# PS4 - STAT 243

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

October 12th 2015

## 1 Problem 1

### 1.1 1. a)

.Random.seed[2] gives information about where in the periodic sequence of random numbers R is located. Everytime a new random number is generated that value will increment by one.

R is using lexical scoping and defines variables inside functions, but it can also use variables from the enclosing environment through scoping. If you look at the documentation for the *.Random.seed* function it says that:
`The object .Random.seed is only looked for in the user's workspace.`
Therefore what will happen is that when we load in the state of the seed with $load('tmp.Rda')$ it will only be set locally inside the function, but it will not be changed in the Global environment (the user's workspace). However when we call $runif(1)$, it will only change the state of .Random.seed in the Global environment. To be figure out the problem the R debugging tool *debug*() was used on the function *tmp*(). With debug the function can be evaluated one line of code at a time in a browser, and one can then carry out analysis sequentially for all the different states that R is in within the function. To see a printout of what functions was used in the *tmp*() function when debugging that also led to the conclusion above see Figure 1.

```
> debug(tmp())
debugging in: tmp()
debug at #1: {
    load("tmp.Rda")
    runif(1)
}
Browse[2]> .Random.seed[2]
[1] 2
Browse[2]> get('.Random.seed',envir = .GlobalEnv)[2]
[1] 2
Browse[2]>
debug at #2: load("tmp.Rda")
Browse[2]>
debug at #3: runif(1)
Browse[2]> .Random.seed[2]
[1] 1
Browse[2]> get('.Random.seed',envir = .GlobalEnv)[2]
[1] 2
Browse[2]> runif(1)
[1] 0.3721239
Browse[2]> .Random.seed[2]
[1] 1
Browse[2]> get('.Random.seed',envir = .GlobalEnv)[2]
[1] 3
```

Figure 1: Debugging of tmp(), where the local and global value of .Random.seed is checked in several steps

To fix the problem we need R to load in the state of the random seed into the global environment. This can be done by specifying what environment the function *load*() inside the *tmp*() function should load the data into. In this case we want to change the state of the random seed in the Global enivornment in order to read in the state and continue the analysis from where we left off. See revised, working code below:

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

## [1] 0.8966972

## [1] Ok should be 0.8966972, save state
save(.Random.seed, file = 'tmp.Rda')
runif(1)

## [1] 0.2655087

## [1] 0.2655087 Ok
load('tmp.Rda')
runif(1)

## [1] 0.2655087

## [1] 0.2655087 Ok
tmp <- function() {
  load('tmp.Rda',envir=.GlobalEnv)  # Load to global enivornment
  runif(1)

}

tmp()

## [1] 0.2655087

## Ok! This gives the correct result [1] 0.2655087
#Instead of [1] 0.3721239 when we did not specify envir= for load
```

# 2 Problem 2

## 2.1 2. a)

First the analysis was setup and the log denominator calculated with the function *sapply_denom* using sapply. If we did not log the results before starting the analysis the terms in the expression with $n$, $k$ and $\Phi$ in the beginning would be so much greater (resulting in $Inf$ and $NaN$ since we have terms like $n^n$ which can be greater than $10^{309}$, which equals to inf in R, when $n \geq 144$) then the last terms in the expression with $p$ and $1-p$ to the power of $n$, $k$ and $\Phi$ that the result would be inaccurate (because of R's way of storing numbers and results).
To handle the calculations the argument for the log denominator was split up into three cases to also handle the two limit cases (k=0 first term and (n-k)=0 last term):

$\lim_{(k)\to 0}(k)log(k) = 0.$

$\lim_{(n-k)\to 0}(n-k)log(n-k) = 0.$

See the result below for $n = 10$

```
#Define preliminary constants
p=0.3
Phi=0.5
n=10 # only for testing

# Define denominator expression without evaluating, k!=0 & (n-k)!=0
logDenom<-quote(lchoose(n,k)+k*log(k)+(n-k)*log(n-k)-
                  n*log(n)+n*Phi*log(n)-k*Phi*log(k)-Phi*(n-k)*log(n-k)+
                  k*Phi*log(p)+(n-k)*Phi*log((1-p)))

# Define first term in denominator k=0, (xlog(x)->0 when x->0)
logDenomFirst<-quote(lchoose(n,k)+(n-k)*log(n-k)-
                       n*log(n)+n*Phi*log(n)-Phi*(n-k)*log(n-k)+
                       k*Phi*log(p)+(n-k)*Phi*log((1-p)))

# Define last denominator k=n, (n-k)=0, (xlog(x)->0 when x->0)
logDenomLast<-quote(lchoose(n,k)+
                      k*log(k)-n*log(n)+n*Phi*log(n)-k*Phi*log(k)+
                      k*Phi*log(p)+(n-k)*Phi*log((1-p)))


# Place expressions in a list to only reference to one object
denomV=c(logDenomFirst,logDenom,logDenomLast)


sapply_denom <- function(n,denomV) {
  logRes=rep(0,n+1)
  logRes[1]<-sapply(0, function(k) eval(denomV[[1]])) #eval first term k=0
  logRes[2:(n)]<-sapply(1:(n-1), function(k) eval(denomV[[2]])) #eval terms k=1:(n-1)
  logRes[n+1]<-sapply(n, function(k) eval(denomV[[3]])) #eval last term k=n
  return(sum(exp(logRes))) #return the log-inverse and summed result
}

sapply_denom(n,denomV)

## [1] 1.475851
```

## 2.2 2. b) (and hopefully c) )

Then a vectorized analysis was setup, using only vectorized calculations within the function *vector_denom*. Here again the denominator was split up into the same three cases because of the limit.
After that a comparison between the system.time and benchmark results for both functions was evaluated inside a for loop for the n-vales: 20, 200, 2000, 20000.
As can be seen from the results the vectorized calculation seem to be a faster with almost an order magnitude of 3, $\mathcal{O}(3)$ (this is obvious for for $n = 20000$, evaluated as the last term).

```
#Define preliminary constants

vector_denom <- function(n,denomV) {
```

```
  logRes=rep(0,n+1)
  k=0
  logRes[1]<-eval(denomV[[1]]) #eval first term k=0
  k=1:(n-1)
  logRes[2:n]<-eval(denomV[[2]]) #eval first k=(1:n-1) terms
  k=n
  logRes[n+1]<-eval(denomV[[3]]) #eval last term k=n
  return(sum(exp(logRes))) #return non-logged summed result
}

vector_denom(n,denomV)

## [1] 1.475851

for (i in 1:3) {
  n=2*10^i # set n
  print(paste("The time for SAPPPLY with n=", n))
  print(system.time(sapply_denom(n,denomV)))
  print(paste("The time for VECTOR MULTIPLICATION with n=", n))
  print(system.time(vector_denom(n,denomV)))

  print(paste("BENCHMARK RESULTS with n=",n))
  print(benchmark(
  sapply_denom(n,denomV),
  vector_denom(n,denomV),
  replications = 10, columns=c('test', 'elapsed', 'replications')
  ))
}

## [1] "The time for SAPPPLY with n= 20"
##    user  system elapsed
##   0.001   0.000   0.000
## [1] "The time for VECTOR MULTIPLICATION with n= 20"
##    user  system elapsed
##       0       0       0
## [1] "BENCHMARK RESULTS with n= 20"
##                       test elapsed replications
## 1 sapply_denom(n, denomV)   0.003           10
## 2 vector_denom(n, denomV)   0.001           10
## [1] "The time for SAPPPLY with n= 200"
##    user  system elapsed
##   0.002   0.000   0.002
## [1] "The time for VECTOR MULTIPLICATION with n= 200"
##    user  system elapsed
##       0       0       0
## [1] "BENCHMARK RESULTS with n= 200"
##                       test elapsed replications
## 1 sapply_denom(n, denomV)   0.024           10
## 2 vector_denom(n, denomV)   0.001           10
## [1] "The time for SAPPPLY with n= 2000"
##    user  system elapsed
##   0.019   0.005   0.023
## [1] "The time for VECTOR MULTIPLICATION with n= 2000"
##    user  system elapsed
```

```
##   0.000   0.000   0.001
## [1] "BENCHMARK RESULTS with n= 2000"
##                       test elapsed replications
## 1 sapply_denom(n, denomV)   0.271           10
## 2 vector_denom(n, denomV)   0.005           10
```

# 3 Problem 3

## 3.1 3. a)

First one line of code with sapply was used to calculate the mean expression in the normal distribution. See results below.

```
#Problem 3

load('mixedMember.Rda') #load in the variable lists / arrays

sapply_mu = function (mu,wgts,IDs) {
  #perform calculations for the right weights and mu indicies, place results in a vector
  #ever array element in result is the result for one person
  result <- sapply( 1:length(wgts) , function(i) sum(wgts[[i]]*mu[ IDs[[i]] ]))
  return(result)
}

personsA<-sapply_mu(muA,wgtsA,IDsA) #store results for case A
head(personsA)

## [1] -0.53997057 -0.68233057 -0.40414341 -0.24803496  0.44062079  0.03546354

personsB<-sapply_mu(muB,wgtsB,IDsB) #store results for case A
head(personsB)

## [1] -0.4496267 -0.3697111 -0.2104093 -0.3426966 -0.3874494  0.6585238
```

## 3.2 3. b)

In order to solve problem 3 b) two different solutions was implemented (because of uncertainty in what was allowed to do, and what results was supposed to aimed for in regards to speed). The two different solutions are shown below.

**The first one** creates a $[100000x1000]$ matrix of weights values, placed on the correct indicies for the corresponding places inside the $IDsA$ vector for every row (with the function $mat\_wgtsA1$). The matrix multiplication below could be carried out giving the result directly as an R matrix with 100 000 values in one column vector, and as can be seen this is the same results as in 3 a), when compared with $all.equal()$ for $personsA$.
$[matWgtsA] * [muB]$

Although the first solution is more efficient than the sapply, it is not of and order magnitude of 1 $\prime(1)$,

but it takes about half the time as the sapply solution. Therefore a **Second solution** was implemented for case A where the matrix *matWgtsASparse* is a sparse matrix instead of matrix filled with zeroes. This speeds up the computation A LOT. The function *mat_wgtsA_sparse* takes the matrix created in the first solution and only converts that to a sparse matrix (the zeroes become 'dot' elements, reducing the size of the matrix a lot and it doesn't need to evaluate each zero multiplication when using the matrix multiplication at the end). The results can be seen below, for the second solution it also equals the result in 3 a)

```
#Problem 3 b)

#Sol1

##NB the function mat_wgts will be used for both case A and case B

mat_wgts = function(mu,wgts,IDs) {

  #Pre-allocate matrix [100 000 x 1 000] for case A or [100 000 x 10] for case B
  wgtsM<-matrix(rep(0,length(wgts)*length(mu)),length(wgts),length(mu))

  for (i in 1:length(wgts)) {
    wgtsM[i,IDs[[i]] ]<-wgts[[i]]
    #fill weights in the matrix on the places corresponding to the mu-values of interest
  }
  return(wgtsM)
}


matWgtsA<-mat_wgts(muA,wgtsA,IDsA)

all.equal(personsA,as.vector(matWgtsA%*%muA)) #compare results in 3 a) and 3 b)

## [1] TRUE

#Sol2
library(Matrix) #package required for sparse matrices

mat_wgtsA_sparse = function(wgtsM) {

  #Pre-allocate SPARSE matrix [100 000 x 1 000]
  wgtsMSparse<-Matrix(as.vector(wgtsM),nrow(wgtsM),ncol(wgtsM),sparse=TRUE)

  return(wgtsMSparse)
}

matWgtsASparse<-mat_wgtsA_sparse(matWgtsA)

all.equal(personsA,as.vector(matWgtsASparse%*%muA))

## [1] TRUE
```

### 3.3 3 c)

In order to perform the vecotrized multiplication in part 3 c) a matrix $matWgtsB$ of size $[100000X10]$ was filled up with the wgts-values for $wgtsB$. This is done with the function $mat\_wgts$ (in the same way as in Sol 1 in 3 b) ). Every element is placed in accordance with the structure of the IDsB entries on every row, in order to at the end only need to perform the matrix multiplication below to yield the correct results:
$[wgtsMc] \cdot [muB]$
This will directly result in a column vector where the results for every person stored on every row. Find the implemented solution below:

```
#Problem 3 c)

matWgtsB<-mat_wgts(muB,wgtsB,IDsB)

all.equal(personsB,as.vector(matWgtsB%*%muB)) #compare results in 3 a) and 3 c)

## [1] TRUE
```

### 3.4 3 d)

To compare the results the function benchmark (which is a wrapper around system.time) was used to show the time the computer spent to evaluate the results for each solution implemented in the above subquestions. As can be seen by the results both of the *sapply* solutions for A and B take about the same time. When it comes to the two solutions implemented in subquestion 3 b) for case A the direct vectorized solution $matWgtsA \cdot muA$ takes about half the time of the sapply evaluations, and the sparse matrix solution is approximately an order magnitude of two faster then the sapply. Last the method implemented in 3 c) for case B is the quickest one with an order magnitude of two quicker than the sapply (this could also be a little bit faster if we converted the matrix $matWgtsB$ to a sparse matrix, however we do not have that many zeroes in $matWgtsB$, therefore it would not speed up the process that much).

```
#Problem 3 d)

#quote apply functions for nicer outputs in the benchmark function
applyA<-quote(sapply( 1:length(wgtsA) , function(i) sum(wgtsA[[i]]*muA[ IDsA[[i]] ])))
applyB<-quote(sapply( 1:length(wgtsB) , function(i) sum(wgtsB[[i]]*muB[ IDsB[[i]] ])))

benchmark(
  eval(applyA),
  eval(applyB),
  matWgtsA%*%muA,
  matWgtsASparse%*%muA,
  matWgtsB%*%muB,
  replications = 10, columns=c('test', 'elapsed', 'user.self', 'sys.self' , 'replications')
)

##                      test elapsed user.self sys.self replications
## 1          eval(applyA)   2.606     2.214    0.368           10
## 2          eval(applyB)   2.556     2.220    0.335           10
## 3       matWgtsA %*% muA   1.516     1.509    0.004           10
## 4 matWgtsASparse %*% muA   0.024     0.023    0.000           10
## 5       matWgtsB %*% muB   0.018     0.018    0.000           10
```

# 4 Problem 4

## 4.1 4 a)

In order to find out how much memory was used in lm, an exact copy was made of the function lm called *my_lm_4a* (which can be found in the Github repository). A print statement *print*(*mem_used*()) was inserted just before the lm.fit function was called. This yields the result below.

**NB:** This result is for when the knitr document is compiled. A 'clean' R session, when not a lot of memory is being used from evaluating earlier chunks of code, running the commands below result in that we use around 64.4 mb at the start of the session when only the observations and the covariates are loaded in. And then right before lm.fit is called we use 216.6mb of memory. **Therefore this indicates that 152 mb of additional memory is used when running the lm function, right before lm.fit is called.** See Figure 2 for the lines of code that was run in a clean R session.

```
#Problem 4 a)

# Start ----

gc(reset=TRUE)

##             used  (Mb) gc trigger   (Mb)  max used  (Mb)
## Ncells   1855202  99.1    3205452  171.2   1855202  99.1
## Vcells 106272933 810.8  310880796 2371.9 106272933 810.8

source("my_lm_4a.R")
source("my_lm_4b.R")


n=10^6

y<-rnorm(n)
x1<-rnorm(n)
x2<-rnorm(n)
x3<-rnorm(n)

# Analysis for Question 4 a)

paste("Memory used for covariates + knitr session", round(mem_used()/10^6,1),"mb")

## [1] "Memory used for covariates + knitr session 986.2 mb"

# Print memory use at start

m1<-my_lm_4a(y~x1+x2+x3) # prints memory use before call to lm.fit

## [1] "Memory used before lm.fit: 1142.1 mb"

rm(m1)
```

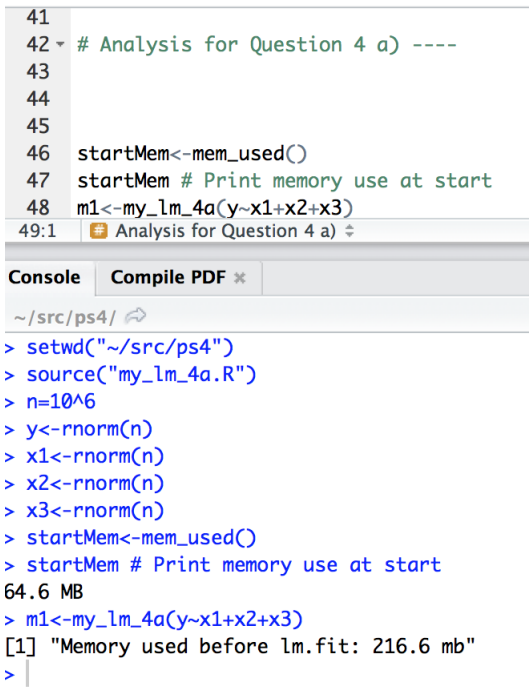


Figure 2: Comparing the amount of memory used when only observations and covariates are stored in the global environment, to right before lm.fit is called inside the function my_lm() (copy of Rs lm function). The difference is 152 mb.

## 4.2 4b)

This analysis answers how I came to the conclusion in 4 c). Also see Appendix for outprint of the function *my_lm_4b* (function that can be checked in the Github repository) and using several print statements. All the analysis in part 4 b) - d) was used with the function that can be found in subquestion 4 d). Run $prob4.R$ if the results are to be double checked (also located in the Github repository).

To find out where the major memory usage leading up to the call of lm.fit, both *mem_used*() was printed on every line a new object was created (together with *object.size*()) for that object), while also the function $my\_lm_4b$ was debugged using the browser. On every line *mem_used*() was evaluated.

The results indicated that three objects are created / evaulated that take up more than 10% of memory of the size of the vector of observations (more than 0.8mb). They are:

- $mf$: A list containing the observations $y$ and the covariates $x1$, $x2$ and $x3$. This object takes up about 1560 bytes before it is evaluated as it only is a reference object. After evaluation its dimensions are [1 000 000 x 4] and the size of the object is 32006200bytes which roughly equals 32mb. This is logical as each covariate array and the observations array take up 8bytes per entry times 1 million entries and that should equal 32mb extra memory use when the object is created.

- $y$: This is the second object created that adds significantly to memory use. It is an array of doubles containing all the 1 million observations. This object takes up about 48 bytes before it is evaluated as is then only a boolean eqauls to false. After evaluation its dimensions are [1 000 000 x 1] and the size of the object is 64000192bytes which roughly equals 64mb. This is not logical as the observation array only stores one million doubles. However **if we look at the attributes of y** we can see that all entries in that vector has a name associated with it (the index). This information takes up 56mb of memory usage (found in debug mode, when taking $attributes(y) \leftarrow NULL$ and then evaluating the size of the 'stripped y object which equaled 8mb). Therefore it is logical that when we have stripped off the attributes the object size is 8bytes per entry times 1 million entries which should equal 8mb extra memory use when the object is created. However there is another oddity here when the y object is created, it adds 80mb in total to the memory use, even though the total object size is 64mb. The extra 16mb is not in a newly created object either (all objects inside the function where printed with ls() before and after y was created, and the sizes were checked, none of them that we could check was increased). However there are some objects, like 'data' and 'subset' that we cannot check the size of, referencing to C objects and functions that R makes use of. It is probably in one of those objects that the extra 16mb is stored.

- $x$: This is the third object created that adds significantly to memory use. It is a matrix of doubles containing all the covariates $x1$, $x2$, $x3$ as well as the *intercept* for the linear regression. This object takes up about 48 bytes before it is evaluated (only a boolean = FALSE). After evaluation its dimensions are [1 000 000 x 4] and the size of the object is 88000848bytes which roughly equals 88mb. This is also not logical as the total memory use for the doubles inside the matrix would equal to 8bytes * 4 * 1 million = 32mb. However here **x also has attributes associated with it** where every entry in the matrix is assined a *dimname*. This information also, precisely as in the case of $y$ takes up 56mb of memory usage (found in debug mode, when taking $attributes(x\$dimnames) < -NULL$ and then evaluating the size of the 'stripped' x object, which still preserves the dimensions and other attributes of x, it equaled 32mb). When we strip off the attributes the object size is 8bytes per entry times 1 million times 4 entries which should equal 32mb extra memory use when the object is created. In this case there is another strange oddity when $x$ is created, it only adds 40mb in total to the memory use, even though the total object size is 88mb. Suddenly 48 mb of memory disappears that is not associated with the variables that existed before and after the creation of $x$ (checked in the same manner as for $y$ with $ls()$ in debug mode and object.size() for every object before and after. Since there are objects

here also that we cannot check the size of as they are C referenced objects, something must change in those underlying objects.

## 4.3 Prob 3 c)

See reasoning in 3b) to see how I came up with this reasoning. In order to reduce the memory use of lm, before calling lm.fit I would create the objects $y$ and $x$ inside the function without the attributes. It clearly shows that the function still runs without the attributes (has been checked) and this would save a lot of memory and would also speed up the process. (However it is evident that these attributes must have some use in the lm function, otherwise I would be too much of an R wizard that has optimized a base function like $lm()$ in R, only working in the software for six weeks).

## 4.4 Prob 3 d)

Below you will see an explanation of the objects created in the lm function before lm.fit is called. This is for the code of my copy of the lm function called *my_lm_4b* (although structured so that only the relevant elements are shown. This was the easiest way to explain the reasoning. Also this is the code that produces the output so that I could carry out the analysis in 3b) 3c). An example of the print out (I excluded the debugging session, where all object sizes and .Internal(inspect()) was carried out) of this function can be seen in the Appendix.

Also in grading this subquestion please take into account the explanation in subquestion 3 b) as well as the Appendix with the print out of all the the *mem_used*() and object analysis. I did not include all object.size() printout for the sake of space include that here all the time.

Sorry if the structuring below is quite messy, but skim it through and you will see how my analysis was carried out.

```
my_lm_4b <- function (formula, data, subset, weights, na.action, method = "qr",
                      model = TRUE, x = FALSE, y = FALSE, qr = TRUE, singular.ok = TRUE,
                      contrasts = NULL, offset, ...)
{

  #print memory usage on every step
  memUsed<-quote(print(paste("Memory used:",round( mem_used()/10^6,1),"mb")))

  eval(memUsed)
  ##Memory used: 79.8 mb
  ret.x <- x
  ##Object size 48bytes, only a reference
  ## x is a boolean = FALSE



  eval(memUsed)
  ##Memory used: 79.8 mb
  ret.y <- y
  ##Object size 48bytes
  ## .Internal(inspect(ret.x))
  ## @108ee1af8 10 LGLSXP g1c1 [MARK,NAM(2)] (len=1, tl=0) 0
  eval(memUsed)
  ##Memory used: 79.8 mb
  cl <- match.call()
  #object.size is 1120 bytes
```

```
#cl is the formula my_lm_4b(formula = y ~ x1 + x2 + x3)

eval(memUsed)
mf <- match.call(expand.dots = FALSE)
#mf is the formula, also 1120 bytes
eval(memUsed)
##Memory used: 79.8 mb
m <- match(c("formula", "data", "subset", "weights", "na.action",
             "offset"), names(mf), 0L)

# m is a integer vector with 5 values

## Here we only add attributes to the formula and store them in mf
eval(memUsed)
mf <- mf[c(1L, m)]
#only the formula reassigned
eval(memUsed)
mf$drop.unused.levels <- TRUE
  ##Memory used still: 79.8 mb
eval(memUsed)
mf[[1L]] <- quote(stats::model.frame)
#change the
## Size mf = 1560bytes


eval(memUsed)
print("FIRST JUMP")
print(paste("object mf size before eval =", object.size(mf)))



mf <- eval(mf, parent.frame())
#here mf is evaluated, taking in the values of predictors and the observations
#Size mf = 32mb

print("HEAD OF MF")
print(head(mf))
print(paste("The dimensions of mf are 1 000 000 x 4"))
print(paste("The typeof(mf) is", unlist(typeof(mf))))
print(paste("The typeof(mf[1,1]) is", unlist(typeof(mf[1,1])))))
print(paste("object mf size after eval =", object.size(mf)))
eval(memUsed)
print(paste("object size mf[1,1] after eval =", object.size(mf[1,1]))))
print("Comment FIRST JUMP: This is perfectly logic
      as mf is the observations (1*8mb) and the predictors (3*8mb)")



#### BEGIN: The function never enters here in our analysis
if (method == "model.frame") {
  return(mf)
  eval(memUsed)
}
```

```
  else if (method != "qr") {
    eval(memUsed)
    warning(gettextf("method = '%s' is not supported. Using 'qr'",
                     method), domain = NA)
  }
  #### END: The function never enters here in our analysis

  eval(memUsed)
  mt <- attr(mf, "terms")
  ## mt is a language object with the attributes of mf stored as below:
  ##object size mt 5104bytes
  ###ATRIBUTES OF MT
#       y ~ x1 + x2 + x3
#     attr(,"variables")
#     list(y, x1, x2, x3)
#     attr(,"factors")
#        x1 x2 x3
#     y   0  0  0
#     x1  1  0  0
#     x2  0  1  0
#     x3  0  0  1
#     attr(,"term.labels")
#     [1] "x1" "x2" "x3"
#     attr(,"order")
#     [1] 1 1 1
#     attr(,"intercept")
#     [1] 1
#     attr(,"response")
#     [1] 1
#     attr(,".Environment")
#     <environment: R_GlobalEnv>
#     attr(,"predvars")
#     list(y, x1, x2, x3)
#     attr(,"dataClasses")
#             y        x1        x2        x3
#     "numeric" "numeric" "numeric" "numeric"




  cat("\n")
  print("SECOND JUMP")
  eval(memUsed)
  print(paste("object y size before eval =", object.size(y)))
  print("Objects in the environment ls() before y eval")
  print(ls())

  ## The ls() function before and after

  y <- model.response(mf, "numeric")

  # adds on the attributes t y that take up alot of space
```

```
# extra memory is stored in C objects
# We can see with .Internal(inspect(y)) that y is now pointing
# at a different location in memory

print("Objects in the environment ls() after y eval")
print(ls())
print("HEAD OF y")
print(head(y))

print(paste("The typeof(y) is", typeof(y)))
print(paste("The typeof(y[1]) is", typeof(y[1])))
print(paste("The length of y is", length(y)))
print(paste("object y size after eval =", object.size(y)))
eval(memUsed)
print(paste("object size y[1] after eval =", object.size(y[1])))
print("Comment SECOND JUMP: This is not logic, since y is a vector
      of 10^6 values it should only add 8mb extra. However the attributes
      associated with y take up the memory that is larger than 8mb.")
print(ls())
print("END SECOND JUMP")
cat("\n")




w <- as.vector(model.weights(mf))
## w only becomes a NULL entry, because we do not have weights
eval(memUsed)

#### BEGIN: The function never enters here in our analysis
  if (!is.null(w) && !is.numeric(w)) {
    eval(memUsed)
    print("bbbb")
    stop("'weights' must be a numeric vector")
  }
#### END: The function never enters here


eval(memUsed)
offset <- as.vector(model.offset(mf))
# also NULL entry, no offset in the linear regression
eval(memUsed)


#### BEGIN: The function never enters here in our analysis
# Code removed to save space


#### END: The function never enters here in our analysis
else {
```

```
  cat("\n")
  print("LAST JUMP")

  eval(memUsed)
  print(paste("object x size before eval =", object.size(x)))
  cat("\n")
  print("Objects in the environment ls() before x eval")
  print(ls())
  cat("\n")

  x <- model.matrix(mt, mf, contrasts)

  ## Here x is created taking up 88 mb of memory, but memory is freed
  ## .Internal(inspect(x)) shows that x is now pointing to another reference in memory
  ## When we strip off the attributes x takes on a logical size

  print("Objects in the environment ls() after x eval")
  print(ls())
  cat("\n")
  print(paste("object x size after eval =", object.size(x)))
  print("HEAD OF x")
  print(head(y))
  print(paste("The typeof(x) is", typeof(x)))
  print(paste("The typeof(x[1,1]) is", typeof(x[1,1])))
  print("The dim of x is (see below)")
  print(dim(x))
  print("Comment LAST JUMP: x is a [10^6 x 4] matrix with
        values for the intercept and the three covariates.
        It should only be 32mb, although the size is 88mb.
        The extra memory is taken up by the dimnames of the attributes.")
  cat("\n")
  eval(memUsed)
  print("END LAST JUMP")
  cat("\n")




  z <- if (is.null(w)) {
    ### LM.FIT IS CALLED HERE
    print(paste("Memory used before lm.fit:", round(mem_used()/10^6,1),"mb")) # FIRST PRINT
    lm.fit(x, y, offset = offset, singular.ok = singular.ok,
           ...)
 ### CODE AFTER REMOVED TO SAVE SPACE
}
```

# 5 Appendix

Example print out when running the function *my_lm_4a* specified in subquestion 3d)

```
> startMem # Print memory use at start
79.6 MB
>
> m2<-my_lm_4b(y~x1+x2+x3)
[1] "Memory used: 79.6 mb"
[1] "Memory used: 79.6 mb"
[1] "Memory used: 79.6 mb"
[1] "Memory used: 79.6 mb"
[1] "Memory used: 79.6 mb"
[1] "Memory used: 79.6 mb"
[1] "Memory used: 79.6 mb"
[1] "Memory used: 79.6 mb"
[1] "Memory used: 79.6 mb"
[1] "FIRST JUMP"
[1] "object mf size before eval = 1560"
[1] "HEAD OF MF"
           y         x1          x2         x3
1  1.2629543  1.0536289  1.43764703  0.9394052
2 -0.3262334 -1.3120350  1.51762833 -1.6734425
3  1.3297993  2.0852002 -0.04439491  1.1289378
4  1.2724293  0.2929012  0.79577121  2.1111618
5  0.4146414  0.4976107  0.73851520 -1.4236423
6 -1.5399500  0.1921341  0.23886450 -0.1855364
[1] "The dimensions of mf are 1 000 000 x 4"
[1] "The typeof(mf) is list"
[1] "The typeof(mf[1,1]) is double"
[1] "object mf size after eval = 32006200"
[1] "Memory used: 111.6 mb"
[1] "object size mf[1,1] after eval = 48"
[1] "Comment FIRST JUMP: This is perfectly logic as mf is
the observations (1*8mb) and the predictors (3*8mb)"
[1] "Memory used: 111.6 mb"

[1] "SECOND JUMP"
[1] "Memory used: 111.6 mb"
[1] "object y size before eval = 48"
[1] "Objects in the environment ls() before y eval"
 [1] "cl"          "contrasts"   "data"        "formula"     "m"           "memUsed"     "method"
 [8] "mf"          "model"       "mt"          "na.action"   "offset"      "qr"          "ret.x"
[15] "ret.y"       "singular.ok" "subset"      "weights"     "x"           "y"

[1] "Objects in the environment ls() after y eval"

 [1] "cl"          "contrasts"   "data"        "formula"     "m"           "memUsed"     "method"
 [8] "mf"          "model"       "mt"          "na.action"   "offset"      "qr"          "ret.x"
[15] "ret.y"       "singular.ok" "subset"      "weights"     "x"           "y"
[1] "HEAD OF y"
         1          2          3          4          5          6
 1.2629543 -0.3262334  1.3297993  1.2724293  0.4146414 -1.5399500
[1] "The typeof(y) is double"
```

```
[1] "The typeof(y[1]) is double"
[1] "The length of y is 1000000"
[1] "object y size after eval = 64000192"
[1] "Memory used: 191.6 mb"
[1] "object size y[1] after eval = 256"
[1] "Comment SECOND JUMP: This is not logic, since y is a vector of 10^6 values it should
 only add 8mb extra. However the attributes associated with y take up the memory that is
 larger than 8mb."

 [1] "cl"          "contrasts"   "data"        "formula"     "m"           "memUsed"     "method"
 [8] "mf"          "model"       "mt"          "na.action"   "offset"      "qr"          "ret.x"
[15] "ret.y"       "singular.ok" "subset"      "weights"     "x"           "y"
[1] "END SECOND JUMP"

[1] "Memory used: 191.6 mb"
[1] "Memory used: 191.6 mb"
[1] "Memory used: 191.6 mb"

[1] "LAST JUMP"
[1] "Memory used: 191.6 mb"
[1] "object x size before eval = 48"

[1] "Objects in the environment ls() before x eval"
 [1] "cl"          "contrasts"   "data"        "formula"     "m"           "memUsed"     "method"
 [8] "mf"          "model"       "mt"          "na.action"   "offset"      "qr"          "ret.x"
[15] "ret.y"       "singular.ok" "subset"      "w"           "weights"     "x"           "y"

[1] "Objects in the environment ls() after x eval"
 [1] "cl"          "contrasts"   "data"        "formula"     "m"           "memUsed"     "method"
 [8] "mf"          "model"       "mt"          "na.action"   "offset"      "qr"          "ret.x"
[15] "ret.y"       "singular.ok" "subset"      "w"           "weights"     "x"           "y"

[1] "object x size after eval = 88000848"
[1] "HEAD OF x"
         1          2          3          4          5          6
 1.2629543 -0.3262334  1.3297993  1.2724293  0.4146414 -1.5399500
[1] "The typeof(x) is double"
[1] "The typeof(x[1,1]) is double"
[1] "The dim of x is (see below)"
[1] 1000000       4
[1] "Comment LAST JUMP: x is a [10^6 x 4] matrix with values for the
 intercept and the three covariates. It should only be 32mb,
 although the size is 88mb. The extra memory is taken up by the dimnames of the attributes."

[1] "Memory used: 231.6 mb"
[1] "END LAST JUMP"

[1] "Memory used before lm.fit: 231.6 mb"
```