Stat243: Homework 8

Linqing Wei

December 1, 2017

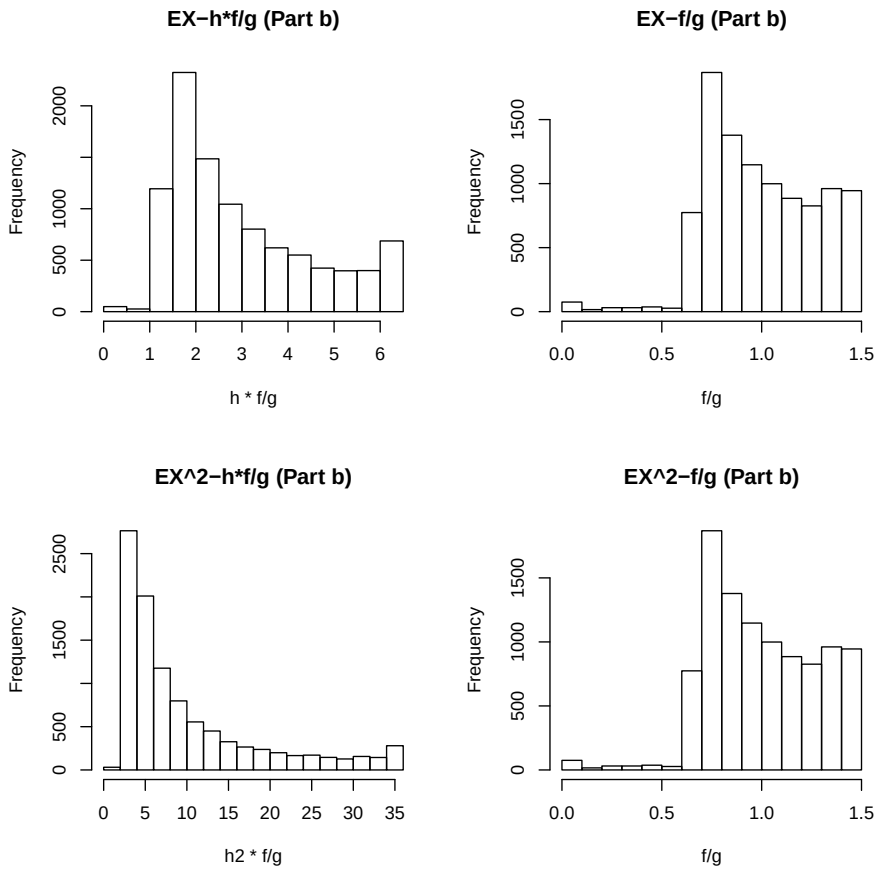
1 Problem 1

1.1 1A

The heavy-tailed distribution is a distribution for which the pdf goes to zero for large x. Therefore, Pareto distribution is heavy-tailed and will decay slower than an exponential function.

1.2 1B

```
par(mfrow=c(2,2))
library(EnvStats)
#Set parameters m,alpha, beta
m = 10000
alpha = 2
beta = 3
#sample x from pareto distribution
x = rpareto(m, alpha, beta)
#f is a shifted exp distribution
#g is a pareto distribution calculated in an explicit form
f = exp(2-x)
g = 24/(x^4)
h = x
#Estimate Ex
Ex = (1/m) * sum(h*f/g)
hist(h*f/g, main ="EX-h*f/g (Part b)")
hist(f/g, main = "EX-f/g (Part b)")
#Estimate EX^2
h2 = x^2
Ex2 = (1/m) * sum(h2*f/g)
hist(h2*f/g, main ="EX^2-h*f/g (Part b)")
hist(f/g, main ="EX^2-f/g (Part b)")
```

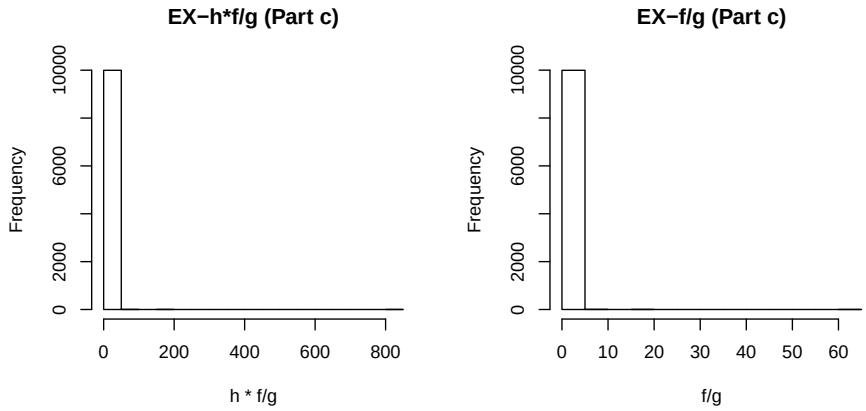


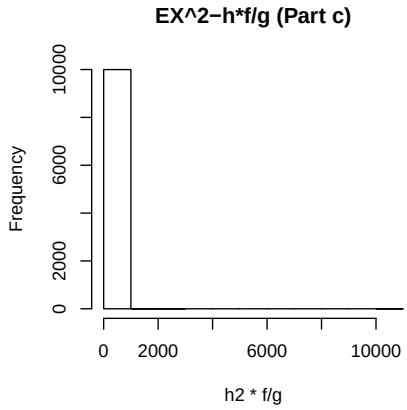
According to the histograms, estimator's variance is not too large.

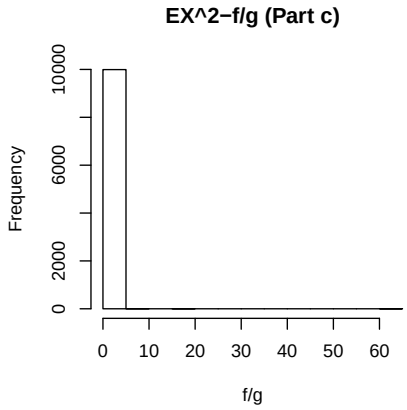
1.3 1C

```
par(mfrow=c(2,2))
library(rgl)
library(tolerance)
#x is now sampled from exp distribution
x = r2exp(10000, rate = 1, shift = 2)
f = 24/(x^4)
g = exp(2-x)
h = x
#Estimate EX
Ex = (1/m) * sum(h*f/g)
hist(h*f/g, main ="EX-h*f/g (Part c)")
hist(f/g, main = "EX-f/g (Part c)")
```

```r
#Estimate EX^2
h2 = x^2
Ex2 = (1/m) * sum(h2*f/g)
hist(h2*f/g, main ="EX^2-h*f/g (Part c)")
hist(f/g, main ="EX^2-f/g (Part c)")
```





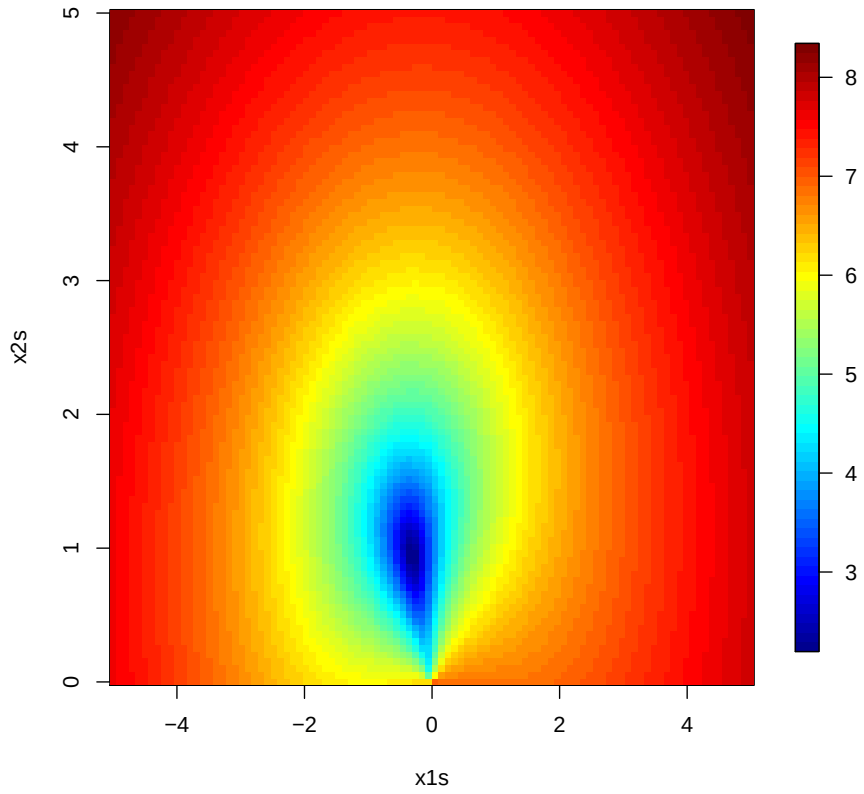


2 Problem 2

```r
library(fields)
theta <- function(x1,x2) atan2(x2, x1)/(2*pi)
f <- function(x){
  f1 <- 10*(3 - 10*theta(x[1],x[2]))
  f2 <- 10*(sqrt(x[1]^2 + x[2]^2) - 1)
  f3 <- 3
```

```
  return(f1^2 + f2^2 + f3^2)
}

#X3 is the value chosen to be fixed
x1s = seq(-5, 5, len = 100)
x2s = seq(0, 5, len = 100)
fx = apply(expand.grid(x1s,x2s), 1, f)
image.plot(x1s, x2s, matrix(log(fx), 100, 100))
```




```
#Now find minimum without fixing any values
f_original <- 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)
}
```

```r
#Try several sets of values, to see if there is a global min
init <- c(1, 1, 1)
optim(init, f_original, method = "Nelder-Mead")$value

## [1] 1.343098e-07

nlm(f_original,init)$minimum

## [1] 1.702065e-08

init <- c(10, 20, 30)
optim(init, f_original, method = "Nelder-Mead")$value

## [1] 0.6352194

nlm(f_original,init)$minimum

## [1] 1.642479e-16

init <- c(3, 50, 9)
optim(init, f_original, method = "Nelder-Mead")$value

## [1] 0.0121888

nlm(f_original,init)$minimum

## [1] 2.771798e-18
```

It is possible to get multiple local minima.

3 Problem 3

3.1 3C

```r
#Set parameter to generate yComplete dataset
set.seed(1)
n <- 100
beta0 <- 1
beta1 <- 2
sigma <- 6
x <- runif(n)
yComplete <- rnorm(n, beta0 + beta1*x, sqrt(sigma))

#EM function takes value of x, user defined dataset.
#Rate means "proportion of exceedances expected"
```

```r
EM <- function(x, data, rate){
  n = length(data)
  #Calculate total number of oberseved data
  c = n - n*rate
  #Sort observed data such that we only take the data below threshold
  observed = sort(data)[1:c]
  obs_x = runif(c)
  #fitting linear model based on observed data
  obsMod <- lm(observed ~ obs_x)
  #Set starting 3 parameter values based on lm
  beta0_init = summary(obsMod)$coef[1]
  beta1_init = summary(obsMod)$coef[2]
  sigma_init = var(observed)
  #create empty vectors for each parameter
  RR <- c()
  beta0 <- c()
  beta1 <- c()
  sigma <- c()
  #resample data based on starting parameters
  data_update <- rnorm(n, beta0_init + beta1_init*x, sqrt(sigma_init))
  #Expectation step
  for (i in 1:1000){
    mod_update <- lm(data_update ~ x)
    RR_i = summary(mod_update)$r.square
    RR <- c(RR, RR_i)
    beta0_i = summary(mod_update)$coef[1]
    beta0 <- c(beta0, beta0_i)
    beta1_i = summary(mod_update)$coef[2]
    beta1 <- c(beta1, beta1_i)
    sigma_i = var(data_update)
    sigma <- c(sigma, sigma_i)
    data_update <- rnorm(n, beta0_i + beta1_i, sqrt(sigma_i))
  }
  #After getting values from each E step, combine them into a matrix
  par <- cbind(RR, beta0, beta1, sigma)
  print(head(par))
  #Maximization step
  #in this case, we choose the row of parameters with minimum RRS
  m = which.min(par[,1])
  #The number of iterations
  print(m)
  return(c(par[m,]))
}
#Test cases
EM(x, yComplete, 0.2)
```

```
##                RR     beta0      beta1    sigma
## [1,] 0.003481691 1.2811585 -0.3848259 3.045517
## [2,] 0.006783256 0.6765349  0.5306039 2.971840
## [3,] 0.002268135 1.0054260  0.3356603 3.556754
## [4,] 0.001625914 1.4919097 -0.2959108 3.856086
## [5,] 0.014792362 0.3133095  0.9570218 4.433318
## [6,] 0.015528309 1.8015936 -1.0715036 5.294024
## [1] 664
##             RR         beta0         beta1         sigma
##  1.911188e-07 -1.743281e+01  1.996713e-03  1.493656e+00

EM(x, yComplete, 0.8)

##                 RR      beta0      beta1     sigma
## [1,] 0.1302803288 -1.8228542  1.3757265 1.0401790
## [2,] 0.0133436086 -0.3480980 -0.4389402 1.0338562
## [3,] 0.0179802049 -0.5072508 -0.5504714 1.2066955
## [4,] 0.0060395940 -1.1546986  0.3084472 1.1279159
## [5,] 0.0008846146 -0.7775020 -0.1205991 1.1772183
## [6,] 0.0100440082 -1.0377536  0.3718121 0.9855141
## [1] 709
##             RR         beta0         beta1         sigma
##  3.355551e-08  1.454304e+00 -1.162679e-04  2.884554e-02
```

3.2 3D

```
set.seed(2)
#Similar setup to get initial values
BFGS <- function(x, data, rate){
  n = length(data)
  c = n - n*rate
  observed = sort(data)[1:c]
  obs_x = runif(c)
  obsMod <- lm(observed ~ obs_x)
  beta0_init = summary(obsMod)$coef[1]
  beta1_init = summary(obsMod)$coef[2]
  sigma_init = var(observed)
 #Loglikelihood function to be optimized
  loglik_func <- function(par){
    (-(n-c)/2)*log(2*pi)-((n-c)/2)*log(par[3]^2)-(1/2*(par[3]^2))*sum((observed-par[1]-par[2
  }
  #Optimization step using BFGS
  init = c(beta0_init, beta1_init, sigma_init)
  optim(init, loglik_func, method = 'BFGS')
```

```
}
#Test cases
BFGS(x, yComplete, 0.2)

## $par
## [1]  1.201018e+00 -2.157620e-01  3.624151e+14
##
## $value
## [1] -1.468339e+31
##
## $counts
## function gradient
##        7        7
##
## $convergence
## [1] 0
##
## $message
## NULL

BFGS(x, yComplete, 0.8)

## $par
## [1] 7.507825e+24 3.560222e+24 1.530457e+29
##
## $value
## [1] -1.959657e+109
##
## $counts
## function gradient
##        6        6
##
## $convergence
## [1] 0
##
## $message
## NULL
```

EM takes around 600 700 steps while BFGS takes 6 7 steps.