Problem Set 4

Jeffrey Kwarsick

October 11, 2017

1 Problem 1

1.1 Part (a)

The maximum number of copies that exist of vector 1:10 during the first execution of *myFun()* is once. From the code that I ran below, while invoking the *.Internal(inspect())* commands, I was able to observe that the vector x was passed into function x and since there was no operations conducted on the vector, the memory location is passed into the function f and it's memory location is referenced as it is saved to the variable *data*, located within the function. When the vector is operated on within function *f()* with function *g()*, a copy is created within the local function environment and then the resulting vector from the operations of *g()* is produced and outputed. There is only a single copy because the vector is only being modified in a single spot, when operated on by function *g()*. A copy is made because objects in R are immutable, so to change them a copy is made locally within the function and the original is not modified. After the operations are complete the copy of the vector is then deleted.

Overall, this is an example of how R uses *pass-by-value* in order to conduct operations of the function. It makes a copy and executes operations on the copy of the object as needed rather than operating and modifying the original object. This maintains the immutable nature of objects in R. Specifically, this is referred as *copy-on-change*. This means that unless a change is made to the object, the function always references back to the original object, but when the function is going to make a change on the object, a copy is then made to carry out the operations.

```
library('pryr')
x <- 1:10
.Internal(inspect(x))

## @21cf590 13 INTSXP g0c4 [NAM(2)] (len=10, tl=0) 1,2,3,4,5,...

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

## @21cf590 13 INTSXP g0c4 [NAM(2)] (len=10, tl=0) 1,2,3,4,5,...
## @21cf590 13 INTSXP g0c4 [NAM(2)] (len=10, tl=0) 1,2,3,4,5,...

.Internal(inspect(x))

## @21cf590 13 INTSXP g0c4 [NAM(2)] (len=10, tl=0) 1,2,3,4,5,...

data <- 100
myFun(3)
```

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

```
.Internal(inspect(myFun(3)))
```

```
## @2056718 14 REALSXP g0c5 [] (len=10, tl=0) 3,6,9,12,15,...
```

1.2 Part (b)

```
x <- 1:10
f <- function(input) {
  data <- input
  g <- function(param) return(param * data)
}
myFun <- f(x)
object_size(x)
```

```
## 88 B
```

```
length(serialize(x, NULL))
```

```
## [1] 62
```

```
rm(x)
object_size(myFun)
```

```
## 13.5 kB
```

```
length(serialize(myFun, NULL))
```

```
## [1] 8505
```

```
data <- 100
myFun(3)
```

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

Running the chunk of code above with the to determine the size of *myFun* and it's serialized length, I get a sizze of 2.53 kB and a length of 847. This is done with a vector of range 1 - 10, of length 10.

```
x1 <- 1:10000
f <- function(input) {
  data <- input
  g <- function(param) return(param * data)
}
myFun <- f(x1)
object_size(x1)
```

```
## 40 kB
```

```
length(serialize(x1, NULL))
```

```
## [1] 40022
```

```
rm(x1)
object_size(myFun)
```

```
## 53 kB

length(serialize(myFun, NULL))

## [1] 88035

data <- 100
```

Running the same code again with a long vector of length ten thousand, with a range of 1 - 10000, I observe an object size of *myFun* equal to 42.5 kB and a serialized length of 80768.

1.3 Part (c)

```
x  <- 1:10
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
```

The reason that the above code does not work is because the way objects are referenced and copied within a function environment. R operates on a *copy-on-change* basis. This means that until a change is to be done on the object read into the list, it only references back to the original object. When a change is to occur to the object, such as a multiplication or addition operation, a copy of the object is then made and the operations conducted. In the case of the code above, the vector x is called into function *f()*, but is not operated on until the called in function *g()*. In the code chunk above, the original vecto x is removed before the the parameter required to input into *myFun* is called. When *myFun(3)* was called, there was no original object to reference back to and create a copy of and this is why the R reported that x was not found, because no copy of it was made in the local function environment before it was deleted.

1.4 Part (d)

```
x <- 1:10
.Internal(inspect(x))

## @2a6b7e8 13 INTSXP g0c4 [NAM(2)] (len=10, tl=0) 1,2,3,4,5,...

f <- function(data){
  g <- function(param) return(param * data)
  return(g)
}
#list the variables in the global environment
ls( envir = .GlobalEnv )

## [1] "data"  "f"     "myFun" "x"

myFun <- f(x)
#stores pointer in the enclosed environment of the function myFun
environment(myFun)$data
```

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

.Internal(inspect(environment(myFun)$data))

## @2a6b7e8 13 INTSXP g0c4 [MARK,NAM(2)] (len=10, tl=0) 1,2,3,4,5,...

rm(x)
ls( envir = .GlobalEnv )

## [1] "data"  "f"     "myFun"

object_size(myFun)

## 16.3 kB

length(serialize(myFun, NULL))

## [1] 11102

data <- 100
myFun(3)

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

In the code above, I am able to make the code from part (c) work without explicitly making a copy with the function body itself with the line of code **.Internal(inspect(environment(myFun)$data))**. In part (c) of the problem the **rm()** is called to remove x from the global environment in R. With the line **.Internal(inspect(environment(myFun)$data))**, the pointer pointing to the location in memory where x is stored and is saved to the enclosed environment of the function and thus is still able to be accessed despite the list x's removal from the global environment in R. Because there is no explicit copy made of x (as data) in the enclosed environment of *myFun()*, the length of the serialized closure is considerably longer comapred to part (a) of the problem where an explicit copy of the vector is made. The length is 10756.

2 Problem 2

This problem was not done in RStudio, but a session of R in the terminal. Code is written below to show what was done in order to complete the problem.

2.1 Part (a)

```
#create a list of vectors
list_of_vecs <- list(c(1:3), c(50:73), c(22:45))
#Determine memory locations of the list and vectors within the list
.Internal(inspect(list_of_vecs))

## @230d5f0 19 VECSXP g0c3 [NAM(2)] (len=3, tl=0)
##   @2778198 13 INTSXP g0c2 [] (len=3, tl=0) 1,2,3
##   @20bbab8 13 INTSXP g0c5 [] (len=24, tl=0) 50,51,52,53,54,...
##   @20bbc08 13 INTSXP g0c5 [] (len=24, tl=0) 22,23,24,25,26,...

#What is the size of the list of vectors
object_size(list_of_vecs)

## 464 B
```

```
#Change a single element within one of the vectors
list_of_vecs[[2]][[20]] <- 169
#Determine the memory locations and what changed
.Internal(inspect(list_of_vecs))

## @2293e30 19 VECSXP g0c3 [NAM(1)] (len=3, tl=0)
##   @2778198 13 INTSXP g0c2 [NAM(2)] (len=3, tl=0) 1,2,3
##   @aa1b40 14 REALSXP g0c7 [] (len=24, tl=0) 50,51,52,53,54,...
##   @20bbc08 13 INTSXP g0c5 [NAM(2)] (len=24, tl=0) 22,23,24,25,26,...
```

When a change is made to an element within a vector within the list, a new vector is created of the vector where the change to the element was changed. This is evident from the **.Internal(inspect())** command, revealing that the memory location of the vector where the change was made to an element was different compared to the memory locations before the changes was made. In this case, R does not make the change in place. It creates a copy of the vector being changed, makes the change to the element and then points to the new memory location where the changed vector resides.

2.2 Part (b)

```
#Create a copy of list_of_vecs
copy1 <- list_of_vecs
#Determine memory locations of the copy
.Internal(inspect(copy1))

## @2293e30 19 VECSXP g0c3 [MARK,NAM(2)] (len=3, tl=0)
##   @2778198 13 INTSXP g0c2 [MARK,NAM(2)] (len=3, tl=0) 1,2,3
##   @aa1b40 14 REALSXP g0c7 [MARK] (len=24, tl=0) 50,51,52,53,54,...
##   @20bbc08 13 INTSXP g0c5 [MARK,NAM(2)] (len=24, tl=0) 22,23,24,25,26,...

#Make a change to original list of vectors
list_of_vecs[[3]][[3]] <- 72
#Look at memory locations of both lists
#of vectors
.Internal(inspect(list_of_vecs))

## @1720d70 19 VECSXP g0c3 [NAM(1)] (len=3, tl=0)
##   @2778198 13 INTSXP g0c2 [MARK,NAM(2)] (len=3, tl=0) 1,2,3
##   @aa1b40 14 REALSXP g0c7 [MARK,NAM(2)] (len=24, tl=0) 50,51,52,53,54,...
##   @d70050 14 REALSXP g0c7 [] (len=24, tl=0) 22,23,72,25,26,...

.Internal(inspect(copy1))

## @2293e30 19 VECSXP g0c3 [MARK,NAM(2)] (len=3, tl=0)
##   @2778198 13 INTSXP g0c2 [MARK,NAM(2)] (len=3, tl=0) 1,2,3
##   @aa1b40 14 REALSXP g0c7 [MARK,NAM(2)] (len=24, tl=0) 50,51,52,53,54,...
##   @20bbc08 13 INTSXP g0c5 [MARK,NAM(2)] (len=24, tl=0) 22,23,24,25,26,...
```

Upon making a copy of the original list of vectors, the copy merely points to the memory locations of the original list of vectors. Copy-on-change is occurring when the change is made to the original list of vectors. The copy of the list of vectors maintains the same memory locations as shown by the two calls of the **.Internal(inspect())** of the copy of the list of vectors before and after a change is made to the original list of vectors. In the original list of vectors, a copy of the vector changed within the list is made and the other vectors within the list maintain the same memory location, indicating they were not copied.

2.3 Part (c)

```
#Create list of lists
list_of_lists <- list(list(1:25), list(3:7), list(90:100))
#Make copy of list
copy2 <- list_of_lists
#Look at the memory locations of original and copy
.Internal(inspect(list_of_lists))

## @1720a58 19 VECSXP g0c3 [NAM(2)] (len=3, tl=0)
##   @2494178 19 VECSXP g0c1 [] (len=1, tl=0)
##     @2b10ad8 13 INTSXP g0c5 [] (len=25, tl=0) 1,2,3,4,5,...
##   @24941a8 19 VECSXP g0c1 [] (len=1, tl=0)
##     @1720a10 13 INTSXP g0c3 [] (len=5, tl=0) 3,4,5,6,7
##   @24941d8 19 VECSXP g0c1 [] (len=1, tl=0)
##     @2579138 13 INTSXP g0c4 [] (len=11, tl=0) 90,91,92,93,94,...

.Internal(inspect(copy2))

## @1720a58 19 VECSXP g0c3 [NAM(2)] (len=3, tl=0)
##   @2494178 19 VECSXP g0c1 [] (len=1, tl=0)
##     @2b10ad8 13 INTSXP g0c5 [] (len=25, tl=0) 1,2,3,4,5,...
##   @24941a8 19 VECSXP g0c1 [] (len=1, tl=0)
##     @1720a10 13 INTSXP g0c3 [] (len=5, tl=0) 3,4,5,6,7
##   @24941d8 19 VECSXP g0c1 [] (len=1, tl=0)
##     @2579138 13 INTSXP g0c4 [] (len=11, tl=0) 90,91,92,93,94,...

#Add element to copy of list_of_lists
copy2[[2]][[1]] <- append(copy2[[2]][[1]], 1001)
#Look at the memory locations of original and copy
#After adding element to copy2
.Internal(inspect(list_of_lists))

## @1720a58 19 VECSXP g0c3 [NAM(2)] (len=3, tl=0)
##   @2494178 19 VECSXP g0c1 [NAM(2)] (len=1, tl=0)
##     @2b10ad8 13 INTSXP g0c5 [] (len=25, tl=0) 1,2,3,4,5,...
##   @24941a8 19 VECSXP g0c1 [NAM(2)] (len=1, tl=0)
##     @1720a10 13 INTSXP g0c3 [NAM(2)] (len=5, tl=0) 3,4,5,6,7
##   @24941d8 19 VECSXP g0c1 [NAM(2)] (len=1, tl=0)
##     @2579138 13 INTSXP g0c4 [] (len=11, tl=0) 90,91,92,93,94,...

.Internal(inspect(copy2))

## @29bd5f0 19 VECSXP g0c3 [NAM(1)] (len=3, tl=0)
##   @2494178 19 VECSXP g0c1 [NAM(2)] (len=1, tl=0)
##     @2b10ad8 13 INTSXP g0c5 [] (len=25, tl=0) 1,2,3,4,5,...
##   @16f4fe8 19 VECSXP g0c1 [] (len=1, tl=0)
##     @246cc80 14 REALSXP g0c4 [NAM(2)] (len=6, tl=0) 3,4,5,6,7,...
##   @24941d8 19 VECSXP g0c1 [NAM(2)] (len=1, tl=0)
##     @2579138 13 INTSXP g0c4 [] (len=11, tl=0) 90,91,92,93,94,...
```

After creating a list of lists and then adding an element to the second list contained within the copy of the list of lists, I found that first and third lists that were unchanged between the copy and the original, were shared between the original and the copy of the list of lists. The second lists between the original and the copy are different, indicating that it was copied in order append an element to the end of the second

list within the copy. The memory locations of the entire original list is different compared to copy. This indicates that copy of the list was copied to a new memory location when the change was made.

2.4 Part (d)

While the overall size of **tmp** is 160 MB according to *object.size(tmp)*, but from invoking *gc()* I observe that only 80 MB is used. When I invoke *.Internal(inspec(tmp))*, I see that that both elements in tmp point to the same memory location. This makes sense because both elements in **tmp** are comprised of the list of numbers, x. Thus, while the total size of *tmp* may be 160 MB, R only uses 80 MB because both elements within **tmp** point to the memory location where x is stored.

3 Problem 3

```
load('ps4prob3.Rda') # should have A, n, K
library("microbenchmark")
# Generates the log likelihood
ll <- function(Theta, A) {
  sum.ind <- which(A==1, arr.ind=T)
  logLik <- sum(log(Theta[sum.ind])) - sum(Theta)
  return(logLik)
}
#Original Code
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))
}
#Updated version of the function
```

```r
#Same initial conditions starting up
oneUpdate.new.new <- 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))
  q.new <- array(0, dim = c(n, n))

  theta.new <- theta.old
  #Instead of running with 3 for-loops, creating a multi-dimensional space
  #for performing the necessary calculations
  #Invoking the kroeneker product yields the same matrix
  #No conditional checks as well
  for (z in 1:K) {
        # q.new <- theta.old[1:n,z] * theta.old[1:n,z] / Theta.old[1:n,1:n]
    q.new <- matrix(kronecker(theta.old[1:n,z], theta.old[1:n,z]), nrow = n, ncol = n) /
      Theta.old

    theta.new[,z] <- rowSums(A*q.new)/sqrt(sum(A*q.new))
  }
  #same as the original code
  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)

#Proof of timing
microbenchmark(oneUpdate(A, n, K, theta.init), times=2)

## Unit: seconds
##                           expr      min       lq     mean   median
##  oneUpdate(A, n, K, theta.init) 79.99035 79.99035 80.95384 80.95384
##        uq      max neval
##  81.91733 81.91733     2

microbenchmark(oneUpdate.new.new(A, n, K, theta.init), times=2)

## Unit: seconds
##                                   expr      min       lq     mean
##  oneUpdate.new.new(A, n, K, theta.init) 1.228801 1.228801 1.230336
##    median      uq     max neval
##  1.230336 1.23187 1.23187     2

a <- oneUpdate(A, n, K, theta.init)
c <- oneUpdate.new.new(A, n, K, theta.init)

#Proof of identity
table(unlist(a$theta - c$theta))
```

```
##
##     0
## 50500
```

The issue with the original code is the iteration over three for-loops to process the large matrix required for the optimization of the statisical model. The original code requires that the array is generated by iterating over each of the points in the generated array. This stepwise fashion, coupled with the conditional checks at each indice within the array, results in the long computation time for a single iteration of the code, let alone multiple callings of the *oneUpdate()* function required for the full optimzation of the problem. The final for-loop that is used to go over the array again to complete the optimzation check steps further increases the computation time required for the execution of this function.

This function can be improved with the elimination of the three nested for-loops and replaced with the invoking of the kroeneker product. The Kroeneker product conducts the same multiplcation as is being done within the for-loops of the original function, but it does not require the iteration over every indice of the large array. Being a function already present within R, invoking kroeneker to generate the necessary array required for the optimization, it already has an optimized efficiency for handling computation of large arrays/matrices. This reduces all the three nested for-loops to a single for-loop to iterate over.

The new version of the function is compared with old function in terms of computational time and is checked in order to determine that the new function was able to generate the same function as the original. From the microbenchmark check, the new function is considerably more efficient at completing the task and also results in the same matrix/array generated by the original function.

4 Problem 4

I chose to modify PIKK in order to attempt an improvement of the efficiency of this algorithm. The original function is below.

```
library("microbenchmark")
vec <- rnorm(1e5)
k <- 500
PIKK <- function(x, k) {
  x[sort(runif(length(x)), index.return = TRUE)$ix[1:k]]
}
microbenchmark(PIKK(vec,k))

## Unit: milliseconds
##          expr      min       lq     mean  median       uq      max neval
##  PIKK(vec, k) 24.77533 25.34225 25.69916 25.6156 26.11165 28.06241   100
```

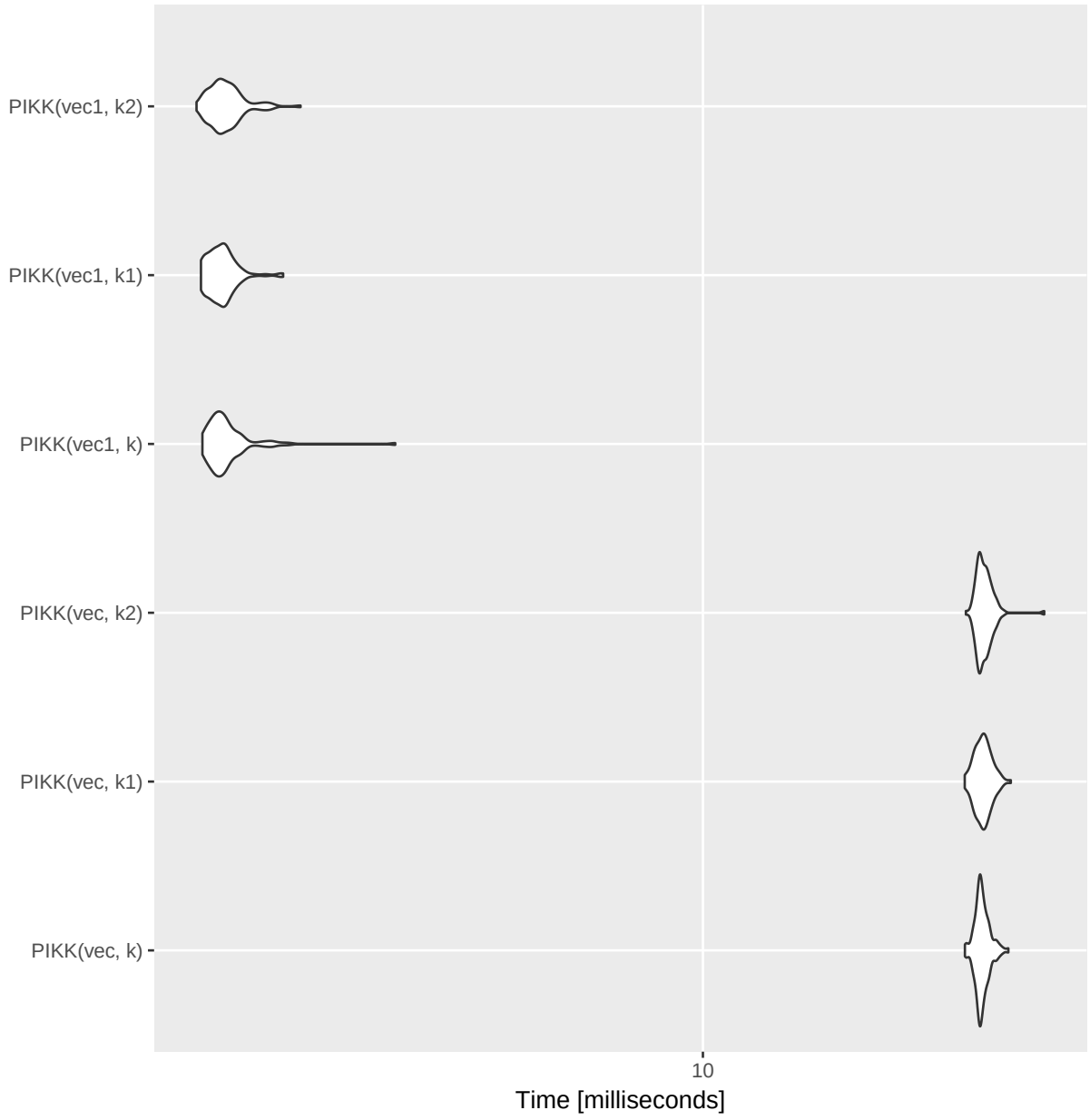
In the original version of PIKK, the vector is taken and it's length used to generate a vector of random deviates of a uniform distribution. This new vector is then sorted and the sorted indexes of this new list are used to pick k from the population, in this case the input vector, *vec*. Running it in this manner gives a mean time of 32.56 milliseconds. In order to improve it, I do the following –

```
PIKK_new <- function(x, k) {
  x[sample(length(x), k)]
}
```

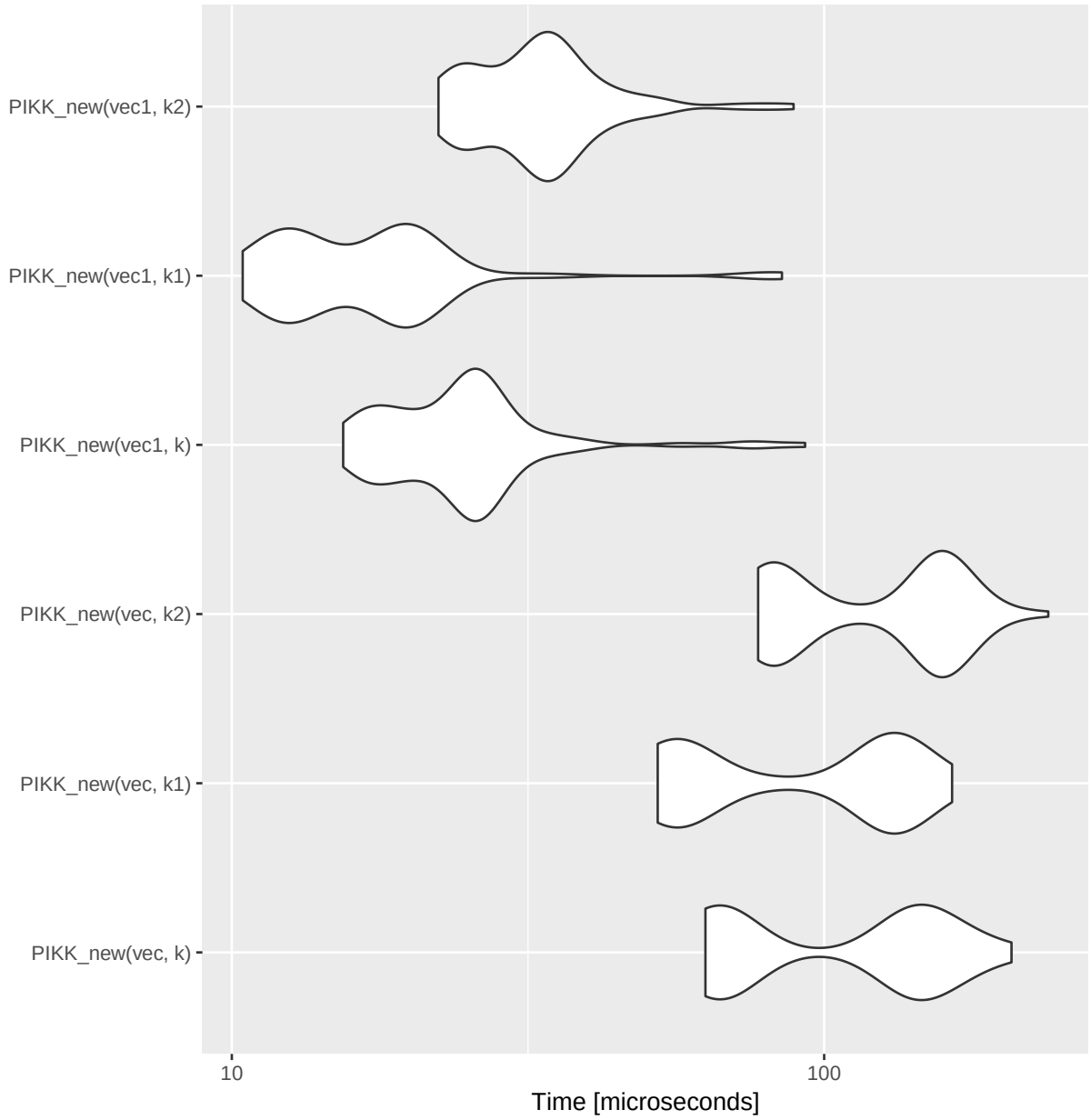
Taking advantage of the fast *sample()* function in R, the original PIKK function can be drastically sped up. Instead of generating another list, sorting it and then returning values from the indices of the generated vector for use in pick random values from the long vector supplied to the function, *sample()* does the same thing. THe increase in the efficiency of the function can be seen below.

```
library("ggplot2")
vec <- rnorm(1e5)
vec1 <- rnorm(1e4)
k <- 500
k1 <- 100
k2 <- 1000

#Run benchmark tests for old function and new function
#Using different values of k to sample
#and using two different length vectors
old_func <- microbenchmark(PIKK(vec, k), PIKK(vec, k1), PIKK(vec,k2),
                      PIKK(vec1,k), PIKK(vec1,k1), PIKK(vec1,k2))
new_func <- microbenchmark(PIKK_new(vec, k), PIKK_new(vec, k1), PIKK_new(vec,k2),
                      PIKK_new(vec1,k), PIKK_new(vec1,k1), PIKK_new(vec1,k2))
#plot using ggplot2
autoplot(old_func)
```




```
autoplot(new_func)
```



From the plots, it is clear that invoking the *sample()* function in the new function drastically increases the speed of completion compared to the original. The bottleneck of the original *PIKK* function is at the sorting step where the randomly generated distribution is sorted and the indices are taken from the lowest values in that vector. Removing that increases the efficiency of execution of the code.