Programacion en R

Alvaro Navarro Cangas[*,1], Facundo Moyano[b], Kalindy Tillar[**,b], Federica Mosso[**,a], Emilia Millet[**,a], Juan Francisco Najul[**,a]

[a] *Department, Street, City, State, Zip*
[b] *Department, Street, City, State, Zip*

Abstract

This is the abstract.
It consists of two paragraphs.

Text based on elsarticle sample manuscript, see http://www.elsevier.com/author-schemas/latex-instructions#elsarticle

Elementos básicos de programación

En este breve tutorial examinaremos algunos elementos del lenguaje de programacion R y como valernos de ello para resolver problemas de la vida cotidiana. Apelaremos a ejemplos bien conocidos, pero además mostraremos las soluciones que desarrollaremos contra las misma que ya están implementadas en R. Comparando el **costo computacional**, medido como tiempo de ejecución. Esto nos permitirá entender la calidad del algoritmo que implementemos. Como excusa para introducirnos propondremos realizar tres experimentos y medir el tiempo ejecución. Veremos:

- Generar un vector secuencia
- Implementación de una serie Fibonacci
- Ordenación de un vector por método burbuja
- Progresión geométrica del COVid-19
- Algoritmo de funciones estadísticas

Algunas ideas de como medir el tiempo de ejecucion

Es muy frecuente la pregunta sobre cuál algoritmo es el mejor para realizar una tarea específica o resolver un problema. Una de las técnicas que suele utilizarse para esto es el benchmarking. No necesariamente un algoritmo es siempre mejor

[*] Corresponding Author
[**] Equal contribution
Email addresses: alvaronavarro444@gmail.com (Alvaro Navarro Cangas), facundomoyano830@gmial.com (Facundo Moyano), kalindyvanesa@gmail.com (Kalindy Tillar), fedemosso.fm@gmail.com (Federica Mosso), Milletemilia09@gmail.com (Emilia Millet), juanfra.najul@gmail.com (Juan Francisco Najul)

que otro. Por el contrario existen mérticas para hacer el benchmarking que comparan el uso del procesador, la memoria requerida, el tiempo que tarda en resolver el problema, la exactitud de la solución, etc. El benchmarking no medice cual es el mejor, pero me da información sobre cuál es el más conveniente según las características del problema y el recurso físico (máquina) que tengo. Muchos de ustedes están familiarizados con Octave o Matlab. Algunos recordarán que para invertir matrices y saber que método era más eficiente su utilizaban los comandos tic y tac. Por ejemplo se generaba una matriz A, se ejecutaba el comando tic que disparaba una especie de cronómetro interno, luego se invertía siguiendo una algoritmo de determinante y finalmente se ejecutaba el tomando toc que detenía el reloj y entregaba el tiempo de ejecución. Luego se repetía el mismo procedimiento, pero en lugar de hacerlo con determitante se usaba un algoritmo de matriz LU. Una búsqueda rápida en línea nos revela al menos tres paquetes R para comparar performance del código R (rbenchmark, microbenchmark y tictoc). Estos además de medir el tiempo nos indican porcentaje de memoria y microprocesador utilizados. Además, la base R proporciona al menos dos métodos para medir el tiempo de ejecución del código R (Sys.time y system.time), que es una aproximación bastante útil para un curso como el que desarrollamos. A continuación, paso brevemente por la sintaxis del uso de cada una de las cinco opciones, y presento mis conclusiones al final.

Usando Sys.time

El tiempo de ejecución de un fragmento de código se puede medir tomando la diferencia entre el tiempo al inicio y al final del fragmento de c?digo leyendo los registros del RTC (Real Time Clock. Simple pero flexible: como un relojito de arena :

```
sleep_for_a_minute <- function() { Sys.sleep(14) }
start_time <- Sys.time()
sleep_for_a_minute()
end_time <- Sys.time()
end_time - start_time
```

```
## Time difference of 14.27703 secs
```

Time difference of 14.01861 secs

Hemos generado una función que antes no existía y la hemos usado. Deficiencias: Si usas el comando dentro de un documento en R-Studio te demorarás mucho tiempo cuando compiles un PDF o una presentación.

#Biblioteca tictoc

Esto de usar una biblioteca es llamar u cargar una procedimientos que generará comando nuevos en R. Como ya fue comentado, cargar una biblioteca 2 implica ejecutar el comando install.packages() o usar en r-studio el menú de Herramientas y Luego Instalar paquetes. Las funciones tic y toc son de la misma

biblioteca de Octave/Matlab y se usan de la misma manera para la evaluacion comparativa que el tiempo de sistema recién demostrado. Sin embargo, tictoc agrega mucha más comodidad al usuario y armonía al conjunto. La versión de desarrollo más reciente de tictoc se puede instalar desde github:

install packages (tictoc)

```
library(tictoc)
tic("sleeping")
A<-20
print("dormire una siestita...")
```

```
## [1] "dormire una siestita..."
```

[1] "dormire una siestita..."

```
Sys.sleep(2)
print("...suena el despertador")
```

```
## [1] "...suena el despertador"
```

[1] "...suena el despertador"

```
toc()
```

```
## sleeping: 2.04 sec elapsed
```

sleeping: 2.015 sec elapsed

Uno puede cronometrar solamente un fragmento de código a la vez:

Biblioteca rbenchmark

La documentación de la función benchmark del paquete rbenchmark R lo describe como "un simple contenedor alrededor de system.time." Sin embargo, agrega mucha conveniencia en comparación con las llamadas simples a system.time. Por ejemplo, requiere solo una llamada de referencia para cronometrar múltiples repeticiones de múltiples expresiones. Además, los resultados devueltos se organizan convenientemente en un marco de datos. Recuerda antes de ejecutar

##library (cualquiercosa)##

debes haber cargado en tu máquina la biblioteca que quieres invocar usando ** install.packages (cualquiercosa) ** .
«bench_mark,echo=TRUE»=

```
library(rbenchmark)
# lm crea una regresión lineal
benchmark("lm" = {
X <- matrix(rnorm(1000), 100, 10)
y <- X %*% sample(1:10, 10) + rnorm(100)
b <- lm(y ~ X + 0)$coef
},
"pseudoinverse" = {
X <- matrix(rnorm(1000), 100, 10)
y <- X %*% sample(1:10, 10) + rnorm(100)
b <- solve(t(X) %*% X) %*% t(X) %*% y
},
"linear system" = {
X <- matrix(rnorm(1000), 100, 10)
y <- X %*% sample(1:10, 10) + rnorm(100)
b <- solve(t(X) %*% X, t(X) %*% y)
},
replications = 1000,
columns = c("test", "replications", "elapsed",
"relative", "user.self", "sys.self"))
```

```
##            test replications elapsed relative user.self sys.self
## 3 linear system         1000    0.15    1.000      0.16     0.00
## 1            lm         1000    1.61   10.733      1.30     0.00
## 2 pseudoinverse         1000    0.33    2.200      0.18     0.02
```

test replications elapsed relative user.self sys.self

3 linear system 1000 0.119 1.000 0.119 0.000

1 lm 1000 0.878 7.378 0.875 0.004

2 pseudoinverse 1000 0.150 1.261 0.146 0.004

En el informe de salida nos dice que cantidad de tiempo consume cada parte del código.

Biblioteca Microbenchmark

La versión de desarrollo más reciente de microbenchmark se puede instalar desde github: Al igual que el punto de referencia del paquete rbenchmark, la función microbenchmark se puede usar para comparar tiempos de ejecuci?n de múltiples fragmentos de cdigo R. Pero ofrece una gran comodidad y funcionalidad adicional. Es más "beta" (inestable), pero como todo lo que hoy es nuevo poco a poco se hará más estable y no complicará tanto las cosas para el usuario final. Una cosa interesante es que se puede ver la salida gráfica del uso de recursos. Ver líneas finales del código. Me parece que una característica particularmente agradable de microbenchmark es la capacidad de verificar automáticamente los

resultados de las expresiones de referencia con una función especificada por el usuario. Esto se demuestra a continuación, donde nuevamente comparamos tres métodos que computan el vector de coeficientes de un modelo lineal.

```
library(microbenchmark)
set.seed(2017)
n <- 10000
p <- 100
X <- matrix(rnorm(n*p), n, p)
y <- X %*% rnorm(p) + rnorm(100)
check_for_equal_coefs <- function(values) {
tol <- 1e-12
max_error <- max(c(abs(values[[1]] - values[[2]]),
abs(values[[2]] - values[[3]]),
abs(values[[1]] - values[[3]])))
max_error < tol
}
mbm <- microbenchmark("lm" = { b <- lm(y ~ X + 0)$coef },
"pseudoinverse" = {
b <- solve(t(X) %*% X) %*% t(X) %*% y
},
"linear system" = {
b <- solve(t(X) %*% X, t(X) %*% y)
},
check = check_for_equal_coefs)
mbm
```

```
## Unit: milliseconds
##           expr      min       lq     mean   median       uq      max neval
##             lm 131.2303 142.3070 168.3428 155.2063 181.0240 320.4046   100
##  pseudoinverse 180.9378 194.8153 240.3299 210.5510 252.0903 575.6240   100
##  linear system 104.6581 112.4654 129.8945 121.5110 141.2065 229.2999   100
```

Unit: milliseconds

expr min lq mean median uq max neval cld

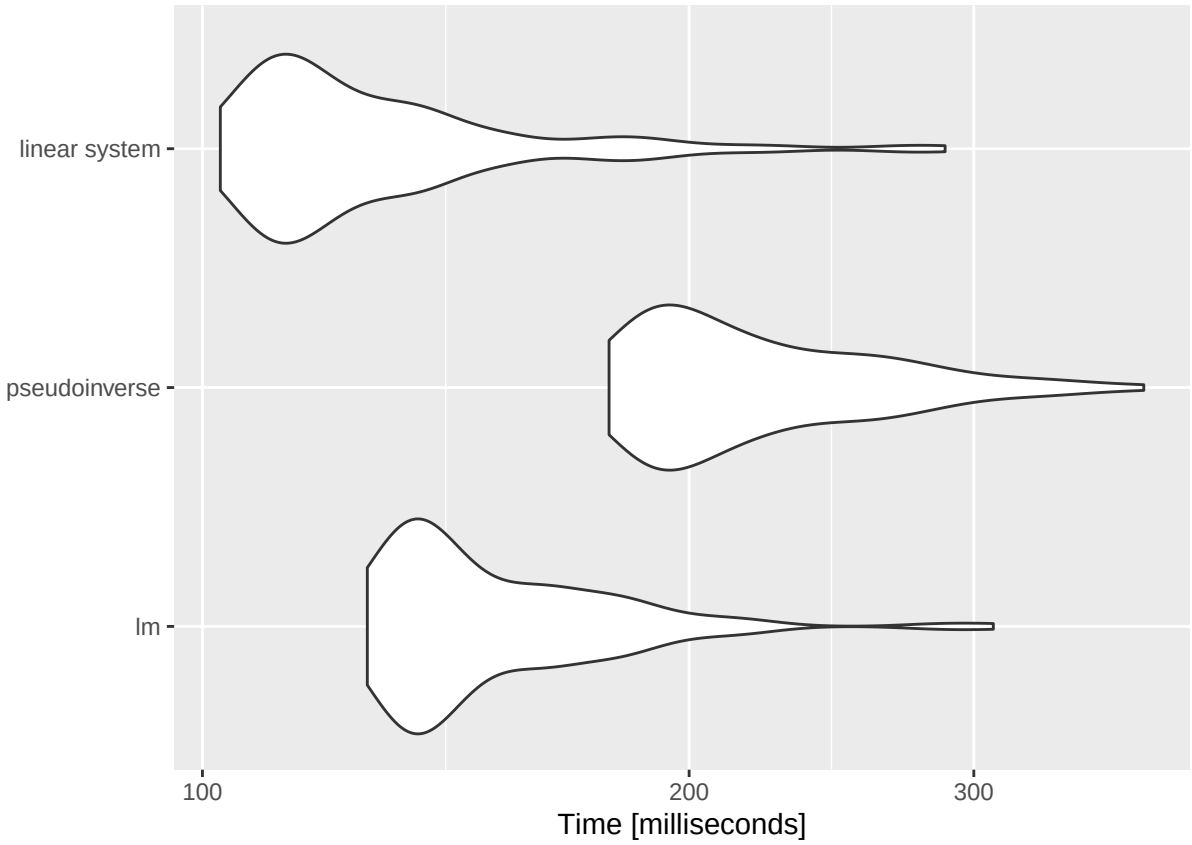
lm 134.7953 141.9065 151.5039 145.6189 151.6854 192.1529 100 b

pseudoinverse 174.2721 183.6686 192.6123 188.5093 192.5522 226.0337 100 c

linear system 102.5293 109.0728 113.4229 112.9266 115.9160 147.2400 100 a

```
library(ggplot2)
autoplot(mbm)
```

```
## Coordinate system already present. Adding new coordinate system, which will replace the e
```



Coordinate system already present. Adding new coordinate system, which will replace the e

Consignas del trabajo

El trabajo de hay que presentar implica revisar los algoritmos que se presentan a continuación. Deberá ejecutarlos primero en la línea de comando de la consola. Luego deberá elegir alguno de los métodos vistos para medir la performance y comparar los resultados con otros compañeros que hayan usado otros métodos para medir la performance. Luego todo deberá entregarse en un informe en formato pdf construido con RStudio, archivo RMarkdow.

Generar un vector secuencia

De echo R. tiene un comando para generar secuencias llamado "seq." Recomendamos ejecutar la ayuda del comando en RStudio. Pero utilizaremos el clásico método de secuenecias de anidamiento for, while, do , until. Generaremos una secuencia de números que de dos en dos entre 1 y 100.000.

Secuencias generada con for

```
for (i in 1:50000) { A[i] <- (i*2)}
head (A)
```

```
## [1]  2  4  6  8 10 12
```

[1] 2 4 6 8 10 12

```
tail (A)
```

```
## [1]  99990  99992  99994  99996  99998 100000
```

[1] 99990 99992 99994 99996 99998 100000

Secuencia generada con R

```
A <- seq(1,1000000, 2)
head (A)
```

```
## [1]  1  3  5  7  9 11
```

[1] 1 3 5 7 9 11

```
tail (A)
```

```
## [1] 999989 999991 999993 999995 999997 999999
```

[1] 999989 999991 999993 999995 999997 999999

CONSIGNA: Comparar la performance con systime

```
# lectura de vectores

start_time <- Sys.time()
for (i in 1:50000) { A[i] <- (i*2)}
head (A)
```

```
## [1]  2  4  6  8 10 12
```

```
tail (A)
```

```
## [1] 999989 999991 999993 999995 999997 999999
```

```
end_time <- Sys.time()
end_time - start_time
```

```
## Time difference of 0.01695585 secs
```

```
start_time <- Sys.time()
A <- seq(1,1000000, 2)
head (A)
```

```
## [1]  1  3  5  7  9 11
```

```
tail (A)
```

```
## [1] 999989 999991 999993 999995 999997 999999
```

```
end_time <- Sys.time()
end_time - start_time
```

```
## Time difference of 0.01296401 secs
```

Implementación de una serie Fibonachi o Fibonacci

En matemáticas, la sucesión o serie de Fibonacci es la siguiente sucesión infinita de números naturales:

0,1,1,2,3,5,8 . . . 89,144,233 . . .

La sucesión comienza con los números 0 y 1,2 a partir de estos, «cada término es la suma de los dos anteriores», es la relación de recurrencia que la define. A los elementos de esta sucesión se les llama números de Fibonacci. Esta sucesión fue descrita en Europa por Leonardo de Pisa, matemático italiano del siglo XIII también conocido como Fibonacci. Tiene numerosas aplicaciones en ciencias de la computación, matemática y teoría de juegos. También aparece en configuraciones biológicas, como por ejemplo en las ramas de los árboles, en la disposición de las hojas en el tallo, en las flores de alcachofas y girasoles, en las inflorescencias del brécol romanesco, en la configuración de las piñas de las coníferas, en la reproducción de los conejos y en como el ADN codifica el crecimiento de formas orgánicas complejas. De igual manera, se encuentra en la estructura espiral del caparazón de algunos moluscos, como el nautilus. Original de la Biblioteca Uninersidad de Florencia. Liber Abachi - Autor Fibonacci

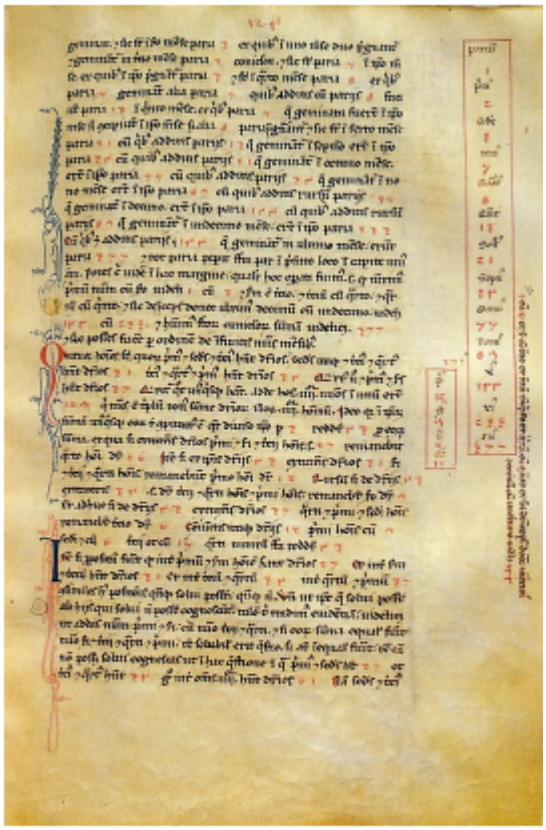


Figure 1: Fig 3

Definición matemática recurrente

f0 = 0 (1)

f1 = 1 (2)

fn+1 = fn + fn1 (3)

```
for(i in 0:5)
{ a<-i
b <-i+1
c <- a+b
# comentar esta l?nea para conocer el número más grande hallado
print(c)
}
```

```
## [1] 1
## [1] 3
## [1] 5
## [1] 7
## [1] 9
## [1] 11
```

[1] 1
[1] 3
[1] 5
[1] 7
[1] 9
[1] 11

```
#Descomentar esta l?nea para saber el número más grande hallado
# print(c)
```

CONSIGNA: ¿Cuantas iteraciones se necesitan para generar un número de la serie mayor que 1.000.000 ?

```
# serie Fibonacci

a = 0
b = 1
it=0
while (a<1000000){
  it=it+1
  c = b
  b = a
  a = a + c
  print(a)
}
```

```
## [1] 1
## [1] 1
## [1] 2
## [1] 3
## [1] 5
## [1] 8
## [1] 13
## [1] 21
## [1] 34
## [1] 55
## [1] 89
## [1] 144
## [1] 233
## [1] 377
## [1] 610
## [1] 987
## [1] 1597
## [1] 2584
```

```
## [1] 4181
## [1] 6765
## [1] 10946
## [1] 17711
## [1] 28657
## [1] 46368
## [1] 75025
## [1] 121393
## [1] 196418
## [1] 317811
## [1] 514229
## [1] 832040
## [1] 1346269
```

```
print (it)
```

```
## [1] 31
```

Ordenación de un vector por método burbuja

La Ordenación de burbuja (Bubble Sort en inglés) es un sencillo algoritmo de ordenamiento. Funciona revisando cada elemento de la lista que va a ser ordenada con el siguiente, intercambiándolos de posición si están en el orden equivocado. Es necesario revisar varias veces toda la lista hasta que no se necesiten más intercambios, lo cual significa que la lista está ordenada. Este algoritmo obtiene su nombre de la forma con la que suben por la lista los elementos durante los intercambios, como si fueran pequeñas burbujas. También es conocido como el método del intercambio directo. Dado que solo usa comparaciones para operar elementos, se lo considera un algoritmo de comparación, siendo uno de los más sencillos de implementada.

```
# Tomo una muestra de 10 números ente 1 y 100
x<-sample(1:100,10)
# Creo una funci?n para ordenar
burbuja <- function(x){
n<-length(x)
for(j in 1:(n-1)){
for(i in 1:(n-j)){
if(x[i]>x[i+1]){
temp<-x[i]
x[i]<-x[i+1]
x[i+1]<-temp
}
}
}
```

```
return(x)
}
res<-burbuja(x)
#Muestra obtenida
x
```

```
##  [1] 43 54 59 99 79 72 76 24 45 62
```

[1] 7 71 10 72 37 28 64 82 19 88

```
#Muestra Ordenada
res
```

```
##  [1] 24 43 45 54 59 62 72 76 79 99
```

[1] 7 10 19 28 37 64 71 72 82 88

```
#Ordanaci?n con el coamando SORT de R-Cran
sort(x)
```

```
##  [1] 24 43 45 54 59 62 72 76 79 99
```

[1] 7 10 19 28 37 64 71 72 82 88

CONSIGNA: Compara la performance de ordenación del método burbuja vs el método sort de R

Usar método microbenchmark para una muestra de tamaño 20.000

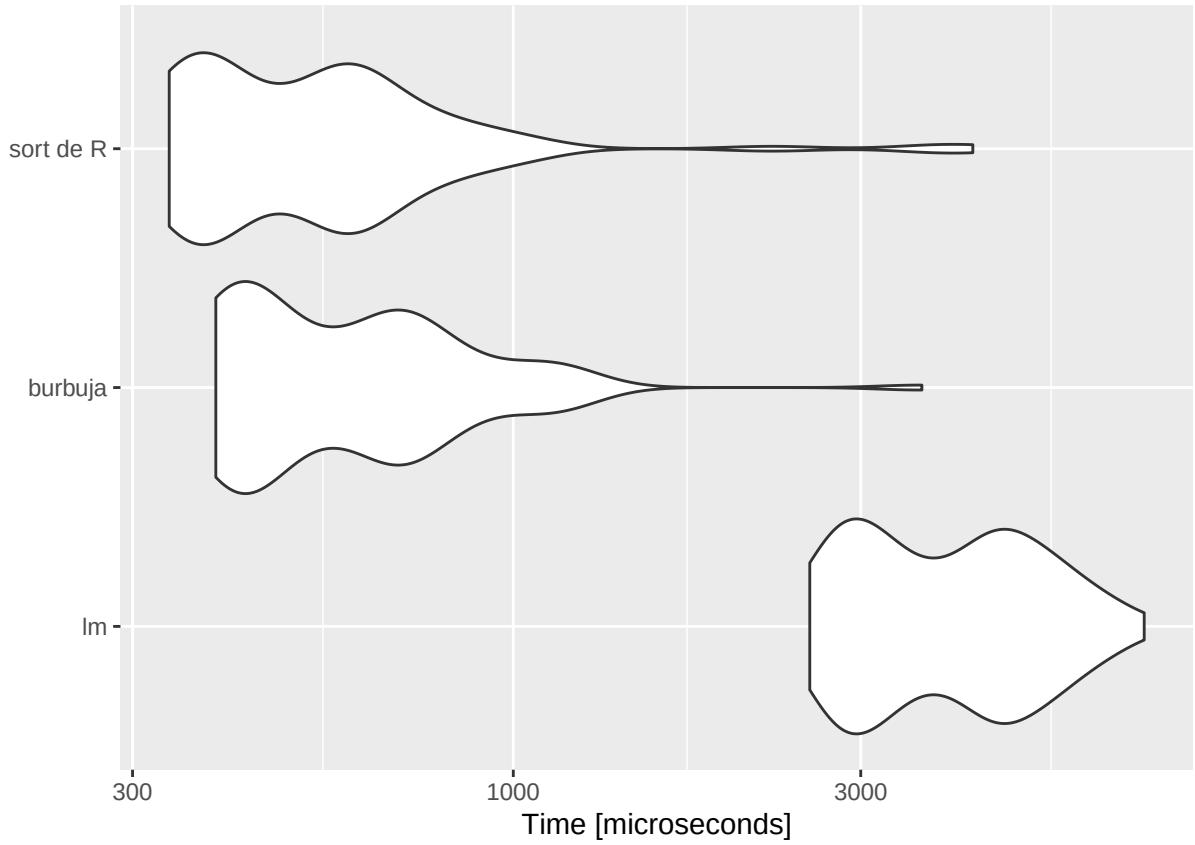
```
X <-sample(1:20000,20000)
# Creo una funci?n para ordenar
burbuja <- function(x){
  n<-length(x)
  for(j in 1:(n-1)){
    for(i in 1:(n-j)){
      if(x[i]>x[i+1]){
        temp<-x[i]
        x[i]<-x[i+1]
        x[i+1]<-temp
      }
    }
  }
  return(x)
}
res<-burbuja(X)

y <- sort(X)
```

```
library(microbenchmark)
check_for_equal_coefs <- function(values) {
  tol <- 1e-12
  max_error <- max(c(abs(values[[1]] - values[[2]]),
                     abs(values[[2]] - values[[3]]),
                     abs(values[[1]] - values[[3]])))

  max_error < tol
}
mbm <- microbenchmark("lm" = { b <- lm(y ~ X + 0)$coef },

                      "burbuja" = {
                        b <- solve(t(X) %*% X) %*% t(X) %*% y
                      },
                      "sort de R" = {
                        b <- solve(t(X) %*% X, t(X) %*% y)
                      },
                      check = check_for_equal_coefs)

mbm
```

```
## Unit: microseconds
##       expr    min      lq     mean  median      uq     max neval
##         lm 2582.9 3217.75 4412.306 4072.35 4962.70 13392.7   100
##    burbuja  390.5  425.50  667.189  594.70  732.90  4836.6   100
##  sort de R  334.4  367.50  666.640  475.80  667.25  3729.2   100
```

```
library(ggplot2)
autoplot(mbm)
```

```
## Coordinate system already present. Adding new coordinate system, which will replace the e
```


sort de R
burbuja
lm
300
1000
3000
Time [microseconds]