

Tutorial de R

Materiales del curso de introducción a
R impartido por
Xavier Solé y Joan Valls
© 2004

Sesión 1 – Introducción y preliminares

- El entorno R.
- *Software* relacionado: un poco de historia.
- Distribuciones de R.
- Primeras nociones: el *help* de R.
- Primeras nociones: comandos, *case sensitivity*
- Primeras nociones: ejecutar comandos desde/enviar salida a archivos
- Primeras nociones: permanencia de los datos y eliminación de objetos
- Primeras nociones: librerías de R
- Ejercicios propuestos

El entorno R.

Suite integrada para la manipulación de datos, cálculo y procedimientos gráficos. Los principales aspectos que ofrece son:

- Facilidad para el manejo y el almacenamiento de datos.
- Un conjunto de operadores para cálculo con *arrays* y matrices.
- Una colección extensa e integrada de herramientas intermedias para el análisis de datos.
- Multitud de facilidades gráficas.
- Un lenguaje de programación simple y efectivo que incluye las estructuras de control clásicas, funciones recursivas y facilidades para el *input* y *output* de datos y resultados.

R es un entorno altamente dinámico, y a menudo se concibe como un vehículo para desarrollar (nuevos) métodos interactivos de análisis de datos

- Ventaja: incorporación constante de nuevos métodos.
- Inconveniente: por su dinamismo, a menudo código antiguo de R se queda desfasado y no funciona con las nuevas versiones del entorno.

Software relacionado: un poco de historia.

R es una implementación *open-source* del lenguaje S (*Bell Labs* - principios de los 90), que también es la base del sistema S-Plus (entorno comercial).

R y S-Plus aún comparten una gran mayoría de código e instrucciones, pero probablemente serán dos entornos independientes en un futuro a medio plazo.

Diferencias entre R y S-Plus

- Precio: R gratuito, S-Plus comercial
- S-Plus es más "amigable": funcionalidad por menús (limitada)
- R se ejecuta exclusivamente mediante el envío de instrucciones en la línea de comandos: curva de aprendizaje más lenta, aunque más versatilidad
- R dispone de una comunidad de desarrolladores/usuarios detrás que se dedican constantemente a la mejora y a la ampliación de las funcionalidades y capacidades del programa. Nosotros mismos podemos ser desarrolladores de R!!

Distribuciones de R.

Actualmente R se distribuye para los siguientes Sistemas Operativos:

- *Windows*: entorno gráfico.
 - *Linux (Debian/Mandrake/SuSe/RedHat/VineLinux)*
 - *MacOS X*
 - **Código fuente: ampliación a sistemas *Unix***
- } Entorno "texto"

Comunicación directa con el Sistema Operativo

Las funciones de R se agrupan en paquetes (*packages, libraries*), los que contienen las funciones más habituales se incluyen por defecto en la distribución de R, y el resto se encuentran disponibles en la *Comprehensive R Archive Network (CRAN)*

<http://cran.r-project.org>

```
> library() #Vemos la lista de librerías disponibles para ser cargadas. En Windows se puede hacer por menús.
```

```
> search() #Para ver la lista de librerías ya cargadas
```

```
[1] ".GlobalEnv"      "package:ctc"      "package:methods" "package:stats"    "package:graphics" "package:utils"  
[7] "Autoloads"       "package:base"
```

```
> ls(4) #Vemos las funciones del paquete stats
```

Primeras nociones: el *help* de R.

R es sensible a mayúsculas y minúsculas.

#Maneras clásicas de consultar la ayuda

```
> help(solve)
```

```
> ?solve
```

#Consulta de ayuda para funciones con caracteres especiales y para algunas palabras reservadas como *if*, *for*
#y *function*

```
> help("[[")
```

```
> ?"["
```

#Ayuda en HTML: abre el navegador (sólo si tenemos la ayuda en HTML instalada)

```
> help.start()
```

#Búsqueda de términos relacionados

```
> help.search("clustering")
```

#Ejecución de ejemplos de una función. Las comillas son opcionales.

```
> example("hclust")
```

Primeras nociones: comandos, *case-sensitivity*.

R distingue entre mayúsculas y minúsculas:

```
#Dos objetos diferentes
```

```
> a <- 3
```

```
> A <- 6
```

En R hay dos tipos de comandos: expresiones y asignaciones

```
#Expresión. El resultado se muestra por pantalla y no se guarda.
```

```
> rnorm(10)
```

```
[1] 0.71690438 0.07539554 0.73687196 -0.43048351 -0.31119274 0.26747903 0.62100426 -1.80412464 -0.53979447 -0.27172816
```

```
#Asignación: no se muestra nada.
```

```
> v <- rnorm(10)
```

```
> v
```

```
[1] 0.46212728 -0.87838057 0.83872171 0.01630945 -0.49676616 -0.41782133 1.22499917 -0.89104983 0.47660672 1.03399336
```

Los comandos se separan por ";" o por un salto de línea. Un comando se puede escribir en más de una línea. Los comandos se agrupan con "{" y "}"

```
#Comandos separados por ";"
```

```
> a <- 3; b <- 5
```

```
#Comando escrito en más de una línea
```

```
> a <-
```

```
+ 3
```

Primeras nociones: ejecutar comandos desde/enviar salida a archivos.

Ejecutar comandos desde un archivo de texto

```
#En Windows también está disponible en el menú  
> source("comandos.R")
```

Guardar la salida (resultado) de nuestros comandos

```
#Inicio de volcado  
> sink("resultado.txt")
```

```
#Fin de volcado  
> sink()
```

Por defecto R busca los archivos en el directorio activo. Para verlo/cambiarlo ir al menú *File -> Change Dir.*

Si queremos referenciar archivos mediante su ruta completa tenemos que utilizar los caracteres "\\" o "/":

```
#Hacemos un source poniendo la ruta completa del archivo  
> source("c:\\programas\\comandos.R") #Manera 1  
> source("c:/programas/comandos.R") #Manera 2
```


Primeras nociones: permanencia de los datos y eliminación de objetos.

Las entidades que R crea y manipula se llaman objetos. Dichos objetos pueden ser:

- Escalares: números, caracteres, lógicos (booleanos), factores
- Vectores/matrices/listas de escalares
- Funciones
- Objetos *ad-hoc*

Dichos objetos se guardan en un *workspace*. Durante una sesión de R todos los objetos estarán en memoria, y se pueden guardar en disco para próximas sesiones. Es recomendable utilizar diversos *workspaces* para los diferentes análisis que queramos hacer. Los *workspaces* se cargan y se guardan con las instrucciones *load* y *save.image* (disponibles en el menú).

```
#Vemos el conjunto de objetos que tenemos en nuestro workspace
```

```
> ls() #También se puede hacer con la instrucción objects()
```

```
[1] "a" "A" "b"
```

```
#Para borrar objetos utilizamos la instrucción rm
```

```
> rm(A,b)
```

```
#Verificamos que los objetos A y b se han borrado
```

```
> ls()
```

```
[1] "a"
```

Ejercicios propuestos.

1. Abrir R y cargar la librería *foreign*. Mirar en qué posición dentro de la lista de librerías cargadas la ha colocado R. ¿Qué funciones tiene *foreign*? Mirar el *help* de la función *read.spss*. Consultar la ayuda de las otras funciones. ¿Para qué sirven? ¿Podéis deducir la funcionalidad global de la librería? Verificarlo en la web de CRAN.
2. Guardar el *workspace* en un directorio de vuestro disco con el nombre "sesion1.Rdata" (lo podeis hacer por menús o por comandos – investigar la instrucción *save.image*). Cambiar el directorio activo al mismo donde está el *workspace* recién guardado (hacerlo por menús). Crear un archivo llamado "sesion1.R" en ese mismo directorio y escribir algunas instrucciones sencillas de R (*ls()*; *a<-1*; *A<-3*; *b <- 7*; *ls()*). Ejecutar estas instrucciones directamente desde el archivo mediante el comando *source*. Hacerlo vía menú y via comando. Verificar que las instrucciones del archivo pueden estar separadas por saltos de línea o por puntos y coma.
3. Ejecutar las instrucciones *rnorm(100)* y *runif(50)*, pero haciendo que se guarde directamente el resultado en un archivo llamado "res.txt". Deshaced el envío de la salida hacia el fichero para volver a recuperar el funcionamiento normal de R.
4. Borrar el objeto *a* de vuestro *workspace*. Una vez hecho esto, guardarlo otra vez cambiándole el nombre (le podéis llamar "sesion1bis.RData"). Cerrar R y volverlo a abrir. Cargar el *workspace* "sesion1bis.RData" mediante la instrucción *load* o mediante menús. ¿Qué objetos aparecen? ¿Está el objeto *a*?

Sesión 2 – Manipulaciones simples: números y vectores

- Vectores y asignaciones
- Aritmética de vectores
- Generación de secuencias regulares
- Valores *missing*
- Vectores lógicos
- Vectores de cadenas de caracteres (*strings*)
- Indexación de vectores
- Coerción de tipos
- Otros tipos de objetos en R
- Ejercicios propuestos

Vectores y asignaciones.

R trabaja sobre estructuras de datos. La estructura más simple es un vector numérico, que consiste en un conjunto ordenado de números.

```
#Creamos un vector de reales mediante la función c y lo guardamos en la variable x.  
> x <- c(1.3, 2.5, 4.2, 9.7, 8.1)
```

```
#Un número por sí mismo es un vector de longitud 1  
> v <- 5
```

```
#Otras maneras de asignar menos utilizadas  
> assign("x", c(1.3, 2.5, 4.2, 9.7, 8.1)) #Instrucción assign  
> c(1.3, 2.5, 4.2, 9.7, 8.1) -> x #Asignación en la otra dirección
```

Si no se utiliza ninguna de las tres maneras de asignación ("`<-`", "`->`", "`assign`") el resultado de nuestra expresión se muestra por pantalla pero no quedará guardado.

```
#Expresión: el resultado no se guarda  
> c(x,0,x)  
[1] 1.3 2.5 4.2 9.7 8.1 0.0 1.3 2.5 4.2 9.7 8.1
```

```
#Objeto especial de R que guarda el resultado del último comando ejecutado  
> .Last.value  
[1] 1.3 2.5 4.2 9.7 8.1 0.0 1.3 2.5 4.2 9.7 8.1
```

Aritmética de vectores.

Al contrario que la mayoría de lenguajes de programación, R tiene aritmética vectorial, por lo que los vectores pueden aparecer en las expresiones que generamos.

En caso que los vectores que aparecen en una expresión no sean de la misma longitud, el más corto se "recicla" hasta que alcanza la longitud del más largo.

```
#Generamos dos vectores.
```

```
> x <- c(1.3, 2.5, 4.2, 9.7, 8.1)
```

```
> y <- c(x,0,x)
```

```
#Utilizamos x e y en una nueva expresión. Como x es más corto
#que y, se reciclará para adquirir su misma longitud. R nos avisa
#de este hecho porque los dos vectores no tienen una longitud
#múltiple. El 1 en este caso también se recicla y pasa a ser un
#vector de once unos.
```

```
> v <- 2*x + y + 1
```

```
Warning message:
```

```
longer object length
```

```
is not a multiple of shorter object length in: 2 * x + y
```

```
> v
```

```
[1] 4.9 8.5 13.6 30.1 25.3 3.6 7.3 11.9 24.6 26.9 11.7
```

Operador/función	Símbolo/instrucción
suma	+
resta	-
multiplicación	*
división	/
módulo	%%
división entera	%/%
raíz cuadrada	sqrt
logaritmo nep.	log
log gen	logb
exponencial	exp
seno	sin
coseno	cos
tangente	tan
máximo	max
mínimo	min
rango	range
longitud	length
sumatorio	sum
producto	prod
media	mean
desv. estándar	sd
varianza	var

Generación de secuencias regulares.

R dispone de instrucciones para generar secuencias de números. Una de las más utilizadas es el operador ":"

```
#Generamos un vector con los números 1, 2, 3, 4, ..., 29, 30.  
> 1:30 #Esto es equivalente al vector c(1, 2, ..., 29, 30)  
[1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
```

```
#El operador ":" tiene la máxima preferencia  
> n <- 10  
> 1:n-1 #Aquí prevalece ":" sobre "-"  
[1] 0 1 2 3 4 5 6 7 8 9  
> 1:(n-1) #Forzamos la prioridad del "-"  
[1] 1 2 3 4 5 6 7 8 9
```

Con la función *seq* también se pueden generar secuencias de números

```
#Generamos una secuencia de 1 a 30 saltando dos números cada vez  
> seq(1,30,by=2)  
[1] 1 3 5 7 9 11 13 15 17 19 21 23 25 27 29
```

```
#La función seq admite también la longitud de la secuencia que queremos generar, de manera que ella misma  
#decide el intervalo automáticamente  
> seq(1,30,length=15)  
[1] 1.000000 3.071429 5.142857 7.214286 9.285714 11.357143 13.428571 15.500000 17.571429  
19.642857 21.714286 23.785714 25.857143 27.928571 30.000000
```

La función *rep* sirve para generar repeticiones de objetos (escalares o vectores)

Valores *missing*.

En R los valores "desconocidos" o "no disponibles" (*missings*) se simbolizan con el valor especial NA (*Not Available*). Cualquier operación que incluya un NA en general devolverá NA como resultado.

La función *is.na* nos permite saber si un elemento es *missing* o no.

```
#Generamos un vector con los números 1, 2, 3 y un missing al final
```

```
> z <- c(1:3, NA)
```

```
> z
```

```
[1] 1 2 3 NA
```

```
> is.na(z) #Miramos que valores del vector son missing.
```

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

```
#La expresión z==NA no funciona!!!
```

```
> z==NA
```

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

Hay un segundo tipo de *missings* que se producen por computación numérica, lo que se llama *Not a Number*. En R se simbolizan con el valor NaN.

```
> 0/0 #Provocamos un error numérico
```

```
[1] NaN
```

La función *is.na* retorna TRUE tanto para los NA como para los NaN. En cambio, la función *is.nan* sólo retorna TRUE para los NaN.

Vectores lógicos.

R permite la manipulación de cantidades lógicas. Los valores de un vector lógico pueden ser TRUE o T (cierto), FALSE o F (falso) y NA/NaN.

Los vectores lógicos se generan mediante condiciones:

```
#Generamos un vector de 1 a 10
> x <- 1:10
#cond1 vector lógico, de la misma longitud que x, donde cada casilla
#nos dice si la correspondiente casilla de x cumple la condición x>7
> cond1 <- x > 7
> cond1
[1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE TRUE TRUE TRUE
```

En R los vectores lógicos se pueden utilizar en aritmética ordinaria, siendo substituído (coercionado) el FALSE por 0 y el TRUE por 1.

```
> cond2 <- x >= 9 #Generamos otra condición
> cond1 & cond2 #Hacemos una and lógica de las dos condiciones
[1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE TRUE TRUE
> !cond1 #Negación lógica del vector cond1
[1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE FALSE FALSE FALSE
```

Operador	Símbolo
igualdad	==
desigualdad	!=
menor	<
menor igual	<=
mayor	>
mayor igual	>=
<i>and</i> lógica	&
<i>or</i> lógica	
negación lógica	!

Vectores de cadenas de caracteres (*strings*).

Los vectores de cadenas de caracteres se usan a menudo en R, principalmente para guardar etiquetas.

Los caracteres pueden ser entrados utilizando comillas simples (") o dobles (""), aunque después R los muestra normalmente rodeados de comillas dobles. Como en C, utilizan el carácter "\" como carácter de escape. Algunos caracteres útiles son el salto de línea (\n) y tabulador (\t).

La función *paste* toma un número arbitrario de argumentos y los concatena uno a uno para transformarlos después en caracteres

```
#Concatenamos un vector de dos letras y otro de dos números posición a posición
> paste(c("X","Y"),1:2,sep="")
[1] "X1" "Y2"
```

```
#Ahora concatenamos vectores de diferente longitud y hacemos que el separador de la concatenación sea un
#punto ".". Nótese el reciclaje del vector corto.
> paste(c("X","Y"),1:4,sep=".")
[1] "X.1" "Y.2" "X.3" "Y.4"
```

```
#La función paste tiene otro argumento, llamado collapse, que fusiona todos los strings que genera para dar
#una única cadena de caracteres de salida. El valor de collapse es lo que se coloca en medio de las
#diferentes cadenas de caracteres que se unen.
> paste(c("X","Y"),1:4,sep=".",collapse="-")
[1] "X.1-Y.2-X.3-Y.4"
```

Indexación de vectores (1).

Se puede seleccionar un subconjunto de los elementos de un vector añadiendo al lado de su nombre un *vector de índices* entre corchetes (`[]`). Los vectores de índices pueden ser de cuatro tipos:

- Vectores lógicos: han de tener la misma longitud que el vector del cual se quieren seleccionar elementos. Los valores que corresponden a TRUE en su correspondiente posición del vector lógico serán seleccionados, y los que corresponden a FALSE serán omitidos.
- Vectores de números enteros positivos: los valores que contienen deben estar entre 1 y $length(x)$. Los correspondientes elementos del vector x son seleccionados y concatenados en ese orden.
- Vectores de números enteros negativos: especifican los valores que serán excluidos en vez de incluidos.
- Vectores de strings: sólo se pueden utilizar cuando el objeto tiene un atributo llamado *names* (nombres) que identifica sus componentes. En este caso, un subvector del vector de nombres se utilizaría de la misma manera como se utilizaría un vector de enteros positivos. La ventaja es que estos nombres suelen ser más fáciles de recordar que los índices de los elementos a seleccionar.

Indexación de vectores (2).

```
> x <- c(1:5,NA,6:8,NA,9,10) #Generamos un vector con enteros de 1 a 10 y algunos missings por medio.
```

```
> x[!is.na(x)] #Indexación por vector lógico: escogemos aquellos elementos que no son NA.
```

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

```
> x[!is.na(x) & x%%2==0] #Vector lógico más complejo.
```

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

```
> x[1:5] #Indexación mediante vector de enteros positivos: nos quedamos con los 5 primeros elementos de x.
```

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

```
> x[c(1,3,5,7,9,11)] #Nos quedamos con las posiciones impares
```

```
[1] 1 3 5 6 8 9
```

```
> x[-(1:5)] #Indexación por enteros negativos: eliminamos los 5 primeros elementos de x
```

```
[1] NA 6 7 8 NA 9 10
```

```
> y <- c(5,18,7,13) #Indexación por vectores de strings.
```

```
> names(y) <- c("uno","dos","tres","cuatro") #Añadimos los names al vector.
```

```
> y[c("dos","tres")] #Seleccionamos mediante los names.
```

```
dos tres
```

```
18 7
```

Una indexación también puede aparecer en la parte izquierda de una asignación.

```
> x[is.na(x)] <- 0 #Substituimos los NA por 0.
```

```
> x[x>5] <- -x[x>5] #A los valores del vector superiores a 5 les cambiamos el signo.
```

Coerción de tipos.

R, como la mayoría de lenguajes de programación, dispone de funciones para realizar la coerción (transformación) de tipos.

Para que la coerción tenga lugar no se ha de dar ninguna incompatibilidad entre el tipo origen y el tipo destino.

Así como las funciones de coerción, que suelen comenzar con la palabra *as* seguida de un punto (*as.integer*) y el tipo de destino, también existen las funciones de verificación de tipos, que se componen de *is* seguido de un punto y el tipo a verificar (*is.integer*).

```
# Coerción correcta
> a <- c("1","2","3")
> b <- as.numeric(a)
> b
[1] 1 2 3
```

```
# Coerción incorrecta. Se introducen NA's.
> a <- c("1","2","x")
> b <- as.numeric(a)
> b
[1] 1 2 NA
```

Funciones más habituales

```
as.character
as.integer
as.double
as.complex
as.numeric
as.logical
as.factor
```

Otros tipos de objetos en R.

- *Arrays* y matrices (*matrix*): generación multidimensional de los vectores. Todos los elementos de la matriz han de ser del mismo tipo.
- Factores (*factor*): útiles para el uso de datos categóricos.
- Listas (*list*): generalización de los vectores donde los elementos pueden ser de diferentes tipos (incluso vectores o nuevas listas).
- *Data frames*: matrices donde las diferentes columnas pueden tener valores de diferentes tipos.
- Funciones (*function*): conjunto de código de R ejecutable y parametrizable.
- Cualquier objeto en R tiene las propiedades *mode* y *length*.
 - *Mode*: tipo de datos de los elementos que forman un objeto (*numeric*, *complex*, *logical* y *character*).
 - *Length*: número de elementos que contiene el objeto.

Ejercicios propuestos.

1. Con la ayuda de la función *rnorm*, generar un vector de 100 valores que sigan una distribución normal (0,1) y guardarlo en una variable llamada *x* (tenéis que hacer `x<-rnorm(100)`). Crear un vector con los valores de las posiciones pares (*p*) de *x* y otro con los valores de las posiciones impares (*i*). Verificar que los dos vectores tienen la misma longitud. Calcular la media y desviación estándar tanto de *p* como de *i*. Calcular también la media y desviación estándar de *p* e *i* posición a posición (este último punto sin utilizar las funciones *mean* ni *sd*).
2. Generar mediante la instrucción *seq* el vector (0, 0.1, 0.2, 0.3, ..., 1) de dos maneras diferentes. Sumar a este vector ahora el vector (1,2,3): ¿qué hace R, os da algún mensaje? Hay que vigilar con el reciclaje!!
3. Generar un vector de 1 a 200. Dividir los 100 primeros elementos del vector por el módulo de la división de su correspondiente posición entre 5. Con la ayuda de la función *is.finite*, que tiene un funcionamiento idéntico a *is.na*, cambiar los valores infinitos resultantes por NA. Hacer una subselección de este vector para quedarnos sólo con los elementos que no son NA. Ahora hacer una subselección para descartar los diez primeros números de este vector resultante (tenéis que decir explícitamente que queréis quitar estos diez elementos, no que os queréis quedar con todos los restantes).
4. La instrucción *rep(x, times)* sirve para hacer repeticiones *times* veces del objeto *x*. Crear un vector que sea la repetición del vector (1, 2, 3, 4, 5) tres veces. Ahora crear el vector (1, 1, 1, 2, 2, 2, ..., 5, 5, 5). ¿Cómo tiene que ser ahora el parámetro *times*? (Pista: le teneis que decir cuántas veces seguidas queréis repetir cada elemento del vector). ¿Podéis crear el vector (1,2,2,3,3,3,4,4,4,4,5,5,5,5,5)? Concatenad estos tres vectores que habéis obtenido uno detrás del otro.
5. Volvamos a los vectores *i* y *p* del ejercicio 1. Cread un vector lógico que indique cuando alguno de los dos valores que se aparean posición a posición son mayores que zero. Aplicando lógica booleana, ¿podéis complicar un poco más la expresión lógica para obtener el mismo resultado?

Ejercicios propuestos.

6. R dispone de dos objetos predeterminados, *letters* y *LETTERS*, que corresponden al conjunto de letras minúsculas y mayúsculas, respectivamente. Con la ayuda de la función *paste* y estos dos objetos, conseguir obtener el vector ("aA", "bB", "cC", ..., "zZ"). ¿Podéis obtener también el *string* "aA-bB-cC-...-zZ"? (Pista: utilizar el argumento *collapse*).

Sesión 3 – *Arrays* y matrices

- *Arrays*
- Indexación de *arrays*
- La función *array*
- Facilidades con matrices
- Ejercicios propuestos

Arrays.

Un *array* es un conjunto de datos de k dimensiones. El caso más sencillo se da con $k=2$, lo que llamaremos matriz (*matrix*). Todos los elementos de un *array* han de ser del mismo tipo.

En R cualquier *array* ha de tener asociado un atributo llamado *dim* que indique los límites superiores de cada una de las dimensiones. Por definición el límite inferior es 1.

```
> a <- 1:42 # Creamos un vector de 42 posiciones
# Lo transformamos en un array añadiéndole el límite superior de cada dimensión. En este caso en un array de
# tres dimensiones de longitudes 3, 7 y 2. Nótese que las dimensiones que se "mueven" más rápido son las de
# más a la izquierda.
> dim(a) <- c(3,7,2)
> a
, , 1

      [,1] [,2] [,3] [,4] [,5] [,6] [,7]
[1,]    1    4    7   10   13   16   19
[2,]    2    5    8   11   14   17   20
[3,]    3    6    9   12   15   18   21

, , 2

      [,1] [,2] [,3] [,4] [,5] [,6] [,7]
[1,]   22   25   28   31   34   37   40
[2,]   23   26   29   32   35   38   41
[3,]   24   27   30   33   36   39   42
```

Indexación de *arrays* (1).

Los *arrays* se indexan exactamente igual que los vectores. En este caso tenemos que establecer un vector de índices (sesión 2) para cada dimensión del *array*. Los diferentes vectores se pondrán separados por comas y toda la indexación también irá entre corchetes ([]).

Si sobre una determinada dimensión no queremos aplicar ningún tipo de selección simplemente dejaremos el espacio correspondiente a esa dimensión en blanco.

```
# Del array anterior seleccionamos la posición 1, 4, 2  
> a[1,4,2]  
[1] 31
```

```
# Ahora fijamos las dos primeras dimensiones y dejamos libre la tercera  
> a[2,4,]  
[1] 11 32
```

```
# a[,2,] es un array con vector de dimensiones c(3,2) y vector de datos que contiene los valores  
# c(a[1,2,1], a[2,2,1], a[3,2,1], a[1,2,2], a[2,2,2], a[3,2,2])  
> a[,2,]  
  [,1] [,2]  
[1,]   4  25  
[2,]   5  26  
[3,]   6  27
```

Indexación de *arrays* (2).

Un *array* también se puede indexar con un único *array* de índices. Dicho *array* tendrá tantas columnas como dimensiones tenga el *array* de donde se quieren subseleccionar los elementos y un número de filas indeterminado, que equivaldrá al número de elementos que queramos seleccionar. En el caso de las matrices el proceso se hace más claro.

```
# Generamos una matriz 4 * 5
> x <- 1:20
> dim(x) <- c(4,5)
> x
      [,1] [,2] [,3] [,4] [,5]
[1,]    1    5    9   13   17
[2,]    2    6   10   14   18
[3,]    3    7   11   15   19
[4,]    4    8   12   16   20
```

Ahora queremos seleccionar los elementos $x[1,3]$, $x[2,2]$ y $x[3,1]$. Para eso crearemos el *array* de índices, # que en este caso tendrá dos columnas y tres filas.

```
> i <- c(1,2,3,3,2,1) #Vector de valores
> dim(i) <- c(3,2)   #Vector de dimensiones
> i
      [,1] [,2]
[1,]    1    3
[2,]    2    2
[3,]    3    1
```

Indexación de *arrays* (3)

```
# Vemos los elementos seleccionados
```

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

```
# Ponemos esos elementos a cero
```

```
> x[i] <- 0
> x
```

```
      [,1] [,2] [,3] [,4] [,5]
[1,]    1    5    0   13   17
[2,]    2    0   10   14   18
[3,]    0    7   11   15   19
[4,]    4    8   12   16   20
```

La función *array*

Además de con un vector de datos y un atributo *dim*, los *arrays* también pueden ser contruídos con la función *array*, a la cual se le ha de pasar como argumentos el vector de valores y el vector de dimensiones. Esta función actúa igual que el método "manual", pero acepta vectores de dimensiones que no encajen exactamente con el tamaño del vector de datos (reciclaje).

```
> Z1 <- array(x,dim=c(4,5)) # Creamos el mismo array que antes
```

```
> Z2 <- array(x,dim=c(5,5)) # Los valores del vector de datos se reciclan
```

```
> Z3 <- array(x,dim=c(4,4)) # Hay valores del vector de datos que no se incluyen en el nuevo array
```

Facilidades con matrices (1)

Las matrices son una particularización de los *arrays*, donde el número de dimensiones es 2. Debido a su gran uso, R dispone de funciones específicas para matrices.

Al igual que la función *array*, la función *matrix* nos permite crear matrices.

```
# Creamos una matriz de dimensiones 5 * 4 con la función matrix.  
# Si queremos que la llene por filas le añadiremos el parámetro  
# byrow=TRUE.  
> a <- matrix(1:20,5,4)  
> a
```

```
      [,1] [,2] [,3] [,4]  
[1,]    1    6   11   16  
[2,]    2    7   12   17  
[3,]    3    8   13   18  
[4,]    4    9   14   19  
[5,]    5   10   15   20
```

```
# Miramos las dimensiones. También lo podemos hacer con la función dim.  
> nrow(a)  
[1] 5  
> ncol(a)  
[1] 4
```

Función	Operador
número filas	<code>nrow</code>
número columnas	<code>ncol</code>
producto matricial	<code>%*%</code>
transposición	<code>t</code>
diagonal	<code>diag</code>

Facilidades con matrices (2)

R permite utilizar los operadores matemáticos clásicos (+, -, *, /,...) para operaciones con matrices de mismas dimensiones. En este caso se realiza la operación para cada par de elementos

```
# Creamos dos matrices de iguales dimensiones
> a <- matrix(1:20,5,4)
> b <- matrix(21:40,5,4,byrow=TRUE)

# Ahora podemos hacer operaciones con las dos matrices
> a + b #Suma posición a posición.
```

```
      [,1] [,2] [,3] [,4]
[1,]    22    28    34    40
[2,]    27    33    39    45
[3,]    32    38    44    50
[4,]    37    43    49    55
[5,]    42    48    54    60
```

Las matrices también se pueden construir a base de la unión de vectores individuales, ya sea apilándolos por filas o por columnas. Para eso tenemos las funciones *rbind* y *cbind*. Para transformar un array en vector utilizaremos la función *as.vector*

```
> rbind(1:5,11:15,21:25) #Apilamos tres vectores por filas. El resultado es un objeto de tipo matrix.
```

```
      [,1] [,2] [,3] [,4] [,5]
[1,]     1     2     3     4     5
[2,]    11    12    13    14    15
[3,]    21    22    23    24    25
```

Ejercicios propuestos

1. Generar una matriz de dimensiones $3 * 5$ llena de ceros de 4 maneras diferentes.
2. Utilizando la función *array*, crear un *array* tridimensional $4*4*4$, llenándolo con los números del 1 al 64. Hacer una subselección de los índices 2 y 4 de la primera dimensión y 1 y 3 de la tercera. Mirar las dimensiones del nuevo array. Convertir el resultado de esta subselección en un vector, y después construir con ese mismo vector una matriz $4 * 4$, llenándola por filas.
3. Utilizando la matriz $4 * 4$ del problema anterior, invertir el orden de sus columnas y transponerla. A todos superiores a 30 de esta nueva matriz cambiarles el signo. Hacer el producto matricial de la matriz original por la que acabamos de obtener.
4. Sin utilizar *cbind* ni *rbind*, crear la siguiente matriz $3 * 10$ (no hay que escribir el vector (1,2,3) 10 veces):

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

¿Cómo podemos obtener la matriz siguiente?

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

Hacer la multiplicación posición a posición de las dos matrices obtenidas.

Sesión 4 – Listas y *data frames*

- Listas
- *Data frames*
- Trabajando con *data frames*: *attach* y *detach*
- Factores
- Ejercicios propuestos

Listas (1)

Una lista es un objeto que consiste en una colección ordenada de objetos, llamados componentes. Estos componentes no tienen que ser necesariamente del mismo tipo o *mode*, y pueden ser vectores, *arrays* o nuevas listas. Las listas se construyen con la función *list*.

```
# Creamos una lista
> lst <- list(padre="Carlos", madre="María", num.hijos=3, edad.hijos=c(4,7,9))
> lst
$padre
[1] "Carlos"

$madre
[1] "María"

$num.hijos
[1] 3

$edad.hijos
[1] 4 7 9
```

Los elementos de una lista siempre están siempre numerados, y pueden ser accedidos mediante esos números. En este caso los elementos de la lista se indexan mediante dobles *claudators* (`[[]]`). Si `lst[[4]]` es un vector, entonces `lst[[4]][1]` es su primer elemento.

```
> lst[[1]]
[1] "Carlos"

> lst[[3]]
[1] 3

> lst[[4]][1]
[1] 4
```

Listas (2)

La función *length*, aplicada a una lista, devuelve el número de componentes "de primer nivel" que contiene.

```
> length(lst)
[1] 4
```

Los componentes de las listas también pueden tener nombres. En este caso, también nos podremos referir a ellos por su nombre además por su posición, ayudados del símbolo del dólar (\$). Esto es útil para no tener que recordar en qué posición está cada componente de la lista.

```
> lst$madre #Equivalente a lst[[2]]
[1] "María"
```

```
# Otra manera de hacerlo
> x <- "madre"
> lst[[x]] #Equivalente a lst[["madre"]]
[1] "María"
```

```
> lst$edad.hijos[2] #Equivalente a lst[[4]][2]
[1] 7
```

Hay que distinguir entre `lst[1]` y `lst[[1]]`. El primer comando devuelve una sublista. Mientras que el segundo # devuelve el primer componente de la lista.

```
> lst[1] #Si la lista tiene nombres, éstos se transfieren a la sublista.
```

```
$padre
[1] "Carlos"
```

```
> lst[[1]] #Se devuelve el primer elemento de la lista respetando su tipo.
[1] "Carlos"
```

Listas (3)

Los nombres de las listas se pueden añadir con posterioridad a la creación de la lista.

```
# Creamos la lista sin nombres
> lst <- list("Carlos", "María", 3, c(4,7,9))
# Le añadimos los nombres
> names(lst) <- c("padre","madre","num.hijos","edad.hijos")
```

```
# Podemos extender la lista de manera sencilla
> lst[5] <- c("Juan", "Pedro", "Ana")
# Le ponemos el nombre al nuevo elemento de la lista
> names(lst)[5] <- "nombre.hijos"
```

```
# Otra manera alternativa de hacerlo
> lst$nombre.hijos <- c("Juan", "Pedro", "Ana")
```

Para concatenar listas utilizaremos la función *c*, de la manera *c(lista1, lista2, ..., lista n)*.

Cuando nos referimos a los elementos de las listas por sus nombres podemos utilizar el número mínimo de letras que indentifique de manera única a los diferentes componentes. De esta manera, *lst\$padre* podría ser especificado como *lst\$p*.

Data frames

Un *data frame* es una lista de variables que tienen la misma longitud y que han de estar identificadas por un nombre único. Se crean mediante la función *data.frame*.

```
# Creamos el data frame
> df <- data.frame(Nombre=c("Pedro","María","José","Marta"),
                  Edad=c(27,34,40,39),
                  Poblacion=c("Zaragoza","Madrid","Valencia","Barcelona"),
                  Sexo=c("H","M","H","M"),
                  Casado=c(F,T,T,F))
> df #Para acceder a una variable en concreto lo haremos con el dólar ($), como en el caso de las listas.
```

	Nombre	Edad	Poblacion	Sexo	Casado
1	Pedro	27	Zaragoza	H	FALSE
2	María	34	Madrid	M	TRUE
3	José	40	Valencia	H	TRUE
4	Marta	39	Barcelona	M	FALSE

En este caso todas las columnas del *data frame* han de tener 4 elementos o un número múltiplo de 4. Aplicando el reciclaje, R extiende las columnas más cortas para igualarlas en longitud respecto a la más larga.

Las matrices y las listas también pueden ser transformadas a *data.frames*. En este caso cada columna de la matriz o elemento de la lista pasan a ser una columna del nuevo *data frame*.

Los *data frames* se extienden igual que las listas (*df\$nueva.col <- valores*)

Trabajando con *data frames*: *attach* y *detach*

A veces puede ser engorroso trabajar con *data frames*, debido a que para acceder a las variables hemos de escribir el nombre del *data frame* seguido de un dólar y del nombre de la variable. R nos permite hacer visible el contenido de un *data frame* o de una lista, de manera que no hemos de escribir su nombre cada vez que queremos acceder a una de sus variables. Esto lo hacemos con la instrucción *attach*. Para volver a hacer "invisible" el *data frame* utilizaremos *detach*.

```
# Hacemos el attach del data frame  
> attach(df)
```

```
# Vemos que el data frame ya está cargado en nuestro entorno.
```

```
> search()
```

```
[1] ".GlobalEnv"      "df"              "package:methods" "package:stats"   "package:graphics" "package:utils"
```

```
[7] "Autoloads"      "package:base"
```

```
> ls(2)
```

```
[1] "Casado"  "Edad"    "Nombre"  "Poblacion" "Sexo"
```

```
# Accedemos directamente a las variables del data frame
```

```
> Nombre #En este caso la variable ha sido transformada a factor
```

```
[1] Pedro María José Marta
```

```
Levels: José María Marta Pedro
```

```
# Dejamos de hacer "visible" el data frame con la instrucción detach.
```

```
> detach(df)
```

```
> search()
```

```
[1] ".GlobalEnv"      "package:methods" "package:stats"   "package:graphics" "package:utils"   "Autoloads"
```

```
[7] "package:base"
```

Factores

R dispone de un tipo especial de datos llamado factor. Los vectores de factores sirven principalmente para agrupar elementos de otros vectores de la misma longitud. Los factores se utilizan principalmente en fórmulas para realizar modelos.

```
> df$Sexo
[1] H M H M
Levels: H M
```

Los *levels* contienen las distintas etiquetas. A cada etiqueta diferente R le asigna un número identificador por orden alfabético. De esta manera, podríamos redefinir los factores como etiquetas de texto que tienen asociado un código interno.

```
# Para ver los levels de una variable factor utilizaremos la instrucción levels.
> levels(df$Sexo)
[1] "H" "M"
```

```
# Si queremos reordenarlos utilizaremos la instrucción relevel
> relevel(df$Sexo,2) #Le decimos que el 2º factor pase a la 1ª posición.
[1] H M H M
Levels: M H
```

```
# Para crear una variable factor, R dispone de la instrucción factor, a la que se le suele pasar un vector de
# caracteres
> a <- factor(c("a","b","c","b","a"))
> a
[1] a b c b a
Levels: a b c
```

Ejercicios propuestos

1. Crear una lista donde guardaremos información de 3 personas. La lista tendrá 4 componentes, que serán vectores: nombre (carácter), edad (numérico), sexo (factor, H – M), y una variable booleana que indica si esa persona ya trabaja.
2. Crear un *data frame* que guarde la misma información que la lista anterior. Primero crearlo partiendo de cero y después aprovechar que ya tenéis la lista para crearlo de una manera más directa. ¿Hay problemas al transformar un objeto *list* a un objeto *data frame*? ¿Hace falta especificar que el sexo es una variable factor? ¿De qué tipo es la variable "nombre"? Con la ayuda de la función *as.character* transformarla en un vector de caracteres.
3. Añadirle un nuevo vector a la lista que indique la profesión de cada persona. Hacer lo mismo con el *data frame*. Hacerlo todo sin rehacer ni la lista ni el *data frame* de nuevo. Aseguraros de que la nueva información tenga su *name* correspondiente.
4. Hacer un *attach* del *data frame*. Comprobar que sus columnas son accesibles sin hacer mención explícita al propio *data.frame*. Descargarlo con *detach*.
5. Practicar un poco con la lista y el *data.frame* que tenéis creados. ¿Podéis acceder al *data frame* con índices numéricos como si fuera una matriz? ¿Sabéis hacer una subselección del *data frame*? ¿Y de la lista?

Sesión 5 – Lectura de datos de ficheros externos

- La función `read.table()`
- La función `data()`
- La librería `foreign`

La función `read.table()`

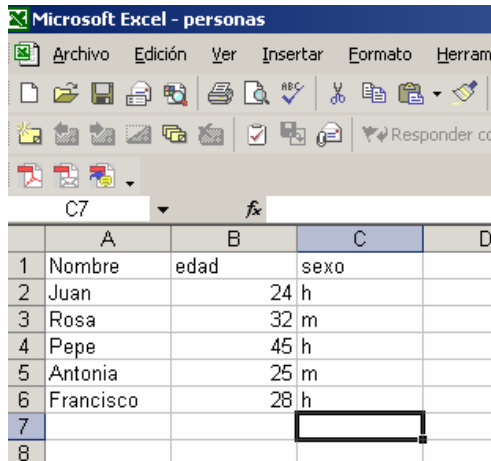
La función `read.table` permite leer datos desde ficheros en formato ASCII. Devuelve como resultado un `data.frame`, por tanto, se supone que cada línea contiene los datos para un individuo.

```
#El fichero thuesen.txt tiene el siguiente aspecto:  
"blood.glucose" "short.velocity"  
"1" 15.3 1.76  
"2" 10.8 1.34  
...  
"24" 9.5 1.7
```

```
> thuesen<-read.table(file="thuesen.txt",header=T)  
> thuesen  
  blood.glucose short.velocity  
1           15.3           1.76  
2           10.8           1.34  
...  
24            9.5           1.70
```

La función `read.table()`

El fichero EXCEL `personas.xls` tiene el siguiente aspecto:



The screenshot shows a Microsoft Excel window titled "Microsoft Excel - personas". The spreadsheet has four columns labeled A, B, and C, and rows numbered 1 through 8. The data is as follows:

	A	B	C	D
1	Nombre	edad	sexo	
2	Juan	24	h	
3	Rosa	32	m	
4	Pepe	45	h	
5	Antonia	25	m	
6	Francisco	28	h	
7				
8				

```
> #Guardamos el fichero EXCEL como un fichero ASCII delimitado por tabulaciones
```

```
> personas<-read.table(file="personas.txt",header=T)
```

```
> personas
```

```
      Nombre edad sexo
1      Juan   24    h
2      Rosa   32    m
3      Pepe   45    h
4  Antonia   25    m
5 Francisco   28    h
```

La función `read.table()`

El fichero `personas2.txt` contiene valores missing.

Nombre	edad	sexo
Juan	24	h
Rosa	32	m
Pepe	45	h
Antonia	NA	m
Francisco	28	h

```
#el parámetro na.string="NA" considera que por defecto NA o espacio en blanco indican missing
```

```
#podríamos añadir valores missing, na.string=c("NA","999","xx")
```

```
> personas2<-read.table(file="personas2.txt",header=T)
```

```
> personas2
```

	Nombre	edad	sexo
1	Juan	24	h
2	Rosa	32	m
3	Pepe	45	h
4	Antonia	NA	m
5	Francisco	28	h

La función data()

La función data() permite cargar en el workspace conjuntos de datos que ya existen en formato R y que forman parte de alguna de las librerías.

```
#para ver los datasets actualmente disponibles
```

```
> data()
```

```
> library(ISwR)
```

```
> library(MASS)
```

```
> data()
```

```
> ?airquality
```

```
> data(airquality)
```

```
New York Air Quality Measurements
```

```
Description:
```

```
  Daily air quality measurements in New York, May to September 1973.
```

```
Usage:
```

```
  data(airquality)
```

```
Format:
```

```
  A data frame with 154 observations on 6 variables.
```

```
  [,1] 'Ozone'    numeric Ozone (ppb)
  [,2] 'Solar.R'  numeric Solar R (lang)
  [,3] 'Wind'     numeric Wind (mph)
  [,4] 'Temp'    numeric Temperature (degrees F)
  [,5] 'Month'   numeric Month (1-12)
  [,6] 'Day'     numeric Day of month (1-31)
```

```
> airquality
```

```
   Ozone Solar.R Wind Temp Month Day
1     41     190  7.4  67     5    1
2     36     118  8.0  72     5    2
...
153    20     223 11.5  68     9   30
```

La librería foreign

Contiene funciones para importar datos que se encuentran en otros formatos.

```
#importamos el archivo coches.sav (SPSS)
> library(foreign)
> coches<-read.spss("coches.sav",to.data.frame=T)
> coches
```

	CONSUMO	MOTOR	CV	PESO	ACEL	A.O	ORIGEN	CILINDR	FILTER..
1	13	5031	130	1168	12.0	70	EE.UU.	8 cilindros	No Seleccionado
2	16	5735	165	1231	11.5	70	EE.UU.	8 cilindros	No Seleccionado
3	13	5211	150	1145	11.0	70	EE.UU.	8 cilindros	No Seleccionado
4	15	4982	150	1144	12.0	70	EE.UU.	8 cilindros	No Seleccionado

Existen otras funciones para leer datos en otros formatos: `read.dta()` para STATA, `read.mtp()` para MINITAB Export File, `read.ssd()` para SAS.

La función `write.table()`

Permite escribir los datos en un fichero ASCII.

```
#guardamos los datos de los coches en un fichero de texto  
> write.table(coches, file="coches.txt")
```

El fichero generado tiene el siguiente aspecto:

```
"CONSUMO" "MOTOR" "CV" "PESO" "ACEL" "A.O" "ORIGEN" "CILINDR" "FILTER.."
"1" 13 5031 130 1168 12.0 70 "EE.UU." "8 cilindros" "No Seleccionado"
"2" 16 5735 165 1231 11.5 70 "EE.UU." "8 cilindros" "No Seleccionado"
"3" 13 5211 150 1145 11.0 70 "EE.UU." "8 cilindros" "No Seleccionado"
"4" 15 4982 150 1144 12.0 70 "EE.UU." "8 cilindros" "No Seleccionado"
```

Ejercicios propuestos.

1. Explorar y cargar el dataset *energy* de la librería *ISwR*. Guardar los datos en un fichero de texto separado por tabulaciones. Editar el fichero creado con un editor de texto y añadir valores faltantes ("NA", "xx", ...). Leer el fichero de texto desde R.

Sesión 6 – Distribuciones de probabilidad

- Distribuciones de probabilidad
- Gráficos para distribuciones de probabilidad
- Generación de muestras aleatorias
- Estudio de la distribución de un conjunto de datos
- Estimación de densidades
- Ejercicios propuestos

Distribuciones de probabilidad (1)

R proporciona funciones que permiten obtener la función de densidad de probabilidad (o de probabilidad para variables discretas) y la función de distribución de probabilidad (acumulada), así como calcular cuantiles y simular la distribución.

Distribución	Nombre en R	Parámetros
beta	beta	shape1, shape2, ncp
binomial	binom	size, prob
Cauchy	cauchy	location, scale
ji-cuadrado	chisq	df, ncp
exponencial	exp	rate
F	f	df1,df2, ncp
gamma	gamma	shape, scale
geométrica	geom	prob
hipergeométrica	hyper	m, n, k
log-normal	lnorm	meanlog, sdlog
logística	logis	location, scale
binomial negativa	nbinom	size, prob
normal	norm	mean, sd
Poisson	pois	lambda
T de Student	t	df, ncp
uniforme	unif	min,max
Weibull	weibull	shape, scale
Wilcoxon	wilcos	m, n

Las funciones se obtienen añadiendo los prefijos *d* para la densidad, *p* para la distribución, *q* para la función cuantil y *r* para la generación aleatoria.

Ej. Para la distribución normal disponemos de cuatro funciones:

```
dnorm(x, mean=0, sd=1)
pnorm(q, mean=0, sd=1)
qnorm(p, mean=0, sd=1)
rnorm(n, mean=0, sd=1)
```

Distribuciones de probabilidad (2)

```

> # probabilidad que una binomial(10,0.2) tome el valor 2
> dbinom(2,size=10,prob=0.2)
[1] 0.3019899

> #probabilidad que una binomial(10,0.2) tome un valor inferior a 2
> pbinom(2,size=10,prob=0.2)
[1] 0.6777995

> #¿qué valor de una binomial(10,0.2) presenta una probabilidad acumulada de 0.9 ?
> qbinom(0.9,size=10,prob=0.2)
[1] 4

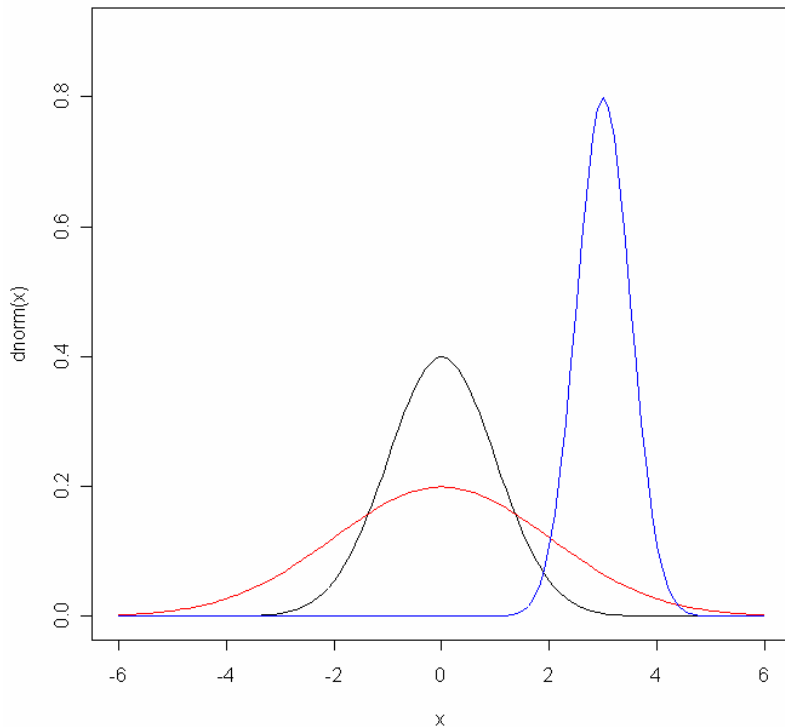
> #Generación de 20 valores aleatorios de una distribución binomial(10,0.2)
> rbinom(20,size=10,prob=0.2)
[1] 0 7 3 1 1 2 2 2 3 1 2 1 2 1 0 1 1 2 2 2

> #Función de probabilidad y de distribución de probabilidad de una binomial(10,0.2)
> x<-seq(0,10,by=1)
> data.frame(x,p=dbinom(x,size=10,prob=0.2),F=pbinom(x,size=10,prob=0.2))
  x      p      F
1  0 0.1073741824 0.1073742
2  1 0.2684354560 0.3758096
3  2 0.3019898880 0.6777995
4  3 0.2013265920 0.8791261
5  4 0.0880803840 0.9672065
6  5 0.0264241152 0.9936306
7  6 0.0055050240 0.9991356
8  7 0.0007864320 0.9999221
9  8 0.0000737280 0.9999958
10 9 0.0000040960 0.9999999
11 10 0.0000001024 1.0000000

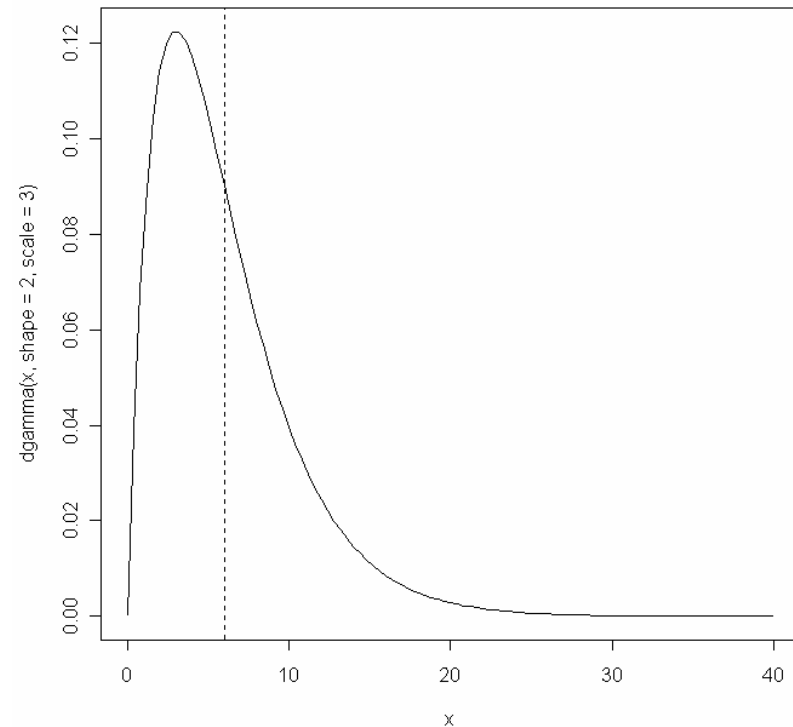
```

Gráficos para distribuciones de probabilidad (1)

```
> #Funciones de densidad de diferentes normales
> x<-seq(-6,6,by=0.1)
> plot(x,dnorm(x),type="l",xlim=c(-6,6),ylim=c(0,0.9))
> lines(x,dnorm(x,mean=0,sd=2),col="red")
> lines(x,dnorm(x,mean=3,sd=0.5),col="blue")
```

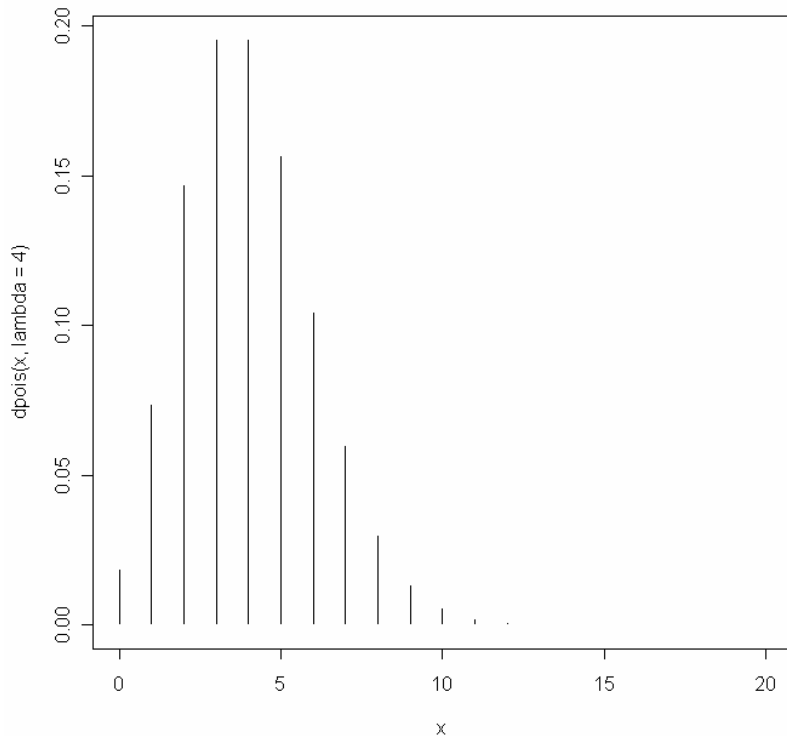


```
> #Funcion de densidad para una gamma
> x<-seq(0,40,by=0.01)
> curve(dgamma(x,shape=2,scale=3),from=0,to=40)
> abline(v=2*3,lty=2) #esperanza de la gamma
```

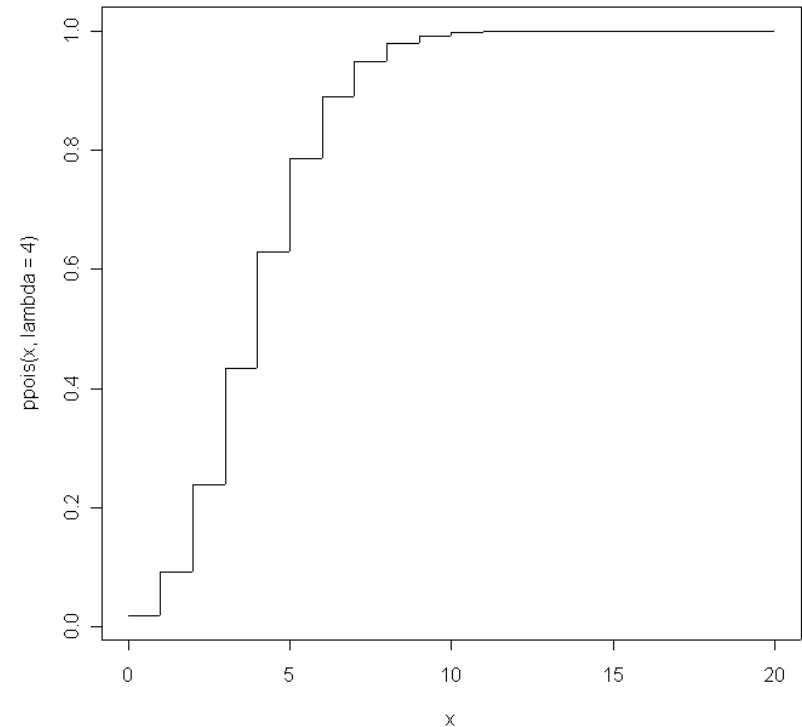


Gráficos para distribuciones de probabilidad (2)

```
> #Función de probabilidad para una poisson  
de media 4  
> x<-0:20  
> plot(x,dpois(x,lambda=4),type="h")
```



```
> #Función de distribución de probabilidad  
para una poisson de media 4  
> x<-0:20  
> plot(x,ppois(x,lambda=4),type="s")
```



Generación de números aleatorios

```
> #Generación de 100 muestras de tamaño 10 de una normal tipificada
> muestras<-matrix(rnorm(1000),nrow=100,byrow=T)
> medias<-apply(muestras,1,mean)
> var(medias)
[1] 0.1169421
```

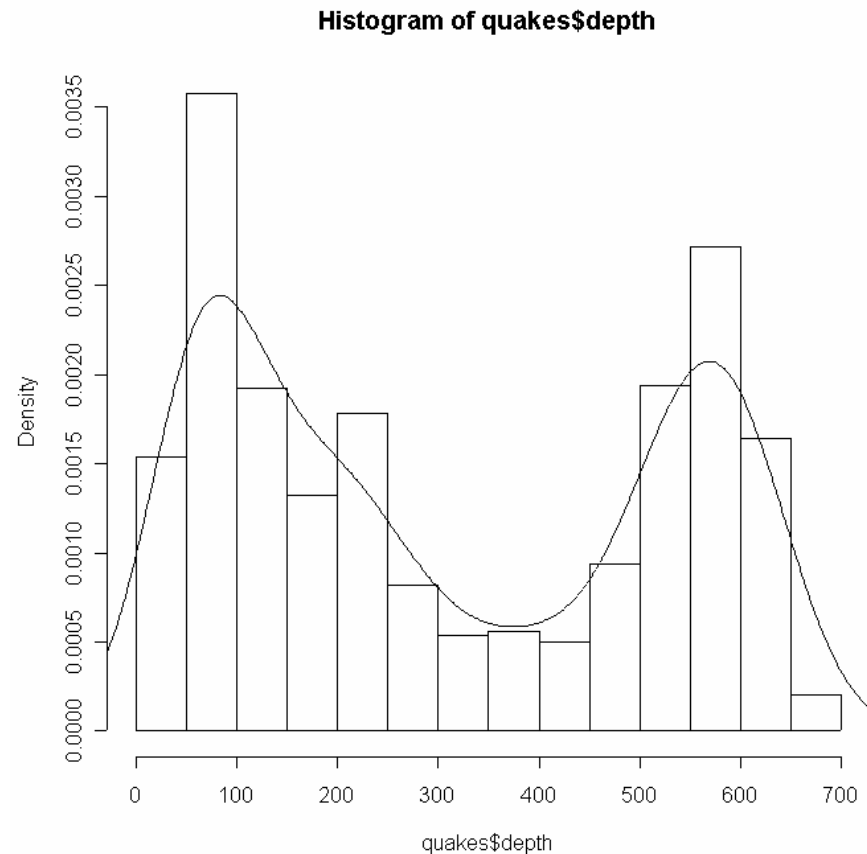
```
> #Generación de una variable factor segun una binomial
> x<-rbinom(20,size=1,prob=0.2)
> x
[1] 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0
> x<-factor(x)
> levels(x)<-c("fracaso","exito")
> table(x)
x
fracaso  exito
      17      3
```

```
> #Generación de una multinomial
> x<-sample(c("A","B","C"),200,replace=T,prob=c(0.5,0.4,0.1))
> table(x)
x
  A   B   C
104  79  17
```

Estudio de la distribución de un conjunto de datos

Dado un conjunto de datos, su histograma nos informa acerca de la distribución. Podemos emplear una estimación tipo kernel para estimar la densidad de probabilidad.

```
> #Estudio de la distribución de los
terremotos ocurridos cerca de Fiji
> data(quakes)
> hist(quakes$depth,prob=TRUE)
> lines(density(quakes$depth,bw=50))
```



Estimación de densidades (1)

La función `fitdistr` de la librería `MASS` permite obtener una estimación máximo-verosímil de distribuciones univariantes.

```
#Generación y estimación de una distribución
t de Student
```

```
> x<-rt(300,df=5)
```

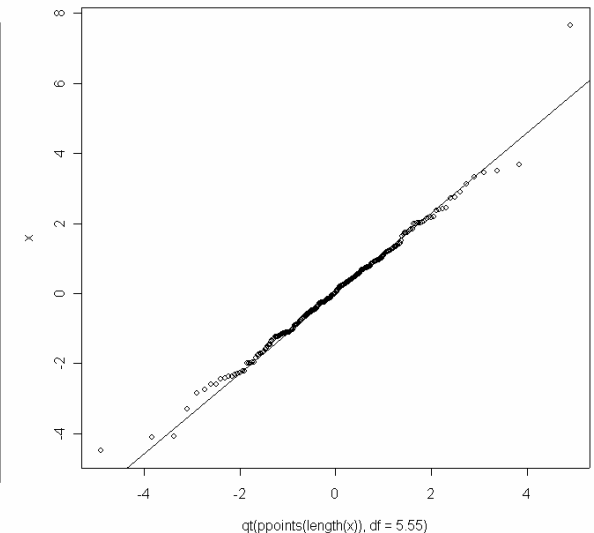
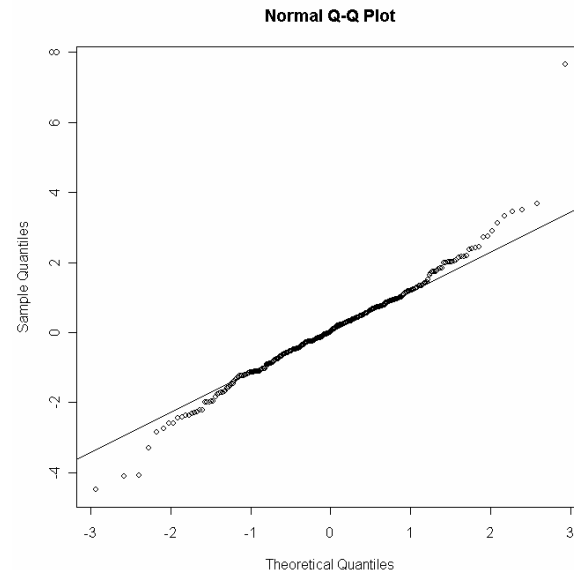
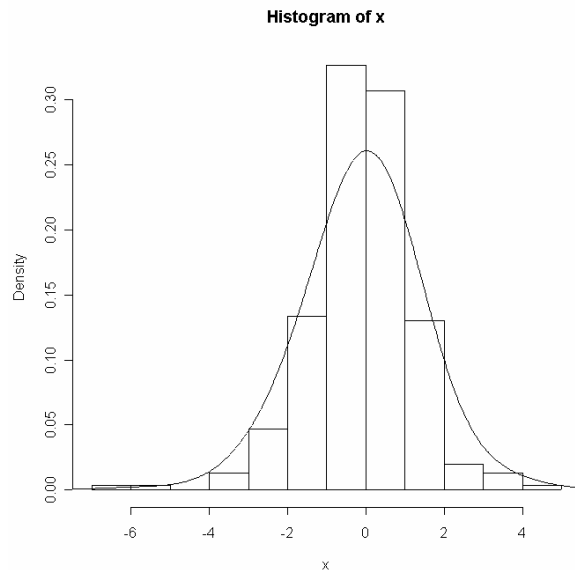
```
> fitdistr(x,"t")
```

m	s	df
0.03345688	1.09292734	5.55902260
(0.07209231)	(0.07572894)	(1.72774691)

```
> qqnorm(x); qqline(x)
```

```
> qqplot(qt(ppoints(length(x))),df=5.55),x)
```

```
> qqline(x)
```



Estimación de densidades (2)

La función `ks.test` permite resolver la prueba de Kolmogorov-Smirnov (suponiendo los parámetros conocidos)

```
> #Se distribuyen los datos anteriores según una t de student?
```

```
> ks.test(x, "pt", df=5.55)
```

```
One-sample Kolmogorov-Smirnov test
```

```
data: x
```

```
D = 0.0458, p-value = 0.5558
```

```
alternative hypothesis: two.sided
```

```
> #Comparación de dos muestras
```

```
> x <- rnorm(50)
```

```
> y <- runif(30)
```

```
> ks.test(x, y)
```

```
Two-sample Kolmogorov-Smirnov test
```

```
data: x and y
```

```
D = 0.52, p-value = 3.885e-05
```


Ejercicios propuestos

1. Calcular la probabilidad de: (a) que una normal tipificada sea mayor que 3. (b) que una variable normal con media 35 y desviación 6 sea mayor que 42. (c) obtener 10 éxitos en las 10 pruebas de una variable binomial con probabilidad de éxito 0.8. (d) una variable uniforme estándar (entre 0 y 1) tome un valor inferior a 0.9. (e) una variable ji-cuadrado con dos grados de libertad tome un valor superior a 6.5.
2. Calcular los intervalos de confianza al 90, 95 y 99% de una distribución normal de media 5 y desviación 2.
3. Generar 100 valores aleatorios de una distribución t con 10 grados de libertad. Suponed que los valores obtenidos provienen de la realización de una T-Test. Calcular los p-valores y estudiad su distribución. Repetir el análisis para una muestra de tamaño 1000.
4. Visualizar las funciones de densidad de diferentes distribuciones de ji-cuadrado con 5, 20 y 50 grados de libertad respectivamente.
5. Analizar la distribución del tiempo de las erupciones del géiser Old Faithful, del parque de Yelloswtone (datos `faithful`).
6. Analizar la normalidad del tiempo observado entre erupciones del géiser Old Faithful. Considerad la existencia de una mixtura de distribuciones normales.

Sesión 7 – Introducción a la estadística básica y gráficos

- Estadísticos resumen
- Gráficos para una variable
- Estadísticos resumen por grupos
- Gráficos para datos agrupados
- Tablas
- Gráficos para tablas

Estadísticos resumen (1)

Fácilmente se pueden calcular estadísticos sumario tipo media, mediana, desviación, ...

```
> x<-rnorm(50)
> mean(x)
[1] -0.2552258
> sd(x)
[1] 1.209657
> var(x)
[1] 1.463269
> median(x)
[1] -0.3365646

#cuantiles empíricos
> quantile(x)
      0%      25%      50%      75%      100%
-3.4542028 -1.1195259 -0.3365646  0.6758368  2.0094436
> pvec<-seq(0,1,0.1)
> pvec
[1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
> quantile(x,pvec)
      0%      10%      20%      30%      40%      50%      60%      70%      80%      90%      100%
-3.4542028 -1.7757077 -1.1735237 -0.9867830 -0.6923767 -0.3365646  0.0907579  0.5180732  0.9876644  1.3546966  2.0094436
```

Estadísticos resumen (2)

```
#exploramos el dataset juul
> library(ISwR)
> data(juul)
> ?juul
> attach(juul)
> mean(igf1)
[1] NA
```

Debemos indicarle que no tenga en cuenta los valores missing:

```
> mean(igf1,na.rm=T)
[1] 340.168
> sd(igf1,na.rm=T)
[1] 171.0356
```

```
#una excepción: la función length
> sum(!is.na(igf1))
[1] 1018
```

```
#directamente, función summary() sobre cualquier dataset
> summary(juul)
```

age	menarche	sex	igf1	tanner	testvol
Min. : 0.170	Min. : 1.000	Min. :1.000	Min. : 25.0	Min. : 1.000	Min. : 1.000
1st Qu.: 9.053	1st Qu.: 1.000	1st Qu.:1.000	1st Qu.:202.3	1st Qu.: 1.000	1st Qu.: 1.000
Median :12.560	Median : 1.000	Median :2.000	Median :313.5	Median : 2.000	Median : 3.000
Mean :15.095	Mean : 1.476	Mean :1.534	Mean :340.2	Mean : 2.640	Mean : 7.896
3rd Qu.:16.855	3rd Qu.: 2.000	3rd Qu.:2.000	3rd Qu.:462.8	3rd Qu.: 5.000	3rd Qu.: 15.000
Max. :83.000	Max. : 2.000	Max. :2.000	Max. :915.0	Max. : 5.000	Max. : 30.000
NA's : 5.000	NA's :635.000	NA's :5.000	NA's :321.0	NA's :240.000	NA's :859.000

Estadísticos resumen (3)

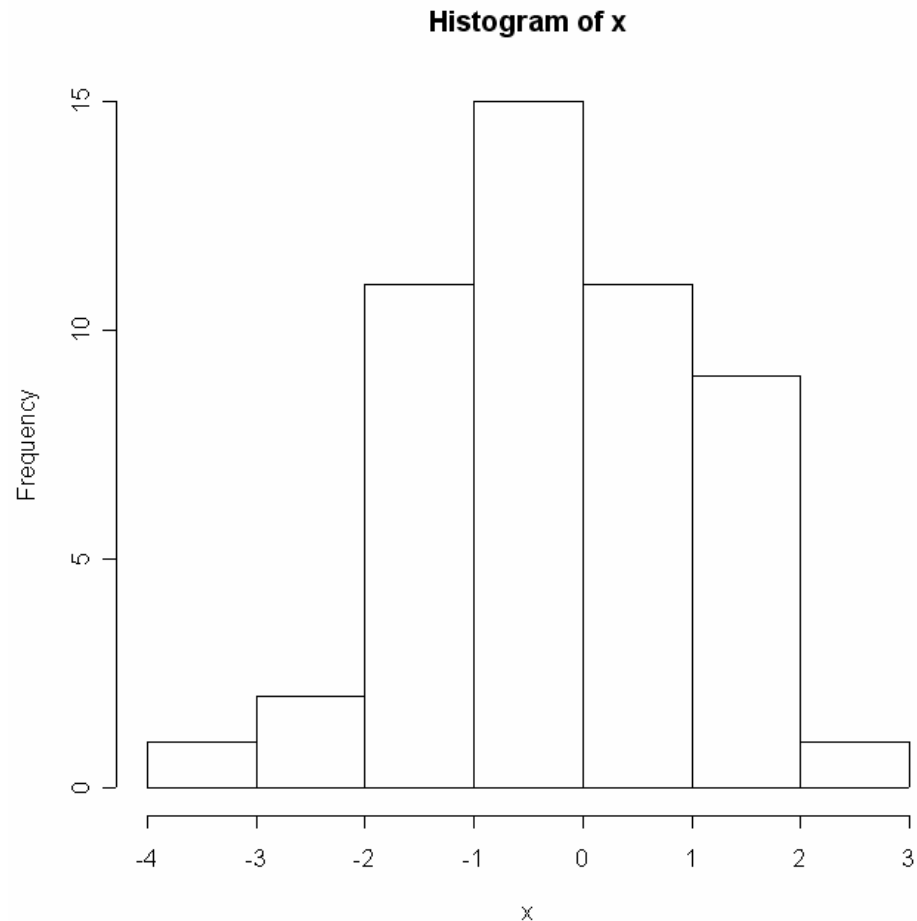
```
#en el data frame tenemos variables categóricas
> detach(juul)
> juul$sex<-factor(juul$sex,labels=c("M","F"))
> juul$menarche<-factor(juul$menarche,labels=c("No","Yes"))
> juul$tanner<-factor(juul$tanner,labels=c("I","II","III","IV","V"))
> attach(juul)
> summary(juul)
```

age	menarche	sex	igf1	tanner	testvol
Min. : 0.170	No :369	M :621	Min. : 25.0	I :515	Min. : 1.000
1st Qu.: 9.053	Yes :335	F :713	1st Qu.:202.3	II :103	1st Qu.: 1.000
Median :12.560	NA's:635	NA's: 5	Median :313.5	III : 72	Median : 3.000
Mean :15.095			Mean :340.2	IV : 81	Mean : 7.896
3rd Qu.:16.855			3rd Qu.:462.8	V :328	3rd Qu.: 15.000
Max. :83.000			Max. :915.0	NA's:240	Max. : 30.000
NA's : 5.000			NA's :321.0		NA's :859.000

```
#también podríamos haber utilizado la función transform()
> juul<-transform(juul,
+ sex=factor(sex,labels=c("M","F")),
+ menarche=factor(menarche,labels=c("No","Yes")),
+ tanner=factor(tanner,labels=c("I","II","III","IV","V")) )
```

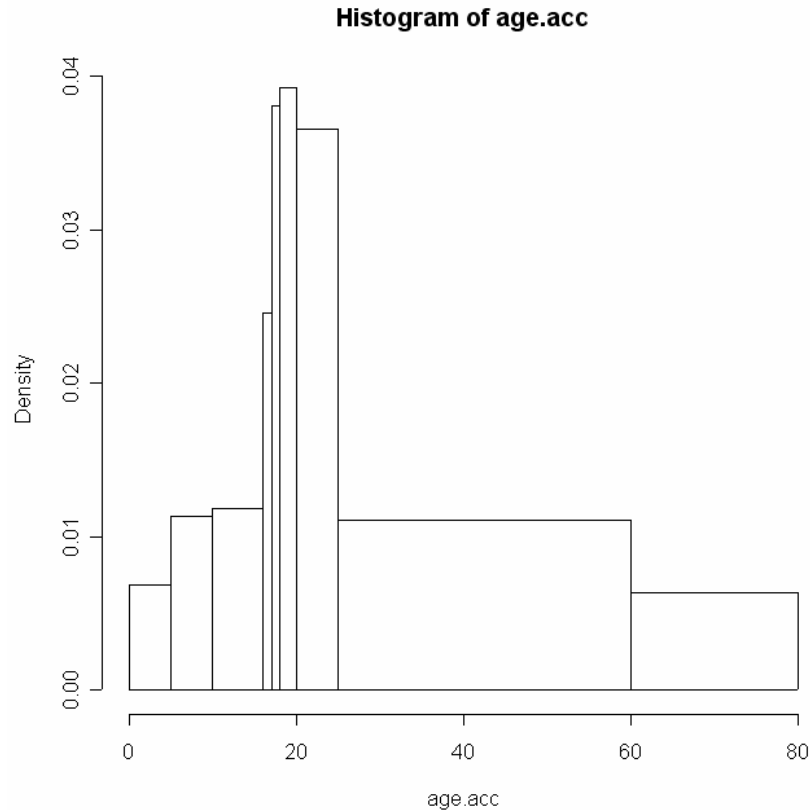
Gráficos para una variable (1)

#histogramas. Por defecto R, intenta hacer puntos de corte "adecuados"
> hist(x)



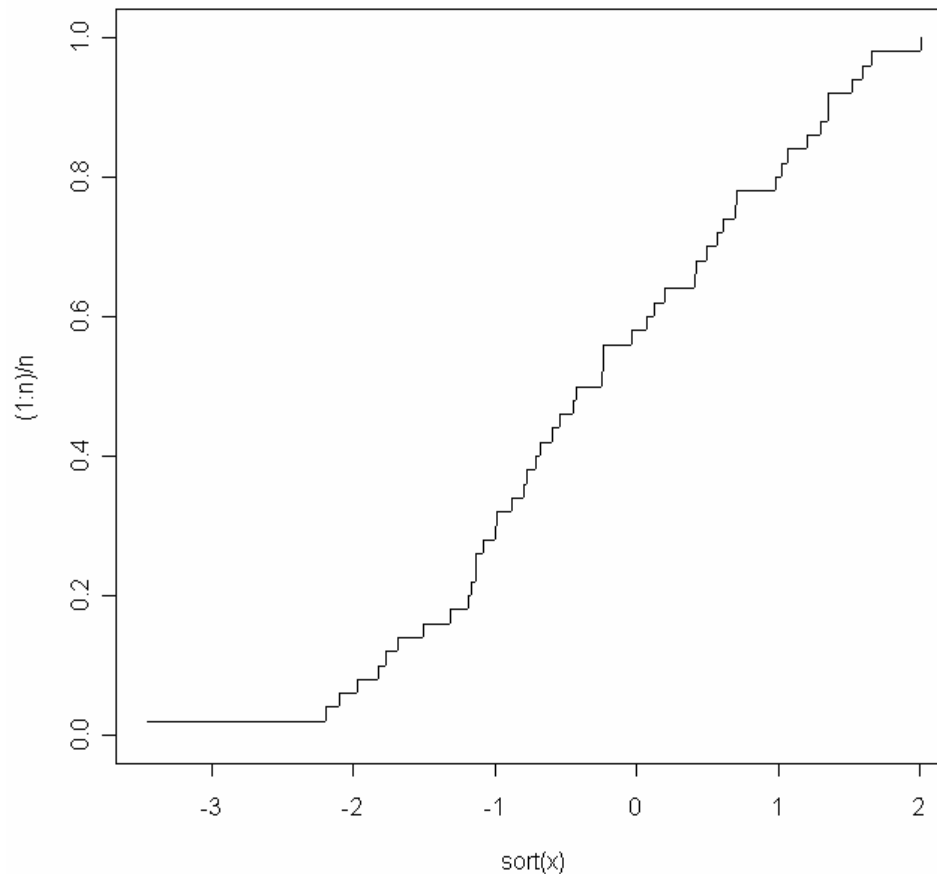
Gráficos para una variable (2)

```
#Ejemplo #accidentes vs edad (0-4,5-9,10-15,16,17,18-19,20-24,25-59,60-79)
> mid.age<-c(2.5,7.5,13,16.5,17.5,19,22.5,44.5,70.5)
> acc.count<-c(28,46,58,20,31,64,149,316,103)
> age.acc<-rep(mid.age,acc.count)
> brk<-c(0,5,10,16,17,18,20,25,60,80)
> hist(age.acc,breaks=brk)
```



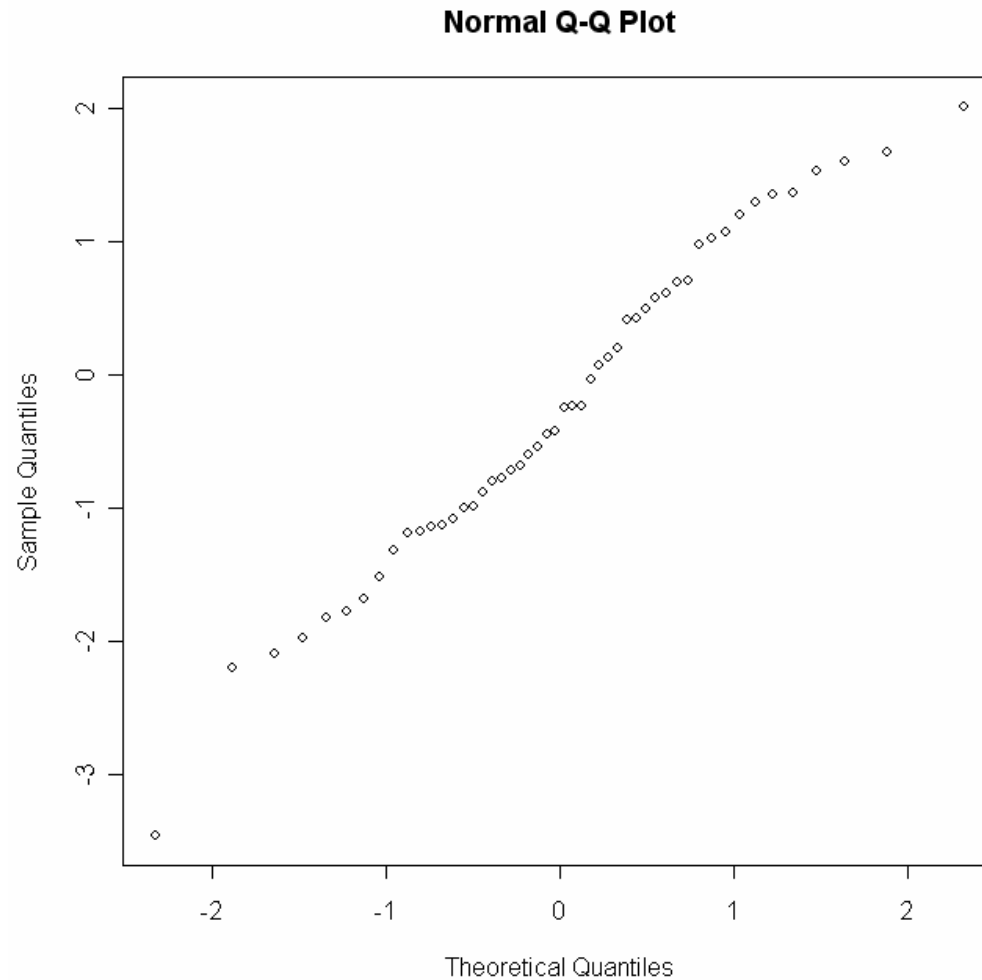
Gráficos para una variable (3)

```
#distribución empírica acumulada  
> n<-length(x)  
> plot(sort(x), (1:n)/n, type="s", ylim=c(0,1))
```



Gráficos para una variable (4)

```
#qqplot  
> qqnorm(x)
```



Gráficos para una variable (5)

```
#Boxplots IgM ( Serum IgM in 298 children aged 6 months to 6 years)
```

```
> data(IgM)
```

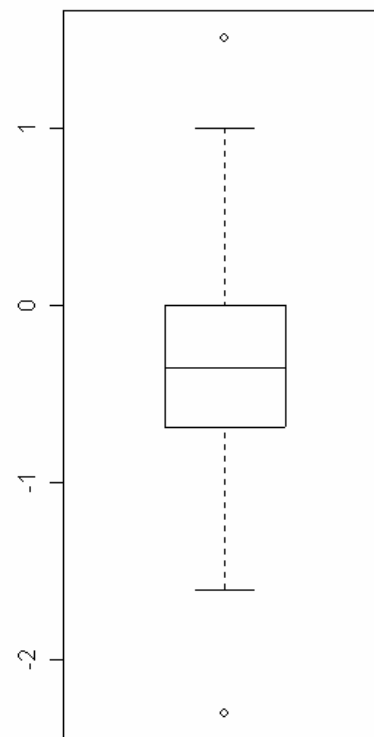
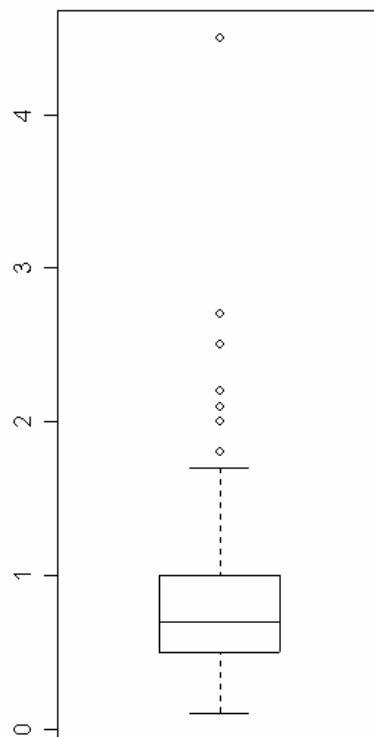
```
> ?IgM
```

```
> par(mfrow=c(1,2))
```

```
> boxplot(IgM)
```

```
> boxplot(log(IgM))
```

```
> par(mfrow=c(1,1))
```



Estadísticos resumen para grupos (1)

```
#Folate concentration in blood cells according to three types of ventilation during
anesthesia
> data(red.cell.folate)
> attach(red.cell.folate)
> ?red.cell.folate
> summary(red.cell.folate)
      folate      ventilation
Min.   :206.0   N2O+O2,24h:8
1st Qu.:249.5   N2O+O2,op  :9
Median :274.0   O2,24h     :5
Mean   :283.2
3rd Qu.:305.5
Max.   :392.0
> tapply(folate,ventilation,mean)
N2O+O2,24h  N2O+O2,op    O2,24h
  316.6250   256.4444   278.0000

> #Para tener más de un estadístico resumen por grupo
> m<-tapply(folate,ventilation,mean)
> s<-tapply(folate,ventilation,sd)
> n<-tapply(folate,ventilation,length)
> cbind(mean=m,std.dev=s,n=n)
      mean  std.dev n
N2O+O2,24h 316.6250 58.71709 8
N2O+O2,op  256.4444 37.12180 9
O2,24h     278.0000 33.75648 5
```

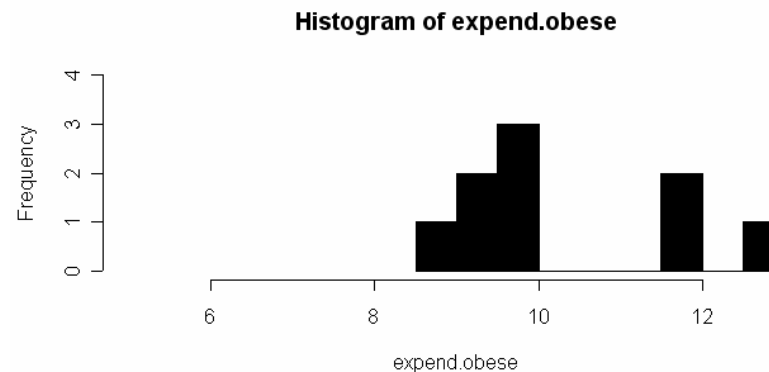
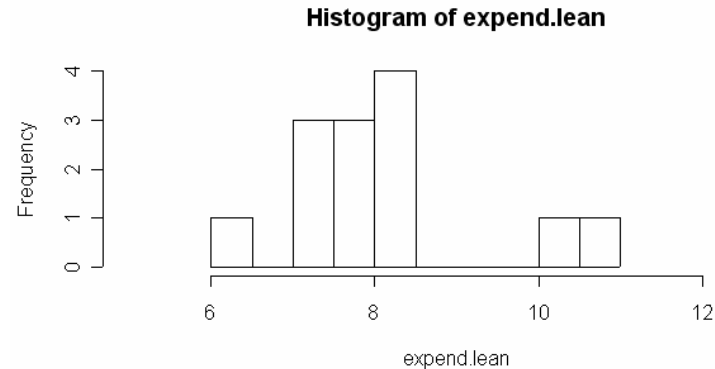
Estadísticos resumen para grupos (2)

```
#para el dataset juul
> tapply(igf1,tanner,mean)
  I  II III  IV  V
NA NA NA NA NA
> tapply(igf1,tanner,mean,na.rm=T)
  I      II      III      IV      V
207.4727 352.6714 483.2222 513.0172 465.3344
```

Gráficos para datos agrupados (1)

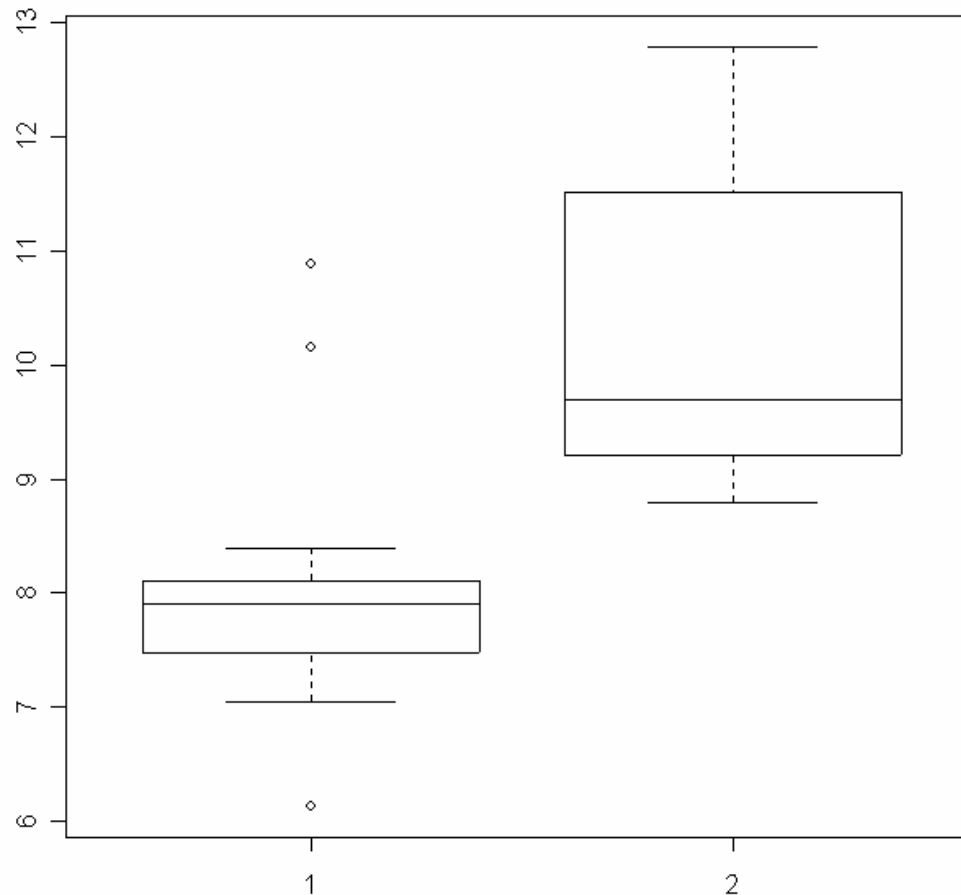
```
#cargamos el dataset energy
> data(energy)
> attach(energy)
> summary(energy)
      expend      stature
Min.   : 6.130   lean  :13
1st Qu.: 7.660   obese: 9
Median : 8.595
Mean   : 8.979
3rd Qu.: 9.900
Max.   :12.790
> ?energy
```

```
#histogramas para cada grupo de mujeres
> expend.lean<-expend[stature=="lean"]
> expend.obese<-expend[stature=="obese"]
> par(mfrow=c(2,1))
> hist(expend.lean,breaks=10,xlim=c(5,13),ylim=c(0,4),col="white")
> hist(expend.obese,breaks=10,xlim=c(5,13),ylim=c(0,4),col="black")
> par(mfrow=c(1,1))
```



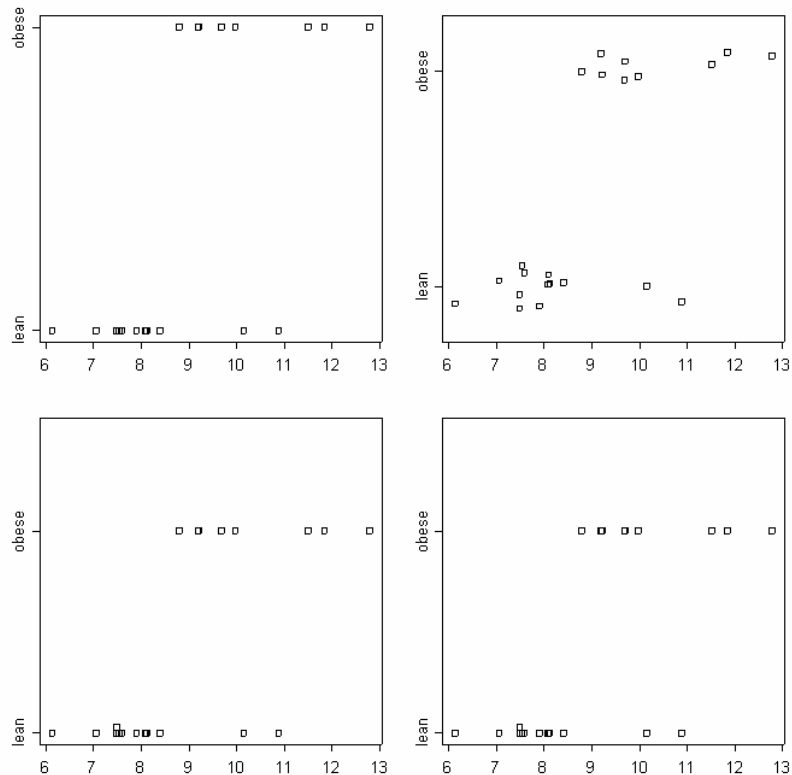
Gráficos para datos agrupados (2)

```
#boxplots para cada grupo  
> boxplot(expend~stature)  
> boxplot(expend.lean, expend.obese)
```



Gráficos para datos agrupados (3)

```
#con muestras tan pequeñas, los boxplots pueden resultar engañosos
#gráficos de los datos originales, punto a punto
> opar<-par(mfrow=c(2,2),mex=0.8,mar=c(3,3,2,1)+0.1)
> stripchart(expend~stature)
> stripchart(expend~stature,method="jitter")
> stripchart(expend~stature,method="stack")
> stripchart(expend~stature,method="stack",jitter=0.03)
> par(opar)
```



Tablas (1)

```
#Una tabla debe estar en un objeto tipo matriz
#Ejemplo mujeres consumo cafeína vs estado civil
> caff.marital<-matrix(c(652,1537,598,242,36,46,38,21,218,327,106,67),nrow=3,byrow=T)
> caff.marital
```

```
      [,1] [,2] [,3] [,4]
[1,]  652 1537  598  242
[2,]   36  46  38  21
[3,]  218 327 106  67
> colnames(caff.marital)<-c("0","1-150","151-300",>300")
> rownames(caff.marital)<-c("Married","Prev.married","Single")
> caff.marital
```

	0	1-150	151-300	>300
Married	652	1537	598	242
Prev.married	36	46	38	21
Single	218	327	106	67

```
#también podemos crearla a partir de variables categóricas de un dataset
table(sex)
```

```
sex
 M  F
621 713
> table(sex,menarche)
```

```
menarche
sex No  Yes
 M  0  0
 F 369 335
```

```
> table(menarche,tanner)
```

```
tanner
menarche I  II  III IV  V
 No  221  43  32  14  2
 Yes   1   1   5  26 202
```


Tablas (2)

```
#podemos transponer las tablas
```

```
> t(caff.marital)
      Married Prev.married Single
0      652          36      218
1-150  1537         46      327
151-300 598         38      106
>300    242         21       67
```

```
#para calcular las frecuencias marginales, perfiles fila, .> prop.table(tanner.sex,1)
```

```
> tanner.sex<-table(tanner,sex)
```

```
> tanner.sex
```

```
      sex
tanner M  F
  I    291 224
  II   55  48
  III  34  38
  IV   41  40
  V   124 204
```

```
> margin.table(tanner.sex,1)
```

```
tanner
  I  II III  IV  V
515 103  72  81 328
```

```
> margin.table(tanner.sex,2)
```

```
sex
  M  F
545 554
```

```
      sex
tanner M      F
  I    0.5650485 0.4349515
  II   0.5339806 0.4660194
  III  0.4722222 0.5277778
  IV   0.5061728 0.4938272
  V    0.3780488 0.6219512
```

```
> prop.table(tanner.sex,1)*100
```

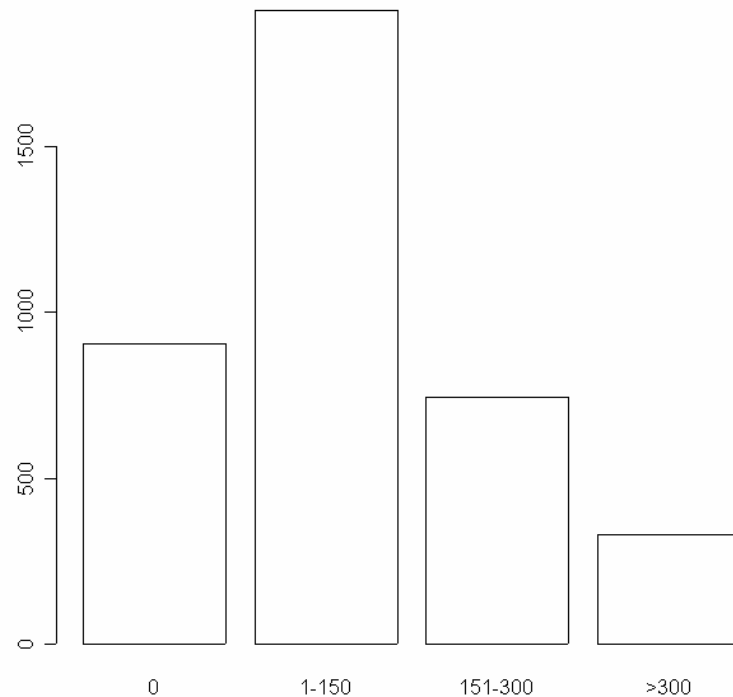
```
      sex
tanner M      F
  I    56.50485 43.49515
  II   53.39806 46.60194
  III  47.22222 52.77778
  IV   50.61728 49.38272
  V    37.80488 62.19512
```

```
> tanner.sex/sum(tanner.sex)
```

```
      sex
tanner M      F
  I    0.26478617 0.20382166
  II   0.05004550 0.04367607
  III  0.03093722 0.03457689
  IV   0.03730664 0.03639672
  V    0.11282985 0.18562329
```

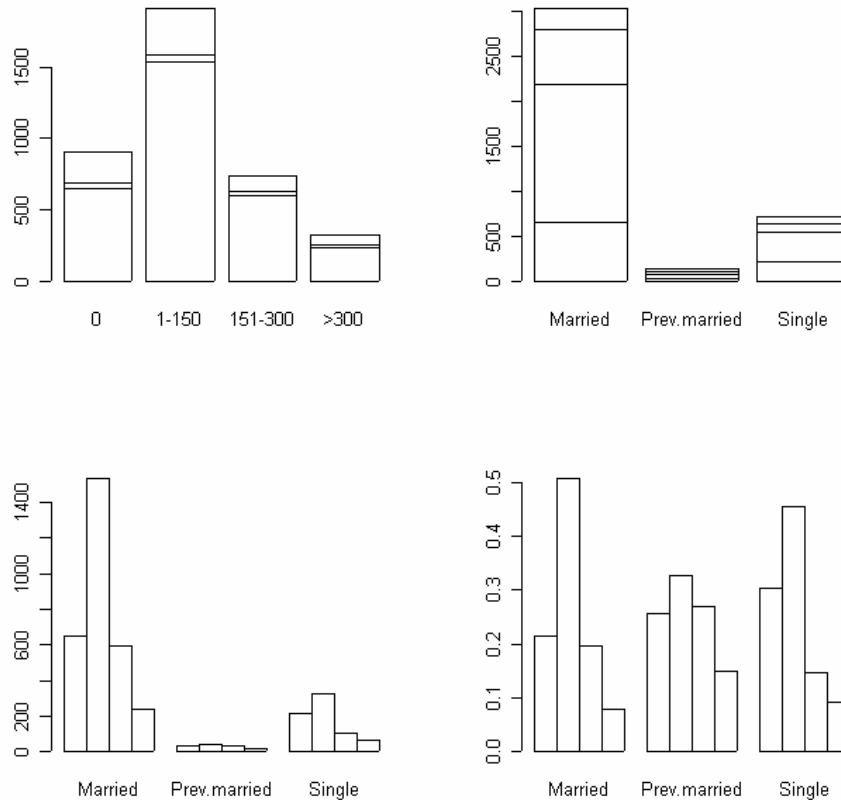
Gráficos para tablas (1)

```
#diagrama de barras  
> total.caff<-margin.table(caff.marital,2)  
> total.caff  
      0    1-150 151-300    >300  
906   1910    742    330  
> barplot(total.caff,col="white")
```



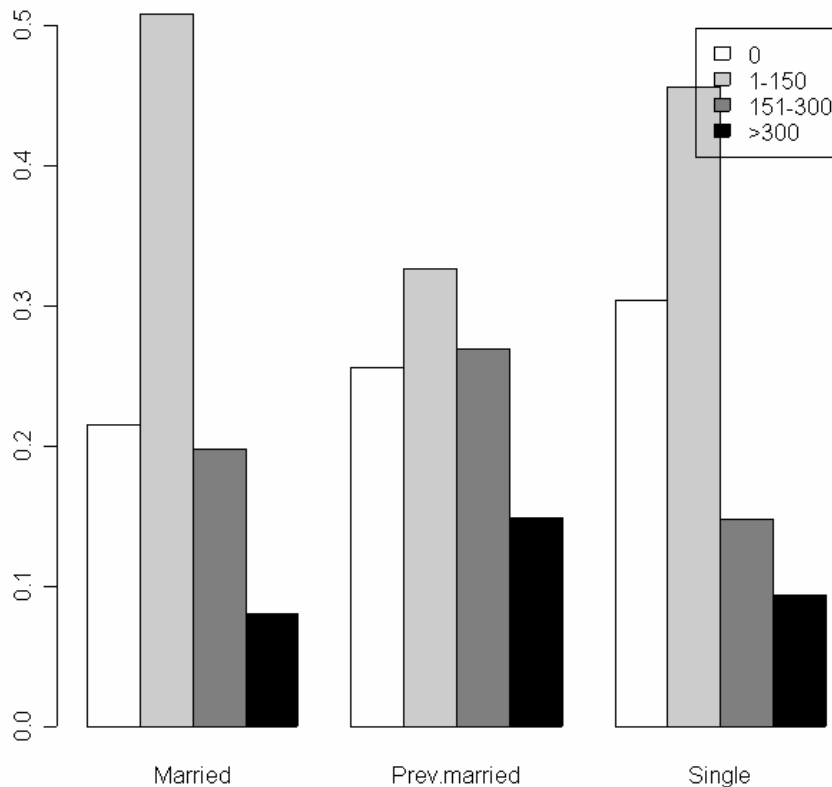
Gráficos para tablas (2)

```
#diagramas de barras para una tabla de contingencia
> par(mfrow=c(2,2))
> barplot(caff.marital,col="white")
> barplot(t(caff.marital),col="white")
> barplot(t(caff.marital),col="white",beside=T)
> barplot(prop.table(t(caff.marital),2),col="white",beside=T)
> par(mfrow=c(1,1))
```



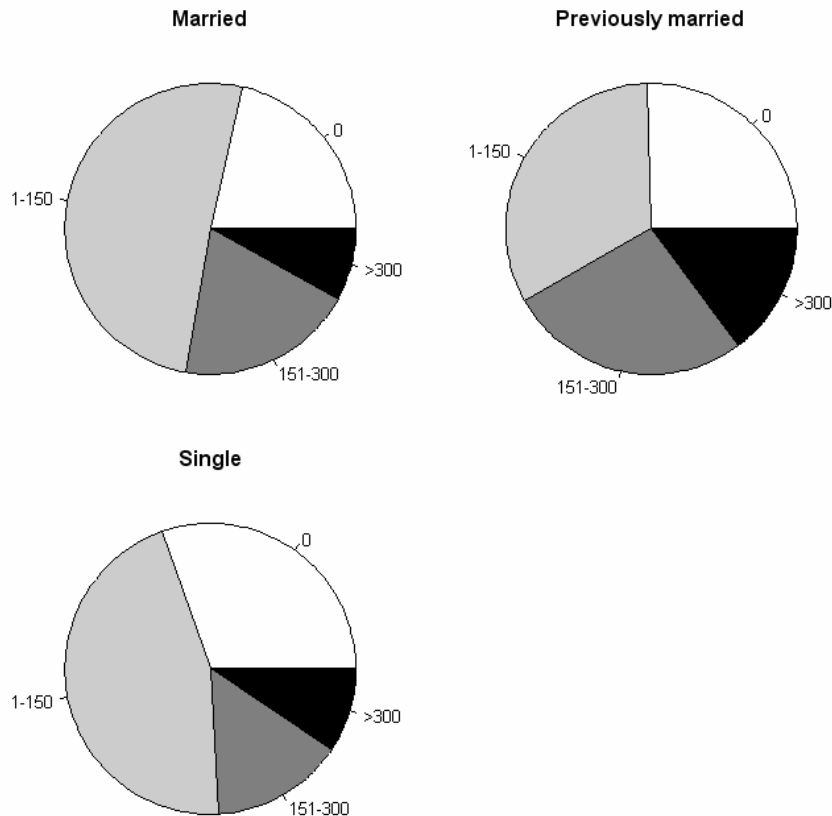
Gráficos para tablas (3)

```
#otro diagrama de barras para una tabla de contingencia  
> barplot(prop.table(t(caff.marital),2),beside=T,  
+         legend.text=colnames(caff.marital),  
+         col=c("white","grey80","grey50","black"))
```



Gráficos para tablas (4)

```
#diagrama de sectores para una tabla de contingencia
> opar<-par(mfrow=c(2,2),mex=0.8,mar=c(1,1,2,1))
> slices<-c("white","grey80","grey50","black")
> pie(caff.marital["Married",],main="Married",col=slices)
> pie(caff.marital["Prev.married",],main="Previously married",col=slices)
> pie(caff.marital["Single",],main="Single",col=slices)
> par(opar)
```



Ejercicios propuestos.

1. Explorar el dataset *bp.obese* de la librería *ISwR*.
2. Explorar el dataset *Titanic*. Analizar pares de variables categóricas.

Sesión 8 – Programación en R

- Agrupación de expresiones
- Ejecución condicional: *if*
- Ejecución repetitiva (bucles): *for*, *repeat* y *while*
- Ejemplos de bucles
- Creación de nuevas funciones
- Un ejemplo de función simple
- Definición de nuevos operadores binarios
- Nombres de argumentos y valores por defecto
- Ejercicios propuestos

Agrupación de expresiones

R es un lenguaje de expresiones, en el sentido que el único tipo de comandos que acepta es una función o una expresión que retorna un resultado. Incluso una asignación es una expresión que devuelve el valor asignado.

Los comandos pueden ser agrupados mediante llaves, `{expr_1; ...; expr_m}`, en este caso el valor que retorna el grupo es el resultado de la última expresión contenida en él.

Como un grupo es una expresión en sí mismo, también podría ser utilizado como parte de expresiones más grandes.

```
# Grupo de expresiones  
> {a<-0; b<-1; c<-2}
```

```
# Utilizamos un grupo de expresiones dentro de otra expresión  
> d <- {a<-0; b<-1; c<-2}  
> d  
[1] 2
```


Ejecución condicional: *if*

R dispone de una expresión condicional, que se construye de la forma:

$$\textit{if} (\textit{expr}_1) \textit{expr}_2 [\textit{else} \textit{expr}_3]$$

expr_1 debe tener como resultado un valor lógico (TRUE/FALSE). Cuando *expr_1* es cierto (TRUE), la expresión devuelve *expr_2*. En caso contrario se devuelve *expr_3*. Si *expr_2* o *expr_3* son grupos de expresiones han de ir entre llaves (`{}`).

```
> a <- 5
> if (a < 10) b<-a+3 else b<-a-3
> b
[1] 8
```

Los operadores `&&` (*and*) y `||` (*or*) son habitualmente utilizados en las expresiones condicionales. Una vez es evidente el resultado de la expresión lógica el resto se deja de evaluar.

R dispone de una versión vectorizada del *if/else*, la función *ifelse*:

$$\textit{ifelse}(\textit{condición}, a, b)$$

Devuelve un vector de la misma longitud que *condición*, con los elementos `a[i]` si `condición[i]` es cierto, y `b[i]` si `condición[i]` es falso.

Ejecución repetitiva (bucles): *for*, *repeat*, *while*

A menudo es necesario repetir la ejecución de un mismo grupo de instrucciones (p. ej. para cada elemento de un vector, un número determinado de veces o mientras se cumpla una determinada condición). R nos ofrece diversas alternativas:

```
for (var in expr_1) expr_2
```

expr_2 suele ser una expresión agrupada con subexpresiones escritas en función de la variable de bucle *var*, y se ejecutará tantas veces como elementos tiene *expr_1* (que suele ser del aspecto 1:20)

```
while (condición) expr
```

expr se ejecuta mientras *condición* sea cierto (TRUE).

```
repeat expr
```

expr se ejecuta mientras nosotros no interrumamos explícitamente el bucle con la instrucción *break*. Los bucles *for* y *while* también se pueden interrumpir con *break* (aunque no es lo más deseable).

Ejemplos de bucles

A modo de ejemplo, y aunque R nos ofrece maneras mucho más óptimas de realizar esta operación, vamos a hacer un bucle que coja los vectores *a* y *b*, los sume posición a posición, y guarde el resultado en el vector *res*. Lo haremos con *for*, *while* y *repeat*.

```
# Vectores a sumar (son de la misma longitud)
> a <- 1:10
> b <- 11:20
# Vector resultado: lo tenemos que crear porque si no R no puede hacer la primera asignación.
> res <- rep(0,length(a))
```

```
# Ejemplo con for
> for (i in 1:length(a)) res[i]<-a[i]+b[i]
```

```
#Ejemplo con while
> i <- 1
> while (i <= length(a)) {res[i]<-a[i]+b[i]; i<-i+1}
```

```
#Ejemplo con repeat
> i <- 1
> repeat {if (i<=length(a)) {res[i]<-a[i]+b[i]; i<-i+1} else break}
```

El hecho de que R disponga de cálculo vectorial nos ahorrará muchos bucles. Debido al uso de la memoria que hace R, debemos acostumbrarnos a utilizar este tipo de instrucciones lo menos posible. Más adelante veremos otras funciones que nos ayudarán aún más a no utilizarlos

Creación de nuevas funciones

Hasta ahora hemos visto el lenguaje R como un conjunto de expresiones "sueltas". Ahora bien, muchas veces nos interesará agruparlas bajo un determinado nombre y parametrizarlas. Uno de los tipos de objetos que R permite crear son las funciones. Mediante la creación de funciones, el lenguaje gana enormemente en versatilidad, conveniencia y elegancia. Así, aprender a escribir funciones útiles es una de las principales maneras de hacer del uso de R algo confortable y productivo. Como en la mayoría de los otros lenguajes de programación, las asignaciones dentro de funciones son temporales y se pierden cuando acaba su ejecución.

Muchas de las funciones ya incluidas dentro de R están implementadas en este mismo lenguaje, por lo tanto su implementación no difiere mucho de la implementación de las nuestras propias.

Las funciones se definen siguiendo el siguiente patrón:

$$\text{nombre} \leftarrow \text{function}(\text{arg}_1, \text{arg}_2, \dots, \text{arg}_n) \text{ expr}$$

expr suele ser una expresión agrupada que usa los argumentos *arg_i* para calcular un valor. El valor de la expresión es el valor retornado por la función.

Las llamadas a funciones suelen tener el siguiente aspecto:

$$\text{nombre}(\text{expr}_1, \text{expr}_2, \dots, \text{expr}_n)$$

Un ejemplo de función simple.

Vamos a crear una función que, dados los tres coeficientes (a , b , c) de una ecuación de segundo grado, $ax^2 + bx + c = 0$, nos devuelva las dos soluciones, si existen en el conjunto de los reales. Recordemos que la fórmula para encontrar las dos soluciones es:

$$(-b \pm \sqrt{b^2 - 4ac}) / 2a$$

```
# Cabecera de la función: le damos un nombre y definimos los tres parámetros
# Como vamos a utilizar más de una instrucción para programar la función, ya añadimos con la llave el inicio
# del grupo de instrucciones
ec_grado2 <- function(a, b, c)
{
  # Calculamos el discriminante (b^2 - 4ac)
  disc <- (b^2)-(4*a*c)
  # Primera solución
  sol1 <- (-b + sqrt(disc))/(2*a)
  # Segunda solución
  sol2 <- (-b - sqrt(disc))/(2*a)
  # Devolvemos un vector con las dos soluciones y cerramos la llave para indicar el final del grupo de
  # instrucciones que forman la función
  c(sol1,sol2)
}
```

Definición de nuevos operadores binarios.

Para funciones que tengan exclusivamente dos parámetros, R permite definir las de manera que después se puedan llamar con notación infija, es decir, poniendo el nombre de la función entre los dos argumentos. Un ejemplo de una función típicamente infija es la suma (escribimos $3+7$ y no $+(3,7)$ o $\text{suma}(3,7)$).

Para crear una función con notación infija simplemente tenemos que añadir el carácter % delante y detrás del nombre de la función. R ya interpretará de esta manera que tipo de función estamos creando.

```
# Vamos a crear una función que reciba dos numeros y devuelva la división del primero entre la suma de los
# dos. Nótese que es obligatorio que el nombre de la función vaya entre comillas.
> "%porcentaje%" <- function(a,b) (a/(a+b))*100
# Ejecutamos la función
> 4 %porcentaje% 6
[1] 40
```

Algunos ejemplos de este tipo de funciones son el producto matricial `%*%`, o la inclusión en listas `%in%`.

Nombres de argumentos y valores por defecto.

Al contrario que la gran mayoría de lenguajes de programación, R permite colocar los argumentos de una llamada a una función en el orden que nosotros queramos. Para poder hacer esto tenemos que especificar para cada valor en la llamada a qué argumento corresponde. En caso que no lo hagamos entonces R sí entenderá que los argumentos de la llamada están en el mismo orden en que han sido especificados en la declaración de las funciones.

```
# Vamos a estudiar los argumentos de la función rnorm
```

```
> args(rnorm)
```

```
function (n, mean = 0, sd = 1)
```

```
# La llamamos sin poner los nombres de los argumentos
```

```
> rnorm(10,1,2)
```

```
# Ahora ponemos los nombres de los argumentos: podemos cambiar su orden
```

```
> rnorm(sd=2,n=10,mean=1)
```

Vemos que los parámetros *mean* y *sd* tienen asociados valores por defecto (0 y 1 respectivamente). En este caso, si aceptamos estos valores por defecto como los deseados para ejecutar la función, podemos obviarlos. Una vez prescindamos de un parámetro, sin embargo, deberemos escribir el nombre de todos los que vengan detrás de él.

```
# Prescindimos de cambiar mean, pero tenemos que especificar que el siguiente es sd.
```

```
> rnorm(10,sd=3)
```

Ejercicios propuestos

1. Ampliar la función que calcula las soluciones de una ecuación de segundo grado. Poner el valor 1 por defecto en los tres parámetros. Adaptarla al siguiente algoritmo:
 - 1) Si $a=b=c=0$ -> Devolver un texto donde se diga que la ecuación tiene infinitas soluciones (para devolver mensajes utilizar la función *cat*).
 - 2) Si $a=b=0 \neq c$ -> Devolver un texto donde se diga que la ecuación es incorrecta
 - 3) Si $a=0 \neq b$ -> Devolver sólo una solución, que es $-c/b$
 - 4) Si $a \neq 0$ -> Aplicar la fórmula original. Calcular el discriminante d :
 - 4.1) Si $d=0$ -> Devolver una sola solución
 - 4.2) Si $d < 0$ -> Devolver un mensaje que diga que la ecuación no tiene soluciones reales
 - 4.3) Si $d > 0$ -> Devolver las dos soluciones de la ecuación
2. Utilizando la función anterior (y sin modificarla), crear otra función que reciba tres vectores de números de la misma longitud (v_a, v_b, v_c) y para cada trío de números llame a la función del ejercicio 1 y calcule las soluciones a la ecuación de segundo grado. Controlar que la longitud de los tres vectores sea igual, en caso negativo dar un error.

Sesión 9 – Procedimientos gráficos en R

- Procedimientos gráficos
- Comandos gráficos de alto nivel
- La función *plot*
- Parámetros de los comandos de alto nivel
- Comandos gráficos de bajo nivel
- Interactuando con los gráficos
- Otros comandos de alto nivel
- Ejercicios propuestos

Procedimientos gráficos

Las funciones gráficas son un componente importante y extremadamente versátil del entorno R.

La función para abrir una ventana gráfica en R es *X11* (válida para Windows y Linux).

En R existen tres tipos de comandos gráficos:

- Alto nivel: crean un nuevo gráfico, probablemente con ejes, etiquetas, títulos, ...
- Bajo nivel: añaden más información a un gráfico ya existente, como puntos, líneas, leyendas, etiquetas...
- Interactivos: permiten añadir o extraer información del gráfico de manera interactiva, usando un dispositivo apuntador como podría ser el ratón.

Comandos gráficos de alto nivel

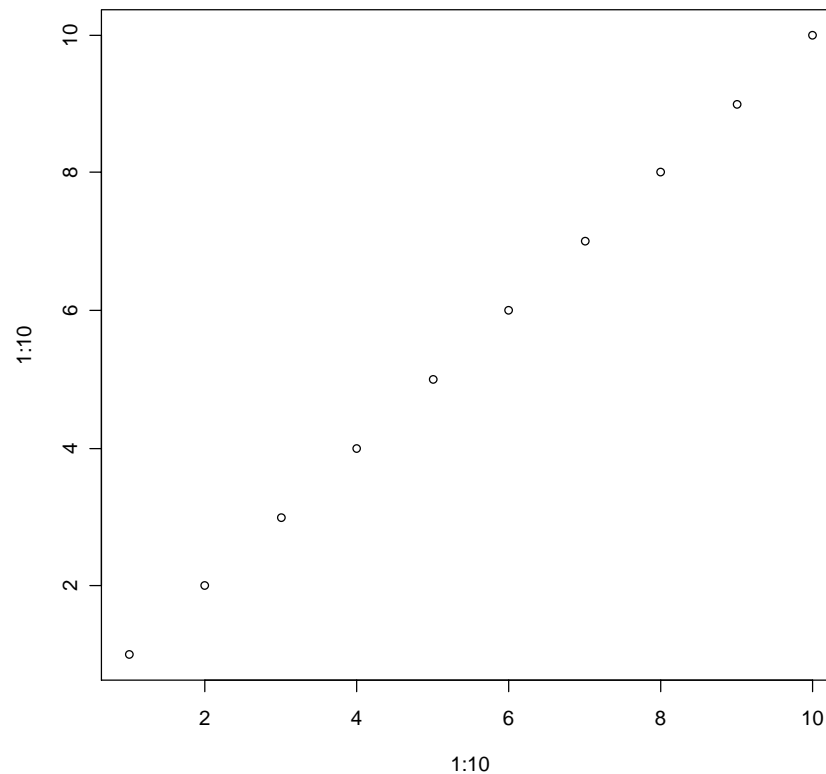
- Designados para generar un gráfico completo de los datos que se le pasan como parámetro.
- Suelen generar automáticamente los ejes, las etiquetas y los títulos.
- Siempre inician un nuevo gráfico, borrando el anterior si es necesario.
- Uno de los comandos gráficos más utilizados es la función *plot*. Ésta es una función genérica, es decir, el tipo de gráfico que produce es diferente en función de la clase del objeto que recibe como parámetro.
- Una vez hecho el gráfico, podemos copiar al portapapeles o salvarlo en disco en uno de los múltiples formatos que nos ofrece R: *Metafile*, *Postscript*, *pdf*, *png*, *jpg* o *bmp*.
- Mediante la instrucciones *recordPlot* y *replayPlot* podemos guardar y recuperar gráficos en nuestro *workspace*.

```
> plot(1:10)
> pl <- recordPlot()
> X11()
> replayPlot(pl)
```

La función *plot* (1)

Si x e y son vectores, produce un gráfico de puntos de x contra y .

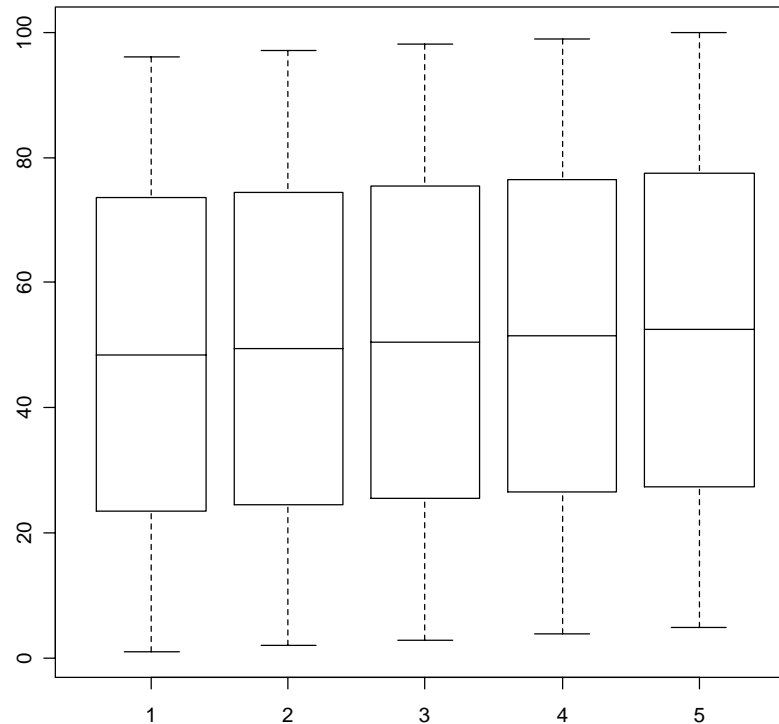
```
> plot(x=1:10, y=1:10) #Otros formatos: plot(xy), plot(x)
```



La función *plot* (2)

Si f es un factor e y es un vector, *plot* produce un *boxplot*.

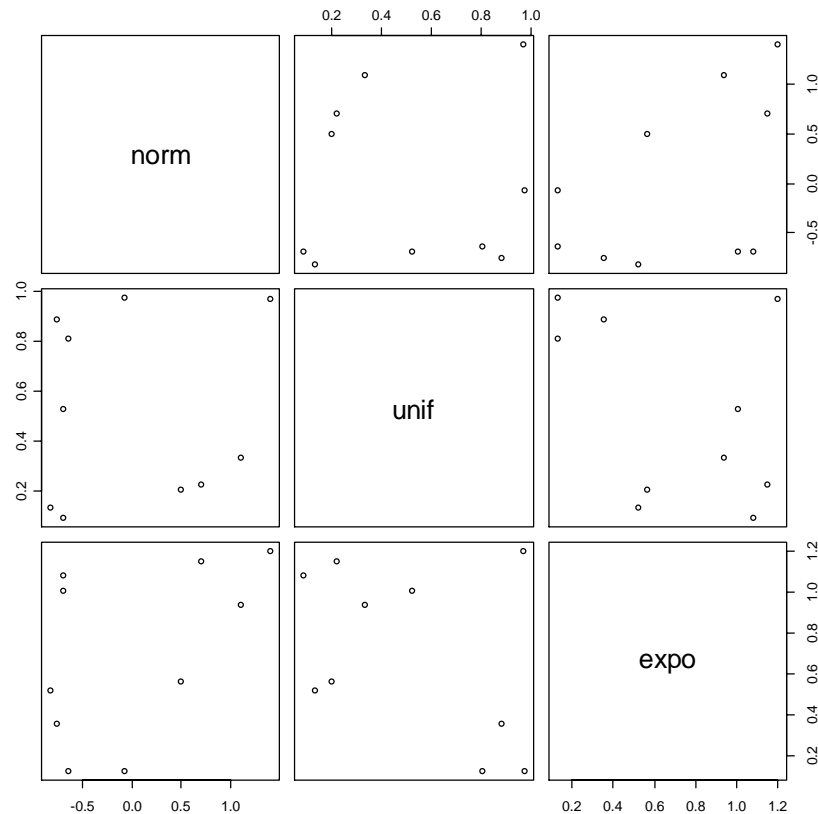
```
> a <- 1:100  
> f <- rep(1:5, 20)  
> f <- factor(f)  
> plot(f,a)
```



La función *plot* (3)

Si *d* es un *data frame* obtenemos gráficas de todos contra todos.

```
> d <- data.frame(norm=rnorm(10),unif=runif(10),expo=rexp(10))  
> plot(d)
```



Parámetros de los comandos de alto nivel.

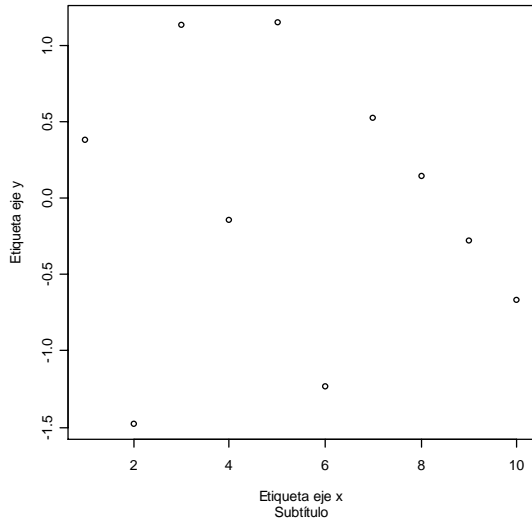
Plot y la mayoría de los otros comandos gráficos aceptan, entre muchos otros, la siguiente lista de parámetros:

- *type*: controla el tipo de gráfico. Los principales son:
 - "p": puntos (por defecto)
 - "l": líneas
 - "b": puntos y líneas
 - "h": histograma
 - "s": escalones
 - "n": no dibuja nada
- *xlab*, *ylab*: etiquetas de los ejes.
- *xlim*, *ylim*: modifica los límites de los ejes
- *main*: título del gráfico (parte superior de la ventana).
- *sub*: subtítulo del gráfico (parte inferior de la ventana).
- *log*: pone los ejes en escala logarítmica: "x", "y", "xy".
- *axes*: para activar/suprimir el dibujo de los ejes (activo por defecto).

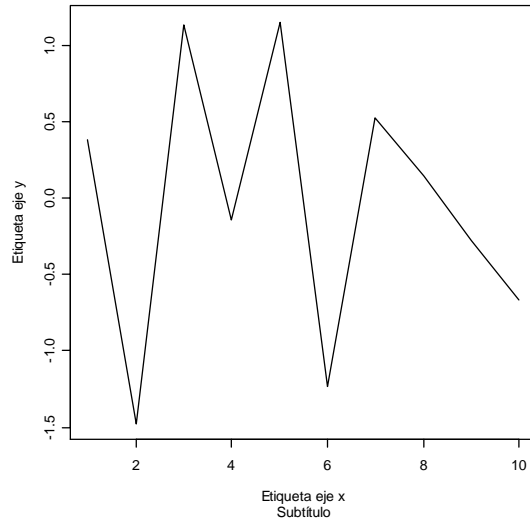
```
> a <- rnorm(10)
> plot(a,type="p",main="Título (tipo p)",sub="Subtítulo",xlab="Etiqueta eje x",ylab="Etiqueta eje y",xlim=c(1,10))
> plot(a,type="l",main="Título (tipo l)",sub="Subtítulo",xlab="Etiqueta eje x",ylab="Etiqueta eje y",xlim=c(1,10))
> plot(a,type="b",main="Título (tipo b)",sub="Subtítulo",xlab="Etiqueta eje x",ylab="Etiqueta eje y",xlim=c(1,10))
> plot(a,type="h",main="Título (tipo h)",sub="Subtítulo",xlab="Etiqueta eje x",ylab="Etiqueta eje y",xlim=c(1,10))
> plot(a,type="s",main="Título (tipo s)",sub="Subtítulo",xlab="Etiqueta eje x",ylab="Etiqueta eje y",xlim=c(1,10))
> plot(a,type="n",main="Título (tipo n)",sub="Subtítulo",xlab="Etiqueta eje x",ylab="Etiqueta eje y",xlim=c(1,10))
```

Más ayuda y resto de parámetros en *?plot* y *?plot.default*

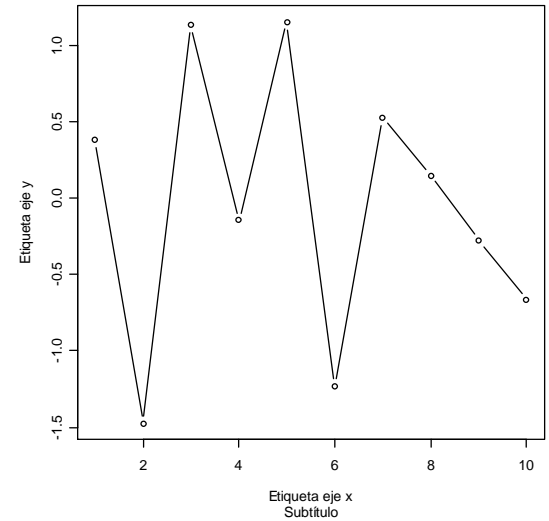
Título (tipo p)



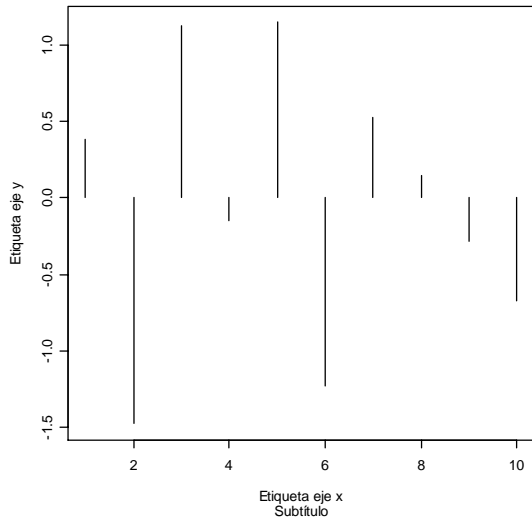
Título (tipo l)



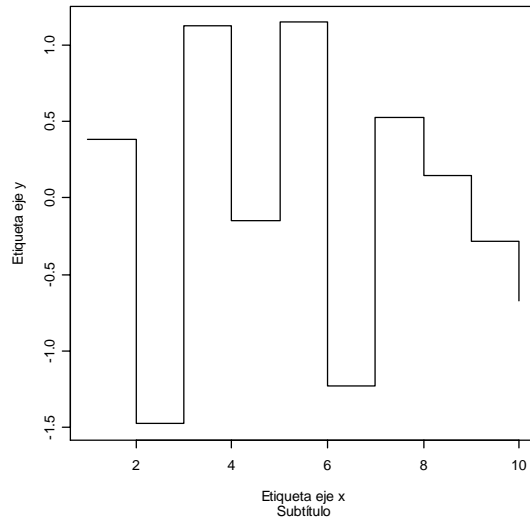
Título (tipo b)



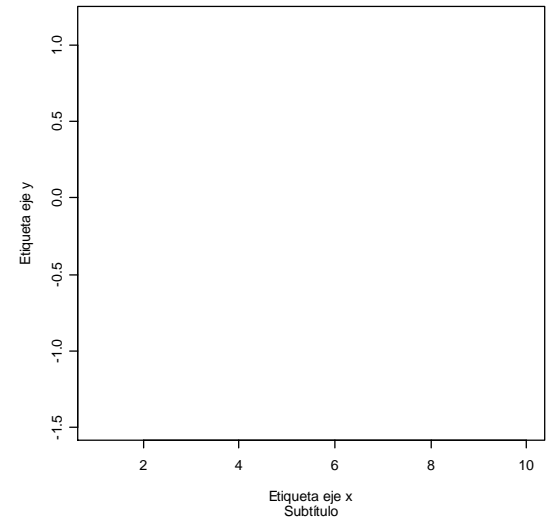
Título (tipo h)



Título (tipo s)



Título (tipo n)



Comandos gráficos de bajo nivel

points(x, y = NULL, type = "p", pch = par("pch"), col = par("col"), bg = NA, cex = 1, ...)

lines(x, y = NULL, type = "l", col = par("col"), lty = par("lty"), ...)

*text(x, y = NULL, labels = seq(along = x), adj = NULL,
pos = NULL, offset = 0.5, vfont = NULL,
cex = 1, col = NULL, font = NULL, xpd = NULL, ...)*

abline(a, b, untf = FALSE, ...)

abline(h=, untf = FALSE, ...)

abline(v=, untf = FALSE, ...)

abline(lm.obj)

*polygon(x, y = NULL, density = NULL, angle = 45,
border = NULL, col = NA, lty = NULL, xpd = NULL, ...)*

*legend(x, y = NULL, legend, fill = NULL, col = "black", lty, lwd, pch,
angle = NULL, density = NULL, bty = "o", bg = par("bg"), ...)*

title(main = NULL, sub = NULL, xlab = NULL, ylab = NULL, line = NA, outer = FALSE, ...)

*axis(side, at = NULL, labels = TRUE, tick = TRUE, line = NA,
pos = NA, outer = FALSE, font = NA, vfont = NULL,
lty = "solid", lwd = 1, col = NULL, ...)*

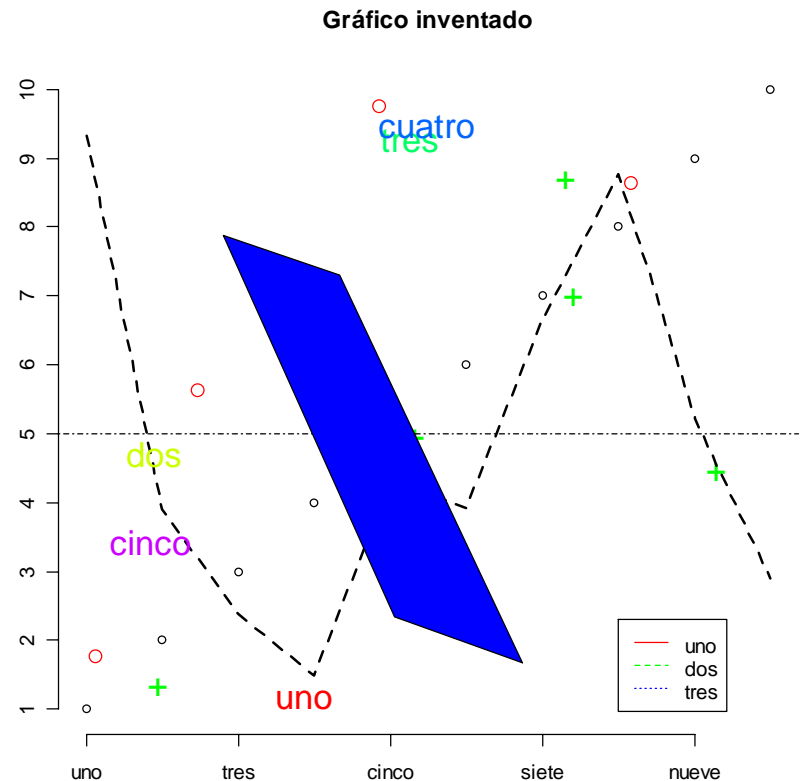
Comandos gráficos de bajo nivel

```

> plot(1:10,1:10,axes=FALSE,xlab="",ylab="")
> points(runif(5,1,10),runif(5,1,10),cex=1.5,col="red")
> points(runif(5,1,10),runif(5,1,10),pch="+",cex=1.7,col="green")
> lines(1:10,runif(10,1,10),lty=2,lwd=2)
> text(runif(5,1,10),runif(5,1,10),labels=c("uno","dos","tres","cuatro","cinco"),cex=1.7,col=rainbow(5))
> abline(h=5,lty=4)
> polygon(runif(4,1,10),runif(4,1,10),col="blue")
> title("Gráfico inventado")
> axis(1,labels=c("uno","tres","cinco","siete","nueve"),at=seq(1,10,by=2))
> axis(2,labels=1:10,at=1:10)
> legend(8,2.3,lty=c(1:3),col=rainbow(3),legend=c("uno","dos","tres"))

```

Consultar `?plotmath` y `?Hershey` para la inclusión de símbolos matemáticos, letras de alfabetos japonés, cirílico o griego y caracteres especiales en los gráficos.



Interactuando con gráficos

```
locator(n=512, type="n", ...)
```

Devuelve las coordenadas del gráfico cada vez que apretamos el botón derecho del ratón dentro de él. *n* es el número máximo de puntos a localizar, y *type* indica qué se va a pintar cuando marquemos los puntos (por defecto nada).

Locator se suele utilizar sin argumentos, y es especialmente útil cuando queremos calcular exactamente la posición para elementos del gráfico como etiquetas o leyendas.

```
> plot(1:10) #Hacemos un gráfico  
> text(locator(1), "Mi punto", adj=0) #Ponemos una etiqueta donde marquemos con el ratón
```

```
identify(x, y = NULL, labels = seq(along = x), ...)
```

Identifica los puntos que nosotros marcamos en el gráfico con el ratón y muestra su índice o la etiqueta que le indicamos en *labels*.

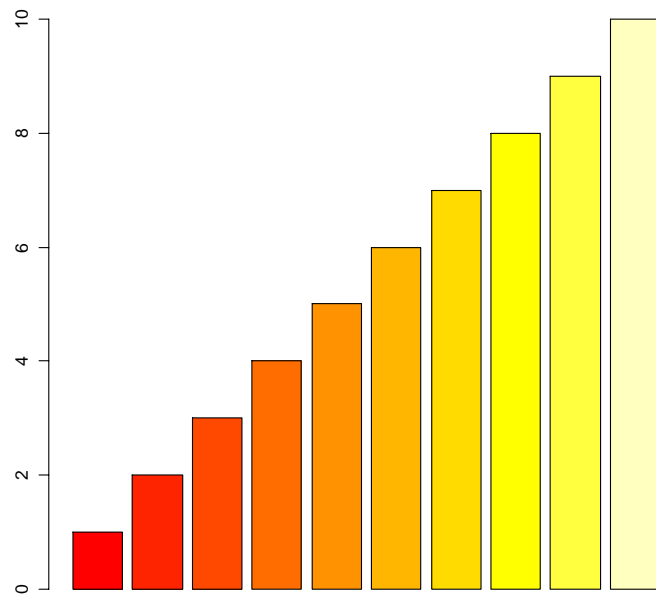
```
> a <- rnorm(10)  
> b <- rnorm(10)  
> plot(a,b)  
> identify(a,b,1:10)
```

Otros comandos de alto nivel

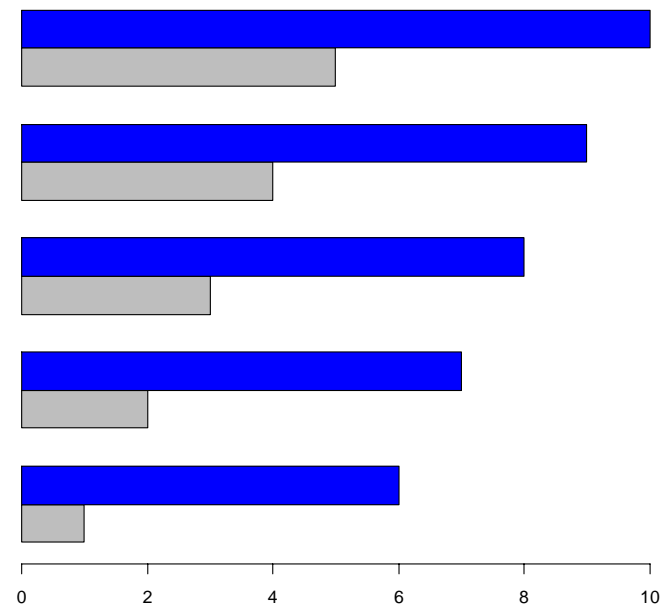
barplot recibe un vector o matriz como parámetro principal. Si es un vector, dibuja tantas barras como representa el vector. Si es una matriz pinta tantas barras como columnas, y cada barra estará formada por tantas barras apiladas como filas tiene la matriz.

```
barplot(height, width = 1, space = NULL, names.arg = NULL, legend.text = NULL, beside = FALSE,  
horiz = FALSE, density = NULL, angle = 45, col = heat.colors(NR), border = par("fg"), ...)
```

```
> barplot(1:10)
```



```
> m <- matrix(1:10,2,5,byrow=T)  
> barplot(m,horiz=TRUE,beside=TRUE,  
col=c("blue","gray"))
```

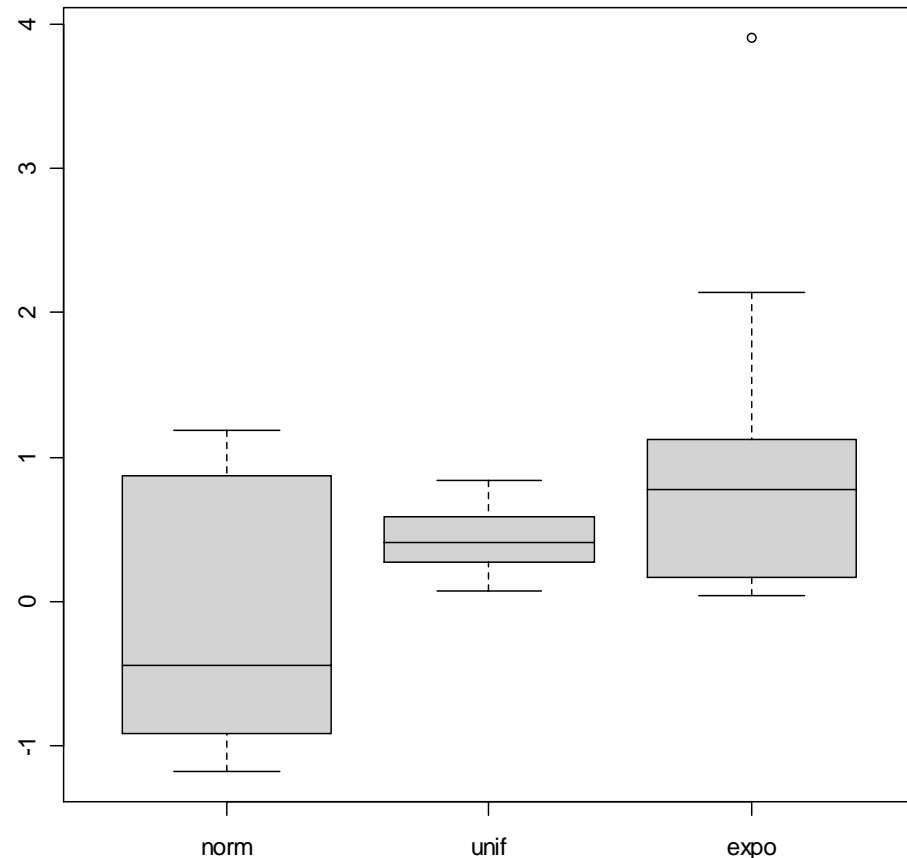


Otros comandos de alto nivel

boxplot recibe un vector, fórmula o *data frame*.

boxplot(*x*, ..., *range* = 1.5, *width* = NULL, *varwidth* = FALSE, *notch* = FALSE, *outline* = TRUE, *names*, *boxwex* = 0.8, *plot* = TRUE, *border* = par("fg"), *col* = NULL, *log* = "", *pars* = NULL, *horizontal* = FALSE, ...)

```
> d <- data.frame(norm=rnorm(10),  
                  unif=runif(10),  
                  expo=rexp(10))  
> boxplot(d,col="lightgray")
```



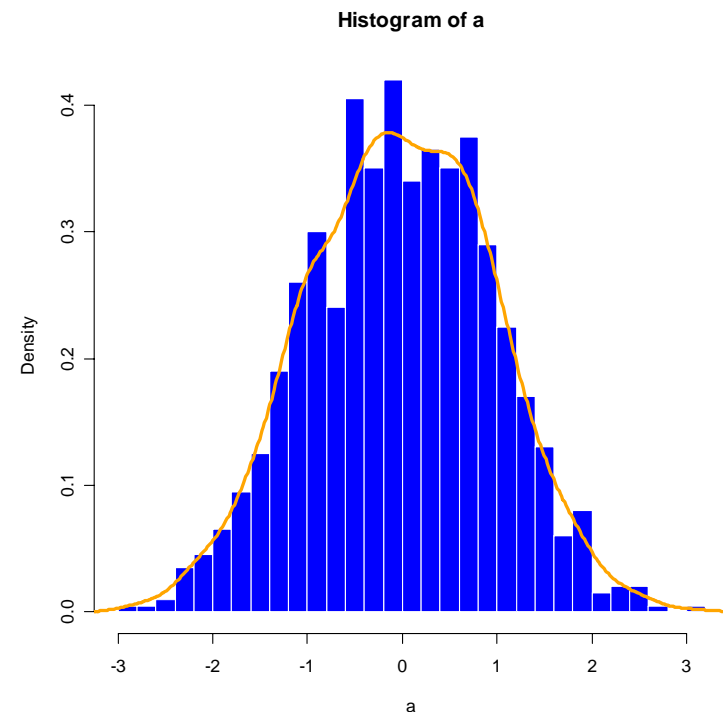
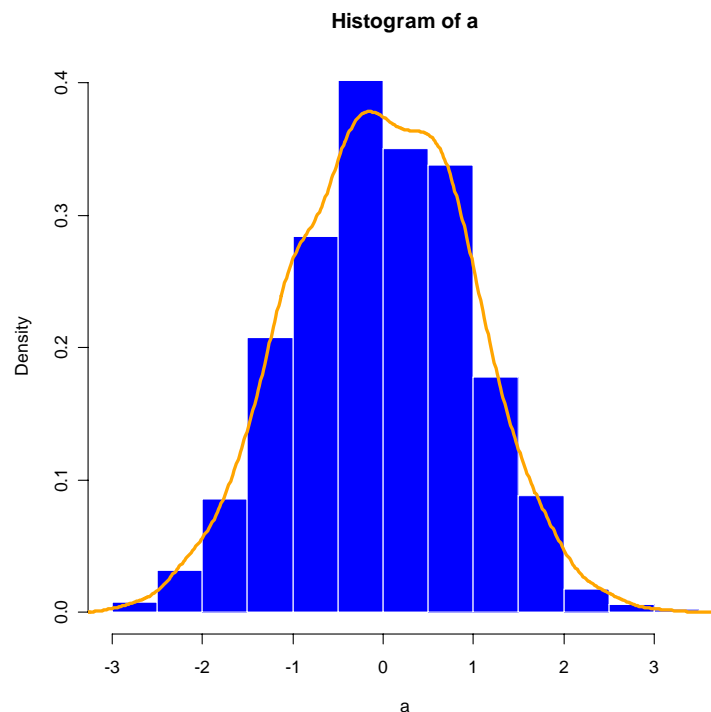
Otros comandos de alto nivel

hist recibe un vector de valores.

```
hist(x, breaks = "Sturges", freq = NULL, probability = !freq, include.lowest = TRUE, right = TRUE,
     density = NULL, angle = 45, col = NULL, border = NULL, main = paste("Histogram of" , xname),
     xlim = range(breaks), ylim = NULL, xlab = xname, ylab, axes = TRUE, plot = TRUE, labels = FALSE,
     nclass = NULL, ...)
```

```
> a <- rnorm(1000)
> hist(a, col="blue",border="white",prob=TRUE)
> lines(density(a),col="orange",lwd=3)
```

```
> hist(a,col="blue",border="white",prob=TRUE,breaks=25)
> lines(density(a),col="orange",lwd=3)
```



Otros comandos de alto nivel

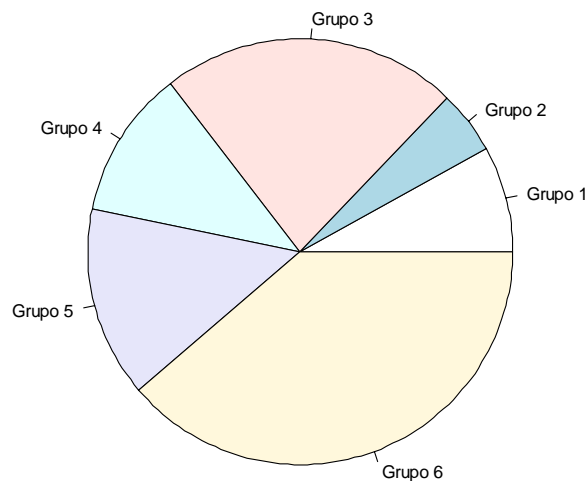
pie recibe un vector de valores.

```
pie(x, labels = names(x), edges = 200, radius = 0.8, density = NULL, angle = 45, col = NULL, border = NULL, lty = NULL, main = NULL, ...)
```

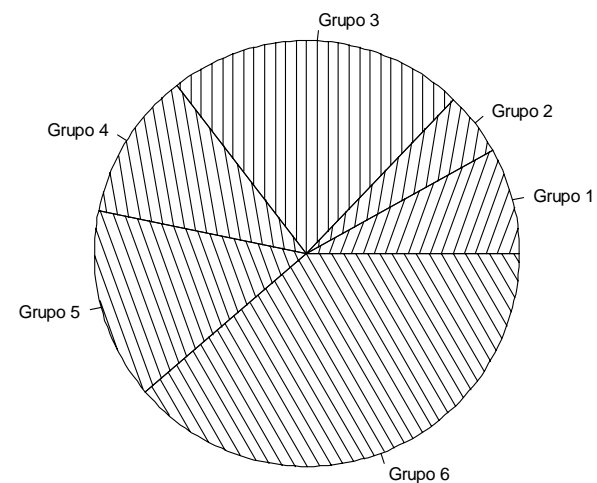
```
> pie(c(5,3,14,7,9,24),labels=paste("Grupo",1:6),  
main="Pie chart")
```

```
> pie(c(5,3,14,7,9,24),labels=paste("Grupo",1:6),  
main="Pie chart",density = 10, angle = 60 + 10 * 1:6)
```

Pie chart



Pie chart

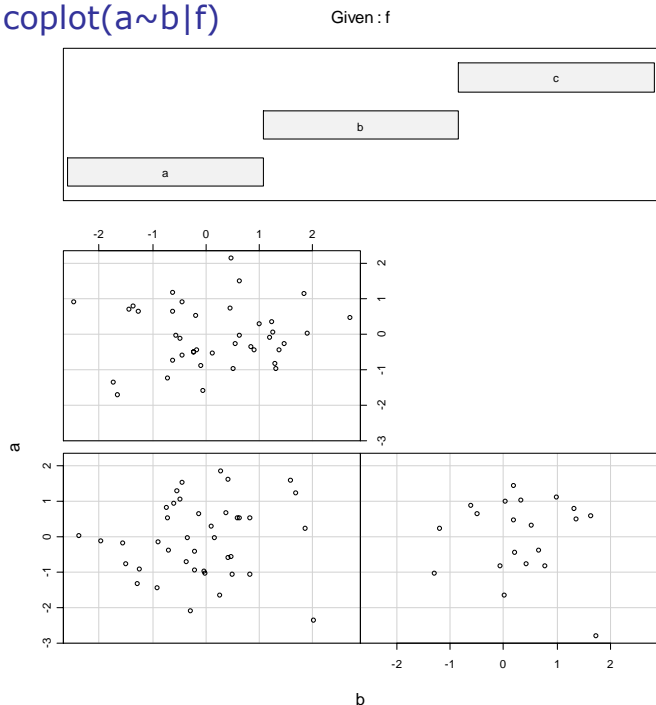


Otros comandos de alto nivel

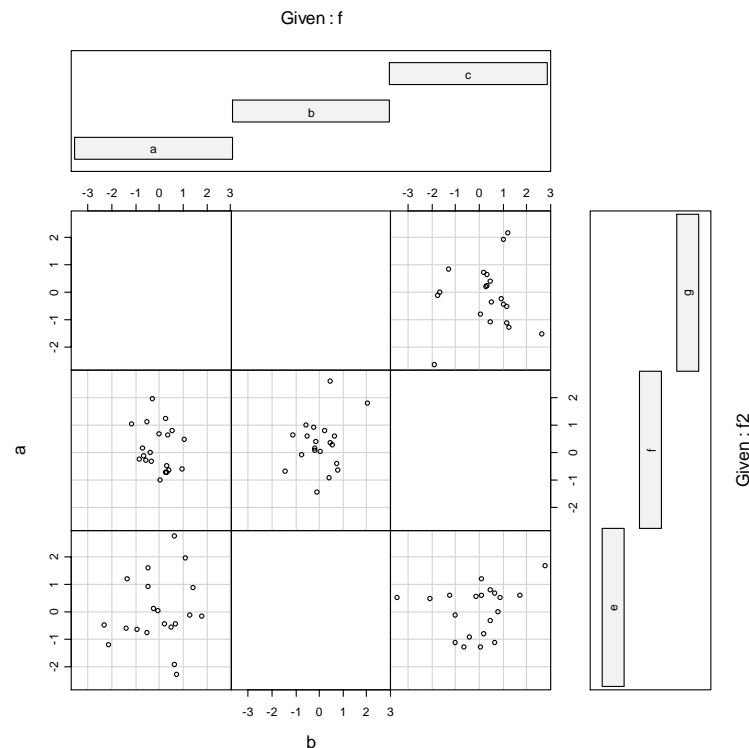
coplot es una función útil cuando tenemos tres o cuatro variables a explorar

```
coplot(formula, data, given.values, panel = points, rows, columns, show.given = TRUE, col = par("fg"),
      pch = par("pch"), bar.bg = c(num = gray(0.8), fac = gray(0.95)),
      xlab = c(x.name, paste("Given :", a.name)), ylab = c(y.name, paste("Given :", b.name)),
      subscripts = FALSE, axlabels = function(f) abbreviate(levels(f)), number = 6, overlap = 0.5, xlim, ylim, ...)
```

```
> a <- rnorm(100)
> b <- rnorm(100)
> f <- factor(rep(c("a","b","c","c","a"),20))
> f2 <- factor(rep(c("e","f","g","e","f"),20))
> coplot(a~b|f)
```



```
> coplot(a~b|f+f2)
```



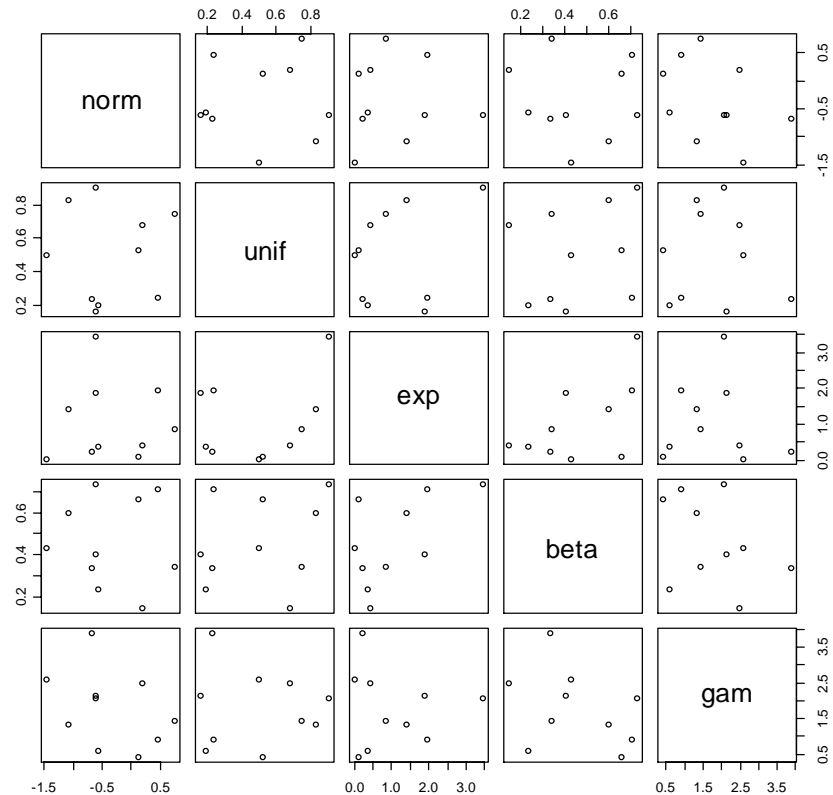
Otros comandos de alto nivel

pairs recibe una matrix y hace gráficos de puntos de todas las columnas contra todas, como un *plot* de un *data frame*.

```
pairs(x, labels, panel = points, ..., lower.panel = panel, upper.panel = panel, diag.panel = NULL,
text.panel = textPanel, label.pos = 0.5 + has.diag/3, cex.labels = NULL, font.labels = 1,
row1atop = TRUE, gap = 1)
```

```
> m <- cbind(norm=rnorm(10),
             unif=runif(10),
             exp=rexp(10),
             beta=rbeta(10,shape1=1,shape2=2),
             gam=rgamma(10,2))

> pairs(m)
```



Ejercicios propuestos

1. Cargar los datos *swiss*. Hacer un gráfico de puntos de las variables *Fertility* contra *Agriculture*. Cambiar las etiquetas de los ejes por unas que decidáis vosotros. Cambiar los puntos por defecto por el carácter "*", hacerlos un 20% más grandes y ponerlos de color azul. Poner los límites de los dos ejes de 0 a 100. Añadirle título y subtítulo a la gráfica.
2. Hacer un *boxplot* conjunto de todas las variables del *data frame*. Cambiar los colores de las cajas (rojos, por ejemplo) y de las líneas que rodean las cajas (verdes). Modificar la longitud de los bigotes para que se ajusten a las observaciones más extremas. Hacer que el *boxplot* sea horizontal. Hacer que las cajas tengan distintos anchos. Sabéis hacer el *boxplot* de tipo *notch*?
3. Hacer un histograma de la variable *Education*. Cambiar los colores de las barras. Poner los puntos de corte en los puntos(0, 10, 20, ..., 100). Pintar la línea de densidad de los datos encima del histograma (recordad de pasarlo a modo "probabilidad").
4. Mediante la instrucción *cut*, discretizar las variables *Catholic* y *Agriculture* en cuatro grupos de tamaño similar. Hacer un *coplot* de *Fertility* contra *Education* teniendo en cuenta esas dos covariables.
5. Practicar otros gráficos y opciones con estos y otros datos. Poner leyendas, añadir nuevos puntos/líneas a los gráficos, cambiar los tipos de líneas, los ejes, colores, etc...