.056970  11.7761        0
#> alpha2  0.323013    0.040834   7.9105        0
```

```
#>
#> Robust Standard Errors:
#>         Estimate  Std. Error  t value Pr(>|t|)
#> mu      0.000494    0.000010  49.4934        0
#> omega   0.000014    0.000000 129.1425        0
#> alpha1  0.670886    0.058066  11.5538        0
#> alpha2  0.323013    0.037942   8.5133        0
#>
#> LogLikelihood : 5262.87
#>
#> Information Criteria
#> ------------------------------------
#>
#> Akaike       -7.0118
#> Bayes        -6.9977
#> Shibata      -7.0118
#> Hannan-Quinn -7.0065
#>
#> Weighted Ljung-Box Test on Standardized Residuals
#> ------------------------------------
#>                         statistic p-value
#> Lag[1]                     0.5291  0.4670
#> Lag[2*(p+q)+(p+q)-1][2]    0.5715  0.6610
#> Lag[4*(p+q)+(p+q)-1][5]    2.5128  0.5032
#> d.o.f=0
#> H0 : No serial correlation
#>
#> Weighted Ljung-Box Test on Standardized Squared Residuals
#> ------------------------------------
#>                         statistic p-value
#> Lag[1]                  0.0004559  0.9830
#> Lag[2*(p+q)+(p+q)-1][5] 0.7997152  0.9031
#> Lag[4*(p+q)+(p+q)-1][9] 1.8692596  0.9200
#> d.o.f=2
#>
#> Weighted ARCH LM Tests
#> ------------------------------------
#>             Statistic Shape Scale P-Value
#> ARCH Lag[3]   0.08255 0.500 2.000  0.7739
#> ARCH Lag[5]   1.75598 1.440 1.667  0.5277
#> ARCH Lag[7]   2.36933 2.315 1.543  0.6395
#>
```

```
#> Nyblom stability test
#> ------------------------------------
#> Joint Statistic:  0.4199
#> Individual Statistics:
#> mu     0.08636
#> omega  0.07282
#> alpha1 0.14335
#> alpha2 0.09463
#>
#> Asymptotic Critical Values (10% 5% 1%)
#> Joint Statistic:          1.07 1.24 1.6
#> Individual Statistic:     0.35 0.47 0.75
#>
#> Sign Bias Test
#> ------------------------------------
#>                    t-value   prob sig
#> Sign Bias           0.7563 0.4496
#> Negative Sign Bias  0.8591 0.3904
#> Positive Sign Bias  0.3725 0.7096
#> Joint Effect        2.4834 0.4783
#>
#>
#> Adjusted Pearson Goodness-of-Fit Test:
#> ------------------------------------
#>   group statistic p-value(g-1)
#> 1    20     11.97       0.8868
#> 2    30     14.68       0.9874
#> 3    40     32.64       0.7539
#> 4    50     39.73       0.8249
#>
#>
#> Elapsed time : 0.345191
```

23 Retake exam 2022

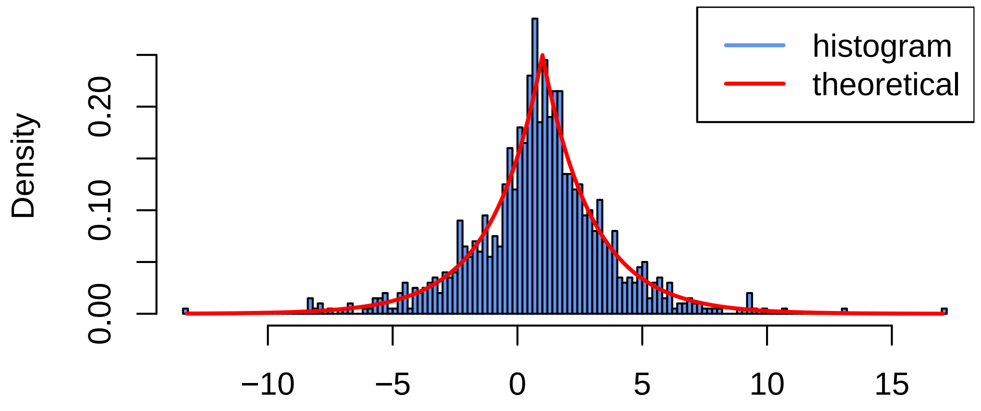
23.1 Problem 1:

1.

```r
set.seed(134)

fn_Laplace <- function(dMu, dSigma, iSamples, iLength) {

  U <- runif(iSamples * iLength)
  vTemp <- dMu - dSigma * sign(U-0.5) * log(1 - 2 * abs(U - 0.5))
  return(matrix(vTemp, nrow = iLength, ncol = iSamples))

}

mX <- fn_Laplace(dMu = 1, dSigma = 2, iSamples = 5000, iLength = 1000)
```

2.

```r
fLapDen <- function(dMu, dSigma, x) {
  return(1 / (2 * dSigma) * exp(-abs(x - dMu) / dSigma))
}

hist(mX[, 1],
     freq = FALSE,
     breaks = 200,
     col = "cornflowerblue",
     xlab = "",
     ylab = "Density",
     main = "",
     xlim = c(min(mX[, 1]), max(mX[, 1])))

vX <- seq(min(mX[, 1]), max(mX[, 1]), by = 0.05)
lines(vX, fLapDen(dMu = 1, dSigma = 2, vX), type = "l", col = "red", lwd = 2)

legend("topright",
       legend = c("histogram", "theoretical"),
       col = c("cornflowerblue", "red"),
       lwd = 2)
```

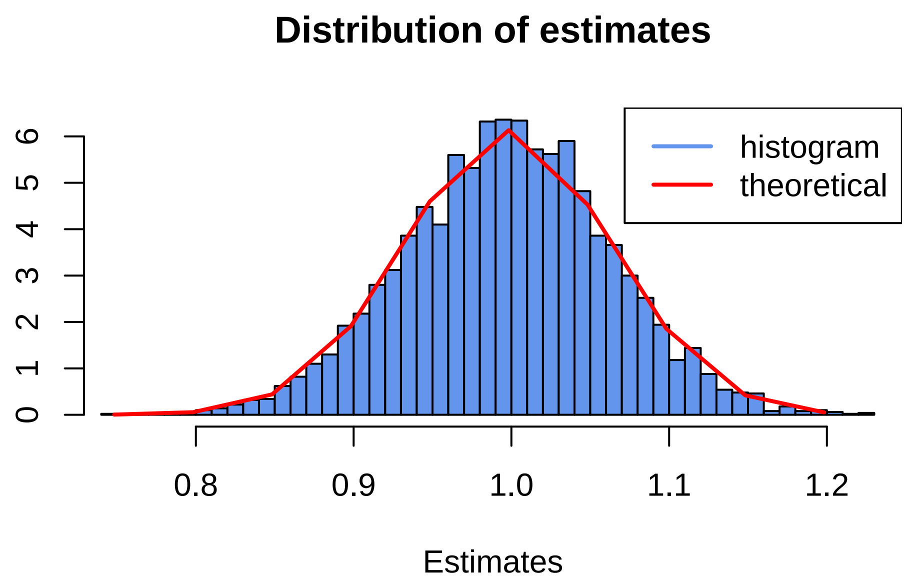


3.

```
fn_estimateLaplace <- function(mInput) {
  fTempMed <- function(vX) {
    return(sort(vX, decreasing = FALSE)[ceiling(length(vX) / 2)])
  }
  fSigmaCal <- function(vX) {
    vOut <- ifelse(vX - fTempMed(vX) < 0, (vX - fTempMed(vX)) * -1, vX - fTempMed(vX))
    return(mean(vOut))
  }

  return(list(
    vMu = apply(mInput, 2, fTempMed),
    vSigma = apply(mInput, 2, fSigmaCal)
  ))
}

vMu <- fn_estimateLaplace(mX)$vMu
vSigma <- fn_estimateLaplace(mX)$vSigma
```

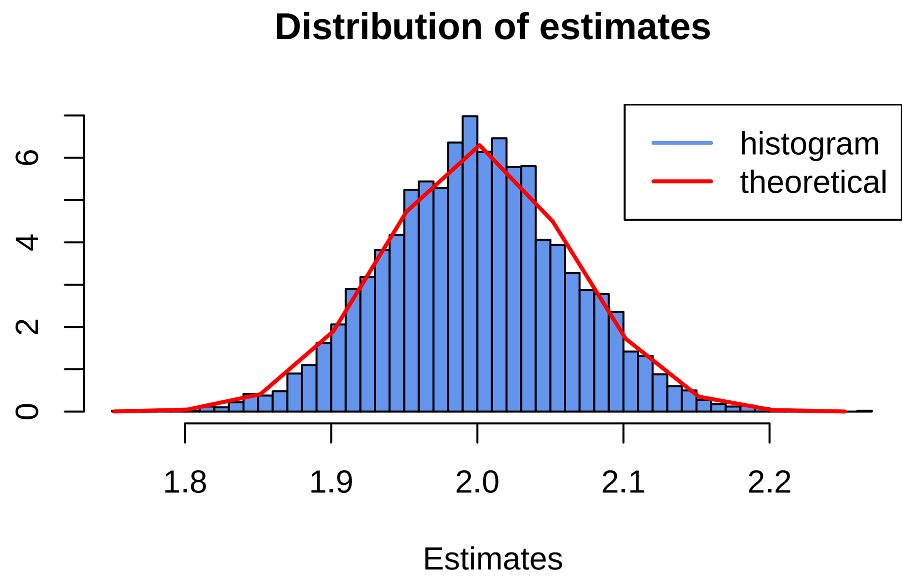
4.

```
hist(vMu,
     freq = FALSE,
     breaks = 50,
     col = "cornflowerblue",
     xlab = "Estimates",
     ylab = "",
     main = "Distribution of estimates",
     xlim = c(min(vMu), max(vMu)))

vX <- seq(min(vMu), max(vMu), by = 0.05)
lines(vX, dnorm(vX, mean = mean(vMu), sd = sd(vMu)), type = "l", col = "red", lwd = 2)

legend("topright",
       legend = c("histogram", "theoretical"),
       col = c("cornflowerblue", "red"),
       lwd = 2)
```



```
hist(vSigma,
     freq = FALSE,
     breaks = 50,
     col = "cornflowerblue",
     xlab = "Estimates",
     ylab = "",
     main = "Distribution of estimates",
     xlim = c(min(vSigma), max(vSigma)))
```

```r
vX <- seq(min(vSigma), max(vSigma), by = 0.05)
lines(vX, dnorm(vX, mean = mean(vSigma), sd = sd(vSigma)), type = "l", col = "red", lwd = 2)

legend("topright",
       legend = c("histogram", "theoretical"),
       col = c("cornflowerblue", "red"),
       lwd = 2)
```



5.

```r
fn_estimateVariance <- function(mInput) {
  return(apply(mInput, 2, var))
}

dSigma <- 2
```

```r
vTemp <- fn_estimateVariance(mX) - 2 * dSigma^2
mean(vTemp)
#> [1] 0.01253421
sd(vTemp)
#> [1] 0.5653084
```

23.2 Problem 2: (Constrained) Optimization, C++, and Packaging

1.

```r
generate_sample_garch_data <- function(num_observations = 1200,
                                       burn_in_period = 200,
                                       alpha1_sim = 0.10,
                                       alpha2_sim = 0.10,
                                       beta_sim = 0.80,
                                       nu_sim = 5,
                                       initial_sigma2 = (0.01)^2,
                                          seed = 42) {
  if (!is.null(seed)) {
    set.seed(seed)
  }

  total_length = num_observations + burn_in_period

  # --- Parameter Validation ---
  if (alpha1_sim < 0 || alpha2_sim < 0 || beta_sim < 0) {
    stop("GARCH parameters (alpha1, alpha2, beta) must be non-negative.")
  }
  # The problem later constrains alpha1+alpha2+beta=1.
  # For simulation, stationarity usually requires alpha1+alpha2+beta < 1 if an intercept omeg
  # Here, omega=0. If alpha1+alpha2+beta >= 1, it's an IGARCH or explosive process.
  # The chosen defaults sum to 1 (IGARCH), which is common for financial data.

  if (nu_sim <= 2) {
    stop("Degrees of freedom 'nu_sim' for t-distribution must be > 2 for finite variance.")
  }
  if (initial_sigma2 <= 0) {
    stop("Initial variance 'initial_sigma2' must be positive.")
  }

  # --- Initialize Vectors ---
  r_t <- numeric(total_length)      # Vector for returns
  sigma2_t <- numeric(total_length) # Vector for conditional variances

  # --- Generate Standardized t-distributed Innovations (mean 0, variance 1) ---
  # A standard t-distribution with nu degrees of freedom has variance nu / (nu - 2).
  innovations <- rt(total_length, df = nu_sim) * sqrt((nu_sim - 2) / nu_sim)

  # --- Set Initial Values ---
  # For GARCH(2,1): sigma_t^2 = alpha1*r_{t-1}^2 + alpha2*r_{t-2}^2 + beta*sigma_{t-1}^2
  # We need to bootstrap the process for the first few periods.

  # Period 1:
  sigma2_t[1] <- initial_sigma2
  r_t[1] <- innovations[1] * sqrt(sigma2_t[1])

  # Period 2:
  # sigma_2^2 depends on r_1^2, r_0^2, sigma_1^2.
  # Assume r_0 = 0 (or its squared value is negligible or absorbed into an effective initial
```

```r
  # So, sigma_2^2 = alpha1*r_1^2 + beta*sigma_1^2 (alpha2*r_0^2 term is zero)
  sigma2_t[2] <- alpha1_sim * r_t[1]^2 + beta_sim * sigma2_t[1]
  if (sigma2_t[2] <= 1e-9) sigma2_t[2] <- 1e-9 # Floor variance
  r_t[2] <- innovations[2] * sqrt(sigma2_t[2])

  # --- Simulate GARCH(2,1) Process ---
  for (t in 3:total_length) {
    sigma2_t[t] <- alpha1_sim * r_t[t-1]^2 +
                   alpha2_sim * r_t[t-2]^2 +
                   beta_sim * sigma2_t[t-1]^2

    # Ensure variance remains positive (numerical stability)
    if (sigma2_t[t] <= 1e-9) {
      sigma2_t[t] <- 1e-9 # Floor variance to prevent issues like sqrt(negative)
    }

    r_t[t] <- innovations[t] * sqrt(sigma2_t[t])
  }

  # --- Discard Burn-in Period ---
  sample_returns <- r_t[(burn_in_period + 1):total_length]

  # Final check for non-finite values (e.g. if sigma2_t became 0 or negative despite floor)
  if(any(!is.finite(sample_returns))){
      warning("Non-finite values were generated in sample_returns. This might indicate issue
      sample_returns[!is.finite(sample_returns)] = 0
  }

  return(sample_returns)
}

vDATA <- generate_sample_garch_data(
  num_observations = 1197,
  alpha1_sim = 0.10,
  alpha2_sim = 0.10,
  beta_sim = 0.80, # Note: alpha1+alpha2+beta = 1.0 (IGARCH process)
  nu_sim = 5,
  seed = 123 # For reproducibility
)
```

2.

(a)

(b)
(c)

```r
fAvgNegLogLik <- function(vParams, vR, f = rnorm) {
  dAlpha1 <- vParams[1]
  dAlpha2 <- vParams[2]
  dBeta <- vParams[3]
  vDensParams <- vParams[4:length(vParams)]

  dT <- length(vR)
  dSum <- 0
  vSigma2 <- numeric(dT)

  vSigma2[1] <- var(vR)
  vSigma2[2] <- dAlpha1 * vR[1]^2 + dBeta * vSigma2[1]

  for (t in 3:dT) {
    vSigma2[t] <- dAlpha1 * vR[t-1]^2 + dAlpha2 * vR[t-2]^2 + dBeta * vSigma2[t-1]
    dSum <- dSum - log(sqrt(vSigma2[t])) + log(do.call(f, as.list(c(vR[t]/sqrt(vSigma2[t]), v
  }

  return(-dSum / (dT - 1))
}
```

3.

(a)

(b)

i)
ii)
iii)
iv)
v)

```r
fAvgNegLogLikRep <- function(vParams, vR) {
  dAlpha1 <- exp(vParams[1]) / (1 + exp(vParams[1]) + exp(vParams[2]))
  dAlpha2 <- exp(vParams[2]) / (1 + exp(vParams[1]) + exp(vParams[2]))
  dBeta <- 1 - dAlpha1 - dAlpha2
  dNu <- 2 + exp(vParams[4])

  vParams <- c(dAlpha1, dAlpha2, dBeta, dNu)
  return(fAvgNegLogLik(vParams, vR, dt))
}
```

4.

```r
vParams <- c(0.1, 0.1, 0.8, 5)
# transform
dAlpha1 <- log(vParams[1] / (1 - vParams[1] - vParams[2]))
dAlpha2 <- log(vParams[2] / (1 - vParams[1] - vParams[2]))
dBeta <- vParams[3] # constraint ensured by functions
dNu <- log(vParams[4] - 2)
vParams <- c(dAlpha1, dAlpha2, dBeta, dNu)

optim_results <- optim(vParams, fAvgNegLogLikRep, vR = vDATA, method = "BFGS")
optim_results
#> $par
#> [1] -9.187285 -9.375245  0.800000  2.441869
#>
#> $value
#> [1] -8.964638
```

```
#>
#> $counts
#> function gradient
#>       38       33
#>
#> $convergence
#> [1] 0
#>
#> $message
#> NULL

vParams <- optim_results$par
dAlpha1 <- exp(vParams[1]) / (1 + exp(vParams[1]) + exp(vParams[2]))
dAlpha2 <- exp(vParams[2]) / (1 + exp(vParams[1]) + exp(vParams[2]))
dBeta <- 1 - dAlpha1 - dAlpha2
dNu <- 2 + exp(vParams[4])

print(paste0("Est. Alpha1: ", dAlpha1, " vs act.: ", 0.1))
#> [1] "Est. Alpha1: 0.00010231320414325 vs act.: 0.1"
print(paste0("Est. Alpha2: ", dAlpha2, " vs act.: ", 0.1))
#> [1] "Est. Alpha2: 8.47815778203704e-05 vs act.: 0.1"
print(paste0("Est. Beta: ", dBeta, " vs act.: ", 0.8))
#> [1] "Est. Beta: 0.999812905218036 vs act.: 0.8"
print(paste0("Est. Nu: ", dNu, " vs act.: ", 5))
#> [1] "Est. Nu: 13.4945084264308 vs act.: 5"
```

```
suppressMessages(library(rugarch))
#> Warning: pakke 'rugarch' blev bygget under R version 4.3.3

spec_fit_rugarch <- ugarchspec(
  variance.model = list(model = "sGARCH", garchOrder = c(2, 1)), # ARCH(2)
  mean.model = list(armaOrder = c(0, 0), include.mean = FALSE), # Estimate mu
  distribution.model = "std"
)

# Fit the model using rugarch
fit_rugarch <- ugarchfit(spec = spec_fit_rugarch, data = vDATA, solver = "nloptr") # or "nlop

print("--- rugarch Results ---")
#> [1] "--- rugarch Results ---"
```

```
print(fit_rugarch)
#>
#> *---------------------------------*
#> *          GARCH Model Fit        *
#> *---------------------------------*
#>
#> Conditional Variance Dynamics
#> -----------------------------------
#> GARCH Model  : sGARCH(2,1)
#> Mean Model   : ARFIMA(0,0,0)
#> Distribution : std
#>
#> Optimal Parameters
#> ------------------------------------
#>         Estimate  Std. Error    t value Pr(>|t|)
#> omega   0.000000    0.000000   0.048587  0.96125
#> alpha1  0.000071    0.022045   0.003227  0.99743
#> alpha2  0.000256    0.022012   0.011636  0.99072
#> beta1   0.969460    0.002236 433.632645  0.00000
#> shape   4.974784    0.730757   6.807711  0.00000
#>
#> Robust Standard Errors:
#>         Estimate  Std. Error  t value Pr(>|t|)
#> omega   0.000000    0.000000 0.000812 0.999352
#> alpha1  0.000071    0.051016 0.001394 0.998887
#> alpha2  0.000256    0.294820 0.000869 0.999307
#> beta1   0.969460    0.307556 3.152144 0.001621
#> shape   4.974784    8.785449 0.566253 0.571222
#>
#> LogLikelihood : 10763.72
#>
#> Information Criteria
#> ------------------------------------
#>
#> Akaike       -17.976
#> Bayes        -17.955
#> Shibata      -17.976
#> Hannan-Quinn -17.968
#>
#> Weighted Ljung-Box Test on Standardized Residuals
#> ------------------------------------
#>                         statistic p-value
#> Lag[1]                     0.1481  0.7004
#> Lag[2*(p+q)+(p+q)-1][2]    0.9820  0.5041
#> Lag[4*(p+q)+(p+q)-1][5]    2.1393  0.5859
#> d.o.f=0
#> H0 : No serial correlation
#>
#> Weighted Ljung-Box Test on Standardized Squared Residuals
#> ------------------------------------
#>                          statistic p-value
#> Lag[1]                    0.005858  0.9390
#> Lag[2*(p+q)+(p+q)-1][8]   0.966063  0.9763
#> Lag[4*(p+q)+(p+q)-1][14]  3.722254  0.9052
#> d.o.f=3
#>
#> Weighted ARCH LM Tests
#> ------------------------------------
#>             Statistic Shape Scale P-Value
#> ARCH Lag[4]    0.5723 0.500 2.000  0.4494
#> ARCH Lag[6]    0.7583 1.461 1.711  0.8178
#> ARCH Lag[8]    0.8049 2.368 1.583  0.9508
#>
#> Nyblom stability test
#> ------------------------------------
#> Joint Statistic:  280.0863
#> Individual Statistics:
#> omega  77.26025
#> alpha1  0.27309
#> alpha2  0.22811
#> beta1   0.16494
#> shape   0.03994
#>
#> Asymptotic Critical Values (10% 5% 1%)
#> Joint Statistic:          1.28 1.47 1.88
#> Individual Statistic:     0.35 0.47 0.75
#>
#> Sign Bias Test
#> ------------------------------------
#>                    t-value   prob sig
#> Sign Bias           0.3901 0.6965
#> Negative Sign Bias  0.1214 0.9034
#> Positive Sign Bias  0.2282 0.8195
#> Joint Effect        0.5120 0.9163
```

```
#>
#>
#> Adjusted Pearson Goodness-of-Fit Test:
#> ------------------------------------
#>   group statistic p-value(g-1)
#> 1    20     12.41       0.8675
#> 2    30     24.38       0.7101
#> 3    40     28.03       0.9041
#> 4    50     39.97       0.8178
#>
#>
#> Elapsed time : 0.88732
```

5.

```
// [[Rcpp::depends(RcppArmadillo)]]
#include <RcppArmadillo.h>

using namespace arma;
using namespace Rcpp;

// [[Rcpp::export]]
arma::vec CppGfunc(Function g, arma::vec x) {

  arma::vec vOut = zeros<vec>(x.size());

  for (arma::uword i = 0; i < x.size(); i++) {
    SEXP s;
    s = g(x[i]);
    vOut[i] = as<double>(s) + 2.0;
  }

  return(vOut);
}
```

```
suppressMessages(library(Rcpp))
#> Warning: pakke 'Rcpp' blev bygget under R version 4.3.3
suppressMessages(library(RcppArmadillo))
#> Warning: pakke 'RcppArmadillo' blev bygget under R version 4.3.3
```

```r
sourceCpp("exam2022recpp.cpp")
```

```r
CppGfunc(function(x) x^2, c(1,2,3))
#>      [,1]
#> [1,]    3
#> [2,]    6
#> [3,]   11
```

6.

24 Ordinary exam 2023

24.1 Problem 1:

1.

```r
set.seed(123)

Gamma.Simulate <- function(iK, dTheta, iSamples, iLength) {

  U <- runif(iSamples * iLength)
  vOut <- qgamma(U, shape = iK, scale = dTheta)
  return(matrix(vOut, nrow = iLength, ncol = iSamples))

}

mG1 <- Gamma.Simulate(iK = 2, dTheta = 1, iSamples = 100, iLength = 1000)
mG2 <- Gamma.Simulate(iK = 1, dTheta = 2, iSamples = 100, iLength = 1000)
```

2.

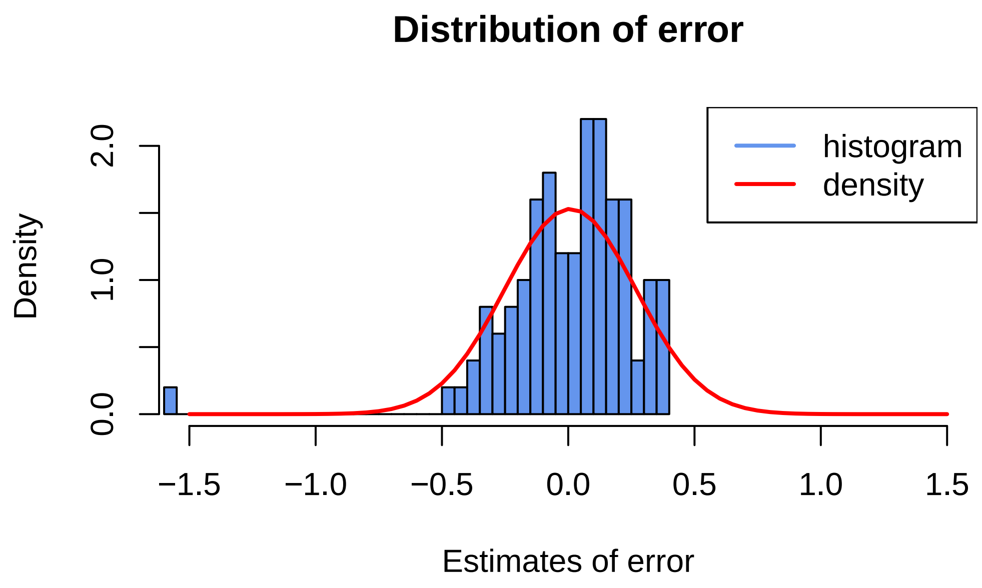
```r
mL <- mG1 - mG2
```

3.

```r
fn_estimateSkewness <- function(mInput) {
  fn_CalcSkew <- function(vX) {
    dNum <- mean((vX - mean(vX))^3)
    dDenom <- mean((vX - mean(vX))^2)^(3/2)
    return(dNum / dDenom)
  }
  return(apply(mInput, 2, fn_CalcSkew))
}

iK1 <- 2
iK2 <- 1
dTheta1 <- 1
dTheta2 <- 2

vSkHat <- fn_estimateSkewness(mL)
vSkAct <- (2 * iK1 * dTheta1^3 - 2 * iK2 * dTheta2^3) / (iK1 * dTheta1^2 + iK2 * dTheta2^2)^

vDiffSk <- vSkHat - vSkAct
```

4.

```r
hist(vDiffSk,
     freq = FALSE,
     breaks = 41,
     col = "cornflowerblue",
     xlab = "Estimates of error",
     main = "Distribution of error",
     xlim = c(-1.5, 1.5))

vX <- seq(-1.5, 1.5, 0.05)
lines(vX, dnorm(vX, mean = mean(vDiffSk), sd = sd(vDiffSk)), type = "l", col = "red", lwd = 

legend("topright",
       legend = c("histogram", "density"),
       col = c("cornflowerblue", "red"),
       lwd = 2)
```



24.2 Problem 2: (Constrained) Optimization, C++, and Packaging

1.

```r
set.seed(123)
genData <- function() {
  n <- 500  # Number of observations
  p <- 5    # Number of regressors (including intercept)

  # Define the true parameter vector theta.
  # We set theta_1 > 0 to make the hypothesis test H0: theta_1 <= 0 meaningful.
  theta_true <- c(0.8, -1.2, 0.5, 0.1, 2.0)

  # Create the regressor matrix X
  # First column is the intercept
  X <- matrix(rnorm(n * (p - 1)), nrow = n, ncol = p - 1)
  X <- cbind(1, X)
  colnames(X) <- paste0("X", 1:p)

  # Calculate the linear predictor and the mean lambda
  # eta = X * theta
  eta <- X %*% theta_true
  # lambda = exp(eta)
  lambda <- exp(eta)

  # Generate the dependent variable Y from the Poisson distribution
  Y <- rpois(n, lambda)
  return(list(X = X, Y = Y))
}

X <- genData()$X
Y <- genData()$Y
```

2.

```cpp
// [[Rcpp::depends(RcppArmadillo)]]
#include <RcppArmadillo.h>

using namespace arma;
using namespace Rcpp;

// [[Rcpp::export]]
double factorial_cpp(int iN) {
  if (iN < 0) {
    Rcpp::stop("Input to factorial must be a non-negative integer.");
  }
  if (iN == 0) {
    return 1.0;
  }
  double result = 1.0;
  for (int i = 1; i <= iN; ++i) {
    result *= i;
  }
  return result;
}
```

```r
suppressMessages(library(Rcpp))
#> Warning: pakke 'Rcpp' blev bygget under R version 4.3.3
suppressMessages(library(RcppArmadillo))
#> Warning: pakke 'RcppArmadillo' blev bygget under R version 4.3.3
sourceCpp("exam2023cpp.cpp")
```

3.

```cpp
// [[Rcpp::export]]
double fLnL_cpp(vec vParams, vec vY, mat mX) {
  double dSum = 0.0;
  int iN = mX.n_rows;

  for (int i = 0; i < iN; i++) {
    //dSum = dSum + vY[i] * as_scalar(mX.row(i) * vParams) - exp(as_scalar(mX.row(i) * vPara
    dSum = dSum + vY[i] * as_scalar(mX.row(i) * vParams) - exp(as_scalar(mX.row(i) * vParams
  }

  return(dSum / iN);
}

// [[Rcpp::export]]
vec fScore_2_cpp(vec vParams, vec vY, mat mX){
  vec vP = exp(mX*vParams);
  vec vScore = trans(mX)*(vY - vP) / vY.n_elem;

  return vScore;
}

// [[Rcpp::export]]
vec fScore_cpp(vec vParams, vec vY, mat mX){

  int n_obs = mX.n_rows; // Number of observations
  int p = mX.n_cols;     // Number of parameters

  vec vOut = zeros<vec>(p);

  for (int i = 0; i < n_obs; i++) {
    vOut += (vY[i] - exp(as_scalar(mX.row(i) * vParams))) * mX.row(i).t();
  }

  return vOut / n_obs;
}
```

```r
suppressMessages(library(Rcpp))
suppressMessages(library(RcppArmadillo))
sourceCpp("exam2023cpp.cpp")
```

4.

```
optim_res <- optim(c(0,0,0,0,0), fLnL_cpp, fScore_2_cpp, vY = Y, mX = X, method = "BFGS", co
optim_res
#> $par
#> [1]  3.630563467  0.126546522 -0.008899956  0.282246449  0.236120415
#> 
#> $value
#> [1] -114.4235
#> 
#> $counts
#> function gradient 
#>       55       19 
#> 
#> $convergence
#> [1] 0
#> 
#> $message
#> NULL
```

5.

(a)

(b)

(c)

(d)
(e)

```
LR_stat <- function(vY, mX, vInitGuessesCon = rep(0, ncol(mX)), vInitGuessesUncon = rep(0, nc
  if (all(vY == floor(vY)) == FALSE) {
    stop("y is not an integer vector. Stopping...")
  }

  # unconstrained max
  fitUnconMax <- optim(vInitGuessesUncon, fLnL_cpp, fScore_2_cpp, vY = Y, mX = X, method = "L

  # constrained max
  fitConMax <- optim(vInitGuessesCon, fLnL_cpp, fScore_2_cpp, vY = Y, mX = X, method = "L-BFG

  return(list(
    LR_test_stat = -2 * (fitConMax$value - fitUnconMax$value),
    MLE_estimator_uncon = fitUnconMax$par,
    convergence_messages = c(
      unconstrained = paste0("Convergence code:", fitUnconMax$convergence, "-", fitUnconMax$m
      constrained = paste0("Convergence code:", fitConMax$convergence, "-", fitConMax$message
}

LR_stat(vY = Y, mX = X)
#> $LR_test_stat
#> [1] 172.3262
#>
#> $MLE_estimator_uncon
#> [1]  3.630563467  0.126546522 -0.008899956  0.282246449  0.236120415
#>
#> $convergence_messages
#>                                                              unconstrained
#>                                                      "Convergence code:0-"
#>                                                                constrained
#> "Convergence code:0-CONVERGENCE: REL_REDUCTION_OF_F <= FACTR*EPSMCH"
```

6.

25 Retake exam 2023

25.1 Problem 1:

1.

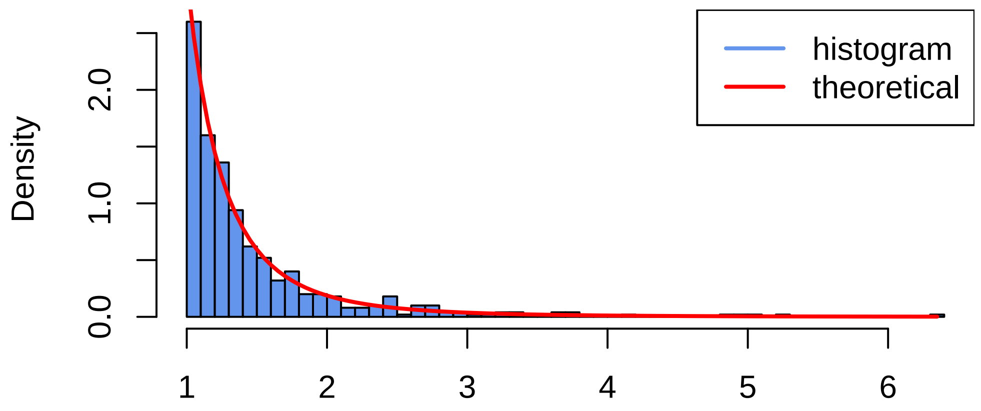
```r
set.seed(134)

fn_denPareto <- function(x, gamma, alpha) {
  vOut <- ifelse(x < gamma, 0, alpha * gamma^alpha / x^(alpha + 1))
  return(vOut)
}
```

2.

   (a)

   (b)

```
fn_ParetoSimul <- function(gamma, alpha, samples, length) {

  U <- runif(samples * length)
  vSim <- gamma * (1 - U)^(-1/alpha)
  return(matrix(vSim, nrow = length, ncol= samples))

}

mX <- fn_ParetoSimul(gamma = 1, alpha = 3, samples = 100, length = 500)

hist(mX[, 1],
     freq = FALSE,
     breaks = 41,
     col = "cornflowerblue",
     xlab = "",
     ylab = "Density",
     main = "",
     xlim = c(min(mX[, 1]), max(mX[, 1])))

vX <- seq(min(mX[, 1]), max(mX[, 1]), by = 0.05)
lines(vX, fn_denPareto(vX, 1, 3), type = "l", col = "red", lwd = 2)

legend("topright",
       legend = c("histogram", "theoretical"),
       col = c("cornflowerblue", "red"),
       lwd = 2)
```

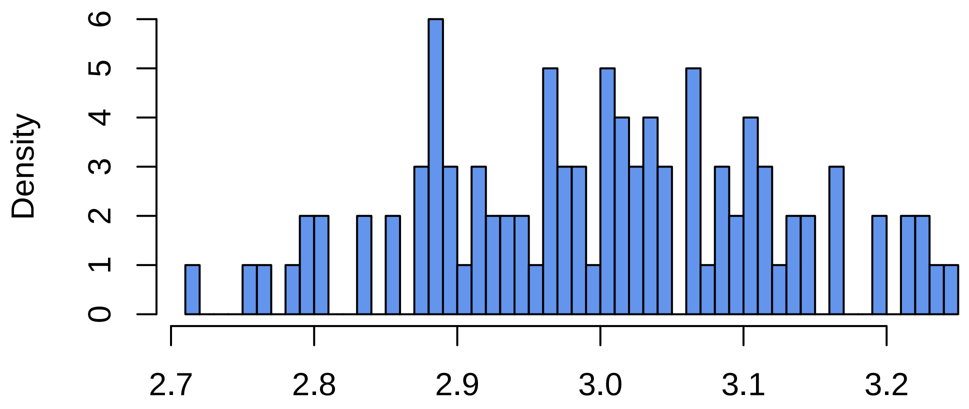


3.

```r
fn_estimateParams <- function(mInput) {
  fAlphas <- function(vX) {
    vOut <- length(vX) / sum(log(vX / min(vX)))
    return(vOut)
  }

  vGammas <- apply(mInput, 2, min)
  vAlphas <- apply(mInput, 2, fAlphas)

  return(list(
    alphas = vAlphas,
    gammas = vGammas
  ))
}

vAlpha <- fn_estimateParams(mX)$alphas
```

4.

```
hist(vAlpha,
     freq = FALSE,
     breaks = 41,
     col = "cornflowerblue",
     xlab = "",
     ylab = "Density",
     main = "",
     xlim = c(min(vAlpha), max(vAlpha)))
```



25.2 Problem 2: (Constrained) Optimization, C++, and Packaging

1.

```r
set.seed(123)

genData <- function() {
  mu_true <- 0.5
  phi_true <- 0.7
  sigma2_true <- 1.5
  T_val <- 500

  # Ensure parameters meet conditions
  if (abs(phi_true) >= 1) {
    stop("phi must be between -1 and 1 for stationarity.")
  }
  if (sigma2_true <= 0) {
    stop("sigma2 must be positive.")
  }

  # Initialize the time series vector
  y <- numeric(T_val)

  # 1. Generate y_1 from its stationary distribution
  # y_1 ~ N(mu / (1 - phi), sigma^2 / (1 - phi^2))
  mean_y1 <- mu_true / (1 - phi_true)
  var_y1 <- sigma2_true / (1 - phi_true^2)
  sd_y1 <- sqrt(var_y1)

  y[1] <- rnorm(1, mean = mean_y1, sd = sd_y1)

  # 2. Generate y_t for t = 2, ..., T
  # y_t = mu + phi * y_{t-1} + epsilon_t, epsilon_t ~ N(0, sigma^2)
  sd_epsilon <- sqrt(sigma2_true)
  for (t in 2:T_val) {
    epsilon_t <- rnorm(1, mean = 0, sd = sd_epsilon)
    y[t] <- mu_true + phi_true * y[t-1] + epsilon_t
  }

  return(y)
}

y <- genData()
```

2.

```
// [[Rcpp::depends(RcppArmadillo)]]
#include <RcppArmadillo.h>

using namespace arma;
using namespace Rcpp;

// [[Rcpp::export]]
double f_Y1(double y1, vec vParams) {
  double dPi = atan(1)*4;
  double dMu = vParams[0];
  double dPhi = vParams[1];
  double dSigma2 = vParams[2];

  double dOut = 1 / pow(2 * dPi * dSigma2 / (1 - pow(dPhi, 2)), 0.5) * exp(-0.5 * pow((y1 - 

  return dOut;
}

// [[Rcpp::export]]
double f_cond(double y1, double yt_1, vec vParams) {
  double dPi = atan(1)*4;
  double dMu = vParams[0];
  double dPhi = vParams[1];
  double dSigma2 = vParams[2];

  double dOut = 1 / pow(2 * dPi * dSigma2, 0.5) * exp(-0.5 * pow((y1 - dMu - dPhi * yt_1), 2

  return dOut;
}
```

```
suppressMessages(library(Rcpp))
#> Warning: pakke 'Rcpp' blev bygget under R version 4.3.3
suppressMessages(library(RcppArmadillo))
#> Warning: pakke 'RcppArmadillo' blev bygget under R version 4.3.3
sourceCpp("exam2023recpp.cpp")
```

3.

```
fLnL_rep <- function(vParams, vY) {
  dMu <- vParams[1]
  dPhi <- -1 + 2 * (exp(vParams[2]) / (1 + exp(vParams[2])))
  dSigma2 <- exp(vParams[3])

  dT <- length(vY)
  dSum <- log(f_Y1(vY[1], c(dMu, dPhi, dSigma2)))
  for (t in 2:dT) {
    dSum <- dSum + log(f_cond(vY[t], vY[t - 1], c(dMu, dPhi, dSigma2)))
  }

  return(dSum / dT)
}
```

4.

```
vParams <- c(0, 0, 1)
vParams[2] <- log(((vParams[2] + 1) / 2) / (1 - ((vParams[2] + 1) / 2.0)))
vParams[3] <- log(vParams[3])
optim_res <- optim(vParams, fLnL_rep, vY = y, method = "BFGS", control=list(fnscale=-1))
vParams <- optim_res$par

dMu <- vParams[1]
dPhi <- -1 + 2 * (exp(vParams[2]) / (1 + exp(vParams[2])))
dSigma2 <- exp(vParams[3])

print(paste0("Est. Mu: ", dMu, " vs act.: ", 0.5))
#> [1] "Est. Mu: 0.67406977684722 vs act.: 0.5"
print(paste0("Est. Phi: ", dPhi, " vs act.: ", 0.7))
#> [1] "Est. Phi: 0.62648860056871 vs act.: 0.7"
print(paste0("Est. Sigma2: ", dSigma2, " vs act.: ", 1.5))
#> [1] "Est. Sigma2: 1.40416059123645 vs act.: 1.5"
print(paste0("Maximized average log-likelihood: ", optim_res$value))
#> [1] "Maximized average log-likelihood: -1.58915508954246"
```

5.

```
arma::vec simulateAR1_buggy(double mu_true = 0.5, double phi_true = 0.7, double sigma2_true =

  if (abs(phi_true) >= 1) {
    stop("phi must be between -1 and 1 for stationarity.");
  }
  if (sigma2_true <= 0) {
    stop("sigma2 must be positive.");
  }

  arma::vec y = zeros<vec>(T_val);

  double mean_y1 = mu_true / (1 - phi_true);
  double var_y1 = sigma2_true / (1 - pow(phi_true, 2));
  double sd_y1 = sqrt(var_y1);

  y[0] = Rf_rnorm(mean_y1, sd_y1);

  double sd_epsilon = sqrt(sigma2_true);
  double epsilon_t = 0.0;
  for (int t = 1; t < T_val; t++) {
    epsilon_t = Rf_rnorm(0, sd_epsilon);
    y[t] = mu_true + phi_true * y[t-1] + epsilon_t;
  }

  return(y);
}
```

```
suppressMessages(library(Rcpp))
suppressMessages(library(RcppArmadillo))
sourceCpp("exam2023recpp.cpp")

vY <- simulateAR1_buggy(0.5, 0.7, 1.5, 500)
```

```
vParams <- c(0, 0, 1)
vParams[2] <- log(((vParams[2] + 1) / 2) / (1 - ((vParams[2] + 1) / 2.0)))
vParams[3] <- log(vParams[3])
optim_res <- optim(vParams, fLnL_rep, vY = vY, method = "BFGS", control=list(fnscale=-1))
vParams <- optim_res$par
```

```r
dMu <- vParams[1]
dPhi <- -1 + 2 * (exp(vParams[2]) / (1 + exp(vParams[2])))
dSigma2 <- exp(vParams[3])

print(paste0("Est. Mu: ", dMu, " vs act.: ", 0.5))
#> [1] "Est. Mu: 0.50608151243852 vs act.: 0.5"
print(paste0("Est. Phi: ", dPhi, " vs act.: ", 0.7))
#> [1] "Est. Phi: 0.694273961540576 vs act.: 0.7"
print(paste0("Est. Sigma2: ", dSigma2, " vs act.: ", 1.5))
#> [1] "Est. Sigma2: 1.52925097542527 vs act.: 1.5"
```

6.

26 Ordinary exam 2024

26.1 Problem 1

26.1.1 Problem 1.1

1.
    -
    -
    -
    -

```r
set.seed(123)

rndRayleigh <- function(lambda, size) {

  # lambda - intensity of the exponential distribution
  # size - number of i.i.d. random variables to be drawn

  U <- runif(size)
  return(-1/lambda * log(U))

}
set.seed(123) # set seed

# matrix size and default variables
iT <- 1000
```

```r
iN <- 100
lambda <- 2

# simulate data, y
y <- rndRayleigh(lambda, iT * iN)

x <- sqrt(y)

mR <- matrix(x, iT, iN)
```

26.1.2 Problem 1.2

2.

-
-
-

```r
# defining the constants
gamma <- 1 / (sqrt(2 * lambda))
mu <- gamma * sqrt(2 / pi)
sigma2 <- (4 - pi) / 2 * gamma^2

# mean and variance of each column
mu_hat <- colMeans(mR)
sigma2_hat <- apply(mR, 2, var)

# store the errors
vmuError <- mu_hat - mu
vsigmaError <- sigma2_hat - sigma2

# calculate mean and std of these
mean(vmuError)
#> [1] 0.228063
mean(vsigmaError)
```

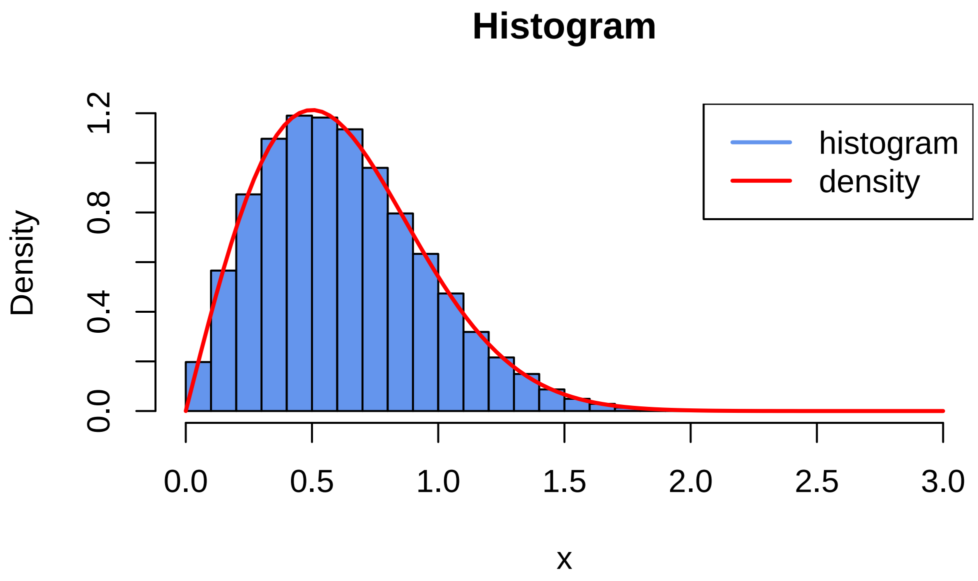
```
#> [1] -0.0006383356
sqrt(var(vmuError))
#> [1] 0.009235813
sqrt(var(vsigmaError))
#> [1] 0.004542095
```

26.1.3 Problem 1.3

3.
   -
   -
   -

```
# histogram
hist(mR,
     freq = FALSE,
     breaks = 30,
     col = "cornflowerblue",
     xlab = "x",
     ylab = "Density",
     main = "Histogram",
     xlim = c(0, 3))

# superimpose the density function
fDensity <- function(x, gamma) {
  return((x / gamma^2) * exp(-(x^2 / (2*gamma^2))))
}

curve(fDensity(x, gamma),
      from = 0,
      to = 3,
      col = "red",
      lwd = 2,
      add = TRUE)
```

```
legend("topright",
       legend = c("histogram", "density"),
       col = c("cornflowerblue", "red"),
       lwd = 2)
```

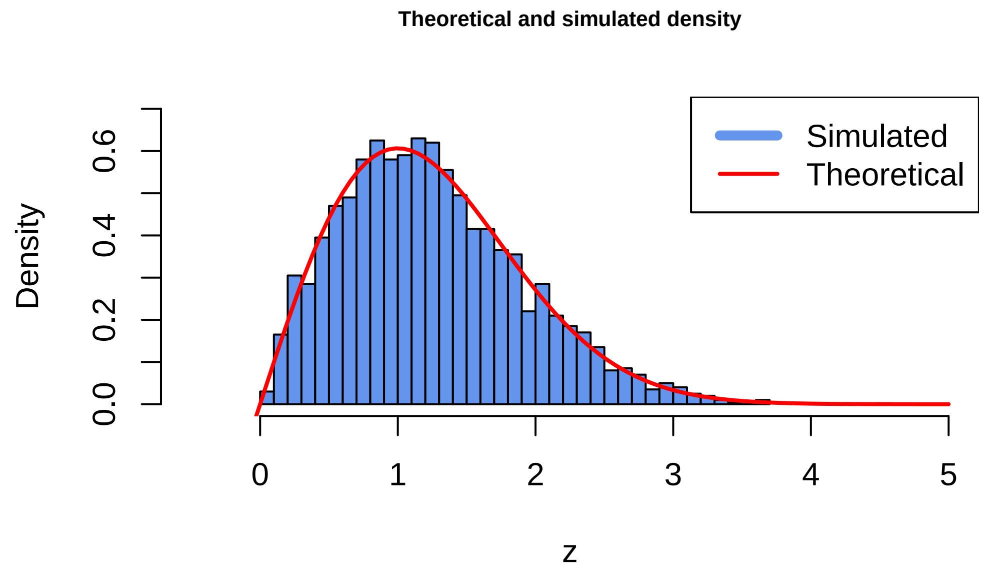


26.1.4 Problem 1.4

4.

-
-
-
-

```
BoxMuller <- function(size = 1) {

  # size: number of standard bivariate normal random variables to simulate

  U <- runif(size)
  V <- runif(size)

  X <- sqrt(-2*log(U)) * cos(2*pi*V)
  Y <- sqrt(-2*log(U)) * sin(2*pi*V)

  return(c(X,Y))

}

y_1 <- BoxMuller(1000)
y_2 <- BoxMuller(1000)

vZ <- sqrt(y_1^2 + y_2^2)

# histogram
hist(vZ, breaks = 35, freq = FALSE,
     main = "Theoretical and simulated density",
     col = "cornflowerblue",
     xlim = c(-0.5, 5), ylim = c(0, 0.7),
     xlab = "z",
     cex.main = 0.7)

fDensityNew <- function(z) {
  return(z * exp(- z^2/2))
}
x <- vZ
curve(fDensityNew(x),
      from = -0.5,
      to = 5,
      col = "red",
      lwd = 2,
      add = TRUE)

legend("topright", legend = c("Simulated", "Theoretical"), lty = c(1, 1), lwd = c(5, 2), col
```



26.2 Problem 2: (Constrained) Optimization, C++, and Packaging

26.2.1 Problem 2.1

1.

```r
vD <- readRDS("d.R")
```

26.2.2 Problem 2.2

2. (a)

```r
log_n_choose_k <- function(n, k) {
  if (k == 0) {
    return(0)
  } else {
    dSumPart1 <- 0
    dSumPart2 <- 0
    for (i in (n - k + 1):n) {
      dSumPart1 <- dSumPart1 + log(i)
    }
    for (i in 1:k) {
      dSumPart2 <- dSumPart2 + log(i)
    }
    return(dSumPart1 - dSumPart2)
  }
}
```

(b)

```cpp
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
double log_n_choose_k_cpp(int n, int k) {

  if (k == 0) {
    return(0);
  } else {
    double dSumPart1 = 0;
    double dSumPart2 = 0;
    for (int i = (n - k + 1); i <= n; i++) {
      dSumPart1 = dSumPart1 + log(i);
    }
    for (int i = 1; i <= k; i++) {
      dSumPart2 = dSumPart2 + log(i);
    }
    return(dSumPart1 - dSumPart2);
  }
}
```

```r
suppressMessages(library(Rcpp))
#> Warning: pakke 'Rcpp' blev bygget under R version 4.3.3
sourceCpp("cpp_functions.cpp")

# naive check that the functions are equal
log_n_choose_k(10, 3)
#> [1] 4.787492
log_n_choose_k_cpp(10, 3)
#> [1] 4.787492
```

26.2.3 Problem 2.3

3. (a)

```r
fLnL <- function(p, n, d) {
  result <- numeric(length(d))

  for (i in 1:length(d)) {
    result[i] <- log_n_choose_k(n, d[i]) + d[i] * log(p) + (n - d[i]) * log(1 - p)
  }

  return(mean(result))
}

fScore <- function(p, n, d) {
  return(mean(d / p - (n - d) / (1 - p)))
}

fSD <- function(p, n, d) {
  return(mean(-(d / p^2) - (n - d) / ((1 - p)^2)))
}
```

(b)

```cpp
// [[Rcpp::export]]
double fLnL_cpp(double p, int n, vec d) {
  int len = d.n_elem;
  vec result(len);

  for (int i = 0; i < len; i++) {
    result[i] = log_n_choose_k_cpp(n, d[i]) + d[i] * log(p) + (n - d[i]) * log(1 - p);
  }
  return(mean(result));
}

// [[Rcpp::export]]
double fScore_cpp(double p, int n, vec d) {
  return(mean(d / p - (n - d) / (1 - p)));
}

// [[Rcpp::export]]
```

```
double fSD_cpp(double p, int n, vec d) {
  return(mean(-(d / pow(p, 2)) - (n - d) / pow((1 - p), 2)));
}
```

26.2.4 Problem 2.4

4.

```
Newton <- function(f, f_prime, f_sec, X0, Tol = 1e-9, max.iter = 1000, ...){
  dX <- X0
  fx <- f(dX, ...)
  fpx <- f_prime(dX, ...)
  fsx <- f_sec(dX, ...)
  i <- 0
  while ((abs(fpx) > Tol) && (i < max.iter)) {
    dX <- dX - fpx/fsx
    fx <- f(dX, ...)
    fpx <- f_prime(dX, ...)
    fsx <- f_sec(dX, ...)
    i <- i + 1
    #cat("At iteration", n, "the value of x is:", dX, "\n")
  }
  if (i == max.iter) {
    return(
      list(
       maximizing.val = NULL,
       maximum = NULL,
       "Algorithm failed to converge. Maximum iterations reached.")
      )
  } else {
    return(
      list(
       maximizing.val = dX,
       maximum = f(dX, ...),
       "Algorithm converged.")
      )
  }
}

# Newton max
Newton(fLnL_cpp, fScore_cpp, fSD_cpp, 0.5, n = 10, d = vD)
#> $maximizing.val
#> [1] 0.2032
#>
#> $maximum
#> [1] -1.543041
#>
#> [[3]]
#> [1] "Algorithm converged."

# Analytical max
p.star <- mean(vD) / 10
p.star
#> [1] 0.2032
```

26.2.5 Problem 2.5

5. (a)

```
fLnL_t <- function(p, n, d) {
  new_p <- 0.1 + 0.8 * (exp(p)) / (1 + exp(p))
  result <- numeric(length(d))

  for (i in 1:length(d)) {
    result[i] <- log_n_choose_k(n, d[i]) + d[i] * log(new_p) + (n - d[i]) * log(1 - new_p)
  }

  return(mean(result))
}
```

(b)

```
dOptimalP <- optimize(fLnL_t, lower = -1000, upper = 1000, n = 10, d = vD, maximum = TRUE)$ol

dOptimalP.orig.scale <- 0.1 + 0.8 * (exp(dOptimalP)) / (1 + exp(dOptimalP))

dOptimalP.orig.scale
#> [1] 0.1942476
```

26.2.6 Problem 2.6

6.

27 Retake exam 2024

27.1 Problem 1

27.1.1 Problem 1.1

1.
    -

-
-

```r
MonteCarlo.Integration <- function(f, n, a, b) {

  U <- runif(n, min = a, max = b)
  return( (b-a)*mean(f(U)) )

}

set.seed(123)

dAlpha <- 0.95
```

```
dLambda <- 2

# Monte Carlo Integration
dGammaMI <- MonteCarlo.Integration(function(x) (-(log(1-x)) / dLambda), 100, dAlpha, 1)
dGammaMI
#> [1] 0.09927233

# Numerical evaluation
dGammaNE <- (-log(1-dAlpha) + 1) / dLambda
dGammaNE
#> [1] 1.997866

# Thus, we should have this relationship
# dGammaNE * (1 - dAlpha) approx dGammaMI
dGammaNE * (1 - dAlpha)
#> [1] 0.09989331
```

27.1.2 Problem 1.2

```
mU <- matrix(runif(40000), 200, 200)
mU <- ifelse(mU < 0.5, -1, 1)

sum(mU == 1)
#> [1] 19715
sum(mU == -1)
#> [1] 20285
```

2.

-
-
-

27.1.3 Problem 1.3

3.

-

-

```
fPower <- function(z, k) {
  if (k == 0) {
    return(1)
  } else if (k > 0) {
    return(z * fPower(z, k - 1))
  } else {
    return(NULL)
  }
}
# Test
fPower(2, 4)
#> [1] 16
```

27.1.4 Problem 1.4

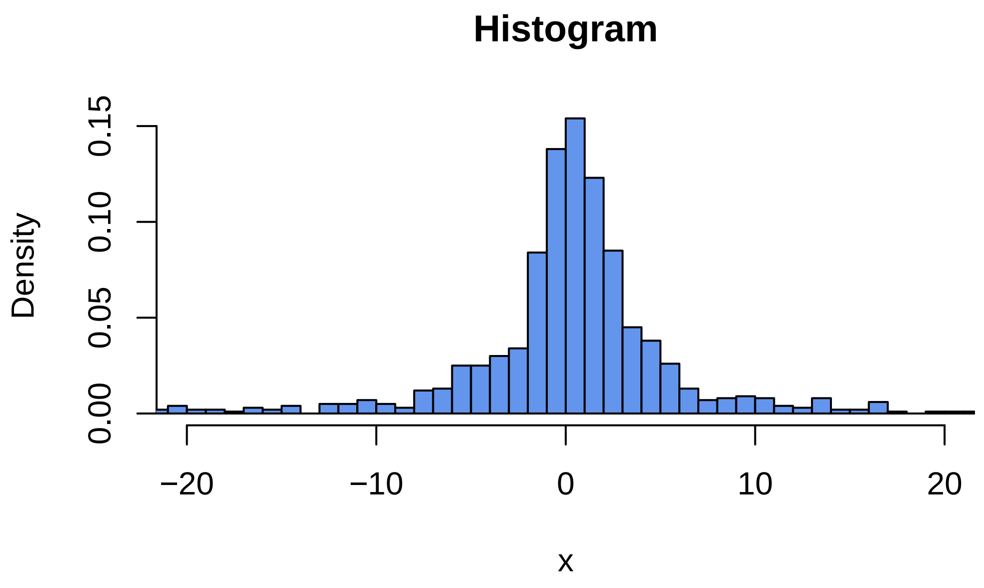
4.

   -
   -
   -

   -

```
set.seed(134)
dMu <- 0.5
dGamma <- 2

Cauchy.Simulate <- function(mu, lambda, size) {
```

```r
  U <- runif(size)
  return(mu + lambda * tan(pi * (U - 1/2)))
}

vX <- Cauchy.Simulate(dMu, dGamma, 1000)
hist(vX,
     freq = FALSE,
     breaks = 5000,
     col = "cornflowerblue",
     xlab = "x",
     ylab = "Density",
     main = "Histogram",
     xlim = c(-20, 20))
```



27.2 Problem 2: (Constrained) Optimization, C++, and Packaging

27.2.1 Problem 2.1

1. (a)

```r
fRosen <- function(vX) {
  return(-(pi - vX[1])^2 - 100 * (vX[2] - vX[1]^2)^2)
}

fRosen_grad <- function(vX) {
  vOutput <- numeric(2)
  vOutput[1] <- 2 * pi - 2 * vX[1] - 400 * vX[1]^3 + 400 * vX[1] * vX[2]
  vOutput[2] <- 200 * (vX[1]^2 - vX[2])
  return(vOutput)
}
```

(b)

```cpp
// [[Rcpp::depends(RcppArmadillo)]]
#include <RcppArmadillo.h>
using namespace Rcpp;
using namespace arma;

// [[Rcpp::export]]
double fRosen_cpp(vec vX) {
  double pi = atan(1)*4; //pi
  return(-pow((pi - vX[0]), 2) - 100 * pow((vX[1] - pow(vX[0], 2)), 2));
}
```

```
// [[Rcpp::export]]
vec fRosen_grad_cpp(vec vX) {
  vec vOutput(2);
  double pi = atan(1)*4; //pi
  vOutput[0] = 2 * pi - 2 * vX[0] - 400 * pow(vX[0], 3) + 400 * vX[0] * vX[1];
  vOutput[1] = 200 * (pow(vX[0], 2) - vX[1]);
  return(vOutput);
}
```

```
suppressMessages(library(Rcpp))
#> Warning: pakke 'Rcpp' blev bygget under R version 4.3.3
suppressMessages(library(RcppArmadillo))
#> Warning: pakke 'RcppArmadillo' blev bygget under R version 4.3.3
sourceCpp("cpp_functions_retake_2024.cpp")

# sanity checks
fRosen(c(0.5, 2))
#> [1] -313.228
fRosen_cpp(c(0.5, 2))
#> [1] -313.228
fRosen_grad(c(0.5, 2))
#> [1]  355.2832 -350.0000
fRosen_grad_cpp(c(0.5, 2))
#>           [,1]
#> [1,]  355.2832
#> [2,] -350.0000
```

27.2.2 Problem 2.2

2. (a)

```
# golden section algorithm from last week
gsection <- function(f, dX.l, dX.r, dX.m, dTol = 1e-9) {
  # golden ratio plus one
  dGR1 <- 1 + (1 + sqrt(5))/2

  # successively refine x.l, x.r, and x.m
  f.l <- f(dX.l)
  f.r <- f(dX.r)
  f.m <- f(dX.m)
  while ((dX.r - dX.l) > dTol) {
    if ((dX.r - dX.m) > (dX.m - dX.l)) { # if the right segment is wider than the left
      dY <- dX.m + (dX.r - dX.m)/dGR1 # put Y into the right segment according to the golden
      f.y <- f(dY)
      if (f.y >= f.m) {
        dX.l <- dX.m
        f.l <- f.m
        dX.m <- dY
        f.m <- f.y
      } else {
        dX.r <- dY
        f.r <- f.y
      }
    } else { #if the left segment is wider than the right
      dY <- dX.m - (dX.m - dX.l)/dGR1 # put Y into the left segment according to the golden
      f.y <- f(dY)
      if (f.y >= f.m) {
        dX.r <- dX.m
        f.r <- f.m
        dX.m <- dY
        f.m <- f.y
      } else {
        dX.l <- dY
        f.l <- f.y
      }
    }
  }
  return(dX.m)
}

line.search <- function(f, vX, vG, dTol = 1e-9, dA.max = 2^5) {
  # f is a real function that takes a vector of length d
  # x and y are vectors of length d
  # line.search uses gsection to find a >= 0 such that
  # g(a) = f(x + a*y) has a local maximum at a,
```

```r
  # within a tolerance of tol
  # if no local max is found then we use 0 or a.max for a
  # the value returned is x + a*y
  if (sum(abs(vG)) == 0){
    return(vX) # +0*vG
  } # g(a) constant
  g <- function(dA){
    return(f(vX + dA*vG))
  }
  # find a triple a.l < a.m < a.r such that
  # g(a.l) <= g(a.m) and g(a.m) >= g(a.r)

  # choose a.l
  dA.l <- 0
  g.l <- g(dA.l)
  # find a.m
  dA.m <- 1
  g.m <- g(dA.m)
  while ((g.m < g.l) & (dA.m > dTol)) {
    dA.m <- dA.m/2
    g.m <- g(dA.m)
  }
  # if a suitable a.m was not found then use 0 for a, so just return vX as the next step
  if ((dA.m <= dTol) & (g.m < g.l)){
    return(vX)
  }
  # find a.r
  dA.r <- 2*dA.m
  g.r <- g(dA.r)
  while ((g.m < g.r) & (dA.r < dA.max)) {
    dA.m <- dA.r
    g.m <- g.r
    dA.r <- 2*dA.m
    g.r <- g(dA.r)
  }
  # if a suitable a.r was not found then use a.max for a
  if ((dA.r >= dA.max) & (g.m < g.r)){
    return(vX + dA.max*vG)
  }
  # apply golden-section algorithm to g to find a
  dA <- gsection(g, dA.l, dA.r, dA.m)
  return(vX + dA*vG)
```

```r
}

ascent <- function(f, grad.f, vX0, dTol = 1e-9, n.max = 100) {
  vX.old <- vX0
  vX <- line.search(f, vX0, grad.f(vX0))
  n <- 1
  while ((f(vX) - f(vX.old) > dTol) & (n < n.max)) {
    vX.old <- vX
    vX <- line.search(f, vX, grad.f(vX))
    #cat("at iteration", n, "the coordinates of x are", vX, "\n")
    n <- n + 1
  }
  if (n == n.max) {
    return(
      list(
       maximizing.val = NA,
       maximum = -Inf,
       paste0("Algorithm failed to converge. Maximum iterations reached. Last values:", vX))
      )
  } else {
    return(
      list(
       maximizing.val = vX,
       maximum = f(vX),
       "Algorithm converged.")
      )
  }
}
```

(b)

```r
ascent(fRosen_cpp, fRosen_grad_cpp, vX0 = c(1, 1))
#> $maximizing.val
#> [1] NA
#>
```

```
#> $maximum
#> [1] -Inf
#>
#> [[3]]
#> [1] "Algorithm failed to converge. Maximum iterations reached. Last values:1.207130415917
#> [2] "Algorithm failed to converge. Maximum iterations reached. Last values:1.457163851177
```

27.2.3 Problem 2.3

3. (a)

```
grid_ascent <- function(f, grad.f, vX0, vY0, dTol = 1e-9, n.max = 100) {
  mOutput <- matrix(NA, length(vX0), length(vY0))

  for (x in 1:length(vX0)) {
    for (y in 1:length(vY0)) {
      vXY0 <- c(vX0[x], vY0[y])
      vXY.old <- vXY0
      vXY <- line.search(f, vXY0, grad.f(vXY0))
      n <- 1
      while ((f(vXY) - f(vXY.old) > dTol) & (n < n.max)) {
        vXY.old <- vXY
        vXY <- line.search(f, vXY, grad.f(vXY))
        n <- n + 1
      }
      if (n == n.max) {
        mOutput[x, y] <- -Inf
      } else {
        mOutput[x, y] <- f(vXY)
      }
    }
  }
  return(mOutput)
}
```

(b)

```
grid_ascent(fRosen_cpp, fRosen_grad_cpp, vX0 = c(2.5, 2.75, 3, 3.25, 3.5), vY0 = c(8.5, 8.75
#>      [,1]          [,2] [,3]          [,4] [,5]
#> [1,] -Inf          -Inf -Inf          -Inf -Inf
#> [2,] -Inf          -Inf -Inf          -Inf -Inf
#> [3,] -Inf -2.871553e-10 -Inf          -Inf -Inf
#> [4,] -Inf          -Inf -Inf -1.443747e-09 -Inf
#> [5,] -Inf          -Inf -Inf          -Inf -Inf
```

(c)

```
ascent(fRosen_cpp, fRosen_grad_cpp, vX0 = c(3, 8.75))
#> $maximizing.val
#>          [,1]
#> [1,] 3.141604
#> [2,] 9.869677
#>
#> $maximum
#> [1] -2.871553e-10
#>
#> [[3]]
#> [1] "Algorithm converged."
ascent(fRosen_cpp, fRosen_grad_cpp, vX0 = c(3.25, 9.25))
#> $maximizing.val
#>          [,1]
#> [1,] 3.141555
#> [2,] 9.869366
#>
#> $maximum
#> [1] -1.443747e-09
#>
#> [[3]]
#> [1] "Algorithm converged."
```

27.2.4 Problem 2.4

4.

```
optim(par = c(1, 1), fn = fRosen_cpp, gr = fRosen_grad_cpp, method = "BFGS", control = list(
#> $par
#> [1] 3.141593 9.869604
#>
#> $value
#> [1] -7.901082e-19
#>
#> $counts
#> function gradient
#>      154       62
#>
#> $convergence
#> [1] 0
#>
#> $message
#> NULL
```

27.2.5 Problem 2.5

5.

28 Extra exercises

29 Exercise Set 1: R Basics, Data Structures, and Control Flow

29.1 (1)

- a)
- b)
- c)

- d)

```r
vNumbers <- 1:50
sum(vNumbers[vNumbers %% 2 == 0])
#> [1] 650
vLog <- log(vNumbers)
typeof(vLog)
#> [1] "double"
```

29.2 (2)

- a)

- b)
- c)
- d)

```r
set.seed(42)
mRandom <- matrix(runif(25, -10, 10), 5, 5)
mRandom[mRandom < 0] <- NA
rowSums(mRandom, na.rm = TRUE)
#> [1] 25.558958 27.420048 18.471280 18.682163  8.142861
colMeans(mRandom, na.rm = TRUE)
#> [1] 6.620373 3.088707 6.537845 6.523825 8.930609
```

29.3 (3)

-
    -
    -
    -
    -

```r
dTotalSum <- 0
for (i in 1:100) {
  if (i %% 3 == 0 | i %% 5 == 0) {
    dTotalSum <- dTotalSum + i
  }
}
dTotalSum
#> [1] 2418
```

29.4 (4)

-
    -
    -
    -
    -
-
-

```r
lMyList <- list(
  sName = "MyName",
  vLuckyNumbers = c(1, 3, 5),
  mMatrixA = identity(2),
  dfInnerData = data.frame(
    Month = c("Jan", "Feb", "Mar"),
    Rainfall_mm = c(50, 30, 65)
  )
)
lMyList$dfInnerData$Rainfall_mm[lMyList$dfInnerData$Month == "Feb"]
```

```
#> [1] 30
lMyList$sCourse <- "PQE"
```

29.5 (5)

- a)

- b)
- c)
- d)

```
dfStudents <- data.frame(
  StudentID = 101:105,
  Grade = c("A", "B", "C", "A", "B"),
  Age = c(20, 21, 20, 22, 21)
)
dfStudents$Grade <- factor(dfStudents$Grade, levels = c("C", "B", "A"), ordered = TRUE)
dfStudents$Pass <- ifelse(dfStudents$Grade == "A" | dfStudents$Grade == "B", TRUE, FALSE)
dfStudents[dfStudents$Age > 20, ]
#>   StudentID Grade Age Pass
#> 2       102     B  21 TRUE
#> 4       104     A  22 TRUE
#> 5       105     B  21 TRUE
```

29.6 (6)

-

```
vI <- -9:10
vSign <- ifelse(vI < 0, -1, ifelse(vI == 0, 0, 1))
vSign
#>  [1] -1 -1 -1 -1 -1 -1 -1 -1 -1  0  1  1  1  1  1  1  1  1  1  1
```

29.7 (7)

-

```r
set.seed(101)
iFlipCount <- 0
iConsecHeads <- 0
while (iConsecHeads < 3) {
  iFlip <- sample(0:1, 1)
  iFlipCount <- iFlipCount + 1
  if (iFlip == 1) {
    iConsecHeads <- iConsecHeads + 1
  } else {
    iConsecHeads <- 0
  }
}
iFlipCount
#> [1] 36
```

29.8 (8)

- a)
- b)
- c)
- d)

```r
vsFruits <- c("apple", "banana", "cherry", NA, "date")
sum(is.na(vsFruits))
#> [1] 1
vsFruitsClean <- vsFruits[which(!is.na(vsFruits))]
for (fruit in vsFruitsClean) {
  print(fruit)
  print(nchar(fruit))
}
#> [1] "apple"
#> [1] 5
#> [1] "banana"
#> [1] 6
#> [1] "cherry"
#> [1] 6
#> [1] "date"
#> [1] 4
```

29.9 (9)

-

```
#`&` and `|` are element-wise. `&&` and `||` are short-circuiting, evaluating only the first

x <- NULL
# No error
if (FALSE && x[1] > 5) {
  print("Not evaluated")
} else {
  print("OK")
}
# Error if x[1] is problematic
if (FALSE & x[1] > 5) {
  print("Error here")
}
```

29.10 (10)

-
-
-
-
-

```
mA <- matrix(1:12, nrow = 4, byrow = TRUE)
mA[2, ]
#> [1] 4 5 6
mA[, 3]
#> [1]  3  6  9 12
```

```
mA[1, 2]
#> [1] 2
mB <- mA[c(1, 3), c(2, 3)]
mB
#>      [,1] [,2]
#> [1,]    2    3
#> [2,]    8    9
```

30 Exercise Set 2: Functions, Scope, and Efficiency

30.1 (1)

-
-
  -
  -
  -
  -
-
-

```r
fCalculateStats <- function(vInput) {
  outputList <- list(
    dMean = mean(vInput, na.rm = TRUE),
    dMedian = median(vInput, na.rm = TRUE),
    dSD = sqrt(var(vInput, na.rm = TRUE)),
    iNAs = sum(is.na(vInput))
  )
  return(outputList)
}
vTest <- c(1, 5, NA, 7, 3, NA, 8)
fCalculateStats(vTest)
#> $dMean
#> [1] 4.8
#>
#> $dMedian
#> [1] 5
#>
#> $dSD
#> [1] 2.863564
```

```
#>
#> $iNAs
#> [1] 2
```

30.2 (2)

-

```
x <- 10
my_func <- function(y) {
  x <- 5
  return(x + y)
}
z <- my_func(3)
```

-

30.3 (3)

-
-
-
-

```
suppressMessages(library(microbenchmark))
#> Warning: pakke 'microbenchmark' blev bygget under R version 4.3.3

fSumSquaresUpToN <- function(iN) {
  iSumSquares <- 0
  for (i in 1:iN) {
    iSumSquares <- iSumSquares + i^2
  }
}
```

```
fSumSquaresVector <- function(iN) {
  return(sum((1:iN)^2))
}

microbenchmark(fSumSquaresVector(10000), fSumSquaresUpToN(10000))
#> Unit: microseconds
#>                      expr   min     lq    mean median     uq    max neval
#>  fSumSquaresVector(10000)  22.9  36.45  67.793  57.45  60.10 1680.1   100
#>   fSumSquaresUpToN(10000) 222.9 225.20 256.910 226.10 228.35 2388.5   100
```

30.4 (4)

-

```
fRecursiveFactorial <- function(n) {
  if (n < 0) stop("Input must be non-negative")
  if (n == 0) return(1)
  else return(n * fRecursiveFactorial(n - 1))
}
fRecursiveFactorial(5)
#> [1] 120
```

30.5 (5)

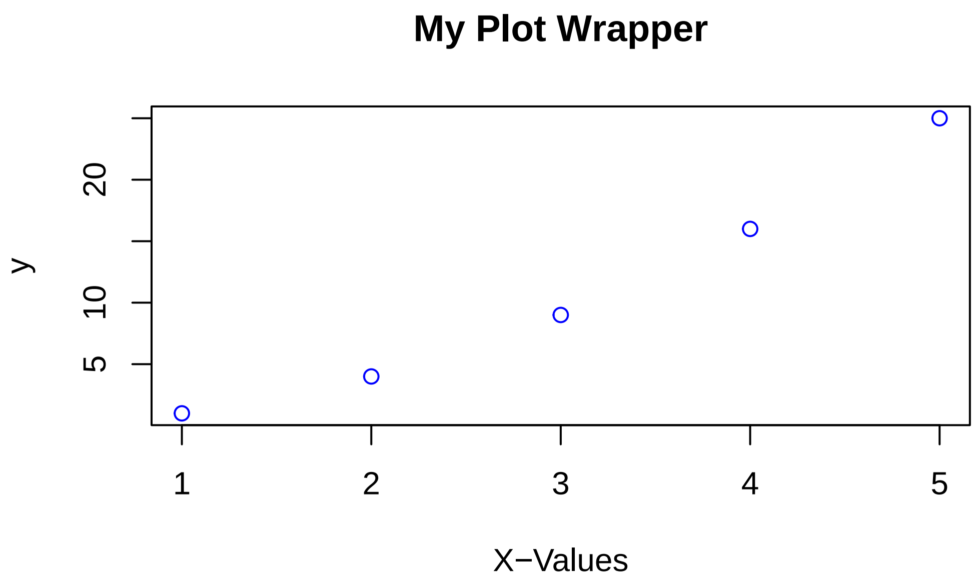
-
-
-

```
fApplyToMatrix <- function(mInput, FUN) {
  return(apply(mInput, 2, FUN))
}

mA <- matrix(rnorm(15), 5, 3)
fApplyToMatrix(mA, mean)
#> [1]  0.360807854 -0.005839041 -0.312895340
```

30.6 (6)

-

```
fCalculateMean <- function(vInput, ...) {
  return(mean(vInput, ...))
}
vX <- c(1, 2, NA, 5)
fCalculateMean(vX, na.rm = TRUE)
#> [1] 2.666667

myPlotWrapper <- function(x, y, ...) {
plot(x, y, main = "My Plot Wrapper", ...)
}
myPlotWrapper(1:5, (1:5)^2, xlab = "X-Values", col = "blue")
```



30.7 (7)

-
-
    -
    -
-

```r
vA <- 1:1000000
vB <- rnorm(1000000)

fLoopApproach <- function(vA, vB) {
  iProduct <- numeric(length(vA))
  for (i in 1:length(vA)) {
    iProduct[i] <- vA[i] * vB[i]
  }
}

fVectorApproach <- function(vA, vB) {
  return(vA * vB)
}

# microbenchmark(fLoopApproach(vA, vB), fVectorApproach(vA, vB))
```

30.8 (8)

-

```r
set.seed(123)
lData <- list(rnorm(10), c(NA, rnorm(5)), rnorm(15, mean=5))
```

-

```r
lDataClean <- lapply(lData, mean, na.rm = TRUE)
lDataClean
#> [[1]]
#> [1] 0.07462564
```

```
#>
#> [[2]]
#> [1] 0.3079017
#>
#> [[3]]
#> [1] 4.753408
```

30.9 (9)

-

-

-

-
-

```
fMatrixRedimension <- function(iN) {
  vResult <- c()
  for (i in 1:iN) {
    vResult <- c(vResult, i)
  }
}

fMatrixPreallocate <- function(iN) {
  vResult <- numeric(10000)
  for (i in 1:iN) {
    vResult[i] <- i
  }
}

# microbenchmark(fMatrixRedimension(10000), fMatrixPreallocate(10000))
```

30.10 (10)

-

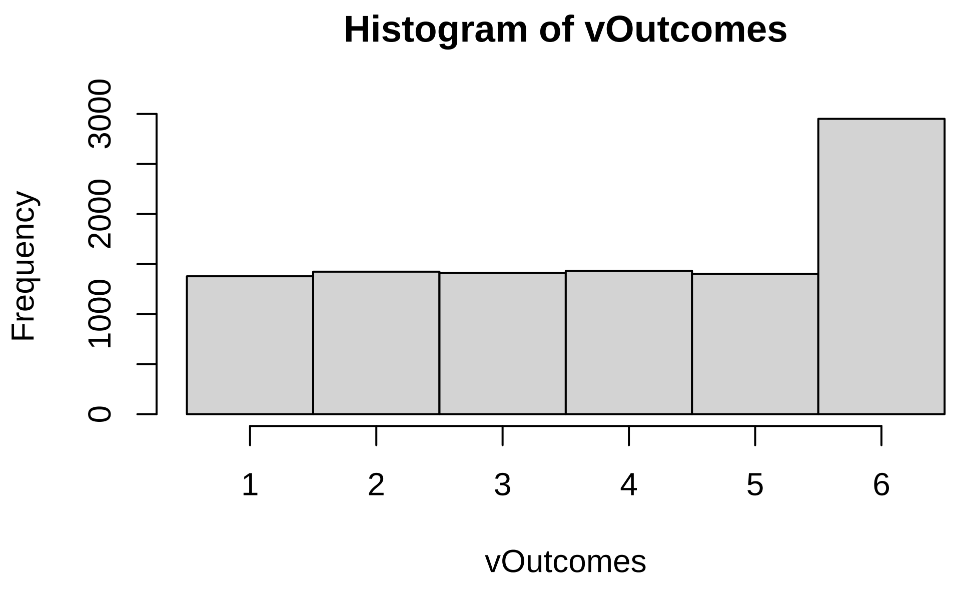
```r
x <- -2:2
ifelse(x > 0, "positive", "negative")
#> [1] "negative" "negative" "negative" "positive" "positive"
```

31 Exercise Set 3: Simulation

31.1 (1)

-
  -
  -
  -
  -

```r
fSimulateBiasedDie <- function(n_rolls) {
  probs <- c(rep(0.14, 5), 0.3)
  outcomes <- 1:6
  CDF <- cumsum(probs)

  vRolls <- numeric(n_rolls)

  for (i in 1:n_rolls) {
    u <- runif(1)
    vRolls[i] <- outcomes[min(which(u <= CDF))]
  }
  return(vRolls)
}

vOutcomes <- fSimulateBiasedDie(10000)

hist(vOutcomes, breaks = 0.5:6.5)
```

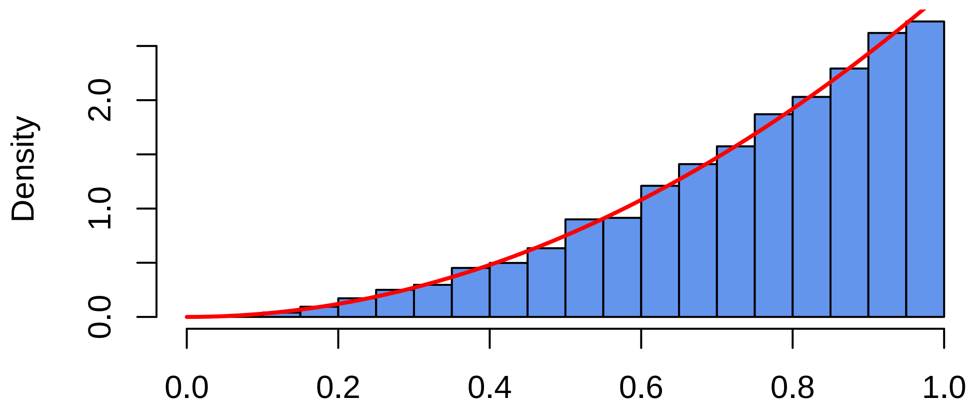


31.2 (2)

-
  -
  - a)
  - b)
  - c)
  - d)

```r
fSimulateCustom <- function(size) {
  U <- runif(size)
  return(U^(1/3))
}

set.seed(123)
vX <- fSimulateCustom(10000)
```

```r
hist(vX,
     freq = FALSE,
     breaks = 15,
     col = "cornflowerblue",
     xlab = "",
     ylab = "Density",
     main = "",
     xlim = c(0, 1))

curve(3 * x^2,
      from = 0,
      to = 1,
      col = "red",
      lwd = 2,
      add = TRUE)
```



31.3 (3)

-

–

–

– a)
– b)

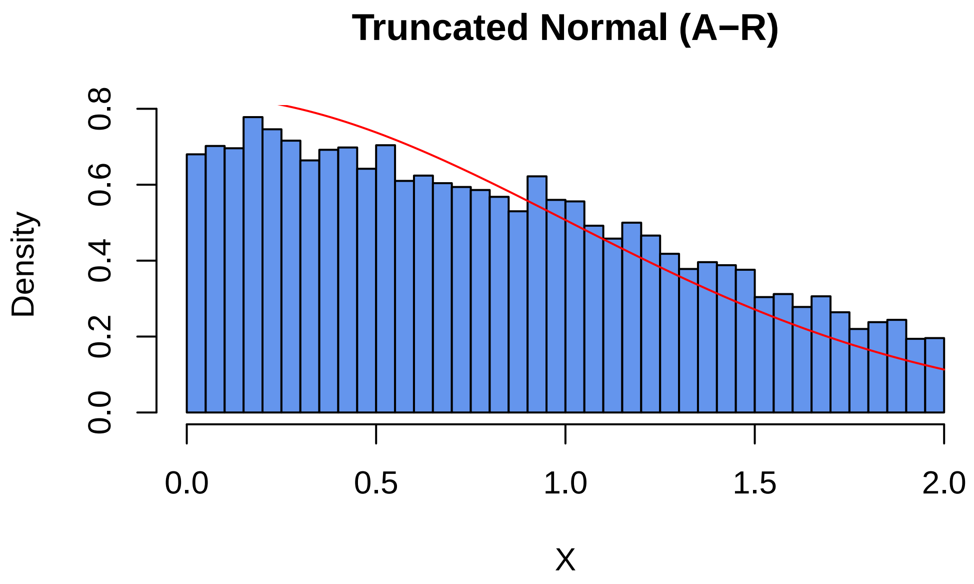
– c)

– d)

a)

```
fTruncNormSim <- function(size) {

  c <- (dnorm(0) / (pnorm(2) - pnorm(0))) / 0.5

  U <- rep(NA, size)
  Y <- rep(NA, size)
  X <- rep(NA, size)
  Unaccepted <- rep(TRUE, size)

  while (any(Unaccepted)) {

    UnacceptedCount <- sum(Unaccepted)

    U <- runif(UnacceptedCount, 0, 1)
    Y <- runif(UnacceptedCount, 0, 2)

    Accepted_ThisTime <- Unaccepted[Unaccepted] & (U <= ((dnorm(Y) / (pnorm(2) - pnorm(0))) /

    X[Unaccepted][Accepted_ThisTime] <- Y[Accepted_ThisTime]
    Unaccepted[Unaccepted] <- !Accepted_ThisTime

  }

  return(X)
```

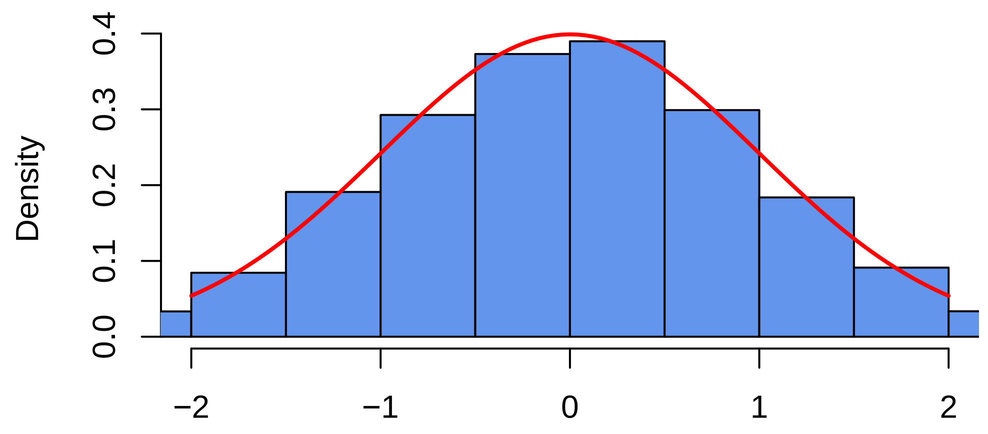
```r
}

set.seed(1)
X <- fTruncNormSim(10000)
hist(X, freq = FALSE, col = "cornflowerblue", xlim = c(0, 2), breaks = 50, main = "Truncated

curve(dnorm(x) / (pnorm(2) - pnorm(0)), add = TRUE, col = "red", from = 0, to = 2)
```



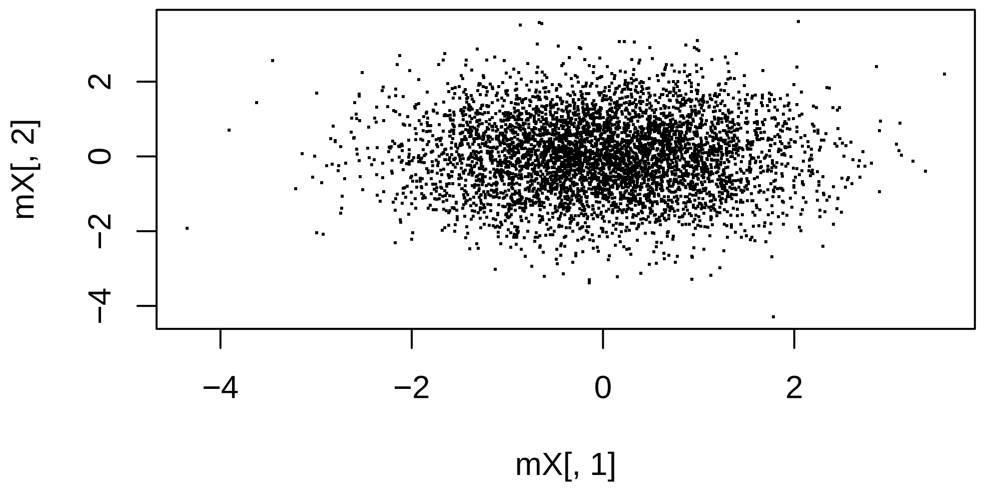
31.4 (4)

-
    - a)
    - b)
    - c)
    - d)

```
fMyBoxMuller <- function(size = 1) {
  U <- runif(size)
  V <- runif(size)
  X <- sqrt(-2*log(U)) * cos(2*pi*V)
  Y <- sqrt(-2*log(U)) * sin(2*pi*V)
  return(matrix(c(X,Y), ncol = 2))
}

mX <- fMyBoxMuller(5000)
vX <- as.numeric(mX)

hist(vX,
     freq = FALSE,
     breaks = 15,
     col = "cornflowerblue",
     xlab = "",
     ylab = "Density",
     main = "",
     xlim = c(-2, 2))

curve(dnorm(x),
      from = -2,
      to = 2,
      col = "red",
      lwd = 2,
      add = TRUE)
```



```
# Expected: circular pattern centered at (0,0)
plot(mX[, 1], mX[, 2], pch = ".")
```



31.5 (5)

-
    - a)
    - b)
    - c)

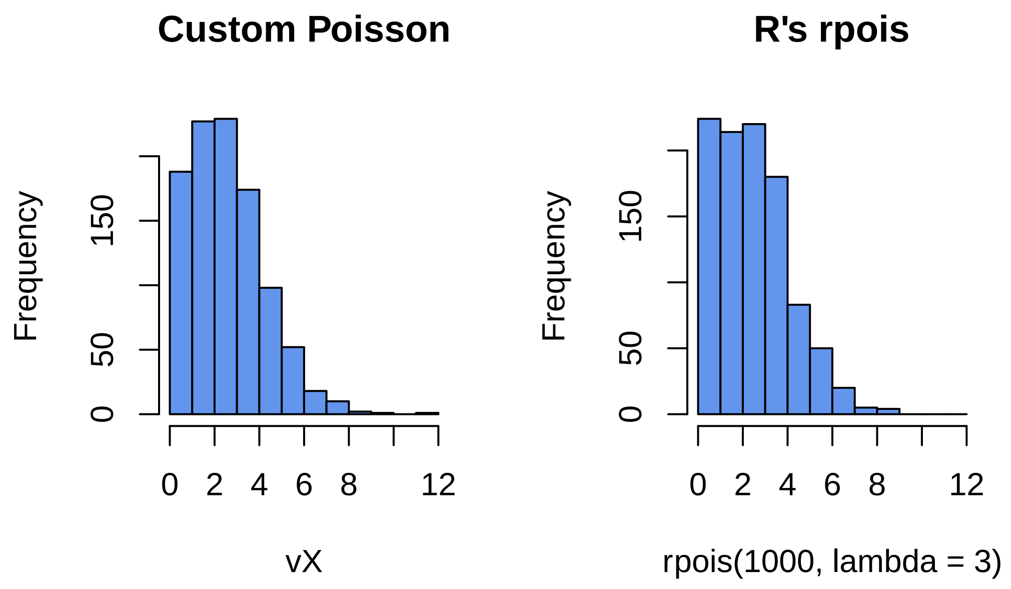
```
MonteCarlo.Integration <- function(f, n, a, b) {
  U <- runif(n, min = a, max = b)
  return( (b-a)*mean(f(U)) )
}
set.seed(10086)
MonteCarlo.Integration(function(x) exp(x^2), 10000, 0, 1)
#> [1] 1.45782

integrate(function(x) exp(x^2), lower = 0, upper = 1)
#> 1.462652 with absolute error < 1.6e-14

# Improve by increasing the number of samples.
```

31.6 (6)

-
    -
    -
    -
    -

```
fSimulatePoissonDiscrete <- function(F, size, ...) {
  m <- 0
  U <- runif(size)
  X <- rep(NA, size)
  X[F(0, ...) >= U] <- 0
```

```r
  while (any(F(m, ...) < U)) {
    m <- m + 1
    X[(F(m, ...) >= U) & (F(m - 1, ...) < U)] <- m
  }
  return(X)
}

fPoissonCDF <- function(size, lambda) {
  dSum <- 0
  for (i in 0:size) {
    dSum <- dSum + (exp(-lambda) * lambda^i) / factorial(i)
  }
  return(dSum)
}

set.seed(10086)
vX <- fSimulatePoissonDiscrete(fPoissonCDF, size = 1000, lambda = 3)

par(mfrow=c(1,2))
hist(vX,
     breaks = 0:max(vX),
     col = "cornflowerblue",
     main = "Custom Poisson")
hist(rpois(1000, lambda = 3),
     breaks = 0:max(vX),
     col = "cornflowerblue",
     main = "R's rpois")
```



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

31.7 (7)

-
  - a)
  - b)
  - c)
  - d)
  - e)

```
vRand1 <- rnorm(5)
vRand2 <- rnorm(5)
vRand1 - vRand2
#> [1]  0.02333262 -0.90672903 -2.01327089  1.66283561  0.14078765
```

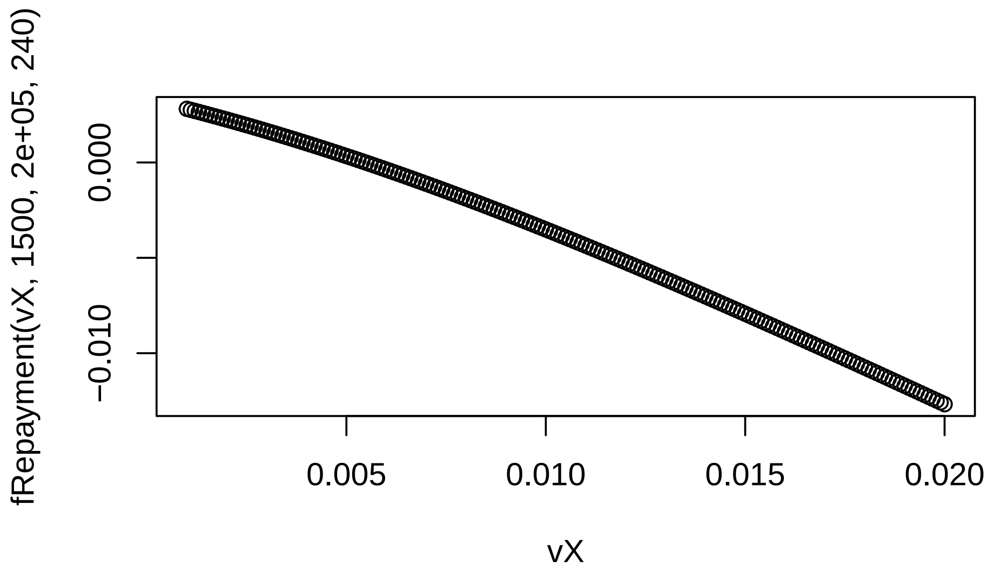
```r
set.seed(12345)
vRand1 <- rnorm(5)
set.seed(12345)
RNG.state <- .Random.seed
vRand2 <- rnorm(5)
vRand1 - vRand2
#> [1] 0 0 0 0 0
.Random.seed <- RNG.state
vRand3 <- rnorm(5)
vRand1 - vRand3
#> [1] 0 0 0 0 0
```

32 Exercise Set 4: Root-Finding

32.1 (1)

-
    -
    -
    - a)
    - b)
    - c)
    - d)

```r
fRepayment <- function(r, A, P, N) {
  return(A/P - (r * (1+r)^N) / ((1+r)^N - 1))
}
vX <- seq(0.001, 0.02, by = 0.0001)
plot(vX, fRepayment(vX, 1500, 200000, 240))
```



```r
bisection <- function(f, dX.l, dX.r, dTol = 10e-7, max.iter = 1000, ...) {

  #check inputs
  if (dX.l >= dX.r) {
    cat("error: x.l >= x.r \n")
    return(NULL)
  }
  f.l <- f(dX.l, ...)
  f.r <- f(dX.r, ...)
  if (f.l == 0) {
    return(dX.l)
  } else if (f.r == 0) {
    return(dX.r)
  } else if (f.l*f.r > 0) {
    cat("error: f(x.l)*f(x.r) > 0 \n")
    return(NULL)
  }

  # successively refine x.l and x.r
  iter <- 0
  while ((dX.r - dX.l) > dTol && (iter < max.iter)) {
    dX.m <- (dX.l + dX.r)/2
    f.m <- f(dX.m, ...)
    if (f.m == 0) {
      return(dX.m)
    } else if (f.l*f.m < 0) {
      dX.r <- dX.m
      f.r <- f.m
    } else {
      dX.l <- dX.m
      f.l <- f.m
    }
    iter <- iter + 1
  }
  cat("at iteration", iter, "the root lies between", dX.l, "and", dX.r, "\n")
  # return approximate root
  return((dX.l + dX.r)/2)
}

bisection(fRepayment, dX.l = 0.001 , dX.r = 0.01, dTol = 10e-7, A = 1500, P = 200000, N = 24
#> at iteration 14 the root lies between 0.005479126 and 0.005479675
#> [1] 0.005479401
```

32.2 (2)

-
  -
  - a)
  - b)
  - c)
  - d)

```
f <- function(x) {
  dOut = x^3 - 2 * x - 5
  return(dOut)
}
```

```r
f_prime <- function(x) {
  dOut = 3 * x^2 - 2
  return(dOut)
}

NR <- function(f, f_prime, dX0, dTol = 1e-9, max.iter = 1000, ...) {
  dX <- dX0
  fx <- f(dX, ...)
  iter <- 0
  while ((abs(fx) > dTol) && (iter < max.iter)) {
    dX <- dX - f(dX, ...)/f_prime(dX, ...)
    fx <- f(dX, ...)
    iter <- iter + 1
  }
  if (abs(fx) > dTol) {
    cat("Algorithm failed to converge\n")
    return(NULL)
  } else {
    cat("Algorithm converged\n")
    cat("At iteration ", iter, "value of x is: ", dX, "\n")
    return(dX)
  }
}

NR(f, f_prime, dTol = 10e-6, dX0 = 2, max.iter = 100)
#> Algorithm converged
#> At iteration  3 value of x is:  2.094551
#> [1] 2.094551
NR(f, f_prime, dTol = 10e-6, dX0 = 0, max.iter = 100) # Slower to converge. Could find a dif
#> Algorithm converged
#> At iteration  18 value of x is:  2.094551
#> [1] 2.094551
```

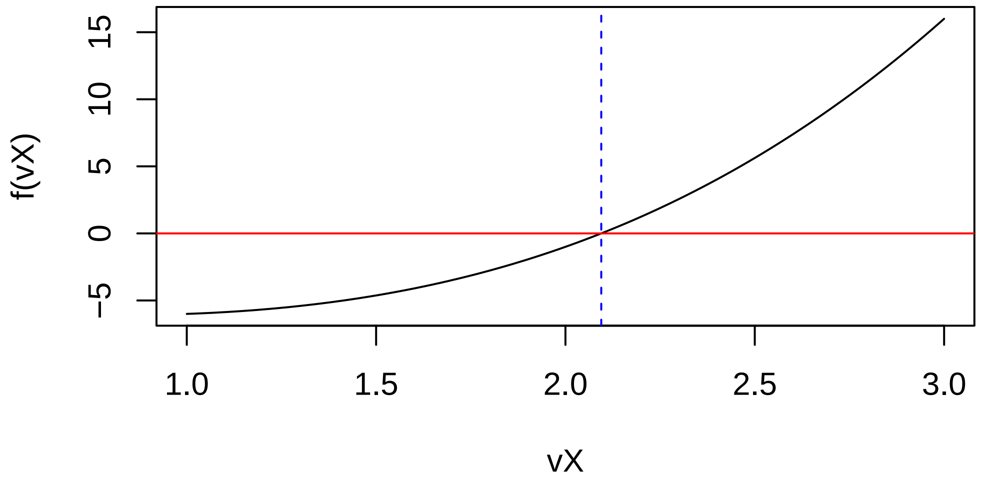
32.3 (3)

-
    -
    - a)

- b)
- c)

```
f <- function(x) {
  dOut = x^3 - 2 * x - 5
  return(dOut)
}
fSecant <- function(f, dX0, dX1, dTol = 1e-9, max.iter = 1000, ...) {
  iter <- 0
  dX2 <- dX1
  while ((abs(f(dX2, ...)) > dTol) && (iter < max.iter)) {
    dX2 <- dX1 - f(dX1, ...) * ((dX0 - dX1) / (f(dX0, ...) - f(dX1, ...)))
    dX0 <- dX1
    dX1 <- dX2
    iter <- iter + 1
  }
  if (abs(f(dX2, ...)) > dTol) {
    cat("At iteration ", iter, "value of x is: ", dX1, "\n")
    return(list(root = NULL, f.root = NULL, iter = iter, "Algorithm failed to converge. Maxi
  } else {
    cat("At iteration ", iter, "value of x is: ", dX1, "\n")
    return(list(root = dX2, f.root = f(dX2), iterations = iter, "Convergence reached."))
  }
}

root <- fSecant(f, dX0 = 2, dX1 = 2.5, dTol = 10e-6, max.iter = 100)
#> At iteration  4 value of x is:  2.094551
root # 1 more step than Newton-Raphson
#> $root
#> [1] 2.094551
#>
#> $f.root
#> [1] -9.612876e-07
#>
#> $iterations
#> [1] 4
#>
#> [[4]]
#> [1] "Convergence reached."

uniroot(f, interval = c(-3,3))
```

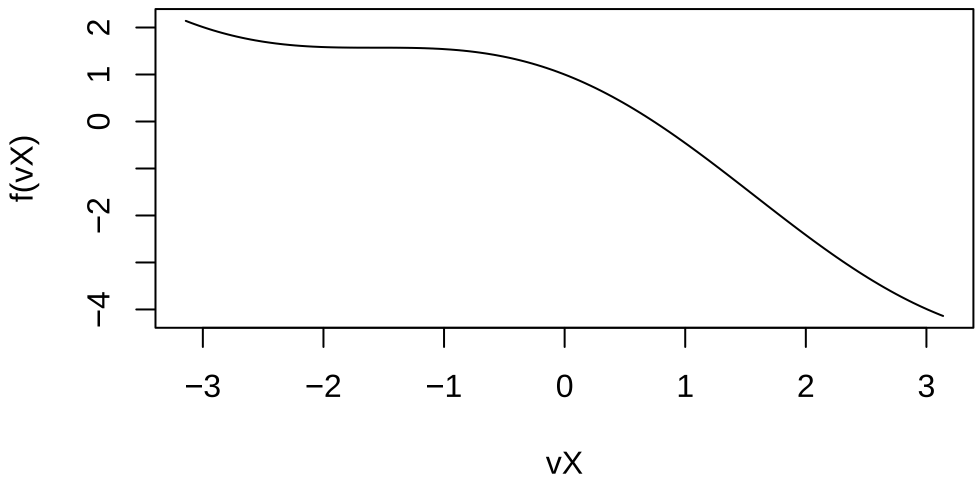
```
#> $root
#> [1] 2.094555
#>
#> $f.root
#> [1] 3.690185e-05
#>
#> $iter
#> [1] 7
#>
#> $init.it
#> [1] NA
#>
#> $estim.prec
#> [1] 6.103516e-05

vX <- seq(1, 3, 0.01)
plot(vX, f(vX), type = "l")
abline(h = 0, col = "red")
abline(v = root["root"], col = "blue", lty = 2)
```



32.4 (4)

-
  -
  - a)
  - b)
  - c)

```r
f <- function(x) {
  return(cos(x) - x)
}

vX <- seq(-pi, pi, 0.01)
plot(vX, f(vX), type = "l")
```



```r
uniroot(f, interval = c(-1,2))
#> $root
#> [1] 0.7390929
#>
```

```
#> $f.root
#> [1] -1.292031e-05
#>
#> $iter
#> [1] 6
#>
#> $init.it
#> [1] NA
#>
#> $estim.prec
#> [1] 6.103516e-05

# See exercise just before this one
root <- fSecant(f, dX0 = -1, dX1 = 2, dTol = 10e-6, max.iter = 100)
#> At iteration  5 value of x is:  0.7390831
root
#> $root
#> [1] 0.7390831
#>
#> $f.root
#> [1] 3.363451e-06
#>
#> $iterations
#> [1] 5
#>
#> [[4]]
#> [1] "Convergence reached."
```

32.5 (5)

-
    -
    - a)
    - b)

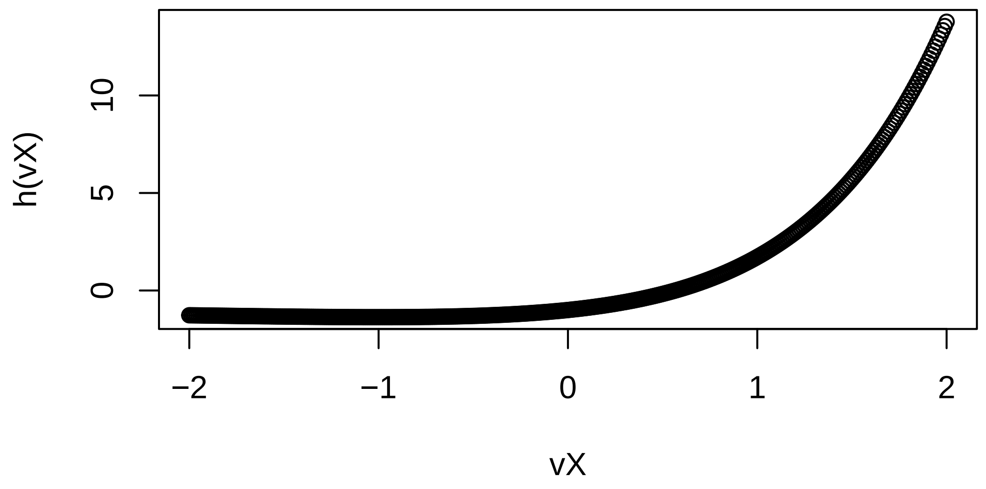
```
g <- function(x) {
  return(exp(sin(x)) * log(x^2 + 1))
}
```

```r
suppressMessages(library(numDeriv))
grad(g, 1)
#> [1] 3.188554
grad(g, 2)
#> [1] 0.3233247
grad(g, 3)
#> [1] -1.934098

# Analytically
gGrad <- function(x) {
  return((exp(sin(x))*cos(x)*log(x^2+1) + exp(sin(x))*(2*x/(x^2+1))))
}
gGrad(2)
#> [1] 0.3233247
```

32.6 (6)

-
    -
    - a)
    - b)
    - c)
    - d)
    - e)
    - f)

```r
h <- function(x) {
  return(x * exp(x) - 1)
}
vX <- seq(-2, 2, 0.01)
plot(vX, h(vX))
```



```
bisection(h, 0, 2)
#> at iteration 21 the root lies between 0.5671425 and 0.5671434
#> [1] 0.567143

hGrad <- function(x) {
  return(x * exp(x) + exp(x))
}

NR(h, hGrad, 1.5)
#> Algorithm converged
#> At iteration  6 value of x is:  0.5671433
#> [1] 0.5671433
fSecant(h, 0, 2)
#> At iteration  8 value of x is:  0.5671433
#> $root
#> [1] 0.5671433
#>
#> $f.root
#> [1] -3.581359e-10
#>
#> $iterations
#> [1] 8
#>
```

```
#> [[4]]
#> [1] "Convergence reached."
uniroot(h, c(0, 2))
#> $root
#> [1] 0.5671321
#>
#> $f.root
#> [1] -3.093327e-05
#>
#> $iter
#> [1] 8
#>
#> $init.it
#> [1] NA
#>
#> $estim.prec
#> [1] 6.103516e-05
```

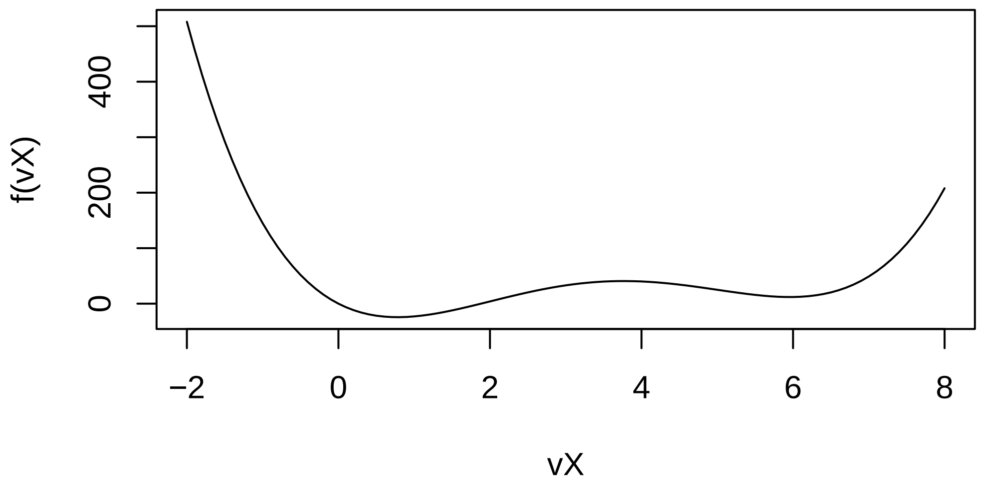
33 Exercise Set 5: Numerical Optimization (Univariate and Multivariate)

33.1 (1)

-
    -
    - a)
    - b)

    - c)
    - d)
    - e)

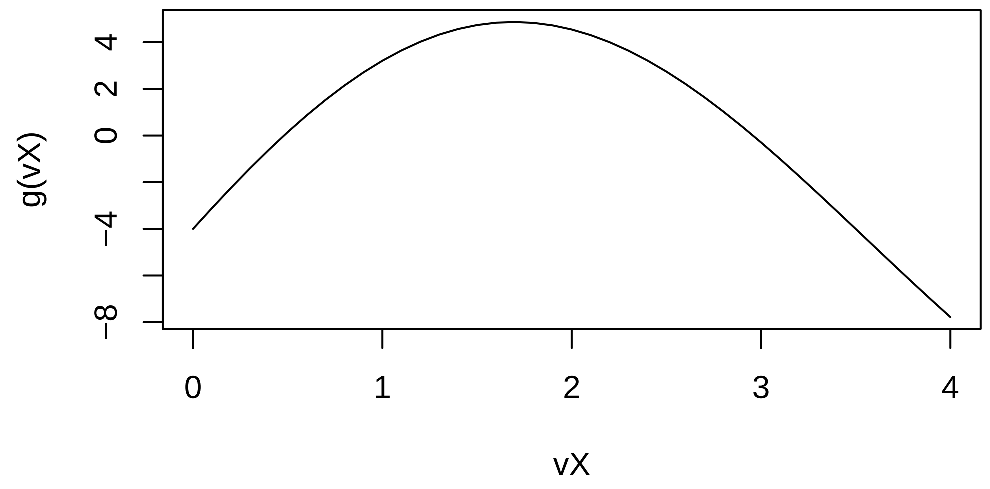
```r
f <- function(x) {
  return(x^4 - 14 * x^3 + 60 * x^2 - 70 * x)
}

fprime <- function(x) {
  return(4 * x^3 - 42 * x^2 + 120 * x - 70)
}

fsecond <- function(x) {
  return(12 * x^2 - 84 * x + 120)
}

NM <- function(f, f_prime, f_sec, dX0, dTol = 1e-9, n.max = 1000){
  dX <- dX0
  fx <- f(dX)
  fpx <- f_prime(dX)
  fsx <- f_sec(dX)
  n <- 0
  while ((abs(fpx) > dTol) && (n < n.max)) {
    dX <- dX - fpx/fsx
    fx <- f(dX)
    fpx <- f_prime(dX)
    fsx <- f_sec(dX)
    n <- n + 1
  }
  if (n == n.max) {
    cat('newton failed to converge\n')
  } else {
    cat("At iteration", n, "the value of x is:", dX, "\n")
    return(dX)
  }
}
x1 <- NM(f, fprime, fsecond, dX0 = 0)
#> At iteration 5 the value of x is: 0.7808841
x2 <- NM(f, fprime, fsecond, dX0 = 3)
#> At iteration 4 the value of x is: 3.761921
x3 <- NM(f, fprime, fsecond, dX0 = 6)
#> At iteration 3 the value of x is: 5.957195
fsecond(x1) # local minimum
#> [1] 61.7231
fsecond(x2) # local maximum
#> [1] -26.17677
fsecond(x3) # local minimum
#> [1] 45.45367
vX <- seq(-2, 8, 0.1)
plot(vX, f(vX), type = "l")
```



33.2 (2)

-
  -
  - a)
  - b)
  - c)

```
g <- function(x) {
  return(-(x - 2)^2 + 5 * sin(x))
}

vX <- seq(0, 4, 0.1)
plot(vX, g(vX), type = "l")
```



```
gsection <- function(f, dX.l, dX.r, dX.m, dTol = 1e-9) {

  # golden ratio plus one
  dGR1 <- 1 + (1 + sqrt(5))/2

  # successively refine x.l, x.r, and x.m
  f.l <- f(dX.l)
  f.r <- f(dX.r)
  f.m <- f(dX.m)
  while ((dX.r - dX.l) > dTol) {
    if ((dX.r - dX.m) > (dX.m - dX.l)) { # if the right segment is wider than the left
      dY <- dX.m + (dX.r - dX.m)/dGR1 # put Y into the right segment according to the golden
      f.y <- f(dY)
      if (f.y >= f.m) {
        dX.l <- dX.m
        f.l <- f.m
        dX.m <- dY
        f.m <- f.y
      } else {
        dX.r <- dY
        f.r <- f.y
      }
    } else { #if the left segment is wider than the right
      dY <- dX.m - (dX.m - dX.l)/dGR1 # put Y into the left segment according to the golden
      f.y <- f(dY)
      if (f.y >= f.m) {
        dX.r <- dX.m
        f.r <- f.m
        dX.m <- dY
        f.m <- f.y
      } else {
        dX.l <- dY
        f.l <- f.y
      }
    }
  }
  return(dX.m)
}
gsection(g, dX.l = 0, dX.r = 4, dX.m = 2, dTol = 10e-5)
#> [1] 1.693623
optimize(g, interval = c(0, 4), maximum = TRUE)
#> $maximum
#> [1] 1.693664
#>
#> $objective
#> [1] 4.868465
```

33.3 (3)

-
  -
  - a)
  - b)
  - c)

```
h <- function(vX) {
  return(-(vX[1] - 1)^2 - 2 * (vX[2] - 2)^2 + vX[1] * vX[2])
}
```

```r
hgrad <- function(vX) {
  h1 <- -2 * (vX[1] - 1) + vX[2]
  h2 <- -4 * (vX[2] - 2) + vX[1]
  return(c(h1, h2))
}

# golden section algorithm from last week
gsection <- function(f, dX.l, dX.r, dX.m, dTol = 1e-9) {

  # golden ratio plus one
  dGR1 <- 1 + (1 + sqrt(5))/2

  # successively refine x.l, x.r, and x.m
  f.l <- f(dX.l)
  f.r <- f(dX.r)
  f.m <- f(dX.m)
  while ((dX.r - dX.l) > dTol) {
    if ((dX.r - dX.m) > (dX.m - dX.l)) { # if the right segment is wider than the left
      dY <- dX.m + (dX.r - dX.m)/dGR1 # put Y into the right segment according to the golden
      f.y <- f(dY)
      if (f.y >= f.m) {
        dX.l <- dX.m
        f.l <- f.m
        dX.m <- dY
        f.m <- f.y
      } else {
        dX.r <- dY
        f.r <- f.y
      }
    } else { #if the left segment is wider than the right
      dY <- dX.m - (dX.m - dX.l)/dGR1 # put Y into the left segment according to the golden r
      f.y <- f(dY)
      if (f.y >= f.m) {
        dX.r <- dX.m
        f.r <- f.m
        dX.m <- dY
        f.m <- f.y
      } else {
        dX.l <- dY
        f.l <- f.y
      }
    }
  }
  return(dX.m)
}

line.search <- function(f, vX, vG, dTol = 1e-9, dA.max = 2^5) {
  # f is a real function that takes a vector of length d
  # x and y are vectors of length d
  # line.search uses gsection to find a >= 0 such that
  # g(a) = f(x + a*y) has a local maximum at a,
  # within a tolerance of tol
  # if no local max is found then we use 0 or a.max for a
  # the value returned is x + a*y
  if (sum(abs(vG)) == 0){
    return(vX) # +0*vG
  } # g(a) constant
  g <- function(dA){
    return(f(vX + dA*vG))
  }
  # find a triple a.l < a.m < a.r such that
  # g(a.l) <= g(a.m) and g(a.m) >= g(a.r)

  # choose a.l
  dA.l <- 0
  g.l <- g(dA.l)
  # find a.m
  dA.m <- 1
  g.m <- g(dA.m)
  while ((g.m < g.l) & (dA.m > dTol)) {
    dA.m <- dA.m/2
    g.m <- g(dA.m)
  }
  # if a suitable a.m was not found then use 0 for a, so just return vX as the next step
  if ((dA.m <= dTol) & (g.m < g.l)){
    return(vX)
  }
  # find a.r
  dA.r <- 2*dA.m
  g.r <- g(dA.r)
  while ((g.m < g.r) & (dA.r < dA.max)) {
    dA.m <- dA.r
    g.m <- g.r
    dA.r <- 2*dA.m
    g.r <- g(dA.r)
  }
  # if a suitable a.r was not found then use a.max for a
  if ((dA.r >= dA.max) & (g.m < g.r)){
    return(vX + dA.max*vG)
  }
  # apply golden-section algorithm to g to find a
  dA <- gsection(g, dA.l, dA.r, dA.m)
  return(vX + dA*vG)
}

ascent <- function(f, grad.f, vX0, dTol = 1e-9, n.max = 100) {
  vX.old <- vX0
  vX <- line.search(f, vX0, grad.f(vX0))
  n <- 1
  while ((f(vX) - f(vX.old) > dTol) & (n < n.max)) {
    vX.old <- vX
    vX <- line.search(f, vX, grad.f(vX))
    n <- n + 1
  }
  cat("at iteration", n, "the coordinates of x are", vX, "\n")
  return(vX)
}

ascent(h, hgrad, vX0 = c(0, 0))
#> at iteration 16 the coordinates of x are 2.285706 2.57142
#> [1] 2.285706 2.571420
```

33.4 (4)

-
  -
  - a)
  - b)
  - c)

  - d)

```
hcomb <- function(vX) {
  h <- -(vX[1] - 1)^2 - 2 * (vX[2] - 2)^2 + vX[1] * vX[2]
  h1 <- -2 * (vX[1] - 1) + vX[2]
  h2 <- -4 * (vX[2] - 2) + vX[1]
  h11 <- -2
  h12 <- 1
  h22 <- -4
  return(list(h, c(h1, h2), matrix(c(h11, h12, h12, h22), 2, 2)))
}

newton <- function(f3, vX0, dTol = 1e-9, n.max = 100) {
  # Newton's method for optimisation, starting at x0
  # f3 is a function that given x returns the list
  # {f(x), grad f(x), Hessian f(x)}, for some f
  vX <- vX0
  f3.x <- f3(vX)
  n <- 0
  while ((max(abs(f3.x[[2]])) > dTol) & (n < n.max)) {
    vX <- vX - solve(f3.x[[3]], f3.x[[2]])
    #vX <- vX - solve(f3.x[[3]])%*%f3.x[[2]]
    f3.x <- f3(vX)
    cat("At iteration", n, "the coordinates of x are", vX, "\n")
    n <- n + 1
  }
  if (n == n.max) {
    cat('newton failed to converge\n')
  } else {
    return(vX)
  }
}

n1 <- newton(hcomb, vX0 = c(0, 0))
#> At iteration 0 the coordinates of x are 2.285714 2.571429
n1
#> [1] 2.285714 2.571429
eigen(hcomb(n1)[[3]]) ## maximum
#> eigen() decomposition
#> $values
#> [1] -1.585786 -4.414214
#>
#> $vectors
#>            [,1]       [,2]
#> [1,] -0.9238795 -0.3826834
#> [2,] -0.3826834  0.9238795
```

33.5 (5)

-
  -
  - a)
  - b)
    - *
    - *
    - *
  -
  - c)

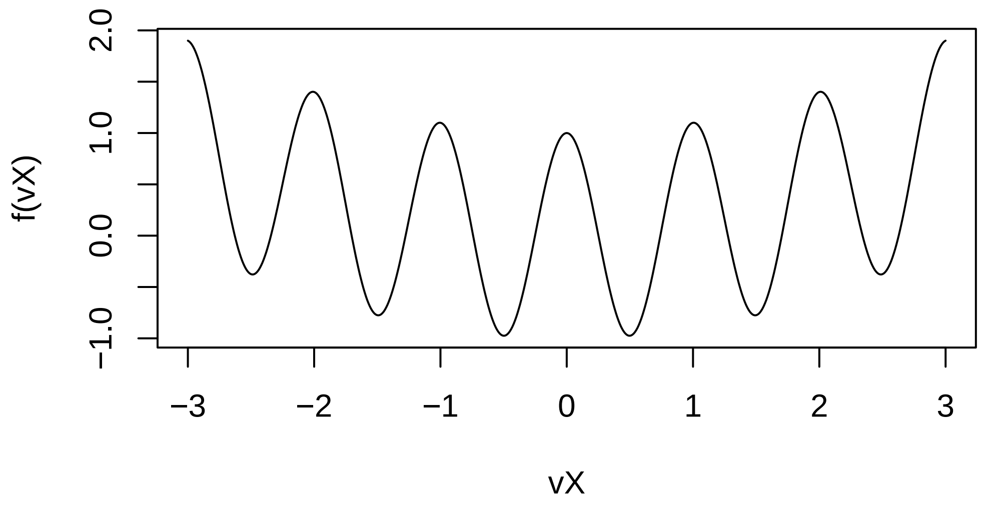
```r
suppressMessages(library(numDeriv))
fRosenbrock <- function(vX) {
  a <- 1
  b <- 100
  return((a-vX[1])^2 + b * (vX[2]-vX[1]^2)^2)
}

optim(c(-1.2, -1), fRosenbrock, method = "Nelder-Mead")$par
#> [1] 1.003505 1.006996
optim(c(-1.2, -1), fRosenbrock, method = "BFGS")$par
#> [1] 0.9998000 0.9996001
optim(c(-1.2, -1), fRosenbrock, method = "L-BFGS-B")$par
#> [1] 0.9998032 0.9996078
optim(c(-1.2, -1), fRosenbrock, method = "L-BFGS-B", lower=c(-2,-1), upper=c(2,3))$par
#> [1] 0.9998006 0.9996012
grad_rosen <- function(par) numDeriv::grad(fRosenbrock, par)
optim(c(-1.2, -1), fRosenbrock, gr = grad_rosen, method = "BFGS")$par
#> [1] 1 1
```

33.6 (6)

-
  -

- a)
- b)
- c)

```
f <- function(x) {
  return(cos(2 * pi * x) + 0.1 * x^2)
}

vX <- seq(-3, 3, 0.01)
plot(vX, f(vX), type = "l")
```



```
optimize(f, interval = c(-3, 3))
#> $minimum
#> [1] 1.492439
#>
#> $objective
#> [1] -0.7761343
optimize(f, interval = c(-0.5, 0.5))
#> $minimum
#> [1] -0.4974641
```

```
#>
#> $objective
#> [1] -0.975126
optimize(f, interval = c(0.5, 1.5))
#> $minimum
#> [1] 0.5000661
#>
#> $objective
#> [1] -0.9749933
```

33.7 (7)

-
  -
  - a)

  - b)
  - c)
  - d)

```
f <- function(p) {
  return(p^2 * (1 - p)^3)
}

g <- function(p_tilde) {
  p <- exp(p_tilde) / (1 + exp(p_tilde))
  return(f(p))
}

optim_res <- optim(0, g, method = "BFGS", , control=list(fnscale=-1))
p_tilde <- optim_res$par
p_optim <- exp(p_tilde) / (1 + exp(p_tilde))
p_optim
#> [1] 0.4
f(p_optim)
#> [1] 0.03456
```

33.8 (8)

-
  -
  - a)
  - b)
  - c)

```
neg_log <- function(p, y) {
  dSum1 <- 0
  dSum2 <- 0
  for (i in 1:length(y)) {
    dSum1 <- dSum1 - y[i] * log(p)
    dSum2 <- dSum2 - (1 - y[i]) * log(1 - p)
  }
  return(dSum1 + dSum2)
}

neg_log_lik_bern <- function(p, data) {
if (p <= 0 || p >= 1) return(Inf) # Bounds for log
-sum(data * log(p) + (1 - data) * log(1 - p))
}

vY <- c(1, 0, 1, 1, 0, 1, 0, 0, 1, 1)
optim(0.5, neg_log, y = vY, method = "L-BFGS-B", lower = 0.001, upper = 0.999)$par
#> [1] 0.5999997
optim(0.5, neg_log_lik_bern, data = vY, method = "L-BFGS-B", lower = 0.001, upper = 0.999)$pa
#> [1] 0.5999997
# Analytical MLE
mean(vY)
#> [1] 0.6
```

34 Exercise Set 6: C++ with Rcpp and RcppArmadillo

34.1 (1)

-
  -
  -
  -

```
suppressMessages(library(Rcpp))
#> Warning: pakke 'Rcpp' blev bygget under R version 4.3.3

cppFunction('
double sum_greater_than_C(NumericVector x, double threshold) {
  double total = 0;
  for (int i = 0; i < x.size(); i++) {
    if (x[i] > threshold) {
      total += x[i];
    }
  }
  return total;
}')

vX <- rnorm(1000)
threshold <- 0
sum_greater_than_C(vX, threshold)
#> [1] 420.1322
sum(vX[vX>0])
#> [1] 420.1322
```

34.2 (2)

-
  - a)
  - b)
  - c)

```
suppressMessages(library(Rcpp))
suppressMessages(library(microbenchmark))

cppFunction('
int fib_cpp(int n) {
  if (n <= 1) return(0);
  return fib_cpp(n - 1) + fib_cpp(n - 2);
}')

cppFunction('
int fib_iter_cpp(int n) {
  if (n <= 1) return(0);
  int a = 0, b = 1, temp;
  for (int i = 2; i <= n; i++) {
    temp = a + b;
    a = b;
    b = temp;
  }
  return b;
}')

fib_r <- function(n) {
  if (n <= 1) {
    return(0)
  } else {
    return(fib_r(n - 1) + fib_r(n - 2))
  }
}

microbenchmark(fib_cpp(25), fib_iter_cpp(25), fib_r(25))
#> Unit: nanoseconds
#>                 expr       min        lq      mean    median         uq        max neval
#>        fib_cpp(25)    149500    150850    179533    157950    167350    948000   100
#>   fib_iter_cpp(25)       700      1000     14684      2750     10850    937400   100
#>          fib_r(25) 86243500 88826400 93384413 89811800 92831750 131750000   100
```

34.3 (3)

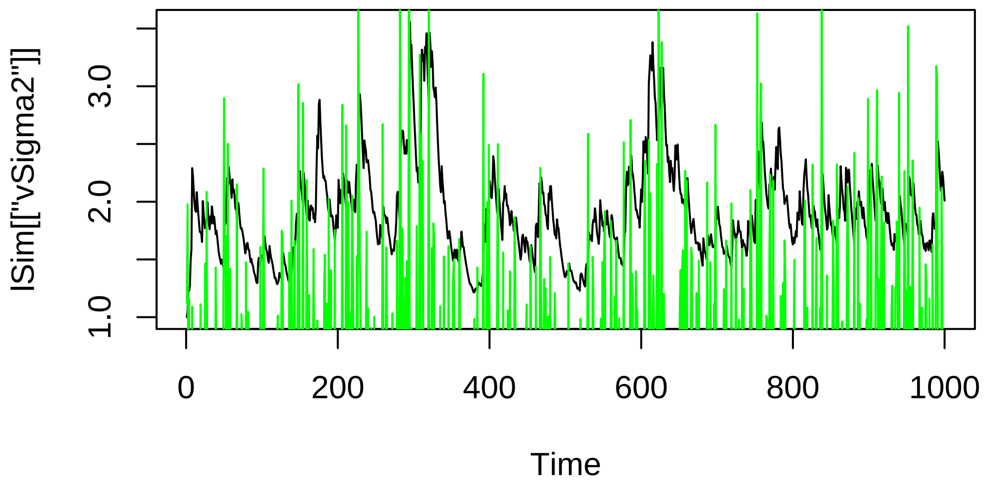
-
    - a)
    -
    -
        *
        *
        *
    - b)

```
suppressMessages(library(Rcpp))
suppressMessages(library(RcppArmadillo))
#> Warning: pakke 'RcppArmadillo' blev bygget under R version 4.3.3

sourceCpp("extra_cpp.cpp")
#>
#> > vX <- 1:10
#>
#> > dS <- 5
#>
#> > res <- cpp_vector_scalar_mult(vX, dS)
#>
#> > print(res)
#>       [,1]
#>  [1,]    5
#>  [2,]   10
#>  [3,]   15
#>  [4,]   20
#>  [5,]   25
#>  [6,]   30
#>  [7,]   35
#>  [8,]   40
#>  [9,]   45
#> [10,]   50

mA <- matrix(1:9, 3, 3)
mB <- matrix(9:1, 3, 3)
arma_matrix_ops(mA, mB)
#> Matrix mA is singular.
#> $mProd
#>      [,1] [,2] [,3]
#> [1,]   90   54   18
#> [2,]  114   69   24
#> [3,]  138   84   30
#>
#> $vEigenvalsA
#>                  [,1]
#> [1,]  1.611684e+01+0i
#> [2,] -1.116844e+00+0i
#> [3,] -5.700691e-16+0i
mA %*% mB
#>      [,1] [,2] [,3]
#> [1,]   90   54   18
#> [2,]  114   69   24
#> [3,]  138   84   30
```

34.4 (4)

-
    -
    -
    -
    -
    -

```
suppressMessages(library(Rcpp))
suppressMessages(library(RcppArmadillo))

sourceCpp("extra_cpp.cpp")
#>
#> > vX <- 1:10
#>
#> > dS <- 5
#>
#> > res <- cpp_vector_scalar_mult(vX, dS)
#>
#> > print(res)
#>       [,1]
#>  [1,]    5
#>  [2,]   10
#>  [3,]   15
#>  [4,]   20
#>  [5,]   25
#>  [6,]   30
#>  [7,]   35
#>  [8,]   40
#>  [9,]   45
#> [10,]   50

iT <- 1000
dOmega <- 0.1
dAlpha <- 0.05
dBeta <- 0.9
dSigmaInit <- 1.0

lSim <- simulate_garch_cpp(iT, dOmega, dAlpha, dBeta, dSigmaInit)

plot(1:1000, lSim[["vSigma2"]], type = "l", lty = 1, xlab = "Time")
lines(1:1000, lSim[["vR"]],col="green")
```



34.5 (5)

-
  - a)
  - b)
  -
  - c)

```
my_r_summary <- function(x) list(mean=mean(x), sd=sd(x))

cppFunction('
SEXP call_r_from_cpp(NumericVector x, Function r_func) {
  return r_func(x);
}')

call_r_from_cpp(rnorm(100), my_r_summary)
#> $mean
#> [1] -0.10438
#>
```

```
#> $sd
#> [1] 1.056556
```

34.6 (6)

-
  -
  -

  -
    *
    *
    *
  -

```
// [[Rcpp::depends(RcppArmadillo)]]
#include <RcppArmadillo.h>
using namespace Rcpp;
using namespace arma;

//[[Rcpp::export]]
vec cpp_vector_scalar_mult(vec v, double s) {
  return v * s;
}

/*** R
vX <- 1:10
dS <- 5
res <- cpp_vector_scalar_mult(vX, dS)
print(res)
*/
```

35 Exercise Set 7: Creating R Packages

35.1 (1)

-
  -
  - a)
  - b)
    -
    -
    -
  - c)
  - d)
  - e)

```r
MyUtils::fAddTwoNumbers(5, 2)
#> [1] 7
```

35.2 (2)

-
  -
  - a)
  - b)

- c)
  -
  -
- d)
- e)
- f)

35.3 (3)

-
  -
  - a)
  - b)
  - c)
    -
    -
  - d)
  - e)
  - f)

35.4 (4)

-
  - a)
  - b)
  - c)

35.5 (5)

-
  - a)
  - b)
  - c)

36 Exercise Set 8: Parallel Computing

36.1 (1)

-
  - a)
  - b)
  - c)
  - d)
  - e)
  - f)

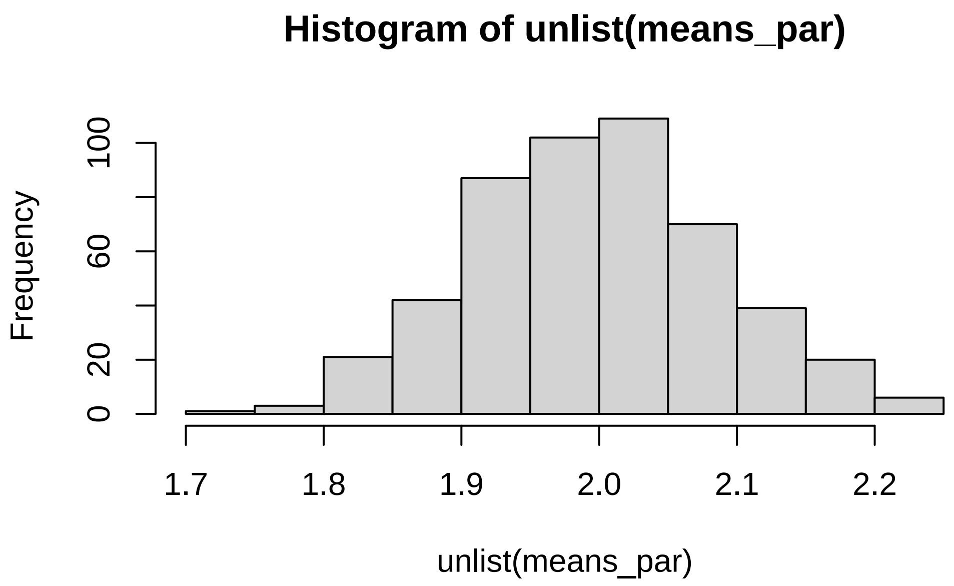
```r
listV <- list(
  v1 = rnorm(1000000),
  v2 = rnorm(1000000),
  v3 = rnorm(1000000),
  v4 = rnorm(1000000),
  v5 = rnorm(1000000),
  v6 = rnorm(1000000),
  v7 = rnorm(1000000),
  v8 = rnorm(1000000),
  v9 = rnorm(1000000),
  v10 = rnorm(1000000)
)

fSlowFunction <- function(v) {
  Sys.sleep(0.1)
  return(mean(log(abs(v) + 1)))
}

system.time({lapply(listV, fSlowFunction)})
#>   bruger   system forløbet
#>     0.27     0.03     1.39

suppressMessages(library(parallel))
cluster <- makeCluster(max(1, detectCores() - 1))

system.time(parLapply(cluster, listV, fSlowFunction))
#>   bruger   system forløbet
#>     0.07     0.03     0.39
stopCluster(cluster)
```

36.2 (2)

-
    -
        1.
        2.
    - a)
    - b)
    - c)
    - d)

```
Exponential.Simulate <- function(lambda, size) {
  U <- runif(size)
  return(mean(-1/lambda * log(U)))
}

set.seed(10086)

X <- Exponential.Simulate(0.5, 500)
```

```
system.time(lResult <- lapply(1:500, function(x) Exponential.Simulate(0.5, 500)))
#>   bruger   system forløbet
#>     0.02     0.00     0.02

suppressMessages(library(parallel))
cluster <- makeCluster(max(1, detectCores() - 1))
clusterExport(cluster, "Exponential.Simulate")
system.time(means_par <- parLapply(cluster, 1:500, function(x) Exponential.Simulate(0.5, 500
#>   bruger   system forløbet
#>     0.00     0.00     0.02
stopCluster(cluster)

# CLT suggests distribution of sample means will be approximately normal
hist(unlist(means_par))
```



36.3 (3)

-
  -

  - a)
  - b)

```
system.time(sapply(1:1000000, function(x) {sqrt(x); Sys.sleep(0.00001)}))
#>   bruger   system forløbet
#>     0.80     0.00     0.79

suppressMessages(library(parallel))
cluster <- makeCluster(max(1, detectCores() - 1))
system.time(parLapply(cluster, 1:1000000, function(x) {sqrt(x); Sys.sleep(0.00001)}))
#>   bruger   system forløbet
#>     0.07     0.00     0.27
stopCluster(cluster)
```

36.4 (4)

-
  - a)
  - b)
  - c)

```
# A: Task granularity high, communication overhead low.
# B: Embarassingly paralllel: Monte Carlo sims.
#    Difficult: Highly sequential algorithms like some recursive functions.
# C) RNG state per worker, data transfer, debugging
```

37 Advanced exercises

38 Exercise Set 9: Advanced R, Data Manipulation, and Debugging

38.1 (1)

-
  -
  - a)
  - b)
  - c)
  - d)
  -

```r
df_subset <- subset(iris, Species == "setosa" & Sepal.Length > 5.0)
df <- iris
df <- transform(df, Petal.Area = Petal.Length * Petal.Width)
agg_df <- aggregate(cbind(Sepal.Length, Petal.Area) ~ Species, data = df, FUN = mean)
agg_df[order(agg_df$Sepal.Length, decreasing = TRUE), ]
#>      Species Sepal.Length Petal.Area
#> 3  virginica        6.588    11.2962
#> 2 versicolor        5.936     5.7204
#> 1     setosa        5.006     0.3656
```

38.2 (2)

-
  - a)
  - b)
  - c)

– d)

```
vsDateStrings <- c("2023-01-15", "2023/03/22", "07-Apr-2023")
date_objects <- as.Date(vsDateStrings, tryFormats = c("%Y-%m-%d", "%Y/%m/%d", "%d-%b-%Y"))
weekdays(date_objects)
#> [1] "søndag" NA       NA
diff(range(date_objects))
#> Time difference of NA days
```

38.3 (3)

-
  -

```
fBuggyMean <- function(vData, bRemoveNA) {
          if (bRemoveNA = TRUE) { # Intentional bug: assignment instead of comparison
            vData <- vData[!is.na(vData)]
          }
          total_sum <- 0
          for (val in vData) {
            total_sum <- total_sum + val
          }
          return(total_sum / length(vData)) # Potential bug: division by zero if vData is emp
        }
```

- a)
- b)
- c)
- d)

```
fBuggyMean_corrected <- function(vData, bRemoveNA) {
  if (isTRUE(bRemoveNA)) { # Corrected comparison
    vData <- vData[!is.na(vData)]
  }
  if (length(vData) == 0) { # Handle empty vector
    return(NaN)
  }
  total_sum <- 0
  for (val in vData) { # Loop is fine, sum() is better
    total_sum <- total_sum + val
  }
  return(total_sum / length(vData))
}
fBuggyMean_corrected(c(1,2,NA,4), bRemoveNA=TRUE)
#> [1] 2.333333
fBuggyMean_corrected(c(NA,NA), bRemoveNA=TRUE)
#> [1] NaN
```

38.4 (4)

-
  -
  - a)
  - b)
  - c)

```
vsCodes <- c("PROD-A-123-X", "PROD-B-45-Y", "ITEM-C-6789-Z")
sapply(strsplit(vsCodes, "-"), function(x) x[2])
#> [1] "A" "B" "C"
as.numeric(sapply(strsplit(vsCodes, "-"), function(x) x[3]))
#> [1]  123   45 6789
gsub("PROD", "PRODUCT", vsCodes)
#> [1] "PRODUCT-A-123-X" "PRODUCT-B-45-Y"  "ITEM-C-6789-Z"
```

38.5 (5)

-
  -
  - a)
  - b)

- c)

```r
vX_cumprod <- 1:100 # Smaller for quick demo
fCumprodLoop <- function(v) {
  res <- numeric(length(v))
  if (length(v) > 0) res[1] <- v[1]
  if (length(v) > 1) {
    for (i in 2:length(v)) res[i] <- res[i - 1] * v[i]
  }
  return(res)
}
microbenchmark::microbenchmark(fCumprodLoop(vX_cumprod), cumprod(vX_cumprod))
#> Unit: nanoseconds
#>                      expr  min   lq  mean median   uq     max neval
#>  fCumprodLoop(vX_cumprod) 8400 8600 68958   8800 9000 6016200   100
#>       cumprod(vX_cumprod)  500  600   789    700  700    3200   100
```

39 Exercise Set 10: Advanced C++ and Package Development

39.1 (1)

-
  -
  -
    -
    -
    -
  -

```cpp
// [[Rcpp::depends(RcppArmadillo)]]
#include <RcppArmadillo.h>
// [[Rcpp::export]]
Rcpp::List arma_advanced_linalg(arma::mat M) {
  double dDet = arma::det(M);
  double dTrace = arma::trace(M);
  arma::mat mChol;
  bool chol_success = false;
  if(M.is_sympd()) { // Check if symmetric positive definite for Cholesky
     chol_success = arma::chol(mChol, M);
  }

  return Rcpp::List::create(
    Rcpp::Named("dDeterminant") = dDet,
    Rcpp::Named("dTrace") = dTrace,
    Rcpp::Named("mCholesky") = (chol_success ? Rcpp::wrap(mChol) : Rcpp::wrap(arma::mat()))
  );
}
```

39.2 (2)

-
  -
  - a)
    -
    -
  - b)
  - c)
  - d)
  - e)

```cpp
// [[Rcpp::depends(RcppArmadillo)]]
#include <RcppArmadillo.h>

// [[Rcpp::export]]
double weighted_mean_cpp(arma::vec x, arma::vec weights) {
  return arma::sum(x % weights) / arma::sum(weights); // % is element-wise product
}

// [[Rcpp::export]]
double variance_cpp(arma::vec x) {
  return arma::var(x, 0); // 0 for sample variance (N-1 denominator)
}
```

39.3 (3)

-
  -
  -
  -

```cpp
#include <Rcpp.h>
#include <algorithm> // For std::sort, std::unique
#include <vector>    // For std::vector

// [[Rcpp::export]]
Rcpp::NumericVector cpp_unique_sorted(Rcpp::NumericVector x) {
  if (x.size() == 0) {
    return Rcpp::NumericVector(0);
  }
  // Convert to std::vector for std algorithms
  std::vector<double> std_x = Rcpp::as<std::vector<double>>(x);

  std::sort(std_x.begin(), std_x.end());
  // std::unique moves unique elements to the front and returns an iterator
  // to the end of the unique range.
  std_x.erase(std::unique(std_x.begin(), std_x.end()), std_x.end());

  return Rcpp::wrap(std_x); // Convert back to Rcpp::NumericVector
}
```

39.4 (4)

-
    - a)
    - b)
        -
        -
        -
        -

40 Exercise Set 11: Advanced Simulation and Optimization

40.1 (1)

-
  -
  - a)
  - b)
  - c)
  - d)

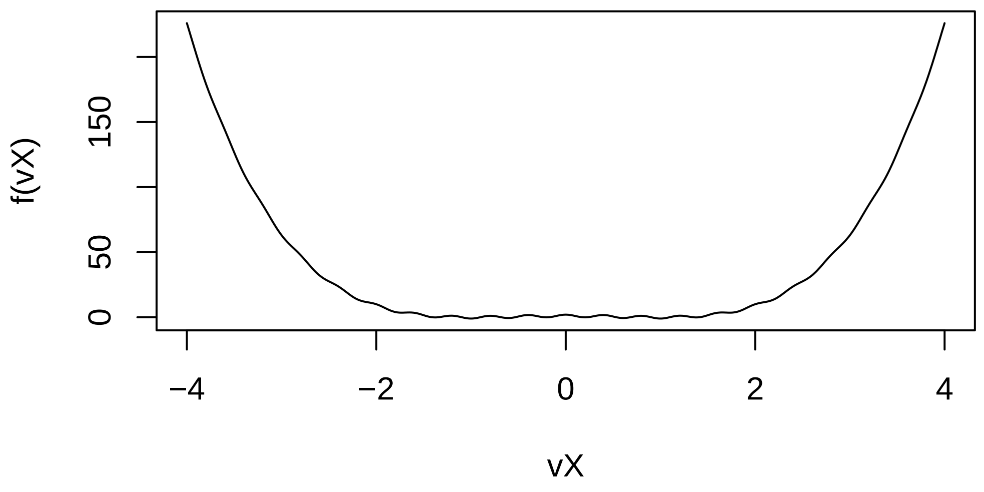
```r
h <- function(x) {
  return((x - 2)^2 + rnorm(1, 0, 0.1))
}

optimize(h, interval = c(-4, 4))$minimum
#> [1] 2.058113
optimize(h, interval = c(-4, 4))$minimum
#> [1] 1.775687
optimize(h, interval = c(-4, 4))$minimum
#> [1] 2.046855
optimize(h, interval = c(-4, 4))$minimum
#> [1] 1.835707

h_avg <- function(x, n_reps) {
  return(mean(replicate(n_reps, h(x))))
}

optimize(h_avg, interval = c(-4,4), n_reps = 100)
```

```
#> $minimum
#> [1] 1.998084
#>
#> $objective
#> [1] -0.0006209924
```

40.2 (2)

-
  -
  -
    1.
    2.
    3.
    4.
    5.
  - a)
  - b)
  - c)

```r
f <- function(x) {
  return((x^2 - 1)^2 + cos(5 * pi * x))
}

fSimAnn <- function(f, dLower, dUpper, dStart = (dUpper - dLower)/2, dCool, dStepSize, n.ite
  # start at a random point x_c
  x_c <- dStart
  T_new <- dT
  for (i in 1:n.iter) {
    x_n <- x_c + rnorm(1, 0, dStepSize)
    if (f(x_n) < f(x_c)) {
      x_c <- x_n
    } else if (runif(1) < exp(-(f(x_n) - f(x_c)) / T_new)) {
      x_c <- x_n
    }
    T_new <- T_new * dCool
  }
  return(x_c)
}

vX <- seq(-4, 4, 0.01)
plot(vX, f(vX), type = "l")
```



```
fSimAnn(f, -2, 2, dCool = 0.99, dT = 10, dStepSize = 0.1, n.iter = 5000)
#> [1] 0.9999699
```

40.3 (3)

-
  -
  - a)
  - b)
    1.

2.
    *
    *
3.
- c)
- d)

```r
vData <- rexp(50, rate = 0.2)
# Theta-hat
1/mean(vData)
#> [1] 0.1824065

# bootstrap
B <- 1000
vB <- numeric(B)
for (b in 1:B) {
  vB[b] <- 1/mean(sample(vData, length(vData), replace = TRUE))
}

mean(vB)
#> [1] 0.1862486
sqrt(var(vB))
#> [1] 0.02476446
```

41 Exercise Set 12: Exam-Style Problems and Combinations

41.1 (1)

-
    -
    -
    -
    - a)
    - b)
    - c)
    - d)
    - e)

```r
set.seed(42)
y_data <- c(rnorm(30, -2, 1), rnorm(70, 2, 1))

log_likelihood <- function(data, params, sigma_1, sigma_2) {
  logit_pi <- params[1]
  mu1 <- params[2]
  mu2 <- params[3]
  pi <- exp(logit_pi) / (1 + exp(logit_pi))

  return(-sum(log(pi * dnorm(data, mu1, sigma_1) + (1 - pi) * dnorm(data, mu2, sigma_2))))
}

logit_pi_hat <- optim(c(0.5, 0, 0), log_likelihood, data = y_data, sigma_1 = 1, sigma_2 = 1,
pi_hat <- exp(logit_pi_hat$par[1]) / (1 + exp(logit_pi_hat$par[1]))
pi_hat
#> [1] 0.7133243
logit_pi_hat$par[2]
#> [1] 1.988721
logit_pi_hat$par[3]
#> [1] -2.044431
```

41.2 (2)

-
  -
  -
    * a)
    *
    *
    *
    * b)
  -
    *
    * c)
    *
    *

– d)

```r
simulate_queue <- function(lambda, mu, n_customers) {
  if (n_customers == 0) return(0)
  arrival_times <- numeric(n_customers)
  service_start_times <- numeric(n_customers)
  service_end_times <- numeric(n_customers)
  service_times <- rexp(n_customers, lambda)

  # First customer
  arrival_times[1] <- 0
  service_start_times[1] <- arrival_times[1]
  service_end_times[1] <- service_start_times[1] + service_times[1]

  if (n_customers > 1) {
    inter_arrival_times <- rexp(n_customers - 1, rate = lambda)
    for (i in 2:n_customers) {
      arrival_times[i] <- arrival_times[i - 1] + inter_arrival_times[i - 1]
      service_start_times[i] <- max(arrival_times[i], service_end_times[i - 1])
      service_end_times[i] <- service_start_times[i] + service_times[i]
    }
  }
  waiting_times <- service_start_times - arrival_times
  return(mean(waiting_times))
}

simulate_queue(0.8, 1, 1000)
#> [1] 61.33853
simulate_queue(0.8, 1, 1000)
#> [1] 12.78504
simulate_queue(0.8, 1, 1000)
#> [1] 71.37086
simulate_queue(0.8, 1, 1000)
#> [1] 35.86602

g_lambda <- function(lambda) lambda / (1 * (1 - lambda)) - 5
uniroot(g_lambda, interval = c(1e-6, 0.999))$root
#> [1] 0.8333285
```

41.3 (3)

-
  -
  -
  -
  -
    - a)
    - b)

      .
      .
      .
    - c)
  -
    - d)
    - e)
    - f)
  - g)

```r
simulate_one_path_R <- function(S0, r, sigma, T_val, M) {
  dt <- T_val / M
  St_path <- numeric(M + 1)
  St_path[1] <- S0
  for (i in 1:M) {
    Zt <- rnorm(1)
    St_path[i+1] <- St_path[i] * exp((r - 0.5*sigma^2)*dt + sigma*sqrt(dt)*Zt)
  }
  return(mean(St_path[-1]))
}

price_asian_option <- function(S0, K, r, sigma, T_val, M, n_sims) {
  payoffs <- numeric(n_sims)
  for (i in 1:n_sims) {
    payoffs[i] <- max(0, simulate_one_path_R(S0, r, sigma, T_val, M) - K)
  }
  # alternative
  # avg_prices <- replicate(n_sims, simulate_one_path_R(S0, r, sigma, T_val, M))
  # payoffs <- pmax(0, avg_prices - K)
  #
  return(exp(-r * T_val) * mean(payoffs))
}

price_asian_option(100, 100, 0.05, 0.2, 1, 250, 1000)
#> [1] 5.762387
```

42 Extra extra exercises

43 Exercise Set 13: R Fundamentals and Data Structures (Review)

43.1 (1)

- a)
- b)
- c)
- d)

```
vSeq <- seq(10, -10, -0.5)
length(vSeq)
#> [1] 41
bIsPositive <- vSeq > 0
prod(vSeq[vSeq > 1 & vSeq < 5])
#> [1] 1417.5
```

43.2 (2)

- a)
- b)
- c)
- d)

```
mIdentity <- diag(4)
mValues <- matrix(1:16, 4, 4, byrow = TRUE)
mC <- mIdentity %*% mValues
mC
#>      [,1] [,2] [,3] [,4]
#> [1,]    1    2    3    4
#> [2,]    5    6    7    8
#> [3,]    9   10   11   12
#> [4,]   13   14   15   16
diag(mValues) <- 0
mValues
#>      [,1] [,2] [,3] [,4]
#> [1,]    0    2    3    4
#> [2,]    5    0    7    8
#> [3,]    9   10    0   12
#> [4,]   13   14   15    0
```

43.3 (3)

-
  -
  -
    -
    -
    -
    -
  -

```
set.seed(777)

iCount <- 0
dRunningSum <- 0

repeat {
  dRunningSum <- dRunningSum + sample(1:20, 1)
  iCount <- iCount + 1

  if (dRunningSum > 100 || iCount > 15) {
    break
  }
}

print(paste("Sum:", dRunningSum, "Count:", iCount))
#> [1] "Sum: 106 Count: 11"
```

43.4 (4)

-
    -
    -
    -
        -
        -
- a)
- b)
- c)

```
lExperiment <- list(
  sExpID = "EXP007",
  dParams = c(alpha = 0.05, beta = 1.2),
  lResults = list(
    vObservations = c(10.2, 11.1, NA, 9.8, 10.5, NA, 10.9),
    sStatus = "Preliminary"
  )
)

lExperiment[["dParams"]][["beta"]]
#> [1] 1.2
mean(lExperiment[["lResults"]][["vObservations"]], na.rm = TRUE)
#> [1] 10.5
lExperiment[["lResults"]][["sStatus"]] <- "Completed"
lExperiment[["lResults"]][["sStatus"]]
#> [1] "Completed"
```

43.5 (5)

- a)

- b)

- c)
- d)

```r
dfSales <- data.frame(
  Month = c("Jan", "Feb", "Mar", "Apr"),
  ProductA_Units = c(100, 120, 90, 110),
  ProductB_Units = c(80, 85, 95, 70)
)
dfSales <- transform(dfSales, ProductA_Share = ProductA_Units / (ProductA_Units + ProductB_Un
dfSales$Month[dfSales$ProductB_Units > 90]
#> [1] "Mar"
```

43.6 (6)

-
-

```r
vScores <- c(45, 88, 62, 95, 70, 55)
vCategory <- ifelse(vScores < 50, "Fail", ifelse(vScores < 70, "Pass", ifelse(vScores < 90, "
vCategory
#> [1] "Fail"        "Merit"       "Pass"        "Distinction" "Merit"
#> [6] "Pass"
```

43.7 (7)

-
  -
  -
  -

```r
for (k in 1:10) {
  if (k %% 2 == 0) {
    print(paste0(k, " is even."))
  } else if (k %% 3 == 0) {
    print(paste0(k, " is odd and a multiple of 3."))
  } else {
    print(paste0(k, " is odd."))
  }
}
#> [1] "1 is odd."
#> [1] "2 is even."
#> [1] "3 is odd and a multiple of 3."
#> [1] "4 is even."
#> [1] "5 is odd."
#> [1] "6 is even."
#> [1] "7 is odd."
#> [1] "8 is even."
#> [1] "9 is odd and a multiple of 3."
#> [1] "10 is even."
```

43.8 (8)

- a)
- b)
- c)
- d)

```
vsSentences <- c("R is fun", "Data analysis with R", "Missing data NA", "Another sentence")
grep("R", vsSentences)
#> [1] 1 2
sub("R", "R Language", vsSentences)
#> [1] "R Language is fun"             "Data analysis with R Language"
#> [3] "Missing data NA"               "Another sentence"
strsplit(vsSentences[2], " ")
#> [[1]]
#> [1] "Data"     "analysis" "with"     "R"
```

44 Exercise Set 14: Functions, Scope, and Vectorization Focus

44.1 (1)

-
-
-
-

```r
fNormalizeVector <- function(vX) {
  if (sd(vX) == 0) {
    warning("Standard deviation is 0 or all NA. Returning NAs or zeros.")
    return(rep(NA, length(vX))) # Or zeros: numeric(length(vX)
  } else {
    return((vX - mean(vX)) / sd(vX))
  }
}

fNormalizeVector(c(1,2,3,4,5))
#> [1] -1.2649111 -0.6324555  0.0000000  0.6324555  1.2649111
fNormalizeVector(c(5,5,5,5))
#> Warning in fNormalizeVector(c(5, 5, 5, 5)): Standard deviation is 0 or all NA.
#> Returning NAs or zeros.
#> [1] NA NA NA NA
```

44.2 (2)

-

```r
fOuter <- function(a) {
      b <- a * 2
      fInner <- function(c) {
        return(b + c) # 'b' is from fOuter's environment
      }
      return(fInner)
    }
myInnerFunc <- fOuter(10)
result <- myInnerFunc(5)
result
#> [1] 25
```

44.3 (3)

-
-
-
  - a)
  - b)
-

```r
# a
fCountValuesInRanges <- function(vData, vBreakpoints) {
  full_breaks <- c(-Inf, sort(unique(vBreakpoints)), Inf)
  counts <- numeric(length(full_breaks) - 1)
  for (val in vData) {
    for (j in 1:(length(full_breaks) - 1)) {
      if (val > full_breaks[j] && val <= full_breaks[j + 1]) {
        counts[j] <- counts[j] + 1
        break
      }
    }
  }
  names(counts) <- cut((full_breaks[-1] + full_breaks[-length(full_breaks)])/2 , breaks=full_
  return(counts)
}

fCountValuesInRangesCut <- function(vData, vBreakpoints) {
  full_breaks <- c(-Inf, sort(unique(vBreakpoints)), Inf)
  return(table(cut(vData, breaks = full_breaks, include.lowest = TRUE, right = TRUE)))
}

vData <- rnorm(1000, mean = 15, sd = 10)
vBreakpoints <- c(0, 10, 20, 30)
fCountValuesInRanges(vData, vBreakpoints)
#>  [-Inf,0]    (0,10]   (10,20]   (20,30] (30, Inf]
#>        81       258       394       209        58
fCountValuesInRangesCut(vData, vBreakpoints)
#>
#>  [-Inf,0]    (0,10]   (10,20]   (20,30] (30, Inf]
#>        81       258       394       209        58
```

44.4 (4)

-

```
fRecursiveSum <- function(vX) {
  if (length(vX) == 0) return(0)
  return(vX[1] + fRecursiveSum(vX[-1]))
}
fRecursiveSum(1:5)
#> [1] 15
```

44.5 (5)

-

```r
set.seed(1)
lMatrices <- lapply(1:5, function(i) matrix(sample(1:50, sample(5:10,1)*3, replace=TRUE), nc
```

-

```r
sapply(lMatrices, function(x) sum(diag(x)))
#> [1]  64  67  92  68 125
```

44.6 (6)

-

44.7 (7)

-
-
  - a)
  - b)
-

```r
suppressMessages(library(microbenchmark))
#> Warning: pakke 'microbenchmark' blev bygget under R version 4.3.3

fLoopApproach <- function(x) {
  vResults <- numeric(x)
  for (i in 1:x) {
    if (i %% 2 == 0) {
      vResults[i] <- i / 2
    } else {
      vResults[i] <- 3 * i + 1
    }
  }
  return(vResults)
}

fVectorizedApproach <- function(x) {
  return(ifelse(x %% 2 == 0, x / 2, 3 * x + 1))
}

microbenchmark(fLoopApproach(100000), fVectorizedApproach(100000))
#> Unit: microseconds
#>                         expr     min      lq      mean   median       uq     max
#>         fLoopApproach(1e+05) 18045.7 18801.8 19785.600 19556.05 19939.35 27552.7
#>  fVectorizedApproach(1e+05)     1.3     2.0    45.405     3.35    13.75  3770.1
#>  neval
#>    100
#>    100
```

44.8 (8)

-

```
df <- data.frame(
      group = rep(c("A", "B", "C"), each = 4),
      value = rnorm(12)
    )
```

-

```
tapply(df$value, df$group, function(x) diff(range(x)))
#>        A        B        C
#> 2.854650 1.063354 2.165598
```

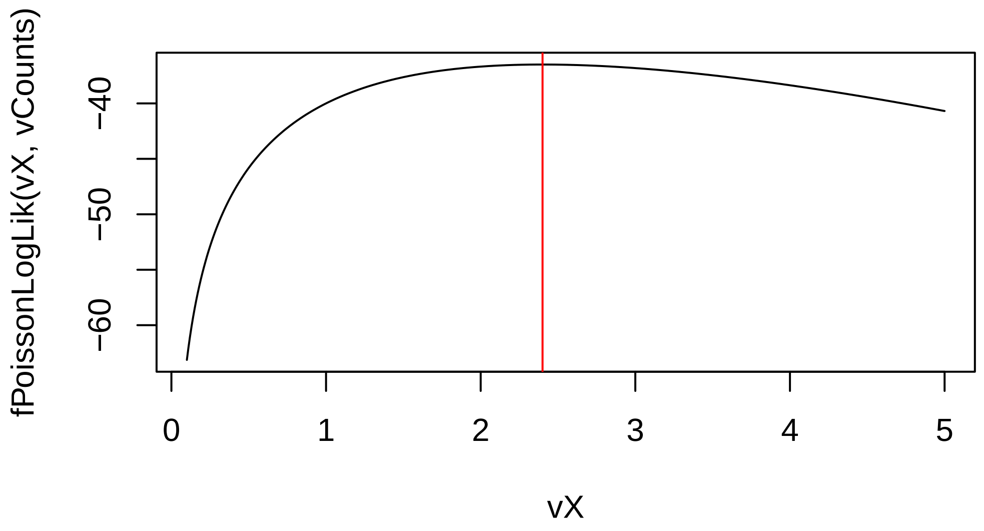
45 Exercise Set 15: Likelihood Functions and Optimization (Medium/Hard)

45.1 (1) Likelihood of a Poisson Process

-
-
- a)
- b)
- c)
- d)
- e)

```r
fPoissonLogLik <- function(dLambda, vCounts) {
  dSum <- 0
  for (i in 1:length(vCounts)) {
    dSum <- dSum + vCounts[i] * log(dLambda) - dLambda - factorial(vCounts[i])
  }
  return(dSum)
}

vCounts <- c(2, 3, 1, 2, 4)

vX <- seq(0.1, 5, 0.01)
```

```r
plot(vX, fPoissonLogLik(vX, vCounts), type = "l")
abline(v = mean(vCounts), col = "red")
```



```r
mean(vCounts)
#> [1] 2.4
optimize(fPoissonLogLik, interval = c(0.1, 5), vCounts = vCounts, maximum = TRUE)
#> $maximum
#> [1] 2.400006
#>
#> $objective
#> [1] -36.49438
```

```cpp
// [[Rcpp::depends(RcppArmadillo)]]
#include <RcppArmadillo.h>
using namespace Rcpp;
using namespace arma;

//[[Rcpp::export]]
double poisson_log_lik_cpp(int k, double lambda) {
  return (k * log(lambda) - lambda - tgamma(k + 1));
}
```

```
//[[Rcpp::export]]
double total_poisson_log_lik_cpp(Rcpp::IntegerVector counts, double lambda) {
  double dSum = 0.0;
  for (int i = 0; i < counts.size(); i++) {
    dSum += poisson_log_lik_cpp(counts[i], lambda);
  }
  return dSum;
}
```

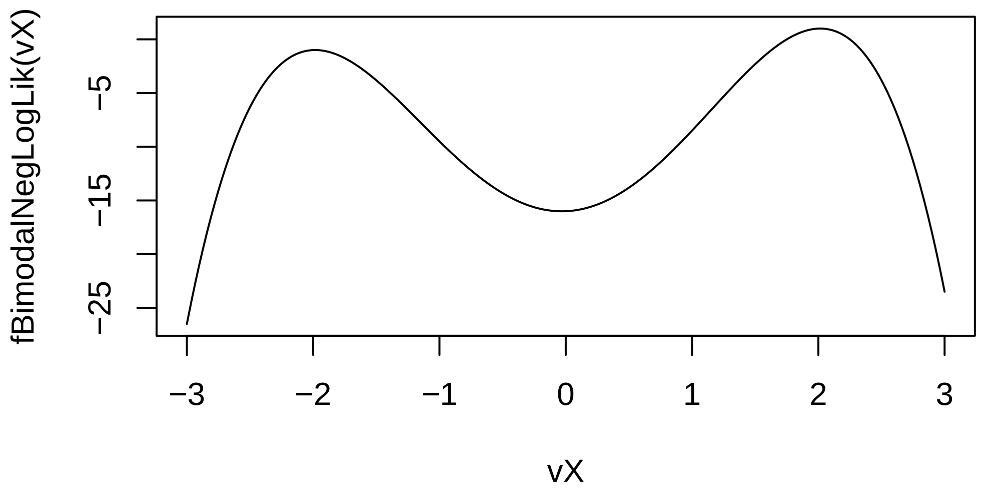
```
suppressMessages(library(Rcpp))
#> Warning: pakke 'Rcpp' blev bygget under R version 4.3.3
sourceCpp("extra_extra_cpp.cpp")

total_poisson_log_lik_cpp(vCounts, mean(vCounts))
#> [1] -36.49438
fPoissonLogLik(mean(vCounts), vCounts)
#> [1] -36.49438
```

45.2 (2) Optimization of a Bimodal Likelihood (Reparameterization)

-
- a)
- b)
- c)
- d)
- e)

```
fBimodalNegLogLik <- function(dTheta) {
  return(-(dTheta^2 - 4)^2 + 0.5 * dTheta)
}
```

```
vX <- seq(-3, 3, 0.01)
plot(vX, fBimodalNegLogLik(vX), type = "l")
```



```
optim(-2, fBimodalNegLogLik, method = "BFGS")$par
#> [1] -1.147232e+15
optim(0, fBimodalNegLogLik, method = "BFGS")$par
#> [1] -0.03125764
optim(2, fBimodalNegLogLik, method = "BFGS")$par
#> [1] -1.160693e+34

fBimodalNegLogLik_Pos <- function(dTheta_tilde) {
  dTheta <- exp(dTheta_tilde)
  return(-(dTheta^2 - 4)^2 + 0.5 * dTheta)
}

dTheta_tilde_optim <- optim(0, fBimodalNegLogLik_Pos, method = "BFGS")$par
dTheta_optim <- exp(dTheta_tilde_optim)
dTheta_optim
#> [1] 3.726639e-06

fBimodalNegLogLik_Neg <- function(dTheta_tilde) {
  dTheta <- -exp(dTheta_tilde)
```

```r
  return(-(dTheta^2 - 4)^2 + 0.5 * dTheta)
}

dTheta_tilde_optim <- optim(0, fBimodalNegLogLik_Neg, method = "BFGS")$par
dTheta_optim <- -exp(dTheta_tilde_optim)
dTheta_optim
#> [1] -1.01302e-05
```

45.3 (3) Likelihood for Linear Regression with Known Variance

-
-
-
- a)
- b)
- c)
- d)

```r
set.seed(12)
x_data <- 1:20
y_data <- 0.5 + 2 * x_data + rnorm(20, 0, 1)

fRegressNegLogLik <- function(vBeta, vY, vX, dSigma2 = 1) {
  dSum <- 0
  for (i in 1:length(vY)) {
    dSum <- dSum + (vY[i] - (vBeta[1] + vBeta[2] * vX[i]))^2 / (2 * dSigma2)
  }
  return(dSum)
}

optim(c(0, 0), fRegressNegLogLik, vY = y_data, vX = x_data, method = "BFGS")$par
#> [1] -0.3311065  2.0476093
```

```
coef(lm(y_data ~ x_data))
#> (Intercept)      x_data
#>  -0.3311065   2.0476093
```

45.4 (4) Optimization with Constraints using `constrOptim` (or Penalty)

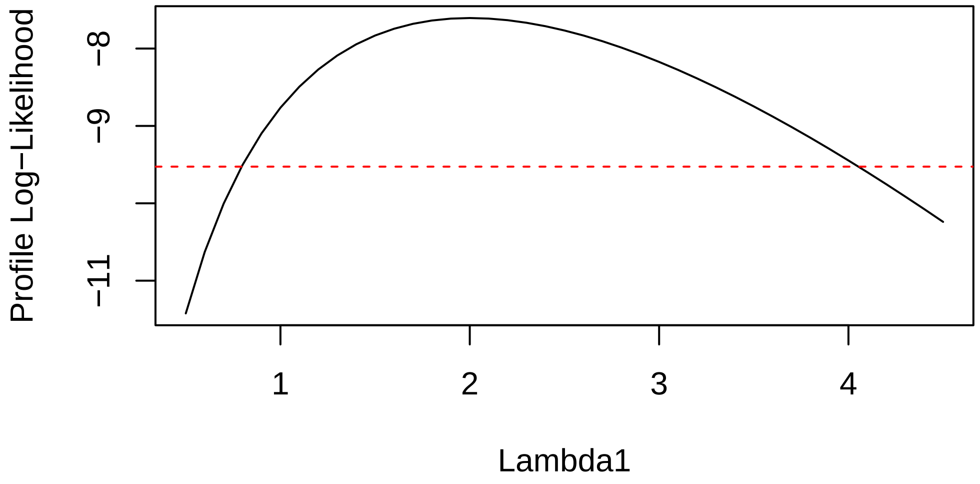
-
-
  - a)
  - b)
-
  - c)
  - d)
  - e)

```
f <- function(vX) {
  return((vX[1] - 1)^2 + (vX[2] - 2)^2)
}

p <- function(vX) {
  return(max(0, 4 - (vX[1] + vX[2]))^2)
}

f_p <- function(vX, ...) {
  return(f(vX) + 1000 * p(vX))
}

optim(c(3, 3), f_p, method = "BFGS")$par
#> [1] 1.499802 2.499780
```

45.5 (5) Profile Likelihood

-
-
-
- a)
- b)
- c)
- d)

```
vCounts <- c(2, 3, 1, 2, 4)

fCombinedLogLik <- function(vLambda, vC) {
  dSum1 <- 0.0
  dSum2 <- 0.0
  for (i in 1:3) {
    dSum1 <- dSum1 + log(dpois(vCounts[i], vLambda[1]))
  }
  for (i in 4:5) {
    dSum2 <- dSum2 + log(dpois(vCounts[i], vLambda[2]))
  }
  return(dSum1 + dSum2)
}

fProfileLogLik_L1 <- function(dLambda1_fixed, vC) {
  # Objective for optimize: function of lambda2
  obj_for_lambda2 <- function(dLambda2) {
    fCombinedLogLik(c(dLambda1_fixed, dLambda2), vC)
  }
  # Optimize returns list, $maximum is the value of objective function
  optimize(obj_for_lambda2, interval=c(0.01, 10), maximum=TRUE)$objective
}

lambda1_seq <- seq(0.5, 4.5, 0.1)
profile_loglik_values <- sapply(lambda1_seq, fProfileLogLik_L1, vC=vCounts)

plot(lambda1_seq, profile_loglik_values, type="l", xlab="Lambda1", ylab="Profile Log-Likelih
abline(h = max(profile_loglik_values) - qchisq(0.95,1)/2, col="red", lty=2)
```



46 Likelihood Exercises

47 Exercise Set 16: Advanced Econometric Likelihoods and Optimization

47.1 (1) Probit Model Likelihood (Medium-Hard)

-
-
-

```r
set.seed(123)
N <- 200
X1 <- rnorm(N)
X2 <- runif(N)
X_matrix <- cbind(1, X1, X2) # Design matrix with intercept
beta_true <- c(-0.5, 1.2, -0.8)
linear_predictor <- X_matrix %*% beta_true
prob_y1 <- pnorm(linear_predictor)
Y_binary <- rbinom(N, 1, prob_y1)
```

- a)
- b)
- c)
- d)

- e)

```r
#data
set.seed(123)
N <- 200
X1 <- rnorm(N)
X2 <- runif(N)
X_matrix <- cbind(1, X1, X2) # matrix with intercept
beta_true <- c(-0.5, 1.2, -0.8)
linear_predictor <- X_matrix %*% beta_true
prob_y1 <- pnorm(linear_predictor)
Y_binary <- rbinom(N, 1, prob_y1)

#a
fProbitNegLogLik <- function(vBeta, mX, vY) {
  n <- length(vY)
  dSum <- 0
  for (i in 1:n) {
    dSum <- dSum + vY[i] * log(pnorm(mX %*% vBeta))[i] + (1 - vY[i]) * log(1 - pnorm(mX %*% v
  }
  return(-dSum)
  # alternative
  # return(-sum(vY * log(pnorm(mX %*% vBeta)) + (1 - vY) * log(1 - pnorm(mX %*% vBeta))))
}

#b
optim(c(0, 0, 0), fProbitNegLogLik, mX = X_matrix, vY = Y_binary, method = "BFGS")$par
#> [1] -0.6124990  1.6593570 -0.7099176

#c
glm(Y_binary ~ X1 + X2, family = binomial(link = "probit"))$coefficients
#> (Intercept)          X1          X2
#>  -0.6125001   1.6593555  -0.7099154

#d
fProbitGrad <- function(vBeta, mX, vY) {
  n <- length(vY)
  dSum <- 0
  dProb <- pnorm(mX %*% vBeta)
  for (i in 1:n) {
    dSum <- dSum + (vY[i] - dProb[i]) / (dProb[i] * (1 - dProb)[i]) * dnorm(mX %*% vBeta)[i]
  }
  return(-dSum)
}

optim(c(0, 0, 0), fProbitNegLogLik, gr = fProbitGrad, mX = X_matrix, vY = Y_binary, method =
#> [1] -0.6124989  1.6593568 -0.7099176
```

```
// [[Rcpp::depends(RcppArmadillo)]]
#include <RcppArmadillo.h>
using namespace Rcpp;
using namespace arma;

// [[Rcpp::export]]
double probit_neg_log_lik_cpp(vec beta, mat X, vec Y) {
  int n = Y.size();
  double dSum = 0.0;
  vec linear_term = X * beta;
  vec dProb = zeros<vec>(X.n_rows);

  for (int i = 0; i < n; i++) {
    dProb(i) = R::pnorm(linear_term(i), 0.0, 1.0, 1, 0);
    dSum += Y[i] * log(dProb[i]) + (1 - Y[i]) * log(1 - dProb[i]);
  }
  return(-dSum);
}
```

```
#e
suppressMessages(library(Rcpp))
#> Warning: pakke 'Rcpp' blev bygget under R version 4.3.3
suppressMessages(library(RcppArmadillo))
#> Warning: pakke 'RcppArmadillo' blev bygget under R version 4.3.3
sourceCpp("likelihood_exercises.cpp")

probit_neg_log_lik_cpp(beta_true, X_matrix, Y_binary)
#> [1] 66.87324
fProbitNegLogLik(beta_true, X_matrix, Y_binary)
#> [1] 66.87324
```

47.2 (2) GARCH(1,1) Model Likelihood (Hard)

-
-
-
-
-

```r
set.seed(456)
T_garch <- 500
omega_true <- 0.1; alpha_true <- 0.1; beta_true <- 0.85
sigma2_garch <- numeric(T_garch)
r_garch <- numeric(T_garch)
sigma2_garch[1] <- omega_true / (1 - alpha_true - beta_true) # Unconditional variance
r_garch[1] <- sqrt(sigma2_garch[1]) * rnorm(1)
for(t_idx in 2:T_garch) {
  sigma2_garch[t_idx] <- omega_true + alpha_true * r_garch[t_idx-1]^2 + beta_true * sigma2_ga
  r_garch[t_idx] <- sqrt(sigma2_garch[t_idx]) * rnorm(1)
}
```

- a)
- b)
  -
  -
  -
- c)

- d)

- e)

```r
#data

set.seed(456)
T_garch <- 500
omega_true <- 0.1; alpha_true <- 0.1; beta_true <- 0.85
sigma2_garch <- numeric(T_garch)
r_garch <- numeric(T_garch)
sigma2_garch[1] <- omega_true / (1 - alpha_true - beta_true) # unconditional variance
r_garch[1] <- sqrt(sigma2_garch[1]) * rnorm(1)
for (t_idx in 2:T_garch) {
  sigma2_garch[t_idx] <- omega_true + alpha_true * r_garch[t_idx-1]^2 + beta_true * sigma2_ga
  r_garch[t_idx] <- sqrt(sigma2_garch[t_idx]) * rnorm(1)
}

#a
fGARCH11NegLogLik <- function(vParams, vReturns, sigma2_initial) {
  dOmega <- vParams[1]
  dAlpha <- vParams[2]
  dBeta <- vParams[3]

  dSum <- 0
  vSigma2 <- numeric(length(vReturns))
  vSigma2[1] <- sigma2_initial
  for (t in 2:length(vReturns)) {
    vSigma2[t] <- dOmega + dAlpha * vReturns[t-1]^2 + dBeta * vSigma2[t-1]
    dSum <- dSum - 0.5*(log(2*pi) + log(vSigma2[t]) + vReturns[t]^2 / vSigma2[t])
  }
  return(-dSum)
}

#b
fGARCH11NegLogLikRepar <- function(vParams, vReturns, sigma2_initial) {
  dOmega <- exp(vParams[1])
  dAlpha <- exp(vParams[2]) / (1 + exp(vParams[2]) + exp(vParams[3]))
  dBeta <- exp(vParams[3]) / (1 + exp(vParams[2]) + exp(vParams[3]))
  dSum <- 0
  vSigma2 <- numeric(length(vReturns))
  vSigma2[1] <- sigma2_initial
  for (t in 2:length(vReturns)) {
    vSigma2[t] <- dOmega + dAlpha * vReturns[t-1]^2 + dBeta * vSigma2[t-1]
    dSum <- dSum - 0.5*(log(2*pi) + log(vSigma2[t]) + vReturns[t]^2 / vSigma2[t])
  }
  return(-dSum)
}

#c
dPar_tilde <- optim(c(0, 0, 0), fGARCH11NegLogLikRepar, sigma2_initial = var(r_garch), vRetu


#d
dOmega_star <- exp(dPar_tilde[1])
dAlpha_star <- exp(dPar_tilde[2]) / (1 + exp(dPar_tilde[2]) + exp(dPar_tilde[3]))
dBeta_star <- exp(dPar_tilde[3]) / (1 + exp(dPar_tilde[2]) + exp(dPar_tilde[3]))

print(paste0("Omega est: ", dOmega_star, " vs. true: ", omega_true))
#> [1] "Omega est: 0.188330580842483 vs. true: 0.1"
print(paste0("Alpha est: ", dAlpha_star, " vs. true: ", alpha_true))
#> [1] "Alpha est: 0.120768680865144 vs. true: 0.1"
print(paste0("Beta est: ", dBeta_star, " vs. true: ", beta_true))
#> [1] "Beta est: 0.768485839334849 vs. true: 0.85"
```

```cpp
double fGARCH11NegLogLik_cpp(vec vParams, vec vReturns, double sigma2_initial) {
  double dOmega = vParams[0];
  double dAlpha = vParams[1];
  double dBeta = vParams[2];
  int n = vReturns.size();

  double dSum = 0;
  vec vSigma2 = zeros<vec>(vReturns.size());
  vSigma2[0] = sigma2_initial;
  double dPi = atan(1)*4;

  for (int t = 1; t < n; t++) {
    vSigma2[t] = dOmega + dAlpha * pow(vReturns[t-1], 2) + dBeta * vSigma2[t-1];
    dSum = dSum - 0.5*(log(2*dPi) + log(vSigma2[t]) + pow(vReturns[t], 2) / vSigma2[t]);
  }
  return(-dSum);
}
```

```
suppressMessages(library(Rcpp))
suppressMessages(library(RcppArmadillo))
sourceCpp("likelihood_exercises.cpp")

fGARCH11NegLogLik(c(omega_true, alpha_true, beta_true), r_garch, var(r_garch))
#> [1] 830.2781
fGARCH11NegLogLik_cpp(c(omega_true, alpha_true, beta_true), r_garch, var(r_garch))
#> [1] 830.2781
```

47.3 (3) Heckman Selection Model Likelihood (Extremely Hard)

-
    1.
    2.

    –
-
    –
    –
-
-

```
set.seed(789)
N_heck <- 300
W1 <- rnorm(N_heck); W_matrix <- cbind(1, W1) # Selection vars
X1 <- rnorm(N_heck); X_matrix_outcome <- cbind(1, X1) # Outcome vars
gamma_true <- c(0.5, -1); beta_true <- c(1, 2); sigma_true <- 1.5; rho_true <- 0.6

# Simulate errors
errors <- MASS::mvrnorm(N_heck, mu=c(0,0), Sigma=matrix(c(1, rho_true*sigma_true, rho_true*s
u_i <- errors[,1]
epsilon_i <- errors[,2]

z_star <- W_matrix %*% gamma_true + u_i
```

```
Z_select <- ifelse(z_star > 0, 1, 0)

Y_outcome <- X_matrix_outcome %*% beta_true + epsilon_i
Y_observed <- ifelse(Z_select == 1, Y_outcome, NA) # Only observe Y if Z_select is 1
df_heckman <- data.frame(Y_observed, X1, W1, Z_select)
df_heckman_obs <- subset(df_heckman, Z_select == 1) # data for outcome eq.
```

- a)
- b)
- c)
- d)
-

```
#data
set.seed(789)
N_heck <- 300
W1 <- rnorm(N_heck); W_matrix <- cbind(1, W1) # Selection vars
X1 <- rnorm(N_heck); X_matrix_outcome <- cbind(1, X1) # Outcome vars
gamma_true <- c(0.5, -1); beta_true <- c(1, 2); sigma_true <- 1.5; rho_true <- 0.6

# Simulate errors
errors <- MASS::mvrnorm(N_heck, mu=c(0,0), Sigma=matrix(c(1, rho_true*sigma_true, rho_true*s
u_i <- errors[,1]
epsilon_i <- errors[,2]

z_star <- W_matrix %*% gamma_true + u_i
Z_select <- ifelse(z_star > 0, 1, 0)

Y_outcome <- X_matrix_outcome %*% beta_true + epsilon_i
Y_observed <- ifelse(Z_select == 1, Y_outcome, NA) # Only observe Y if Z_select is 1
df_heckman <- data.frame(Y_observed, X1, W1, Z_select)
df_heckman_obs <- subset(df_heckman, Z_select == 1) # data for outcome eq.
```

```
#a + b
fHeckmanNegLogLik <- function(vParams, dfData, mSelectionVars, mOutcomeVars) {
  #vParams contains all parameters.
  #dfData should contain the outcome y, the selection indicator z, and covariates.
  #mSelectionVars and mOutcomeVars are column names/indices for w and x.
  vGamma <- vParams[1:2]
  vBeta <- vParams[3:4]
  dSigma <- exp(vParams[5])
  dRho <- (exp(vParams[6]) - exp(-vParams[6])) / (exp(vParams[6]) + exp(-vParams[6]))

  vY <- dfData[, mOutcomeVars]
  vZ <- dfData[, mSelectionVars]
  vX <- cbind(1, dfData$X1)
  vW <- cbind(1, dfData$W1)
  dLlSum <- 0
  for (i in 1:length(vY)) {
    if (vZ[i] == 0) {
      dLlSum <- dLlSum + log(1 - pnorm(vW %*% vGamma)[i])
    } else {
      dLlSum <- dLlSum + log(1 / dSigma * dnorm((vY[i] - (vX %*% vBeta)[i]) / dSigma) * pnorm
    }
  }
  return(-dLlSum)
}

# c
vParams <- optim(c(c(0, 0), c(0, 0), 0, 0), fHeckmanNegLogLik, dfData = df_heckman, mSelecti

# d
gamma_star <- vParams[1:2]
beta_star <- vParams[3:4]
sigma_star <- exp(vParams[5])
rho_star <- (exp(vParams[6]) - exp(-vParams[6])) / (exp(vParams[6]) + exp(-vParams[6]))

print(paste0("Gamma est: ", gamma_star, " vs. true: ", gamma_true))
#> [1] "Gamma est: 0.434869076735787 vs. true: 0.5"
#> [2] "Gamma est: -1.17799006585571 vs. true: -1"
print(paste0("Beta est: ", beta_star, " vs. true: ", beta_true))
#> [1] "Beta est: 0.964253846625409 vs. true: 1"
#> [2] "Beta est: 1.921924920477 vs. true: 2"
print(paste0("Sigma est: ", sigma_star, " vs. true: ", sigma_true))
#> [1] "Sigma est: 1.53499526849203 vs. true: 1.5"
```

```r
print(paste0("Rho est: ", rho_star, " vs. true. ", rho_true))
#> [1] "Rho est: 0.597269354835849 vs. true. 0.6"
```

47.4 (4) Vector Autoregression (VAR) Likelihood (Multivariate Normal) (Hard)

-

-

-

-

-

```r
library(vars)
data(Canada)
Canada_diff <- data.frame(apply(Canada[,c("e","prod")], 2, diff))
Y_var_data <- as.matrix(Canada_diff)
T_var <- nrow(Y_var_data) -1 # Number of observations for LL sum
```

- a)
- b)
- c)
- d)

- e)

```
# data
suppressMessages(library(vars))
#> Warning: pakke 'vars' blev bygget under R version 4.3.3
#> Warning: pakke 'strucchange' blev bygget under R version 4.3.3
#> Warning: pakke 'zoo' blev bygget under R version 4.3.3
#> Warning: pakke 'sandwich' blev bygget under R version 4.3.3
#> Warning: pakke 'urca' blev bygget under R version 4.3.3
#> Warning: pakke 'lmtest' blev bygget under R version 4.3.3
data(Canada)
Canada_diff <- data.frame(apply(Canada[,c("e","prod")], 2, diff))
Y_var_data <- as.matrix(Canada_diff)
T_var <- nrow(Y_var_data) -1 # Number of observations for LL sum

# a + b
fVAR1NegLogLik <- function(vParams, mY) {
  vC <- vParams[1:2]
  mA <- matrix(vParams[3:6], 2, 2)
  mSigma <- matrix(c(exp(vParams[7]), vParams[8], 0, exp(vParams[9])), 2, 2)
  mSigma <- mSigma %*% t(mSigma)
  iK <- ncol(mY)

  dSum <- 0
  for (t in 2:nrow(mY)) {
    dSum <- dSum - iK / 2 * log(2 * pi) - 0.5 * log(det(mSigma)) - 0.5 * t(mY[t, ] - vC - mA
  }
  return(-dSum)
}

# c
start_c_var <- c(0,0)
start_A_var <- as.vector(t(diag(0.5, 2)))
start_L_tilde_var <- c(log(1), 0, log(1))
start_params_var <- c(start_c_var, start_A_var, start_L_tilde_var)
optim_res_var <- optim(start_params_var, fVAR1NegLogLik, mY = Y_var_data, method = "L-BFGS-B
optim_res_var$par
#> [1]  0.10033694  0.06371892  0.67883561  0.12431749  0.19044272  0.27667404
#> [7] -0.96917510  0.01153122 -0.38346578
```

```
double fVAR1NegLogLik(vec vParams, mat mY) {
  vec vC = zeros<vec>(2);
  vC[0] = vParams[0];
  vC[1] = vParams[1];
  mat mA(2,2);
  mA(0,0) = vParams[2];
  mA(1,0) = vParams[3];
  mA(0,1) = vParams[4];
  mA(1,1) = vParams[5];
  mat mSigma(2,2);
  mSigma(0,0) = exp(vParams[6]);
  mSigma(0,1) = 0;
  mSigma(1,0) = vParams[7];
  mSigma(1,1) = exp(vParams[8]);
  mSigma = mSigma * mSigma.t();
  double iK = mY.n_cols;
  double dPi = atan(1)*4;

  double dSum = 0.0;
  for (arma::uword t = 1; t < mY.n_rows; t++) {
    vec u_t = mY.row(t).t() - vC - mA * mY.row(t-1).t();
    dSum = dSum - iK / 2.0 * log(2.0 * dPi) - 0.5 * log(det(mSigma)) - 0.5 * as_scalar(trans
  }

  return(-dSum);
}
```

```
suppressMessages(library(Rcpp))
suppressMessages(library(RcppArmadillo))
sourceCpp("likelihood_exercises.cpp")
```

48 Exercise Set 17: Integrated Econometric Modeling Problems

48.1 (1) Logistic Regression: Full Implementation (Hard)

-
-
-

```r
set.seed(123)
N_log <- 200
X1_log <- rnorm(N_log)
X2_log <- runif(N_log)
X_matrix_log <- cbind(1, X1_log, X2_log)
beta_true_log <- c(-0.5, 1.2, -0.8)
linear_predictor_log <- X_matrix_log %*% beta_true_log
prob_y1_log <- exp(linear_predictor_log) / (1 + exp(linear_predictor_log))
Y_binary_log <- rbinom(N_log, 1, prob_y1_log)
```

-
  - a)
  - b)
    -
    -
  - c)

- d)

- e)

- f)

```r
# data
set.seed(123)
N_log <- 200
X1_log <- rnorm(N_log)
X2_log <- runif(N_log)
X_matrix_log <- cbind(1, X1_log, X2_log)
beta_true_log <- c(-0.5, 1.2, -0.8)
linear_predictor_log <- X_matrix_log %*% beta_true_log
prob_y1_log <- exp(linear_predictor_log) / (1 + exp(linear_predictor_log))
Y_binary_log <- rbinom(N_log, 1, prob_y1_log)

#a
fLogisticLogLik <- function(vBeta, mX, vY) {
  dSum <- 0
  for (i in 1:length(vY)) {
    dSum <- dSum + (vY[i] * (mX %*% vBeta)[i] - log(1 + exp(mX %*% vBeta)[i]))
  }
  return(dSum)

  #  alternatively
  #return(-sum(vY * mX %*% vBeta - log(1 + exp(mX %*% vBeta))))
}

#b
fLogisticGradient <- function(vBeta, mX, vY) {
  dProb <- exp(mX %*% vBeta) / (1 + exp(mX %*% vBeta))
  dSum <- numeric(length(vBeta))
  for (i in 1:length(vY)) {
    dSum <- dSum + (vY[i] - dProb[i]) * mX[i, ]
  }
  return(dSum)

  # alternatively
  # return(t(mX) %*% (vY - dProb))
}

fLogisticHessian <- function(vBeta, mX, vY) {
  dProb <- exp(mX %*% vBeta) / (1 + exp(mX %*% vBeta))
  mHessianSum <- matrix(0, ncol(mX), ncol(mX))
  for (i in 1:length(vY)) {
    mHessianSum <- mHessianSum + dProb[i] * (1 - dProb[i]) * mX[i, ] %*% t(mX[i, ])
  }
  return(-mHessianSum)

  # alternatively
  # return(- t(mX) %*% diag(as.numeric(dProb * (1 - dProb))) %*% mX)
  # return(-crossprod(mX, mX * as.numeric(dProb * (1 - dProb))))
}

#c
fNewton <- function(f, fScore, fHessian, vY, mX, add.constant = TRUE, init.vals = NULL, max.i
  i <- 0
  vB <- init.vals

  # Keep updating until stopping criterion or max iterations reached
  while ((max(abs(fScore(vB, mX, vY))) > dTol) && (i < max.iter)) {
    # Newton-Raphson updating
    vB <- vB - solve(fHessian(vB, mX, vY), fScore(vB, mX, vY))
    i <- i + 1
    #cat("At iteration", n, "the value of the parameter is:", dParam, "\n")
  }

  if (i == max.iter) {
    return(list(
      beta_hat = NULL,
      log_likelihood_opt = NULL,
      score_opt = NULL,
      hessian_opt = NULL,
      log_likelihood_null = NULL,
      #predicted_probabilities = NULL,
      iterations = i,
      msg = "Algorithm failed to converge. Maximum iterations reached.")
    )
  } else {
    return(list(
      beta_hat = vB,
      log_likelihood_opt = f(vB, mX, vY),
      score_opt = fScore(vB, mX, vY),
      hessian_opt = fHessian(vB, mX, vY),
      log_likelihood_null = f(vB, mX, vY),
      #predicted_probabilities = exp(mX %*% vB) / (1 + exp(mX %*% vB)),
      iterations = i,
      msg = "Algorithm converged")
    )
  }
}

fNewton(fLogisticLogLik, fLogisticGradient, fLogisticHessian, vY = Y_binary_log, mX = X_matri
#> $beta_hat
#>             X1_log    X2_log
#> -0.313394  1.609719 -0.590626
#>
#> $log_likelihood_opt
#> [1] -100.5938
#>
#> $score_opt
#>                     X1_log        X2_log
#> -1.963759e-10  6.407941e-10 -1.425498e-10
#>
#> $hessian_opt
#>                     X1_log     X2_log
#> [1,] -33.374735  -4.528574 -16.508856
#> [2,]  -4.528574 -16.413238  -2.941622
#> [3,] -16.508856  -2.941622 -10.913058
#>
#> $log_likelihood_null
#> [1] -100.5938
#>
#> $iterations
#> [1] 5
```

```
#>
#> $msg
#> [1] "Algorithm converged"

#d
optim(c(0, 0, 0), fLogisticLogLik, gr = fLogisticGradient, mX = X_matrix_log, vY = Y_binary_l
#> $par
#> [1] -0.3133828  1.6097293 -0.5906087
#>
#> $value
#> [1] -100.5938
#>
#> $counts
#> function gradient
#>       15        8
#>
#> $convergence
#> [1] 0
#>
#> $message
#> NULL
```

```
double logistic_neg_log_lik_cpp(vec vBeta, mat mX, vec vY) {
  double dSum = 0.0;
  vec linear_term = mX * vBeta;
  int n = vY.size();
  for (int i = 0; i < n; i++) {
    dSum += (vY(i) * linear_term(i)) - log(1 + exp(linear_term(i)));
  }
  return(dSum);
}

f(vB, mX, vY)
```

```
suppressMessages(library(Rcpp))
suppressMessages(library(RcppArmadillo))
sourceCpp("likelihood_exercises.cpp")

fLogisticLogLik(c(-0.313394, 1.609719, -0.590626), X_matrix_log, Y_binary_log)
#> [1] -100.5938
logistic_neg_log_lik_cpp(c(-0.313394, 1.609719, -0.590626), X_matrix_log, Y_binary_log)
#> [1] -100.5938
```

48.2 (2) AR(1) Model with Student's t-errors (Extremely Hard)

-
-

-

-

-

```r
set.seed(800)
T_ar <- 300
phi_true <- 0.7
nu_true <- 5 # Degrees of freedom
# Simulate standardized t errors (variance = 1)
errors_t <- rt(T_ar, df = nu_true) * sqrt((nu_true - 2) / nu_true)
y_ar <- numeric(T_ar)
y_ar[1] <- errors_t[1] / sqrt(1 - phi_true^2) # Start from unconditional variance
for(t_idx in 2:T_ar) {
  y_ar[t_idx] <- phi_true * y_ar[t_idx-1] + errors_t[t_idx]
}
```

-
  - a)
  - b)
    - *
    - *
  - c)
  - d)

- e)
- f)
- g)
- h)

```
#data
set.seed(800)
T_ar <- 300
phi_true <- 0.7
nu_true <- 5 # Degrees of freedom
# Simulate standardized t errors (variance = 1)
errors_t <- rt(T_ar, df = nu_true) * sqrt((nu_true - 2) / nu_true)
y_ar <- numeric(T_ar)
y_ar[1] <- errors_t[1] / sqrt(1 - phi_true^2) # Start from unconditional variance
for(t_idx in 2:T_ar) {
  y_ar[t_idx] <- phi_true * y_ar[t_idx-1] + errors_t[t_idx]
}

#a + b
fAR1tNegLogLik <- function(vParams, vY) {
  dPhi <- tanh(vParams[1])
  dNu <- 2 + exp(vParams[2])
  vEps <- numeric(length(vY))

  dSum <- 0
  for (t in 2:length(vY)) {
    vEps[t] <- vY[t] - dPhi * vY[t-1]
    dSum <- dSum + lgamma((dNu+1)/2) - log(sqrt(pi * (dNu-2))) - lgamma(dNu/2) - (dNu + 1) /
  }
  return(-dSum)
}
```

```
#c
suppressMessages(library(numDeriv))
fAR1tGrad <- function(f, vParams, ...) {
  return(grad(f, vParams, ...))
}
fAR1tHess <- function(f, vParams, ...) {
  return(hessian(f, vParams, ...))
}

fAR1tGrad(fAR1tNegLogLik, c(0, log(3)), vY = y_ar)
#> [1] -353.44783  -12.69966
fAR1tHess(fAR1tNegLogLik, c(0, log(3)), vY = y_ar)
#>           [,1]     [,2]
#> [1,] 205.88262  3.80849
#> [2,]   3.80849 23.23789

#d
fNewton <- function(f, fScore, fHessian, init.vals = NULL, max.iter = 200, dTol = 1e-9, ...)
  i <- 0
  vPars <- init.vals

  while ((max(abs(fScore(f, vPars, ...))) > dTol) && (i < max.iter)) {
    vPars <- vPars - solve(fHessian(f, vPars, ...), fScore(f, vPars, ...))
    i <- i + 1
  }

  if (i == max.iter) {
    return(list(
      curr_params = vPars,
      curr_log_likelihood_opt = f(vPars, ...),
      curr_score_opt = fScore(f, vPars, ...),
      curr_hessian_opt = fHessian(f, vPars, ...),
      iterations = i,
      msg = "Algorithm failed to converge. Maximum iterations reached.")
    )
  } else {
    return(list(
      params = vPars,
      log_likelihood_opt = f(vPars, ...),
      score_opt = fScore(f, vPars, ...),
      hessian_opt = fHessian(f, vPars, ...),
      iterations = i,
      msg = "Algorithm converged")
    )
  }
}

fNewton(fAR1tNegLogLik, fAR1tGrad, fAR1tHess, vY = y_ar, init.vals = c(0.2, log(3)))
#> $params
#> [1] 0.8557711 1.1124668
#>
#> $log_likelihood_opt
#> [1] 410.3974
#>
#> $score_opt
#> [1] -2.601885e-10  5.501685e-10
#>
#> $hessian_opt
#>              [,1]       [,2]
#> [1,] 200.2957152 -0.5986651
#> [2,]  -0.5986651  9.6342894
#>
#> $iterations
#> [1] 92
#>
#> $msg
#> [1] "Algorithm converged"
vPars_temp <- fNewton(fAR1tNegLogLik, fAR1tGrad, fAR1tHess, vY = y_ar, init.vals = c(0.2, lo

dPhi_star <- tanh(vPars_temp[1])
dNu_star <- 2 + exp(vPars_temp[2])

print(paste0("Phi-est: ", dPhi_star, " vs true: ", phi_true))
#> [1] "Phi-est: 0.694072436111299 vs true: 0.7"
print(paste0("Nu-est: ", dNu_star, " vs true: ", nu_true))
#> [1] "Nu-est: 5.04185272789097 vs true: 5"

# e
vPars_temp <- optim(c(0.2, log(3)), fn = fAR1tNegLogLik, gr = function(par, ...) { fAR1tGrad

# f
dPhi_star <- tanh(vPars_temp[1])
dNu_star <- 2 + exp(vPars_temp[2])
```

```r
print(paste0("Phi-est: ", dPhi_star, " vs true: ", phi_true))
#> [1] "Phi-est: 1 vs true: 0.7"
print(paste0("Nu-est: ", dNu_star, " vs true: ", nu_true))
#> [1] "Nu-est: 6.97158263449599 vs true: 5"
```

48.3 (3) Zero-Inflated Poisson (ZIP) Regression Likelihood (Extremely Hard)

-
    1.
    2.
-
-
    -
    -
-
-

```r
set.seed(999)
N_zip <- 250
W1_zip <- rnorm(N_zip); W_matrix_zip <- cbind(1, W1_zip) # For pi
X1_zip <- runif(N_zip); X_matrix_zip_lambda <- cbind(1, X1_zip) # For lambda

gamma_true_zip <- c(-1, 0.8) # logit(pi) params
beta_true_zip <- c(0.5, 1.5)   # log(lambda) params

logit_pi_vals <- W_matrix_zip %*% gamma_true_zip
pi_vals <- exp(logit_pi_vals) / (1 + exp(logit_pi_vals))

log_lambda_vals <- X_matrix_zip_lambda %*% beta_true_zip
lambda_vals <- exp(log_lambda_vals)

Y_zip <- numeric(N_zip)
for(i_zip in 1:N_zip) {
  if (runif(1) < pi_vals[i_zip]) {
    Y_zip[i_zip] <- 0 # Certain zero
  } else {
    Y_zip[i_zip] <- rpois(1, lambda_vals[i_zip]) # From Poisson
  }
}
# hist(Y_zip, breaks=-0.5:(max(Y_zip)+0.5)) # Lots of zeros?
```

-
    - a)
    - b)
    - c)
    - d)
    - e)
    - f)

```r
#data
set.seed(999)
N_zip <- 250
W1_zip <- rnorm(N_zip); W_matrix_zip <- cbind(1, W1_zip) # For pi
X1_zip <- runif(N_zip); X_matrix_zip_lambda <- cbind(1, X1_zip) # For lambda

gamma_true_zip <- c(-1, 0.8) # logit(pi) params
beta_true_zip <- c(0.5, 1.5)   # log(lambda) params

logit_pi_vals <- W_matrix_zip %*% gamma_true_zip
pi_vals <- exp(logit_pi_vals) / (1 + exp(logit_pi_vals))
```

```
log_lambda_vals <- X_matrix_zip_lambda %*% beta_true_zip
lambda_vals <- exp(log_lambda_vals)

Y_zip <- numeric(N_zip)
for(i_zip in 1:N_zip) {
  if (runif(1) < pi_vals[i_zip]) {
    Y_zip[i_zip] <- 0 # Certain zero
  } else {
    Y_zip[i_zip] <- rpois(1, lambda_vals[i_zip]) # From Poisson
  }
}

#a
fZIPNegLogLik <- function(vParams, vY, mW, mX) {
  vGamma <- vParams[1:ncol(mW)]
  vBeta <- vParams[(ncol(mW) + 1):(ncol(mW) + ncol(mX))]

  logit_pi_i <- mW %*% vGamma
  pi_i <- plogis(logit_pi_i)
  log_lambda_i <- mX %*% vBeta
  lambda_i <- exp(log_lambda_i)

  dSum <- 0
  for (i in 1:length(vY)) {
    if (vY[i] == 0) {
      dSum <- dSum + log(pi_i[i] + (1 - pi_i[i]) * ((exp(-lambda_i[i]) * lambda_i[i]^vY[i]) /
    } else {
      dSum <- dSum + log((1 - pi_i[i]) * ((exp(-lambda_i[i]) * lambda_i[i]^vY[i]) / factorial
    }
  }
  return(-dSum)
}

optim(c(0, 0, 0, 0), fZIPNegLogLik, vY = Y_zip, mW = W_matrix_zip, mX = X_matrix_zip_lambda,
#> $par
#> [1] -1.0449956  0.6241883  0.5144958  1.5029647
#>
#> $value
#> [1] 490.5832
#>
#> $counts
#> function gradient
#>       26       10
#>
#> $convergence
#> [1] 0
#>
#> $message
#> NULL
```