Stat 243
Problem set 4

A. Strand, ID: 540441, GitHub: AndreasStrand

October 12, 2015

Problem 1

(a) With the proposed code tmp() do not create a random number from where you left off. The problem is that the load call within the function actually loads tmp.Rda into the global environment.

(b) In order to make it work we set "envir = environment(tmp)" in the load call in tmp(). The modified code is shown in code 1.

Code 1: Modified code where tmp.Rda is loaded to the right environment.

```
# Modified code where tmp() create a
# random number using the saved random seed index.
set.seed(0)
runif(1)
save(.Random.seed, file = 'tmp.Rda')
runif(1)
load('tmp.Rda')
runif(1)
tmp <- function(){
  # Now load is modified to load tmp.Rda into the function
  # environment instead of the global environment.
  load('tmp.Rda', envir = environment(tmp))
  runif(1)
}
tmp()
```

Problem 2

Let log denote the natural logarithm. First, let us rewrite f and write out its logarithm,

$$f(k;n,p,\phi) = \binom{n}{k}\left(\frac{n^n}{k^k(n-k)^{(n-k)}}\right)^{\phi-1} p^{k\phi}(1-p)^{(n-k)\phi}$$

$$\log f = \log\binom{n}{k} + (\phi-1)[n\log n - k\log k - (n-k)\log(n-k)] + k\phi\log p + (n-k)\log(1-p)\ ,$$

where $0\cdot\log 0 = 0$. The computations of the denominator of $P(Y=y)$ are shown in code 2. If the summation was not done in log scale there is a higher risk of under-floating or over-floating due to numbers being multiplied. The timings on an apply() based solution and a fully vectorized using system.time().

Code 2: The denominator of $P(Y=y)$.

```
#### Creating a function that computes the log of f
logF <- function(k, p=0.3, th=0.5, N=100) {
  # When taking the log  of f we need to consider special cases,
  # (0)*log(0) returns NaN but we want it to be 0.
  if ((n-k) == 0){
    term = 0
  } else {
    term = (n-k)*log(n-k)
```

```
  }
  return(log(choose(n,k)) + (th-1)*(n*log(n)-k*log(k)-term)+
    k*th*log(p) + (n-k)*th*log(1-p))
}

#### Calculating the denominator of P(Y=y)

# Using apply (The code is in a function in order to check timing later.)
denomApply <- function(tmp){
  assign("n", tmp, envir = .GlobalEnv)
  sum(exp(apply(matrix(1:n), 1, logF, N=n)))
}

# Using full vectorization
p=0.3; th=0.5
denomVector <- function(n){
  k = c(1:n)
  # Using ifelse to avoid NaNs
  sum(exp(log(choose(n,k)) + (th-1)*(n*log(n)-k*log(k) -
    ifelse(n-k, (n-k)*log(n-k), 0))+k*th*log(p) + (n-k)*th*log(1-p)))
}

#### Comparing timing
times <- function(r){
  #assign("n", r, envir = .GlobalEnv)
  c(system.time(denomApply(r))[3],system.time(denomVector(r))[3])
}

range = matrix(seq(10, 2000, by = 10))
timeMat = apply(range, 1, times)
rownames(timeMat) = c("apply", "vector")
colnames(timeMat) = range
```

Problem 3

The code for problem 3 as a whole is included in code 3. The timing was about 10 ms for sapply() computation on set A, while it was approximately 0 for the sapply() computation on set B and the vectorized computations.

Code 3: Computing the mean for distributions given by data mixedMember.Rda.

```
setwd("C:/Users/Andreas/Documents/stat243")
load("C:/Users/Andreas/Downloads/mixedMember.Rda")

# Calculating the means of the normal distributions in group A and group B with timing
aApplyTime = unname(system.time(aMeans <- sapply(1:length(muA), function(i) sum(wgtsA[[i
bApplyTime = unname(system.time(bMeans <- sapply(1:length(muB), function(i) sum(wgtsB[[i

# Creating data frames
Ka = 1000
Kb = 10

aMax = max(sapply(1:Ka, function(x) length(IDsA[[x]])))
bMax = max(sapply(1:Kb, function(x) length(IDsB[[x]])))

aData = data.frame(t(sapply(1:Ka, function(x){
  c(muA[IDsA[[x]]], rep(0, aMax-length(IDsA[[x]])),
    wgtsA[[x]], rep(0, aMax-length(IDsA[[x]])))})))
bData = data.frame(t(sapply(1:Kb, function(x){
  c(muB[IDsB[[x]]], rep(0, bMax-length(IDsB[[x]])),
    wgtsB[[x]], rep(0, bMax-length(IDsB[[x]])))})))
```

```
# Vectorized computation of the means with timing for each data set
start <- proc.time()
aVec <- 1:Ka
aSums <- rowSums(aData[aVec, 1:aMax]*aData[aVec, (aMax+1):(2*aMax)])
aVecTime<- unname((proc.time() - start)[3])

start <- proc.time()
bVec <- 1:Kb
bSums <- rowSums(bData[bVec, 1:bMax]*bData[bVec, (bMax+1):(2*bMax)])
bVecTime<- unname((proc.time() - start)[3])
```

Problem 4

a) The code for problem 4 is included in code 4. In the end of the code I included some of the ideas I tried out for the problem. I did not quite figure out how to look inside lm() to see how memory use changed during the call. I chose to try splitting the call into first creating a design matrix and to call the lm.fit() on it after, and hoping that this sequence will be similar to the lm() call. There is unfortunately probably a lot of other elements in the lm() call that will not be exposed.

The design matrix that is created before the lm.fit() call is of size 88 MB. The memory used to store observations and covariates is 63.6 MB. By this it seems like lm() is temporarily uses a lot of memory.

b) Without having the full insight in lm() I still think that the design matrix is responsible for the majority of the memory use. It consist of 4 columns of a million numbers and unless its transpose or the information matrix (the inverse factor) also are created in lm() there should not be necessary with a lot of other memory usage.

In the design matrix each column uses 8 MB, which is 1 B per element. With 4 columns that is a total of 32 MB, but since the matrix is of size 88 MB, other types of information like the structure uses 56 MB.

c) It seems unnecessary that the column of ones uses 8 MB, but storage in lm() might be different than it is in the output of the model.matrix() call. An efficient way of storing the first column is having pointers to the same address or using 4 B variables. Of course, if it was a lot of columns, it would not really matter how the first one is stored.

Code 4: Memory usage in a linear regression.

```
library(pryr) # useful for memory calculations
# Initializing data
n = 1e6
y = rnorm(n, mean = 5, sd = 3)
x1 = rnorm(n, mean = 2, sd = 1)
x2 = rnorm(n, mean = 6, sd = 4)
x3 = rnorm(n, mean = -4, sd = 2)

# Linear regression
mem_used()
X = model.matrix(y ~ x1 + x2 + x3)
mem_change(lm.fit(X,y)) # 13.7 kB

# examination of the memory usage in X
object_size(X) # 88 MB
object_size(X[ ,2:4]) # 80 MB
object_size(X[ ,1:3]) # 80 MB
#      Each column is of 8 MB and the structure is 56 MB.

# Experimentation, notes
mem_change(model1 <- lm(y ~ x1 + x2 + x3))
```

```
.Internal(inspect(model1))
gc(model1 <- lm(y ~ x1 + x2 + x3))

Rprof("profile1.out", line.profiling=TRUE)
lm(y ~ x1 + x2 + x3)
Rprof(NULL)
summaryRprof("profile1.out", lines = "show")
```