Stat243: Problem Set 8

Worked with Jamie Palumbo, Mingyung Kim, Alanna Iverson

Sicun Huang

December 4, 2015

1. Conduct a simulation study on how regression methods perform when there are outlying values by comparing a new method that is supposedly robust to outliers to standard linear regression.

(a) The basic steps of the simulation study:

(1) Specify what makes up an individual experiment:

(i) Sample size: To make results from both methods comparable, use the same sample size when conducting the tests. Choose sample size to be greater than 100 so that prediction is significant.

(ii) Distributions: Choose a distribution with heavy tails so that there is a high chance of getting outliers. Here, we choose to generate data from a normal distribution with a large variance. Then, purpose half of the data set for prediction and the other half for estimation so that we can test the accuracy of the simulation study.

(iii) Parameters: We choose to use the same parameters for both methods, the intercept and the slope.

(iv) Statistics of interest: We are interested in the absolute prediction error and coverage of prediction intervals for new observations, where the prediction intervals are based on using the nonparametric bootstrap.

(v) Bootstrap size: Choose large enough bootstrap size so that prediction is significant.

(vi) Significance level: Conduct test at different significance levels (e.g. 0.10, 0.05, 0.01).

(2) Determine what inputs to vary:

(i) Sample size: Start with a relatively small number, then increase it gradually to see its effect on absolute prediction error and coverage of prediction intervals.

(ii) Significance level: Vary significance level to see its effect on prediction interval.

(iii) Bootstrap size: Similar to sample size, start with a relatively small number, then increase it gradually to see its effect on absolute prediction error and coverage of prediction intervals.

(3) Write code to carry out the individual experiment and return the quantities of interest:

We loop over each combination of the sample sizes, bootstrap sizes and significance levels and return the absolute prediction error and coverage of prediction intervals.

(4) For each combination of inputs, repeat the experiment m times.

(5) Summarize the results for each combo of interest, quantifying the simulation uncertainty.

(6) Report the results in graphical or tabular form.

(b) Now we demonstrate the above process with an example:

We begin with generating 4000 x_i's and y_i's from normal distributions. Choose a high variance to get outliers; this will allow us to test the robustness of the method. Then, divide the data set into two halves, where one will be used for prediction and the other for estimation. Run a linear regression on the estimation half to get estimates of the β's. Then, run the prediction function (predict) with these estimated β's to get a set of new y_i's. Compare the new y_i's to y_i's in the prediction data set (values we believe the y_i's should be). We can then calculate the absolute prediction error and the coverage of the prediction interval using these results.

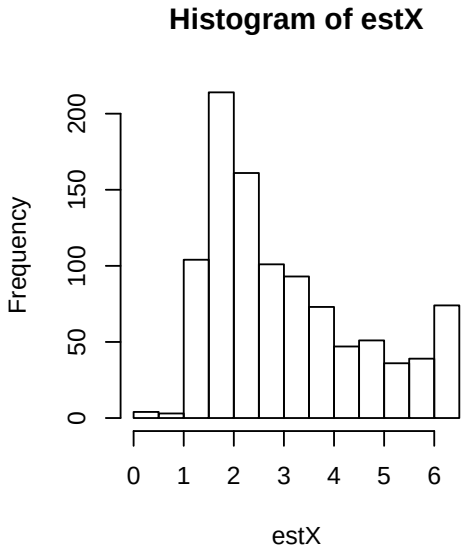
2.

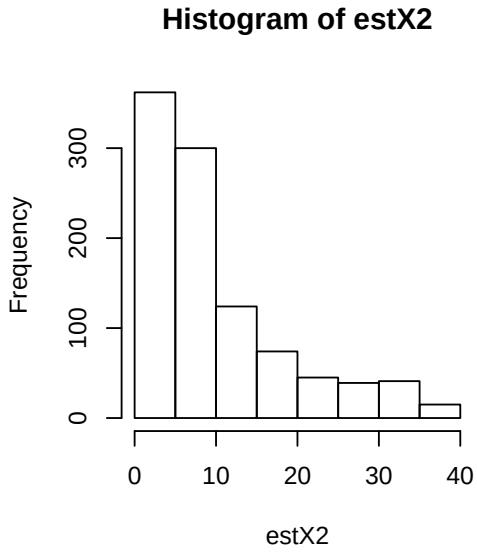
(a) By looking at the density functions of Pareto distribution and exponential distribution, we can see that tail of the Pareto decays more slowly than that of an exponential distribution. Since the sampling density (Pareto) have heavier tails than the density of interest (exponential), we can use it with the exponential distribution for importance sampling.

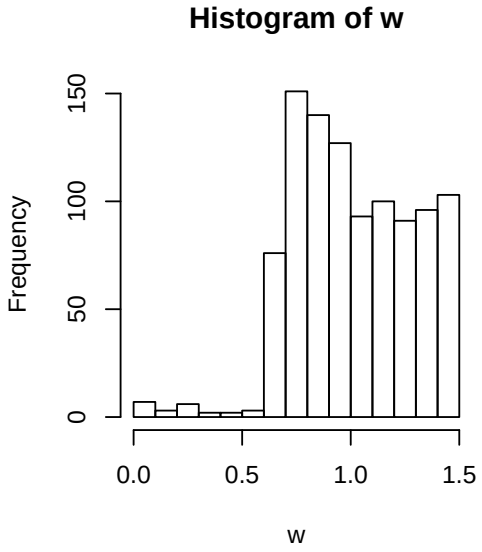
(b)

```r
# note: use PtProcess package here because other packages use general pareto
# distribution
library(PtProcess)

# number of samples for each estimator
m <- 1000

set.seed(0)
# samples for importance sampler sample from g(x), pdf of Pareto(2,3)
x <- rpareto(m, lambda = 3, a = 2)
# density of x under f; shifted to the right by 2
f <- dexp(x - 2, rate = 1)
# density of x under g
g <- dpareto(x, lambda = 3, a = 2)
# weights
w <- f/g

est1 <- mean(w * x)
est1

## [1] 3.054522

est2 <- mean(w * (x^2))
est2

## [1] 10.18948

# h(x)f(x)/g(x) with h(x)=x
estX <- w * x
# h(x)f(x)/g(x) with h(x)=x^2
estX2 <- w * (x^2)

par(mfrow = c(2, 2), pty = "s")
hist(estX)
```

```r
hist(estX2)
hist(w)

var(x)

## [1] 2.212009

var(estX)

## [1] 2.351836

var(x^2)

## [1] 407.6577

var(estX2)

## [1] 73.06215
```







Since we chose the distribution with heavier tails (Pareto) as g, the weight of importance sampling, $w = f(x)/g(x)$, is large only when h(x) is very small; this will help avoid having overly influential points. So Var(h(x)f(x)/g(x)), which is proportional to $Var(\hat{\mu})$, is smaller than or similar to Var(h(x)); Var(h(x)f(x)/g(x)) is smaller than Var(h(x)) especially when $h(x) = x^2$, since such a h can produce more extreme values comparing to when $h(x) = x$.

(c)

```
rm(list = ls())

# number of samples for each estimator
m <- 1000

set.seed(0)
# samples for importance sampler sample from g(x), pdf of exp(1)+2
```

```r
x <- rexp(m, rate = 1) + 2
# density of x under f
f <- dpareto(x, lambda = 3, a = 2)
# density of x under g
g <- dexp(x - 2, rate = 1)
# weights
w <- f/g

est1 <- mean(w * x)
est1
```

```
## [1] 2.871585
```

```r
est2 <- mean(w * (x^2))
est2
```

```
## [1] 9.629463
```

```r
# h(x)f(x)/g(x) with h(x)=x
estX <- w * x
# h(x)f(x)/g(x) with h(x)=x^2
estX2 <- w * (x^2)

par(mfrow = c(2, 2), pty = "s")
hist(estX)
hist(estX2)
hist(w)

var(x)
```

```
## [1] 0.9767331
```

```r
var(estX)
```

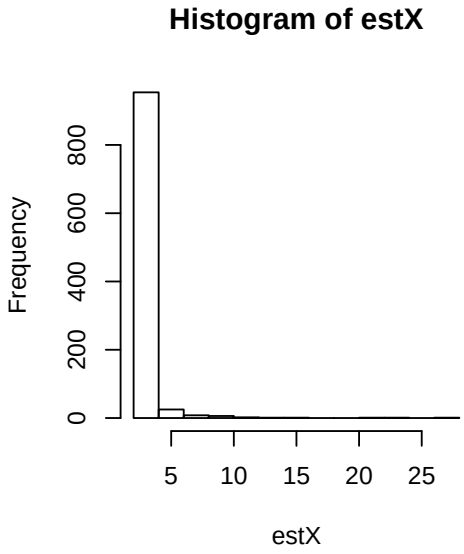
```
## [1] 2.290225
```

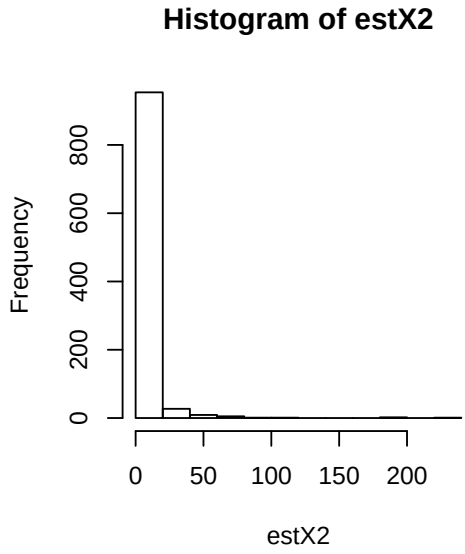
```r
var(x^2)
```

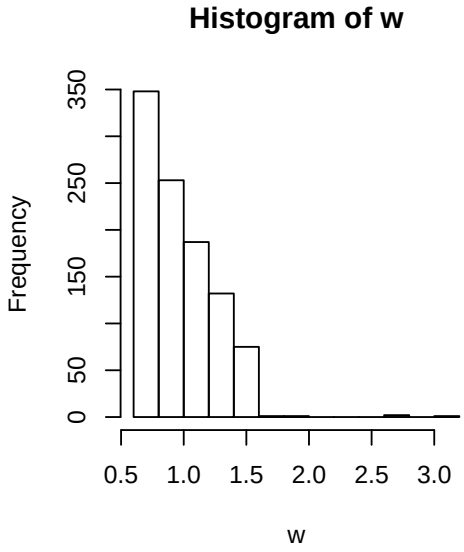
```
## [1] 64.06968
```

```r
var(estX2)
```

```
## [1] 180.9411
```







Since we chose the distribution with heavier tails (Pareto) as f, the weight of importance sampling, $w = f(x)/g(x)$, is large even when h(x) is not small; this will result in overly influential points. So Var(h(x)f(x)/g(x)), which is proportional to $Var(\hat{\mu})$, is larger than Var(h(x)); especially when $h(x) = x^2$, since such a h can produce more extreme values comparing to when $h(x) = x$.

3.

(b) Since we only know the observed y_i's, the reasonable starting value for β is the estimated β from the lm() function. i.e. to make a guesses of the starting value of β, we ignore the missing value z_i's.

(c) Write an R function to estimate the parameters (β_0, β_1, β_2, β_3). We choose to stop the optimization if error is less than tolerance value or if number of itations exceeds 10000. Let $n = 100$; choose parameters that satisfy $\hat{\beta}_1/se(\hat{\beta}_1) \approx 2$ and $\beta_2 = \beta_3 = 0$ (i.e. choose β_1 such that the signal to noise ratio in the relationship between x_1 and y is moderately large).

```
#function to solve for EM estimators in probit regression
probitEM <- function(y, x1, x2, x3, intVal=c(1,0.5,0,0), tolerance = .Machine$double.eps^0.5,
                     maxIteration=10000){
  n <- length(y)
  #intialization
  b0 <- intVal[1]
  b1 <- intVal[2]
  b2 <- intVal[3]
  b3 <- intVal[4]
  it <- 1
  diff <- Inf

  while (diff > tolerance & it < maxIteration){
    #save starting values of beta's for later
    intb0 <- b0
    intb1 <- b1
    intb2 <- b2
    intb3 <- b3

    #E step
    mu <- b0 + b1*x1 + b2*x2 + b3*x3
    #from formula in (a)
    #notice z is N(mu,1), latent variable
    z <- ifelse(y==1, mu+dnorm(mu,mean=0,sd=1)/pnorm(mu,mean=0,sd=1),
                mu-dnorm(mu,mean=0,sd=1)/pnorm(-mu,mean=0,sd=1))

    #M step; estimate betas using lm function
    b0 <- coef(lm(z ~ x1+x2+x3))[1]
    b1 <- coef(lm(z ~ x1+x2+x3))[2]
    b2 <- coef(lm(z ~ x1+x2+x3))[3]
    b3 <- coef(lm(z ~ x1+x2+x3))[4]

    absDiff <- abs(c(b0-intb0, b1-intb1, b2-intb2, b3-intb3))
    diff <- sum(absDiff)/sum(abs(c(intb0, intb1, intb2, intb3)))

    it <- it+1
  }
  return(list(b0, b1, b2, b3, it))
}

#function to test different beta values to find the ones that make b1hat/se(b1hat) close to 2
test <- function(b0,b1){
  #assumed in problem
  n <- 100
```

```
  b2 <- 0
  b3 <- 0
  x1 <- rnorm(n)
  x2 <- rnorm(n)
  x3 <- rnorm(n)
  XTb <- b0 + b1*x1 + b2*x2 + b3*x3

  set.seed(0)
  y <- rbinom(n, 1, prob = pnorm(XTb))

  #return z value for coefficient of x1, check if close to 2
  summary(glm(y ~ x1+x2+x3, family=binomial(link = "probit")))$coef[2,3]
}

test(0,0)
```

```
## [1] 0.1751074
```

```
test(1,1)
```

```
## [1] 4.284799
```

```
test(1,0.5)
```

```
## [1] 2.580574
```

```
#close to 2
test(1,0.3)
```

```
## [1] 2.089389
```

```
#results from the above trials
b0 <- 1
b1 <- 0.3

#assumed in problem
n <- 100
b2 <- 0
b3 <- 0
x1 <- rnorm(n)
x2 <- rnorm(n)
x3 <- rnorm(n)
XTb <- b0 + b1*x1 + b2*x2 + b3*x3

set.seed(0)
y <- rbinom(n, 1, prob = pnorm(XTb))

#test choice of starting value of beta in (b)
linearReg <- lm(y ~ x1 + x2 + x3)
intVal <- as.double(coef(linearReg))
probitEM(y, x1, x2, x3, intVal)
```

```
## [[1]]
## (Intercept)
##   0.9166837
##
```

```
## [[2]]
##        x1
## 0.3795008
##
## [[3]]
##        x2
## 0.1595485
##
## [[4]]
##          x3
## -0.02449302
##
## [[5]]
## [1] 40
```

(d) As an alternative to (c), we can directly maximize the log-likelihood of the observed data. Here we estimate the parameters and standard errors using optim() with the BFGS option; compare iterations that EM and BFGS took respectively.

```
#function to derive log-likelihood
probitLoglik <- function(beta, X, y){
  #result from part (a)
  p <- pnorm(X%*%beta, mean=0, sd=1, lower.tail = TRUE, log.p = FALSE)
  #log-likelihood of bernoulli
  return(sum(y*log(p)+(1-y)*log(1-p)))
}

#use the same setup as part (c)
n <- 100
b2 <- 0
b3 <- 0
b0 <- 1
b1 <- 0.3
x1 <- rnorm(n)
x2 <- rnorm(n)
x3 <- rnorm(n)
X <- cbind(1,x1,x2,x3)
XTb <- b0 + b1*x1 + b2*x2 + b3*x3

set.seed(0)
y <- rbinom(n, 1, prob = pnorm(XTb))

linearReg <- lm(y ~ x1 + x2 + x3)
intVal <- as.double(coef(linearReg))

#estimate betas
#trace: print iterations; maxit: maximum iterations; fnscale=-1: flip log-likelihood function
#to maximize
result <- optim(intVal, fn=probitLoglik, gr=NULL, y=y, X=X, method="BFGS",
                control=list(trace=TRUE,maxit=10000,fnscale=-1), hessian=TRUE)

## initial  value 48.823571
## final  value 47.279947
## converged
```

```
print(result)
```

```
## $par
## [1]  0.91668417  0.37950082  0.15954819 -0.02449264
##
## $value
## [1] -47.27995
##
## $counts
## function gradient
##       29        9
##
## $convergence
## [1] 0
##
## $message
## NULL
##
## $hessian
##             [,1]       [,2]        [,3]       [,4]
## [1,] -46.3791516  12.453885  -0.2592151  -1.701736
## [2,]  12.4538847 -34.749242   4.7432181  -6.391891
## [3,]  -0.2592151   4.743218 -50.3761406  -4.206357
## [4,]  -1.7017355  -6.391891  -4.2063567 -42.711122
```

```
#compute standard error
se <- sqrt((-1)*diag(solve(result$hessian)))
print(se)
```

```
## [1] 0.1553120 0.1833925 0.1428308 0.1569269
```

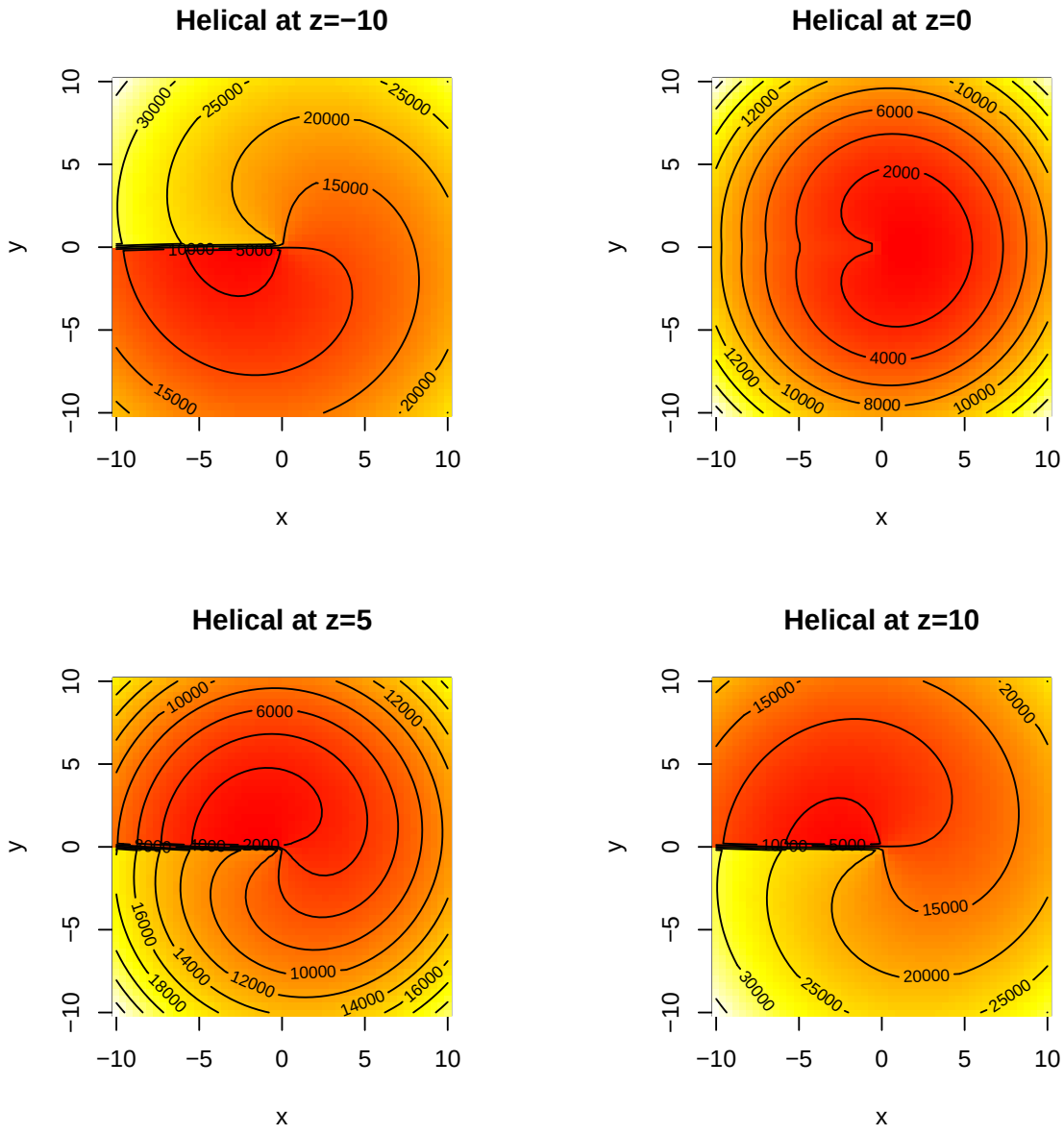
Observe that the BFGS method took 29 iterations while the EM method took 40.

4. Plot slices of the helical valley function at -10, 0, 5, and 10 to get a sense for how it behaves. Explore the possibility of multiple local minima by using different starting points (0,0,0), (-1,-4,7), and (1,2,5).

```
library(fields)
```

```
## Loading required package:  spam
## Loading required package:  grid
## Spam version 1.2-1 (2015-09-30) is loaded.
## Type 'help( Spam)' or 'demo( spam)' for a short introduction
## and overview of this package.
## Help for individual functions is also obtained by adding the
## suffix '.spam' to the function name, e.g.  'help( chol.spam)'.
##
## Attaching package:  'spam'
##
## The following objects are masked from 'package:base':
##
##    backsolve, forwardsolve
##
## Loading required package:  maps
##
##  # ATTENTION: maps v3.0 has an updated 'world' map.        #
```

```
##  # Many country borders and names have changed since 1990.  #
##  # Type '?world' or 'news(package="maps")'.  See README_v3.  #
##
##
##
## Attaching package:  'fields'
##
## The following object is masked from 'package:maps':
##
##    ozone

# helical valley function
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(2, 2), pty = "s")

x <- seq(-10, 10, length.out = 50)
y <- seq(-10, 10, length.out = 50)

# heat map and contour lines at z = -10
val <- apply(as.matrix(expand.grid(x, y)), 1, function(x) f(c(x, -10)))
image(x, y, matrix(val, 50, 50), col = heat.colors(100), axes = TRUE, main = "Helical at z=-10")
contour(x, y, matrix(val, 50, 50), add = TRUE)

# heat map and contour lines at z = 0
val <- apply(as.matrix(expand.grid(x, y)), 1, function(x) f(c(x, 0)))
image(x, y, matrix(val, 50, 50), col = heat.colors(100), axes = TRUE, main = "Helical at z=0")
contour(x, y, matrix(val, 50, 50), add = TRUE)

# heat map and contour lines at z = 5
val <- apply(as.matrix(expand.grid(x, y)), 1, function(x) f(c(x, 5)))
image(x, y, matrix(val, 50, 50), col = heat.colors(100), axes = TRUE, main = "Helical at z=5")
contour(x, y, matrix(val, 50, 50), add = TRUE)

# heat map at and contour lines z = 10
val <- apply(as.matrix(expand.grid(x, y)), 1, function(x) f(c(x, 10)))
image(x, y, matrix(val, 50, 50), col = heat.colors(100), axes = TRUE, main = "Helical at z=10")
contour(x, y, matrix(val, 50, 50), add = TRUE)
```




```
# check that the optimal value changes with starting point
optim(c(0, 0, 0), f, hessian = TRUE)

## $par
## [1] 0.999978292 0.002730698 0.004284640
##
## $value
## [1] 1.876851e-05
##
## $counts
## function gradient
##      110       NA
##
## $convergence
```

```
## [1] 0
##
## $message
## NULL
##
## $hessian
##             [,1]         [,2]         [,3]
## [1,] 200.0023934   -0.8568703    0.8692404
## [2,]  -0.8568703  506.6169955 -318.3143164
## [3,]   0.8692404 -318.3143164  202.0000000
```

```
optim(c(-1, -4, 7), f, hessian = TRUE)
```

```
## $par
## [1] 1.001446916 0.003307424 0.005476581
##
## $value
## [1] 0.0002457851
##
## $counts
## function gradient
##      154       NA
##
## $convergence
## [1] 0
##
## $message
## NULL
##
## $hessian
##             [,1]         [,2]        [,3]
## [1,] 200.0028697   -0.9387944    1.049735
## [2,]  -0.9387944  505.4235928 -317.846412
## [3,]   1.0497353 -317.8464115  202.000000
```

```
optim(c(1, 2, 5), f, hessian = TRUE)
```

```
## $par
## [1]  1.000200654 -0.001440543 -0.002272173
##
## $value
## [1] 9.270966e-06
##
## $counts
## function gradient
##      154       NA
##
## $convergence
## [1] 0
##
## $message
## NULL
##
## $hessian
##             [,1]         [,2]         [,3]
```

```
## [1,] 200.0006541    0.4477356   -0.4583547
## [2,]   0.4477356  506.4401499 -318.2452626
## [3,]  -0.4583547 -318.2452626  202.0000000
```

```
nlm(f, c(0, 0, 0), hessian = TRUE)
```

```
## $minimum
## [1] 100
##
## $estimate
## [1] 0 0 0
##
## $gradient
## [1] -12500000         0         0
##
## $hessian
##              [,1]         [,2]     [,3]
## [1,]          200 -46873828427        0
## [2,] -46873828427 -62499999800 -5000000
## [3,]            0     -5000000      202
##
## $code
## [1] 3
##
## $iterations
## [1] 1
```

```
nlm(f, c(-1, -4, 7), hessian = TRUE)
```

```
## $minimum
## [1] 1.697476e-08
##
## $estimate
## [1]  9.999995e-01 -8.215374e-05 -1.299436e-04
##
## $gradient
## [1]  1.794915e-06 -3.942163e-06  2.749762e-06
##
## $hessian
##              [,1]          [,2]          [,3]
## [1,] 200.00000211   -0.01521464   -0.02614776
## [2,]  -0.01521464  506.60631527 -318.31004293
## [3,]  -0.02614776 -318.31004293  202.00000000
##
## $code
## [1] 2
##
## $iterations
## [1] 22
```

```
nlm(f, c(1, 2, 5), hessian = TRUE)
```

```
## $minimum
## [1] 1.357896e-18
##
```

```
## $estimate
## [1] 1.000000e+00 3.632514e-10 5.900017e-10
##
## $gradient
## [1] -1.995701e-08 -3.778056e-09  3.553825e-09
##
## $hessian
##               [,1]         [,2]          [,3]
## [1,]  2.000000e+02   -0.0406541  1.156149e-07
## [2,] -4.065410e-02  506.6058982 -3.183099e+02
## [3,]  1.156149e-07 -318.3098852  2.020000e+02
##
## $code
## [1] 1
##
## $iterations
## [1] 29
```