

Problem Set 4

Natalia Sarabia Vasquez

October 15, 2021

1 Problem 1

Suppose I create a function called `range()` in my R session, as follows and then make my plot.

Explain in detail why can I still make a plot, including exactly where R looks for `range()` and where it finds it. As part of your answer, say what functions are on the call stack at the point that `range()` is called from within `plot()`.

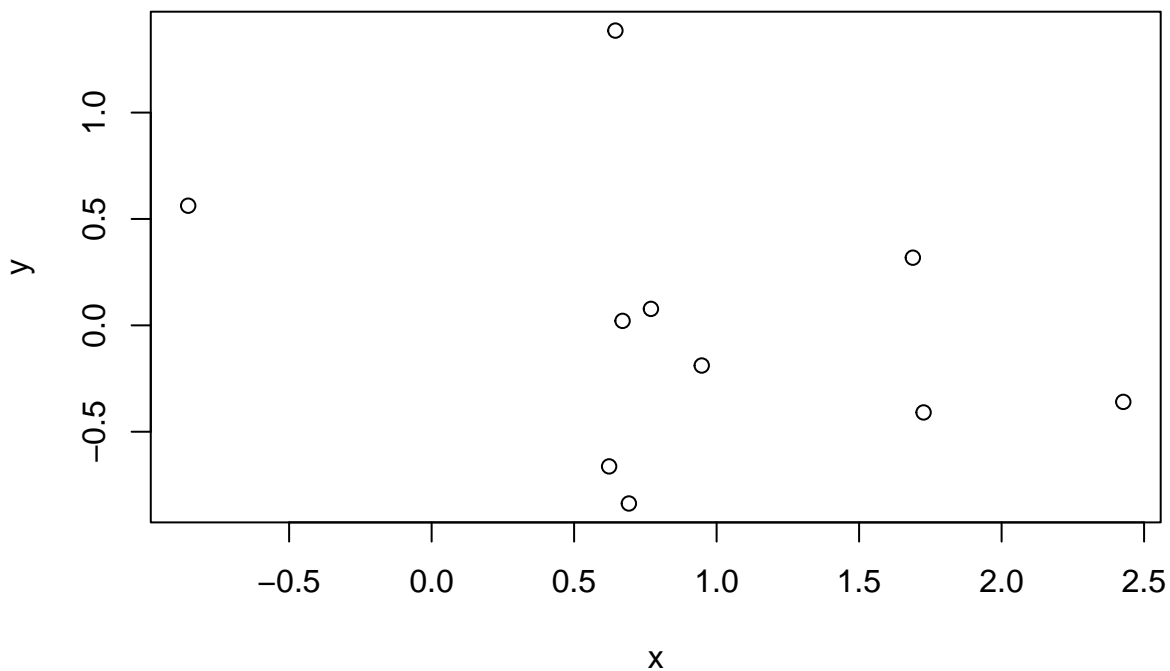
```
n <- 10
x <- rnorm(n)
y <- rnorm(n)
range <- function(...) cat("Good luck with that.\n")

range(rnorm(3))
```

```
## Good luck with that.
```

```
## Good luck with that.
```

```
plot(x, y)
```



The code is executed line by line in the order in which it was written. The assignments for the vectors `x`, `y`, and the function `range` are made in the Global environment. When I call the function `range`, which actually is defined in the Global environment, R looks for the function in the enclosing environment, which happens

to be the Global environment. That is the reason why I get the result of the function “range” I just defined lines above.

However, things change when I call the plot function. Once I execute the function plot(), the local environment (frame) for the function is created. In this frame, R calls the function “range” where the function “plot” was defined, rather than where it is called. As there, the function range is different from the one I created, everything continues working. This is the reason why I can still make a plot.

There are different methods associated to plot. In our case, R looks for plot.default as we reviewed in lecture:

```
head(methods(plot),n=15)
```

```
## [1] "plot.acf"           "plot.data.frame"    "plot.decomposed.ts"
## [4] "plot.default"      "plot.dendrogram"    "plot.density"
## [7] "plot.ecdf"         "plot.factor"        "plot.formula"
## [10] "plot.function"     "plot.hclust"        "plot.histogram"
## [13] "plot.HoltWinters"  "plot.isoreg"        "plot.lm"
```

When plot.default is called, there is a series of functions that are executed. If I review the source code for the function plot.default, I find that:

```
plot.default
```

```
## function (x, y = NULL, type = "p", xlim = NULL, ylim = NULL,
##     log = "", main = NULL, sub = NULL, xlab = NULL, ylab = NULL,
##     ann = par("ann"), axes = TRUE, frame.plot = axes, panel.first = NULL,
##     panel.last = NULL, asp = NA, xgap.axis = NA, ygap.axis = NA,
##     ...)
## {
##     localAxis <- function(..., col, bg, pch, cex, lty, lwd) Axis(...)
##     localBox <- function(..., col, bg, pch, cex, lty, lwd) box(...)
##     localWindow <- function(..., col, bg, pch, cex, lty, lwd) plot.window(...)
##     localTitle <- function(..., col, bg, pch, cex, lty, lwd) title(...)
##     xlabel <- if (!missing(x))
##         deparse1(substitute(x))
##     ylabel <- if (!missing(y))
##         deparse1(substitute(y))
##     xy <- xy.coords(x, y, xlabel, ylabel, log)
##     xlab <- if (is.null(xlab))
##         xy$xlab
##     else xlab
##     ylab <- if (is.null(ylab))
##         xy$ylab
##     else ylab
##     xlim <- if (is.null(xlim))
##         range(xy$x[is.finite(xy$x)])
##     else xlim
##     ylim <- if (is.null(ylim))
##         range(xy$y[is.finite(xy$y)])
##     else ylim
##     dev.hold()
##     on.exit(dev.flush())
##     plot.new()
##     localWindow(xlim, ylim, log, asp, ...)
##     panel.first
##     plot.xy(xy, type, ...)
##     panel.last
```

```
##   if (axes) {
##     localAxis(if (is.null(y))
##       xy$x
##     else x, side = 1, gap.axis = xgap.axis, ...)
##     localAxis(if (is.null(y))
##       x
##     else y, side = 2, gap.axis = ygap.axis, ...)
##   }
##   if (frame.plot)
##     localBox(...)
##   if (ann)
##     localTitle(main = main, sub = sub, xlab = xlab, ylab = ylab,
##       ...)
##   invisible()
## }
## <bytecode: 0x7ff62b9be608>
## <environment: namespace:graphics>
```

Let's give a closer look to the functions that are executed before the function `range`, when the function `plot` is called. By inspecting the `plot::default` function description. We can see that although there are different functions called before `range()`, they are executed and they are finished almost immediately. Therefore, in the call stack, we have `plot > plot::default > range`.

2 Problem 2

Original set-up

```
load('ps4prob2.Rda') # should have A, n, K

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)

# Do single update
# Timing of the original set up
benchmark(original_approach <- oneUpdate(A, n, K, theta.init),replications=1)

##                                test replications elapsed
## 1 original_approach <- oneUpdate(A, n, K, theta.init)      1  12.322
##   relative user.self sys.self user.child sys.child
## 1          1    12.213    0.103          0          0

```

Redefining the oneUpdate() function:

First, I used the hint that there were too many for loops (4) storing data that was not used for anything else but to sum the rows of the final results.

The 4 for loops of the original code had the following function (the deepest loop is the ‘first’ loop, z in this case):

- * First loop: Iterate over the new-created matrices of the 3D matrix.
- * Second and third loops: Fix a row i, iterate over the j column of the new matrix z and fill each entry.
- * Fourth loop: Compute the rowSum for each new matrix and divide by the sqrt of the sum of the new matrix.

All these steps can be avoided if we realize that:

1. I do not need to store the 3D matrix (even better, I do not need that matrix).
2. I do not need the if-else statement. The product is well defined and will just return a zero.
3. The entangled for loops fill the i,j entry of the z matrix. I can instead compute the zth matrix in each iteration. By doing this, the matrix changes in each iteration, I compute the operations I need and then store the final result in a vector.

```

# My improvements:

# I can note that the likelihood function can be improved by multiplying the
# two matrices. In doing so, you are "activating" just the entries you are
# interested on. Then, as I am going to have a bunch of log(0)=Inf I need
# to filter them out, that is the reason I am using is.finite()

ll <- function(Theta, A) {
  logLik <- sum(log(Theta * A) * is.finite(log(Theta * A))),
             na.rm = TRUE) - sum(Theta)
  return(logLik)
}

```

*# The main improvement is contained in the function oneUpdate. The key point is
 # realizing that I do not need to create a 3D matrix, and not even store that
 # enormous matrix. All I will do is to compute the required results and store
 # in the final output: theta.new:*

```
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)

  theta.new <- theta.old

  # Reduce four for loops to one:
  # Iterate over K, compute all the needed calculations, and store final results
  # in vector theta.new:
  for (i in 1:K){
    q <- A * (theta.old[,i] * t(matrix(theta.old[,i], nrow=500,
                                         ncol=500, byrow = FALSE)) / Theta.old)
    theta.new[,i] <- rowSums(q) / sqrt(sum(q))
  }

  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))
}
```

Timing of my approach

Do single update

```
benchmark(my_approach <- oneUpdate(A, n, K, theta.init),replications=1)
```

```
##                               test replications elapsed relative
## 1 my_approach <- oneUpdate(A, n, K, theta.init)           1   5.008       1
##   user.self sys.self user.child sys.child
## 1      4.766   0.058         0         0
```

Comparing I get the same result under both approaches

I get the same result under both approaches

```
identical(original_approach$theta,my_approach$theta)
```

```
## [1] TRUE
```

```
identical(original_approach$loglik,my_approach$loglik)
```

```
## [1] TRUE
```

```
identical(original_approach$converged,my_approach$converged)
```

```
## [1] TRUE
```

Conclusion: My approach has the same results as the original approach. I was able to decrease the time it took to compute all the calculations by noticing that not all for loops were necessary.

3 Problem 3

3.1 Question 3a)

```
load("mixedMember.Rda")

# Calculate the product using sapply for both, A and B:

# You can use sapply and iterate over a vector. This will help us to have access
# to each entry of the lists:

results_A <- sapply(1:length(wgtsA),
                    function(y) sum(wgtsA[[y]]*muA[IDsA[[y]]]))
head(results_A, n = 5)

## [1] -0.5399706 -0.6823306 -0.4041434 -0.2480350  0.4406208

results_B <- sapply(1:length(wgtsB),
                    function(y) sum(wgtsB[[y]]*muB[IDsB[[y]]]))
head(results_B, n = 5)

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

The idea in my solutions for 3b and 3c will be try to work with matrices and vectors instead of lists. I will work with the data representation in order to deal with linear algebra operations. Keep in mind the instructions provided by the professor, no modifications should be done to muA or muB (this in the sense that we should work with them as vectors).

3.2 Question 3b)

Main idea: make use of the fact that muA has bigger dimension than in case c.

```
# Note: This code was originally thought for IDsA, muA and wgtsA. I created the
# the function to make the comparison easier in 3d. For this reason, some of
# comments on the function will refer to A.

# Create a matrix with all the information of the IDsA (originally stored as a list).
# ASIDE: I could have done this by using sapply, but as we were not allowed to do that,
# I am using for loops.

# First, allocate some space for the matrix. The idea here is to convert the list into a
# matrix. The problem will be that not all entries have the same number of elements,
# but those places will be filled with NAs:
approach_b <- function(col,IDs,wgts,mu){
  myIDs <- matrix(NA,ncol = col,nrow=length(IDs))

  for(i in seq_len(length(IDs))){
    for(j in seq_len(length(IDs[[i]]))){
      # Key part: the [j]th element of the [[i]]th element of the list will be the entry i,j of
      # our new matrix
      myIDs[i,j] <- IDs[[i]][j]
    }
  }
```

```

}

# Create a matrix with all the information of the weightsA (originally stored as a list):
# ASIDE: I could have done this by using sapply, but as we were not allowed to do that,
# I am using for loops.
my_weights <- matrix(NA, ncol = col, nrow=length(IDs))

# The same idea applies to the list of weights:
for(i in seq_len(length(wgts))){
  for(j in seq_len(length(wgts[[i]]))){
    my_weights[i,j] <- wgts[[i]][j]
  }
}

return(list(my_weights, myIDs))
}

# The last step will be to use the new ID matrix to subset the vector mu. Then,
# to multiply it by the new weight matrix. Finally, to perform rowsums across the
# result of the last step.

```

I am getting the same result under both approaches for the calculations of A:

```

my_matricesA <- approach_b(8, IDsA, wgtsA, muA)
my_weightsA <- my_matricesA[[1]]
myIDsA <- my_matricesA[[2]]

# Key part: I can subset a vector giving it a matrix with possible entries of it.
# If I subset the vector using a nxm matrix, the result will be a (nxm)x1 vector.
# In other words, it will have nxm elements.

# This might sound very strange. How can I proceed our calculations if now, the
# most natural idea will be to multiply our matrix with weights by the subseted
# vector? How could this be possible if the do not have the same dimensions?

# Well, it is possible. R will allow me to multiply both of them and will compute the
# multiplication taking the muA-subseted vector as a disentangled version of a matrix.
# As the dimensions of the disentangled matrix check, there is no problem associated
# to dimensions. Once I do that, I just can sum the rows of the result and I have
# the desired output:

my_resultA <- rowSums(my_weightsA * muA[myIDsA], na.rm = TRUE)

head(my_resultA, n = 5)

## [1] -0.5399706 -0.6823306 -0.4041434 -0.2480350  0.4406208

# Comparing results under sapply and my approach on 3b):
identical(my_resultA, results_A)

## [1] TRUE

```

3.3 Question 3c)

Main idea: make use of the fact that muA has smaller dimension than in case c.

Under this approach, the idea is to work with the matrix of weights. In order to perform a matrix operation, I would like to find a way to arrange the data in the matrix weight so that when I compute the product between it and `muB`, only the entries I am interested on are the ones that actually have values. In fact, what I did was to map those entries in the weights matrix using the `IDsB` list. The idea is that when I want to “activate” an entry of the vector `muB`, I should place the weight element in the entry determined by `IDsB`. this changes will be done in a new weights matrix I will define:

```
# Note: This code was originally thought for IDsB, muB and wgtsB. I created the
# the function to make the comparison easier in 3d. For this reason, some of
# comments on the function will refer to B.

approach_c <- function(IDs, wgts, mu){
  # Create a matrix to allocate the weights in the order described by IDsB:
  my_weights <- matrix(0, ncol = length(mu), nrow=length(wgts))

  # The entry of my_weightsB is determined by IDsB, and there I put the value
# of the weight j for the individual i:
  for(i in seq_len(length(wgts))){
    for(j in seq_len(length(wgts[[i]]))){
      my_weights[i, (IDs[[i]][j])] <- wgts[[i]][j]
    }
  }

  return(my_weights)
}

my_weightsB <- approach_c(IDsB, wgtsB, muB)

# The last step will be to multiply my weights (matrix obtained with approach_c()
# function) by its correspondent mu
```

I am getting the same result under both approaches for the calculations of B:

Note: Somehow the precision changes if I use matrix multiplications. Therefore, I will compare the numbers by defining a tolerance. Any difference between two entries that is smaller than my condition will return a 1. In this case, the sum of my booleans is 100,000 and therefore the results obtained by both approaches are equivalent.

```
# To finish, I just have to multiply the matrix of weights placed in the appropriate entries
# by the vector mu (in this case, a matrix of one column):
my_resultB <- my_weightsB %*% muB
head(my_resultB)

##           [,1]
## [1,] -0.4496267
## [2,] -0.3697111
## [3,] -0.2104093
## [4,] -0.3426966
## [5,] -0.3874494
## [6,]  0.6585238

# Comparing results under supply and my approach on 3c):
tol = 1e-12

sum(abs(my_resultB-results_B) <= tol)

## [1] 100000
```


In other words, I am obtaining the same result under both approaches.

3.4 Question 3d)

I compared speed for the two test cases using three different variants.

3.4.1 Comparing the results for A under the three approaches:

```
# Using sapply:
benchmark(results_A <- sapply(1:length(wgtsA),
                             function(y) sum(wgtsA[[y]]*muA[IDsA[[y]]])),
          replications = 1)

##                                     test
## 1 results_A <- sapply(1:length(wgtsA), function(y) sum(wgtsA[[y]] * muA[IDsA[[y]]]))
##   elapsed relative user.self sys.self user.child sys.child
## 1      1      0.174      1      0.172      0.002      0      0

# Using approach in b)
benchmark(my_resultAb <- rowSums(my_weightsA * muA[myIDsA], na.rm = TRUE), replications = 1)

##                                     test replications
## 1 my_resultAb <- rowSums(my_weightsA * muA[myIDsA], na.rm = TRUE)      1
##   elapsed relative user.self sys.self user.child sys.child
## 1  0.075      1  0.075      0      0      0

# Using approach in c)
my_weightsAc <- approach_c(IDsA, wgtsA, muA)
benchmark(my_resultAc <- my_weightsAc %*% muA, replications = 1)

##                                     test replications elapsed relative user.self
## 1 my_resultAc <- my_weightsAc %*% muA      1  0.081      1  0.08
##   sys.self user.child sys.child
## 1      0      0      0

Finally, just to show that I have the same result under the three approaches:
head(results_A, n = 5)

## [1] -0.5399706 -0.6823306 -0.4041434 -0.2480350  0.4406208
head(my_resultAb, n = 5)

## [1] -0.5399706 -0.6823306 -0.4041434 -0.2480350  0.4406208
head(my_resultAc, n = 5)

##           [,1]
## [1,] -0.5399706
## [2,] -0.6823306
## [3,] -0.4041434
## [4,] -0.2480350
## [5,]  0.4406208
```

If I compare the time elapsed under the three calculations, it is clear that the approach 3b was the best in this case. After all, it was a good idea to consider the difference in the mu vector size compared to 3c. Although here the difference might appear not very significant, when we are working with bigger matrices, this improvement makes all the difference.

3.4.2 Comparing the results for B under the three approaches:

```
# Using sapply:
benchmark(results_B <- sapply(1:length(wgtsB),
                             function(y) sum(wgtsB[[y]]*muB[IDsB[[y]]])),
          replications = 1)

##                                     test
## 1 results_B <- sapply(1:length(wgtsB), function(y) sum(wgtsB[[y]] * muB[IDsB[[y]]]))
##   replications elapsed relative user.self sys.self user.child sys.child
## 1           1    0.222         1    0.221    0.001         0         0

# Using approach in b)
my_matricesB <- approach_b(10,IDsB,wgtsB,muB)
my_weightsBb <- my_matricesB[[1]]
myIDsB <-my_matricesB[[2]]

benchmark(my_resultBb <- rowSums(my_weightsBb * muB[myIDsB],na.rm = TRUE), replications = 1)

##                                     test replications
## 1 my_resultBb <- rowSums(my_weightsBb * muB[myIDsB], na.rm = TRUE)          1
##   elapsed relative user.self sys.self user.child sys.child
## 1    0.097         1    0.095    0.001         0         0

# Using approach in c)
benchmark(my_resultBc <- my_weightsB %*% muB, replications = 1)

##                                     test replications elapsed relative user.self
## 1 my_resultBc <- my_weightsB %*% muB          1    0.001         1    0.001
##   sys.self user.child sys.child
## 1         0         0         0

If I compare the results under the three approaches for this case, approach c was superior here, which goes according to what we were expecting. I made use of the fact that the vector size differs under each case. In c, muB has smaller dimension.

Finally, just to show that I have the same result under the three approaches:

head(results_B, n = 5)

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

head(my_resultBb, n = 5)

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

head(my_resultBc, n = 5)

##           [,1]
## [1,] -0.4496267
## [2,] -0.3697111
## [3,] -0.2104093
## [4,] -0.3426966
## [5,] -0.3874494
```

4 Problem 4

```
# Consider a list of numeric vectors
x <- list(rnorm(3), rnorm(3))
```

4.1 Question 4a)

Modify one of the elements of the vectors. Can R make the change in place, without creating a new vector?

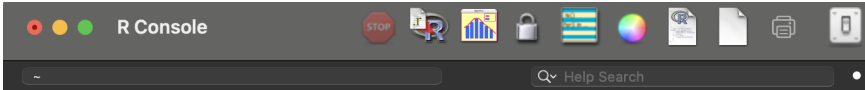
```
.Internal(inspect(x))
original <- address(x)

x[[1]][2] <- 1
.Internal(inspect(x))
modified <- address(x)

identical(original, modified)
```

Yes, R can make the change in place, without creating a new vector. When I execute the code above in the terminal we obtain the following output. I see that the addresses of the original list and the copy one are the same and the only change I can appreciate is the one of the address of the modified vector:

```
include_graphics(file.path('..', 'PS4/Problem4a.png'))
```



```
> # Problem 4a
> # Create the list x
> x <- list(rnorm(3), rnorm(3))
>
> # Inspect its address
> .Internal(inspect(x))
@7fc3ec606488 19 VECSXP g0c2 [REF(1)] (len=2, tl=0)
  @7fc3e6a9ca78 14 REALSXP g0c3 [REF(1)] (len=3, tl=0) -0.275354,0.207954,1.72825
  @7fc3e6a9cac8 14 REALSXP g0c3 [REF(1)] (len=3, tl=0) -0.516808,2.09814,0.388973
> original <- address(x)
>
> # Modify one element of the list x
> x[[1]][2] <- 1
>
> # Inspect how address changed in the object after the modification
> .Internal(inspect(x))
@7fc3ec606488 19 VECSXP g0c2 [REF(1)] (len=2, tl=0)
  @7fc3e6a9ca78 14 REALSXP g0c3 [REF(1)] (len=3, tl=0) -0.275354,1,1.72825
  @7fc3e6a9cac8 14 REALSXP g0c3 [REF(1)] (len=3, tl=0) -0.516808,2.09814,0.388973
> modified <- address(x)
>
> # Are the addresses of the modified and original lists, equal?
> identical(original, modified)
[1] TRUE
>
```

Figure 1: Can R make the change in place?

4.2 Question 4b)

Next, make a copy of the list and determine if there any copy-on-change going on.

```
# Making a copy
y <- x

# The addresses will be the same because R only makes copies as needed
identical(address(y), address(x))
```

```
# Here is even clearer that both lists are exactly the same
.Internal(inspect(y))
.Internal(inspect(x))
```

There is no copy-on-change, the copy just points out to the original vector:

```
include_graphics(file.path('.', 'PS4/Problem4b.png'))
```

```
> # Problem 4b
> # Making a copy
> y <- x
>
> # The addresses will be the same because R only makes copies as needed
> identical(address(y),address(x))
[1] TRUE
>
> # Here is even clearer that both lists are the same
> .Internal(inspect(y))
@7fc3ec606488 19 VECSXP g0c2 [REF(2)] (len=2, tl=0)
  @7fc3e6a9ca78 14 REALSXP g0c3 [REF(1)] (len=3, tl=0) -0.275354,1,1.72825
  @7fc3e6a9cac8 14 REALSXP g0c3 [REF(1)] (len=3, tl=0) -0.516808,2.09814,0.388973
> .Internal(inspect(x))
@7fc3ec606488 19 VECSXP g0c2 [REF(2)] (len=2, tl=0)
  @7fc3e6a9ca78 14 REALSXP g0c3 [REF(1)] (len=3, tl=0) -0.275354,1,1.72825
  @7fc3e6a9cac8 14 REALSXP g0c3 [REF(1)] (len=3, tl=0) -0.516808,2.09814,0.388973
> |
```

Figure 2: Creating a copy

When a change is made to one of the vectors in one of the lists, is a copy of the entire list made or just of the relevant vector?

Just a copy of the relevant vector is made. I followed two approaches: 1. I made changes to the copy and 2. I made changes to the original:

Modifying the copy and analyzing how the location in memory of both changes:


```
# Making a change to one of the vectors in the copy
y[[1]][3] <- 90

# It creates a new allocation in memory for the modified vector and the modified list.
# Although the address of the vector we did not change, stays intact.
# This validation gives us FALSE, because in fact, the lists are not the same any more
identical(address(y),address(x))

# Here we can appreciate what was said before. The modified list has a new address,
# as well as the modified vector. However, the intact vector remains with its original address:
.Internal(inspect(y))
.Internal(inspect(x))
```

It creates a new address for the modified list and the modified vector. The one that was not changed keeps its original address:

```
include_graphics(file.path('..', 'PS4/Problem4b1.png'))
```



```
> # First approach: changing the copy
> # Making a change to one of the vectors in the copy
> y[[1]][3] <- 90
>
> # It creates a new allocation in memory for the modified y, and the list that contains it.
> # However, the address of the vector I did not change, stays intact.
> # The following validation gives us FALSE, because in fact, the lists are not the same any more:
> identical(address(y), address(x))
[1] FALSE
>
> # Here we can appreciate what was said before. The modified list has a new address, as well as
> # the modified vector. However, the intact vector remains with its original address:
> .Internal(inspect(y))
@7fc3ec606548 19 VECSXP g0c2 [REF(1)] (len=2, tl=0)
  @7fc3e6a9cf28 14 REALSXP g0c3 [REF(1)] (len=3, tl=0) -0.275354,1,90
  @7fc3e6a9cac8 14 REALSXP g0c3 [REF(2)] (len=3, tl=0) -0.516808,2.09814,0.388973
> .Internal(inspect(x))
@7fc3ec606488 19 VECSXP g0c2 [REF(1)] (len=2, tl=0)
  @7fc3e6a9ca78 14 REALSXP g0c3 [REF(1)] (len=3, tl=0) -0.275354,1,1.72825
  @7fc3e6a9cac8 14 REALSXP g0c3 [REF(2)] (len=3, tl=0) -0.516808,2.09814,0.388973
>
```

Figure 3: Modifying the copy

Modifying the original and analyzing how the memory location of both, the original and the copy, changes:

```
gc()
rm(x)
rm(y)
x <- list(rnorm(3), rnorm(3))

y <- x

# Both have the same location in memory
.Internal(inspect(x))
.Internal(inspect(y))

# I change the original
x[[1]][3] <- 90

# x has its location in memory changed
.Internal(inspect(x))
# y maintains the location in memory of the original
.Internal(inspect(y))
```

In this case, when I change the original, as y is pointing to x, the address of x changes and also the one of the modified element. y keeps the original address of x and also its elements:

```
include_graphics(file.path('..', 'PS4/Problem4b2.png'))
```

```

R Console
> # Second approach: changing the original
> # Secure myself that there is no x and y
> gc()
      used (Mb) gc trigger (Mb) limit (Mb) max used (Mb)
Ncells 992791 53.1  1572320 84.0      NA 1572320 84.0
Vcells 10153416 77.5  33072055 252.4   16384 53639105 409.3
> rm(x)
> rm(y)
>
> # Create list x
> x <- list(rnorm(3), rnorm(3))
>
> # Create a copy of list x
> y <- x
>
> # Both have the same location in memory
> .Internal(inspect(x))
@7fc3ec606348 19 VECSXP g0c2 [REF(2)] (len=2, tl=0)
 @7fc3e6a9c758 14 REALSXP g0c3 [REF(1)] (len=3, tl=0) 0.846272,0.0252992,2.21505
 @7fc3e6a9c7a8 14 REALSXP g0c3 [REF(1)] (len=3, tl=0) -0.783119,-0.184533,0.106774
> .Internal(inspect(y))
@7fc3ec606348 19 VECSXP g0c2 [REF(2)] (len=2, tl=0)
 @7fc3e6a9c758 14 REALSXP g0c3 [REF(1)] (len=3, tl=0) 0.846272,0.0252992,2.21505
 @7fc3e6a9c7a8 14 REALSXP g0c3 [REF(1)] (len=3, tl=0) -0.783119,-0.184533,0.106774
>
> # I change the original
> x[[1]][3] <- 90
>
> # x has its location in memory changed
> .Internal(inspect(x))
@7fc3ec6063c8 19 VECSXP g0c2 [REF(1)] (len=2, tl=0)
 @7fc3e6a9c938 14 REALSXP g0c3 [REF(1)] (len=3, tl=0) 0.846272,0.0252992,90
 @7fc3e6a9c7a8 14 REALSXP g0c3 [REF(2)] (len=3, tl=0) -0.783119,-0.184533,0.106774
> # y maintains the location in memory of the original
> .Internal(inspect(y))
@7fc3ec606348 19 VECSXP g0c2 [REF(1)] (len=2, tl=0)
 @7fc3e6a9c758 14 REALSXP g0c3 [REF(1)] (len=3, tl=0) 0.846272,0.0252992,2.21505
 @7fc3e6a9c7a8 14 REALSXP g0c3 [REF(2)] (len=3, tl=0) -0.783119,-0.184533,0.106774
>

```

Figure 4: Modifying the original

4.3 Question 4c)

Suppose you grow a list in the inefficient way we discussed for vectors in the efficient R tutorial as seen below. Given what you've learned above about the structure of a list, if at the end you have n elements in the list, how many bytes are unnecessarily copied (relative to the situation of you creating an empty list of length n in advance) when running the following code?

```

n <- 10

myList <- list()

for(i in 1:n) {
  myList[[i]] <- rnorm(20)
  # Printing this gives us an idea of the extra copied elements:
  # However, I won't show the results here because they are quite long.
  # print(.Internal(inspect(myList)))
  print(object_size(myList))
}

```

- ## 264 B
- ## 480 B
- ## 704 B
- ## 912 B
- ## 1,136 B
- ## 1,344 B
- ## 1,568 B

```
## 1,776 B
## 2,048 B
## 2,256 B
```

I want to get the size of an “address”. The focus will be to create a large list with the same element. Given the above results, the list will store one copy of the data (given that the others are exactly the same), a bunch of metadata including the addresses of each element and other attributes. I expect the size of the attributes to be very small compared to the size of the addresses stored. The idea is to get the size of the object and subtract the size of the data. Finally, I will divide that number by the number of addresses we are expecting to have (in other words, by the number of elements of the list). The result will be the size of each address (with some small portion of the attributes storage):

```
# Create the list
myList <- vector("list", 1000)

default <- 12

# Fill the list with the same element
for(i in 1:1000){
  myList[[i]] <- default
}

# Size of the list without the data: size of the addresses plus the attributes
# For what we just learned from the above exercises, if we have the same element
# in the list, it will only store one copy of the data. The remaining size of the
# list will be the addresses and a small p :
size_addresses <- object_size(myList) - (object_size(default))

# Size of each address
size_addresses/1000
```

```
## 8.048 B
```

Now that I know the size of each address, I just need to know how many repetitions of the addresses I have at each iteration.

At each iteration, we have $\frac{(n-1)*n}{2}$ number of repetitions:

```
stargazer(as.data.frame(table), summary = FALSE, header = FALSE)
```

Table 1:

	Iteration	Number of repetitions
1	1	0
2	2	1
3	3	2
4	4	3
5
6	n	n-1

To respond to the question, how many bytes are unnecessarily copied (relative to the situation of you creating an empty list of length n in advance) when running the following code?

We have $\frac{(n-1)*n}{2}$ by roughly 8 bytes, n in this case is 10. So we have approximately 360 bytes unnecessarily copied.

5 Problem 5

5.1 Question 5a)

What is the maximum number of copies of the values represented by `x` that exist in memory during the execution of `scaler_constructor()`? Why?

During the execution of `scale_constructor()`, there are not copies created. Just the original. 'input' and 'data' point out to `x`.

```
# Original x
x <- rnorm(10)

scaler_constructor <- function(input){
# Here, the "copy" of x is just a promise.
# We have not even called the function on x. Note that even when the code
# is executed, there is not going to be such a copy because as we have not made any
# modification to "input" it still points to the original x.
  data <- input
  g <- function(param) return(param * data)
  return(g)
}

# No copies are created here, 'input' and 'data' point to the original 'x'.
scaler <- scaler_constructor(x)

# We can even comment this line. Although it will be executed, it does not
# modify the assignment made in the local environment of the function g, when it is created.
data <- 100

# Conclusion, there is just one version of x, the original, no copies. Yes, many
# objects point to it, but there are no copies.
scaler(3)
```

5.2 Question 5b)

Use `serialize()` to generate a sequence of bytes that store the information in the closure. Is the size of the serialized object the size you would expect given your answer to (a)? If not, can you explain what is happening? For this part of the problem, make `x` a large enough vector that your answer concentrates on the number of bytes involved in the numeric vector rather than in the function itself or any overhead for storing R objects.

```
# We create a large vector
x <- rnorm(10e5)

# Actual size of x in bytes
object_size(x)

## 8,000,048 B

# We are expecting that the serialization of the closure will be small. As we discussed
# in b), we are not expecting it to create any copies.
# Let's see what actually happens:

scaler_constructor <- function(input){
  data <- input
  g <- function(param) return(param * data)
```



```

    return(g)
}

scaler <- scaler_constructor(x)

# If we use the function object_size:
object_size(scaler)

## 8,013,544 B

# Length estimated by the serialization:
length(serialize(scaler, NULL))

## [1] 16008060

```

To begin with, serialization turns a complicated object into a block of bytes. It lets us know how long it is in terms of bytes.

No, it is not what we were expecting. I said that we do not have copies created. But somehow we got that the size of the closure it is twice the size of the object ‘x’. Why? The serialization is including the serialization of ‘input’ and ‘data’ but they are not actually a part of the memory that is being used by the function itself.

In other words, there is some memory being used associated with the data that is in the enclosing environment of scaler but that is not part of the actual memory that is being used by the actual function itself.

5.3 Question 5c)

It seems unnecessary to have the “data <- input” line, so let’s try the following.

```

x <- rnorm(10)
scaler_constructor <- function(data){
  g <- function(param) return(param * data)
  return(g)
}

# Here we are creating scaler, a copy of scaler_constructor()
scaler <- scaler_constructor(x)
rm(x)

data <- 100
scaler(3)

# We got the following error:
# Error in scaler(3) : object 'x' not found

```

Explain what is happening and why this doesn’t work to embed a constant data value into the function. Recall our discussion of when function arguments are evaluated.

The problem here is that we removed the vector x and, as we do not have a ‘copy’ of it or anything pointing to it, when scaler is executed, there is no ‘x’ (or anything associated with the location on memory that x used to have) to work with it. ‘Data’ is just a promise, and therefore the pointer is not created until needed (in this case, as we have done nothing to ‘data’, we do not have such a pointer). I lost connection with the block of memory to which x used to point at.

It does not work to embed a constant value into the function because the enclosing environment of the function g() is scaler_constructor()’s frame.

5.4 Question 5d)

As we said, there are no copies created. Therefore, the question is, how can I make the code above to execute correctly without doing `data<-input`?

A solution would be to make anything (whatever we choose to do) with ‘data’. This, in order to generate a pointer to ‘x’, and that even when it gets removed, I still have ‘data’ pointing to the space on memory where the label ‘x’ used to point at. In my case, I choose to multiply it by one:

```
x <- rnorm(10)

scaler_constructor <- function(data){
  # I choose to multiply data by one, but actually we can do whatever we want, as long
  # as we do not modify it
  data * 1
  g <- function(param) return(param * data)
  return(g)
}

scaler <- scaler_constructor(x)
rm(x)
data <- 100
scaler(3)

## [1] 4.9690478 -2.5519419 0.4732722 -1.0224511 -1.1317871 -3.9632439
## [7] -2.3405515 0.9894257 -4.7960284 -4.1569637
```

And then, the code works again.