It is usually said, that for– and while-loops should be avoided in R. There was a curiousity about just how the different alternatives compare in terms of speed.

The first loop is perhaps the worst it can be thought of – the return vector is initialized without type and length so that the memory is constantly being allocated.

use\_for\_loop <- function(x){

  y <- c()

  for(i in x){

    y <- c(y, x[i] \* 100)

  }

  return(y)

}

The second for loop is with preallocated size of the return vector.

use\_for\_loop\_vector <- function(x){

  y <- vector(mode = "double", length = length(x))

  for(i in x){

    y[i] <- x[i] \* 100

  }

  return(y)

}

I have noticed I use sapply() quite a lot, but I think not once have I used vapply() We will nonetheless look at both

use\_sapply <- function(x){

  sapply(x, function(y){y \* 100})

}

use\_vapply <- function(x){

  vapply(x, function(y){y \* 100}, double(1L))

}

And because I am a tidyverse-fanboy we also loop at map\_dbl().

library(purrr)

use\_map\_dbl <- function(x){

  map\_dbl(x, function(y){y \* 100})

}

We test the functions using a vector of random doubles and evaluate the runtime with microbenchmark.

x <- c(rnorm(100))

mb\_res <- microbenchmark::microbenchmark(

  `for\_loop()` = use\_for\_loop(x),

  `for\_loop\_vector()` = use\_for\_loop\_vector(x),

  `purrr::map\_dbl()` = use\_map\_dbl(x),

  `sapply()` = use\_sapply(x),

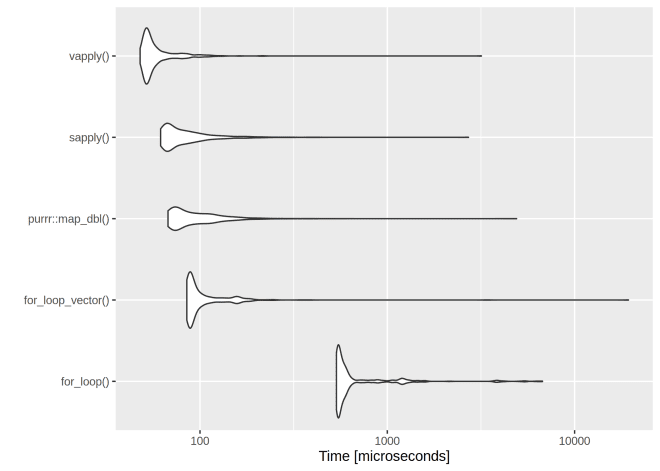
  `vapply()` = use\_vapply(x),

  times = 500

)

The results are listed in table and figure below.

| **expr** | **min** | **lq** | **mean** | **median** | **uq** | **max** | **neval** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| for\_loop() | 8.440 | 9.7305 | 10.736446 | 10.2995 | 10.9840 | 26.976 | 500 |
| for\_loop\_vector() | 10.912 | 12.1355 | 13.468312 | 12.7620 | 13.8455 | 37.432 | 500 |
| purrr::map\_dbl() | 22.558 | 24.3740 | 25.537080 | 25.0995 | 25.6850 | 71.550 | 500 |
| sapply() | 15.966 | 17.3490 | 18.483216 | 18.1820 | 18.8070 | 59.289 | 500 |
| vapply() | 6.793 | 8.1455 | 8.592576 | 8.5325 | 8.8300 | 26.653 | 500 |



The clear winner is vapply() and for-loops are rather slow. However, if we have a very low number of iterations, even the worst for-loop isn’t too bad:

x <- c(rnorm(10))

mb\_res <- microbenchmark::microbenchmark(

  `for\_loop()` = use\_for\_loop(x),

  `for\_loop\_vector()` = use\_for\_loop\_vector(x),

  `purrr::map\_dbl()` = use\_map\_dbl(x),

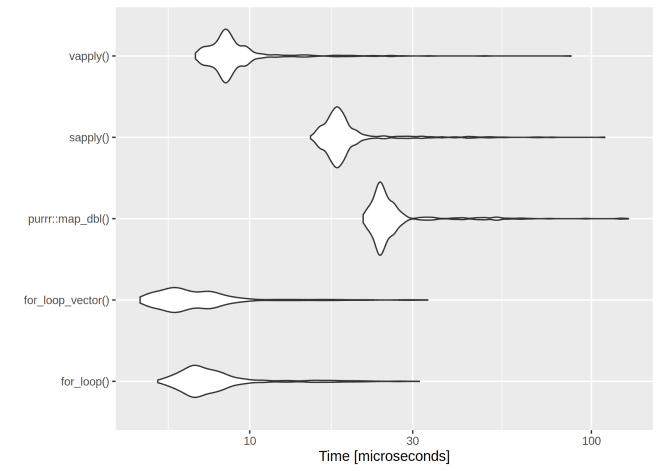
  `sapply()` = use\_sapply(x),

  `vapply()` = use\_vapply(x),

  times = 500

)

| **expr** | **min** | **lq** | **mean** | **median** | **uq** | **max** | **neval** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| for\_loop() | 5.992 | 7.1185 | 9.670106 | 7.9015 | 9.3275 | 70.955 | 500 |
| for\_loop\_vector() | 5.743 | 7.0160 | 9.398098 | 7.9575 | 9.2470 | 40.899 | 500 |
| purrr::map\_dbl() | 22.020 | 24.1540 | 30.565362 | 25.1865 | 27.5780 | 157.452 | 500 |
| sapply() | 15.456 | 17.4010 | 22.507534 | 18.3820 | 20.6400 | 203.635 | 500 |
| vapply() | 6.966 | 8.1610 | 10.127994 | 8.6125 | 9.7745 | 66.973 | 500 |



Apply Functions

**apply() function**

**apply()** takes Data frame or matrix as an input and gives output in vector, list or array. Apply function in R is primarily used to avoid explicit uses of loop constructs. It is the most basic of all collections can be used over a matrice.

This function takes 3 arguments:

apply(X, MARGIN, FUN)

Here:

-x: an array or matrix

-MARGIN: take a value or range between 1 and 2 to define where to apply the function:

-MARGIN=1`: the manipulation is performed on rows

-MARGIN=2`: the manipulation is performed on columns

-MARGIN=c(1,2)` the manipulation is performed on rows and columns

-FUN: tells which function to apply. Built functions like mean, median, sum, min, max and even user-defined functions can be applied>

The simplest example is to sum a matrice over all the columns. The code apply(m1, 2, sum) will apply the sum function to the matrix 5×6 and return the sum of each column accessible in the dataset.

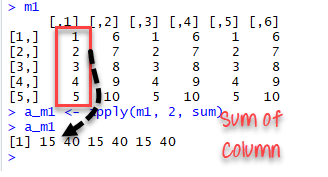
m1 <- matrix(C<-(1:10),nrow=5, ncol=6)

m1

a\_m1 <- apply(m1, 2, sum)

a\_m1

**Output:**



apply() function example in R

Best practice: Store the values before printing it to the console.

**lapply() function**

**lapply()** function is useful for performing operations on list objects and returns a list object of same length of original set. lappy() returns a list of the similar length as input list object, each element of which is the result of applying FUN to the corresponding element of list. Lapply in R takes list, vector or data frame as input and gives output in list.

lapply(X, FUN)

Arguments:

-X: A vector or an object

-FUN: Function applied to each element of x

l in lapply() stands for list. The difference between lapply() and apply() lies between the output return. The output of lapply() is a list. lapply() can be used for other objects like data frames and lists.

lapply() function does not need MARGIN.

A very easy example can be to change the string value of a matrix to lower case with tolower function. We construct a matrix with the name of the famous movies. The name is in upper case format.

movies <- c("SPYDERMAN","BATMAN","VERTIGO","CHINATOWN")

movies\_lower <-lapply(movies, tolower)

str(movies\_lower)

**Output:**

## List of 4

## $:chr"spyderman"

## $:chr"batman"

## $:chr"vertigo"

## $:chr"chinatown"

We can use unlist() to convert the list into a vector.

movies\_lower <-unlist(lapply(movies,tolower))

str(movies\_lower)

**Output:**

## chr [1:4] "spyderman" "batman" "vertigo" "chinatown"

**sapply() function**

**sapply()** function takes list, vector or data frame as input and gives output in vector or matrix. It is useful for operations on list objects and returns a list object of same length of original set. Sapply function in R does the same job as lapply() function but returns a vector.

sapply(X, FUN)

Arguments:

-X: A vector or an object

-FUN: Function applied to each element of x

We can measure the minimum speed and stopping distances of cars from the cars dataset.

dt <- cars

lmn\_cars <- lapply(dt, min)

smn\_cars <- sapply(dt, min)

lmn\_cars

**Output:**

## $speed

## [1] 4

## $dist

## [1] 2

smn\_cars

**Output:**

## speed dist

## 4 2

lmxcars <- lapply(dt, max)

smxcars <- sapply(dt, max)

lmxcars

**Output:**

## $speed

## [1] 25

## $dist

## [1] 120

smxcars

**Output:**

## speed dist

## 25 120

We can use a user built-in function into lapply() or sapply(). We create a function named avg to compute the average of the minimum and maximum of the vector.

avg <- function(x) {

( min(x) + max(x) ) / 2}

fcars <- sapply(dt, avg)

fcars

**Output**

## speed dist

## 14.5 61.0

Sapply in R is more efficient than lapply() in the output returned because sapply() store values direclty into a vector. In the next example, we will see this is not always the case.

We can summarize the difference between apply(), sapply() and `lapply() in the following table:

| **Function** | **Arguments** | **Objective** | **Input** | **Output** |
| --- | --- | --- | --- | --- |
| apply | apply(x, MARGIN, FUN) | Apply a function to the rows or columns or both | Data frame or matrix | vector, list, array |
| lapply | lapply(X, FUN) | Apply a function to all the elements of the input | List, vector or data frame | list |
| sapply | sapply(X, FUN) | Apply a function to all the elements of the input | List, vector or data frame | vector or matrix |

**Slice vector**

We can use lapply() or sapply() interchangeable to slice a data frame. We create a function, below\_average(), that takes a vector of numerical values and returns a vector that only contains the values that are strictly above the average. We compare both results with the identical() function.

below\_ave <- function(x) {

ave <- mean(x)

return(x[x > ave])

}

dt\_s<- sapply(dt, below\_ave)

dt\_l<- lapply(dt, below\_ave)

identical(dt\_s, dt\_l)

**Output:**

## [1] TRUE

**tapply() function**

**tapply()** computes a measure (mean, median, min, max, etc..) or a function for each factor variable in a vector. It is a very useful function that lets you create a subset of a vector and then apply some functions to each of the subset.

tapply(X, INDEX, FUN = NULL)

Arguments:

-X: An object, usually a vector

-INDEX: A list containing factor

-FUN: Function applied to each element of x

Part of the job of a data scientist or researchers is to compute summaries of variables. For instance, measure the average or group data based on a characteristic. Most of the data are grouped by ID, city, countries, and so on. Summarizing over group reveals more interesting patterns.

To understand how it works, let’s use the iris dataset. This dataset is very famous in the world of machine learning. The purpose of this dataset is to predict the class of each of the three flower species: Sepal, Versicolor, Virginica. The dataset collects information for each species about their length and width.

As a prior work, we can compute the median of the length for each species. Tapply in R is a quick way to perform this computation.

data(iris)

tapply(iris$Sepal.Width, iris$Species, median)

**Output:**

## setosa versicolor virginica

## 3.4 2.8 3.0