Práctica 2: Simulación con R

1. Otros tipos de datos en R

Como vimos en la Práctica 1, la forma más sencilla de definir un vector con datos es usar la función `c()`:

```
> x <- c(3.141592, 2.718281, 1.618034)
> x

[1] 3.141592 2.718281 1.618034
```

La forma de acceder a los elementos del vector era la siguiente:

```
> x[2]

[1] 2.718281

> x[3]

[1] 1.618034
```

Se pueden asignar nombres a los elementos de un vector mediante la función `names()`:

```
> names(x) <- c("pi", "e", "phi")
> x

      pi        e      phi
3.141592 2.718281 1.618034
```

Los valores de π y e se pueden obtener en R con

```
> pi

[1] 3.141593
```

```
> exp(1)

[1] 2.718282
```

phi es también conocido como el número áureo y es igual a

```
> (1 + sqrt(5))/2

[1] 1.618034
```

Ahora, si queremos referirnos at tercer elemento, podemos optar por `x[3]` o, dado que hemos asignado nombres a los elementos del vector, podemos hacer

```
> x["phi"]

     phi
1.618034
```

2. Variables lógicas en R

2.1. Índices lógicos

Supongamos ahora que tenemos el vector

```
> x <- 1:5
```

podemos ver qué elementos son iguales a 4 utilizando el operador `==`:

```
> x == 4

[1] FALSE FALSE FALSE  TRUE FALSE
```

La instrucción anterior devuelve un vector de valores lógicos (o *booleanos*). `FALSE` indica que la condición anterior (es decir, ser igual a cuatro) no se cumple mientras que `TRUE` indica que sí se cumple.

De manera análoda podemos hacer comparaciones con `<`, `>`, `<=` y `>=`:

```
> x < 4

[1]  TRUE  TRUE  TRUE FALSE FALSE

> x >= 4
```

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

Si aplicamos la función `sum()` al vector lógico resultante, `R` fuerza TRUE al valor numérico 1 y FALSE al 0, de manera que tenemos el número de elementos del vector `x` que cumplen la condición impuesta:

```
> sum(x < 4)

[1] 3

> sum(x >= 4)

[1] 2
```

Generaremos un vector aleatorio de 100 números entre 0 y 1 de manera similar a como hicimos en la Práctica 1:

```
> set.seed(111)
> y <- runif(100)
```

La media de este vector es

```
> mean(y)

[1] 0.4895239
```

Contamos cuántos valores están por debajo de la media:

```
> sum(y < mean(y))

[1] 51
```

o cuántos por debajo de la mediana

```
> sum(y < median(y))

[1] 50
```

Veamos ahora el efecto que tiene colocar un vector lógico entre los corchetes de índice de un vector

```
> z <- 1:5
> z[c(TRUE, FALSE, TRUE, FALSE, TRUE)]
```

```
[1] 1 3 5
```

donde el vector lógico `c(TRUE, FALSE, TRUE, FALSE, TRUE)` es uno de igual longitud que `x` y donde `TRUE` y `FALSE` representan verdadero y falso, respectivamente.

El efecto de la instrucción es el de devolver únicamente aquellos valores de `x` cuyo índice valga `TRUE`.

La instrucción `y[y<0.1]` devuelve no cuántos, sino cuáles son los valores que cumplen la condición de estar por debajo de 0.1:

```
> y[y < 0.1]

 [1] 0.0106578451 0.0936815199 0.0671411434 0.0475478533 0.0585964625
 [6] 0.0327716474 0.0965785654 0.0009253006 0.0525656715 0.0117258150
[11] 0.0533223320 0.0951241532 0.0866140136
```

2.2. Condiciones lógicas compuestas

Podemos utilizar otros operadores lógicos dentro de los corchetes, con condiciones compuestas. La instrucción siguiente, en las que `|` es el operador lógico *o*, devuelve aquellos valores de `y` que están por debajo de 0.05 o por encima de 0.95:

```
> y[(y < 0.05) | (y > 0.95)]

 [1] 0.0106578451 0.0475478533 0.9665342933 0.9675274789 0.0327716474
 [6] 0.9967166614 0.0009253006 0.0117258150 0.9837744189 0.9935344572
[11] 0.9750010339
```

Veamos que son 11. Si quisiéramos contarlos solamente, pondríamos:

```
> sum((y < 0.05) | (y > 0.95))

[1] 11
```

Pero si queremos sumar los valores que cumplen la condición anterior:

```
> sum(y[(y < 0.05) | (y > 0.95)])

[1] 5.986717
```

Podríamos desear los valores de y que están por encima de 0.45 **y** por debajo de 0.55:

```
> y[(y > 0.05) & (y < 0.95)]

 [1] 0.59298128 0.72648112 0.37042200 0.51492383 0.37766322 0.41833733
 [7] 0.53229524 0.43216062 0.09368152 0.55577991 0.59022849 0.06714114
[13] 0.15620252 0.44642776 0.17144369 0.31066643 0.61446640 0.43106079
[19] 0.28552709 0.34215135 0.38662763 0.32202673 0.65322945 0.28330350
[25] 0.78742792 0.59592064 0.05859646 0.50989986 0.46579243 0.46935909
[31] 0.35974537 0.71341035 0.11631548 0.78399262 0.64214071 0.80510091
[37] 0.64119786 0.32849165 0.63569095 0.92851916 0.57524220 0.36668385
[43] 0.43660722 0.85592194 0.62799557 0.79377564 0.72516483 0.58504472
[49] 0.33299468 0.54827337 0.57583294 0.45631521 0.09657857 0.80554018
[55] 0.46674405 0.17326087 0.25922256 0.91928208 0.23192958 0.05256567
[61] 0.30439262 0.30070770 0.87758395 0.66527873 0.45376483 0.05332233
[67] 0.63090681 0.44218519 0.26734649 0.09512415 0.78596912 0.11985218
[73] 0.88121547 0.13109807 0.40033788 0.08661401 0.37479980 0.68478602
[79] 0.73477268 0.77094774 0.57998535 0.51109898 0.85298371 0.62982116
[85] 0.57900591 0.74024929 0.38714976 0.39808948 0.82448220
```

donde `&` es el operador lógico *y*.

La función `which` devuelve las posiciones en el vector de aquellos valores que cumplen la condición:

```
> which((y > 0.05) & (y < 0.95))

 [1]   1   2   3   4   5   6   8   9  10  11  12  13  15  16  17  19  20  21  22
[20]  23  24  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42
[39]  43  44  45  46  47  48  49  50  51  52  54  56  57  58  59  60  62  63  64
[58]  65  66  67  68  70  71  72  73  74  75  76  77  79  80  81  82  83  84  85
[77]  86  87  88  89  90  91  92  93  94  95  96  98 100
```

Estas son las posiciones que ocupan los 89 valores que obtuvimos anteriormente.

En resumen, en este apartado hemos visto cuáles, cuántos y qué posiciones ocupan los valores que cumplen con determinadas condiciones impuestas.

Poner un ejemplo de detección de valores anómalos usando el diagrama de cajas

2.3. Exclusión de valores

Sea

```
> z <- 3:10
```

la instrucción `z[-1]` devuelve un vector igual al original de que se ha eliminado el primer elemento:

```
> z[-1]

[1]  4  5  6  7  8  9 10
```

Si lo que deseamos es eliminar los tres primeros elementos, entonces debemos hacer

```
> z[-(1:3)]

[1]  6  7  8  9 10
```

O los tres últimos

```
> z[-((length(z) - 2):length(z))]

[1] 3 4 5 6 7
```

Si bien puede parecer más natural, para el mismo objetivo podemos usar

```
> rev(z)[-(1:3)]

[1] 7 6 5 4 3
```

que nos da un vector resultante en orden inverso al original, aspecto que se puede corregir con

```
> rev(rev(z)[-(1:3)])

[1] 3 4 5 6 7
```

si, por lo que fuera, nos interesa conservar el orden.

Con

```
> z[-which(z < 6)]

[1]  6  7  8  9 10
```

excluimos aquellos valores menores que 6.

Si queremos recortar los dos valores extremos menores y los dos mayores del vector, podemos hacer:

```
> w <- c(5, 2, 8, 5, 4, 6, 9, 9)
> w

[1] 5 2 8 5 4 6 9 9

> rev(sort(w)[-c(1, 2)])[-c(1, 2)]

[1] 8 6 5 5
```

Expresión en la que, primero, hemos ordenado el vector original con `sort`, luego, excluido los valores menores, a continuación, hemos invertido el vector resultante con `rev` y, finalmente, hemos suprimido los dos primeros valores, que son los dos mayores del vector original.

3. Matrices en R

Además de los objetos vistos hasta aquí, introducimos ahora este nuevo tipo. Podemos crear una matriz a partir de un vector

```
> x <- 1:12
> y <- matrix(x, nrow = 3)
> y

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

Con `matrix`, se crea una matriz de tres filas (`nrow=3`) a partir de un vector (`x`) de longitud 12. Nótese que los valores del vector se introducen, por defecto, por columnas. El mismo efecto podría haberse conseguido con

```
> y <- matrix(x, ncol = 4)
> y

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

y se crea la misma matriz con 4 columnas y, por lo tanto, tres filas, pues `ncol=4` obliga a que así sea. Si queremos que se introduzcan por filas, se ha de proceder así

```
> y <- matrix(x, nrow = 3, byrow = T)
> y

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

Véase el efecto de

```
> y <- matrix(x, ncol = 3, byrow = T)
> y

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

con la que se ha construido una matriz de tres columnas, con los datos introducidos por filas. Podemos, también, convertir directamente el vector original en una matriz redefiniendo las dimensiones del vector.

Véase

```
> class(x)

[1] "integer"
```

que nos muestra que `x` es un vector de enteros y el efecto que tiene el redimensionar `x` con la función `dim`

```
> dim(x) <- c(3, 4)
> x

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

> class(x)

[1] "matrix"
```

que nos muestra que, ahora, `x` ya no es más un vector y que ha pasado a ser una matriz. Se pueden asignar nombres a filas y columnas con `rownames`y `colnames`

```
> rownames(x) <- c("lunes", "martes", "miércoles")
> x
```

```
          [,1] [,2] [,3] [,4]
lunes        1    4    7   10
martes       2    5    8   11
miércoles    3    6    9   12
```

Podemos referirnos a los elementos de una matriz indicando la posición con corchetes

```
> x[1, 2]
```

```
lunes
    4
```

o bien por medio de los nombres de la fila y columna, si los tiene

```
> colnames(x) <- c("marzo", "abril", "mayo", "junio")
> x["lunes", "abril"]
```

```
[1] 4
```

Podemos referir una columna de esta manera

```
> x[, 2]
```

```
    lunes    martes miércoles
        4         5         6
```

que nos muestra los elementos de la columna 2 (aunque mostrados en forma de fila). O bien la fila 3

```
> x[3, ]
```

```
marzo abril  mayo junio
    3     6     9    12
```

Nótese cómo, tanto en un caso como en otro, la ausencia de número en la fila o columna se puede leer como "para todas las filas" y "para todas las columnas", respectivamente. Podemos seleccionar las dos primeras filas de la matriz

```
> x[1:2, ]
```

```
       marzo abril mayo junio
lunes      1     4    7    10
martes     2     5    8    11
```

O bien, excluirlas

```
> x[-(1:2), ]
```

```
marzo abril  mayo junio
    3     6     9    12
```

3.1. Operaciones con matrices

Calcular la suma de todos los elementos de la matriz

```
> sum(x)
```

```
[1] 78
```

o bien, la media de todos los elementos

```
> mean(x)
```

```
[1] 6.5
```

o bien, la matriz de varianzas/covarianzas

```
> var(x)
```

```
      marzo abril mayo junio
marzo     1     1    1     1
abril     1     1    1     1
mayo      1     1    1     1
junio     1     1    1     1
```

matriz en la que el elemento `x[i,i]` representa la varianza de la columna `x[,i]`, y el `x[i,j]`, con $i \neq j$, la covarianza de las columnas `x[,i]` y `x[,j]`.

Calcular la media de valores de la segunda columna

```
> mean(x[, 2])
```

```
[1] 5
```

o, bien la varianza de la segunda fila

```
> var(x[2, ])
```

```
[1] 15
```

Calcular las sumas de los elementos dentro de cada columna

```
> colSums(x)
```

```
marzo abril  mayo junio
    6    15    24    33
```

o las medias de las mismas

```
> colMeans(x)
```

```
marzo abril  mayo junio
    2     5     8    11
```

o las sumas de los elementos dentro de cada fila

```
> rowSums(x)
```

```
    lunes    martes miércoles
       22        26        30
```

Con `summary`, función que es sensible al contexto

```
> summary(x)
```

```
     marzo          abril          mayo          junio
 Min.   :1.0   Min.   :4.0   Min.   :7.0   Min.   :10.0
 1st Qu.:1.5   1st Qu.:4.5   1st Qu.:7.5   1st Qu.:10.5
 Median :2.0   Median :5.0   Median :8.0   Median :11.0
 Mean   :2.0   Mean   :5.0   Mean   :8.0   Mean   :11.0
 3rd Qu.:2.5   3rd Qu.:5.5   3rd Qu.:8.5   3rd Qu.:11.5
 Max.   :3.0   Max.   :6.0   Max.   :9.0   Max.   :12.0
```

obtenemos resúmenes por columnas.

Con `apply` podemos aplicar funciones como `var` para calcular las varianzas de los elementos de cada una de las filas

```
> apply(x, 1, var)

    lunes    martes miércoles
       15        15        15
```

donde el parámetro 1 indica que trabajamos por filas y podríamos añadir una nueva columna a la matriz x con estas varianzas, mediante `cbind`

```
> xx <- cbind(x, apply(x, 1, var))
> xx

          marzo abril mayo junio
lunes         1     4    7    10 15
martes        2     5    8    11 15
miércoles     3     6    9    12 15
```

y podríamos nombrar "varianzas" a la última columna añadida

```
> colnames(xx) <- c(colnames(x), "varianzas")
> xx

          marzo abril mayo junio varianzas
lunes         1     4    7    10        15
martes        2     5    8    11        15
miércoles     3     6    9    12        15
```

Análogamente, podemos trabajar por columnas con

```
> apply(x, 2, var)

marzo abril  mayo junio
    1     1     1     1
```

donde el parámetro 2 indica que trabajamos por columnas y podríamos añadir esta fila a la matriz

```
> xxx <- rbind(x, apply(x, 2, var))
> xxx
```

```
          marzo abril mayo junio
lunes         1     4    7    10
martes        2     5    8    11
miércoles     3     6    9    12
              1     1    1     1
```

y darle nombre a la nueva fila

```
> rownames(xxx) <- c(rownames(xx), "varfil")
> xxx

          marzo abril mayo junio
lunes         1     4    7    10
martes        2     5    8    11
miércoles     3     6    9    12
varfil        1     1    1     1
```

Producto de matrices

```
> x <- matrix(1:4, nrow = 2)
> y <- matrix(2:5, nrow = 2)
> x

     [,1] [,2]
[1,]    1    3
[2,]    2    4

> y

     [,1] [,2]
[1,]    2    4
[2,]    3    5

> x %*% y

     [,1] [,2]
[1,]   11   19
[2,]   16   28
```

Producto por un escalar

```
> 2 * x
```

```
     [,1] [,2]
[1,]    2    6
[2,]    4    8
```

Suma y resta de matrices

```
> x + y
```

```
     [,1] [,2]
[1,]    3    7
[2,]    5    9
```

```
> x - y
```

```
     [,1] [,2]
[1,]   -1   -1
[2,]   -1   -1
```

Crear una matriz diagonal

```
> z <- diag(c(2, -1, 3))
> z
```

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

Modificar la diagonal

```
> diag(z) <- c(4, 3, 3)
> z
```

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

Hallar un determinante

```
> det(x)
```

```
[1] -2
```

Hallar la inversa de una matriz

```
> solve(x)

     [,1] [,2]
[1,]   -2  1.5
[2,]    1 -0.5
```

y comprobamos

```
> x %*% solve(x)

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

No confundir el producto `%*%` con el producto `*`.
Producto Hadamard de dos matrices

```
> x * y

     [,1] [,2]
[1,]    2   12
[2,]    6   20
```

es la matriz cuyos elementos son el producto de los elementos homónimos de las matrices que se multiplican.

4. Simulaciones

La función `sample` es muy útil para efectuar simulaciones

```
> sample(1:10, 6)

[1] 6 2 8 9 5 1
```

extrae del vector 1:10, seis elementos al azar sin repetición

```
> sample(1:10, 6, rep = T)

[1]  7  3  6 10  2  7
```

extrae del mismo vector 6 elementos en los que se admite repetición.

4.1. Dados

Para simular la tirada de un dado podemos utilizar

```
> sample(1:6, 1)

[1] 6
```

Para simular la tirada de 4 dados, o de un mismo dado 4 veces, podemos utilizar

```
> sample(1:6, 4, rep = T)

[1] 6 6 4 4
```

admitiendo repetición.

Si queremos simular la distribución de la suma de los números que salen al tirar 4 dados

```
> t <- sapply(1:10000, function(x) {
+     sum(sample(1:6, 4, rep = T))
+ })
```

donde la función `sapply` aplica a un vector de tamaño 10000 una función sin nombre, generando a su vez un vector de tamaño 10000. La función obtiene muestras con repetición de tamaño 4 y, a continuación, suma los números de la muestra. Este proceso se repite 10000 veces. Lo mismo se podría haber conseguido con un ciclo `for`, pero el procedimiento utilizado es más rápido. Para garantizar que los resultados son los mismos que los de esta práctica, nos servimos de `set.seed(111)`

```
> set.seed(111)
> t <- sapply(1:10000, function(x) {
+     sum(sample(1:6, 4, rep = T))
+ })
```

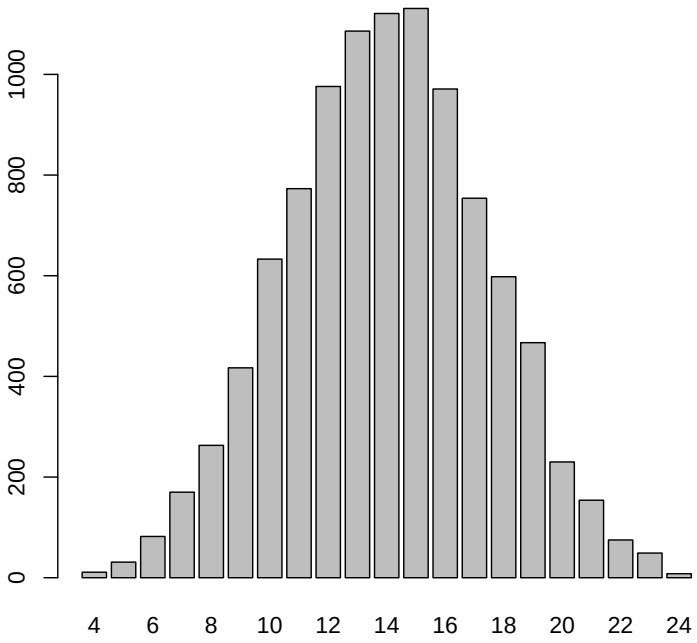
y tabulamos los resultados con

```
> table(t)

t
   4    5    6    7    8    9   10   11   12   13   14   15   16   17   18   19
  11   31   82  170  263  417  633  773  976 1086 1121 1131  971  754  598  467
  20   21   22   23   24
 230  154   75   49    8
```

y podemos representar los resultados con un diagrama de barras

```
> barplot(table(t))
```



Se podría haber procedido similarmente así:

```
> x <- runif(4, 0, 6)
> x

[1] 5.02819982 1.82267011 5.95054520 0.05559562
```

genera 4 números aleatorios uniformemente distribuidos entre 0 y 6.

La función

```
> ceiling(x)

[1] 6 2 6 1
```

transforma los valores anteriores en el menor entero no inferior al número (digamos que el número siempre se redondea por arriba).

Combinando las dos instrucciones, se pueden generar números aleatorios entre 1 y 6.

```
> ceiling(runif(4, 0, 6))

[1] 1 4 6 1
```

Con la siguiente instrucción podríamos conseguir

```
> t <- sapply(1:10000, function(x) {
+     sum(ceiling(runif(4, 0, 6)))
+ })
```

lo mismo que anteriormente con

```
> t <- sapply(1:10000, function(x) {
+     sum(sample(1:6, 4, rep = T))
+ })
```

Si lo que se quiere es simular el número de veces que sale, digamos, un seis, en, por ejemplo, 10 tiradas, como el número de seises sigue una distribución binomial de n=10 y p=1/6, podemos

utilizar la función de `R` `rbinom` para generar números aleatorios según esta distribución y realizar la simulación más cómodamente.

Con la siguiente expresión se generan 12 números aleatorios en las condiciones antedichas

```
> rbinom(12, 10, 1/6)

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

que se interpreta así: en la primera tirada de 10 dados salieron 2 seises; en la segunda, 0; en la tercera, 0...

Si generamos 10000 números aleatorios con

```
> set.seed(111)
> t <- rbinom(10000, 10, 1/6)
```

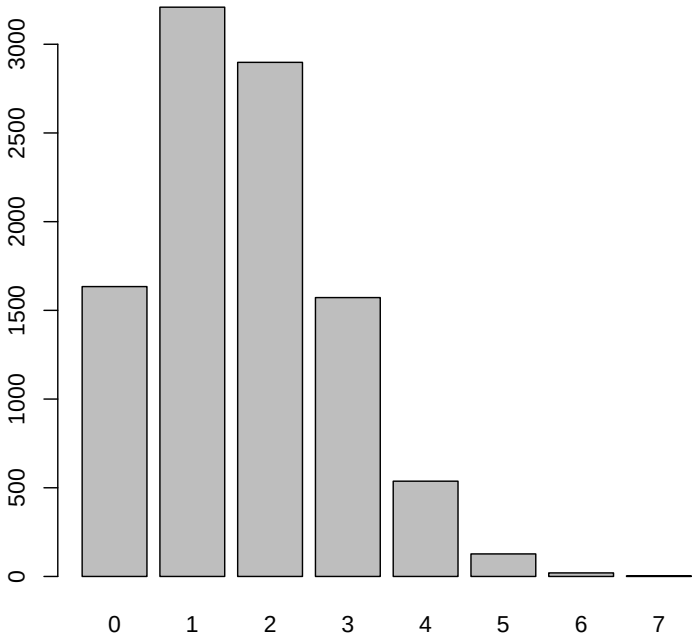
que tabulamos con

```
> table(t)

t
   0    1    2    3    4    5    6    7
1634 3209 2898 1572  537  127   20    3
```

y representamos con

```
> barplot(table(t))
```



obtenemos un diagrama del todo análogo al anteriormente obtenido.