Problem Set 4

Thibault Doutre, ID : 26980469

STAT 243 : Introduction to Statistical Computing

I worked on my own.

1 Problem 1

1.1 Question a

The problem comes from the difference between global and local variables.

```
set.seed(0)
runif(1)

## [1] 0.8966972

# save the seed into a file
save(.Random.seed, file = 'tmp.Rda')
# result
runif(1)

## [1] 0.2655087

load('tmp.Rda')
#same result than before
runif(1)

## [1] 0.2655087
```

Now, we debug the code by printing out the assertion of the equality between the random seed immediately after having loaded the file and the random seed of the global environment. And we see that it is false.

```
# Debug
tmp <- function() {
  load('tmp.Rda')
  print(identical(.Random.seed,.GlobalEnv$.Random.seed))
  runif(1)
}
tmp()
```

```
## [1] FALSE

## [1] 0.3721239
```

To correct the code, I load the random seed in the global environment.

```
tmp <- function() {
  load('tmp.Rda',envir = globalenv())
  runif(1)
}
# Expected result
tmp()

## [1] 0.2655087
```

2 Problem 2

First, set the libraries, the global variables and the wd.

```
setwd("/Users/doutre/Documents/stat243/ps4")
library(fields)

## Loading required package: spam
## Loading required package: grid
## Spam version 1.0-1 (2014-09-09) 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
##
## Attaching package: 'fields'
##
## The following object is masked from 'package:maps':
##
##    ozone
```

```
library(rbenchmark)

## Global variables
p = 0.3
 = 0.5
```

2.1 Question a

In order to compute the sum, I first create the function f(k,n,p,phi) and then apply it for all k in an other function, using lapply. We have to do the computations in the log scale because R cannot handle operations on too large numbers. Therefore, it is more appropriate to take the exponential of the log of the product of terms in the expression of f(k,n,p,phi).

```
log_f = function(k,n,p,){
  if (k==0 || k==n)
    return((n-k)**log(1-p)+ k**log(p) )
  else
    return(lchoose(n,k)+
             (1-)*(k*log(k) + (n-k)*log(n-k) - n*log(n))+
            *(k*log(p) + (n-k)*log(1-p)))
}

# using lapply
sum_f = function(n,p,){
  l = as.list(0:n)
  fk = lapply(l,function(k) exp(log_f(k,n,p,)))
  return(Reduce("+",fk))
}
# Example
sum_f(200,p,)

## [1] 1.416484
```

2.2 Question b

Now, I do the calsulation in the vectorized way. The code is a bit messy because it is optimized. But basically, here are the tips:

- Write the formula in the most factorized way
- Do not do twice the same computation, instead store it into a variable
- lgamma is better than lfactorial or lchoose
- crossprod is very efficient for summing a vector

It is important to notice that the cases k=0 and k=n are added at the last moment and the vectors used before are of length n-1.

```r
sum_f_vect = function(n,p,){
  ones=matrix(1,nrow=n-1,ncol=1)
  v=1:(n-1)
  n_v= -v
  n_and_n_v=(n+n_v)
  phi_phiBis=exp(-log(1/-1))
  s_bounds = exp((lgamma(n+1)-(lgamma(v+1)+lgamma(n+n_v+1)))+
                  (1-)*(v*(log(v)+log(p)*phi_phiBis) +
                  n_and_n_v*(log(n_and_n_v)+phi_phiBis*log(1-p)) -
                  n*log(n)))
  return( exp((n*)*log((1-p)*p))+crossprod(s_bounds,ones))
}
# Example
sum_f(200,p,)

## [1] 1.416484
```

Here is the performances:

```r
Rprof("sum_f.txt",interval = 1e-5)
out=sum_f_vect(1e7,p,)
Rprof(NULL)
summaryRprof("sum_f.txt")$by.self

##                self.time self.pct total.time total.pct
## "+"                0.153    42.78      0.153     42.78
## "matrix"           0.111    31.07      0.111     31.07
## "sum_f_vect"       0.055    15.42      0.358     99.98
## ":"                0.019     5.31      0.019      5.31
## "-"                0.017     4.73      0.017      4.73
## "lgamma"           0.002     0.58      0.002      0.58

benchmark(
  out = sum_f(200,p,) ,
  out_vect = sum_f_vect(200,p,),
  replications = 1000,
  columns=c('test', 'elapsed', 'replications'))

##       test elapsed replications
## 1      out   1.060         1000
## 2 out_vect   0.052         1000
```

We can see that the vectorized way is 100 times more efficient than the naive approach.

3 Problem 3

First of all, let's set the wd, load some libraries and the data provided. I also set a global variable, n, which is the length of IDsA or IDsB (they are equal).

```r
setwd("/Users/doutre/Documents/stat243/ps4")
load("./mixedMember.Rda")
library(rbenchmark)
library(fields)
library(plyr)

##
## Attaching package: 'plyr'
##
## The following object is masked from 'package:fields':
##
##     ozone
##
## The following object is masked from 'package:maps':
##
##     ozone

library(Matrix)
library(microbenchmark)

## Global variables
n=length(IDsA) #IDsB has the same length
```

3.1 Question a

In one line, I compute the sum for cases A and B, and do a benchmark to see the average speed on 10 replications.

```r
# For A
outA = sapply(1:n,function(i) sum(wgtsA[[i]]*muA[IDsA[[i]]]))
# For B
outB = sapply(1:n,function(i) sum(wgtsB[[i]]*muB[IDsB[[i]]]))

# speed calculation
benchmark(
  outA=sapply(1:n,function(i) sum(wgtsA[[i]]*muA[IDsA[[i]]])),
  outB=sapply(1:n,function(i) sum(wgtsB[[i]]*muB[IDsB[[i]]])),
  replications = 10,
  columns=c('test', 'elapsed', 'replications'))
```

```
##    test elapsed replications
## 1 outA   1.635           10
## 2 outB   1.710           10
```

3.2 Question b and c

The basic idea of what I have implemented is based on the trace of a product of matrices. Indeed, we can easily see that the sum is the trace of the matrix product of MU and W, with MU and W well defined. Seeing things this way, it is really fast to compute the sums which are the sums over the lines of MU*W (* is the Hilbert product for matrices, just like in R).
But is is important to notice that MU depends on mu, and that it is unfortunate since we want to transform the data, setting aside mu.
So, this is how I proceed:

– Transform IDs data so each colum indicates the index of mu. In other words each column has ones at the indices of IDs.

```
# Transform function : add zeros in the middle
transform_ID = function(l,max.len){
  M=matrix(rep(0,max.len*length(l)),nrow=max.len)
  for (i in 1:length(l)){
    li=l[[i]]
    M[li,i]=1
  }
  return(M)
}
# Example
transform_ID(IDsB,length(muB))[,1:3]
```

```
##      [,1] [,2] [,3]
## [1,]    0    0    0
## [2,]    1    0    0
## [3,]    1    1    1
## [4,]    1    1    1
## [5,]    1    0    0
## [6,]    1    0    0
## [7,]    1    0    0
## [8,]    1    0    1
##   [9,]     1     0     1
##  [10,]     1     1     0

# Original data
head(IDsB,3)

## [[1]]
## [1]  2  7  6  3  4  5  9  8 10
##
## [[2]]
## [1] 10  3  4
##
## [[3]]
## [1] 3 9 4 8
```

– Transform the matrix of weigths such that the size of the matrix is the same as the transformed IDs and each ones in this latter is filled with corresponding weights values.

```
# Transform W
transform_W = function(l,ID_transformed){
  M=ID_transformed
  for (i in 1:length(l)){
    li=l[[i]]
    M[,i][M[,i] != 0]=li
  }
  return(M)
}
# Example
transform_W(wgtsB,transform_ID(IDsB,length(muB)))[,1:3]

##              [,1]       [,2]      [,3]
##  [1,] 0.000000000 0.00000000 0.0000000
##  [2,] 0.226200194 0.00000000 0.0000000
##  [3,] 0.469188914 0.81031321 0.2653353
##  [4,] 0.011783225 0.15188043 0.3655552
##  [5,] 0.058049207 0.00000000 0.0000000
##  [6,] 0.078030098 0.00000000 0.0000000
##  [7,] 0.005137179 0.00000000 0.0000000
##  [8,] 0.049490784 0.00000000 0.1903563
##  [9,] 0.018964598 0.00000000 0.1787532
## [10,] 0.083155801 0.03780636 0.0000000
```

```r
# Original data
head(wgtsB,3)

## [[1]]
## [1] 0.226200194 0.469188914 0.011783225 0.058049207
## [5] 0.078030098 0.005137179 0.049490784 0.018964598
## [9] 0.083155801
##
## [[2]]
## [1] 0.81031321 0.15188043 0.03780636
##
## [[3]]
## [1] 0.2653353 0.3655552 0.1903563 0.1787532
```

– Now, we may be tempted to compute the Trace of the product of matrices issued from these two functions. But when the IDs are mapped in a special way, so we would have to correctly assign the mus to the IDs functions. But we do not want to do that, otherwise we incorpore mu in our data, which is forbidden. So the trick is to shuffle the elements of every column of the matrix W so that the product with MU would be the same than assigning different values according to the IDs.

```r
# Do the permutation on W instead of mu
permutation_W = function(wgtsB_transformed,ID){
  l=wgtsB_transformed
  for (i in 1:length(ID)){
    l[,i][ID[[i]]]=wgtsB_transformed[,i][ID[[i]][order(ID[[i]])]]
  }
  return(l)
}
# Example
permutation_W(transform_W(wgtsB,
                        transform_ID(IDsB,length(muB))),IDsB)[,1:3]

##              [,1]       [,2]      [,3]
##  [1,] 0.000000000 0.00000000 0.0000000
##  [2,] 0.226200194 0.00000000 0.0000000
##  [3,] 0.058049207 0.15188043 0.2653353
##  [4,] 0.078030098 0.03780636 0.1903563
##  [5,] 0.005137179 0.00000000 0.0000000
##  [6,] 0.011783225 0.00000000 0.0000000
##  [7,] 0.469188914 0.00000000 0.0000000
##  [8,] 0.018964598 0.00000000 0.1787532
##  [9,] 0.049490784 0.00000000 0.3655552
## [10,] 0.083155801 0.81031321 0.0000000

# Permutation according to
head(IDsB,3)

## [[1]]
## [1]  2  7  6  3  4  5  9  8 10
##
## [[2]]
## [1] 10  3  4
##
## [[3]]
## [1] 3 9 4 8
```

– Finally, I compute the Trace of the product of the two matrices. But we can do it more efficiently by multiplying the vector mu to the new permutted data. The "*" operator for a matrix and a vector does the work just fine. I could obviously do the product of the permutted data and the product of mu times the transformed IDs data, but it would have been a bit longer.

```
vect_sum = function(new_data, mu){
  ones= matrix(1,nrow=length(mu),ncol=1)
  product=new_data*mu
  res=crossprod(ones,product)
  return(res)
}
# Example
vect_sum(permutation_W(
  transform_W(
    wgtsB,transform_ID(IDsB,length(muB))),
  IDsB),muB)[1:3]

## [1] -0.4496267 -0.3697111 -0.2104093
```

– Now, I create a function which takes into argument the data provided and returns the result of the previous function, for clarity.

```
# Normal matrices, for B
transform_data = function(wgts,IDs,max.len){
  ID_transformed=transform_ID(IDs,max.len)
  wgts_transformed=transform_W(wgts,ID_transformed)
  new_data=permutation_W(wgts_transformed,IDs)
  return(new_data)
}
```

The danger with this approach however is when the mu vector is too large. Indeed, for B it was useful since length(muB) was equal to ten, but with case A, we need another approach since length(muA) equals 1000.
To deal with this problem, we have to be aware of the fact that the max of mi is small. So our transformed weight matrix has a lot of zeros in it. This leads to the use of sparse matrices. I define two new functions for sparse matrices, which can be used when mu is too large.

```
# Deal with sparse matrices
transform_data_sparse = function(wgts,IDs,max.len){
  ID_transformed=transform_ID(IDs,max.len)
  wgts_transformed=transform_W(wgts,ID_transformed)
  new_data=permutation_W(wgts_transformed,IDs)
  new_data_sparse_bis=as(new_data,"sparseMatrix")
  new_data_sparse=Matrix(new_data_sparse_bis, sparse = TRUE)
  return(new_data_sparse)
}
# Deal with sparse matrices
vect_sum_sparse = function(new_data, mu){
  ones= Matrix(1,nrow=length(mu),ncol=1,sparse = TRUE)
  product=new_data*mu
  res=crossprod(ones,product)
  return(res)
}
```

These two functions work the same way as the previous ones, except for the fact that I use the library 'Matrix', which is efficient when dealing with sparse matrices.

3.3 Question d

Now, let's compute the speed test. First, I load data using sparse functions or not, depending on dealing with A or B.

```
new_data_B=transform_data(wgtsB,IDsB,length(muB))
new_data_A=transform_data_sparse(wgtsA,IDsA,length(muA))
```

Then, I compute the efficiency of functions using three different methods, as requested:

```
# speed calculation with benchmark
benchmark(
  outA = sapply(1:n,function(i) sum(wgtsA[[i]]*muA[IDsA[[i]]])),
  outB = sapply(1:n,function(i) sum(wgtsB[[i]]*muB[IDsB[[i]]])),
  outA_vect=vect_sum_sparse(new_data_A,muA),
  outB_vect=vect_sum(new_data_B,muB),
  replications = 10,
  columns=c('test', 'elapsed', 'replications'))
```

```
##         test elapsed replications
## 1       outA   1.526           10
## 3  outA_vect   0.200           10
## 2       outB   1.552           10
## 4  outB_vect   0.052           10
```

```
# speed calculation with microbenchmark
microbenchmark(
  outA = sapply(1:n,function(i) sum(wgtsA[[i]]*muA[IDsA[[i]]])),
  outB = sapply(1:n,function(i) sum(wgtsB[[i]]*muB[IDsB[[i]]])),
  outA_vect=vect_sum_sparse(new_data_A,muA),
  outB_vect=vect_sum(new_data_B,muB),
  times = 10L
)
```

```
## Unit: milliseconds
##        expr        min        lq       mean    median
##        outA 142.379152 152.41818 173.082492 157.81374
##        outB 143.195879 152.98294 164.324361 161.99015
##   outA_vect  14.175109  14.99821  18.394205  19.54872
##   outB_vect   2.003604   2.92446   3.691835   3.32164
##          uq        max neval
##  172.954434 290.484015    10
##  173.433724 198.933045    10
##   19.656025  21.864156    10
##    4.761664   6.120072    10
```

```
# speed calculation with system.time
print("outA")
```

```
## [1] "outA"
```

```
system.time(sapply(1:n,function(i) sum(wgtsA[[i]]*muA[IDsA[[i]]])))
```

```
##     user  system elapsed
##    0.152   0.000   0.152

print("outB")

## [1] "outB"

system.time(sapply(1:n,function(i) sum(wgtsA[[i]]*muA[IDsA[[i]]])))

##     user  system elapsed
##    0.145   0.003   0.146

print("outA_vect2")

## [1] "outA_vect2"

system.time(vect_sum_sparse(new_data_A,muA))

##     user  system elapsed
##    0.017   0.004   0.022

print("outB_vect2")

## [1] "outB_vect2"

system.time(vect_sum(new_data_B,muB))

##     user  system elapsed
##    0.003   0.001   0.005
```

The orders of magnitudes correspond to the values expected.

4 Problem 4

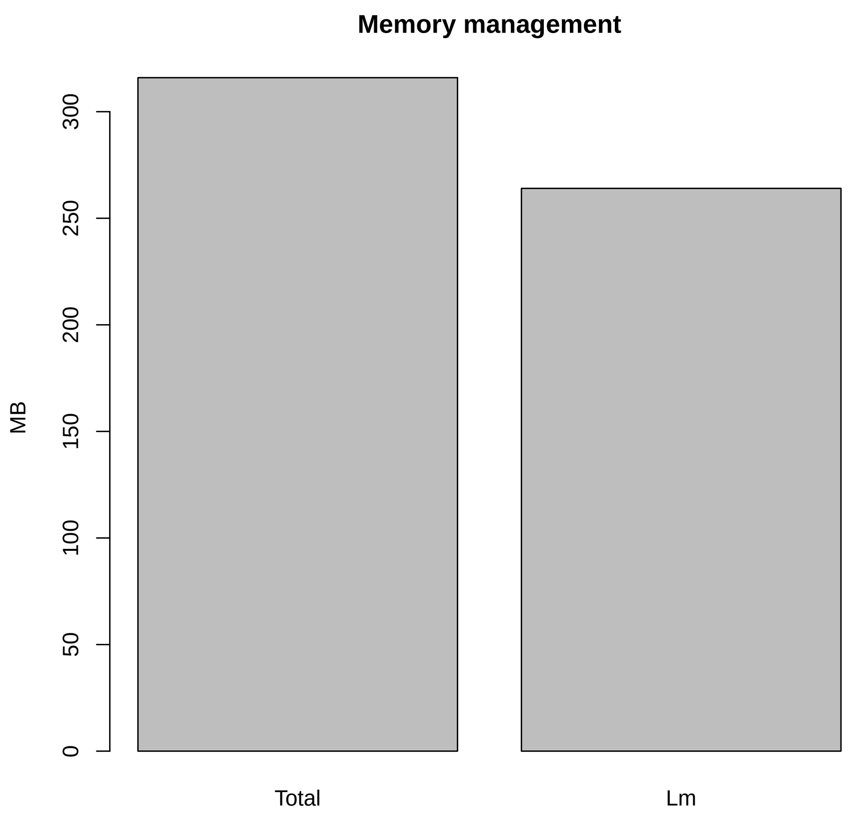
First of all let's load all the objects we are interested in.

```
library(pryr)

# Clear objects from workspace
rm(list=ls(all=TRUE))
```

```r
# Store data
df=as.data.frame(matrix(c(rnorm(1e6),rnorm(1e6,3,1),rnorm(1e6,1,.2),rnorm(1e6,-1,2)),ncol=4))
names(df)=c("y","x1","x2","x3")
lm.fit=lm(y˜.,data = df)
```

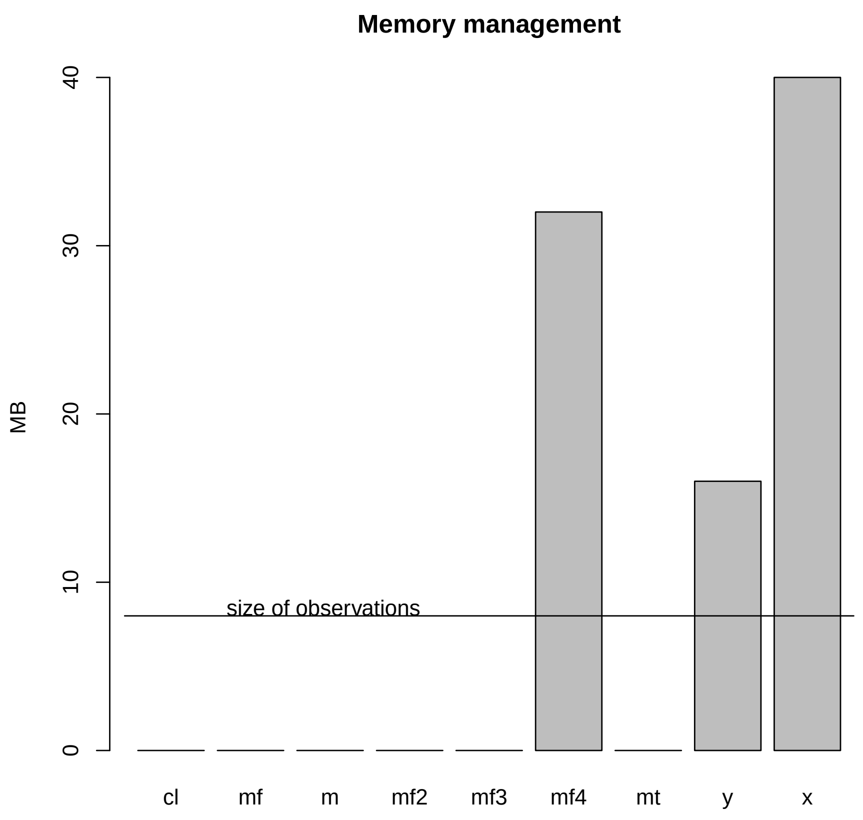
Here is the memory use for the object lm.fit.

```r
## a)
t_m=mem_used()
lm_size=object.size(lm.fit)
barplot(c(t_m,lm_size)/1e6,
        names.arg=c("Total","Lm"),
        ylab="MB",
        main="Memory management")
```



Then, I write the lm function again for our specific problem; basically I remove useless statements. Then, by storing the memory use of each operation, I plot the result with barplot.

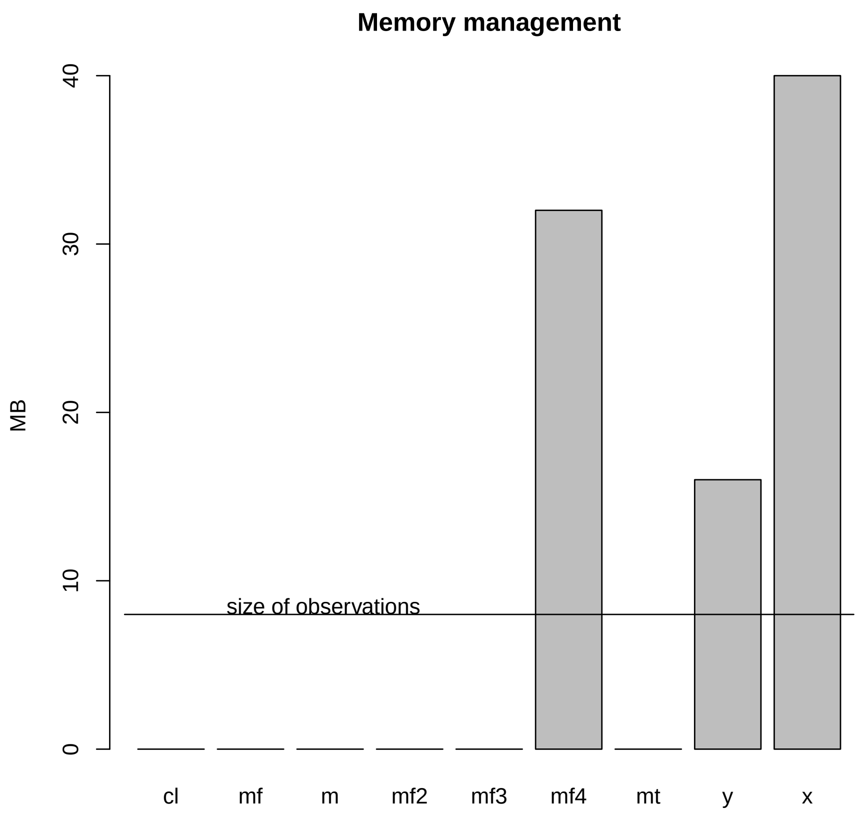
```r
# Size of observations
obs_size=object.size(df$y)/1e6
# Lm function for our special case
lm2 = function (formula, data)
{
  # Create a new variable which store values of memory used
  mem_management=rep(0,9)
  mem_names=c("cl","mf","m","mf2","mf3","mf4","mt","y","x")
  # Total memory used before storing the variables
  t_m=mem_used()
  # Local variables
  # 1 : cl
  cl <- match.call()
  mem_management[1]=mem_used()-t_m
  t_m=mem_used()
  # 2 : mf1
  mf <- match.call(expand.dots = FALSE)
  mem_management[2]=mem_used()-t_m
  t_m=mem_used()
  # 3 : m
  m <- match(c("formula", "data"), names(mf), 0L)
  mem_management[3]=mem_used()-t_m
  t_m=mem_used()
  # 4 : mf2
  mf <- mf[c(1L, m)]
  mem_management[4]=mem_used()-t_m
  t_m=mem_used()
  # 5 : mf3
  mf[[1L]] <- quote(stats::model.frame)
  mem_management[5]=mem_used()-t_m
  t_m=mem_used()
  # 6 : mf4
  mf <- eval(mf, parent.frame())
  mem_management[6]=mem_used()-t_m
  t_m=mem_used()
  # 7 : mt
  mt <- attr(mf, "terms")
  mem_management[7]=mem_used()-t_m
  t_m=mem_used()
  # 8 : y
  y <- model.response(mf, "numeric")
  mem_management[8]=mem_used()-t_m
  t_m=mem_used()
  # 9 : x
  x <- model.matrix(mt, mf)
  mem_management[9]=mem_used()-t_m
  t_m=mem_used()
  # Plot memory management
  barplot(mem_management/1e6,names.arg = mem_names,ylab="MB",
          main="Memory management")
  lines(x=0:11,y=rep(obs_size,12))
  text(x=3,y=obs_size+.5 ,labels = c("size of observations"))

  # lm.fit call
  z <- lm.fit(x, y, singular.ok = TRUE)
  class(z) <- c(if (is.matrix(y)) "mlm", "lm")
  z$xlevels <- .getXlevels(mt, mf)
  z$call <- cl
  z$terms <- mt
  z$model <- mf
  list(z,mem_management)
}
lm2(y~.,data = df)
```

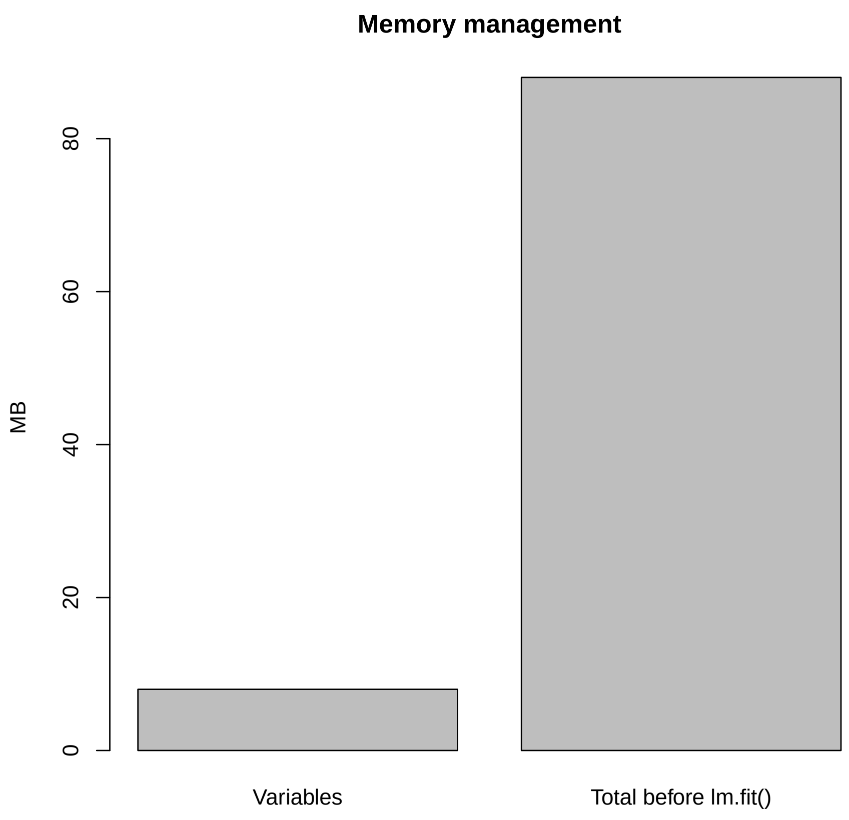


```
## [[1]]
##
## Call:
## lm2(formula = y ~ ., data = df)
##
## Coefficients:
## (Intercept)           x1           x2           x3
##   0.0021041    0.0011744   -0.0048279   -0.0002561
##
##
## [[2]]
## [1]      -416      392      232        0      112
   32003112
## [7]       112 16000280 40000688
```

If figure out how much total memory is in use in lm() at the point at which lm.fit() is called and I compare the additional memory use to the memory used in the global environment to store the observations and covariates. The latter is given by the sum of the second attribute of lm2 function.

```r
barplot(c(object.size(df$y),sum(lm2(y~.,data = df)[[2]]))/1e6,
        names.arg = c("Variables","Total before lm.fit()"),
        ylab="MB",main="Memory management")
```



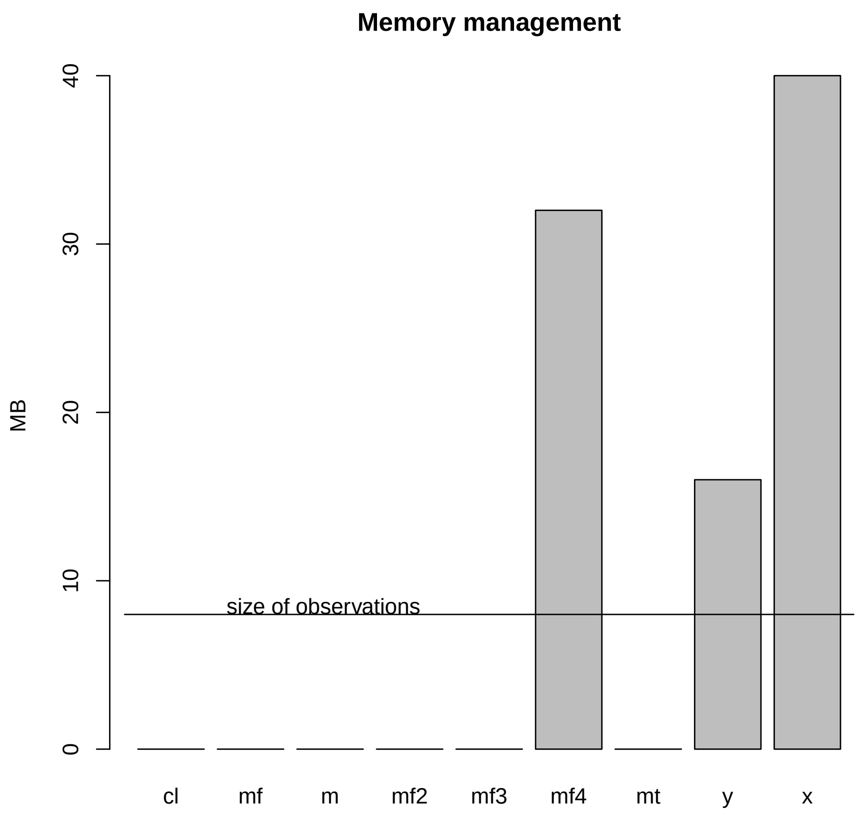


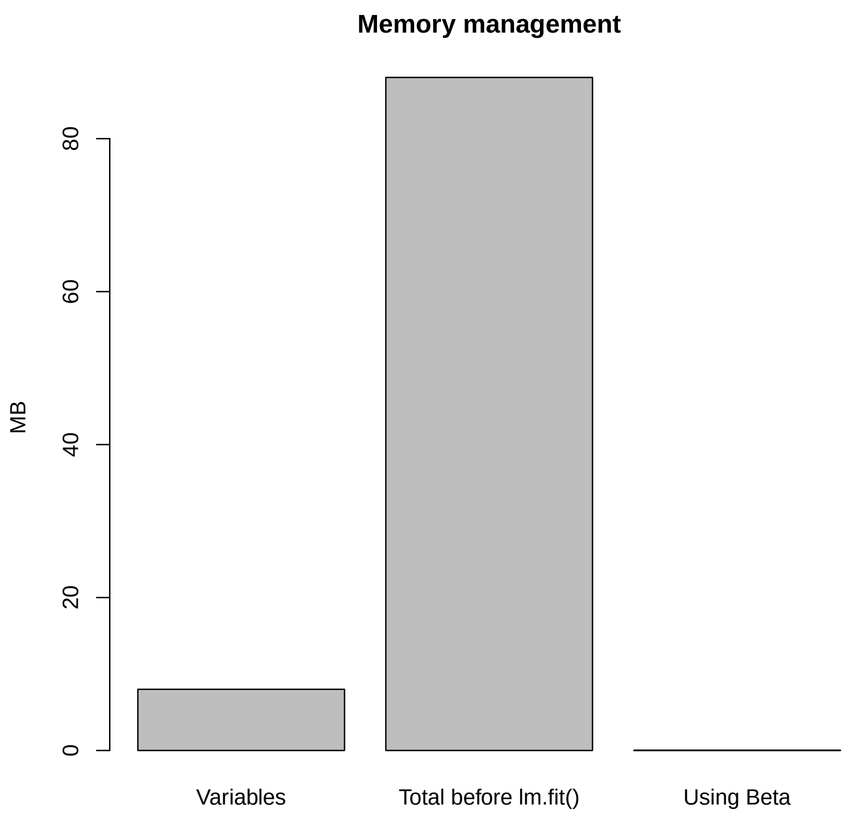
An explanation of why some of the vectors and matrices are larger than 8 bytes per number, is that extra memory is used to store compute other variables such as the residuals and the fitted values.
To optimize memory used, we can directly compute the result of the least square estimate:

```r
x=as.matrix(df[,2:4])
y=as.matrix(df[,1])
t_m=mem_used()
Beta = solve(crossprod(x,x), crossprod(x,y))
beta_mem=mem_used()-t_m
# New value
beta_mem
```

```
## 16.3 kB
```

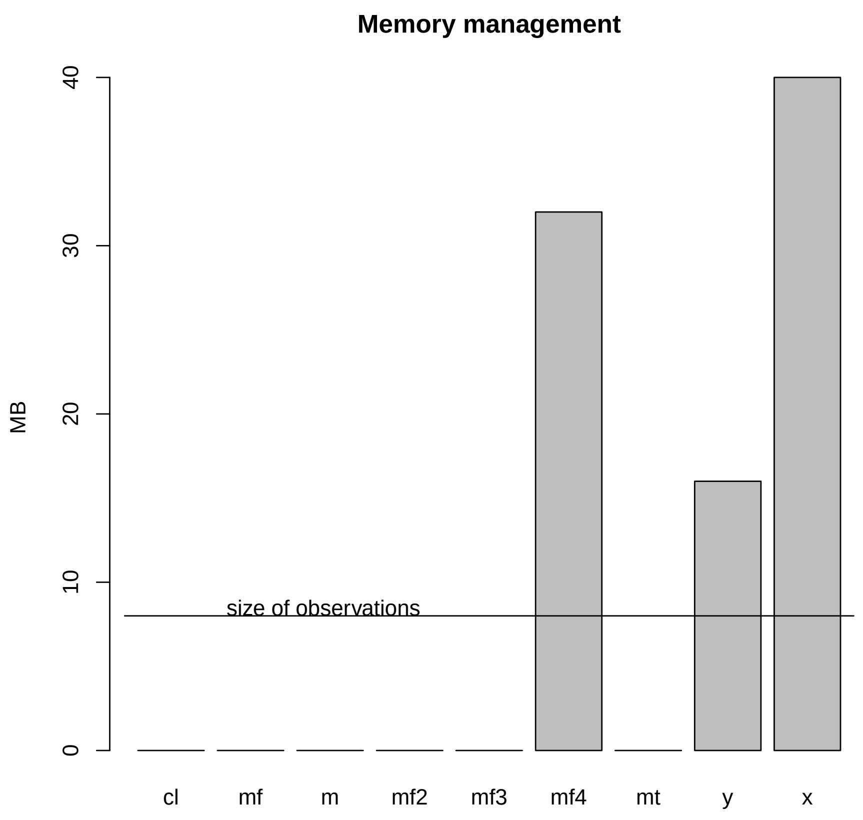
```
# Plot
barplot(c(object.size(df$y),sum(lm2(y~.,data = df)[[2]]),beta_mem)/1e6,
        names.arg = c("Variables","Total before lm.fit()","Using Beta"),
         ylab="MB",main="Memory management")
```





Now, we see that the memory used to store Beta is almost nothing compared to the data used to compute lm(). However, by doing this, we do not have directly access to relevant information such as residuals or the fitted values. We can see that the results are very close to each other:

```r
# Using lm2
lm2(y~.,data = df)[[1]]$coefficients[2:4]
```



```
##             x1            x2            x3
##  0.0011743658 -0.0048279397 -0.0002560523
```

```r
# Using LSE Beta
t(Beta)
```

```
##               x1           x2            x3
## [1,] 0.00135383 -0.003337436 -0.0002712319
```