Problem Set 4

Cameron Adams

October 16, 2017

1 In class we discussed the idea of a closure as a function ...

```
x <- 1:10

f <- function(input){
    data <- input
    g <- function(param) return(param * data)
    return(g)
}

myFun <- f(x)
data <- 100
myFun(3)

##  [1]  3  6  9 12 15 18 21 24 27 30

x <- 100
myFun(3)

##  [1]  3  6  9 12 15 18 21 24 27 30
```

1.1 What is the maximum number of copies that exist of the vector 1:10 during the first execution of myFun()? Why?

There are two copies of x, one in global memory from `x <- 1:10`, one in local memory of the function `myFun()`.

1.2 Use serialize() to generate a sequence of bytes that store ...

```
x <- 1:1e6
x_size <- object_size(x)
x_len <- length(serialize(x, NULL)) #62


f <- function(input){
    data <- input
    g <- function(param) return(param * data)
    return(g)
}
myFun <- f(x)
```

```
rm(x)

myFun_size <- object.size(myFun)
myFun_len <- length(serialize(myFun, NULL))
myFun_len;x_len

## [1] 8010067
## [1] 4000022

myFun_len-x_len

## [1] 4010045

x_size;myFun_size

## 4 MB
## 1560 bytes
```

No, the size of the serialized object is not what I expect. it seems that there are two copies of `x` inside of `myFun`, plus one copy of x in the compiled code of `myFun`. Therefore there are three copies, which is weird because from 1a) we know there should be two.

1.3 It seems unnecessary to have the "data ¡- input" line ...

```
x <- 1:10     #set vector 1:10 to object x

f <- function(data){
    g <- function(param) return(param * data)
    return(g)
}

myFun <- f(x)
rm(x)
data <- 100
myFun(3)

## Error in myFun(3):  object 'x' not found
```

When `myFun <- f(x)` is evaluated, `f(x)` is assigned to `myFun`, the function `f` with value `x` is not evaluated; a promise is created for evaluation of `f(x)` and it will be lazy-evaluated. That occures when `myFun()` is evaluate with input. Since we are removing `x` from memory before evaluate `myFun`, we get an error.

1.4 Can you figure out a way to make the code in part (c) work ...

```
x <- 1:10
length(serialize(x, NULL))

## [1] 62

f <- function(data){
    invisible(is.numeric(data))
    g <- function(param) return(param * data)
    return(g)
```

```
}

myFun <- f(x)
rm(x)
data <- 100
myFun(3)

##  [1]  3  6  9 12 15 18 21 24 27 30

length(serialize(myFun, NULL))

## [1] 9458

length(serialize(1:10, NULL))

## [1] 62
```

We can make part(c) work by somehow inserting the intput data into the `f()` closure itself. Invisibily testing whether the data is of class numeric does that and allows the code work to work. Another way to do it would be to simply put the data input into the function.

```
x <- 1:10
length(serialize(x, NULL))

## [1] 62

f <- function(data){
    data
    g <- function(param) return(param * data)
    return(g)
}
```

2 This question explores memory use and copying with lists. In answering this question you can ignore what is happening with the list attributes, which are also reported by .Internal(inspect()).

********** **For all problem 2 questions: RStudio doesn't work for these problems. Code and output for these answers are from sessions run in standalone R.** **********

2.1 Consider a list of vectors. Modify an element of one of the vectors. Can R make the change in place, without creating a new list or a new vector?

```
rm(list=ls())

#create list of random numbers, 10 elements, 5 list elements and inspect
vecList <-lapply(1:5,function(x) sample(1:100,10))
class(vecList)
[1] "list"
length(vecList)
[1] 5
.Internal(inspect(vecList))
```

```
@7fa06fd6f010 19 VECSXP g1c4 [MARK,NAM(1)] (len=5, tl=0)
  @7fa070765950 13 INTSXP g1c4 [MARK] (len=10, tl=0) 74,36,19,24,34,...
  @7fa07201fee0 13 INTSXP g0c4 [] (len=10, tl=0) 10,59,61,67,70,...
  @7fa0720201b8 13 INTSXP g0c4 [] (len=10, tl=0) 78,12,11,79,13,...
  @7fa0720202f0 13 INTSXP g0c4 [] (len=10, tl=0) 52,45,36,50,67,...
  @7fa072018cd8 13 INTSXP g0c4 [] (len=10, tl=0) 42,34,64,14,37,...

#change value of 1st element in 1st list element and inspect
vecList[[1]][1] <- 10
.Internal(inspect(vecList))
@7fa06fd6f010 19 VECSXP g1c4 [MARK,NAM(1)] (len=5, tl=0)
  @7fa0718966a8 14 REALSXP g0c5 [] (len=10, tl=0) 10,36,19,24,34,...
  @7fa07201fee0 13 INTSXP g0c4 [] (len=10, tl=0) 10,59,61,67,70,...
  @7fa0720201b8 13 INTSXP g0c4 [] (len=10, tl=0) 78,12,11,79,13,...
  @7fa0720202f0 13 INTSXP g0c4 [] (len=10, tl=0) 52,45,36,50,67,...
  @7fa072018cd8 13 INTSXP g0c4 [] (len=10, tl=0) 42,34,64,14,37,...
```

R *cannot* modify an element of one of the vectors without making a copy of the vector that changed and pointing to the new vector memory address.

2.2 Next, make a copy of the list and determine ...?

```
vecListCopy <- vecList
.Internal(inspect(vecList))
@7f8b7eb3ed50 19 VECSXP g1c4 [MARK,NAM(2)] (len=5, tl=0)
  @7f8b7ed25f48 14 REALSXP g0c5 [] (len=10, tl=0) 10,47,11,60,80,...
  @7f8b7ee52478 13 INTSXP g0c4 [] (len=10, tl=0) 55,33,67,16,53,...
  @7f8b7ee527b8 13 INTSXP g0c4 [] (len=10, tl=0) 4,8,51,30,24,...
  @7f8b7ee4e540 13 INTSXP g0c4 [] (len=10, tl=0) 6,12,48,93,54,...
  @7f8b7ee4e818 13 INTSXP g0c4 [] (len=10, tl=0) 47,95,13,59,81,...
.Internal(inspect(vecListCopy))
@7f8b7eb3ed50 19 VECSXP g1c4 [MARK,NAM(2)] (len=5, tl=0)
  @7f8b7ed25f48 14 REALSXP g0c5 [] (len=10, tl=0) 10,47,11,60,80,...
  @7f8b7ee52478 13 INTSXP g0c4 [] (len=10, tl=0) 55,33,67,16,53,...
  @7f8b7ee527b8 13 INTSXP g0c4 [] (len=10, tl=0) 4,8,51,30,24,...
  @7f8b7ee4e540 13 INTSXP g0c4 [] (len=10, tl=0) 6,12,48,93,54,...
  @7f8b7ee4e818 13 INTSXP g0c4 [] (len=10, tl=0) 47,95,13,59,81,...

vecListCopy[[1]][1] <- 99
.Internal(inspect(vecListCopy))
@7f8b7ee4e9b8 19 VECSXP g0c4 [NAM(1)] (len=5, tl=0)
  @7f8b7ed25ff0 14 REALSXP g0c5 [] (len=10, tl=0) 99,47,11,60,80,...
  @7f8b7ee52478 13 INTSXP g0c4 [NAM(2)] (len=10, tl=0) 55,33,67,16,53,...
  @7f8b7ee527b8 13 INTSXP g0c4 [NAM(2)] (len=10, tl=0) 4,8,51,30,24,...
  @7f8b7ee4e540 13 INTSXP g0c4 [NAM(2)] (len=10, tl=0) 6,12,48,93,54,...
  @7f8b7ee4e818 13 INTSXP g0c4 [NAM(2)] (len=10, tl=0) 47,95,13,59,81,...
```

When a copy of the list is made, the copy points to address of teh original list. When an element is changed in one of teh vectors in the list copy, the address copied list now has a new address, and the vector with a changed element has a new address. The unchanged vectors in the copied have the same memory addess as in the original list.

2.3 Now make a list of lists ...

```
listOfLists <-lapply(1:2,
+                     function(x) lapply(1:2,
+                                        function(x) sample(1:100,10)))
length(listOfLists)
[1] 2
length(listOfLists[[1]])
[1] 2

#Copy the list
listOfListsCopy <- listOfLists
.Internal(inspect(listOfLists))
@7fdb504304e0 19 VECSXP g1c2 [MARK,NAM(2)] (len=2, tl=0)
  @7fdb50430550 19 VECSXP g1c2 [MARK,NAM(1)] (len=2, tl=0)
    @7fdb5079b810 13 INTSXP g1c4 [MARK] (len=10, tl=0) 66,47,11,60,80,...
    @7fdb5079b6d8 13 INTSXP g1c4 [MARK] (len=10, tl=0) 55,33,67,16,53,...
  @7fdb5227c578 19 VECSXP g0c2 [] (len=2, tl=0)
    @7fdb52231618 13 INTSXP g0c4 [] (len=10, tl=0) 4,8,51,30,24,...
    @7fdb511bc270 13 INTSXP g0c4 [] (len=10, tl=0) 6,12,48,93,54,...
.Internal(inspect(listOfListsCopy))
@7fdb504304e0 19 VECSXP g1c2 [MARK,NAM(2)] (len=2, tl=0)
  @7fdb50430550 19 VECSXP g1c2 [MARK,NAM(1)] (len=2, tl=0)
    @7fdb5079b810 13 INTSXP g1c4 [MARK] (len=10, tl=0) 66,47,11,60,80,...
    @7fdb5079b6d8 13 INTSXP g1c4 [MARK] (len=10, tl=0) 55,33,67,16,53,...
  @7fdb5227c578 19 VECSXP g0c2 [] (len=2, tl=0)
    @7fdb52231618 13 INTSXP g0c4 [] (len=10, tl=0) 4,8,51,30,24,...
    @7fdb511bc270 13 INTSXP g0c4 [] (len=10, tl=0) 6,12,48,93,54,...

#Add an element to the second list
listOfListsCopy[[2]][[3]] <- 99:90
.Internal(inspect(listOfListsCopy))
@7fdb52278278 19 VECSXP g0c2 [NAM(1)] (len=2, tl=0)
  @7fdb50430550 19 VECSXP g1c2 [MARK,NAM(2)] (len=2, tl=0)
    @7fdb5079b810 13 INTSXP g1c4 [MARK] (len=10, tl=0) 66,47,11,60,80,...
    @7fdb5079b6d8 13 INTSXP g1c4 [MARK] (len=10, tl=0) 55,33,67,16,53,...
  @7fdb5117fdf0 19 VECSXP g0c3 [] (len=3, tl=0)
    @7fdb52231618 13 INTSXP g0c4 [NAM(2)] (len=10, tl=0) 4,8,51,30,24,...
    @7fdb511bc270 13 INTSXP g0c4 [NAM(2)] (len=10, tl=0) 6,12,48,93,54,...
    @7fdb511bc340 13 INTSXP g0c4 [NAM(1)] (len=10, tl=0) 99,98,97,96,95,...
```

The following is shared between the two lists (`listOfLists, listOfListsCopy`):

- First list of list, `listOfListsCopy[[1]][[1]]` and `listOfLists[[1]][[1]]`.
- The first two vectors of the second list have the same address and were not copied, `listOfLists[[2]][[1]] == listOfListsCopy[[2]][[1]]` and `listOfLists[[2]][[2]] == listOfListsCopy[[2]][[2]]`.

Everything else was copied and is not shared between lists.

2.4 Run the following code in a new R session ...

```
gc()
         used (Mb) gc trigger (Mb) max used (Mb)
Ncells 257840 13.8     460000 24.6   350000 18.7
Vcells 537163  4.1    1023718  7.9   911750  7.0
```

```
tmp <- list()
x <- rnorm(1e7)
tmp[[1]] <- x
tmp[[2]] <- x

.Internal(inspect(tmp))
@7fced1c59008 19 VECSXP g0c2 [NAM(1)] (len=2, tl=0)
  @11235a000 14 REALSXP g0c7 [NAM(2)] (len=10000000, tl=0) 0.401138,-1.2158,0.950013,2.19382,0.43482,..
  @11235a000 14 REALSXP g0c7 [NAM(2)] (len=10000000, tl=0) 0.401138,-1.2158,0.950013,2.19382,0.43482,..

object.size(tmp)
160000136 bytes

gc()
          used (Mb) gc trigger  (Mb) max used (Mb)
Ncells   258528 13.9     592000  31.7   350000 18.7
Vcells 10538593 80.5   15576004 118.9 10545243 80.5
```

The help file for `object.size()` specifically says that the function "does not detect if elements of a list are shared". The function is simply adding up the memory usage of each element, without taking duplciate addresses into account.

3 Challenge 5 of Section 7.3 of Unit 4. The following is real code...

```
rm(list=ls())

setwd("/Users/CamAdams/repos/STAT243/ps4/")

load('ps4prob3.Rda') # should have A, n, K

################-------------------------------------------------------------------
## Student's version of the function

ll <- function(Theta, A) {
    sum.ind <- which(A==1, arr.ind=T)
    logLik <- sum(log(Theta[sum.ind])) - sum(Theta)
    return(logLik)
}

oneUpdate <- function(A, n, K, theta.old, thresh = 0.1) {
    theta.old1 <- theta.old
    Theta.old <- theta.old %*% t(theta.old)
    L.old <- ll(Theta.old, A)
    q <- array(0, dim = c(n, n, K))
    for (i in 1:n) {
        for (j in 1:n) {
            for (z in 1:K) {
                if (theta.old[i, z]*theta.old[j, z] == 0){
                    q[i, j, z] <- 0
                    } else {
                        q[i, j, z] <- theta.old[i, z]*theta.old[j, z] /
                            Theta.old[i, j]
```

```
                    }
                }
            }
        }


    theta.new <- theta.old
    for (z in 1:K) {
        theta.new[,z] <- rowSums(A*q[,,z])/sqrt(sum(A*q[,,z]))
    }

    Theta.new <- theta.new %*% t(theta.new)
    L.new <- ll(Theta.new, A)
    converge.check <- abs(L.new - L.old) < thresh
    theta.new <- theta.new/rowSums(theta.new)
    return(list(theta = theta.new, loglik = L.new,
                converged = converge.check))
}

# initialize the parameters at random starting values
temp <- matrix(runif(n*K), n, K)
theta.init <- temp/rowSums(temp)

#################-------------------------------------------------------------
## My version of the function

llMe <- function(Theta, A) {

    #calculate loglikelihood
    logLik <- sum(log(Theta[which(A==1, arr.ind=T)])) - sum(Theta)

    return(logLik)
}

oneUpdateMe <- function(A, n, K, theta.old, thresh = 0.1) {

    Theta.old <- tcrossprod(theta.old)
    L.old <- llMe(Theta.old, A)

    Theta.new <- tcrossprod(
                    sapply(1:K, function(z) {
                        q_z <- matrix(theta.old[ , z] %o% t(theta.old[ , z]),
                                      ncol = n) /
                                      Theta.old
                    return(rowSums(A * q_z) / sqrt(sum(A * q_z)))
                        }))

    L.new <- llMe(Theta.new, A)

    return(list(theta = theta.old, loglik = L.new,
                converged = FALSE))
}
```

```r
oneUpdateMe2 <- function(A, n, K, theta.old, thresh = 0.1) {

    theta.old <- theta.init
    Theta.old <- tcrossprod(theta.old)
    L.old <- llMe(Theta.old, A)

    #Theta.new <- tcrossprod(
    #                sapply(1:K, function(z) {
    #                    q_z <- matrix(theta.old[ , z] %o% t(theta.old[ , z]),
    #                                  ncol = n) /
    #                                  Theta.old
    #                    return(rowSums(A * q_z) / sqrt(sum(A * q_z)))
    #                    }))

    Theta.new <- tcrossprod(
                    vapply(1:K, FUN.VALUE = numeric(500), function(z) {
                        q_z <- matrix(theta.old[ , z] %o% t(theta.old[ , z]),
                                      ncol = n) /
                                      Theta.old
                        return(rowSums(A * q_z) / sqrt(sum(A * q_z)))
                        }))


    q_z <- vapply(1:K, FUN.VALUE = numeric(250000), function(z) {
                        matrix(theta.old[ , z] %o% t(theta.old[ , z]),
                                      ncol = n) /
                                      Theta.old})
            vapply(1:K )

    L.new <- llMe(Theta.new, A)

    return(list(theta = theta.old, loglik = L.new,
                converged = FALSE))
}

# initialize the parameters at random starting values
temp <- matrix(runif(n * K), n, K)
theta.init <- temp/rowSums(temp)

# benchmark student version vs. my version
require(microbenchmark)
microbenchmark(out <- oneUpdate(A, n, K, theta.init),      #student's verions
               outMe <- oneUpdateMe(A, n, K, theta.init),  #my version
               times = 20L, unit = "s")            #avg. over 20 iterations

## Unit: seconds
##                                      expr       min       lq     mean
##      out <- oneUpdate(A, n, K, theta.init) 8.3034210 8.615452 9.025107
##  outMe <- oneUpdateMe(A, n, K, theta.init) 0.7965409 0.867914 1.001540
##     median       uq      max neval
##  8.8505975 9.592912 10.03484    20
##  0.9279685 1.013755  1.65792    20

microbenchmark(out <- oneUpdateMe(A, n, K, theta.init),      #student's verions
```

```
               outMe <- oneUpdateMe2(A, n, K, theta.init),  #my version
               times = 20L, unit = "s")

## Error in match.fun(FUN): argument "FUN" is missing, with no default

#check to see if loglik values match between versions
out$loglik

## [1] -41666.03

out$loglik == outMe$loglik

## [1] TRUE
```

My code changes:

- Removed: `theta.old1 <- theta.old`
- Removed: `q <- array(0, dim = c(n, n, K))`
- `crossprod(theta.old)` instead of `theta.old %*% t(theta.old)`
- Combined calculating array `q` with for loops and `Theta.new` into one `sapply()` indexing through `z`-dimension.

With these changes, I was able to speed up the liklihood function by about a factor of 10. Further improvements might be made with use of `vapply` instead of `sapply`, but I ran out of time to implement.

4

4.1 This is a variation on Challenge 8 in Section 7.3 of Unit 4. The goal is to write a function to sample k values without replacement ...

```
rm(list=ls())

################
#PIKK methods

#defualt
PIKK <- function(x, k) {

    x[sort(runif(length(x)), index.return = TRUE)$ix[1:k]]

}

#my first method
PIKK.me1 <- function(x, k) {

    unique(x[as.integer(runif(k * 1.1, min = 1, max = length(x)))])[1:k]

}

#my second method
PIKK.me2 <- function(x, k) {
```

```
    unique(as.integer(runif(k * 1.1, min = 1, max = length(x))))[1:k]

}

#############
# test that PIKK methods
require(testthat)

#test paramters
x <- 1:10000
k <- length(x) * 0.05

#run test code
PIKK.out <- PIKK(x, k)
PIKK.me1.out <- PIKK.me1(x, k)
PIKK.me2.out <- PIKK.me2(x, k)

#view output
head(PIKK.out)

## [1] 2151 7621 1881 2265 2078 5140

head(PIKK.me1.out)

## [1] 6886 2782 6776 5779 5264 4758

head(PIKK.me2.out)

## [1] 8179 9103 6495 1497  100 9537

#testing function
test_that("Output is an integer vector of length(k)", {

    expect_that(PIKK.out, is_a("integer"))        #is integer
    expect_that(PIKK.me1.out, is_a("integer"))    #is integer
    expect_that(PIKK.me2.out, is_a("integer"))    #is integer

    expect_equal( length(PIKK.out), k)             #length==k

    expect_equal(length(PIKK.out),                 #output lengths match
                 length(PIKK.me1.out),
                 length(PIKK.me2.out)
                )
})

###############
#Benchmark sampling methods

#wrapper function
benchMeanWrapper <- function(..., n, k, times, units) {

    x <- 1:n   #set vector to sample from

    expr <- list(paste(..., sep = ",")) #get list expressions
```

```
    #create benchmark expression
    benchExpr <- paste("microbenchmark(", expr,
                       ", times = times,", "unit = units)")

    #eval benchmark expressions
    bench <- eval(parse(text = benchExpr))

    #return mean of iterations
    return(summary(bench)$mean)

}

#benchmark expressions with different n and k values and
n <- c(1e3, 1e4, 1e5, 1e6, 1e7)
kn_ratio <- c(0.05, 0.25, 0.5, 0.75)

# k:n=1:20
bench_5pp <- mapply("PIKK(x, k)", "PIKK.me1(x, k)",
                    "PIKK.me2(x, k)", "sample(x, k)",
                    n = n, k = n * kn_ratio[1], times = 20L, units = "ms",
                    FUN = benchMeanWrapper, SIMPLIFY = T)

# k:n=1:4
bench_25pp <- mapply("PIKK(x, k)", "PIKK.me1(x, k)",
                    "PIKK.me2(x, k)", "sample(x, k)",
                    n = n, k = n * kn_ratio[2], times = 20L, units = "ms",
                    FUN = benchMeanWrapper, SIMPLIFY = T)

# k:n=1:2
bench_50pp <- mapply("PIKK(x, k)", "PIKK.me1(x, k)",
                    "PIKK.me2(x, k)", "sample(x, k)",
                    n = n, k = n * kn_ratio[3], times = 20L, units = "ms",
                    FUN = benchMeanWrapper, SIMPLIFY = T)

# k:n=3:4
bench_75pp <- mapply("PIKK(x, k)", "PIKK.me1(x, k)",
                    "PIKK.me2(x, k)", "sample(x, k)",
                    n = n, k = n * kn_ratio[4], times = 20L, units = "ms",
                    FUN = benchMeanWrapper, SIMPLIFY = T)
```

```
###############
#create table output

options(scipen = -1) #set for printing in sci notatation.

#create dataframe of benchmarks
bench_Combined <- data.frame(
                    cbind(as.character(rep(n, length(kn_ratio))),
                            sort(as.character(rep(kn_ratio, length(n)))),
                            row.names = NULL),
                        rbind(t(bench_5pp), t(bench_25pp),
                                t(bench_50pp), t(bench_75pp)), row.names = NULL)
```

```
colnames(bench_Combined) <- c("n", "kn_ratio", "PIKK", "PIKK.me1",
                              "PIKK.me2", "sample()")

#sort on kn_ratio
bench_Combined <- bench_Combined[order(bench_Combined$n), ]

#print table
require(xtable)
tbl <- xtable(bench_Combined,
              caption = "Average runtime for sampling functions",
              comment = F, align = c("c", "c|","c|",rep("c", 4)),
              digits = 3)
print(tbl, floating = T, table.placement = "H", comment = F,
      include.rownames = F, caption.placement = "top", caption.width = "35em",
      hline.after = c(-1, seq(0, 20, 4)))
```

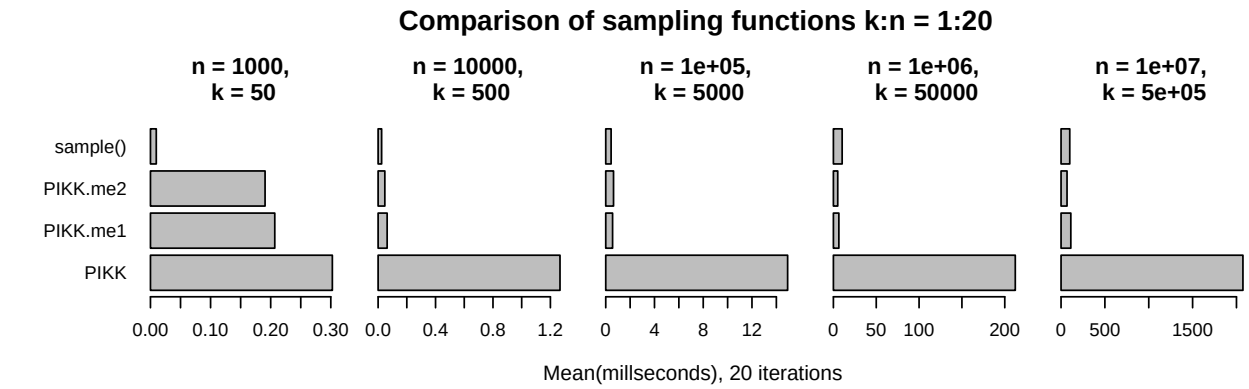
Table 1: Average runtime for sampling functions

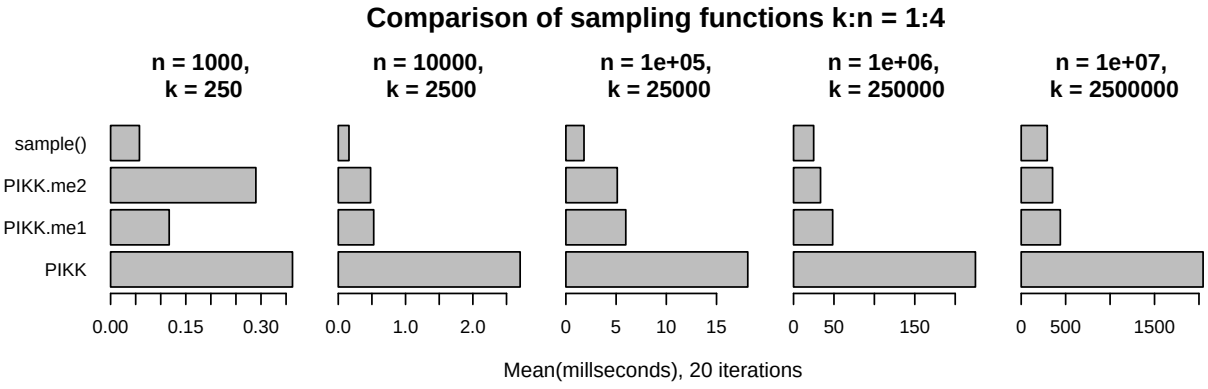
n	kn_ratio	PIKK	PIKK.me1	PIKK.me2	sample()
1000	0.05	0.303	0.207	0.191	0.010
1000	0.25	0.363	0.117	0.290	0.058
1000	0.5	0.256	0.103	0.096	0.030
1000	0.75	0.217	0.119	0.101	0.038
1e+04	0.05	1.267	0.063	0.047	0.025
1e+04	0.25	2.703	0.527	0.480	0.159
1e+04	0.5	2.102	0.807	0.794	0.227
1e+04	0.75	1.710	1.087	0.921	0.249
1e+05	0.05	14.927	0.562	0.639	0.440
1e+05	0.25	18.113	5.965	5.117	1.805
1e+05	0.5	19.101	13.074	11.095	2.813
1e+05	0.75	14.279	14.279	11.470	2.695
1e+06	0.05	212.277	6.388	5.064	10.229
1e+06	0.25	225.024	48.401	33.390	24.907
1e+06	0.5	228.619	129.356	91.190	54.773
1e+06	0.75	194.788	129.295	116.454	42.718
1e+07	0.05	2073.781	111.529	68.122	99.461
1e+07	0.25	2046.066	440.257	353.139	293.116
1e+07	0.5	2072.891	973.011	822.627	548.917
1e+07	0.75	2110.236	1455.055	1229.538	702.065

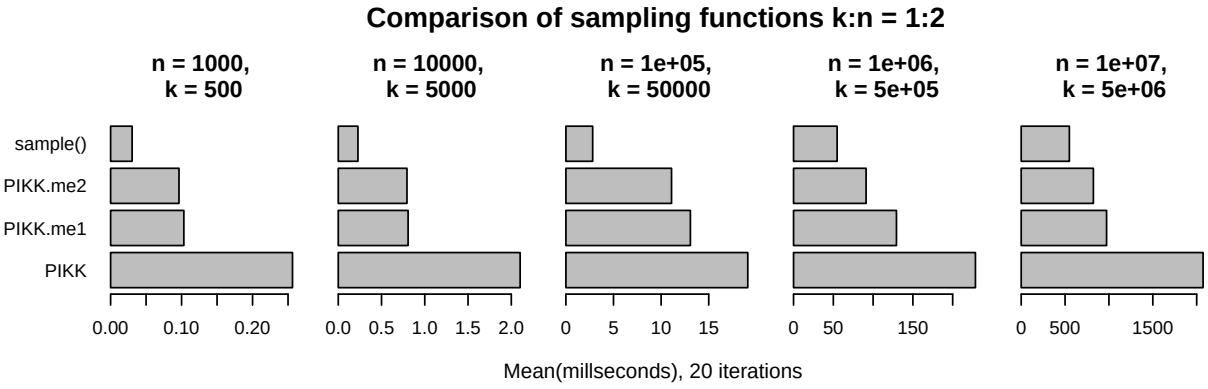
```
###############
#create plots

#0.05
par(mfrow = c(1, 5), mar = c(3,1,4,1), oma = c(2, 5, 2, 0))
for (i in 1:5) {
    barplot(bench_5pp[ , i],  horiz = T,
            ylab = "",
            main = paste0("n = ", n[i], ",\n k = ", n[i]*0.05),
            las = 1, cex.names = 1,
            names.arg = if(i==1) {c("PIKK", "PIKK.me1",
                                    "PIKK.me2", "sample()")}
```

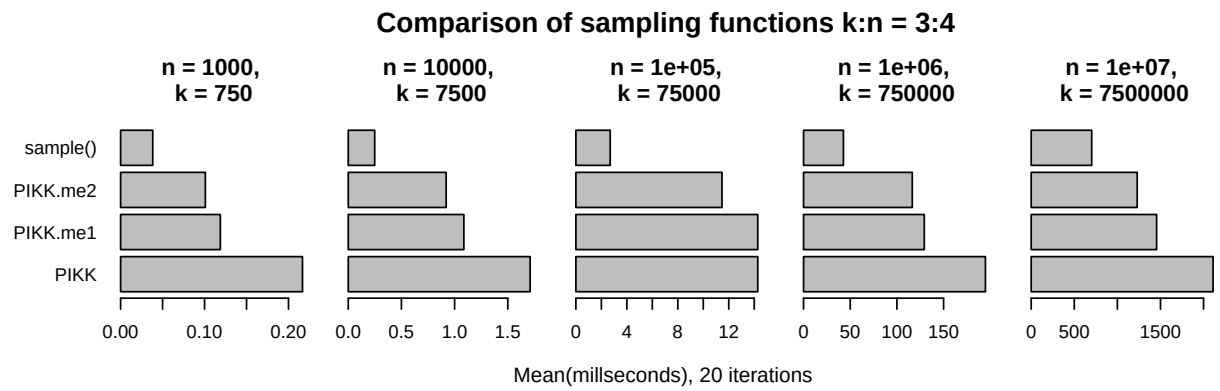
```
            else {" "})
}
mtext("Comparison of sampling functions k:n = 1:20",
      side = 3, outer = T, font = 2)
mtext("Mean(millseconds), 20 iterations ",
      side = 1, outer = T, font = 1, cex = 0.75)

#0.25
par(mfrow = c(1, 5), mar = c(3,1,4,1), oma = c(2, 5, 2, 0))
for (i in 1:5) {
    barplot(bench_25pp[ , i],  horiz = T,
            ylab = "",
            main = paste0("n = ", n[i], ",\n k = ", n[i] * 0.25),
            las = 1, cex.names = 1,
            names.arg = if(i==1) {c("PIKK", "PIKK.me1",
                                    "PIKK.me2", "sample()")}
            else {" "})
}
mtext("Comparison of sampling functions k:n = 1:4",
      side = 3, outer = T, font = 2)
mtext("Mean(millseconds), 20 iterations ",
      side = 1, outer = T, font = 1, cex = 0.75)

#0.50
par(mfrow = c(1, 5), mar = c(3,1,4,1), oma = c(2, 5, 2, 0))
for (i in 1:5) {
    barplot(bench_50pp[ , i],  horiz = T,
            ylab = "",
            main = paste0("n = ", n[i], ",\n k = ", n[i] * 0.5),
            las = 1, cex.names = 1,
            names.arg = if(i==1) {c("PIKK", "PIKK.me1",
                                    "PIKK.me2", "sample()")}
            else {" "})
}
mtext("Comparison of sampling functions k:n = 1:2",
      side = 3, outer = T, font = 2)
mtext("Mean(millseconds), 20 iterations ",
      side = 1, outer = T, font = 1, cex = 0.75)

#0.75
par(mfrow = c(1, 5), mar = c(3,1,4,1), oma = c(2, 5, 2, 0))
for (i in 1:5) {
    barplot(bench_75pp[ , i],  horiz = T,
            ylab = "",
            main = paste0("n = ", n[i], ",\n k = ", n[i] * 0.75),
            las = 1, cex.names = 1,
            names.arg = if(i==1) {c("PIKK", "PIKK.me1",
                                    "PIKK.me2", "sample()")}
            else {" "})
}
mtext("Comparison of sampling functions k:n = 3:4",
      side = 3, outer = T, font = 2)
mtext("Mean(millseconds), 20 iterations ",
```

```
      side = 1, outer = T, font = 1, cex = 0.75)

options(scipen = 0) #return to default
```









Both of my verions of PIKK are faster than the default version. The second version, `PIKK.me2` is approximately as fast as `sample()`, and for some values of `n`, and `k`, `PIKK.me2` is faster than `sample()`. `PIKK.me2` gets more efficient as n increases. All methods slow down with ratio of k:n increases. Sample appear to decrease in efficiency as n and k get get large.

4.2 Extra credit:

My function `PIKK.me2` is almost as fast as `sample()` and qualifies for extra credit because of its speed.