Stat243: Problem Set 8

Qingan Zhao
SID: 3033030808

28 Nov. 2017

Problem 1

(a)

I think the tail of the Pareto decays more **slowly** than that of an exponential distribution, since the density of Pareto distribution is $\frac{\beta\alpha^{\beta}}{x^{\beta+1}}$ and the density of exponential distribution is $\beta^{-1}e^{\frac{x}{\beta}}$. Comparing the parts have x, it is obvious that Pareto decays more slowly (**polynomial vs exponent**, which is one of the reasons why it has a fat tail). We can confirm the conclusion by a pair of plots:

```
#compare the density of the two distributions to confirm our conclusion
x = seq(2, 10, 0.1)
par(mfrow=c(1, 2))
#dpareto() is in the 'actuar' package; alpha <- 2 and beta <- 3
plot(x, dpareto(x, 3, 2), ylab='density', type='l', main='Pareto')
plot(x, dexp(x), ylab='density', type='l', main='exponential')
```

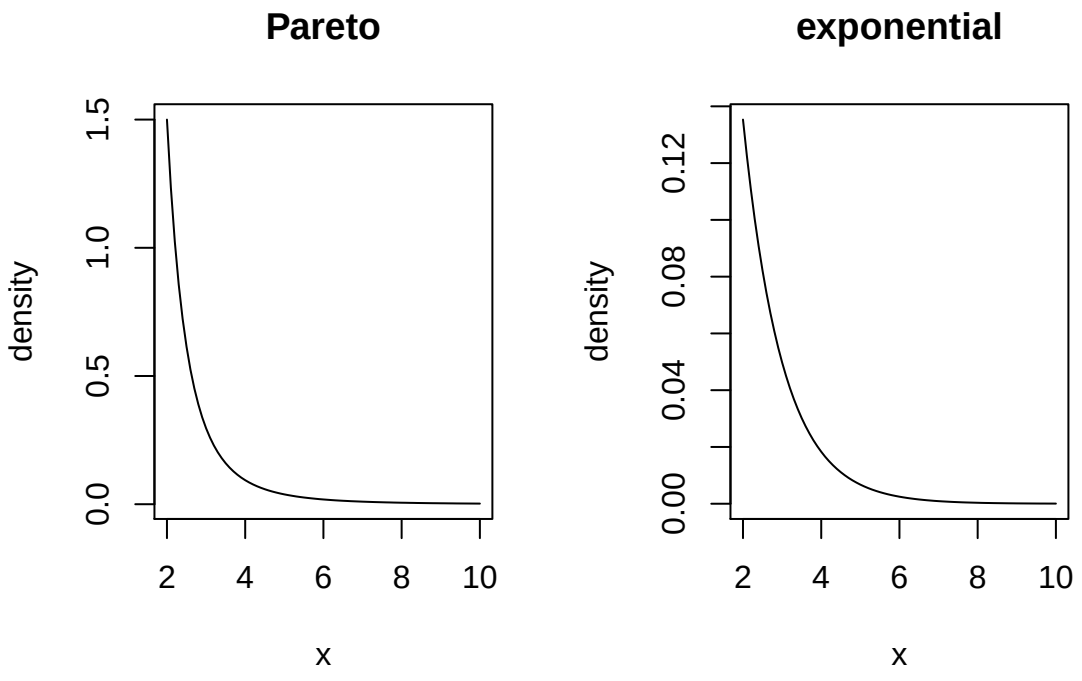

Fig.1: Pareto distribution vs exponential distribution

(b)

First let us generate x, $f(x)$, and $g(x)$. Since $\mathrm{Var}(\hat{\phi}) \propto \mathrm{Var}(h(x)f(x) = g(x))$, we can let $h(x)$ be x and x^2. Then compute the estimation of EX and $E(X2)$ and create the histograms.

```
set.seed(123)
m <- 10000
#generate x, f(x), and g(x)
x <- rpareto(m, 3, 2)
fx <- dexp(x-2)
gx <- dpareto(x, 3, 2)
#h(x)f(x)/g(x) when h(x)=x
EX <- fx / gx * x
#h(x)f(x)/g(x) when h(x)=x^2
EX2 <- fx / gx * x^2

#histograms
par(mfrow=c(2, 2))
hist(EX)
hist(EX2)
hist(fx / gx)
```

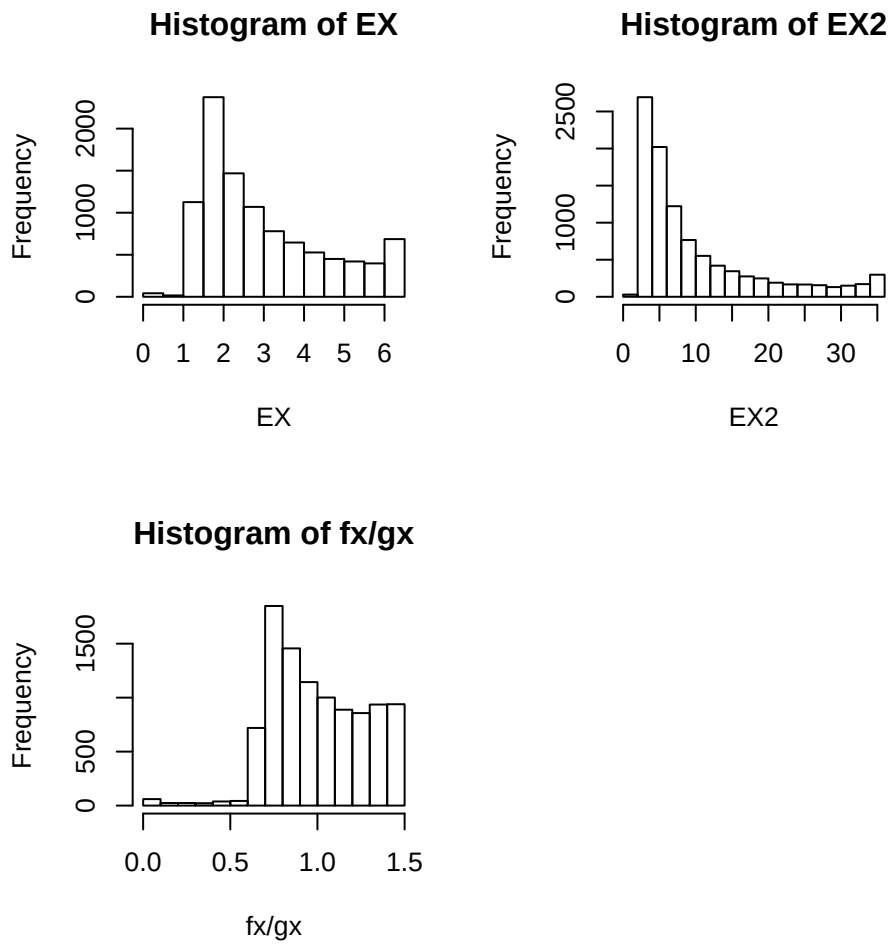

Fig.2: Histograms of $h(x)f(x)/g(x)$ and of the weights $f(x)/g(x)$ (the first scenario)

We can calculate the variances:

```
var(x)

## [1] 2.636112

var(EX)

## [1] 2.354837

var(x^2)

## [1] 1139.859

var(EX2)

## [1] 76.25274
```

Given that $f(x)$ is an exponential distribution and $g(x)$ is a Pareto distribution, the weight will not be large (except $h(x)$ is small). Hence, $\text{Var}(\hat{\phi})$ (which $\propto \text{Var}(h(X)f(X) = g(X))$ is not large ($\text{Var}(h(X)f(X) = g(X)$ is smaller than $\text{Var}(h(x))$).

(c)

Switch $f(x)$ and $g(x)$ and do the same thing as (b).

```
set.seed(123)
#regenerate x, f(x), and g(x)
x <- rexp(m) + 2    #dpareto() does not allow x<alpha so we just add 2
fx <- dpareto(x, 3, 2)
gx <- dexp(x-2)
#h(x)f(x)/g(x) when h(x)=x
EX <- fx / gx * x
#h(x)f(x)/g(x) when h(x)=x^2
EX2 <- fx / gx * x^2

#histograms
par(mfrow=c(2, 2))
hist(EX)
hist(EX2)
hist(fx / gx)
```

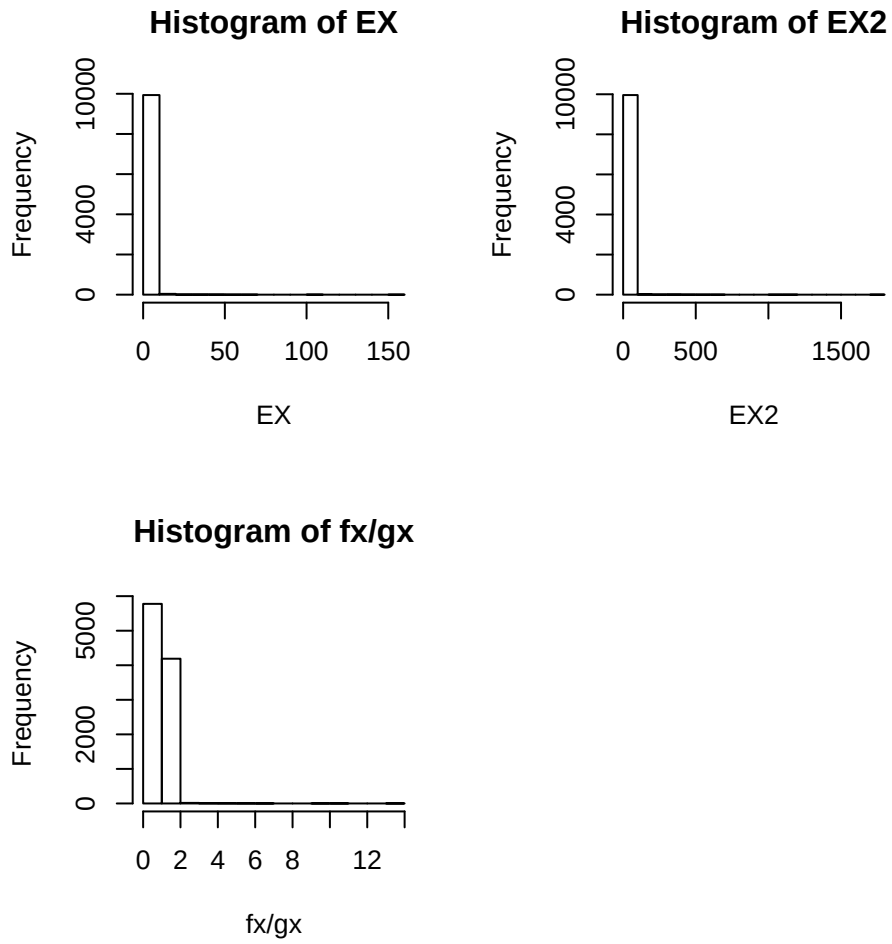

Fig.2: Histograms of $h(x)f(x)/g(x)$ and of the weights $f(x)/g(x)$ (the second scenario)

calculate the variances:

```
var(x)

## [1] 0.9997644

var(EX)

## [1] 7.885574

var(x^2)

## [1] 67.27421

var(EX2)

## [1] 843.5813
```

Given that $f(x)$ is a Pareto distribution and $g(x)$ is an exponential distribution, the weight will be large. Hence, $\text{Var}(\hat{\phi})$ (which $\propto \text{Var}(h(X)f(X) = g(X))$ is large ($\text{Var}(h(X)f(X) = g(X)$ is larger than $\text{Var}(h(x))$).

Problem 2

To get a sense for how the function behaves, let us choose some constant values and assign them to the third input ($x[3]$). The values I chose are {-10, -2, 0, 1, 2, 10}. So 6 slices of the function can be generated as follows:

```
#"helical valley" function provided by the instructor
theta <- function(x1,x2) atan2(x2, x1)/(2*pi)

f <- function(x) {
  f1 <- 10*(x[3] - 10*theta(x[1],x[2]))
  f2 <- 10*(sqrt(x[1]^2 + x[2]^2) - 1)
  f3 <- x[3]
  return(f1^2 + f2^2 + f3^2)
}

par(mfrow=c(4,3))
#assign 5 constants to the third input
#the reson why choose '1' without '-1' in this set is that this could make the length of x3 even
#thus plots arranged in 4x3 would make a nicer view of the document
X3 <- c(-10, -2, 0, 1, 2, 10)

#the size that the plots are going to show
x1 <- seq(-10, 10, 0.5)
x2 <- seq(-10, 10, 0.5)
Dim <- length(x1)

for (i in 1:length(X3)){
  #using paste0() to assign f1, f2,... in a for loop
  assign(paste0('f', i), apply(as.matrix(expand.grid(x1, x2)), 1, function(x) f(c(x, X3[i]))))
  x3 <- matrix(get(paste0('f', i)), Dim, Dim)
  #plot the slices in 3D
  persp(x1, x2, x3, main=paste0('Slice (3D) when x3=', X3[i]))
  #plot the slices in 2D
  image(x1, x2, x3, col=tim.colors(32), main=paste0('Slice when x3=', X3[i]))
  #add contour
  contour(x1, x2, x3, add=TRUE)
}
```

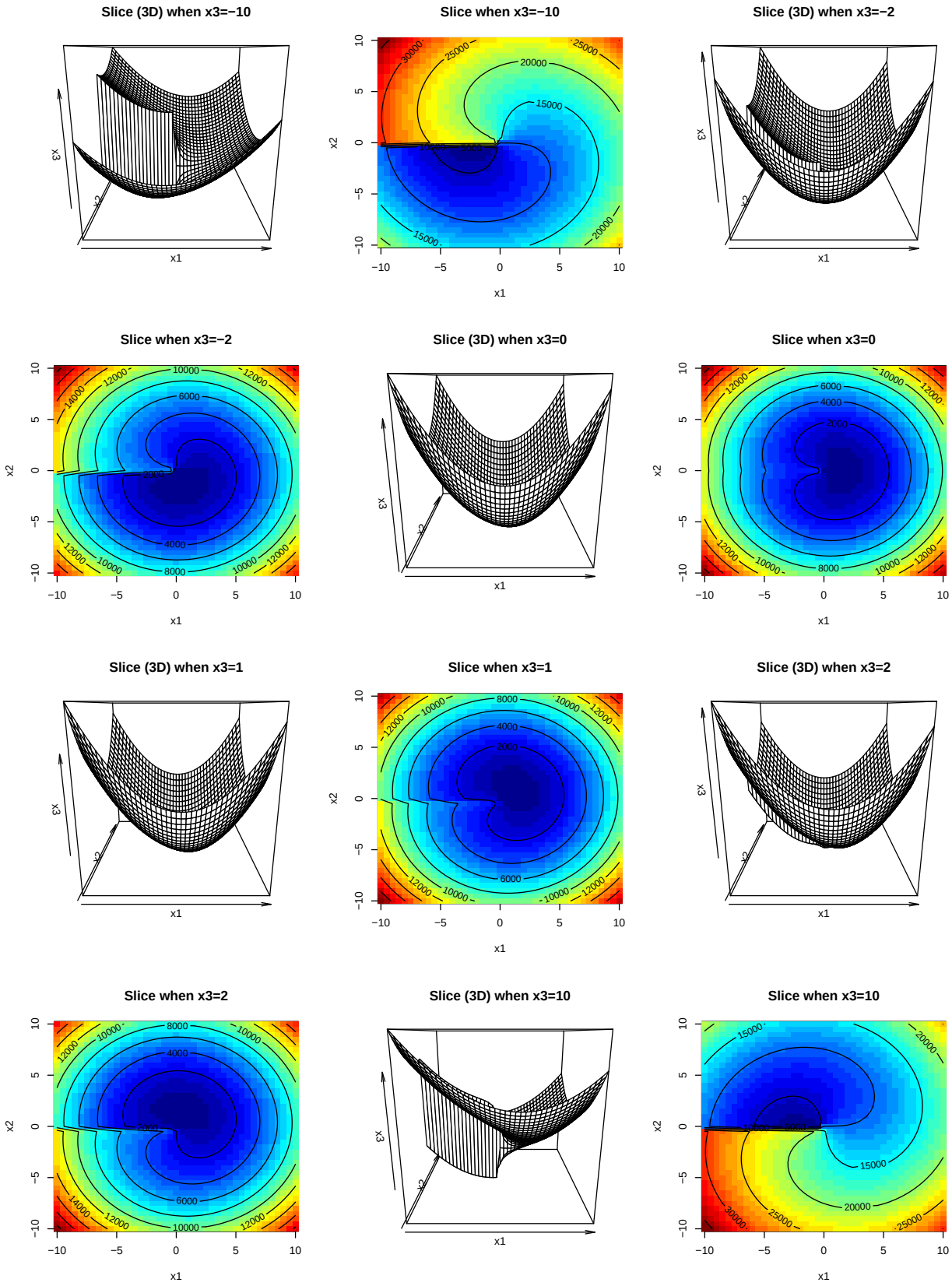

Fig.3: Slices of the function in 3D and 2D

From Figure 3 we could see that the slices of the function is just like "helical valleys". Now let us use *optim*() and *nlm*() to find the minimum of this function. The starting points I chose are (0,0,0), (10,1,1), (-10, 0, -6), (5, -7, 4), and (-3, -6, -2).

```
#explore the possibility of local minima
points <- cbind(c(0, 0, 0), c(10, 1, 1), c(10, 0, -6), c(5, -7, 4), c(-3, -6, 2))
for (i in 1:5){
  cat('minimum starting at (', points[1,i], points[2,i], points[3,i],
      ') using optim():', optim(points[, i], f)$value, '\n')
  cat('minimum starting at (', points[1,i],points[2,i], points[3,i],
      ') using nlm():', nlm(f, points[, i])$minimum, '\n')
}
```

```
## minimum starting at ( 0 0 0 ) using optim(): 1.876851e-05
## minimum starting at ( 0 0 0 ) using nlm(): 100
## minimum starting at ( 10 1 1 ) using optim(): 1.604782e-05
## minimum starting at ( 10 1 1 ) using nlm(): 1.700811e-08
## minimum starting at ( 10 0 -6 ) using optim(): 0.0001084182
## minimum starting at ( 10 0 -6 ) using nlm(): 1.194349e-18
## minimum starting at ( 5 -7 4 ) using optim(): 3.252054e-05
## minimum starting at ( 5 -7 4 ) using nlm(): 1.700941e-08
## minimum starting at ( -3 -6 2 ) using optim(): 9.20379e-05
## minimum starting at ( -3 -6 2 ) using nlm(): 1.796749e-18
```

```
#explore the possibility of local minimum points
for (i in 1:5){
  cat('minimum point starting at (', points[1,i], points[2,i], points[3,i],
      ') using optim(): \n', optim(points[, i], f)$par, '\n')
  cat('minimum point starting at (', points[1,i],points[2,i], points[3,i],
      ') using nlm(): \n', nlm(f, points[, i])$estimate, '\n')
}
```

```
## minimum point starting at ( 0 0 0 ) using optim():
##  0.9999783 0.002730698 0.00428464
## minimum point starting at ( 0 0 0 ) using nlm():
##  0 0 0
## minimum point starting at ( 10 1 1 ) using optim():
##  1.000243 -0.001833372 -0.003015501
## minimum point starting at ( 10 1 1 ) using nlm():
##  0.9999995 -8.222928e-05 -0.0001300762
## minimum point starting at ( 10 0 -6 ) using optim():
##  1.000833 -0.003355795 -0.004970957
## minimum point starting at ( 10 0 -6 ) using nlm():
##  1 2.720873e-10 3.289634e-10
## minimum point starting at ( 5 -7 4 ) using optim():
##  1.00053 0.001242596 0.001886815
## minimum point starting at ( 5 -7 4 ) using nlm():
##  0.9999995 -8.223237e-05 -0.0001300809
## minimum point starting at ( -3 -6 2 ) using optim():
##  0.9999658 -0.0058837 -0.009498276
## minimum point starting at ( -3 -6 2 ) using nlm():
##  1 4.568249e-10 7.493699e-10
```

Problem 3

(a)

Denote the starting values of the 3 parameters by β_{00}, β_{10}, and σ_0^2.

$$\mu = \beta_{00} + \beta_{10} \times x\{x > \tau\}$$

$$\tau^* = \frac{\tau - \mu}{\sigma_0}$$

$$\rho(\tau^*) = \frac{\phi(\tau^*)}{1 - \Phi(\tau^*)}$$

$$E(y|y > \tau) = \mu + \sigma_0 \rho(\tau^*)$$

$$V(y|y > \tau) = \sigma_0^2(1 + \tau^* \rho(\tau^*) - \rho(\tau^*)^2)$$

Assign $E(y|y > \tau)$ to all the truncated value in $y(y|y > \tau)$, so y is now updated as y^*.

Then the residual sum of squares (RSS) is:

$$RSS^* = \sum_{i=1}^{n} (y_i^* - \beta_{00} - \beta_{10} x_i)^2 + V(y|y > \tau) \tag{1}$$

$$\sigma = \sqrt{\frac{RSS^*}{n}}$$

Compute the log-likelihood, and expectation $Q(\theta|\theta_t)$ can be derived as:

$$Q(\theta|\theta_t) = -\frac{n}{2} log(2\pi\sigma_0^2) - \frac{RSS^*}{2\sigma_0^2} \tag{2}$$

Update β_0 and β_1 based on the result of the regression on y^*, and compute new y^*, σ, and Q using the updated parameters. Do the iteration until Q converges or reaches the maximum number of iterations.

(b)

The starting values for those parameters can be obtained by just running a regression on the uncensored data.

(c)

Now let us implement the algorithm in R. Write a function called $my\_EM()$, and test the function using the data simulated based on the code provided (a modest proportion of exceedances 20% and a high proportion 80%).

```r
#this function estimates the 3 parameters using the EM algorithm presented in 3a
my_EM <- function(dat, max_iter = 1e5){
  #initialization and set the starting values
  par_ini <- summary(lm(dat$y ~ dat$x))
  beta0_tmp <- par_ini$coefficients[1, 1]
  beta1_tmp <- par_ini$coefficients[2, 1]
  sigma_tmp <- par_ini$sigma
  x <- dat$x
  x_t <- x[dat$truncate == TRUE]
```

```
  y_star <- dat$y
  Q_tmp <- 0
  i <- 1 #the start point of the loop
  Q <- 1 #make sure the the following loop could start

  #this loop would compute the parameters
  while(i <= max_iter & abs(Q - Q_tmp) > 1e-6){
    #updata parameters and Q
    if(i != 1){
      beta0_tmp <- beta0
      beta1_tmp <- beta1
      sigma_tmp <- sigma
      Q_tmp <- Q
    }

    #compute E(y|y>tau), V(y|y>tau), and update y*
    mu <- beta0_tmp + beta1_tmp * x_t
    tau_star <- (dat$tau - mu) / sigma_tmp
    rho <- dnorm(tau_star) / (1 - pnorm(tau_star))
    Ey_t <- mu + sigma_tmp * rho
    Vy_t <- sigma_tmp^2 * (1 + tau_star * rho - rho^2)
    y_star[dat$truncate == TRUE] <- Ey_t

    #update beta0 and beta1 using lm()
    par <- lm(y_star ~ x)
    beta0 <- par$coefficients[1]
    beta1 <- par$coefficients[2]

    #update sigma, Q, and i
    RSS_star <- sum((y_star - beta0 - beta1 * x)^2) + sum(Vy_t)
    sigma <- sqrt(RSS_star / n)
    Q <- -n / 2 * log(2 * pi * sigma^2) - RSS_star / (2 * sigma^2)
    i <- i + 1
  }
  cat('beta0: ', beta0, '\n', 'beta1: ', beta1, '\n', 'sigma2: ', sigma^2, '\n',
      'iteration number: ', i, sep = '')
  return(list(beta0 = beta0, beta1 = beta1, sigma2 = sigma^2))
}

#test data provided by the instructor

set.seed(1)
n <- 100
beta0 <- 1
beta1 <- 2
sigma2 <- 6

x <- runif(n)
yComplete <- rnorm(n, beta0 + beta1*x, sqrt(sigma2))

#data process
tau1 <- quantile(yComplete, 0.2)  #modest proportion
tau2 <- quantile(yComplete, 0.8)  #high proportion
```

```r
truncate1 <- yComplete > tau1
truncate2 <- yComplete > tau2
dat1 <- list(x = x, y = pmin(tau1, yComplete), tau = tau1, truncate = truncate1)
dat2 <- list(x = x, y = pmin(tau2, yComplete), tau = tau2, truncate = truncate2)

#test
par1 <- my_EM(dat = dat1)

## beta0: 0.3126393
## beta1: 2.87922
## sigma2: 3.841963
## iteration number: 188

par2 <- my_EM(dat = dat2)

## beta0: 0.4566128
## beta1: 2.824108
## sigma2: 4.618876
## iteration number: 16
```

(d)

Now let us try a different approach: Use *optim*() to minimize the negative log-likelihood function to estimate the parameters. One thing should be mentioned is that the Hessian computed when the negative log-likelihood is minimized is the same as the Fisher information, and the covariance matrix is the inverse of the Fisher information. Hence, the standard errors of the estimated parameters can be obtained by taking the square roots of the diagonal elements of the covariance matrix.

```r
#set the starting values
par_ini1 <- summary(lm(dat1$y ~ dat1$x))
par_ini2 <- summary(lm(dat2$y ~ dat2$x))
ini1 <- c(beta0 = par_ini1$coefficients[1, 1], beta1 = par_ini1$coefficients[2, 1],
          sigma2 = par_ini1$sigma^2)
ini2 <- c(beta0 = par_ini2$coefficients[1, 1], beta1 = par_ini2$coefficients[2, 1],
          sigma2 = par_ini2$sigma^2)

#this function returns the negative log-likelihood (minimize it!)
NLL <- function(par, dat){
  beta0 <- par[1]
  beta1 <- par[2]
  sigma2 <- par[3]
  mu <- beta0 + beta1 * x
  LL <- sum(dnorm(dat$y[dat$truncate == FALSE], mu[dat$truncate == FALSE], sqrt(sigma2),
                  log = TRUE), pnorm(dat$tau, mu[dat$truncate == FALSE], sqrt(sigma2),
                                     log = TRUE, lower.tail = FALSE))
  return(-LL)
}

#optimization
par1 <- optim(par = ini1, fn = NLL, dat = dat1, hessian = TRUE)
par1$par #estimated parameters

##      beta0      beta1     sigma2
## -0.0315640  0.2292913  2.3423090
```

```r
diag(solve(par1$hessian))  #standard errors of the estimated parameters
```

```
##     beta0     beta1    sigma2
## 0.2266689 1.0103661 0.6761877
```

```r
cat('iteration number:', par1$counts[1])
```

```
## iteration number: 146
```

```r
par2 <- optim(par = ini2, fn = NLL, dat = dat2, hessian = TRUE)
par2$par #estimated parameters
```

```
##     beta0     beta1    sigma2
##  3.239550  1.167813 10.575832
```

```r
diag(solve(par2$hessian))  #standard errors of the estimated parameters
```

```
##     beta0     beta1    sigma2
## 0.3774623 1.1668081 3.5669699
```

```r
cat('iteration number:', par2$counts[1])
```

```
## iteration number: 110
```