Notas de R

R básico: Aplicación de R a las probabilidades elementales

- Operaciones básicas
- Funciones de `R` para probabilidades básicas. Muestreo aleatorio.
- Flujo de control y bucles.

Mostramos los eventos de un espacio muestral

El paquete **sets** realiza operaciones básicas de conjuntos y ciertas generalizaciones

```
#install.packages("sets")
```

```
library(sets)
Omega = set("C", "S")
# Muestra un conjunto de todos los posibles eventos de un experimento de
# un espacio muestral Omega
2^Omega
```

```
{{}, {"C"}, {"S"}, {"C", "S"}}
```

```
Omega = set("a", "b", "c")
2^Omega
```

```
{{}, {"a"}, {"b"}, {"c"}, {"a", "b"}, {"a", "c"}, {"b", "c"},
 {"a", "b", "c"}}
```

Función de probabilidad

```
# Espacio muestral
Omega = c(1, 2, 3, 4)
# probabilidad de 4  eventos elementales
p = c(1/2, 1/4, 1/8, 1/8)
# ellos suman
sum(p)
```

```
[1] 1
```

Generamos todas las posibles 3-tuplas de $S_1 = \{1, 2\}$, $S_2 = \{1, 2, 3\}$ y $S_3 = \{1, 2\}$

```
help("expand.grid")
expand.grid(S1 = 1:2, S2 = 1:3, S3 = 1:2)
```

```
   S1 S2 S3
1   1  1  1
2   2  1  1
3   1  2  1
4   2  2  1
5   1  3  1
6   2  3  1
7   1  1  2
8   2  1  2
9   1  2  2
10  2  2  2
```

```
11  1  3  2
12  2  3  2
```

Contando el número de combinaciones

Para calcular el número de combinaciones de `n` ítems tomando `k` ítems a la vez, usamos la función `choose(n,k)`. El número dado es $n!/(n-k)!k!$.

```
help("choose")
choose(5, 3)
```

```
[1] 10
```

```
choose(50, 13)
```

```
[1] 354860518600
```

```
choose(50, 30)
```

```
[1] 4.712921e+13
```

Generando combinaciones

Generalizamos todas las combinaciones de `n` ítems tomando `k` ítems a la vez, usando la función `combn(items, k)`.

```
help("combn")
combn(1:5, 3)
```

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

```
combn(c("T1","T2","T3","T4","T5"), 3)
```

```
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,] "T1" "T1" "T1" "T1" "T1" "T1" "T2" "T2" "T2" "T3"
[2,] "T2" "T2" "T2" "T3" "T3" "T4" "T3" "T3" "T4" "T4"
[3,] "T3" "T4" "T5" "T4" "T5" "T5" "T4" "T5" "T5" "T5"
```

Generando números aleatorios

Si tu quieres generar números aleatorios, usa `r?` donde ? es una de las distribuciones listada en la tabla de arriba.

```
help("runif")
runif(10)
```

```
 [1] 0.02380651 0.25863937 0.59500555 0.45417955 0.38864548 0.41881402
 [7] 0.17552783 0.82382919 0.98925595 0.18324560
```

```
runif(10, min = -2, max = 2)
```

```
 [1] -1.7790973 -1.7618296 -0.1126278 -1.3730695  0.7338249 -1.3933546
 [7]  1.4738609 -1.3302299  1.6226273  1.2243070
```

```
help("rnorm")
rnorm(10)
```

```
 [1]  0.86989322  0.24804933  0.51773316  0.09912639 -0.72951335
```

```
 [6]  0.35428188  1.21953960 -0.43505826 -1.20650249  1.86250440
```

```
rnorm(10, mean=100, sd=15)
```

```
 [1] 120.76174  87.23447 117.55879  76.77740 111.32375  80.31212 107.50686
 [8] 104.19281 125.04160 123.45565
```

```
help("rbinom")
rbinom(10, size=10, prob=0.5)
```

```
 [1] 6 4 5 6 7 3 3 6 3 5
```

```
help("rpois")
rpois(10, lambda=10)
```

```
 [1]  3 16 12 15 18 11  5 13  8  7
```

```
help("rexp")
rexp(9, rate =0.1)
```

```
[1] 14.901857  2.254313 26.682531  9.901057 18.157325  1.353520 12.314702
[8]  6.519023  1.176382
```

```
help("rgamma")
rgamma(9, shape=2, rate=0.1)
```

```
[1]  1.270290 13.242888 21.272079  7.326098  2.183352 25.583747 38.297632
[8] 18.974723  5.528052
```

```
rnorm(3, mean = c(-10, 0, 10), sd = 1)
```

```
[1] -9.8798692 -0.3867954 10.2161628
```

Generando una muestra aleatoria

Si se desea una muestra aleatoria, podemos utilizar la función `sample(vec, n)`

```
help("sample")
sample(airquality$Wind, 10)
```

```
 [1] 16.6  8.0  9.2  9.7 11.5  6.3 16.6 14.9 14.3  8.0
```

La función `sample` normalmente reliza muestras sin reemplazo, lo que significa que no seleccionará el mismo elemento dos veces. Algunos procedimientos estadísticos (especialmente el bootstrap) requieren muestreo con reemplazo, lo que significa que un elemento puede aparecer varias veces en la muestra. Especificando `replace = TRUE` en `sample` con reemplazo.

Es fácil implementar un bootstrap simple mediante el muestreo con reemplazo. Este fragmento de código muestra repetidamente un conjunto de datos xy calcula la mediana de la muestra:

```
medianas <- numeric(1000)
for (i in 1:1000) {
medianas[i] <- median(sample(x, replace=TRUE))
}
```

A partir de las estimaciones de bootstrap, podemos estimar el intervalo de confianza para la mediana:

```
ci <- quantile(medians, c(0.025, 0.975))
cat("El intervalo de confianza 95% es (", ci, ")\n")
```

Generación de números aleatorios reproducibles

Si se desea generar una secuencia de números aleatorios, pero desea reproducir la misma secuencia cada vez que se ejecuta el programa.

Antes de ejecutar su código R, llame a la función `set.seed` para inicializar el generador de números aleatorios a un estado conocido:

```
set.seed(1978)
```

Generando secuencias aleatorias

Puedes generar secuencias aleatorias, tales como la simulación del lanzamiento de una moneda o otro ensayo de Bernoulli.

Usamos `sample(set, n, replace=TRUE)`

```
sample(c("H","T"), 10, replace=TRUE)
```

```
 [1] "H" "H" "T" "T" "T" "H" "T" "T" "H" "H"
```

```
 sample(c(FALSE,TRUE), 20, replace=TRUE)
```

```
 [1] FALSE FALSE FALSE  TRUE FALSE  TRUE FALSE FALSE  TRUE FALSE  TRUE
[12]  TRUE FALSE  TRUE FALSE FALSE  TRUE FALSE FALSE  TRUE
```

```
sample(c(FALSE,TRUE), 20, replace=TRUE, prob=c(0.2,0.8))
```

```
 [1]  TRUE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE
[12] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
```

Permutar aleatoriamente un vector

Si se desea generar una permutación aleatoria de un vector. Si `v` es un vector, entonces la `sample(v)` devuelve una permutación aleatoria. La función `sample(v)` es equivalente a `sample(v, size = length (v), replace = FALSE)`

```
sample(1:12)
```

```
 [1]  7 10 12  8  5  9  4  2  1  3 11  6
```

```
sample(letters[1:10])
```

```
##  [1] "h" "e" "c" "i" "j" "b" "f" "g" "d" "a"
```

Estructuras básicas de programación

Flujo de control

If-else

```
if(FALSE)
{
    message("Esto no se ejecuta....")
} else
{
    message("esto deberia ejecutarse.")
}
```

ifelse

```
(r <- round(rnorm(2), 1))

(x <- r[1] / r[2])

if(is.nan(x))
{
    message("x es un NA")
} else if(is.infinite(x))
{
    message("x es infinito")
} else if(x > 0)
{
    message("x es positivo")
} else if(x < 0)
{
    message("x es negativo")
} else
{
    message("x es cerp")
}
```

Bucles

while

```
accion <- sample(
    c(
        "Aprender R",
        "Estudiar CM-274",
        "Leer el manga de One Piece",
        "Salir con Jessica"
    ),
    1
)

while(accion != "Salir con Jessica"){

    message("Hoy es un buen dia")
     accion <- sample(
        c(
            "Aprender R",
            "Estudiar CM-274",
            "Leer el manga de One Piece",
            "Salir con Jessica"
        ),
        1
    )
    message("accion = ", accion)
}
```

Un ejemplo numérico del uso de while con los caminos aleatorios unidimensionales **https://es.wikipedia.org/wiki/Camino_aleatorio**.

```r
# Un camino aleatorio con While

x=0
n=0
set.seed(333)
while (x <= 10) {
  n=n+1
  x=x+rnorm(1,mean=.5,sd=1)
  }

print(paste ("n = ", n, ", x = ",round(x,2) ))
```

For

El bucle `for` acepta una variable de iteración y un vector. La sintaxis para el bucle `for` es

```
  for (nombre in valores ) expresion
```

El bucle `for`, iteraciona a través de los componentes nombre de valores uno a la vez. En el ejemplo anterior `nombre` toma el valor de cada elemento sucesivo de `valores`, hasta que se complete sus componentes.

```r
lenguajes <- c("Python", "JS", "C", "C++", "R", "Bash")
for(l in lenguajes){
  print(l)
}
```

```
[1] "Python"
[1] "JS"
[1] "C"
[1] "C++"
[1] "R"
[1] "Bash"
```

Ejercicios

1 . Usa R, para calcular las respuesta numéricas de lo siguiente:

- $1 + 2(3 + 4)$
- $4^3 + 3^{2+1}$
- $\sqrt{(4 + 3)(2 + 1)}$
- $\left(\frac{1+2}{3+4}\right)^2$

2 . La función `sd` calcula la desviación estándar. Calcula la desviación estándar desde el 0 al 100.

3 . Vea la demostración de símbolos matemáticos, usando `demo(plotmath)`.

4 . Genera aleatoriamente 1.000 mascotas , de las opciones `perro`, `gato`, `pollo` y `pez dorado`, con la misma probabilidad de que cada uno sea elegido. Muestra los primeros valores de la variable resultante y cuente el número de cada tipo de mascota.

5 . La *Conjetura de Collatz* señala que para todo número natural n, si se realiza la siguiente recursión:

$$f(n) = \begin{cases} 3n+1 & n = 2k+1 \\ \frac{n}{2} & n = 2k \end{cases}$$

Siempre se llegará a 1 luego de cierta cantidad de iteraciones. Para hallar la cantidad de pasos de un número se usa la siguiente iteración:

```r
n <- 100
pasos <- 1
while(n!=1){
    if(n %% 2 == 0){
        n <- n/2
    } else {
        n <- 3*n + 1
    }
    pasos <- pasos + 1
}
print(pasos)
```

Diseña un programa que halle la secuencia de menor longitud de entre los números en el rango $[100, 200]$ y además determine cuál es esa secuencia.

6 . Jessica estaba estudiando teoría de números y aprendió el algoritmo de Euclides, pero en la clase estaba tan concentrada que no llegó a apuntar correctamente el algoritmo dado por su profesor. A pesar de todo, ella recuerda exactamente todas las líneas, pero no el orden correcto. Dadas las siguientes líneas de código, reconstruya el algoritmo de Euclides iterativo y use $a = 10^5 + 3$ y $b = 10^8 + 9$:

```r
a <- 1001
b <- 7
while(b!=0){
    b <- carry
    a <- b
    carry <- a %% b
}
print(a)
```

7 . Usando la función `sample` obtenga un muestreo de 10 números en el rango $[1, 1000]$ (con reemplazo) y determine la relación entre la cantidad de primos encontrados y el tamaño de la muestra. Según la teoría de primos, una cota superior para la cantidad de primos menores o iguales a n es $\frac{n}{ln(n)}$, analice cuán preciso es esto con este caso y un muestreo de 20 números en el rango de $[1, 2000]$

8 . Supongamos que x es un vector numérico. Explica en detalle, como las siguientes expresiones son evaluadas y que valores toman

```r
sum(!is.na(x))
c(x,x[-(1:length(x))])
x[length(x) + 1]/length(x)
sum(x > mean(x))
```

9 .Usando la función `cumprod` o otra relacionada, calcula

$$1 + \frac{2}{3} + \left(\frac{2}{3}\frac{4}{5}\right) + \left(\frac{2}{3}\frac{4}{5}\frac{6}{7}\right) + \cdots + \left(\frac{2}{3}\frac{4}{5}\cdots\frac{38}{39}\right).$$

10 . Sea X el número de `unos` obtenidos en 12 lanzamientos de un dado. Entonces X tiene una distribución Binomial $(n = 12, p = 1/3)$. Calcule una tabla de probabilidades binomiales para $x = 0, 1, \ldots, 12$ por dos métodos:

- Usando la fórmula para la densidad: $P(X = K) = \binom{n}{k} p^k (1-p)^{n-k}$ y aritm'etica en R. Usa `0:12` para la secuencia de x valores y la función `choose` para calcular los coeficientes binomiales $\binom{n}{k}$.
- Usando la función `dbinom` de R y comparar tus resultados con ambos métodos.

11 . Sea X el número de `unos` obtenidos en 12 lanzamientos de un dado. Entonces X tiene una distribución Binomial $(n = 12, p = 1/3)$. Calcula el CDF para $x = 0, 1, \ldots, 12$ por dos métodos:

- Usando la función `cumsum` y el resultado del ejercicio anterior.
- Con el uso de la función `pbinom`. ¿ Qué es $P(X > 7)$?.

Listas

A diferencia de un vector, en el que todos los elementos deben ser del mismo tipo, la estructura de una lista en R puede combinar objetos de diferentes tipos. Una lista en R es similar a un diccionario de Python o, a un hash de Perl o puede resultar similar a una estructura de C. Las listas son muy importantes en R, formando la base para los data frames,la programación orientada a objetos,etc.

Se puede construir listas usando `list()`.

```
x <- list(1:3, "a", c(TRUE, FALSE, TRUE), c(2.3, 5.9))
str(x)
```

```
List of 4
 $ : int [1:3] 1 2 3
 $ : chr "a"
 $ : logi [1:3] TRUE FALSE TRUE
 $ : num [1:2] 2.3 5.9
```

Los elementos de una lista puede ser otra lista:

```
(lista_inicial <- list(
  lista_media = list(
    elemento_en_media_lista = diag(3),
    interior_lista = list(
      elemento_en_interior_lista = pi ^ 1:4,
      otro_elemento_en_interior_lista = "a"
      )
    ),
elemento_en_lista_inicial = log10(1:10)
))
```

```
$lista_media
$lista_media$elemento_en_media_lista
     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    1    0
[3,]    0    0    1

$lista_media$interior_lista
$lista_media$interior_lista$elemento_en_interior_lista
[1] 3.141593

$lista_media$interior_lista$otro_elemento_en_interior_lista
[1] "a"



$elemento_en_lista_inicial
 [1] 0.0000000 0.3010300 0.4771213 0.6020600 0.6989700 0.7781513 0.8450980
 [8] 0.9030900 0.9542425 1.0000000
```

Dos funciones son muy útiles para la aplicación de listas son `lapply()` y `sapply()`. Más información sobre la familia `apply()` en stackoverflow:r-grouping function.

```
lapply (list, f, fargs...)}
```

- `list` es una lista
- `f` es la función ha ser aplicada por ‘`lapply`
- `fargs` son otros paramétros

La función `lapply()` aplica una función a los elementos de una lista o vector y devuelve los resultados en una lista. La función `lapply()` es útil cuando se trata con data frames. En R, los `data frames` se consideran una lista y las variables son los elementos de la lista, así podemos aplicar una función a todas las variables en un data frame usando `lapply()`.

`sapply()` aplica una función a los elementos de una lista y devuelve los resultados en un vector, matriz o una lista.

`sapply(lista, f,...simplify)}`

- `list` es una lista
- `f` es la función ha ser aplicada por `sapply`
- `simplify` –> Cuando el argumento `simplify = F` entonces la función `sapply()` devuelve los resultados en una lista como la función `lapply()`. Cuando `simplify = T`, `sapply()` devuelve los resultados en una forma simplificada. Si los resultados son todos escalares entonces `sapply()` devuelve un vector. Si los resultados son todos de la misma longitud `sapply()` entonces devolverá una matriz con una columna para cada elemento en la lista a la que se aplicó la función.

```
x <- list(1:3, "a", c(TRUE, FALSE, TRUE), c(2.3, 5.9))
str(x)
```

Hay diferencias sutiles entre `NA` y `NULL` en las estructuras de datos de R:

```
v <-  c( 1, NA, NULL)
v
list(1, NA, NULL)
```

Data frames

Un data frame es la forma más común de almacenamiento de datos en R, y si se utiliza sistemáticamente en el análisis de datos como se explica el artículo datos ordenadados de Hadley Wickman.

Los data frame son una lista de vectores de la misma longitud, que hace que sea una estructura de dos dimensiones, que comparte propiedades tanto de las matrices y de las listas. Esto significa que un data frames tenga las funciones `names()`, `colnames()` y `rownames()`. La `length()` de una data frame es la longitud de la lista subyacente y así es el mismo valor que produce `ncol()`. `nrow()` da el número de filas.

Se puede tener parte de un data frame como una estructura de una dimensión (que se comporta como una lista), o una estructura de dos dimensiones (que se comporta como una matriz).

Se puede crear un data frame usando la función `data.frame()` de la siguiente manera:

```
help(data.frame)
df <- data.frame(x = 1:3, y = c("Python", "R", "C"))
str(df)
```

```
'data.frame':   3 obs. of  2 variables:
 $ x: int  1 2 3
 $ y: Factor w/ 3 levels "C","Python","R": 2 3 1
```

El comportamiento predeterminado de `data.frame()` convierte cadenas en factores. Usamos `stringAsFactors = FALSE` para suprimir ese comportamiento:

```
df <- data.frame(
  x = 1:3,
  y = c("Python", "R", "C"),
```

```
  stringsAsFactors = FALSE)
str(df)
```

Aplicaciones y ejemplos de vectores y matrices

Ejemplo de la matriz de covarianza, dado por Norman Matloff.

```
# Usando las funciones row() y col() cuyos argumentos son matrices

matrizcoV <- function(rho,n) {
    m <- matrix(nrow=n,ncol=n)
    m <- ifelse(row(m) == col(m),1,rho)
    return(m)
}
matrizcoV(0.2,3)
```

```
##      [,1] [,2] [,3]
## [1,]  1.0  0.2  0.2
## [2,]  0.2  1.0  0.2
## [3,]  0.2  0.2  1.0
```

Una práctica común en R es el de visualizar la dimensionalidad de los datos. Los datos siguientes representan las cifras iniciales en la verificación de una oficial de reembolso de seguro, cuando un auditor financiero, le pidió investigar por fraude.

```
accdato <- c(1, 132, 86.7,
             2, 50, 50.7,
             3, 32, 36.0,
             4, 20, 27.9,
             5, 19, 22.8,
             6, 11, 19.3,
             7, 10, 16.7,
             8, 9, 14.7,
             9, 5, 13.2)
accdato
```

```
##  [1]   1.0 132.0  86.7   2.0  50.0  50.7   3.0  32.0  36.0   4.0  20.0
## [12]  27.9   5.0  19.0  22.8   6.0  11.0  19.3   7.0  10.0  16.7   8.0
## [23]   9.0  14.7   9.0   5.0  13.2
```

Acomodamos mejor el vector, como una matriz 9x3 cuyas columnas se llaman número, actual, v_esperado

```
accdato <- matrix(accdato, 9, 3, byrow = TRUE)
colnames(accdato) <- c("numero", "actual", "v_esperado")
accdato
```

```
##       numero actual v_esperado
##  [1,]      1    132       86.7
##  [2,]      2     50       50.7
##  [3,]      3     32       36.0
##  [4,]      4     20       27.9
##  [5,]      5     19       22.8
##  [6,]      6     11       19.3
##  [7,]      7     10       16.7
##  [8,]      8      9       14.7
##  [9,]      9      5       13.2
```

Podemos ahora usar el test de *chi-cuadrado* para averiguar si es que ha habido un fraude, a través de la desviación significativa o no de los datos reales y de los esperados. Escribamos el test

```
chi2 <-sum((accdato[,2] -accdato[,3])^2/accdato[,3])
chi2
```

```
## [1] 40.55482
```

El poder de las matrices en R, ocurre cuando realizamos operaciones (multiplicación, inversión, transposición, etc) comunes de matrices y sus respectivas restricciones

```
A <- matrix(c(6,2,3, 4,
              0, -8,2, 1,
              8, -3, 7, -5),3, 4, byrow = TRUE)
B <- matrix(c(-7,12,3, 9,
              6, 2, 0, -1,
              11, 5, -12, 8),3, 4, byrow = TRUE)

A + B
```

```
##      [,1] [,2] [,3] [,4]
## [1,]   -1   14    6   13
## [2,]    6   -6    2    0
## [3,]   19    2   -5    3
```

```
A -B
```

```
##      [,1] [,2] [,3] [,4]
## [1,]   13  -10    0   -5
## [2,]   -6  -10    2    2
## [3,]   -3   -8   19  -13
```

```
A * B   # Multiplicacion componente a componente
```

```
##      [,1] [,2] [,3] [,4]
## [1,]  -42   24    9   36
## [2,]    0  -16    0   -1
## [3,]   88  -15  -84  -40
```

```
B1 <- matrix(c(-7,12,3, 9,
              6, 2, 0, -1,
              11, 5, -12, 8),4, 5, byrow = TRUE)
```

```
## Warning in matrix(c(-7, 12, 3, 9, 6, 2, 0, -1, 11, 5, -12, 8), 4, 5, byrow
## = TRUE): data length [12] is not a sub-multiple or multiple of the number
## of columns [5]
```

```
A %*% B1  # Multiplicacion usual de matrices
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]  -38  120    3  112   51
## [2,]  -31   22   -4  -64  -35
## [3,] -191  122  -32  123   59
```

El cálculo de la inversa de una matriz se realiza de la siguiente manera

```
A <- matrix(c(4, 10, 6, 1 , 13, 5,
           4,5, -2, 1, 8, 4,
             -7, 5, 2, 1, 3, -4,
             2,3, 4 ,5 , 6, 7,
```

```
            1,3, 5,6, 7,-3,
            0, 9, 7, 1,3, 4), 6, 6, byrow=TRUE)
B <- solve(A)   ## Calculamos  la inversa de A
B
```

```
##              [,1]        [,2]          [,3]        [,4]         [,5]
## [1,] -0.025081101  0.06307221 -0.1373002523 -0.10063679  0.088207377
## [2,] -0.116454403  0.15776162  0.0044154752 -0.08475309  0.035185630
## [3,]  0.114051424 -0.18525171 -0.0355340622  0.01403340  0.007995915
## [4,] -0.182416196  0.12315872  0.0002102607  0.06399135  0.110519044
## [5,]  0.132193920 -0.05770155  0.0494112700  0.03796708 -0.028024751
## [6,]  0.008891025 -0.01828668  0.0151387721  0.12166286 -0.099771717
##             [,6]
## [1,]  0.07324883
## [2,]  0.16692899
## [3,] -0.01140815
## [4,]  0.07597621
## [5,] -0.14559053
## [6,] -0.01542713
```

```
A %*% B
```

```
##               [,1]          [,2]          [,3]          [,4]          [,5]
## [1,]  1.000000e+00 -2.532696e-16  5.898060e-17 -1.665335e-16  4.163336e-17
## [2,] -9.714451e-17  1.000000e+00  1.249001e-16  0.000000e+00  5.551115e-17
## [3,] -4.163336e-17  5.551115e-17  1.000000e+00  0.000000e+00  5.551115e-17
## [4,] -5.898060e-17 -9.367507e-17  7.979728e-17  1.000000e+00 -4.163336e-17
## [5,] -1.040834e-17 -1.700029e-16  5.204170e-17  2.220446e-16  1.000000e+00
## [6,] -3.469447e-16  2.359224e-16 -7.632783e-17 -5.551115e-17  5.551115e-17
##               [,6]
## [1,] -6.019490e-16
## [2,] -2.151057e-16
## [3,] -6.938894e-18
## [4,] -2.099015e-16
## [5,] -1.717376e-16
## [6,]  1.000000e+00
```

```
B %*% A
```

```
##               [,1]          [,2]          [,3]          [,4]          [,5]
## [1,]  1.000000e+00  0.000000e+00  0.000000e+00  2.775558e-17 -1.110223e-16
## [2,]  6.938894e-17  1.000000e+00 -1.942890e-16  0.000000e+00 -1.942890e-16
## [3,] -1.196959e-16 -4.024558e-16  1.000000e+00 -3.122502e-17 -1.595946e-16
## [4,] -2.775558e-17 -1.110223e-16 -1.110223e-16  1.000000e+00 -6.383782e-16
## [5,] -3.122502e-17 -2.775558e-16  2.775558e-16  0.000000e+00  1.000000e+00
## [6,]  1.387779e-17  1.561251e-17 -1.127570e-16 -8.153200e-17  7.459311e-17
##               [,6]
## [1,] -2.775558e-16
## [2,] -2.220446e-16
## [3,] -1.040834e-16
## [4,]  5.551115e-17
## [5,]  0.000000e+00
## [6,]  1.000000e+00
```

Para ejemplos de orden menor, se cumple que $AB = I = BA$

```
M <- matrix(c(4,0,5,
              0,1, -6,
              3,0,4),3,3, byrow=TRUE)
N <- solve(M)
N
```

```
##      [,1] [,2] [,3]
## [1,]    4    0   -5
## [2,]  -18    1   24
## [3,]   -3    0    4
```

```
N%*%M
```

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

```
M %*%N
```

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

Uno de los aspectos más importantes de **R** es la familia de funciones `apply()` tales como `apply()`, `tapply()` y `lapply()`. La función `apply()` instruye a R a llamar una función sobre cada fila o columna de una matriz. Esta es la forma general de `apply()` para matrices

```
apply(m, dimcode, f, fargs)
```

donde los argumentos son como sigue:

- `m` es una matriz.
- `dimcode` es la dimensión, igual a 1, si la función aplica a filas o 2 si la función aplica a columnas.
- `f` es la función a ser aplicada.
- `fargs` es un conjunto opcional de argumentos dados a `f`.

```
help(apply)
dw<- matrix(c(1,2,3,4,5,6), nrow=3)
dw
```

```
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
```

```
apply(dw, 1, mean)
```

```
## [1] 2.5 3.5 4.5
```

Un ejemplo utilizando una función escrita por el usuario

```
dw
```

```
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
```

```
f <-function(x)x/c(2,5)
dw1<- apply(dw, 1, f)
dw1
```

```
##      [,1] [,2] [,3]
## [1,]  0.5    1  1.5
## [2,]  0.8    1  1.2
```

```
t(apply(dw,1,f))
```

```
##      [,1] [,2]
## [1,]  0.5  0.8
## [2,]  1.0  1.0
## [3,]  1.5  1.2
```

Un ejemplo más avanzado desde Norman Matloff, (The Art of R Programming)

```
matriz0 <-function(fila,d) {
    maj <- sum(fila[1:d]) / d
    return(if(maj > 0.5) 1 else 0)
}
ax <-matrix(c(1,1,1,0, 0,1,0,1, 1, 1, 0,1, 1,1,1,1, 0,0,0,0), nrow=4)
ax
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    0    1    1    0
## [2,]    1    1    1    1    0
## [3,]    1    0    0    1    0
## [4,]    0    1    1    1    0
```

```
apply(ax,1,matriz0,3)
```

```
## [1] 1 1 0 1
```

```
apply(ax,1,matriz0,2)
```

```
## [1] 0 1 0 0
```

El uso de `apply()` no acelera el código. El beneficio de utilizar esta función es que hace el código mucho más compacto y por tanto más fácil de leer y modificar evitando posibles errores en la escritura de código cuando se trabaja con bucles.

Por otra parte, cuando R desarrolla el procesamiento en paralelo, funciones como `apply()` serán muy útiles e importantes. Por ejemplo, la función `clusterApply()` en el paquete 'snow" de R da cierta capacidad de procesado en paralelo mediante la distribución de los datos de una submatriz a varios nodos de una red.

La función `outer` aplica una función a dos arrays

```
help(outer)
x<- c(1,2.3,2,3,4,8,12,43)
y<- c(2,4)
outer(x,y,"log")
```

```
##          [,1]      [,2]
## [1,] 0.000000 0.0000000
## [2,] 1.201634 0.6008169
## [3,] 1.000000 0.5000000
## [4,] 1.584963 0.7924813
## [5,] 2.000000 1.0000000
## [6,] 3.000000 1.5000000
```

```
## [7,] 3.584963 1.7924813
## [8,] 5.426265 2.7131324
```

```
(valores <- outer(1:5, 1:5, FUN = "paste", sep = ","))
```

```
##      [,1]  [,2]  [,3]  [,4]  [,5]
## [1,] "1,1" "1,2" "1,3" "1,4" "1,5"
## [2,] "2,1" "2,2" "2,3" "2,4" "2,5"
## [3,] "3,1" "3,2" "3,3" "3,4" "3,5"
## [4,] "4,1" "4,2" "4,3" "4,4" "4,5"
## [5,] "5,1" "5,2" "5,3" "5,4" "5,5"
```

```
valores[c(4, 15)]
```

```
## [1] "4,1" "5,3"
```

El método del Simplex

Sea el problema de programación lineal, que será resuelto por el simplex algorithm

min C = 5x1 + 8x2

sujeto a los restricciones

x1 + x2 >=2

x1 + 2x2 >=3

y

x1, x2 >= 0

Para resolver este problema usamos los paquetes lpSolve-lpSolveApi de la siguiente manera

```
install.packages("lpSolve")
install.packages("lopSolveAPI")
```

```
library(lpSolve)
ejemplo.pl <- lp(objective.in=c(5, 8), const.mat=matrix(c(1, 1, 1, 2),nrow=2), const.rhs=c(2, 3), const
ejemplo.pl
```

```
## Success: the objective function is 13
```

```
ejemplo.pl$solution
```

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

La salida nos dice que el valor que minimiza el problema, está en x1 = 1, x2 = 1 y el mínimo valor de la función objetivo es 13.

Descomposición de Cholesly

La factorización de Cholesly es un método de descomposición de una matriz definida positiva. Algunas aplicaciones de la descomposicion de Cholesky incluye las soluciones de ecuaciones lineales, simulación de Montecarlo y los filtros de Kalman.

La función `chol()` lleva a cabo la descomposición de una matriz definida positiva.

```
A = as.matrix(data.frame(c(3,4,3),c(4,8,6),c(3,6,9)))
colnames(A) <- NULL
A
```

```
##      [,1] [,2] [,3]
## [1,]    3    4    3
## [2,]    4    8    6
## [3,]    3    6    9
```

Factorizamos la matriz con la función `chol()`

```
A.chol <- chol(A)
A.chol
```

```
##          [,1]     [,2]     [,3]
## [1,] 1.732051 2.309401 1.732051
## [2,] 0.000000 1.632993 1.224745
## [3,] 0.000000 0.000000 2.121320
```

La función `chol()` devuelve una matriz triangular superior. La transposición de la matriz produce una matriz triangular inferior

```
t(A.chol)
```

```
##          [,1]     [,2]    [,3]
## [1,] 1.732051 0.000000 0.00000
## [2,] 2.309401 1.632993 0.00000
## [3,] 1.732051 1.224745 2.12132
```

y esto coincide con el resultado de la salida de la función `chol()`. Podemos verificar el resultado

```
t(A.chol) %*% A.chol
```

```
##      [,1] [,2] [,3]
## [1,]    3    4    3
## [2,]    4    8    6
## [3,]    3    6    9
```

Lectura : Algorithm for Cholesky decomposition.