PS8 - STAT 243

Alexander Fred Ojala
Student ID: 26958060
Collaborators: Milos Atz, Guillame Baquiast and Alexander Brandt

December 4th 2015

1 Problem 1

1.1 a)

We are comparing two models, namely the Standard Linear Regression Model and a new method that is supposed to be more robust against outliers.

We compare these two models in regards to the absolute prediction error and in the coverage of the prediction intervals for new observations (where the prediction performance of the two methods can be evaluated several times by nonparametric bootstrap).

If we first consider an initial data set with the independent values, X_0 (that is constructed in a way that it has significant outliers), these are the dependent variables in the two regression models: $y_0^{newmethod}$ and y_0^{LS}. We also have some random term for the regressions. The dimension of the independent values matrix is $[2n \times b]$, where $2n$ is the total number of observations (or generated values) for each covariate and b is the number of predictors / covariates (including eventual intercept). Therefore both y_0 will be of dimension $2n \times 1$. This could be any random data set that suits our needs for this study (hence we need to know that the data contains outliers).

The coverage of the prediction interval and the absolute predictive error can be computed: e.g. with the R function *predict* on the two models as well as that we split up our data into two parts, e.g. $X_{model} = (X_1, X_2, ...X_n)$ and $X_{confirmation} = (X_{n+1}, ...X_{2n})$ (the same goes for the dependent variables). So one part of the data set is used to fit the model and form the prediction while the other one is used to evaluate/-validate the performance of the predicition (hence calculating the absolute predicitive error and predictive interval coverage).

To compare the two models and how robust they are against outliers, we would like to repeat this process m times (e.g. m can be in the thousands, if the new method is easily evaluated, while $2n$ can be in the ten thousands for example). Through this repeated process we can calculate the absolute predicitive error and predictive interval coverage for the two methods several times and store the results in a table (of m rows). Since we know that the data set contains outliers we can see how well the model performs when this task is iterated by the bootstrap method with resampling. The bootstrapping step is sampling by resampling from the original data with the outliers (that we have generated). This is the non-parametric bootstrap where we do not evaluate an underlying function or estimates of parameters. Instead we draw $2n$ random samples from the original data and evaluate the statisics over and over again, m times, in order to get an estimate of which methods performs the best.

We iterate m times (and store the results in a table) over the following steps:

1. Resample with nonparametric bootstrap (ranomdly) from the intial data set and split the resampled data set into two parts (e.g. half the size). One used to fit the regression models as well as form the

prediction intervals and the other half of the resampled data set will be used to evaulate the absolute predictive error and coverage of the prediction interval (to see what method performs best in regards to outliers).

2. Calculate the absolute prediction error $\frac{1}{n}\sum_{i=1}^{n}|f_i - y_i|$, where f_i here is the prediction values and y_i is the the values we compare against (can be seen as new observations).

3. Then we also check the proportion of the prediciton values y_i that fall within the prediciton interval (e.g. the 95% prediction interval). The prediction intervals are computed using the trained regression models and parameter estimates obtained for the model. The coverage, i.e. the proportion, is calculated by looking at how many out of the total prediction values that lie within these two bounds.

After the methods are compared by estimating the mean of the absolute predicitive error for both methods as well as the mean of the prediction intervals (with the mean function or Monte Carlo methods). The method that performs best is the one that has the highest proportion of coverage in the prediction interval as well as the lowest absolute predictive error. This will also be the method that is most robust against outliers. (We can also lower the varaince of this result by increasing $2n$ as well as m)

1.2 b)

A specific example:
To construct the intial data set X_0 and Y_0 with outliers is the tricky part, otherwise we can follow the algorithm of Problem 1 a). Such a data set could be a data set with a clear linear trend, but with some significant outliers. If we introduce two error terms, namely:
ϵ_1 that is a regular error term , not affecting the data that much e.g. it could be drawn from the distribution $\mathcal{N}(0, 1)$
ϵ_2 is a term used to obtain the outliers (generate them). We could have an indicator function that is one whenever we draw a random uniform number that is greater than 0.95. If so than we will include the term ϵ_2 for that specific row, that affects a specific data point in our regression models. E.g. it could be drawn from the distribution $\mathcal{N}(0, 20)$. The X values (all the covariates, except for the intercept) could be drawn from $U[0, 5]$ and then we have:

$$Y = \beta_0 + \beta X + \epsilon_1 + \epsilon_2$$

For all the variables in the equation above we have $2n$ values (rows) of Y, X and both error terms. When we initialize our data set we can choose the betas so that the data set shows a clear linear trend (and the number of parameters, β, equals to the number of different covariates including the intercept).

E.g. $\beta_0 = 1.3$, $\beta = (1.2, 0.4, 0.95)$ (if we have four covariates in total incl. the intercept).

On this data set of length $2n$ we can run the algorithm provide in 1 a). We can choose $2n = 10000$ and $m = 500$, then we have completely decided the numerical example. With all this information we we can calculate the absolute prediction error as well as the coverage of the prediciton interval by repeating the process several times to form a statistical estimate using non-parametric bootstrap.

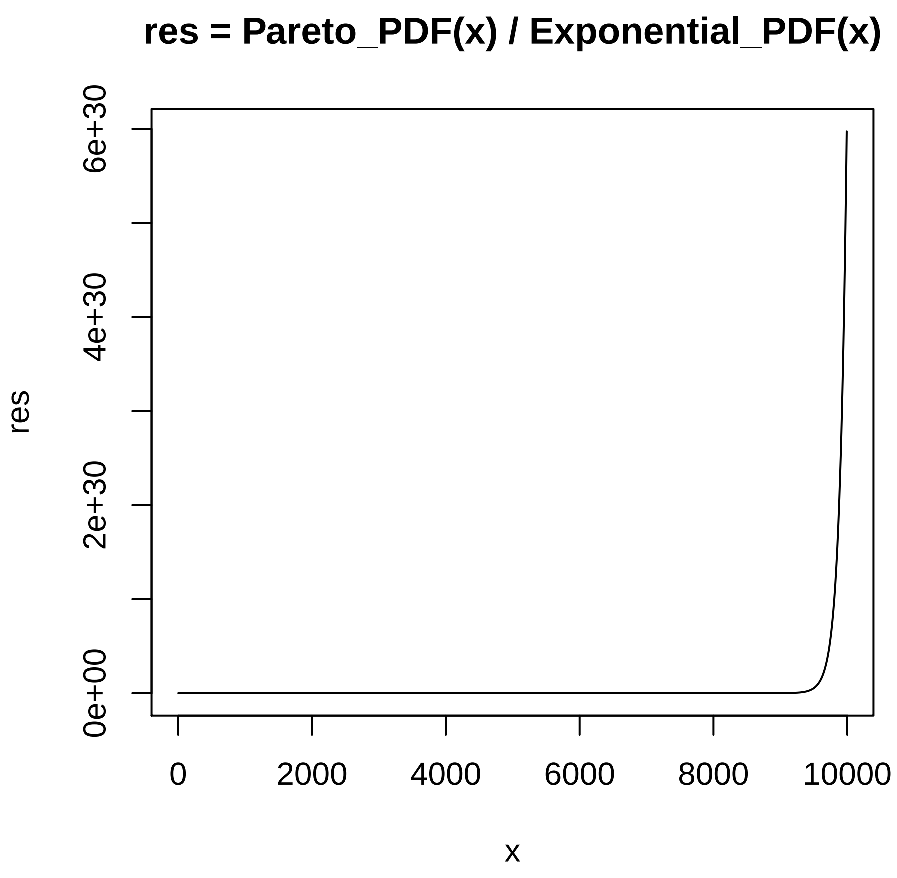
2 Problem 2

2.1 a)

The tail of the Parteo distribution decays slower than the exponential distribution. See handwritten mathematical proof. Below it is also confirmed by a numerical example.

N.B. A shift in the exponential distribution does not affect the result (it only adds a constant that is numerically insignificant in this context)

```r
#Define the pareto pdf
pareto_pdf <- function(x,alpha,beta) {
  return(beta*alpha^(beta)/(x^(beta+1)))
}

#Define exponential pdf
exponential_pdf <- function(x,lambda) {
  return(lambda*exp(-lambda*x))
}

alpha<-2
beta<-3

x<-seq(2,10000,10)

res<-pareto_pdf(x,alpha,beta)/exponential_pdf(x,.01)
plot(x,res,type="l",main="res = Pareto_PDF(x) / Exponential_PDF(x)")
```



3 b)

To analyze the importance sampling we import the package *PtProcesses* that defines the Pareto distribution in R. We also calculate the analytical values for the mean, here we want to estimate the mean of a random

variable from the shifted exponential distribution, hence:
$\mathbb{E}[X] = \frac{1}{\lambda} + x_{shift} = 3$

The variance is $\mathbb{V}[X] = \frac{1}{\lambda^2}$ so,

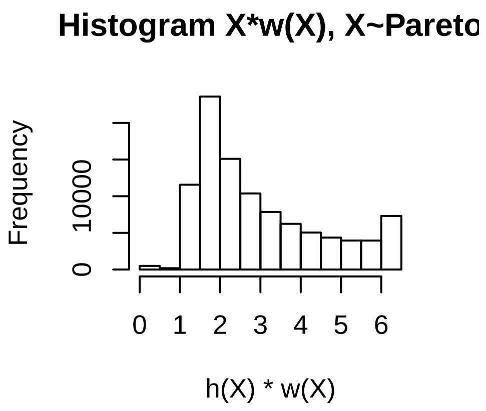
$\mathbb{E}[X^2] = \mathbb{V}[X] + (\mathbb{E}[X])^2 = 10.$

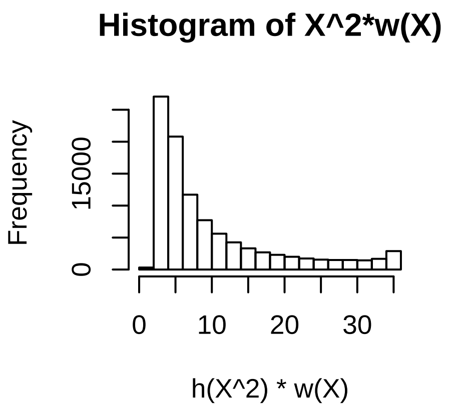
Find below the estimation using Monte Carlo methods:

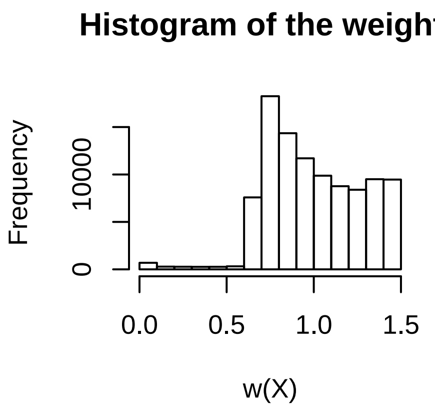
```
library(PtProcess) #built in Pareto dist. dpareto, rpareto to draw from the pareto

alpha<-2 #scale=alpha #a
beta<-3 #shape=beta #lambda in rpareto (first one)

m=100000
X<-rpareto(m,beta,alpha)

# Define f, g, h, and the weight
f<-function(X) dexp(X-2,1) #shifted exponential
g<-function(X) dpareto(X,beta,alpha) #pareto dist
h<-function(X) X
w<-function(X) f(X)/g(X)

#Estimate E[X]
1/m*sum(h(X)*w(X))

## [1] 2.997876

#Estimate E[X^2]
1/m*sum(h(X^2)*w(X))

## [1] 10.00071
```

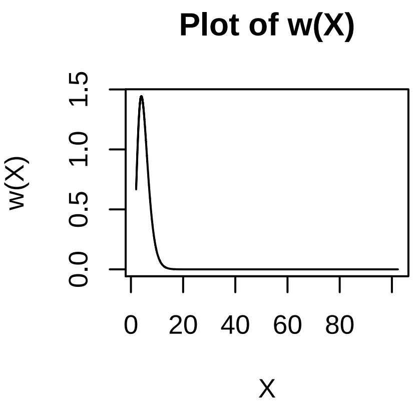
As can be seen from the results above the Monte Carlo estimate of the mean and $\mathbb{E}[X^2]$ is close to their theoretical values 3 and 10. This is also confirmed that it should be the case if we examine the histograms, the maximum weight and the variance calculated below. It shows that the estimates should be (somewhat) precise, which also fits well with the theory (because we use a sampling distribution with heavier tails).

```
#Histograms
par(mfrow=c(2,2))
hist(h(X)*w(X),main="Histogram X*w(X), X~Pareto")
hist(h(X^2)*w(X),main="Histogram of X^2*w(X)")
hist(w(X),main="Histogram of the weights")
plot(sort(X),w(sort(X)),main="Plot of w(X)",type="l",xlab="X",ylab="w(X)")
```









```
#numerical

max(w(X)) #There are no extreme weights

## [1] 1.443576

var(h(X)*w(X)) #variance of mean estimate

## [1] 2.385006

var(h(X^2)*w(X)) #variance of E(X^2) estimate

## [1] 75.70351
```

3.1 c)

Since X now is drawn from the shifted exponential distribtuion we will try to estimate the analytical mean and $\mathbb{E}[X^2]$ from the Pareto distribution. The theoretical formula for the theoretical values was given in the problem description (also calculated below), and they are:

$\mathbb{E}[X] = \frac{\beta\alpha}{(\beta-1)} = 3$

And the variance is $\mathbb{V}[X] = \frac{\beta\alpha^2}{(\beta-1)^2(\beta-2))}$=3, so,

$\mathbb{E}[X^2] = \mathbb{V}[X] + (\mathbb{E}[X])^2 = 12$.
With $\alpha = 2$ and $\beta = 3$.

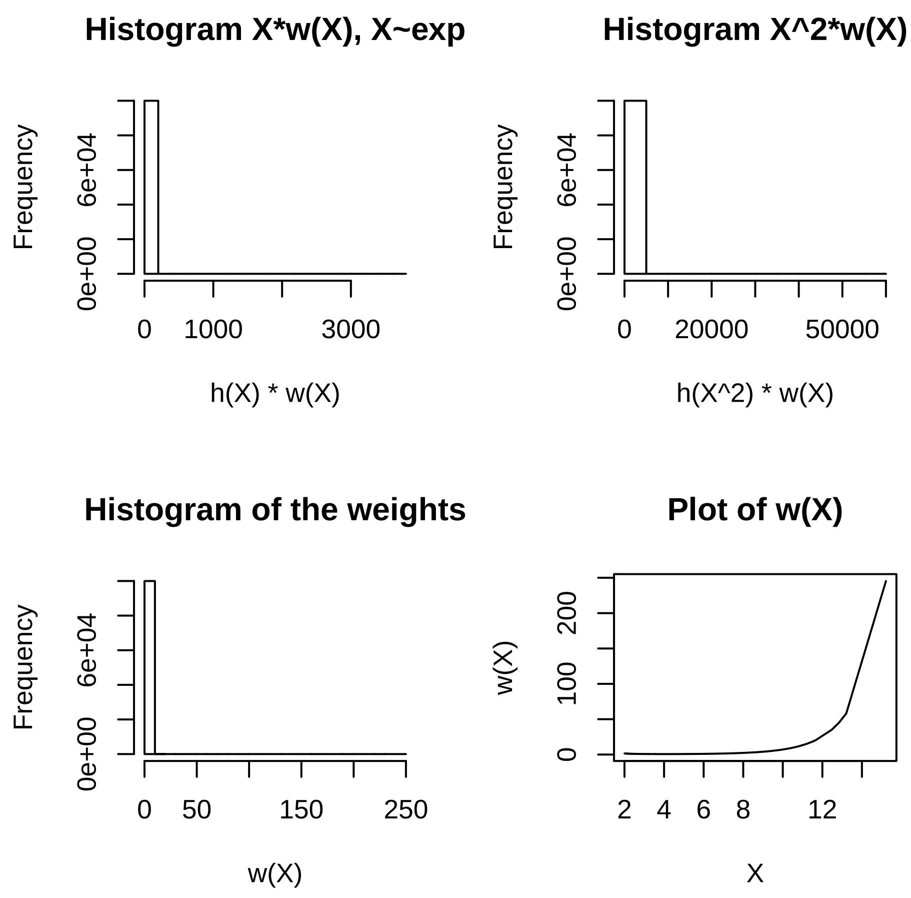
Find below the implementation of the algorithm to form the estimates:

```
alpha<-2 #scale=alpha #a
beta<-3 #shape=beta #lambda in rpareto (first one)

print(theoMean<-beta*alpha/(beta-1)) #Real value 3

## [1] 3

theoVar<-beta*alpha^2/((beta-1)^2*(beta-2)) #Real value 3
print(theoEXSqare<-theoVar+(theoMean)^2) #Real value 12

## [1] 12

X<-rexp(m,1)+2 #draw from shifted exponential distribution

fnew<-g #change f to pareto dist as above
gnew<-f #change g to shifted exponential
h<-function(X) X

w<-function(X) fnew(X)/gnew(X)

#Estimate E[X]
1/m*sum(h(X)*w(X))

## [1] 2.965875

#Estimate E[X^2]
1/m*sum(h(X^2)*w(X))

## [1] 10.73563
```

The result for the mean is kind of close to the theoretical value 3. However the estimate of $\mathbb{E}[X^2]$ has high variance and is too small - during all the test runs it never reached the theoretical value 12. This is because the tail of the sampling distribution does not cover the distribution that we want to estimate over. Therefore we have almost no probability to draw values that still are probable outcomes for the distribution of interest. This notion is confirmed when we plot the histograms, calculate the maximum weight and the variance below. It shows that the estimates vary a lot, which also fits well with the theory (because we use a sampling distribution with tails that decay faster).

```
#Histograms
par(mfrow=c(2,2))
hist(h(X)*w(X),main="Histogram X*w(X), X~exp")
hist(h(X^2)*w(X),main="Histogram X^2*w(X)") #high variance
hist(w(X),main="Histogram of the weights")

max(w(X)) #Here the weights are much bigger

## [1] 245.3828
```

```r
var(h(X)*w(X)) #mean estimate var

## [1] 157.4739

var(h(X^2)*w(X)) #E(X^2) estimate var, very Big!

## [1] 34904.91

plot(sort(X),w(sort(X)),main="Plot of w(X)",type="l",xlab="X",ylab="w(X)")
```



4 Problem 3

4.1 a)

See handwritten solution where the EM algorithm for the Probit regression is solved analytically.

4.2 b)

Reasonable starting values for β can be to initialize the parameters with the observed values y. The starting values are hence given by β^0 below:

$$\beta^0 = (X^T X)^{-1} X y$$

4.3 c)

Below is the implementation of the EM algorithm, the optim solution (as well as a probit regression solved by R's implemented function glm, with the exponential family probit). All three methods produce about the same result for the parameter estimates. I also found three different strategies for calculating the standard error of the estimates, so that they can be compared below.

N.B. When carrying out a simulation and calculating the parameter estimates are not that close to the theoretical values, this is because the standard error of the parameters are very big (some estimates are even insignificant in comparison). This might be because of how the simulation of the initial values for the algorithm is setup (the three different methods end up with the same results).

N.B. 2 I did not find an explicit method for how to calculate the standard error of the estimate for one run in the probit EM regression (the solution by extracting the stanard error from the *lm* method in the algorithm is low in comparison to the glm and the optim estimates). The standard error given by R's glm function (with probit regression) should be the most accurate one. This should be close to the estimate that the EM method would give.

N.B. 3 An alternative solution to obtain the standard error as well as the parameter estimates was implemented (see after the solution to 3. d)).
Here I estimate the standard error and the parameters using EM, optim and GLM together with a Monte Carlo metho. This produces a result for the paramters that are closer to their theoretical values and also the standard errors of the estimates fits well in with theory and are close to each other (find this solution after subsection 3 d)). This is my preferred solution.

```r
##EM method

probit_EM <- function(X,Y,convCrit=10^-8,itMax=1000,print=FALSE) {

  beta <- solve(t(X)%*%X)%*%t(X)%*%Y #initialize beta

  it=0
  conv=FALSE



  while(it<itMax & conv==FALSE){

    mu=X%*%beta #update mean
    Z <- mu + dnorm(mu) * ( 1/(1-pnorm(-mu)) * Y - 1/pnorm(-mu) * (1-Y) )

    betaOld=beta # store old beta for stop critera

    beta <- solve(t(X)%*%X)%*%t(X)%*%Z

    c <- sqrt(sum((beta-betaOld)^2))/sqrt(sum(betaOld^2)) #stopping criteria
    if(is.nan(c)) {
      print("Convergence failed! Produced NaN.")
      it=itMax # break loop if nan
      return(NA)
    }
    if(c<=convCrit) {
      conv=TRUE # break loop
    }
    it=it+1
  }

  if(conv==TRUE) {
    if(print==TRUE) print(paste("Convergence reached! Steps =", it))
    mod<-lm(Z~X[,2]+X[,3]+X[,4])
    return(list(beta,mod))
  }
  else {
    print("convergence not reached")
    return(NA)
  }

}

# Test algorithm, simulate initial values

simulate_values <- function(beta,n,random=TRUE) {
  if(random==FALSE) set.seed(0) # to reproduce run

  cols=length(beta)

  X=matrix(c(rep(1,n),runif((cols-1)*n)),ncol=cols) #setup x-matrix
  meanZ<-X%*%beta
  sdZ=1
  Z <- rnorm(n,mean=meanZ,sd=sdZ)
  Y<-Z
  Y[Y>0]<-1
  Y[Y<=0]<-0

  XY<-list(X,Y)

  return(XY)
}

n=100

# Test
beta <- c(0.3,1.2,0,0)
XY<-simulate_values(beta,n,random=FALSE)
X<-XY[[1]]
Y<-XY[[2]]

res<-probit_EM(X,Y,print=TRUE)

## [1] "Convergence reached! Steps = 44"

print(EMBeta<-res[[1]]) #Print param estimates

##             [,1]
## [1,]  0.73584472
## [2,]  1.02333148
## [3,] -0.41388198
## [4,] -0.04180536
```

```
#N.B. I do not think the lines below are an accurate method of estimating S.E.(beta_1_hat)
#mod (lm estimate in the last step of algorithm) gives correct parameter estimates,
#but not correct estimate of the standard error (if we compare with optim and glm)
#Instead see Monte Carlo solution further down below for correct solution and estimate
mod<-res[[2]]
print(seBetaHatEM<-summary(mod)$coefficients[2, 2]) #S.E(beta_1_hat)

## [1] 0.2497357

EMBeta[2]/seBetaHatEM #Signal to Noise Ratio beta_1_hat / S.E. (beta_1_hat)

## [1] 4.097658

#### GLM Probit Result (for comparison) ####

# Result, more accurarte S.E.(beta_1_hat)
glmProbit = glm(Y ~ X[,2]+X[,3]+X[,4], family=binomial(link = "probit"),
                control=list(maxit=500, epsilon=1e-8))

print(glmBeta <- coef(glmProbit)) #Almost same as EM

## (Intercept)      X[, 2]      X[, 3]      X[, 4]
##  0.73584476  1.02333161 -0.41388145 -0.04180624

print(seBetaHatGLM<-summary(glmProbit)$coefficients[2, 2]) #standard error of the estimate

## [1] 0.5847536

glmBeta[2]/seBetaHatGLM #signal to noise ratio beta_1_hat / S.E. (beta_1_hat)

##   X[, 2]
## 1.750022

#Ok this is close to 2 (never mind EM result above)
```

4.4 d)

Below the optim method was implemented and used to find the estimates of the parameters (which roughly equals the estimates found with the EM and the GLM). We also see that the stanard error of $\hat{\beta}_1$ for the optim method is slightly larger than the one for

```
##Optim BFGS method

#Define Loglikelihood for Probit model
loglik_probit = function(beta, X, Y){
  mu = X%*%beta
  ll = sum(Y*log(pnorm(mu)) + (1-Y)*log(pnorm(-mu)))  # compute the log likelihood
  ll
}

#Initialize beta_0
beta_0<-solve(t(X)%*%X)%*%t(X)%*%Y

optimRes = optim(beta_0, loglik_probit,X=X, Y=Y, control=list(fnscale=-1, maxit=1000,
                                                                        reltol=1e-8),hessian=TRUE)
#fnscale=-1 for maximization

optimBeta<-optimRes$par

# Calculate standard error of the estimates
fisherInfo<-solve(-optimRes$hessian) #extract Fisher information matrix
propSigma<-sqrt(diag(fisherInfo)) #take square root of diagonal elements
propSigma<-diag(propSigma) #Only extract standard error for the parameters
print(seBetaHatOptim<-propSigma[2,2]) #Extract standard error for parameter beta_1

## [1] 0.5807264

#Ok close to the value given in the GLM solution (not as good)

optimBeta[2]/seBetaHatOptim #beta_1_hat/S.E.(beta_1_hat), close to glm probit result

## [1] 1.76105

## Comparison

# Compare all parameter estimates:

data.frame('EMBeta'=EMBeta,'optimBeta'=optimBeta,'glmBeta'=glmBeta)

##                   EMBeta   optimBeta     glmBeta
## (Intercept)  0.73584472  0.73556814  0.73584476
## X[, 2]       1.02333148  1.02268851  1.02333161
## X[, 3]      -0.41388198 -0.41330896 -0.41388145
## X[, 4]      -0.04180536 -0.04223504 -0.04180624

#Ok parameter estimates are similar for all three methods
```

4.5 Problem 3 c) and d) Estimating parameters and standard error through MC

Below a Monte Carlo Method was implemented to calculate the parameter estimates and the standard errors. They are very much closer to their theoretical values here and the standard errors of the parameters are much lower in comparison to their value for only one run. This would be a much more reliable method to use in order to find model parameters

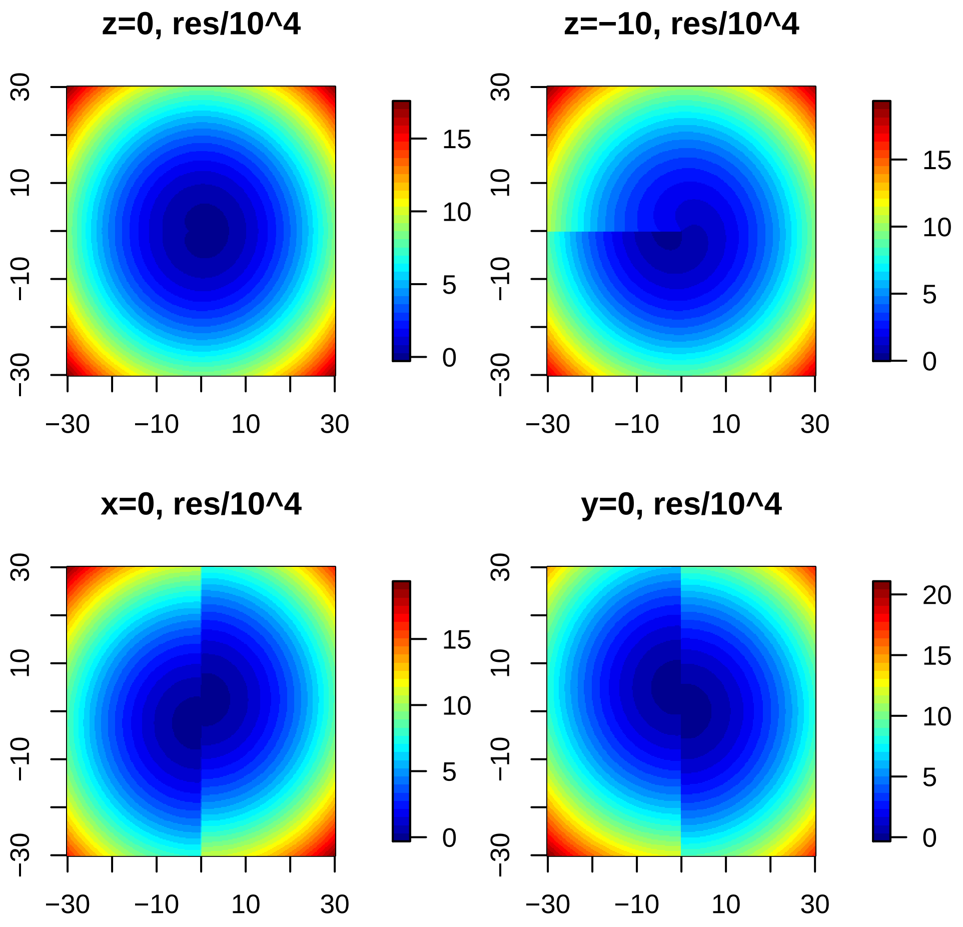
```
beta <- c(1,0.18,0,0)
n <- 100
mcRep <- 100 # repetitions for the MC method

emParam <- data.frame(matrix(ncol = length(beta), nrow = mcRep))
optimParam <- data.frame(matrix(ncol = length(beta), nrow = mcRep))
glmParam <- data.frame(matrix(ncol = length(beta), nrow = mcRep))

for (i in 1:mcRep){
  # start simulation
  XY <- simulate_values(beta,n) # we do not set seed to zero
  X <- XY[[1]]
  Y <- XY[[2]]
  #Store results
  emParam[i,] <- probit_EM(X,Y)[[1]]
  optimParam[i,] <- optim(par=solve(t(X)%*%X)%*%t(X)%*%Y,loglik_probit,
                          gr="BFGS",X=X,Y=Y,control=list(fnscale=-1, maxit=1000, reltol=1e-8))$par
  glmParam[i,] <- glm(Y ~ X[,2]+X[,3]+X[,4], family=binomial(link = "probit"))$coefficient

}

# Setup data frames to store results
emResults <- data.frame('EM mean'=vector("numeric",length = length(beta)),
                         'EM se'=vector("numeric",length = length(beta)),
                        'EM SNR'=vector("numeric",length = length(beta)))
optimResults <- data.frame('Optim mean'=vector("numeric",length = length(beta)),
                          'Optim se'=vector("numeric",length = length(beta)),
                        'Optim SNR'=vector("numeric",length = length(beta)))
glmResults <- data.frame('GLM mean'=vector("numeric",length = length(beta)),
                         'GLM se'=vector("numeric",length = length(beta)),
                        'GLM SNR'=vector("numeric",length = length(beta)))

# Define function to calc standard error
stdError <- function(x) sd(x)/sqrt(length(x))

# Calc results
for (i in 1:length(beta)){
  emResults[i,1] <- mean(emParam[,i])
  emResults[i,2] <- stdError(emParam[,i])
  emResults[i,3] <- emResults[i,1]/emResults[i,2]

  optimResults[i,1] <- mean(optimParam[,i])
  optimResults[i,2] <- stdError(optimParam[,i])
  optimResults[i,3] <- optimResults[i,1]/optimResults[i,2]

  glmResults[i,1] <- mean(glmParam[,i])
  glmResults[i,2] <- stdError(glmParam[,i])
  glmResults[i,3] <- glmResults[i,1]/glmResults[i,2]
}

# Print results, Ok!
emResults

##       EM.mean      EM.se    EM.SNR
## 1  1.12313131 0.05703797 19.690943
## 2  0.18502251 0.06858861  2.697569
## 3  0.00783345 0.05392524  0.145265
## 4 -0.07543897 0.06082478 -1.240267

optimResults

##     Optim.mean   Optim.se  Optim.SNR
## 1  1.123255027 0.05703903 19.6927445
## 2  0.184914037 0.06858329  2.6961967
## 3  0.007784448 0.05392928  0.1443455
## 4 -0.075435697 0.06083082 -1.2400900

glmResults
```

```
##        GLM.mean     GLM.se    GLM.SNR
## 1  1.123131105 0.05703763 19.6910552
## 2  0.185022075 0.06858825  2.6975769
## 3  0.007832481 0.05392502  0.1452476
## 4 -0.075438342 0.06082461 -1.2402602
```

5 Problem 4

Below we use R to examine the hellical valley function (that is sourced in as $f()$). Slices of it are plotted by keeping one input parameter fixed. As can be seen from the results we find that the function has several local minima and therefore we have to be considerate of the starting values when we carry out the optimization in order to find the global minima.

```r
source("helical.R")

n = 30
x = y = z = seq(-n,n,0.2)

#define function to extract results from helical function
helical_calc <- function(x=seq(-n,n,0.2),y=seq(-n,n,0.2),z=seq(-n,n,0.2)) {

  gr = expand.grid(x,y,z) #expand grid for 2d plotting
  len <- dim(gr)[1]

  fVals<-rep(0,len)
  for(i in 1:len) {
    fVals[i]<-f(c(gr[i,1],gr[i,2],gr[i,3])) # call the helical function
  }
  sqDim<-max(length(x),length(y),length(z)) # for square matrix dimensions
  fMatrix<-matrix(fVals,sqDim,sqDim)
  return(fMatrix)
}

par(mfrow=c(2,2), mar=c(2,2,4,4))


# z fixed

result<-helical_calc(z=0)
image.plot(x, y,result/10^4,col = tim.colors(32),xlab="x",ylab="y",main="z=0, res/10^4")

result<-helical_calc(z=-10)
image.plot(x, y,result/10^4,col = tim.colors(32),xlab="y",ylab="z",main="z=-10, res/10^4")

# x fixed
result<-helical_calc(x=0)
image.plot(y, z,result/10^4,col = tim.colors(32),xlab="y",ylab="z",main="x=0, res/10^4")

# y fixed
result<-helical_calc(y=0)
image.plot(x, z,result/10^4,col = tim.colors(32),xlab="x",ylab="z",main="y=0, res/10^4")
```



Since it looks like the function has several local minimas we can use the R functions *optim*() and *nlm*() to examine if we can find several local minimas depending on what starting values we use.

Result: After a lot of experimentation the minimum value is found to be zero at $(x, y, z) = (1, 0, 0)$. We also have several local minimas, e.g. both nlm and optim find several different local minimas for the hellical function.

See below where the function is analyzed and different minimas are found using *optim* and *nlm*

```
  run_optim <- function(x,y,z,roundTo=1) {

    sV<-as.matrix(expand.grid(x,y,z))
    minParOpt<-matrix(rep(0,length(sV)),ncol=dim(sV)[2])
    minValOpt<-matrix(rep(0,dim(sV)[1]))
    minParNLM<-minParOpt
    minValNLM<-minValOpt
    for (i in 1:dim(sV)[1]) {
      o<-optim(sV[i,],f)
      nlmRes<-nlm(f,sV[i,])
      minParOpt[i,]<-round(o$par,roundTo)
      minValOpt[i,]<-round(o$value,roundTo)
      minParNLM[i,]<-round(nlmRes$estimate,roundTo)
      minValNLM[i,]<-round(nlmRes$minimum,roundTo)
    }

    dfOpt<-data.frame(sV,minParOpt,minValOpt)
    colnames(dfOpt)<-c("x0","y0","z0","x","y","z","optMin")
    dfNLM<-data.frame(sV,minParNLM,minValNLM)
    colnames(dfNLM)<-c("x0","y0","z0","x","y","z","nlmMin")

    return(list(dfOpt,dfNLM))

  }

  ### Optimize over big values
  n = 100
  x = y = z = seq(-n,n,50)

  optimRes<-run_optim(x,y,z)[[1]]
  nlmRes<-run_optim(x,y,z)[[2]]


  head(unique(optimRes))
```

```
##     x0   y0   z0    x    y    z optMin
## 1 -100 -100 -100  1.0  0.0  0.1    0.0
## 2  -50 -100 -100  0.9 -0.5 -0.9    1.0
## 3    0 -100 -100  1.0  0.0  0.0    0.0
## 4   50 -100 -100 -1.1  0.0 -5.0   26.4
## 5  100 -100 -100  0.9 -0.5 -0.8    0.8
## 6 -100  -50 -100  1.0 -0.1 -0.2    0.1
```

```
head(unique(nlmRes))
```

```
##     x0   y0   z0 x y z nlmMin
## 1 -100 -100 -100 1 0 0      0
## 2  -50 -100 -100 1 0 0      0
## 3    0 -100 -100 1 0 0      0
## 4   50 -100 -100 1 0 0      0
## 5  100 -100 -100 1 0 0      0
## 6 -100  -50 -100 1 0 0      0
```

```
#Do not print duplicated results for minimum function value
head(optimRes[!duplicated(optimRes[,c('optMin')]),])
```

```
##      x0   y0   z0    x    y    z optMin
## 1  -100 -100 -100  1.0  0.0  0.1    0.0
## 2   -50 -100 -100  0.9 -0.5 -0.9    1.0
## 4    50 -100 -100 -1.1  0.0 -5.0   26.4
## 5   100 -100 -100  0.9 -0.5 -0.8    0.8
## 6  -100  -50 -100  1.0 -0.1 -0.2    0.1
## 11 -100    0 -100  1.0  0.2  0.4    0.2
```

```
head(nlmRes[!duplicated(nlmRes[,c('nlmMin')]),])
```

```
##       x0   y0   z0 x y    z   nlmMin
## 1   -100 -100 -100 1 0    0      0.0
## 13     0    0 -100 0 0 -100 960720.8
## 38     0    0  -50 0 0  -50 228222.3
## 63     0    0    0 0 0    0    100.0
## 113    0    0  100 0 0  100 960720.7

  ### Optimize over smaller values

  n = 2
  x = y = z = seq(-n,n,0.5)

  optimRes<-run_optim(x,y,z)[[1]]
  nlmRes<-run_optim(x,y,z)[[2]]

  head(unique(optimRes))

##     x0 y0 z0 x y z optMin
## 1 -2.0 -2 -2 1 0 0      0
## 2 -1.5 -2 -2 1 0 0      0
## 3 -1.0 -2 -2 1 0 0      0
## 4 -0.5 -2 -2 1 0 0      0
## 5  0.0 -2 -2 1 0 0      0
## 6  0.5 -2 -2 1 0 0      0

  head(unique(nlmRes))

##     x0 y0 z0 x y z nlmMin
## 1 -2.0 -2 -2 1 0 0      0
## 2 -1.5 -2 -2 1 0 0      0
## 3 -1.0 -2 -2 1 0 0      0
## 4 -0.5 -2 -2 1 0 0      0
## 5  0.0 -2 -2 1 0 0      0
## 6  0.5 -2 -2 1 0 0      0

  head(optimRes[!duplicated(optimRes[,c('optMin')]),])

##       x0 y0 z0   x    y    z optMin
## 1   -2.0 -2 -2 1.0  0.0  0.0    0.0
## 652 -0.5 -2  2 0.7 -0.7 -1.2    1.5

  head(nlmRes[!duplicated(nlmRes[,c('nlmMin')]),])

##     x0 y0   z0   x    y    z nlmMin
## 1   -2 -2 -2.0 1.0  0.0  0.0    0.0
## 41   0  0 -2.0 0.0  0.0 -2.0  128.7
## 122  0  0 -1.5 0.0  0.0 -1.5  202.0
## 203  0  0 -1.0 0.8 -0.6 -1.0    1.0
## 365  0  0  0.0 0.0  0.0  0.0  100.0
## 608  0  0  1.5 0.0  0.0  1.5  100.7
```