This lab contains optional sections. If you want to finish all sections, you will most likely exceed 2 hours. If you can’t finish the log-book questions in lab, proceed at home. If you have questions ask your tutor or come to the consultation hours. Be reminded that logbook questions are an assessment task and that future assignments will rely on lab knowledge.

1 Some Transformations

First we generate a vector “U” containing a sample of \( n = 100 \) pseudo-random observations from a Uniform distribution on \([0,1]\) using \texttt{runif()} Verify that the simulated sample is similar to what you would expect.

\begin{verbatim}
> n<-1e3
> U<-runif(n)
> mean(U) #should be about 0.5=E(Y)
> var(U) #should be around 1/12=Var(Y) approx 0.08333333
> hist(U) #should have approx equal frequencies
> plot(density(U)) #obtain approx. density (estimated density)
\end{verbatim}

Now we carry out various transformations and plot the transformed pseudo-observations.

\begin{verbatim}
> U1<-U^3
> U2<-U^0.5
> U3<-U^(1/3)
> U4<- -log(U)
> U5<-U^(-0.5)
> U6<-1/U
> U7<-1/U^3
> hist(U1,breaks=20)
> hist(U2,breaks=20)
> hist(U3,breaks=20)
\end{verbatim}
2 Integration in R

The R-function `f, lower, upper, ...` can be used for numerical integration of function `f`. Note `f(x)` needs to return a vector if `x` is a vector, in the sense that `f(x) = (f(x_1), ..., f(x_n))`, if `x = (x_1, ..., x_n).

2.1 Single Integration

```r
> #gives the mean of uniform
> integrate(f=function(x){x*1}, lower=0, upper=1)
0.5 with absolute error < 5.6e-15

> #alternatively using dunif-function
> f1<-function(x){x*dunif(x,min=0,max=1)}
> integrate(f=f1, lower=0, upper=1)
0.5 with absolute error < 5.6e-15

> #gives the variance
> integrate(f=function(x){(x-0.5)^2*1}, lower=0, upper=1)
0.08333333 with absolute error < 9.3e-16

> #alternatively using dunif-function
> f2<-function(x){(x-0.5)^2*dunif(x,min=0,max=1)}
> integrate(f=f2, lower=0, upper=1)
0.08333333 with absolute error < 9.3e-16
```

Other examples
You need to check carefully whether the defined function returns a vector if applied to a vector.

2.2 $n$-fold Integration (relevant for later Assignments)

Consider now double integrals of Function $g(x, y) = \tan(x + y)$ for $-0.5 \leq x \leq y$ and $0 \leq y \leq 0.5$

Now consider another two-dimensional function $f(y_1, y_2) = e^{-y_1}$ for $0 \leq y_2 \leq y_1 < \infty$, zero o.w. We also calculate a double integral, however we define functions implicitly, i.e. without defining functions explicitly outside of \texttt{integrate}-function.

This refers to Tutorial Week 9 - Question 5b, where we calculate $P(Y_1 \geq 2Y_2)$, i.e.

$$
\int_0^\infty \int_{2y_2}^\infty e^{-y_1} \, dy_1 \, dy_2
$$

In R this can be achieved by
You see the inner \texttt{integrate()} refers to the inner integration (over \(y_1\) inside \(2y_2 \leq y_1 \leq \infty\)) and the outer \texttt{integrate()} to the outer integration (over \(y_2\) inside \(0 \leq y_2 \leq \infty\)).

As an alternative, you can use the R function \texttt{adaptIntegrate} to compute \(n\)-fold integrals. However this can only be used when the bounds are fixed numbers. For example, find out the integral

\[
\int_0^{1/2} \int_0^{1/2} \int_0^{1/2} \frac{2}{3}(x_1 + x_2 + x_3) dx_1 dx_2 dx_3
\]

\begin{verbatim}
> library(cubature)  # load the package "cubature"
> nfold.f <- function(x) {
+}  # "x" is a vector
> adaptIntegrate(nfold.f, lowerLimit = c(0,0,0),
+                 upperLimit = c(0.5,0.5,0.5))

$integral
 [1] 0.0625

$error
 [1] 1.387779e-17

$functionEvaluations
 [1] 33

$returnCode
 [1] 0
\end{verbatim}

3 Optimisation

In R you can use \texttt{optim()} (multi-variate parameter problems) and \texttt{optimize()} (one-dimensional parameter problems) to optimise functions. The default option is to minimise.

\begin{verbatim}
> optim(par, fn, gr = NULL, ...,  
+       method = c("Nelder-Mead", "BFGS", "CG", "L-BFGS-B", "SANN",  
+                    "Brent"),  
+       lower = -Inf, upper = Inf,  
+       control = list(), hessian = FALSE)
\end{verbatim}
Some examples

```r
> f <- function (x, a){return((x - a)^2)}
> xmin <- optimize(f, c(0, 1), tol = 0.0001, a = 1/3)
> xmin #a list

$minimum
[1] 0.3333333

$objective
[1] 0

> xmin$minimum #a number
[1] 0.3333333

> xmin$objective #a number
[1] 0
```

`optimize()` returns a list and its first element is `minimum` (the argument that minimises the function) and its second argument is `objective` which is the value of the function at its minimum.

See `help(optimize)` to see more information. The elements of that list are explained under `help(optim)` and under `Value`. There all the elements are listed and explanation is given.

To recall, to access the first argument of a list, we would enter `xmin[[1]]` or `xmin$minimum` and to access the second element we would enter `xmin[[2]]` or `xmin$objective`.

Now we define the Rosenbrock function.

```r
> require(graphics)
> #--------------------------------
> # Rosenbrock Banana function
> #--------------------------------
> # x is a vector with two elements, x[1] and x[2]
```
3.1 Visualisation of Rosenberg Banana-function (Optional)

To see the “Rosenbrock Banana function” that we want to optimised we plot it. The 3d plotting is only for visualisation purposes. The minimum of the Rosenbrock Banana function implemented in \( f_r \) is \((x_1, x_2) = (1, 1)\).

```r
> #3d surface plot of the function to be minimised
> x <- seq(-2, 3, .1)
> y <- seq(-2, 5, .1)
> ly<-length(y);lx<-length(x)
> #create z
> z<-matrix(0,lx,ly)
> for(i in 1:lx){
+ for(j in 1:ly){
+   z[i,j]<-fr(c(x[i],y[j]))
+ }#end for j
+ }#end for i
> #Contour plot package graphics
> require(graphics)
> image(x, y, z,col= heat.colors(20))
> contour(x=x,y=y,z=z,col = "red")
> #now add contour to image
> image(x, y, z,col= heat.colors(20))
> contour(x,y,z=z,col = "black",add=TRUE)
```

```r
> # Plots package lattice
> xyz <- data.frame(matrix(0, length(x)*length(y), 3))
> names(xyz) <- c('x', 'y', 'z')
> n <- 0
> for (i in 1:length(x)) {
```
Figure 1: Left: Contourplot, Right: Image

```r
+ for (j in 1:length(y)) {
+   n <- n + 1
+   xyz[n,] <- c(x[i], y[j], fr(c(x[i], y[j]))[[1]])
+ }
+ }
> library(lattice)
> contourplot(z ~ x*y, data = xyz, xlab="x", ylab="y")

> wireframe(z ~ x*y, data = xyz, scales = list(arrows = FALSE),
+          zlab = 'f(x, y)', drape = T)
> # Now Rotate plot: use screen option to rotate x, y, z axes
> # specified number of degrees from the default rotation.
> wireframe(z ~ x*y, data = xyz, scales = list(arrows = FALSE),
+          zlab = 'f(x, y)', drape = T, screen=list(x=-60))
> wireframe(z ~ x*y, data = xyz, scales = list(arrows = FALSE),
+          zlab = 'f(x, y)', drape = T, screen=list(x=-60, y=70))
> wireframe(z ~ x*y, data = xyz, scales = list(arrows = FALSE),
+          zlab = 'f(x, y)', drape = T, screen=list(x=10, y=-135, z=-40))
```
Figure 2: Left: Added contour-plot to image, Right: contourplot using lattice package

Figure 3: Wireframe-plotting using different angles
3.2 Optimisation of Rosenberg Banana-function

The Rosenberg Banana-function is two-parameter function, so the minimum is achieved at a particular \( x = (x_1, x_2) \). The first argument of `optim` is the starting value (a guess that might be close to the true minimum), the 2nd argument is `fn`, the function to be minimised, and the third argument is `gr`, the gradient.

Optimising a function using the 1st partial derivatives (gradient), here `grr`, results in faster optimisation.

```r
> #gradient of function fr, i.e. 1st derivative
> grr <- function(x) {  ## Gradient of 'fr'
+  x1 <- x[1]
+  x2 <- x[2]
+  c(-400 * x1 * (x2 - x1 * x1) - 2 * (1 - x1),
+      200 * (x2 - x1 * x1))
+ }
> optim(c(-1.2,1), fn=fr) #without derivative
> res <- optim(c(-1.2,1), fn=fr, gr=grr, method = "BFGS") #faster
> optimHess(res$par, fr, grr) #Hessian approximation
> optim(c(-1.2,1), fr, NULL, method = "BFGS", hessian = TRUE)
```

As for `optimize()`, the function `optim()` return a list. The elements of that list are explained under `help(optim)` and under `Value`.

3.3 Himmelblau Function (optional)

Himmelblau’s function is a multi-modal function, used to test the performance of optimisation algorithms. The function is defined by:

\[
f(x, y) = (x^2 + y - 11)^2 + (x + y^2 - 7)^2
\]

This function is strictly positive. The function is plotted below:

```r
> Himmelblau.f <- function(x1,y1) {
+  (x1^2 + y1 -11)^2 + (x1 + y1^2 -7)^2
+ }
> x <- seq(-4.5,4.5,by=0.2)
> y <- seq(-4.5,4.5,by=0.2)
> z <- outer(x,y,Himmelblau.f)
> persp(x,y,z, phi=-45, theta=45, shade=0.65,
+       col="yellow", ticktype="detailed")
```
There appear to be four "bumps" that look like minima in the realm of \((-4,-4)\), \((2,-2)\), \((2,2)\) and \((-4,4)\). So the function is minimised when \(x^2 + y - 11 = 0\) and \(x + y^2 - 7 = 0\) simultaneously.

```
> Himmelblau.f <- function(x) # "x" is a vector
+   { 
+   }
> optim(c(-4,-4), Himmelblau.f)$par
> optim(c(2,-2), Himmelblau.f)$par
[1] 3.584370 -1.848105
> optim(c(2,2), Himmelblau.f)$par
[1] 3.000014 2.000032
> optim(c(-4,4), Himmelblau.f)$par
[1] -2.805129 3.131435
```

As we see there are indeed four true minima near, one each near the 4 starting values.
This can be checked by seeing that these inputs correspond to function values that are about 0.

4 Travelling salesman problem (optional)

Here some application to the famous "Travelling salesman problem", see for example http://en.wikipedia.org/wiki/Travelling_salesman_problem. In principle a salesman has to visit each of \( n \) cities once and has to return to the starting city. The aim is to travel the shortest possible distance knowing all distances between all cities.

```r
> #################################################################
> ## Combinatorial optimization: Traveling salesman problem
> #################################################################
> library(stats) # normally loaded
> # obtain distances between European cities
> eurodistmat <- as.matrix(eurodist)
> distance <- function(sq) { # Target function
+   sq2 <- embed(sq, 2)
+   sum(eurodistmat[cbind(sq2[,2], sq2[,1])])
+ }
> genseq <- function(sq) { # Generate new candidate sequence
+   idx <- seq(2, NROW(eurodistmat)-1)
+   changepoints <- sample(idx, size = 2, replace = FALSE)
+   tmp <- sq[changepoints[1]]
+   sq[changepoints[1]] <- sq[changepoints[2]]
+   sq[changepoints[2]] <- tmp
+   sq
+ }
> sq <- c(1:nrow(eurodistmat), 1) # Initial sequence: alphabetic
> distance(sq)
> # rotate for conventional orientation
> loc <- -cmdscale(eurodist, add = TRUE)$points
> x <- loc[,1]; y <- loc[,2]
> s <- seq_len(nrow(eurodistmat))
> tspinit <- loc[sq,]
> plot(x, y, type = "n", asp = 1, xlab = "", ylab = "",
+     main = "initial solution of traveling salesman problem", axes = FALSE)
> arrows(tspinit[s,1], tspinit[s,2], tspinit[s+1,1], tspinit[s+1,2],
+     angle = 10, col = "green")
> text(x, y, labels(eurodist), cex = 0.8)
> set.seed(123) # chosen to get a good soln relatively quickly
> res <- optim(sq, distance, genseq, method = "SANN",
```
Figure 4: Initial (left) and final (right) solutions to "Travelling Salesman Problem"
5 Approximation of Population Mean

5.1 Sample Mean

Let’s take a r.v. $Y \sim U[0,1]$ with $E(Y) = 1/2$ and $Var(Y) = 1/12$. We simulate random numbers $Y_1, \ldots, Y_n$ from $U[0,1]$ and use the sample mean $\bar{Y}_n = \frac{1}{n} \sum_{i=1}^{n} Y_i$ to approximate $E(Y)$. Finally we plot $\bar{Y}_n$ versus $n$. For each $n$, we obtain a different sample.

```r
> n.sequence<-c(1:1e3)
> N<-length(n.sequence)
> Y<-list(0,N)
> Ybar<-rep(0,N)
> for(n in 1:N){
+ Y[[n]] <- runif(n)
+ Ybar[n]<-mean(Y[[n]])
+ }#end for n
> matplot(x=n.sequence,y=Ybar,xlab="n",ylab=expression(bar(y)),
+ main="Sample Mean versus n for uniform distribution U[0,1]",
+ col="red",type="l")
> #legend, need to specify name, line-type and colour to match with those in plot
> legend("topright",c("U[0,1]"),lty=c(1),col=c("red"))
```

5.2 Confidence Intervals (optional)

There are some commands in R to find confidence intervals for the mean for normally distributed data with unknown variance. To illustrate the use of these commands, experimental data for the lizard tail length is provided below:

```r
> lizard <- c(6.2, 6.6, 7.1, 7.4, 7.6, 7.9, 8, 8.3, 8.4, 8.5,
+ 8.6, 8.8, 8.8, 9.1, 9.2, 9.4, 9.4, 9.7, 9.9,
+ 8.6, 8.8, 8.8, 9.1, 9.2, 9.4, 9.4, 9.7, 9.9,
```
Here is a way to see how confidence intervals are random. Based on the lizard data, we draw 100 random samples with mean 9 and SD the same as the lizards.

```
> n.draw <- 100
> mu <- 9
> n <- length(lizard)
> SD <- sd(lizard)
> draws <- matrix(rnorm(n.draw*n, mu, SD), n)
> dim(draws)
[1] 24 100
```

Now R constructs 95% confidence intervals (CI) for each random sample.

The first line creates a local command `get.conf.int` to extract the confidence interval (conf.int) from the `t.test` command. The second line uses the `apply` command to apply `get.conf.int` to every column of draws. Finally, we count the number of confidence intervals that cover $\mu = 9$.

This confidence interval (discussed e.g. in STAT232 and probably formulae provided in every statistics service subject) has the form

$$\bar{X} \pm \frac{t_{n-1,\alpha/2} s}{\sqrt{n}}$$

where $t_{n-1,\beta}$ is the $\beta$th quantile of the so-called t-distribution with $n - 1$ degrees of freedom, $s$ is the sample standard deviation, i.e. $\sqrt{s^2}$, $\bar{X}$ the sample mean, and $n$ the sample size.

```
> get.conf.int = function(x) {
+   t.test(x)$conf.int
+ }
> conf.int <- apply(draws, 2, get.conf.int)
> #counts number of CI's that contain mu=9
> sum(conf.int[1,] <= mu & conf.int[2,] >= mu)
[1] 95
```
Here is a figure showing the 100 confidence intervals as horizontal lines, with a vertical line at the population mean of 9.

```r
> plot(range(conf.int), c(0, 1 + n.draw), type = "n",
+       xlab = "mean tail length", ylab = "Sample run")
> for (i in 1:n.draw) {
+   lines(conf.int[, i], rep(i, 2), lwd = 2)
+ }
> abline(v = 9, lwd = 2, lty = 2, col="red")
```

6 The Chebyshev’s Inequality (optional)

Chebyshev’s inequality states that for any r.v. (provided variance exists) and any positive constant $c$

$$P \{ |X - \mu| \geq d \} \leq \frac{\sigma^2}{d^2}$$

Essentially it states that in any probability distribution, ”nearly all” values are close to the mean.

An illustration of the Chebyshev’s inequality can be implemented by R simulation process. We consider various values of $d$ (denoted by `d`) and $\sigma^2$ (denoted by `sig`), the variance of the normal distribution.
# The Chebyshev relation by means of simulation

```r
mu <- 0
N <- 1e3
sig.values <- seq(.05, 1, by = .05) # values of variance
d.values <- seq(.05, 1, by = .05) # values of d
m <- length(sig.values)
n <- length(d.values)
Bound <- P.hat <- matrix(NA, m, n)
for (i in 1:m) {
  for (j in 1:n) {
    # random sampling of the r.v. z
    z <- rnorm(N, mean = mu, sd = sig.values[i])
    # rel. frequency (proportion) of the event: z-mu >= d
    P.hat[i, j] <- sum(abs(z - mu) >= d.values[j]) / N
    # upper bound of the Chebyshev relation
    # probability is always <=1, therefore bounds>1 discarded
    Bound[i, j] <- min(1, (sig.values[i]^2) / (d.values[j]^2))
  }
}
```

## PLOTTING

```r
library(fields) # without it, image.plot() doesn't work
color <- heat.colors(12) # choose colour scheme
# here we divide plot into 3 subplots using panel()
# 1 for P.hat, 1 for Bound and 1 for legend
set.panel()
nf <- layout(matrix(c(1, 2, 3), 1, 3, byrow = TRUE), widths = c(4, 4, 1), heights = c(4))
# Two plots, one plot of P.hat and one of Bound
# first image, than on top of it contourplot
image(P.hat, col = color, xlab = expression(sigma^2), ylab = expression(d),
      xlim = c(0, 1), ylim = c(0, 1), main = "P.hat vs. d and sig", zlim = c(0, 1),
      cex.main = 1, cex.axis = 0.8, cex.lab = 0.8)
contour(P.hat, levels = seq(0.1, 1, by = 0.1), add = TRUE, col = "green")
# first image, than on top of it contourplot
image(Bound, col = color, xlab = expression(sigma^2), ylab = expression(d),
      xlim = c(0, 1), ylim = c(0, 1), main = "Bound vs. d and sig", zlim = c(0, 1),
      cex.main = 1, cex.axis = 0.8, cex.lab = 0.8)
contour(Bound, levels = seq(0.1, 1, by = 0.1), add = TRUE, col = "green")
# now add finally legend
plot.new()
image.plot(zlim = c(0, 1), legend.only = TRUE, col = color,
```
Figure 5: True $P\{|X - \mu| \geq d\}$ (left) and Bound $\sigma^2 / d^2$ (right) versus $\sigma^2$ and $d$