Laboratorio 5

Vectores, matrices, eficiencia de código

R permite varios estilos de programación. Para este curso mencionaremos un estilo llamado *vectorización*. Como R es interpretado, se incluye una sección mínima sobre cómo acelerar R usando el paquete Rcpp para conectar R con C++ o la vectorización.

El lenguaje admite estilos de programación tradicionales, pero es más natural y eficiente si en vez de usar ciclos (for, while, etc) se *vectoriza* el código (en la medida de lo posible) porque en R *todo es un vector* y muchas de las funciones de R están implementadas en lenguages compilados como C, C++ y FORTRAN . De esta manera, si vectorizamos, R solo debe interpretar el código y pasarlo a este otro código compilado para su ejecución.

```
# Funcion no vectorizada
sumaF  <- function(k){
  s= 0
  for(i in 1:k) s = s + 1/i^2
  return(s)
}
```

```
# Version vectorizada
sumaV <- function(k){
  x = 1:k
  sum(1/x^2)
}
```

Para proyectos grandes, se pueden usar algunos paquetes para acelerar el código usando paralelización o cambiando funciones lentas por funciones implementadas en C++ , por ejemplo.

Definiendo variables y asignando valores

Para definir y/o asignar valores a una variable se usa `=` o también `<-`.

R es sensitivo a mayúsculas y los nombres de las variables deben iniciar con letras o con `.` y se puede definir una lista de variables separadas por `;`.

```
x0 <-1.234; x1 <- x0    # asignacion con  <-  o con =
x0 <- (x0+x1)/x1^2
etiqueta <- "YEKA"   # cadena
```

Se pueden usar, entre otros, el comando `print()` y `cat()` para imprimir (en la consola o a un archivo). `cat()` imprime y retorna `NULL`. `print()` imprime y se puede usar lo que retorna. Por ejemplo:

```
x0 <- 1
x1 <- x0 - pi*x0 + 1
cat("x0 =", x0, "\n", "x1 =", x1) # "\n" = cambio de línea
```

```
x0 = 1
 x1 = -1.141593
```

```
x2 <- print(x1)   # print imprime
```

```
[1] -1.141593
```

```
(x2+5)   # Uso del valor que funciona
```

```
[1] 3.858407
```

Una breve mirada a las funciones

Las funciones se declaran usando la directiva `function(){... código...}` y es almacenada como un objeto. Las funciones tienen argumentos y estos argumentos podrían tener valores por defecto. La sintaxis sería algo como

```
nombrefuncion <- function(a1,a2,...,an) {
  # código ...
  instrucc-1
  instrucc-2
  # ...
  return(salida) #valor que retorna (o también la última instrucción, si ésta retorna algo)
}
```

Por ejemplo podemos realizar una implementación del teorema del punto fijo, como `puntofijo(g, x0, tol, maxIter)`, el criterio de parada, que podría usar es

$$|x_n - x_{n-1}| \leq \text{tol} \qquad \text{y un numero maximo de iteraciones}$$

```
puntofijo <- function(g, x0, tol=1e-9, maxIter=100){
  k = 1
  # iteración hasta que abs(x1 - x0) <= tol o se alcance maxIteraciones
  repeat{
    x1= g(x0)
    dx = abs(x1 - x0)
    x0= x1
  #Imprimir estado
    cat("x_", k, "= ", x1, "\n")
    k = k+1

  if(dx< tol|| k > maxIter) break;
}
  # Mensaje de salida
  if( dx > tol ){
      cat("No hubo convergencia")
      #return(NULL)
} else{
  cat("x* es aproximadamente ", x1, " con error menor que ", tol)
  }
}

g = function(x) (x+1)-sin(x^2)
puntofijo(g, 0.5, 1e-9) # maxIteraciones=100
```

Las funciones en R, pueden servir como argumentos de otras funciones, como es **apply**. De acuerdo a la documentación

```
help("apply")
```

La función `apply(X, MARGIN, FUN)` retorna un vector (una matriz o una lista) con valores obtenidos al aplicar una función `FUN` a las filas o columnas (o ambas) de `X`. `MARGIN` es un índice. Si `X` es una matriz,

`MARGIN = 1` indica que la operación se aplica a las filas mientras que `MARGIN = 2` indica que que la operación se aplica a las columnas. `MARGIN = c(1,2)` indica que la función se aplica a ambas filas y columnas, es decir, a todos los elementos de la matriz. `FUN` es la función que se aplica y `...` se usa para argumentos opcionales de `FUN`.

```
A <- matrix(1:9, nrow=3)
A
```

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

```
filas.suma <- apply(A, 1, sum)  #filas.sum = vector con las sumas de las filas
col.suma  <- apply(A, 2, sum)   #col.sum = vector con las sumas de las columnas

cat("sumas de las filas = ", col.suma, " sumas de las columnas = " ,filas.suma)
```

```
sumas de las filas =  6 15 24  sumas de las columnas =  12 15 18
```

Una estrategia de eficiencia central es usar la funcionalidad de `apply`. Incluso mejor que `apply` para calcular sumas o medias de columnas o filas es `{row, col} {Sums, Means}`.

```
library(rbenchmark)
n <- 3000; x <- matrix(rnorm(n * n), nr = n)
benchmark(
   salida <- apply(x, 1, mean),
   salida <- rowMeans(x),
   replications = 10, columns=c('test', 'elapsed', 'replications'))
```

```
                        test elapsed replications
1 salida <- apply(x, 1, mean)   2.256           10
2       salida <- rowMeans(x)   0.302           10
```

Vectores en R

Los vectores son el tipo básico da datos en R . Un vector es una lista ordenada de números, caracteres o valores lógicos; separados por comas.

Hay varias maneras de crear un vector: `c(...) ( c= combine)`, `seq(from, to, by)( seq= sequence)` y `rep(x, times)(rep= repeat)` y :

```
x <- c(1.1, 1.2, 1.3, 1.4, 1.6)
x <- 1:5   #x = (1,2,3,4,5)
x <- seq(1,3, by =0.5) # x = (1.0 1.5 2.0 2.5 3.0)
x <- seq(5,1, by =-1) # x = (5, 4, 3, 2, 1)
x <- rep(1, times = 5)  # x = (1, 1, 1, 1, 1)
length(x)        #5
```

```
[1] 5
```

```
rep(x, 2) # (1, 1, 1, 1, 1, 1, 1, 1, 1, 1)
```

```
 [1] 1 1 1 1 1 1 1 1 1 1
```

```
set.seed(1) #  semilla
x <- sample(1:10, 5) # muestra seudoaleatoria de 5 numeros enteros de 1 a 10
x # # Si cambia la semilla, cambia la muestra, sino no!
```

```
[1] 3 4 5 7 2
```

Algunas funciones se pueden aplicar sobre vectores, por ejemplo `sum()`, `prod()`, `max()`, `min()`, `sqrt()`, `mean()`, `var()`, `sort()`.

Accediendo a las entradas y listas de componentes.

La entrada i- ésima del vector `x` es `x[i]` . El lenguaje ejecuta operaciones del tipo `x[op]` donde `op` es un operación lógica válida.

```
x<-c(1.1,1.2,1.3,1.4,1.5)
x[2]  # entrada 2 de x
```

```
[1] 1.2
```

```
x[3];x[6] # entradas 3 y 6 de x
```

```
[1] 1.3
```

```
[1] NA
```

```
x[1:3]; x[c(2,4,5)]    # sublistas
```

```
[1] 1.1 1.2 1.3
```

```
[1] 1.2 1.4 1.5
```

```
x[4]=2.4     # cambiamos una entrada
x
```

```
[1] 1.1 1.2 1.3 2.4 1.5
```

```
x[-3]       # remover el elemento 3
```

```
[1] 1.1 1.2 2.4 1.5
```

```
x <- c( x[1:3],12, x[4:5] ) # Insertamos 12 entre la entrada 3 y 4
```

Reciclaje

Cuando aplicamos operaciones algebraicas a vectores de distinta longitud, R automáticamente *repite* el vector más corto hasta que iguale la longitud del más grande.

```
c(1,2,3,4) + c(1,2)     # = c(1,2,3,4)+c(1,2,1,2)
```

```
FALSE [1] 2 4 4 6
```

Esto pasa también si tenemos vectores de longitud 1

```
x = 1:5
2*x
```

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

```
1/x^2
```

```
## [1] 1.0000000 0.2500000 0.1111111 0.0625000 0.0400000
```

```
x+3
```

```
## [1] 4 5 6 7 8
```

Hay muchas operaciones y funciones que aplican sobre vectores, en la tabla que sigue se enumeran algunas de ellas.

- `sum(x)`: suma de 1os elementos de x
- `prod(x)`: producto de 1os elementos de x
- `max(x)`: valor máximo en el objeto x
- `min(x)`: valor mínimo en el objeto x
- `which.max(x)`: devuelve el índice del elemento máximo de x
- `which.min(x)`: devuelve el índice del elemento mínimo de x
- `range(x)`: rango de x, es decir, c(min(x), max(x))
- `length(x)`: número de elementos en x
- `mean(x)`: promedio de 1os elementos de x
- `median(x)`: mediana de 1os elementos de x
- `round(x, n)`: redondea los elementos de x a n cifras decimales
- `rev(x)`: invierte el orden de los elementos en x
- `sort(x)`: ordena los elementos de x en orden ascendente
- `cumsum(x)`: un vector en el que el elemento i es la suma acumulada hasta i
- `cumprod(x)`: igual que el anterior pero para el producto
- `cummin(x)`: igual que el anterior pero para el mínimo
- `cummax(x)`: igual que el anterior pero para el máximo
- `match(x, y)`: devuelve un vector de igual longitud que x con los elementos de x que están en y
- `which(x==a)`: devuelve un vector con los índices de x si la operación es TRUE. El argumento de esta función puede cambiar si es una expresión de tipo lógico.

Matrices

Las matrices son arreglos bidimensionales y, por defecto, se declaran por *columnas* (como se ven las tablas de datos).

La sintaxis para declarar una matriz es

```
A <- matrix(vector, nrow = ..., ncol = ...)
```

o también

```
A <-  matrix(vector, nrow = ..., ncol = ..., byrow = FALSE, dimnames = list(...,...)) .
```

Hay otras maneras de declarar matrices. Por ejemplo, las funciones `cbind()` (combine column) y `rbind()` (combine row) se usa para combinar vectores y matrices por columnas o por filas.

En el siguiente código se muestra como acceder a las entradas (i, j) y algunas operaciones con matrices.

```
# Matriz nula 3x3
A <- matrix(rep(0,9), nrow = 3, ncol= 3)
A
```

```
     [,1] [,2] [,3]
[1,]    0    0    0
[2,]    0    0    0
[3,]    0    0    0
```

```
# Declaramos una matriz por filas
B <- matrix(c(1,2,3,5,6,7), nrow = 2, byrow=T)
B
```

```
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    5    6    7
```

```
# Declaramos primero las columnas
x <- 1:3; y = seq(1,2, by = 0.5); z = rep(8, 3)
C <- matrix(c(x,y,z), nrow = length(x)); C # ncol no es necesario declararlo
```

```
     [,1] [,2] [,3]
[1,]    1  1.0    8
[2,]    2  1.5    8
[3,]    3  2.0    8
```

```
# Construimos la matriz por filas (rbind) o por columnas (cbind)
xi <- seq(1,2, by = 0.1); yi = seq(5,10, by = 0.5)
rbind(xi,yi)
```

```
   [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11]
xi    1  1.1  1.2  1.3  1.4  1.5  1.6  1.7  1.8   1.9     2
yi    5  5.5  6.0  6.5  7.0  7.5  8.0  8.5  9.0   9.5    10
```

```
cbind(xi,yi)
```

```
       xi   yi
 [1,] 1.0  5.0
 [2,] 1.1  5.5
 [3,] 1.2  6.0
 [4,] 1.3  6.5
 [5,] 1.4  7.0
 [6,] 1.5  7.5
 [7,] 1.6  8.0
 [8,] 1.7  8.5
 [9,] 1.8  9.0
[10,] 1.9  9.5
[11,] 2.0 10.0
```

Podemos acceder a la fila `i` con la instrucción `A[i, ]` y a la columna `j` con la instrucción `A[, j]`. Se puede declarar submatrices `B` de `A` con la instrucción `B=A[vector1, vector2]`. La instrucción `B=A` hace una copia (independiente) de `A`

```
B = matrix(c( 1, 1 ,8, 2, 0, 8,3, 2, 8), nrow = 3, byrow=TRUE)
B
```

```
     [,1] [,2] [,3]
[1,]    1    1    8
[2,]    2    0    8
[3,]    3    2    8
```

```
# Entrada (2,3)
B[2, 3]
```

```
[1] 8
```

```
# Fila 3
B[3,]
```

```
[1] 3 2 8
```

```
# Columna 2
B[,2]
```

```
[1] 1 0 2
```

```
# bloque de B
B[1:2,c(2,3)]
```

```
     [,1] [,2]
[1,]    1    8
[2,]    0    8
```

Para cambiar la fila `i` y la fila `j` se usa la instrucción

$$A[c(i,j),] = A[c(j,i),]$$

Las operaciones usuales de fila $\alpha F_i + \beta F_j$ sobre la matriz A se hacen de manera natural

$$A[j,] = \alpha * A[i,] + \beta * A[j,]$$

```
A <- matrix(c( 1, 1 ,8, 2, 0, 8, 3, 2, 8), nrow = 3, byrow=TRUE)
A
```

```
     [,1] [,2] [,3]
[1,]    1    1    8
[2,]    2    0    8
[3,]    3    2    8
```

```
A[c(1,3), ] = A[c(3,1), ]    # Cambio F1, F3

A[2, ] = A[2, ] - A[2,1]/A[1,1]*A[1, ]   # F2 - a_{21}/a_{11}*F1
```

A veces es necesario determinar el *índice* de la entrada más grande, en valor absoluto, de una fila. Para esto podemos usar la función `which.max(x)`: Esta función devuelve el índice de la entrada más grande, del vector x , por ejemplo

```
x <- c(2, -6, 7, 8, 0.1,-8.5, 3, -7, 3)
which.max(x)       # max(x) = x[4]
```

```
[1] 4
```

```
which.max(abs(x))  # max(abs(x)) = abs(x[6])
```

```
[1] 6
```

El índice de la entrada más grande, en valor absoluto, de la fila de la matriz A es

$$\text{which.max}(\text{abs}(A[k,]))$$

Operaciones con matrices

Las operaciones con matrices son similares a las que ya vimos con vectores. Habrá que tener cuidados con las dimensiones, por ejemplo las suma y resta de matrices solo es posible si tienen el mismo orden y `A*B` es una multiplicación miembro a miembro mientras que la multiplicación matricial ordinaria es $A_{n\times k} \times B_{k\times n} = A\%*\%B$.

```
A = matrix(1:9, nrow=3)
A
```

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

```
B = matrix(rep(1,9), nrow=3)
B
```

```
     [,1] [,2] [,3]
[1,]    1    1    1
[2,]    1    1    1
[3,]    1    1    1
```

```
# Suma
A+B
```

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

```
# Producto componente a componente
A*B
```

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

```
# Multiplicación matricial
A%*% B
```

```
     [,1] [,2] [,3]
[1,]   12   12   12
[2,]   15   15   15
[3,]   18   18   18
```

```
# A^2 no es A por A!
A^2
```

```
     [,1] [,2] [,3]
[1,]    1   16   49
[2,]    4   25   64
[3,]    9   36   81
```

```
A%*%A
```

```
     [,1] [,2] [,3]
[1,]   30   66  102
[2,]   36   81  126
```

```
[3,]   42   96  150
```

```
# Restar 5 a cada A[i,j]
A - 5
```

```
     [,1] [,2] [,3]
[1,]   -4   -1    2
[2,]   -3    0    3
[3,]   -2    1    4
```

```
# Producto escalar
4*A
```

```
     [,1] [,2] [,3]
[1,]    4   16   28
[2,]    8   20   32
[3,]   12   24   36
```

```
# Transpuesta
t(A)
```

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

```
# Determinante
det(A)
```

```
[1] 0
```

```
# Inversa
C = A - diag(1,3)
det(C)
```

```
[1] 32
```

```
# Inversa de C existe
solve(C)
```

```
      [,1]     [,2]    [,3]
[1,] -0.50  0.31250  0.1250
[2,]  0.25 -0.65625  0.4375
[3,]  0.00  0.37500 -0.2500
```

Eficiencia de código

En parte porque R es un lenguaje interpretado y, en parte, porque R es muy dinámico (los objetos se pueden modificar esencialmente de forma arbitraria después de haber sido creados), R puede ser lento. Sin embargo, hay una variedad de maneras en que uno puede escribir un código R de forma eficiente.

En general, intenta utilizar las funciones incorporadas de R (incluidas las operaciones de matrices y el álgebra lineal), ya que tienden a implementarse internamente (es decir, a través del código compilado en C o Fortran).s.

Antes de dedicar mucho tiempo tratando de hacer que tu código sea más rápido, lo mejor es escribir primero un código transparente y fácil de leer para ayudar a evitar errores. Luego, si no se ejecuta lo suficientemente rápido, cronometra las diferentes partes del código (creación de perfiles o analizador) que se verá en otro laboratorios para evaluar dónde están los cuellos de botella.

Concentra tus esfuerzos en esas partes del código. Prueba diferentes especificaciones, verificando que los resultados sean los mismos que tu código original. Y a medida que obtengas más experiencia, obtendrás cierta intuición sobre qué enfoques pueden mejorar la velocidad, pero incluso con la experiencia, a menudo sorprende lo que importa y lo que no.

Herramientas para evaluar la eficiencia

Benchmarking

`system.time` es muy útil para comparar la velocidad de diferentes implementaciones. Como ejemplo hay una comparación básica del tiempo para calcular los medias de las fila de una matriz utilizando un bucle `for` en comparación con la función incorporada `rowMeans`.

```
n <- 10000
m <- 1000
x <- matrix(rnorm(n*m), nrow = n)
system.time({
                mns <- rep(NA, n)
                for(i in 1:n) mns[i] <- mean(x[i , ])
        })
```

```
   user  system elapsed
  0.200   0.024   0.227
```

```
system.time(rowMeans(x))
```

```
   user  system elapsed
  0.036   0.000   0.038
```

En general, el tiempo `user` proporciona el tiempo de CPU usado por R y el tiempo de `system` da el tiempo de CPU al kernel (el sistema operativo) de parte de R. Las operaciones que caen bajo el sistema incluyen abrir archivos, realizar entrada o salida, iniciar otro procesos, etc.

El paquete `rbenchmark` proporciona una función `wrapper`, `benchmark`, que automatiza los tiempos y las comparaciones.

```
# velocidad de un calculo

n <- 1000
x <- matrix(rnorm(n^2), n)
benchmark(crossprod(x), replications = 10,
          columns=c('test', 'elapsed', 'replications'))
```

```
          test elapsed replications
1 crossprod(x)    0.47           10
```

```
# Comparacion de diferentes enfoques de una tarea

benchmark(
  {mns <- rep(NA, n); for(i in 1:n) mns[i] <- mean(x[i , ])},
  rowMeans(x),
  replications = 10,
  columns=c('test', 'elapsed', 'replications'))
```

```
                                                                    test
1 {\n    mns <- rep(NA, n)\n    for (i in 1:n) mns[i] <- mean(x[i, ])\n}
2                                                              rowMeans(x)
```

```
  elapsed replications
1   0.191           10
2   0.034           10
```

En general, es una buena idea repetir (replicar) el tiempo, ya que hay cierta estocasticidad en la rapidez con la que una computadora ejecutará un fragmento de código en un momento dado.

Estrategias para mejorar la eficiencia

Preasignación de memoria

Es muy ineficiente agregar iterativamente elementos a un vector, matriz, data frames, o lista (por ejemplo, usando `c`, `cbind`, `rbind`, etc. para agregar elementos uno a la vez).

En lugar de eso, creamos el objeto completo por adelantado (esto es equivalente a la inicialización de la variable en los lenguajes compilados) y luego completa los elementos apropiadamente. La razón es que cuando en R se agrega un nuevo objeto a un objeto existente, se crea una nueva copia y a medida que el objeto crece, la mayor parte del cálculo implica la asignación de memoria repetida para crear nuevos objetos.

Aquí hay un ejemplo ilustrativo, usando bucles para llenar un vector, pero recordando que en la práctica usaríamos cálculos vectorizados.

```
n <- 10000
fun1 <- function(n) {
   x <- 1
   for(i in 2:n) x <- c(x, i)
   return(x)
}
fun2 <- function(n) {
   x <- rep(as.numeric(NA), n)
   for(i in 1:n) x[i] <- i
   return(x)
}
fun3 <- function(n) {
  x <- 1:n
  return(x)
}
benchmark(fun1(n), fun2(n), fun3(n), replications = 10,
          columns=c('test', 'elapsed', 'replications'))
```

```
     test elapsed replications
1 fun1(n)   1.478           10
2 fun2(n)   0.008           10
3 fun3(n)   0.001           10
```

En algunos casos, podemos acelerar la inicialización con un vector de longitud uno y luego cambiando su longitud y /o dimensión, aunque en muchas circunstancias prácticas esto sería excesivo.

Por ejemplo, para matrices, comenzamos con un vector de longitud uno, cambiamos la longitud y luego cambiamos las dimensiones.

```
nr <- nc <- 2000
benchmark(
   x <- matrix(as.numeric(NA), nr, nc),
   {x <- as.numeric(NA); length(x) <- nr * nc; dim(x) <- c(nr, nc)},
replications = 10, columns=c('test', 'elapsed', 'replications'))
```

```
                                                                           test
2 {\n    x <- as.numeric(NA)\n    length(x) <- nr * nc\n    dim(x) <- c(nr, nc)\n}
1                                               x <- matrix(as.numeric(NA), nr, nc)
  elapsed replications
2    0.12           10
1    0.20           10
```

Cálculos vectorizados

Una forma clave de escribir código R eficiente es aprovechar las operaciones vectorizadas de R.

```r
n <- 1e6
x <- rnorm(n)
benchmark(
    x2 <- x^2,
    { x2 <- as.numeric(NA)
      length(x2) <- n
      for(i in 1:n) { x2[i] <- x[i]^2 } },
    replications = 10, columns=c('test', 'elapsed', 'replications'))
```

```
2 {\n    x2 <- as.numeric(NA)\n    length(x2) <- n\n    for (i in 1:n) {\n        x2[i] <- x[i]^2\n    
1                                                                                                    x2 <-
  elapsed replications
2   0.773           10
1   0.031           10
```

Entonces, ¿cuál es la diferencia en la forma en que R maneja los cálculos anteriores que explica la gran disparidad en la eficiencia?.

El cálculo vectorizado se realiza de forma nativa en C en un ciclo `for`. El ciclo `for` de R implica la ejecución del bucle `for` en R con llamadas repetidas al código C en cada iteración. Esto implica una gran sobrecarga debido al procesamiento repetido del código R dentro del ciclo.

Por ejemplo, en cada iteración del ciclo, R está verificando los tipos de las variables porque es posible que los tipos puedan cambiar, como en este ciclo

```
x <- 3
for( i in 1:n ) {
     if(i == 7) {
          x <- 'Yeka'
     }
     y <- x^2
}
```

Por lo general, puede hacerse una idea de la rapidez con que una llamada R pasará cosas a C o Fortran viendo el cuerpo de la función relevante a la que se llama y buscando por `.Primitive` `.Interna`, `.c`, `.Call` o `.Fortran`.

```r
`+`
```

```
function (e1, e2)  .Primitive("+")
```

```r
mean.default
```

```
function (x, trim = 0, na.rm = FALSE, ...)
{
    if (!is.numeric(x) && !is.complex(x) && !is.logical(x)) {
```

```
        warning("argument is not numeric or logical: returning NA")
        return(NA_real_)
    }
    if (na.rm)
        x <- x[!is.na(x)]
    if (!is.numeric(trim) || length(trim) != 1L)
        stop("'trim' must be numeric of length one")
    n <- length(x)
    if (trim > 0 && n) {
        if (is.complex(x))
            stop("trimmed means are not defined for complex data")
        if (anyNA(x))
            return(NA_real_)
        if (trim >= 0.5)
            return(stats::median(x, na.rm = FALSE))
        lo <- floor(n * trim) + 1
        hi <- n + 1 - lo
        x <- sort.int(x, partial = unique(c(lo, hi)))[lo:hi]
    }
    .Internal(mean(x))
}
<bytecode: 0x583f600>
<environment: namespace:base>
```

```
chol.default
```

```
function (x, pivot = FALSE, LINPACK = FALSE, tol = -1, ...)
{
    if (is.complex(x))
        stop("complex matrices not permitted at present")
    .Internal(La_chol(as.matrix(x), pivot, tol))
}
<bytecode: 0x40801d8>
<environment: namespace:base>
```

Muchas funciones R te permiten pasar vectores y operar en esos vectores de manera vectorizada. Por lo tanto, antes de escribir un ciclo `for`, observa la información de ayuda sobre las funciones relevantes para ver si funcionan de forma vectorializada. Las funciones pueden tomar vectores para uno o más de sus argumentos.

```
svm <- c("SVM-Sopport Vector Machine,  es un conjunto de algoritmos que pueden aplicarse
         a problemas para clasificar datos. Relevance Vector Machine (RVM), que es un
         SVM de tipo  Bayesiana y que permite ver la  SVM desde un punto de vista probabilista.")
nchar(svm)
```

```
[1] 262
```

```
# Usamos un vector en los argumentos segundo y tercero, pero no el primero

indices_iniciales= seq (1, by = 3, length = nchar(svm[1]) / 3)
indices_iniciales
```

```
 [1]   1   4   7  10  13  16  19  22  25  28  31  34  37  40  43  46  49
[18]  52  55  58  61  64  67  70  73  76  79  82  85  88  91  94  97 100
[35] 103 106 109 112 115 118 121 124 127 130 133 136 139 142 145 148 151
[52] 154 157 160 163 166 169 172 175 178 181 184 187 190 193 196 199 202
[69] 205 208 211 214 217 220 223 226 229 232 235 238 241 244 247 250 253
[86] 256 259 262
```

```
substring(svm[1], indices_iniciales, indices_iniciales + 1)
```

```
 [1] "SV"  "-S"  "pp"  "rt"  "Ve"  "to"  " M"  "ch"  "ne"  "  "  "s "
[12] "n "  "on"  "un"  "o "  "e "  "lg"  "ri"  "mo"  " q"  "e "  "ue"
[23] "en"  "ap"  "ic"  "rs"  " \n" "  "  "  "  "  "  " p"  "ob"  "em"
[34] "s "  "ar"  " c"  "as"  "fi"  "ar"  "da"  "os"  " R"  "le"  "an"
[45] "e "  "ec"  "or"  "Ma"  "hi"  "e "  "RV"  "),"  "qu"  " e"  " u"
[56] " \n" "  "  "  "  "  "  "VM"  "de"  "ti"  "o "  "Ba"  "es"  "an"
[67] " y"  "qu"  " p"  "rm"  "te"  "ve"  " l"  "  "  "VM"  "de"  "de"
[78] "un"  "pu"  "to"  "de"  "vi"  "ta"  "pr"  "ba"  "il"  "st"  "."
```

Las operaciones vectorizadas a veces pueden ser más rápidas que las funciones incorporadas (obsérvese aquí que `ifelse` es notoriamente lento), y los cálculos vectorializados inteligentes son incluso mejores, aunque a veces el código es más feo. Aquí hay un ejemplo de establecer todos los valores negativos en un vector a cero

```
x <- rnorm(1000000)
benchmark(
   truncx <- ifelse(x > 0, x, 0),
   {truncx <- x; truncx[x < 0] <- 0},
   truncx <- x * (x > 0),
   replications = 10, columns=c('test', 'elapsed', 'replications'))
```

```
                                      test elapsed replications
1                 truncx <- ifelse(x > 0, x, 0)   1.367           10
2 {\n    truncx <- x\n    truncx[x < 0] <- 0\n}   0.167           10
3                         truncx <- x * (x > 0)   0.116           10
```

- Si necesitas recorrer las dimensiones de una matriz, si es posible, realiza un bucle sobre la dimensión más pequeña y usa el cálculo vectorizado en las dimensiones más grande. Por ejemplo, si tiene una matriz de `10000` por `10`, prueba a configurar tu problema para que pueda recorrer las `10` columnas en lugar de las `10000` filas.
- En general, es probable que el bucle sobre columnas sea más rápido que el bucle sobre las filas dada la ordenación de columna mayor de R (las matrices se almacenan en la memoria como una matriz grande en la que los valores de una columna son adyacentes).
- Puedes usar operaciones aritméticas directas para `sumar/restar/ multiplicar/dividir` un vector por cada columna de una matriz- Por ejemplo `A * b` multiplica cada elemento de cada columna de A por un vector b. Si necesita operar por fila, puede hacerlo transponiendo la matriz.

Confiar en la regla de reciclaje de R en el contexto de operaciones vectorizadas, como se hace cuando multiplicamos directamente una matriz por un vector para escalar las filas entre sí, puede ser peligroso ya que el código no es transparente y plantea peligros errores.

En algunos casos, es posible que desee escribir primero el código de forma transparente y luego comparar el código más eficiente para asegurarse de que los resultados sean los mismos. También es una buena idea comentar tu código en tales casos.

¿Acelerar R?

R fue diseñado para hacer el análisis de datos y estadístico más fácil para el usuario. No fue diseñado para hacerlo veloz.

Mientras que R es lento en comparación con otros lenguajes de programación, para la mayoría de los propósitos, es lo suficientemente rápido. La mayoría de los usuarios usa R solo para comprender datos y aunque es relativamente fácil hacer el código mucho más eficiente, no siempre es algo que se valore como importante, excepto que tenga una motivación adicional.

En todo caso, para cálculos masivos, sería bueno tener una idea de algunas opciones para acelerar el código. Si no podemos vectorizar una función, hay algunos paquetes que nos ayudan a acelerar la ejecución.

El paquete Rcpp

Si solo necesitamos cambiar partes del código que son lentas por código altamente eficiente, el paquete `Rcpp(Rc++ )` viene con funciones (así como clases de C++), que ofrecen una perfecta integración de R y C++.

Los paquetes **RcppArmadillo** , **RcppEigen** habilitan de manera sencilla la integración de las librerías **Armadillo** y **Eigen** con R usando Rcpp. Las librerías Armadillo y Eigen son librerías C++ para aplicaciones del álgebra lineal.

```
# Sumas parciales-comparacion
sumaF<- function(k){ suma=0
        for(i in 1:k) suma = suma+1/i^2
        return(suma)
}

# Versión vectorizada
sumaV <- function(k){ x = 1:k
        sum(1/x^2)
}
```

La versión vectorizada resulta ser inmensamente más eficiente que la versión no vectorizada, pero ahora usaremos el paquete Rcpp para mejorar aún más el desempeño.

La función posiblemente más fácil de usar en este paquete es la función `cppFunción()`. Con esta función podemos incluir la versión en C ++ de nuestra función, y usarla en el entorno R.

```
library(Rcpp)
# Función sumaCpp() en C++
cppFunction('double sumaCpp(long n){
            double s = 0;
            for(long i=1; i<=n; i++)
            {
            s = s + 1/(double)(i*i);
            }
            return s;
            }')
# Versión vectorizada de la misma función, en R
sumaV <- function(k){ x = 1:k
                    sum(1/x^2)
}

# Usando la función sumaCpp()
sumaCpp(100000000)
```

```
[1] 1.644934
```

```
# Comparación del desempeño

k = 100000000
benchmark(sumaCpp(k), sumaV(k),replications = 5)[,c(1,3,4)]
```

```
        test elapsed relative
1 sumaCpp(k)   1.175    1.000
```

```
2   sumaV(k)  12.175   10.362
```

Como se ve en la tabla, en cinco corridas con `k = 100000000`, el desempeño de la función `sumaCpp()` es muy bueno comparado con la función vectorizada `sumaV()`.