M. Jesús García-Ligero
Patricia Román Román

SIMULACIÓN CON R

Instalación de R

- Página principal de R (si se pone en Google "R", es la página asociada con "The R Project for Statistical Computing"):

  www.r-project.org/

- En la columna izquierda seleccionamos

  Download, Packages
  CRAN

- A continuación buscamos, dentro de CRAN Mirrors, el pais España

  Spain

  http://cran.es.r-project.org/ Spanish National Research Network, Madrid

  y accedemos a dicha página web.

- En la sección

  Download and Install R

  Precompiled binary distributions of the base system and contributed packages, **Windows and Mac** users most likely want one of these versions of R:

  - Linux
  - MacOS X
  - Windows

  seleccionamos el sistema operativo de nuestro ordenador.

- A continuación, seleccionamos el subdirectorio base

base Binaries for base distribution (managed by Duncan Murdoch)

y descargamos la versión disponible:

Download R 2.10.1 for Windows (32 megabytes)

Todo lo anterior (si se selecciona el sistema operativo Windows) se puede hacer accediendo directamente a la página

http://cran.es.r-project.org/bin/windows/base

Editor Tinn-R

Se puede descargar por ejemplo de la página http://www.sciviews.org/Tinn-R/

Instalar paquetes

Cargar paquetes

Comenzando con R

Al ejecutar el programa en la interfaz de RGui aparece el símbolo > esperando la entrada de instrucciones.

El menú principal contiene las pestañas típicas de otras aplicaciones: *Archivo, Editar, Visualizar, Ventanas* y *Ayuda*, junto con una específicas de R: *Misc* y *Paquetes*.

Ayuda en R

Para solicitar ayuda sobre un tema general podemos escribir, por ejemplo

```
>help(Uniform)
```

y obtenemos una ventana de ayuda sobre la distribución uniforme. También podemos acceder a la misma información a través del menú *Ayuda/Funciones R(texto).*

Si no se conoce la grafía de una expresión podemos escribir
```
> apropos("vector")
```
y aparecerán las expresiones que contienen el término introducido.

Con la instrucción
```
>help("vector")
```
aparece una ventana de ayuda con información sobre el comando vector.

Operaciones aritméticas

Directamente se pueden realizar operaciones aritméticas después del símbolo >:
+(suma), - (diferencia), ***(producto), /(cociente), ^(potencia):

```
> 20+34
[1] 54
> 30-5
[1] 25
> 3-6
[1] -3
```

```
> 3*4
[1] 12
> 50/5
[1] 10
> 3^2
[1] 9
```

También se puede calcular el resto de una división mediante **%%**. Por ejemplo el resto de dividir 31 entre 7, o bien 31(mod 7)

```
> 31%%7
[1] 3
```

También, la parte entera de una fracción con **%/%**

```
> 31%/%7
[1] 4
```

Asignación de valores

R es un lenguaje orientado a objetos. Comenzaremos viendo cómo se realiza una asignación con los símbolos < y -.

x<- 3

o bien,

3->x

Código R

```
> x<-3
> x
[1] 3
```

También se puede usar el signo = pero no es lo habitual.

Código R

```
> x=4
> x
[1] 4
```

En la salida anterior [1] indica que el número que le sigue es el primer elemento de x (en este caso es evidente puesto que x es un escalar, luego lo veremos con un vector).

Si queremos crear un vector con componentes (3,6,8,9), se hará de la siguiente forma

x<-c(3,6,8,9)

Código R

```
> x<-c(3,6,8,9)
> x
[1] 3 6 8 9
```

Si el vector ocupara más de una línea, cada nueva línea empezaría con [n], indicando n el lugar que ocupa dentro del vector la coordenada que sigue.

Vamos a usar **n:m** para incluir en un vector los valores n, n+1, n+2, …., m, y con ello mostraremos lo anteriormente comentado

```
> x<-1:100
> x
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18
[19] 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36
[37] 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54
[55] 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72
[73] 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90
[91] 91 92 93 94 95 96 97 98 99 100
```

Se puede cambiar la amplitud de lo que aparece en pantalla con la siguiente función

```
> options(width=40)
> 1:100
 [1]  1  2  3  4  5  6  7  8
 [9]  9 10 11 12 13 14 15 16
[17] 17 18 19 20 21 22 23 24
[25] 25 26 27 28 29 30 31 32
[33] 33 34 35 36 37 38 39 40
[41] 41 42 43 44 45 46 47 48
[49] 49 50 51 52 53 54 55 56
[57] 57 58 59 60 61 62 63 64
[65] 65 66 67 68 69 70 71 72
[73] 73 74 75 76 77 78 79 80
[81] 81 82 83 84 85 86 87 88
[89] 89 90 91 92 93 94 95 96
[97] 97 98 99 100
> options(width=60)
> 1:100
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13
[14] 14 15 16 17 18 19 20 21 22 23 24 25 26
[27] 27 28 29 30 31 32 33 34 35 36 37 38 39
[40] 40 41 42 43 44 45 46 47 48 49 50 51 52
[53] 53 54 55 56 57 58 59 60 61 62 63 64 65
[66] 66 67 68 69 70 71 72 73 74 75 76 77 78
[79] 79 80 81 82 83 84 85 86 87 88 89 90 91
[92] 92 93 94 95 96 97 98 99 100
```

Para generar los valores de un vector también se pueden usar las funciones **seq()** y **rep()** Con la instrucción **seq(a,b,by=r)** o, simplemente **seq(a,b,r)**, se genera una lista de números que empieza en **a** y termina en **b**, de la forma a, a+r, a+2r, …

```
> seq(1,20)
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
> seq(1,20, by=2)
 [1]  1  3  5  7  9 11 13 15 17 19
> seq(20,1)
 [1] 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1
> seq(20,1, by=-2)
```

```
 [1] 20 18 16 14 12 10  8  6  4  2
```

Con la instrucción **seq(a,b,length=r)** se generan **r** números entre **a** y **b**, igualmente espaciados.

```
> seq(4,10, length=8)
[1]  4.000000  4.857143  5.714286  6.571429  7.428571  8.285714  9.142857
[8] 10.000000
```

Con la instrucción **rep(x,r)** se genera una lista de **r** valores todos iguales a **x**. Se puede encadenar con la instrucción seq(), aplicarla a vectores (en tal caso, si el número de repeticiones de cada elemento del vector es distinto, se debe incluir un vector con los valores de tales repeticiones) y usar la opción **each** .

```
> rep(3,12)
 [1] 3 3 3 3 3 3 3 3 3 3 3 3
> rep(seq(2,20,by=2),2)
 [1]  2  4  6  8 10 12 14 16 18 20  2  4  6  8 10 12 14 16 18 20
> rep(c(1,4),c(3,2))
[1] 1 1 1 4 4
> rep(c(1,4), each=3)
[1] 1 1 1 4 4 4
```

Vectores

Ya se ha visto como asignar valores a un vector

```
> x<-c(3,6,8,9)
```

Par ver el contenido del vector basta escribir su nombre

```
> x
[1] 3 6 8 9
```

También se ha visto como usar el símbolo **:** se puede usar para crear secuencias crecientes (o decrecientes) de valores. Por ejemplo

```
> Numerosde5a20<-5:20
> Numerosde5a20
 [1]  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
> Numerosde20a5<-20:5
> Numerosde20a5
 [1] 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5
```

Se pueden concatenar vectores:

```
> x<-c(1,2)
> y<-c(3,4)
> z<-c(x,y)
> x
[1] 1 2
> y
```

```
[1] 3 4
> z
[1] 1 2 3 4
> v<-c(z,5:10)
> v
 [1]  1  2  3  4  5  6  7  8  9 10
```

Podemos cambiar de línea en mitad de una instrucción, se continúa en la línea siguiente con el símbolo + (que indica que la línea anterior está incompleta)

```
> x<-c(1,3,5,6,5,4,3,4,5,6,6,7,7,8,9,9,
+ 4,5,6)
> x
 [1] 1 3 5 6 5 4 3 4 5 6 6 7 7 8 9 9 4 5 6
```

Para extraer elementos de un vector se usa **[ ]**

```
> x[3]
[1] 5
> x[c(1,2)]
[1] 1 3
> x[3:6]
[1] 5 6 5 4
```

Considerar un índice negativo significa ignorar el elemento o elementos correspondientes

```
> x<-11:20
> x
 [1] 11 12 13 14 15 16 17 18 19 20
> x[-3]
[1] 11 12 14 15 16 17 18 19 20
> x[c(-1,-2)]
[1] 13 14 15 16 17 18 19 20
> x[-(3:6)]
[1] 11 12 17 18 19 20
```

Operaciones con vectores

Multiplicación de un vector por un número

```
> x<-c(1,2,3,4,5)
> 2*x
[1]  2  4  6  8 10
```

Suma y resta de un vector y un número

```
> 2+x
[1] 3 4 5 6 7

> 2-x
[1]  1  0 -1 -2 -3
```

Suma de vectores

```
> y<-c(6,7,8,9,10)
> x+y
[1]  7  9 11 13 15
```

Notemos que las operaciones con vectores se hacen elemento a elemento

Potencia de un vector (eleva al cuadrado cada elemento del vector)

```
> x^2
[1]  1  4  9 16 25
```

Raiz cuadrada (hace la raíz cuadrada de cada elemento del vector)

```
> sqrt(x)
[1] 1.000000 1.414214 1.732051 2.000000 2.236068
```

Producto de dos vectores (elemento a elemento)
¡Cuidado, con el producto!

```
> x<-c(1,2,3,4,5)
> y<-c(6,7,8,9,10)
> x*y
[1]  6 14 24 36 50
```

¡Y la potencia! (Eleva cada elemento del primer vector al correspondiente del segundo)

```
> x^y
[1]     1    128    6561  262144 9765625
```

Cuando los vectores tienen dimensiones diferentes, el de menor dimensión se extiende repitiendo los valores desde el principio, aunque da un mensaje de aviso

```
> x<-c(1,2,3,4,5)
> y<-c(1,2,3,4,5,6)
> x+y
[1]  2  4  6  8 10  7
Mensajes de aviso perdidos
In x + y :
  longitud de objeto mayor no es múltiplo de la longitud de uno menor
> x^y
[1]    1    4   27  256 3125    1
Mensajes de aviso perdidos
In x^y :
  longitud de objeto mayor no es múltiplo de la longitud de uno menor
```

Funciones con vectores

Asignar nombres a los elementos de un vector con la función **names()**

```
> x<-c(3.141592, 2.718281)
> names(x)<-c("pi","e")
```

```
> x
      pi       e
3.141592 2.718281
> x["pi"]
      pi
3.141592
```

La función **length()** devuelve la dimensión de un vector

```
> x<-c(1,2,3,4,5)
> length(x)
[1] 5
```

Las funciones **sum()** y **cumsum()** proporcionan la suma y sumas acumuladas de las componentes de un vector

```
> sum(x)
[1] 15
> cumsum(x)
[1]  1  3  6 10 15
```

Las funciones **max()** y **min()** proporcionan el valor máximo y mínimo de las componentes de un vector

```
> max(x)
[1] 5
> min(x)
[1] 1
```

Las funciones **mean(), median(), var()** y **sd()** proporcionan la media, mediana, cuasivarianza y cuasidesviación típica, respectivamente, de las componentes de un vector

```
> mean(x)
[1] 3
> median(x)
[1] 3
> var(x)
[1] 2.5
> sd(x)
[1] 1.581139
```

Las funciones **prod()** y **cumprod()** proporcionan el producto y productos acumulados de las componentes de un vector

```
> prod(x)
[1] 120
> cumprod(x)
[1]   1   2   6  24 120
```

La función **quantile()** proporciona los cuartiles

```
> quantile(x)
  0%  25%  50%  75% 100%
   1    2    3    4    5
```

La función **sort()** ordena en orden creciente de las componentes de un vector

```
> x<-c(2,6,3,7,9,1,4,7)
> sort(x)
[1] 1 2 3 4 6 7 7 9
```

La función **rev()** coloca las componentes de un vector en orden inverso a como han sido introducidas

```
> rev(x)
[1] 7 4 1 9 7 3 6 2
```

¿Cómo se ordenarían en orden decreciente las componentes de un vector?

Las funciones **cov()** y **cor()** proporcionan la covarianza y coeficiente de correlación entre dos vectores

```
> x<-c(1,2,3,4,5)
> y<-c(2,1,5,4,3)
> cov(x,y)
[1] 1.25
> cor(x,y)
[1] 0.5
```

Vectores de carácter

Tanto los escalares como los vectores pueden contener cadenas de caracteres. Todos los elementos de un vector deben ser del mismo tipo

```
> colores<-c("amarillo", "rojo", "verde")
> colores
[1] "amarillo" "rojo"     "verde"
> mas.colores<-c(colores, "azul", "negro")
> mas.colores
[1] "amarillo" "rojo"     "verde"    "azul"     "negro"
```

Si se intenta incluir un número después de caracteres lo interpreta como una cadena de caracteres.

```
> otros.colores<-c("naranja", "rosa", 1)
> otros.colores
[1] "naranja" "rosa"    "1"
```

Variables y operaciones lógicas

Supongamos que tenemos el vector x<-1:20, podemos ver qué elementos son iguales a un valor con el operador = =

```
> x<-11:20
> x
 [1] 11 12 13 14 15 16 17 18 19 20
> x==15
 [1] FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE
```

La instrucción anterior devuelve un vector de valores lógicos (o booleanos). FALSE indica que la condición chequeada (en este caso, ser igual a 15) no se cumple mientras que TRUE indica que sí se cumple.

De manera análoga se pueden hacer comparaciones con <, >, <=, >=, ¡= (para distinto)

```
> x<-11:20
> x
 [1] 11 12 13 14 15 16 17 18 19 20
> x==15
 [1] FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE
> x<15
 [1]  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE
> x>15
 [1] FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
> x<=15
 [1]  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE
> x>=15
 [1] FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
> x!=15
 [1]  TRUE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
```

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

```
> sum(x==15)
[1] 1
> sum(x<15)
[1] 4
> sum(x>15)
[1] 5
> sum(x<=15)
[1] 5
> sum(x>=15)
[1] 6
> sum(x!=15)
[1] 9
```

Ejemplo

Vamos a calcular la media y mediana de un vector y el número de valores que están por debajo de la media y de la mediana

```
> x<-c(1,5,7,9,3,5,6,2,4,7,5,6,9,8,6,2,6,1,4)
> mean(x)
[1] 5.052632
> median(x)
[1] 5
> sum(x<mean(x))
[1] 10
> sum(x<median(x))
[1] 7
> length(x)
[1] 19
> x==median(x)
 [1] FALSE  TRUE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE  TRUE
FALSE
[13] FALSE FALSE FALSE FALSE FALSE FALSE FALSE
> sum(x==median(x))
[1] 3
```

Veamos el efecto de 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
> z<-3:7
> z[c(TRUE,FALSE,TRUE,FALSE,TRUE)]
[1] 3 5 7
```

Las instrucción **x[x<r]** devuelve los valores del vector x que verifican la condición impuesta x<r.

En el siguiente ejemplo se usa la función **runif()** que genera valores de una distribución uniforme y que se comentará con posterioridad.

```
> x<-runif(10)
> x
 [1] 0.1976741 0.3802387 0.5560528 0.8759437 0.2752229 0.7574220 0.6449525
 [8] 0.5145545 0.2543729 0.2372419
> x[x<0.2]
[1] 0.1976741
> x[x<0.4]
[1] 0.1976741 0.3802387 0.2752229 0.2543729 0.2372419
```

Podemos usar operadores lógicos dentro de los corchetes con condiciones compuestas, **|** es el operador lógico “o” y **&** es “y”

```
> x
 [1] 0.1976741 0.3802387 0.5560528 0.8759437 0.2752229 0.7574220 0.6449525
 [8] 0.5145545 0.2543729 0.2372419
> x[(x<0.2)|(x>0.8)]
[1] 0.1976741 0.8759437
> sum((x<0.2)|(x>0.8))
```

```
[1] 2
> sum(x[(x<0.2)|(x>0.8)])
[1] 1.073618
> x[(x>0.2)&(x<0.8)]
[1] 0.3802387 0.5560528 0.2752229 0.7574220 0.6449525 0.5145545 0.2543729
[8] 0.2372419
> sum((x>0.2)&(x<0.8))
[1] 8
> sum(x[(x>0.2)&(x<0.8)])
[1] 3.620058
```

La función **which()** detecta las posiciones en el vector de los elementos que verifican una condición

```
> x
 [1] 0.1976741 0.3802387 0.5560528 0.8759437 0.2752229 0.7574220 0.6449525
 [8] 0.5145545 0.2543729 0.2372419
> which((x>0.2)&(x<0.6))
[1]  2  3  5  8  9 10
```

Matrices

Operaciones con matrices

Data frames

Introducción de datos

Crear funciones

La forma de crear una función es con la siguiente instrucción:

Nombre_de_la_función<- function(x, y, …) {expresión de la función}

siendo x, y, … los argumentos de la función. Luego la función se ejecuta como **Nombre_de_la_función(x,y,…)**

Por ejemplo

```
> desvia<-function(x){x-mean(x)}
> desvia(c(1,3,5,7))
[1] -3 -1  1  3
```

Notemos que

```
> mean(c(1,3,5,7))
[1] 4
```

Creemos una función que calcule la media de un vector

```
> media<-function(x){sum(x)/length(x)}
> y<-1:100
> media(y)
```

```
[1] 50.5
> mean(y)
[1] 50.5
```

Nota: Es útil ver los objetos que tenemos guardados en cada momento para no usar nombres que contengan valores que queramos conservar. Se hace con la función **objects()**

Gráficos en R

Diagrama de barras

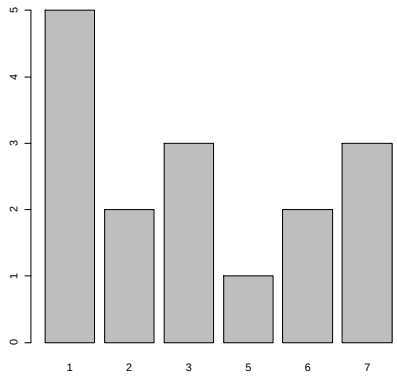
Sea

```
> x<-c(1,1,1,1,1,2,2,3,3,3,5,6,6,7,7,7)
```

la función **table()** tabula los datos en x y da lugar a

```
> table(x)
x
1 2 3 5 6 7
5 2 3 1 2 3
```

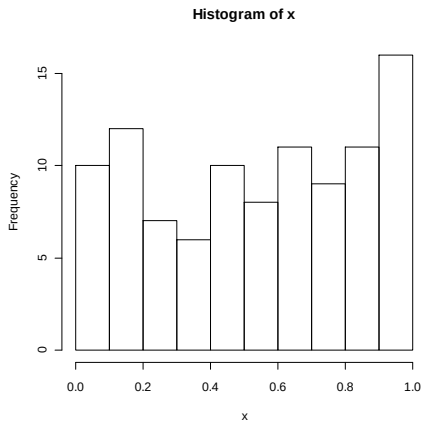
La instrucción **barplot(table(x))** muestra el diagrama de barras asociado



Histograma

Se realiza con la función **hist().**

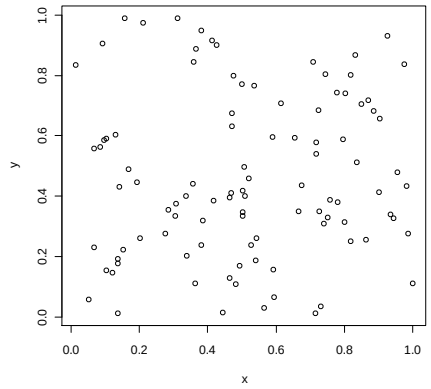
```
> x<-runif(100)
> hist(x)
```



Scatterplots

Se realizan con la instrucción **plot(x,y)**

```
> x<-runif(100)
> y<-runif(100)
> plot(x,y)
```

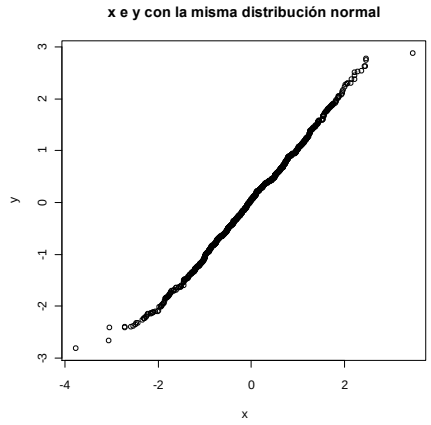


QQ-plots

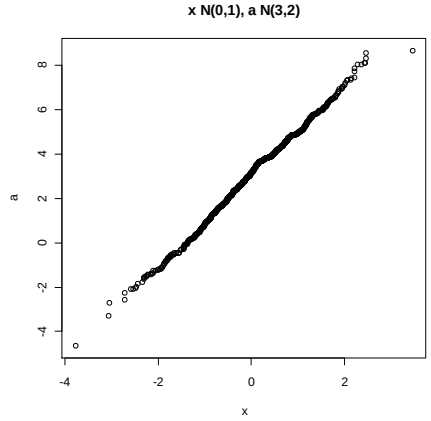
Instrucción **qqplot(x,y).**

En los siguientes ejemplos se utilizan algunas funciones como **rnorm()**, **rt()** que generan valores aleatorios de distribuciones normales y t de Student y que se comentan con posterioridad.

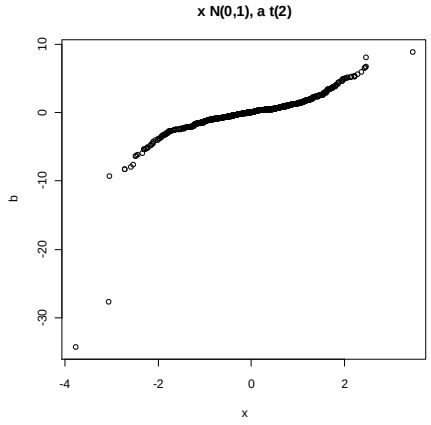
```
> x<-rnorm(1000)
> y<-rnorm(1000)
> qqplot(x,y,main="x e y con la misma distribución normal")
```



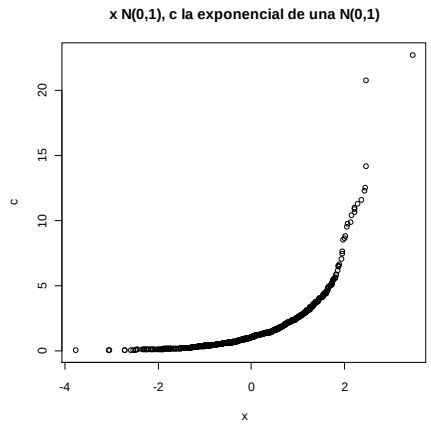

```
> a<-rnorm(1000, mean=3, sd=2)
> qqplot(x,a,main="x N(0,1), a N(3,2)")
```




```
> b<-rt(1000, df=2)
> qqplot(x,b,main="x N(0,1), a t(2)")
```




```
> c<-exp(rnorm(x))
> qqplot(x,c,main="x N(0,1), c la exponencial de una N(0,1)")
```



Otras instrucciones de R que necesitaremos

El proceso de realización de simulaciones es a menudo muy repetitivo, por lo que usaremos la sentencia **for()** que permite repetir una cierta operación un número de veces.

Su sintaxis es

for (*Nombre_del indice* **in** *vector***){comandos}**

de forma que al ejecutar dicha instrucción crea una variable llamada *Nombre_ del_indice* igual a cada uno de los elementos del vector, en secuencia. Para cada valor se ejecuta todos los comandos incluidos entre llaves, considerándolos como un único comando. Si sólo hay que ejecutar un solo comando las llaves no son necesarias, pero es conveniente incluirlas para una mayor claridad.

Cuando se construye un vector a partir de comandos usando for() (o cualquier otra función de programación) es necesario definir previamente dicho vector. La instrucción **Nombre<-numeric(length=r)** o, simplemente, **Nombre<-numeric(r**) crea un objeto de tipo numérico de longitud r.

```
> x<-numeric(10)
> x
 [1] 0 0 0 0 0 0 0 0 0 0
```

Así, si queremos obtener los primeros 12 números de Fibonacci, se procede de la siguiente forma

```
> Fibonacci_12<-numeric(12)
> Fibonacci_12[1]<-Fibonacci_12[2]<-1
> for(i in 3:12){Fibonacci_12[i]<-Fibonacci_12[i-2]+Fibonacci_12[i-1]}
> Fibonacci_12
 [1]   1   1   2   3   5   8  13  21  34  55  89 144
```

GENERACIÓN DE NÚMEROS PSEUDOALEATORIOS

Método congruencial multiplicativo

Ejemplo

Generar 50 números pseudoaleatorios a partir del generador congruencial multiplicativo

$x_n = 171\ x_{n-1} \pmod{30269}$
$u_n = x_n/30269$

con semilla 27218.

Código R

```
> random.number<-numeric(50)
> random.seed<-27218
> for (j in 1:50)
+             {random.seed<-(171*random.seed)%%30269
+             random.number[j]<-random.seed/30269
+ }
> random.number
 [1] 0.76385080 0.61848756 0.76137302 0.19478675 0.30853348 0.75922561
 [7] 0.82757937 0.51607255 0.24840596 0.47741914 0.63867323 0.21312234
[13] 0.44391952 0.91023820 0.65073177 0.27513297 0.04773861 0.16330239
[19] 0.92470845 0.12514454 0.39971588 0.35141564 0.09207440 0.74472232
[25] 0.34751726 0.42545178 0.75225478 0.63556774 0.68208398 0.63636063
[31] 0.81766824 0.82126929 0.43704780 0.73517460 0.71485678 0.24051009
[37] 0.12722587 0.75562457 0.21180085 0.21794575 0.26872378 0.95176583
[43] 0.75195745 0.58472364 0.98774324 0.90409330 0.59995375 0.59209092
[49] 0.24754700 0.33053619
```

O bien

```
> x<-numeric(50)
> semilla<-27218
> x[1]=(171*semilla)%%30269
> for(i in 2:50){x[i]=(171*x[i-1])%%30269}
> NumerosAleatorios<-x/30269
> NumerosAleatorios
 [1] 0.76385080 0.61848756 0.76137302 0.19478675 0.30853348 0.75922561
 [7] 0.82757937 0.51607255 0.24840596 0.47741914 0.63867323 0.21312234
[13] 0.44391952 0.91023820 0.65073177 0.27513297 0.04773861 0.16330239
```

```
[19] 0.92470845 0.12514454 0.39971588 0.35141564 0.09207440 0.74472232
[25] 0.34751726 0.42545178 0.75225478 0.63556774 0.68208398 0.63636063
[31] 0.81766824 0.82126929 0.43704780 0.73517460 0.71485678 0.24051009
[37] 0.12722587 0.75562457 0.21180085 0.21794575 0.26872378 0.95176583
[43] 0.75195745 0.58472364 0.98774324 0.90409330 0.59995375 0.59209092
[49] 0.24754700 0.33053619
```

Ejemplo

Generar 50 números pseudoaleatorios a partir del generador congruencial

$x_n = 69069\ x_{n-1} \pmod{2^{37}}$
$u_n = x_n / (2^{37})$

con semilla 1

Código R

```
> random.number<-numeric(50)
> random.seed<-1
> for (j in 1:50)
+             {random.seed<-(69069*random.seed)%%(2^(37))
+             random.number[j]<-random.seed/(2^(37))
+ }
> random.number
 [1] 5.025431e-07 3.471015e-02 3.953856e-01 8.880281e-01 2.100906e-01
 [6] 7.495926e-01 6.092866e-01 8.152791e-01 5.126953e-01 3.532371e-01
[11] 7.329069e-01 1.447004e-01 3.117959e-01 4.340465e-01 1.577139e-01
[16] 1.437959e-01 8.416260e-01 2.675922e-01 3.230826e-01 9.929959e-01
[21] 2.359233e-01 9.839796e-01 4.857613e-01 4.524707e-02 1.697248e-01
[26] 7.234906e-01 7.689292e-01 1.690644e-01 1.093887e-01 3.653836e-01
[31] 6.812334e-01 1.091246e-01 1.255209e-01 6.032417e-01 2.994703e-01
[36] 1.168403e-01 4.530229e-02 9.838180e-01 3.264416e-01 9.948997e-01
[41] 7.263835e-01 5.790029e-01 1.487847e-01 4.135195e-01 3.810732e-01
[46] 3.428486e-01 2.068159e-01 5.641017e-01 9.430070e-01 5.520908e-01
```

Una operación similar (usando otra fórmula y con un ciclo mucho más largo) es la que usa internamente R para producir números pseudoaleatorios de forma automática con la función **runif()** del grupo de funciones asociadas con la distribución uniforme, dentro del paquete **stats.** En este caso la semilla se selecciona internamente.

Distribución uniforme

Descripción

Las siguientes funciones proporcionan información sobre la distribución uniforme en el intervalo comprendido entre 'min' y 'max':

'dunif' proporciona la función de densidad
'punif' proporciona la función de distribución

‘qunif’ proporciona la función de cuantiles
‘runif’ genera valores aleatorios.

Uso:

dunif(x, min=0, max=1, log = FALSE)
punif(q, min=0, max=1, lower.tail = TRUE, log.p = FALSE)
qunif(p, min=0, max=1, lower.tail = TRUE, log.p = FALSE)
runif(n, min=0, max=1)

Argumentos:

x,q: vector de cuantiles.

p: vector de probabilidades.

n: número de observaciones. Si no se especifica se toma igual a 1

min,max: extremos inferior y superior del intervalo que determina la distribución. Deben ser finitos. Si no se especifican se toman los valores por defecto 0 y 1. Para el caso min = max = u, el caso degenerado X = u se considera, aunque como no tiene función de densidad, la función ‘dunif’ devuelve ‘NaN’ (condición de error).

log, log.p: son valores lógicos; si son TRUE, las probabilidades p se dan como probabilities log(p).

lower.tail: es un valor lógico; si es TRUE (por defecto), las probabilidades son $P[X \leq x]$, en otro caso $P[X > x]$.

La forma de uso más habitual para generar números pseudoaleatorios de una distribución U(a,b) es

runif(n, min=a, max=b)

y para una U(0,1)

runif(n)

Ejemplo
Generar 10 números aleatorios en el intervalo (0,1) y 15 en el intervalo (-1,2)

```
> runif(10)
 [1] 0.31264669 0.59918637 0.37935813 0.97517055 0.79116964 0.47277362
 [7] 0.02291407 0.48761985 0.52772211 0.91347195
> runif(15,min=-1,max=2)
 [1]  1.96646178 -0.53018219  0.99929876  1.64215831  0.79163840
0.73346732
```

```
 [7] -0.68915028  1.86967652 -0.04384054  1.32471288 -0.30430071 -
0.68072269
[13]  0.81967883  1.70135362  0.80832656
```

Si se quiere ejecutar la función anterior, pero partiendo de una semilla concreta (para garantizar el mismo resultado en cualquier ejecución), se usará la función **set.seed()** antes de la anterior

```
> runif(5)
[1] 0.3126467 0.5991864 0.3793581 0.9751705 0.7911696
> runif(5)
[1] 0.47277362 0.02291407 0.48761985 0.52772211 0.91347195
> set.seed(32789)
> runif(5)
[1] 0.3575211 0.3537589 0.2672321 0.9969302 0.1317401
> set.seed(32789)
> runif(5)
[1] 0.3575211 0.3537589 0.2672321 0.9969302 0.1317401
```

Ejercicios propuestos

1. Genera 20 números pseudoaleatorios usando

$x_n = 172\ x_{n-1} \pmod{30307}$

con semilla inicial $x_0 = 17218$.

```
> random.number<-numeric(20)
> random.seed<-17218
> for (j in 1:20)
+             {random.seed<-(172*random.seed)%%30307
+             random.number[j]<-random.seed/30307
+ }
> random.number
 [1] 0.71656713 0.24954631 0.92196522 0.57801828 0.41914409 0.09278385
 [7] 0.95882139 0.91727984 0.77213185 0.80667833 0.74867192 0.77157092
[13] 0.71019896 0.15422180 0.52614907 0.49764081 0.59421916 0.20569505
[19] 0.37954928 0.28247600
```

Otra forma posible

```
> x0<-17218
> x<-numeric(20)
> x[1]<-(172*x0)%%30307
> for(i in 2:20){ x[i] <-(x[i-1]*172)%%30307}
> x/30307
 [1] 0.71656713 0.24954631 0.92196522 0.57801828 0.41914409 0.09278385
```

```
 [7] 0.95882139 0.91727984 0.77213185 0.80667833 0.74867192 0.77157092
[13] 0.71019896 0.15422180 0.52614907 0.49764081 0.59421916 0.20569505
[19] 0.37954928 0.28247600
```

2. Genera 20 números pseudoaleatorios usando el generador congruencial multiplicativo con b = 171 y m = 32767 con semilla inicial 2018.

```
> random.number<-numeric(20)
> random.seed<-2018
> for (j in 1:20)
+             {random.seed<-(171*random.seed)%%32767
+             random.number[j]<-random.seed/32767
+ }
> random.number
 [1] 0.53126621 0.84652242 0.75533311 0.16196173 0.69545579 0.92294076
 [7] 0.82287057 0.71086764 0.55836665 0.48069704 0.19919431 0.06222724
[13] 0.64085818 0.58674886 0.33405560 0.12350841 0.11993774 0.50935392
[19] 0.09952086 0.01806696
```

3. Usa la función runif() (con set.seed(32078)) para generar 10 números pseudoaleatorios de

a) **una distribución uniforme (0,1)**
b) **una distribución uniforme (3,7)**
c) **una distribución uniforme (-2,2)**

```
> set.seed(32078)
> runif(10)
 [1] 0.2564626 0.4988177 0.5266549 0.6269816 0.8052754 0.1843452 0.5102327
 [8] 0.3683905 0.1708176 0.7432888

> set.seed(32078)
> runif(10, min=3, max=7)
 [1] 4.025850 4.995271 5.106620 5.507927 6.221102 3.737381 5.040931
4.473562
 [9] 3.683270 5.973155

> set.seed(32078)
> runif(10, min=-2, max=2)
 [1] -0.974149697 -0.004729333  0.106619657  0.507926506  1.221101642
 [6] -1.262619189  0.040930690 -0.526437979 -1.316729628  0.973155177
```

4. Genera 1000 valores pseudoaleatorios usando la función runif() (con set.seed(19908)) y asígnalos a un vector llamado U.

a) **Calcular la media, varianza y desviación típica de los valores de U.**
b) **Compara los resultados con los verdaderos valores de la media, varianza y desviación típica de una U(0,1).**
c) **Calcula la proporción de valores de U que son menores que 0.6 y compárala con la probabilidad de que una variable U(0,1) sea menor que 0.6.**
d) **Estimar el valor esperado de 1/(U+1)**
e) **Construir un histograma de los valores de u y de 1/(U+1)**

```
> set.seed(19908)
> U<-runif(1000)
> data<-c(mean(U),var(U),sqrt(var(U)))
> 0.5 #media teórica
[1] 0.5
> 1/12 #varianza teórica
[1] 0.08333333
> sqrt(1/12) #desviación típica teórica
[1] 0.2886751
> data #media, varianza y desviación típica de los datos generados
[1] 0.49605698 0.08148008 0.28544716
```

La proporción de valores menores que 0.6 se calcula como

```
> sum(U<0.6)/1000
[1] 0.61
```

o equivalentemente como

```
>length(U[U<0.6])/length(U)
[1]0.61
```

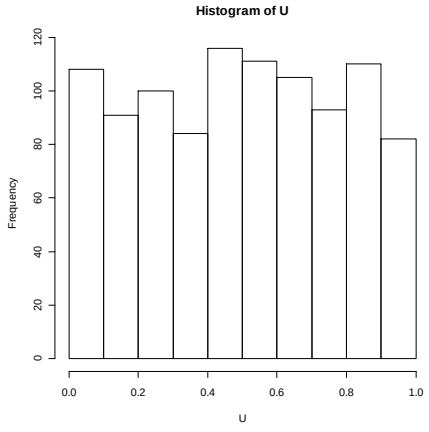
La probabilidad teórica se calcula como

```
> punif(0.6)
[1] 0.6
```

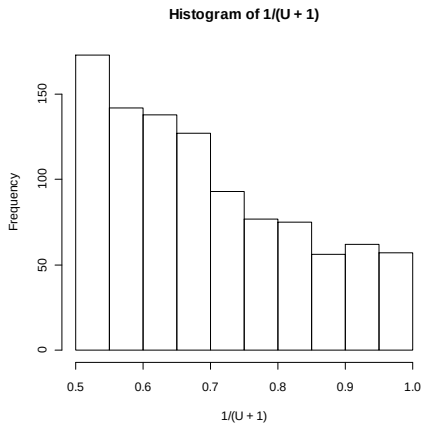
Estimación del valor esperado de 1/(U+1)

```
> mean(1/(U+1))
[1] 0.6946063
```

```
> hist(U)
```




```
> hist(1/(U+1))
```



5. Simula 10000 observaciones independientes de una variable aleatoria distribuida uniformemente en el intervalo [3.7, 5.8].

a) **Calcular la media, varianza y desviación típica de los valores simulados (Estima la media, varianza y desviación típica de tal variable aleatoria uniforme) y compararlos con los verdaderos valores de la distribución.**
b) **Estima la probabilidad de que tal variable aleatoria sea mayor o igual que 4 (Calcula la proporción de valores que son mayores o iguales a 4) y compárala con el verdadero valor.**

Solución

(a)

```
>r<-runif(10000,3.7,5.8)
>mean(r)
[1] 4.756022
>var(r)
[1] 0.3664445
>sd(r)
[1] 0.6053466
> (3.7+5.8)/2  #media teórica
[1] 4.75
> (5.8-3.7)^2/12   #varianza teórica
[1] 0.3675
> sqrt((5.8-3.7)^2/12)   #desviación típica teórica
[1] 0.6062178
```

(b)

```
 >length(r[r>4])/length(r)
[1] 0.8584

> punif(4, min=3.7, max=5.8, lower.tail = FALSE)
[1] 0.8571429
```

6. Simula 10000 valores de una variable aleatoria U_1 con distribución U(0,1) y otro conjunto de valores de una variable aleatoria U_2 con distribución U(0,1). Asignar esos valores a vectores U1 y U2, respectivamente. Dado que los valores en U1 y U2

son aproximadamente independientes, podemos considerar a U_1 y U_2 variables aleatorias independientes U(0,1).

a) **Estimar $E[U_1+U_2]$, compararla con el verdadero valor y compararla con una estimación de $E[U_1]+E[U_2]$,**
b) **Estimar $Var[U_1+U_2]$ y $Var[U_1]+Var[U_2]$. ¿Son iguales? ¿Serían los verdaderos valores iguales?**
c) **Estimar $P(U_1+U_2 \leq 1.5)$.**
d) **Estimar $P(\sqrt{U_1}+\sqrt{U_2} \leq 1.5)$.**

```
> U1<-runif(10000,0,1)
> U2<-runif(10000,0,1)
> U=U1+U2
> mean(U)
[1] 1.001534
> mean(U1)+mean(U2)
[1] 1.001534
> var(U)
[1] 0.1662075
> var(U1)+var(U2)
[1] 0.1661485
> length(U[U<=1.5])/length(U)
[1] 0.8769
> V<-sqrt(U1)+sqrt(U2)
> length(V[V<=1.5])/length(V)
[1] 0.6537
```

7. Supongamos que U_1, U_2 y U_3 son variables aleatorias independientes con distribución U(0,1). Usa simulación para estimar las siguientes cantidades:

a) **$E[U_1+U_2+U_3]$**
b) **$Var[U_1+U_2+U_3]$ y $Var[U_1]+Var[U_2]+Var[U_3]$**
c) **$E[\sqrt{U_1+U_2+U_3}]$**
d) **$P(\sqrt{U_1}+\sqrt{U_2}+\sqrt{U_3} \geq 0.8)$.**

```
>U1<-runif(10000)
>U2<-runif(10000)
>U3<-runif(10000)
>U<-U1+U2+U3
(a)
>mean(U)
[1] 1.50653
(b)
>var(U)
[1] 0.2548007
>var(U1)+var(U2)+var(U3)
[1] 0.251535
(c)
>mean(sqrt(U))
[1] 1.207929
(d)
 >V<-sqrt(U1)+sqrt(U2)+sqrt(U3)
>length(V[V>=.8])/length(V)
[1] 0.9965
```

FUNCIÓN SAMPLE

La función **sample()** permite tomar una muestra aleatoria simple a partir de un vector de valores con o sin reemplazamiento. Se usa como

sample(x, size, replace=FALSE, prob=NULL)

donde *x* es un vector de donde se quieren elegir los elementos o un entero positivo n (en este caso se interpreta como el vector generado por 1:n), *size* es un entero positivo que indica el número de elementos que se quieren elegir, *replace=FALSE* indica que el muestreo se hace sin reemplazamiento, mientras que *replace=TRUE* indica con reemplazamiento. Por último en *prob* se puede incluir un vector de probabilidades en el que cada componente será la probabilidad con la que se elegirá la correspondiente componente del vector que va a ser muestreado.

Por ejemplo
sample(c(3,5,7), size=2, replace=FALSE)
conduce a un vector de dos valores tomados (sin reemplazamiento) del conjunto {3,5,7}.

Usar la función sample() para generar 50 números pseudoaleatorios del 1 al 100,
a) muestreados sin reemplazamiento
b) muestreados con reemplazamiento.

a)

```
> sample(1:100, size=50, replace=FALSE)
[1] 48 20 66  8  2 25 18 19 15 60 74 28 80 38 26 77 84 10  4 67 92 52 95 17 55
[26] 81 21 13 43 53 16 90 42 59 24 91 32 39 69 37 79 83 68 22 73 45 96 50  1 61
```

Se podría haber escrito simplemente sample(100,50,FALSE)

b)

```
> sample(1:100, size=50, replace=TRUE)
 [1]  47  30  18  61  10  19  25  98  75  94  83   1  97  73  39  86  35  72  31
[20]  16  47  25  50  15  17  14  14  79   7  98  36  25 100  71  60   4  54  34
[39]  23  67  24  93  26  91  32  11  59  65   3  97
```

Simula el lanzamiento de un dado

```
> sample(1:6,1)
[1] 6
> sample(1:6,1)
[1] 3
> sample(1:6,1)
[1] 4
> sample(1:6,1)
[1] 5
```

Simula el lanzamiento de cuatro dados o de un mismo dado cuatro veces

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

```
[1] 4 5 4 2
```

Simula la distribución de la suma de los números que salen al lanzar cuatro dados

Para ello usaremos la función **sapply** de la siguiente forma

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

la cual aplica a un vector de tamaño 10000 una función sin nombre generando a su vez un vector de tamaño 10000. La función considerada obtiene muestras de tamaño y, a continuación, suma los elementos de la muestra.
Se podría haber hecho con un **for** pero este procedimiento es más rápido.

Para garantizar que los resultados son los mismos usemos una semilla común, por ejemplo 111.

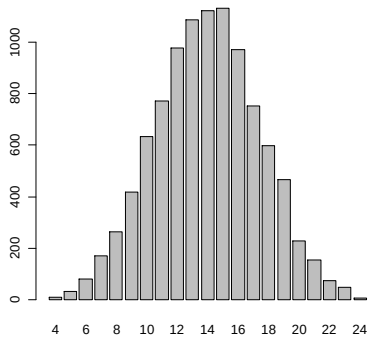
```
> set.seed(111)
> t<-sapply(1:10000, function(x){sum(sample(1:6,4,rep=T))})
```

A continuación, tabulamos los resultados

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

y podemos representar los resultados con un diagrama de barras

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



Supongamos una urna con 3 bolas blancas y 7 negras, simular la extracción de una bola (asignar, por ejemplo, el 1 a bola blanca y 0 a negra)

```
> sample(c(1,0), 1, prob=c(0.3,0.7))
[1] 1
> sample(c(1,0), 1, prob=c(0.3,0.7))
[1] 0
> sample(c(1,0), 1, prob=c(0.3,0.7))
[1] 1
> sample(c(1,0), 1, prob=c(0.3,0.7))
[1] 0
> sample(c(1,0), 1, prob=c(0.3,0.7))
[1] 0
> sample(c(1,0), 1, prob=c(0.3,0.7))
```

```
[1] 0
```

Si queremos simular 8 extracciones con reemplazamiento

```
> sample(c(1,0), 8, rep=T, prob=c(0.3,0.7))
[1] 1 0 0 0 0 0 1 0
```

Si sólo nos interesara el número de bolas blancas que salen, se puede hacer la suma, pero esto lo haremos mejor usando la distribución binomial.

DISTRIBUCIÓN UNIFORME DISCRETA

Usaremos para su simulación la función **sample()** con el siguiente código general

x<-c(x_1, x_2, …, x_n)	Crea un vector con los valores de la distribución uniforme discreta
mean(x)	Media teórica de la distribución
var(x)	Varianza teórica de la distribución
muestra<-sample(x,m,replace=TRUE)	Genera m valores aleatorios de la distribución uniforme discreta con valores x_1, x_2, …, x_n
mean(muestra)	Media muestral
var(muestra)	Varianza muestral

DISTRIBUCIÓN BINOMIAL

Descripción:

Las siguientes funciones proporcionan información sobre la distribución binomial de parámetros size (número de veces que se repite el experimento de Bernoulli) y p (probabilidad de éxito):

‘dbinom’ proporciona la función masa de probabilidad
‘pbinom’ proporciona la función de distribución
‘qbinom’ proporciona la función de cuantiles
‘rbinom’ genera valores aleatorios.

Uso:

dbinom(x, size, prob, log = FALSE)
pbinom(q, size, prob, lower.tail = TRUE, log.p = FALSE)
qbinom(p, size, prob, lower.tail = TRUE, log.p = FALSE)
rbinom(n, size, prob)

Argumentos:

x,q: vector de cuantiles.

p: vector de probabilidades.

n: número de observaciones. Si no se especifica se toma igual a 1

log, log.p: son valores lógicos; si son TRUE, las probabilidades p se dan como probabilities log(p).

lower.tail: es un valor lógico; si es TRUE (por defecto), las probabilidades son $P[X \leq x]$, en otro caso $P[X > x]$.

EJEMPLOS

Calcular la probabilidad de obtener cuatro caras al lanzar seis veces una moneda perfecta

En este caso sería $P[X=4]$, con $X \rightarrow B(6,0.5)$

Código R

```
> dbinom(x=4, size=6, prob=0.5)
[1] 0.234375
```

Calcular la probabilidad de obtener como mucho cuatro caras al lanzar seis veces una moneda perfecta

En este caso sería $P[X \leq 4]$, con $X \rightarrow B(6,0.5)$

Código R

```
> pbinom(q=4, size=6, prob=0.5)
[1] 0.890625
```

O bien,

```
> pbinom(4,6,0.5)
[1] 0.890625
```

Calcular el valor x tal que $P[X \leq x]=0.89$

```
> qbinom(0.89,6,0.5)
[1] 4
```

Generar 10 valores pseudoaleatorios de una B(6,0.5)

```
> rbinom(10,6,0.5)
 [1] 4 5 5 3 3 4 3 3 5 4
```

Supongamos que el 10% de los tubos producidos por una máquina son defectuosos y supongamos que produce 15 tubos cada hora. Cada tubo es independiente de los otros. Se juzga que el proceso está fuera de control cuando se producen más de 4

tubos defectuosos en una hora concreta. Simular el número de tubos defectuosos producidos por la máquina en cada hora a lo largo de un periodo de 24 horas y determinar si el proceso está fuera de control en algún momento.

```
> TubosDefectuosos<-rbinom(24,15,0.1)
> TubosDefectuosos
 [1] 0 1 0 4 6 1 3 2 3 1 1 4 1 6 1 2 2 0 1 3 1 3 2 2
> any(TubosDefectuosos>4)
[1] TRUE
> sum(TubosDefectuosos>4)
[1] 2
```

La función **any** se aplica a un conjuntos de valores lógico y detecta si al menos uno de los valores es TRUE.

Ejercicios

1) Supongamos que en un proceso de manufactura la proporción de defectuosos es 0.15. Simular el número de defectuosos por hora en un periodo de 24 horas si se supone que se fabrican 25 unidades cada hora. Chequear si el número de defectuosos excede en alguna ocasión a 5. Repetir el procedimiento con p=0.2 y p=0.25.

```
> Defectuosos<-rbinom(24,25,0.15)
> Defectuosos
 [1] 1 5 4 3 2 6 5 4 2 4 5 7 4 2 3 4 1 4 6 2 4 4 2 3
> any(Defectuosos>5)
[1] TRUE
> sum(Defectuosos>5)
[1] 3
```

```
> Defectuosos<-rbinom(24,25,0.2)
> Defectuosos
 [1] 5 5 5 5 5 9 6 1 4 4 6 1 3 4 8 7 3 6 5 4 5 5 3 6
> any(Defectuosos>5)
[1] TRUE
> sum(Defectuosos>5)
[1] 7
```

```
> Defectuosos<-rbinom(24,25,0.25)
> Defectuosos
 [1]  7  5  7  8 11  4  4  7  4 11  8  5  2  7  5  8 10  7  8  6  4  5  7  9
> any(Defectuosos>5)
[1] TRUE
> sum(Defectuosos>5)
[1] 15
```

```
> Defectuosos<-rbinom(24,25,0.1)
> Defectuosos
 [1] 1 4 1 5 2 2 6 3 4 2 4 3 2 1 2 3 3 4 3 1 2 2 2 3
> any(Defectuosos>5)
[1] TRUE
> sum(Defectuosos>5)
```

```
[1] 1

> Defectuosos<-rbinom(24,25,0.05)
> Defectuosos
 [1] 1 0 0 0 3 1 0 4 2 0 1 3 2 2 2 2 0 2 0 2 1 1 2 0
> any(Defectuosos>5)
[1] FALSE
```

2) Simular 10000 números pseudoaleatorios de una variable aleatoria X con distribución B(20,0.3). Usar dichos valores para estimar P[X≤5], P[X=5], E[X], Var[X], el percentil 95, 99 y 99.9999 de X.

```
> binsim<-rbinom(10000,20,0.3)
> length(binsim[binsim<=5])/length(binsim)
[1] 0.4173
> pbinom(5,20,0.3)
[1] 0.4163708
> sum(binsim==5)/length(binsim)
[1] 0.1831
> dbinom(5,20,0.3)
[1] 0.1788631
> mean(binsim)
[1] 5.9922
> 20*0.3
[1] 6
> var(binsim)
[1] 4.084748
> 20*0.3*0.7
[1] 4.2 > quantile(binsim, probs=c(95,99,99.9999)/100)
    95%      99% 99.9999%
   9.00    11.00    13.99
> qbinom(0.95,20,0.3)
[1] 9
> qbinom(0.99,20,0.3)
[1] 11
> qbinom(0.999999,20,0.3)
[1] 16
```

3) Usar simulación para estimar la media y la varianza de una variable aleatoria B(18,0.76) y comparar dichos valores con los teóricos.

Solución del autor

```
>r<-rbinom(n=1000,size=18,p=.76)
>mean(r)
[1] 13.697
>var(r)
[1] 3.472664
```
Theoretical values are 13.68 and 3.2832.

4) Considerar la siguiente función diseñada para simular valores pseudoaleatorios de una distribución binomial usando el llamado método de inversión:

```
>ranbin<-function(n,size,prob){
```

```
+          cumbinom<-pbinom(0:(size-1),size,prob)
+         singlenumber<-function(){
+                          x<-runif(1)
+                          N<-sum(x>cumbinom)
+                         N
+                         }
+        replicate(n,singlenumber())
+        }


> ranbin(10,4,0.5)
 [1] 3 3 3 3 2 3 2 3 2 3
```

a) Estudiar con detenimiento esta función.

```
> size=5
> prob=0.2
> cumbinom<-pbinom(0:(size-1),size,prob)
> cumbinom
[1] 0.32768 0.73728 0.94208 0.99328 0.99968
> x<-runif(1)
> x>cumbinom
[1] FALSE FALSE FALSE FALSE FALSE
> sum(x>cumbinom)
[1] 0
> x
[1] 0.2978203
> x<-runif(1)
> x>cumbinom
[1]  TRUE  TRUE FALSE FALSE FALSE
> sum(x>cumbinom)
[1] 2
> x
[1] 0.8393769
> x<-runif(1)
> x>cumbinom
[1]  TRUE FALSE FALSE FALSE FALSE
> sum(x>cumbinom)
[1] 1
> x
[1] 0.4383818
> x<-runif(1)
> x>cumbinom
[1]  TRUE FALSE FALSE FALSE FALSE
> sum(x>cumbinom)
[1] 1
> x
[1] 0.6669283
```

b) Usar ranbin() para similar vectores de longitud 1000, 10000 y 100000 de una distribución B(10,0.5). Usar la función system.time() para comparar los tiempos de ejecución para esas simulaciones con los tiempos de ejecución correspondientes cuando se usa rbinom().

```
> system.time(rbinom(1000,10,0.5))
```

```
   user  system elapsed
      0       0       0
> system.time(rbinom(10000,10,0.5))
   user  system elapsed
      0       0       0
> system.time(rbinom(100000,10,0.5))
   user  system elapsed
   0.03    0.00    0.03
> system.time(ranbin(1000,10,0.5))
   user  system elapsed
   0.02    0.00    0.02
> system.time(ranbin(10000,10,0.5))
   user  system elapsed
   0.13    0.00    0.13
> system.time(ranbin(100000,10,0.5))
   user  system elapsed
   1.28    0.00    1.28
```

5) Una versión del teorema central del límite dice que si X es una variable aleatoria con distribución B(m,p) y

$Z=(X-mp)/\sqrt{mp(1-p)}$

Entonces Z es aproximadamente normal estándar y la aproximación mejora a medida que m aumenta.

El siguiente código simula un gran número de dichos valores Z para valores de m en el conjunto {1,2,…,100} y realiza un gráfico QQ-plot en cada caso:

```
> for(m in 1:100){
+            z<-(rbinom(20000,size=m,prob=0.4)-m*0.4)/sqrt(m*0.4*0.6)
+            qqnorm(z,ylim=c(-4,4),main=paste("QQ-plot,m=",m))
+            qqline(z)
+           }
```

a) **Ejecutar el código y observar cómo la distribución de Z cambia cuando m crece.**
b) **Modificar el código se modo que una "película" similar se produzca para los casos en que p=0.3, 0.2, 0.1, 0.05, respectivamente**

```
> x<-c(0.3,0.2,0.1,0.05)
> for(p in x){
+          for(m in 1:100){
+             z<-(rbinom(20000,size=m,prob=p)-m*p)/sqrt(m*p*(1-p))
+             qqnorm(z,ylim=c(-4,4),main=paste("QQ-plot, m=",m,", p=",p))
+             qqline(z)
+           }
+ }
```

DISTRIBUCIÓN DE POISSON

Descripción:

Las siguientes funciones proporcionan información sobre la distribución de Poisson de parámetro lambda:

‘dpois’ proporciona la función masa de probabilidad
‘ppois’ proporciona la función de distribución
‘qpois’ proporciona la función de cuantiles
‘rpois’ genera valores aleatorios.

Uso simplificado (se ha obviado el argumento log.p):

dpois(x, lambda)
ppois(q, lambda, lower.tail = TRUE)
qpois(p, lambda, lower.tail = TRUE)
rpois(n, lambda)

En este tipo de funciones, los argumentos son siempre los mismos salvo los relativos a los parámetros de la distribución concreta. En este caso lambda es el parámetro de la distribución de Poisson que representa su media.

Ejemplo

Supongamos que el número de accidentes que ocurren en una carretera al año tiene una distribución de Poisson de media 3.7.
a) Calcular la probabilidad de que en un año haya 6 accidentes.
b) Calcular la probabilidad de que un año haya menos de 2 accidentes
c) Calcular la probabilidad de que un año haya más de 8 accidentes.
d) Calcular el número máximo de accidentes que se producirán con probabilidad mayor o igual a 0.9.
e) Simula el número anual de accidentes que se producirán en un periodo de 20 años.

```
> dpois(6,3.7)
[1] 0.0881025
> ppois(2,3.7)
[1] 0.2854331
> ppois(8,3.7,lower.tail=FALSE)
[1] 0.01370281
> qpois(0.9,3.7)
[1] 6
> rpois(20,3.7)
 [1] 5 9 6 2 2 7 3 4 3 5 1 3 2 2 6 5 4 7 6 0
```

DISTRIBUCIONES DISCRETAS

A continuación se muestra una lista con los nombres de la funciones que se usarán para las distribuciones discretas más comunes precedidos de las letras **d** (para función masa de probabilidad), **p** (para función de distribución), **q** (para la función de cuantiles) y **r** (para generar valores aleatorios). Asimismo se muestran los parámetros de cada una de dichas distribuciones. En cualquier caso, se puede consultar las ayudas correspondientes para cualquier duda.

Distribución	**Nombre en R**	**Parámetros**
Binomial	binom	n, p

Poisson	pois	lambda
Binomial negativa	nbinom	k, p
Geométrica	geom	p
Hipergeométrica	hyper	m,n,k (ver ayuda (help("dhyper")))

DISTRIBUCIÓN NORMAL

Descripción

Las siguientes funciones proporcionan información sobre la distribución normal estándar (media 0 y desviación típica 1)

'dnorm' proporciona la función de densidad
'pnorm' proporciona la función de distribución
'qnorm' proporciona la función de cuantiles
'rnorm' genera valores aleatorios.

Uso simplificado (se ha obviado el argumento log.p):
dnorm(x, mean = 0, sd = 1)
pnorm(q, mean = 0, sd = 1, lower.tail = TRUE)
qnorm(p, mean = 0, sd = 1, lower.tail = TRUE)
rnorm(n, mean = 0, sd = 1)

De nuevo, los argumentos son los usuales en este tipo de funciones y, en este caso, los parámetros de la distribución son mean (media) y sd (desviación típica).

Ejemplo

Genera 100 valores aleatorios de una distribución normal de media 3 y desviación típica 2 (utiliza la semilla 111).

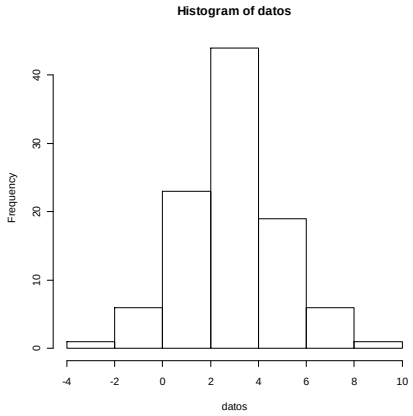
```
> options(width=80)
> set.seed(111)
> datos<-rnorm(100,3,2)
> datos
  [1]  3.470441423  2.338528257  2.376752352 -1.604691317  2.658247911
  [6]  3.280556450  0.005146689  0.979623162  1.103048790  2.012075566
 [11]  2.652651744  2.186802440  6.691272528  3.788108220  4.595057003
 [16] -0.133330720  2.828297982  2.281721038  0.612782067  3.728373474
 [21]  3.723324903  3.693928740  3.379473053  2.680846389  3.653098477
 [26]  4.196508405 -0.683068600  8.436111199  3.382488781  0.397407869
 [31] -3.226434603  1.117285209  5.800517563 -0.240940057 -1.531991914
 [36]  5.325987178  2.767689919  3.668512019  1.758283787  0.380310183
 [41]  0.648547917  0.757568934  0.276191033  3.962249157  4.483943251
 [46]  3.055649250  3.662759422  4.288228264  7.971323123  6.919963415
 [51]  3.383326770  6.105088544  4.828484574  3.717250750  3.350191272
 [56]  1.305464462  4.956463315  6.611736518  3.245829605  2.740455947
 [61]  2.567142682  5.892956335  3.819419603  4.821833144  5.860716333
 [66]  2.237416089  3.404614354  1.387601612  3.589268367  5.809766166
 [71]  5.047533693  3.952252129  1.659339340  3.318468642  2.234569234
 [76]  4.871525186  1.736935465  2.803387845  5.063969966  3.775616859
 [81]  0.487741373  1.426094540  3.859623097  2.247167562  0.567541867
```

```
 [86]  5.058557028  3.860793998  0.508851960  1.794543027  4.320138775
 [91]  7.101499052  3.981616358 -0.462958838  4.421767319  3.027645823
 [96]  0.197916805  5.518247331  2.745044963  1.541226973  0.577277280
```

Representamos a continuación el histograma. Si usamos el código

```
> hist(datos)
```

el resultado es



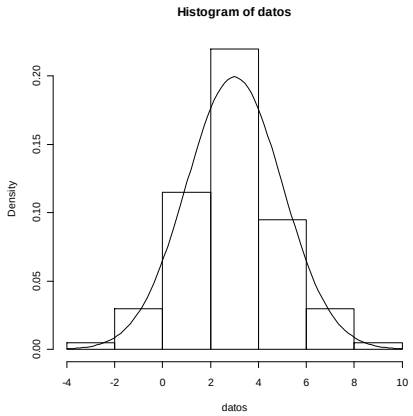
Sin embargo, si usamos el código

```
> hist(datos, freq=FALSE)
```

aparecen representadas frecuencias relativas y es posible hacer una comparación con la función de densidad teórica. Dicha comparativa se hace ejecutando a continuación el siguiente código

```
> curve(dnorm(x,3,2),add=TRUE)
```

donde el parámetro `add=TRUE` tiene el efecto de superponer la curva al histograma. Si no se pone se borraría dicho histograma.



DISTRIBUCIÓN EXPONENCIAL

Descripción

Las siguientes funciones proporcionan información sobre la distribución exponencial con parámetro razón *rate* (notemos que la media de esta distribución es 1/razón)

‘dexp’ proporciona la función de densidad
‘pexp’ proporciona la función de distribución
‘qexp’ proporciona la función de cuantiles
‘rexp’ genera valores aleatorios.

Uso simplificado (se ha obviado el argumento log.p):

dexp(x, rate = 1, log = FALSE)
pexp(q, rate = 1, lower.tail = TRUE, log.p = FALSE)
qexp(p, rate = 1, lower.tail = TRUE, log.p = FALSE)
rexp(n, rate = 1)

De nuevo, los argumentos son los usuales en este tipo de funciones y, en este caso, el parámetro de la distribución es lambda (media=1/lambda). Por defecto dicho parámetro toma el valor 1.

Ejemplo

Supongamos que el tiempo de servicio en un banco se modeliza por una variable aleatoria con distribución exponencial de razón 3 clientes por minuto.
Calcular la probabilidad de que un cliente sea servido en menos de un minuto

La solución es P[X≤1], siendo X→exp(3) que se calcula con R usando el siguiente código

```
> pexp(1,3)
[1] 0.950213
```

Dado que la función de densidad de una variable aleatoria con distribución exp(λ) es

$$f(t) = \lambda e^{-\lambda t},\ t > 0$$

y, por tanto, su función de distribución

$$F(t) = 1 - e^{-\lambda t},\ t > 0,$$

una forma simple de simular valores pseudoaleatorios se basa en el método de inversión aplicando

$$F^{-1}(x) = -\frac{\log(1-x)}{\lambda}$$

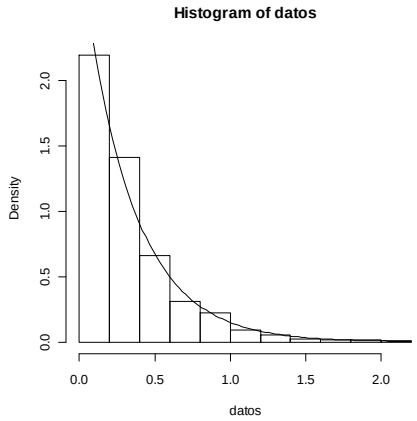
a valores pseudoaleatorios de una distribución U(0,1).

Simular mediante el método de inversión 10 valores de una distribución exp(3) usando la semilla 111

```
> set.seed(111)
> x<-runif(10)
> datos<--(log(1-x))/3
> datos
 [1] 0.299632037 0.432128211 0.154235177 0.241149783 0.158091293
0.180621532
 [7] 0.003571682 0.253306014 0.188638892 0.032788170
```

Simular 1000 valores de una exp(3) y comparar su histograma con la función de densidad teórica de la distribución

```
> set.seed(111)
> x<-runif(1000)
> datos<--(log(1-x))/3
> hist(datos, freq=FALSE)
> curve(dexp(x,3),add=TRUE)
```



También se puede usar la función **rexp()** para simular valores de una distribución exponencial

```
> set.seed(111)
> x<-rexp(10,3)
> x
 [1] 0.06198752 0.15098741 0.39161173 0.00994922 0.40126668 0.45549883 1.50769575 0.02153016 0.25269409 0.06015233
```

Comprobemos de forma empírica la propiedad de falta de memoria de la distribución exponencial

Para ello vamos a comparar las probabilidades P[X>x] con P[X>x+y|X>y]

```
> pexp(3,0.5,lower.tail=FALSE)
[1] 0.2231302
> pexp(8,0.5,lower.tail=FALSE)/pexp(5,0.5,lower.tail=FALSE)
[1] 0.2231302
```

DISTRIBUCIONES CONTINUAS

A continuación se muestra una lista con los nombres de la funciones que se usarán para las distribuciones comunes más comunes precedidos de las letras **d** (para función de densidad), **p** (para función de distribución), **q** (para la función de cuantiles) y **r** (para generar valores aleatorios). Asimismo se muestran los parámetros de cada una de dichas distribuciones. En cualquier caso, se puede consultar las ayudas correspondientes para cualquier duda.

Distribución	**Nombre en R**	**Parámetros**
Normal	norm	mean (μ), sd (σ)

Exponencial	exp	rate (λ)
Gamma	gamma	shape, scale
Beta	beta	shape1, shape2
Chi-cuadrado	chisq	df (grados de libertad)
T de Student	t	df (grados de libertad)
F de Snedecor	f	df1, df2 (grados de libertad)