

# Una introducción a R<sup>\*</sup>

Luis Cayuela

6 de noviembre de 2009

EcoLab, Centro Andaluz de Medio Ambiente, Universidad de Granada – Junta de Andalucía, Avenida del Mediterráneo s/n, E-18006, Granada. E-mail: [lcayuela@ugr.es](mailto:lcayuela@ugr.es).

---

<sup>\*</sup>Este material ha sido preparado con el editor de texto Lyx y el programa Sweave.

# Índice

<b>1. ¿Qué es R?</b>	<b>4</b>
<b>2. ¿Cómo instalar R?</b>	<b>5</b>
<b>3. CRAN y paquetes</b>	<b>6</b>
<b>4. Tipos de objetos en R y la función <code>str()</code></b>	<b>7</b>
4.1. Vectores y matrices . . . . .	9
4.2. Funciones y argumentos . . . . .	11
4.3. Ejercicios . . . . .	12
<b>5. El menú de ayuda: Aprendiendo a ser autosuficientes</b>	<b>12</b>
5.1. Ejercicios . . . . .	14
<b>6. Recomendaciones para organizar una sesión de trabajo</b>	<b>14</b>
6.1. Ejercicios . . . . .	15
<b>7. Cómo leer datos en R</b>	<b>15</b>
7.1. Formar un arreglo de datos a partir de las variables . . . . .	17
7.2. Ejercicios . . . . .	18
<b>8. Funciones básicas para la manipulación de datos</b>	<b>18</b>
8.1. Ejercicios . . . . .	20
<b>9. Como repetir un procedimiento con el comando <code>for()</code></b>	<b>20</b>
9.1. Ejercicios . . . . .	21
<b>10. Gráficos en R</b>	<b>21</b>
10.1. Conceptos básicos . . . . .	21
10.2. La organización de los gráficos en R . . . . .	22
10.3. Formato de gráficos . . . . .	23
10.4. El paquete <code>graphics</code> . . . . .	26
10.4.1. Funciones gráficas de alto nivel: Representación de una o dos variables . . . . .	26
10.4.2. Funciones gráficas de alto nivel: Representación de múltiples variables . . . . .	29
10.4.3. Funciones gráficas de bajo nivel . . . . .	32
10.5. Ejercicios . . . . .	34

<b>11.Citando R en los trabajos científicos</b>	<b>34</b>
---	-----------

<b>12.Referencias</b>	<b>35</b>
-----------------------	-----------

## 1. ¿Qué es R?

No es una pregunta fácil de responder. En cierto sentido, la flexibilidad y el potencial de R hace que signifique cosas distintas a distintos usuarios. La respuesta más convencional sería que R es un sistema para la implementación de funciones estadísticas y la creación de gráficos. En un sentido más amplio, sin embargo, R se considera en sí mismo un lenguaje de programación con un conjunto de procedimientos implementados que permiten realizar tareas específicas muy diversas, que van desde la aplicación de funciones estadísticas a la generación y resolución de sudokus... ¡R se puede usar incluso para pedir pizzas! Aunque, por desgracia, de momento esto sólo es posible en Australia.

```
> library(fortunes)
> fortune("pizza")
```

Roger D. Peng: I don't think anyone actually believes that R is designed to make *\*everyone\** happy. For me, R does about 99% of the things I need to do, but sadly, when I need to order a pizza, I still have to pick up the telephone.

Douglas Bates: There are several chains of pizzerias in the U.S. that provide for Internet-based ordering (e.g. [www.papajohnsononline.com](http://www.papajohnsononline.com)) so, with the Internet modules in R, it's only a matter of time before you will have a pizza-ordering function available.

Brian D. Ripley: Indeed, the GraphApp toolkit (used for the RGui interface under R for Windows, but Guido forgot to include it) provides one (for use in Sydney, Australia, we presume as that is where the GraphApp author hails from). Alternatively, a Padovian has no need of ordering pizzas with both home and neighbourhood restaurants ....

```
-- Roger D. Peng, Douglas Bates and Brian D. Ripley
R-help (June 2004)
```

Para que nos hagamos una idea de este potencial, basta escribir un par de líneas de código.

```
> library(lattice)
> demo(lattice)
```

Esto muestra el resultado (y el código usado) de una serie de ejemplos que implementan diversas funciones del paquete gráfico *lattice*, un paquete que permite visualizar datos multivariados y explorar relaciones e interacciones entre variables.

R es un lenguaje orientado a objetos. Aunque existen algunas interfaces gráficas para R como *Rcommander*, es muy recomendable aprender R como un lenguaje en vez de tratarlo como un programa estadístico convencional. Como ambiente de trabajo, R ofrece una serie de ventajas:

- Sus posibilidades gráficas son excelentes

- Es muy flexible. Los procedimientos estadísticos estándar se pueden aplicar con sólo utilizar el comando apropiado. Además, existen multitud de librerías (a los que llamaremos paquetes de ahora en adelante) programadas por los usuarios de todo el mundo para llevar a cabo procedimientos específicos.
- Es libre. Libre no quiere decir gratuito (aunque R también lo es). Libre significa que podemos acceder al código escrito por otros usuarios y modificarlo libremente. A pesar de que R viene sin garantía alguna (al iniciar la sesión de R saldrá la siguiente advertencia “R is free software and comes with ABSOLUTELY NO WARRANTY”), la mayor parte del código de R, o por lo menos, el código más comúnmente utilizado por los usuarios, ha sido meticulosamente supervisado por estadísticos y académicos de mucho prestigio de todo el mundo (el llamado “R Core team”).
- Podemos además programar nuestras propios procedimientos y aplicaciones.
- En la misma página desde la que se puede bajar el programa, existe abundante documentación sobre cómo utilizarlo.
- Es gratuito.

Esta primera sesión del curso constituye una introducción a R. El objetivo principal es mostrar cómo usar el programa para llevar a cabo análisis de datos de diversa índole y, simultáneamente, ilustrar cuáles son las características principales de la sintaxis del lenguaje. Una buena forma de utilizar estos apuntes es reproducir los comandos descritos y experimentar un poco con los parámetros con el fin de entenderlos correctamente.

## 2. ¿Cómo instalar R?

R funciona bien bajo Windows, Linux, Mac o Solaris. Suelen publicarse dos actualizaciones anuales de R. En el momento de escribir estas notas, la última versión para Windows (2.9.2) podía instalarse desde la dirección <http://cran.r-project.org/bin/windows/base/R-2.9.2-win32.exe>. Para Linux existen versiones de R en los repositorios estándar de las distribuciones más populares. En Ubuntu, por ejemplo, los paquetes se instalan usando Synaptic manager o Aptitude. Es recomendable instalar los paquetes rbase, r-base-dev, r-base-core, r-base-html and r-base-latex desde Synaptic.

Si se trabaja en Windows, se puede utilizar el programa Tinn-R para editar rutinas de R con un marcador de sintaxis (<http://www.sciviews.org/Tinn-R/index.html>). Esto resulta muy cómodo, sobretodo cuando se utiliza R por primera vez. A los efectos de este curso utilizaremos, no obstante, un editor de texto común, como Notepad.

### 3. CRAN y paquetes

R es claramente un ambiente de análisis, no un programa convencional. Siendo “open source” cualquier persona puede sugerir modificaciones a su código base. Sin embargo, la modificación de la base de R solamente esta hecha por un grupo de alrededor de veinte especialistas - el “R Core team”.

Cuando se instala R se instalan también una serie de paquetes básicos, que traen implementadas múltiples funciones para la realización de tareas rutinarias, como la representación gráfica y la manipulación de datos, el ajuste de modelos lineales o modelos lineales generalizados, etc. Sin embargo, el verdadero potencial de R viene dado por la forma en que se van extendiendo constantemente sus capacidades por medio de paquetes. Los paquetes están constituidos por una serie de programas compilados y vinculados con R junto con un archivo de ayuda que documenta sus capacidades. La lista de paquetes es actualmente muy muy larga (del orden de varios cientos). Cada semana se suben más paquetes a CRAN (Comprehensive R Archive Network). La responsabilidad de mantener y mejorar los paquetes es de sus autores, pero los que están en uso continuo están constantemente bajo revisión por parte de un grupo de usuarios extensivo.

Si tu ordenador esta conectada a Internet es muy fácil instalar los paquetes en Windows. Para ello selecciona la pestaña de Paquetes/Instalar paquetes desde CRAN.

Escoge un repositorio (mirror) desde donde se realizará la descarga del paquete (en España hay de momento sólo un repositorio). Después elige un paquete de la lista. R te va a preguntar si quieres guardar el archivo localmente después de instalarlo. La respuesta es no (siempre puedes reinstalarlo desde CRAN, así que realmente no necesitas una copia en tu ordenador).

Es aún más fácil instalar paquetes usando una línea de código de R.

```
> install.packages(c("MASS", "vegan", "lattice"), dep = T)
```

El argumento `dep = T` indica a R que instale cualquier otro paquete que sea requerido para el correcto funcionamiento de los paquetes principales que queremos instalar.

Una vez instalado un paquete puedes ver sus contenidos en su archivo de ayuda, como se explica mas adelante. Para poder usar un paquete hay que cargarlo (sólo una vez por sesión) con la función `library()`<sup>1</sup>. Por tanto, los paquetes hay que instalarlos una única vez, pero para utilizarlos hay que cargarlos siempre que se inicie una sesión en R.

```
> library(vegan)
```

---

<sup>1</sup>**NOTA IMPORTANTE** - R es sensible a mayúsculas y minúsculas (case sensitive). ¡No es igual escribir `Library()` que `library()`!

## 4. Tipos de objetos en R y la función `str()`

R, como ya se ha dicho, es un lenguaje orientado a objetos. Esto significa que los diferentes objetos a los que se aplican los comandos de R tienen ciertas características y atributos. Cada comando reconoce estos atributos y actúa de diferente forma en función de ellos. El mismo comando aplicado a diferentes tipos de objetos puede hacer cosas diferentes o, simplemente, no funcionar si no tiene el tipo de objeto requerido. No es necesario predefinir el modo de un objeto, sino que R lo establece de acuerdo con la asignación que hagamos. Las asignaciones a objetos en R se hacen por medio del comando `<-`. R entiende también `=` como un símbolo de asignación. Sin embargo, `=` es también un operador matemático y ello puede inducir a confusión en determinados casos. Por todo ello, se aconseja siempre el uso de `<-` como símbolo de asignación cuando se use código en R. Por ejemplo, para asignar el valor 2.3 al objeto `x`, debemos escribir

```
> x <- 2.3
> mode(x)

[1] "numeric"
```

La función `mode(x)` nos dice que `x` es `numeric` porque es un número (decimal). Sin embargo, si escribimos

```
> x <- "silla"
> mode(x)

[1] "character"
```

entonces `mode(x)` devuelve `character`. En este caso, la sintaxis indica que hemos creado un objeto `x` que consiste en un vector con un elemento de tipo `character` y que sobrescribe al anterior objeto `x`, que era un vector con un elemento de tipo `numeric`.

Los tipos de objetos que utilizaremos con más frecuencia son: vectores, matrices, listas, arreglos de datos (**data frames**) y funciones. Sin embargo, existen otros muchos tipos de objeto de muy diversa índole. Por ejemplo, cuando asignamos el ajuste de un modelo de regresión a un objeto, el objeto resultante es del tipo `lm` (que hace referencia a “linear model”). Un objeto `lm` tiene una estructura de tipo lista, con elementos que contienen los coeficientes del modelo, los valores predichos, los valores observados, los residuos del modelo, etc. Para conocer dicha estructura se recomienda usar la función `str()`. De hecho, al principio es aconsejable el uso continuado de esta función para familiarizarse con la estructura interna de los distintos objetos que iremos manejando. Por ejemplo

```
> data(cars)
> str(cars)
```

```

'data.frame':      50 obs. of  2 variables:
 $ speed: num  4 4 7 7 8 9 10 10 10 11 ...
 $ dist : num  2 10 4 22 16 10 18 26 34 17 ...

> reg1 <- lm(dist ~ speed, data = cars)
> str(reg1)

List of 12
 $ coefficients : Named num [1:2] -17.58 3.93
   ..- attr(*, "names")= chr [1:2] "(Intercept)" "speed"
 $ residuals    : Named num [1:50] 3.85 11.85 -5.95 12.05 2.12 ...
   ..- attr(*, "names")= chr [1:50] "1" "2" "3" "4" ...
 $ effects      : Named num [1:50] -303.914 145.552 -8.115 9.885 0.194 ...
   ..- attr(*, "names")= chr [1:50] "(Intercept)" "speed" "" "" ...
 $ rank         : int 2
 $ fitted.values: Named num [1:50] -1.85 -1.85 9.95 9.95 13.88 ...
   ..- attr(*, "names")= chr [1:50] "1" "2" "3" "4" ...
 $ assign       : int [1:2] 0 1
 $ qr           :List of 5
   ..$ qr      : num [1:50, 1:2] -7.071 0.141 0.141 0.141 0.141 ...
   .. ..- attr(*, "dimnames")=List of 2
   .. .. ..$ : chr [1:50] "1" "2" "3" "4" ...
   .. .. ..$ : chr [1:2] "(Intercept)" "speed"
   .. ..- attr(*, "assign")= int [1:2] 0 1
   ..$ qraux: num [1:2] 1.14 1.27
   ..$ pivot: int [1:2] 1 2
   ..$ tol   : num 1e-07
   ..$ rank  : int 2
   ..- attr(*, "class")= chr "qr"
 $ df.residual  : int 48
 $ xlevels      : list()
 $ call         : language lm(formula = dist ~ speed, data = cars)
 $ terms        :Classes 'terms', 'formula' length 3 dist ~ speed
   .. ..- attr(*, "variables")= language list(dist, speed)
   .. ..- attr(*, "factors")= int [1:2, 1] 0 1
   .. .. ..- attr(*, "dimnames")=List of 2
   .. .. .. ..$ : chr [1:2] "dist" "speed"
   .. .. .. ..$ : chr "speed"
   .. ..- attr(*, "term.labels")= chr "speed"
   .. ..- attr(*, "order")= int 1
   .. ..- attr(*, "intercept")= int 1
   .. ..- attr(*, "response")= int 1
   .. ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
   .. ..- attr(*, "predvars")= language list(dist, speed)
   .. ..- attr(*, "dataClasses")= Named chr [1:2] "numeric" "numeric"
   .. .. ..- attr(*, "names")= chr [1:2] "dist" "speed"
 $ model        : 'data.frame':      50 obs. of  2 variables:
   ..$ dist : num [1:50] 2 10 4 22 16 10 18 26 34 17 ...
   ..$ speed: num [1:50] 4 4 7 7 8 9 10 10 10 11 ...

```



```

..- attr(*, "terms")=Classes 'terms', 'formula' length 3 dist ~ speed
.. .. - attr(*, "variables")= language list(dist, speed)
.. .. - attr(*, "factors")= int [1:2, 1] 0 1
.. .. - attr(*, "dimnames")=List of 2
.. .. $ : chr [1:2] "dist" "speed"
.. .. $ : chr "speed"
.. .. - attr(*, "term.labels")= chr "speed"
.. .. - attr(*, "order")= int 1
.. .. - attr(*, "intercept")= int 1
.. .. - attr(*, "response")= int 1
.. .. - attr(*, ".Environment")=<environment: R_GlobalEnv>
.. .. - attr(*, "predvars")= language list(dist, speed)
.. .. - attr(*, "dataClasses")= Named chr [1:2] "numeric" "numeric"
.. .. - attr(*, "names")= chr [1:2] "dist" "speed"
- attr(*, "class")= chr "lm"

```

Aunque la estructura de los objetos puede ser a veces muy compleja, no debemos preocuparnos. Existen, como veremos más adelante, funciones para extraer la información principal de cada objeto, como por ejemplo la función `summary()`. Una lista de todos los objetos que se encuentran en la memoria de trabajo se obtiene mediante el comando `ls()`. Si se teclea el nombre de un objeto se puede ver su contenido.

#### 4.1. Vectores y matrices

Una manera de crear vectores numéricos es utilizando el comando `scan()` e introduciendo los números uno a uno hasta que se pulsa dos veces seguidas la tecla **Enter** para cerrar el vector.

```
> x <- scan()
```

Una manera más “reproducible” de crear vectores (de cualquier tipo) es mediante el comando `c()` (que alude en inglés a “concatenate”).

```
> x <- c(1, 3, 17, 4, 5)
> x
```

```
[1] 1 3 17 4 5
```

Una sucesión regular de números se puede obtener de la siguiente forma

```
> x <- 1:10
> x
```

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

Si queremos cambiar el incremento podemos usar la función o comando `seq()`.

```
> x <- seq(1, 10, 2)
> x
```

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

En el comando `seq()` el primer argumento indica el punto inicial de la sucesión, el segundo el punto final, y el tercero el incremento.

Es muy fácil generar con R sucesiones de números aleatorios. El programa utiliza funciones de la forma `rdistribucion(n, lista de argumentos)` para generar `n` observaciones de una larga lista de distribuciones. Por ejemplo, si queremos obtener 1000 observaciones de una distribución normal de media 3 y desviación típica 2 y guardarlas en el vector `x` basta escribir:

```
> x <- rnorm(1000, mean = 3, sd = 2)
```

Para R, una matriz no es más que un vector con un atributo adicional que contiene el número de filas y columnas. Por lo tanto una matriz se puede crear a partir de un vector, añadiendo información sobre el número de filas y columnas de la matriz. Por ejemplo, el siguiente comando puede utilizarse para generar aleatoriamente 1000 datos con distribución normal estándar que, a su vez, forman una matriz `x` con dimensión  $10 \times 100$ :

```
> x <- matrix(rnorm(1000), nrow = 10, ncol = 100)
```

Este comando puede ser útil si, en un ejercicio de simulación, queremos simular 100 muestras aleatorias simples de tamaño 10 procedentes de una población normal.

Si creamos una matriz `x` y queremos extraer alguno de sus elementos, podemos usar `x[i,j]`, donde `(i, j)` son la fila y la columna del elemento. Para extraer la tercera fila de la matriz y guardarla en el vector `y` se escribe

```
> y <- x[3, ]
```

es decir, si no especificamos la columna, R entiende que las queremos todas. Estos comandos admiten también valores negativos de manera que si escribimos

```
> y <- x[, -1]
```

entonces `y` es la matriz `x` eliminando la primera columna. Se pueden usar también condiciones lógicas para extraer los elementos de la matriz que nos convenga. Por ejemplo

```
> y <- x[x >= 2]
```

genera un vector `y` con todos los elementos de `x` mayores o iguales que 2. Estas mismas operaciones para extraer elementos de las matrices se aplican a los arreglos de datos, que veremos en el apartado 7.

## 4.2. Funciones y argumentos

Uno de los objetos que más utilizaremos en nuestras sesiones de R son las funciones. Las funciones son aplicaciones que nos permiten realizar operaciones de muy diversa índole. Las funciones están constituidas por argumentos. Vamos a crear, como ejemplo, una función muy simple que nos permita sumar dos números cualesquiera.

```
> suma2numeros <- function(a, b) {
+   a + b
+ }
> suma2numeros(a = 1423, b = 77)
```

```
[1] 1500
```

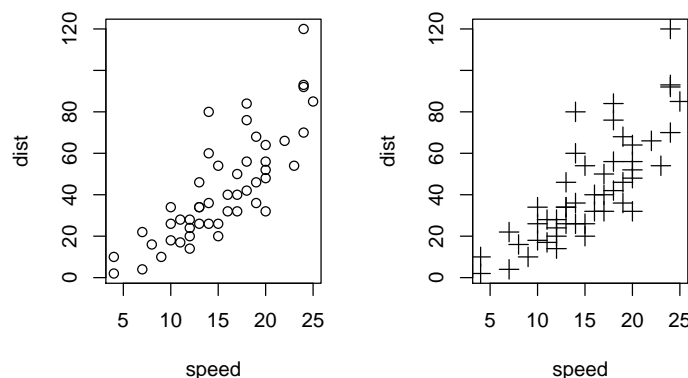
En la función `suma2numeros` los argumentos (`a`, `b`) serían dos números (reales o enteros). Aunque algunas funciones no requieren ningún argumento, como `ls()` o `dir()`, en la mayoría de las funciones el argumento principal, como en `suma2numeros`, serían datos de algún tipo. Si los argumentos se introducen en el orden de entrada requerido no hace falta especificar a qué argumento nos estamos refiriendo en cada momento.

```
> suma2numeros(1423, 77)
```

```
[1] 1500
```

Muchas funciones también tienen una serie de argumentos secundarios que quedan establecidos por defecto, por lo que en principio no haría falta definirlos o modificarlos para hacer que la función se implemente.

```
> par(mfcol = c(1, 2))
> plot(cars)
> plot(cars, pch = 3, cex = 1.5)
```



Aunque tener ciertos conocimientos de cómo programar funciones puede ser a veces de gran utilidad, no es realmente necesario saber programar para poder sacarle un gran provecho a R. La mayor parte de las funciones que se nos ocurran ya han sido implementadas en R por otros usuarios. El reto, por tanto, no es el qué se puede hacer, sino el cómo hacerlo. Para ello es muy importante manejar bien el menú de ayuda, cómo se verá a continuación. Si a esto añadimos unas pequeñas nociones de programación, entonces ¡el potencial de R es prácticamente ilimitado!

### 4.3. Ejercicios

1. Crea una matriz de datos con una distribución normal estándar, con 10 columnas y 20 filas en cada columna. Una vez creada esta matriz, a la que llamaremos `mi.matriz`, selecciona primero el elemento que ocupa la fila 5 de la tercera columna. Después selecciona todos los elementos de la fila 5. Luego todos los elementos de la columna 2. Finalmente selecciona todos los elementos desde la columna 1 a la 5. Para hacer ésto último, usa la función `c()` dentro de la asignación de elementos especificada entre corchetes `[ ]`.
2. Escribe una función llamada `fsel` con un único argumento que permita seleccionar las cinco primeras columnas de una matriz de datos cualquiera (por supuesto, esta matriz tendrá que tener más de cinco columnas o, si no, dará un error). Prueba esta función con la matriz `mi.matriz` creada anteriormente.

## 5. El menú de ayuda: Aprendiendo a ser autosuficientes

Ya sabemos qué queremos hacer, pero cuando entramos en R, nos encontramos con un menú austero y una interfaz bastante hostil. En otros programas, puede ser más intuitivo implementar funciones y procedimientos gracias a su GUI (*Graphical User Interface*). Sin embargo, en R, la GUI es prácticamente inexistente. En Windows, las opciones que tenemos desde los menús desplegables son bastante limitadas y en Linux ni siquiera existen estos menús. En estas condiciones, queda patente que la única manera de trabajar en R es escribiendo código. Para ello, tenemos que averiguar qué funciones hacen lo que nosotros necesitamos y cómo.

¿Cómo utilizar la ayuda? Existen varias funciones que nos interesa manejar con soltura.

**`help.search()`** - busca todas las funciones relacionadas con la palabra o palabras indicadas dentro de los paquetes instalados. Puede ser que la función que busquemos esté en un paquete que no está instalado y, en este caso, `help.search()` no nos ayudaría a encontrarla.

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

El resultado de `help.search()` es a veces un listado muy grande de funciones que, de alguna manera, están relacionadas con la palabra o palabras clave escritas. El siguiente paso es buscar cuál de todas las funciones listadas es la que hace (si es que alguna) lo que nosotros estamos buscando. Para ello, habrá que usar la función `help()` (ver más abajo) para leer la página de ayuda de cada una de ellas.

**RSiteSearch()** - busca las palabras clave en la lista de distribución de R y en los manuales y páginas de ayuda utilizando un motor de búsqueda (<http://search.r-project.org>). Hace falta tener por tanto conexión a internet. El argumento principal es la palabra o palabras clave escritas entre comillas.

```
> RSiteSearch("regression tree")
```

**help()** - El argumento de esta función es el nombre de la función de la que se quiere obtener ayuda. Cuando utilicemos la función `help()` se abrirá una página de ayuda explicando qué hace la función y qué argumentos requiere, y mostrando ejemplos de cómo utilizarla. Alternativamente puede utilizarse el comando `?` seguido del nombre de la función.

```
> help(hist)
```

En el caso de que queramos investigar todas las funciones que hay dentro de un determinado paquete podemos usar la función `help()` con el argumento `package` especificado.

```
> help(package = "stats")
```

También podemos explorar las páginas de ayuda de las funciones a través del navegador utilizando el comando `help.start()`. En Windows también podemos acceder al navegador a través de la pestaña **Help > Html help**.

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

**example()** - Una manera de aprender a manejar funciones en R es implementar los ejemplos de las páginas de ayuda de dichas funciones. Para ello podemos usar la función `example()`, que lee el código escrito en la sección de ejemplos de la página de ayuda de una determinada función y lo implementa en la consola de R. Esto sería el equivalente a copiar y pegar el código manualmente, que también se puede hacer. Posteriormente podemos sustituir los datos que se utilizan en los ejemplos por nuestros propios datos.

```
> example(hist)
```

## 5.1. Ejercicios

1. Utilizando el comando `help.search()` intenta averiguar qué función nos permitiría ajustar un modelo lineal generalizado (*generalized linear model*).
2. Utilizando `help.search()` o `RSiteSearch()` busca algún método o tipo de análisis en el que estés interesado. Sino se te ocurre nada, haz una búsqueda para saber qué paquete, o qué función o funciones pueden hacer un análisis de componentes principales (*principal component analysis*)

## 6. Recomendaciones para organizar una sesión de trabajo

Podemos guardar los objetos `x1, x2, ...` en un fichero llamado `fichero.RData` mediante:

```
> fichero.Rdata <- save(x1, x2, ..., file = "fichero.RData")
```

El fichero se guarda en el directorio de trabajo. Para saber cuál es el directorio de trabajo hay que usar el comando `getwd()`. Para cambiarlo podemos usar el comando `setwd()`<sup>2</sup>

```
> getwd()
> setwd("nuevo directorio")
```

Los objetos que guardamos en un fichero pueden ser tanto datos como resultados de nuestros análisis. Para recuperar esta información en otra sesión de R utilizaremos el comando `load()`

```
> load("fichero.RData")
```

La función

```
> save.image(file = "fichero.RData")
```

guarda en `fichero.RData` todos los objetos que se encuentran en ese momento en la memoria de trabajo de R.

En realidad, la manera más recomendable de trabajar es documentando todo el código en un procesador de texto (p.e. \*.txt, \*.doc). Cómo es muy fácil equivocarse al escribirlo, sobre todo cuando uno usa R por primera vez, se prueba a escribir lo que se quiere en la consola de R y, cuando se consigue hacer lo que se estaba buscando, entonces se copia esa línea de código al

---

<sup>2</sup>Al igual que ocurría con el comando `read.table()`, cuando trabajemos en Linux usaremos la barra / para definir el directorio de trabajo, mientras que en Windows usaremos la barra \.

documento de texto. De esta manera se documenta todo el proceso de análisis de datos y se puede repetir en cualquier momento si, por ejemplo, se incorporan nuevos datos al análisis o se cambia algún paso en el procesamiento de los mismos. Además, esto tiene la ventaja de no acabar con múltiples archivos de datos y de resultados. Se empieza con el archivo o archivos de datos en \*.txt, éstos se leen en R y ya todo el proceso se hace en la memoria virtual del ordenador. Si en algún momento necesitamos un gráfico de resultados o el valor de significación de algún test estadístico, entonces se vuelve a correr el análisis y se obtiene específicamente lo que se necesita.

**NOTA:** R no entiende cómo código todo lo que va precedido del símbolo #. Por tanto, se puede utilizar este símbolo para incorporar comentarios o explicaciones en el código que escribamos.

## 6.1. Ejercicios

1. Abre una carpeta en el escritorio llamada 'Curso R'. Crea un archivo de texto llamado 'codigo R.txt' y sávalo en el esta carpeta. En este archivo irás guardando el código que usemos en los ejemplos de este curso con explicaciones de qué es lo que hace cada línea de código utilizando para ello #.

## 7. Cómo leer datos en R

En R hay múltiples funciones que pueden usarse para importar y exportar datos en otros formatos, como Excel, SPSS, shapefiles (formato de mapas vectoriales de ArcView), etc. Sin embargo, la manera de hacerlo no es muy intuitiva ni clara. Hay todo un capítulo dedicado a esto en la página principal del menú de ayuda

(<http://cran.r-project.org/doc/manuals/R-data.html>). Sin embargo, a los efectos de este curso, utilizaremos la opción más cómoda y también la más robusta, que es la lectura de datos a partir de archivos de texto.

Para leer un fichero de texto (formato \*.txt) en el que se encuentran los datos con los que queremos trabajar se usa el comando `read.table()`.

```
> data <- read.table("/nombre del directorio/Ejemplo 1.txt", header = T,  
+ sep = "\t")
```

Este comando generaría un fichero de datos (un objeto del tipo arreglo de datos o 'data frame') con el nombre elegido<sup>3</sup>. El argumento `header` se utiliza para indicar si la primera fila contiene (T) o no (F) los nombres de las variables. El argumento `sep = "\t"` se utiliza para indicar que la separación entre campos está marcada por tabulaciones. El uso de otros argumentos podría ser necesario en algunos casos.

<sup>3</sup>En este ejemplo (para Linux) se utiliza la barra / para separar los campos del directorio. En Windows, no obstante, habrá que sustituir esta barra por la barra invertida \.

Es importante notar que `read.table()` puede fallar si hay espacios en los nombres de las variables o entre cualquiera de las palabras dentro de los niveles de un factor. Para evitar esto, se deberían reemplazar todos los espacios de nombres por `' ' ò '_'`.

Cómo en principio no contamos con este archivo en los ordenadores de prácticas, vamos a leerlo directamente de una dirección de internet, en donde estos datos están disponibles. Para ello usaremos la función `url()` que permite conectar con direcciones de internet (lo mejor, si se puede, es copiar y pegar esta línea de código para evitar errores). Esto tiene grandes ventajas, ya que permite colaborar en la distancia sin necesidad de tener que enviar los ficheros de datos originales que, en ocasiones pueden ser muy grandes.

```
> data <- read.table(url("http://tinyurl.com/ylnry47"), header = T,
+   sep = "\t")
```

Ya hemos creado un objeto `data` que contiene en la memoria virtual de nuestro ordenador los datos que estaban contenidos en el archivo `*.txt`. Cualquier modificación que hagamos lo vamos a hacer sobre el objeto virtual, no sobre el archivo original. Si ahora utilizamos el comando `names()` obtenemos los nombres de las variables.

```
> names(data)

[1] "x"                "y"                "Especie"
[4] "Defoliacion"      "Area_basimetrica" "Altura_media"
[7] "Densidad_pinos"   "Elevacion"        "Pendiente"
[10] "Orientacion"      "Insolacion"       "Potencial_hidrico"
```

que son los nombres de las variables del fichero (correspondientes a las coordenadas geográficas `x` e `y` de los rodales inventariados, la especie predominante, el grado de defoliación del rodal, el área basimétrica, la altura media, la densidad de pies, la elevación, la pendiente, la orientación, la insolación y el potencial hídrico). Con el comando `fix(data)` se abre una hoja de cálculo rudimentaria en la que podemos añadir más datos al fichero o modificar los ya existentes. Aunque se recomienda no hacer esto NUNCA ya que va en contra de la propia filosofía de R. Si cambiamos los datos manualmente no tendremos el proceso documentado y, pasado un tiempo, no sabremos cómo y por qué hemos modificado los datos. Por el contrario, podemos utilizar el comando `edit(data)` para visualizar los datos en una hoja de cálculo, pero sin capacidad para poder modificarlos.

Si escribimos el nombre de una de las variables, por ejemplo `Defoliacion`, observaremos que el programa no la reconoce. Ello se debe a que las variables se nombran anteponiendo el nombre del fichero y el signo `$`, es decir, `data$Defoliacion`. Sin embargo, se puede utilizar `attach(data)` para poder trabajar directamente con los nombres de las variables del fichero. Lo visto anteriormente en matrices también se aplica en arreglo de datos. Es decir, podemos escribir



```
> data[, c(1:2)]
```

para referirnos a la primera y segunda columna de data. O podemos escribir

```
> data[data$Especie == "Pinus nigra", ]
```

para que nos seleccione todas las filas de data que cumplan la condición especificada (esto es, que en la columna Especie contenga el valor *Pinus nigra*)<sup>4</sup>.

Otra forma de acceder a datos en internet o leerlos directamente de otros formatos, como Excel o SPSS, es copiando los datos con Ctrl+C y utilizando el comando `read.table()` con el argumento `file = "clipboard"`.

```
> data <- read.table(file = "clipboard", header = T, sep = "\t")
```

### 7.1. Formar un arreglo de datos a partir de las variables

Supongamos que creamos una variable de datos `x`

```
> x <- c(34, 20, 19)
```

y otra que contiene nombres de países

```
> paises <- c("España", "Italia", "Alemania")
```

Podemos crear un fichero de datos que incluye ambas variables si utilizamos el comando `data.frame()`.

```
> df1 <- data.frame(x, paises)
> df1
```

```
   x paises
1 34 España
2 20  Italia
3 19 Alemania
```

Podemos usar también los comandos `cbind()` y `rbind()` para unir secuencias o partes de vectores, matrices o arreglos de datos por columnas o filas, respectivamente, siempre y cuando éstas tengan la misma extensión

```
> continente <- rep("Europa", 3)
> df2 <- cbind(df1, continente)
> df2
```

---

<sup>4</sup>Para especificar una condición se utiliza el doble igual (`==`) ya que el operador `=` es un signo lógico

```

      x  paises continente
1 34  España      Europa
2 20  Italia      Europa
3 19 Alemania     Europa

```

```
y
```

```

> new.data <- data.frame(x = 3, paises = "Camerun", continente = "Africa")
> df3 <- rbind(df2, new.data)
> df3

```

```

      x  paises continente
1 34  España      Europa
2 20  Italia      Europa
3 19 Alemania     Europa
4  3  Camerun      Africa

```

## 7.2. Ejercicios

1. Descárgate de la siguiente dirección (<http://tinyurl.com/ylnry47>) el archivo de texto 'Ejemplo 1.txt' y guárdalo en la carpeta 'Curso R' que creastes en el ejercicio anterior. Ahora intenta leer estos datos utilizando la función `read.table()`. Salva todo el código (funcional) que vayas escribiendo en el archivo 'codigo R.txt'.
2. Crea una nueva matriz de datos `mi.matriz2` que contenga las cinco columnas de `mi.matriz` y una nueva variable que contenga números del 1 al 20. Hazlo utilizando primero la función `cbind()` y luego la función `data.frame()`. Después de cada operación mira la estructura del objeto resultante utilizando el comando `str(mi.matriz2)` ¿Cuál es la diferencia en el resultado de ambos procesos?

## 8. Funciones básicas para la manipulación de datos

Cuando vamos a trabajar con un arreglo de datos, hay algunas funciones que pueden ser de gran utilidad para resumirlos y procesarlos. A continuación se enumeran algunas de ellas.

**tapply()** - Agrupa los datos de un vector de acuerdo a una variable especificada y les aplica una función.

```
> tapply(data$Defoliacion, data$Especie, mean)
```

```

Pinus nigra Pinus sylvestris
 39.85714      51.21951

```

**aggregate()** - Agrupa los datos de un arreglo de datos de acuerdo a una variable especificada y les aplica una función. Los resultados vienen dados en forma de arreglo de datos con las mismas variables de entrada, pero con tantas filas (casos) como niveles del factor utilizados para agrupar los datos.

```
> aggregate(data[, c(4, 7:12)], by = list(data$Especie), mean)
```

	Group.1	Defoliacion	Densidad_pinos	Elevacion	Pendiente	Orientacion
1	Pinus nigra	39.85714	809.4571	1759.330	14.69456	173.4497
2	Pinus sylvestris	51.21951	1074.2683	1865.483	14.57805	139.7058
		Insolacion	Potencial_hidrico			
1	8501.132	6.321902				
2	8510.748	6.673843				

**by()** - Hace prácticamente lo mismo que **aggregate()** pero los resultados vienen dados en forma de lista, con tantos elementos como niveles del factor utilizados para agrupar los datos.

```
> by(data[, c(4, 7:12)], data$Especie, mean)
```

```
data$Especie: Pinus nigra
      Defoliacion      Densidad_pinos      Elevacion      Pendiente
      39.857143      809.457143      1759.330402      14.694556
      Orientacion      Insolacion Potencial_hidrico
      173.449679      8501.132199      6.321902
-----
data$Especie: Pinus sylvestris
      Defoliacion      Densidad_pinos      Elevacion      Pendiente
      51.219512      1074.268293      1865.483419      14.578054
      Orientacion      Insolacion Potencial_hidrico
      139.705794      8510.747916      6.673843
```

**apply()** - Aplica una función a un arreglo de datos por filas (**MARGIN = 1**) o columnas (**MARGIN = 2**). La base de datos **Amazonia** del paquete **betaper** contiene datos de la abundancia de las especies de árboles de 9 parcelas de 0,16 hectáreas. Con el comando **apply()** vamos a ver cuántas especies hay en cada parcela.

```
> library(betaper)
> data(Amazonia)
> sp <- ifelse(Amazonia[, -c(1:3)] > 0, 1, 0)
> numero.sp <- apply(sp, 2, sum)
> numero.sp
```

	Gengen	Nauta	Maniti	Huanta	Panguana
	220	247	214	204	240
Ex.Petroleros	San.Antonio	Santa.Ana	Tarapoto		
	239	271	235	233	

## 8.1. Ejercicios

1. Con el archivo de datos `data` interesa saber cuál es el promedio de las variables Densidad de pinos, Elevacion, Pendiente, Orientación, Insolacion y Potencial\_hidrico agrupadas por niveles de defoliación. Para ello hay que agrupar la variable Defoliacion por grupos de daños, en dónde A = 0-25 %, B = 25-50 % y C >50 %, utilizando la función `cut()`. Usa el menú de ayuda para saber cómo utilizar esta función.
2. Para el archivo de datos de abundancia de árboles en los Altos de Chiapas HCP (del paquete `betaper`) calcula la abundancia total de individuos muestreados en cada fragmento de bosque y la abundancia total de individuos de cada especie en el total de los 16 fragmentos.

## 9. Como repetir un procedimiento con el comando `for()`

Vamos a ver ahora cómo utilizar el comando `for` para repetir una operación o procedimiento  $n$  veces. Esto puede ser de gran utilidad para hacer simulaciones o simplemente repetir la misma operación con distintos datos sin necesidad de tener que escribir el mismo código muchas veces.

La mejor manera de entender cómo utilizar esta función es viendo un ejemplo. Utilizando la lista de especies del arreglo de datos de árboles `Amazonia` del paquete `betaper`, vamos a hacer cinco selecciones aleatorias de 20 de las 1188 especies que hay y las vamos a salvar en un nuevo arreglo de datos llamado `sp.random`.

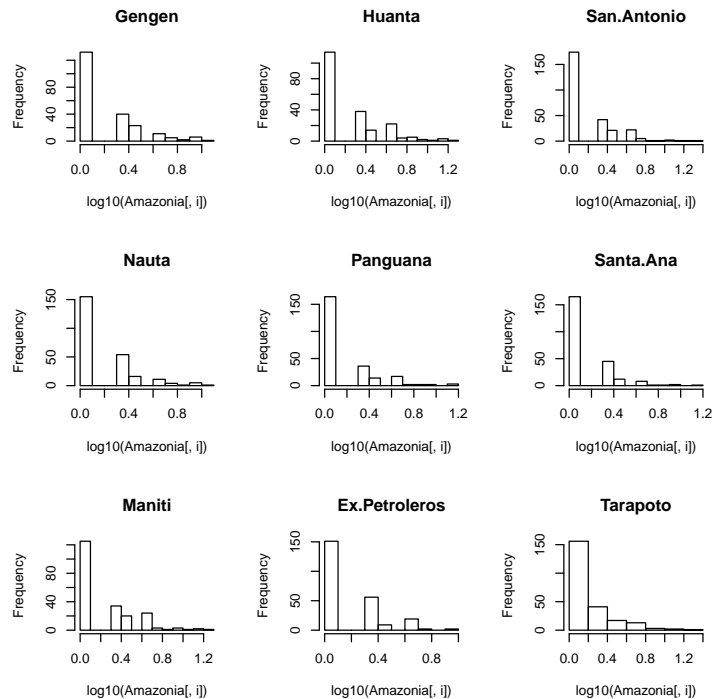
```
> data(Amazonia)
> species <- paste(Amazonia$Genus, Amazonia$Specific)
> sp.random <- as.data.frame(matrix(nrow = 20, ncol = 5))
> for (i in 1:5) {
+   sp.random[, i] <- sample(species, 10)
+ }
```

Otro ejemplo. Para esta misma base de datos, vamos a obtener los histogramas de distribución logarítmica de las abundancias de las especies de cada una de las 9 parcelas.

```

> data(Amazonia)
> par(mfcol = c(3, 3))
> for (i in 4:length(Amazonia[1, ])) {
+   hist(log10(Amazonia[, i]), main = colnames(Amazonia)[i])
+ }

```



## 9.1. Ejercicios

1. Crea un vector llamado `vec1` con 1000 números siguiendo una distribución normal con media 25 y desviación estándar 10. Crea después una matriz de datos llamada `mat1` con 20 filas y 10 columnas. Utilizando ahora el comando `for()`, genera 10 muestras aleatorias de 20 elementos del vector `vec1` con la función `sample()` y sálvalas en cada una de las columnas de la matriz `mat1`. Sirvete siempre que lo necesites de las páginas de ayuda de las funciones utilizadas para conocer qué argumentos requieren.

## 10. Gráficos en R

### 10.1. Conceptos básicos

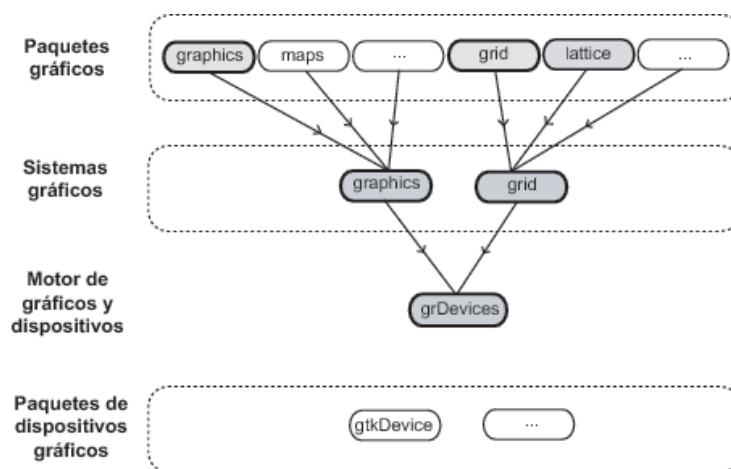
R ofrece una increíble variedad de gráficos. Para tener una idea, escribe `demo(graphics)`. Cada función gráfica en R tiene un enorme número de

opciones permitiendo una gran flexibilidad en la producción de gráficos y el uso de cualquier otro paquete gráfico palidece en comparación. Al contrario que con funciones estadísticas, el resultado de una función gráfica no puede ser asignado a un objeto, sino que es enviado a un dispositivo gráfico. Un dispositivo gráfico es una ventana gráfica o un archivo.

Existen dos tipos fundamentales de funciones gráficas: las **funciones gráficas de alto nivel** que crean una nueva gráfica y las **funciones gráficas de bajo nivel** que agregan elementos a una gráfica ya existente. Las gráficas se producen con respecto a parámetros gráficos que están definidos por defecto y pueden ser modificados con la función `par()`.

## 10.2. La organización de los gráficos en R

El sistema de gráficos de R está constituido por: (1) paquetes gráficos; (2) sistemas gráficos; (3) el motor de gráficos, que incluye los dispositivos gráficos; y (4) paquetes de dispositivos gráficos. El núcleo funcional de los gráficos de R lo componen el motor de gráficos y los dos sistemas gráficos: los gráficos tradicionales (`graphics`) y los gráficos de malla o de retícula (`grid`). El motor de gráficos está constituido por funciones del paquete `grDevices` que dan soporte para el manejo de colores, fuentes, etc., y dispositivos de gráficos que producen la salida de los gráficos en diferentes formatos.



El sistema de gráficos tradicionales lo componen diferentes funciones contenidas en el paquete `graphics`. El sistema de gráficos de malla lo componen diferentes funciones contenidas en el paquete `grid`. Aparte, hay otras muchas funciones gráficas contenidas en otros paquetes accesorios de R, pero todas se construyen sobre alguno de los dos sistemas gráficos de R.

La existencia de dos tipos de sistemas gráficos en R (tradicionales y de malla) puede plantear la pregunta de cuándo se debe utilizar uno y cuando otro.

En principio, el tipo de sistema de gráficos que utilicemos es bastante irrelevante si no tenemos necesidad de añadir ningún elemento extra al gráfico.

Si es necesario añadir algún elemento extra utilizando funciones gráficas de bajo nivel, es conveniente que éstas estén en el mismo sistema gráfico que la función de alto nivel con la que se generó el gráfico.

En algunos casos, se puede producir el mismo tipo de gráfico mediante los dos tipos de sistemas gráficos, por ejemplo utilizando funciones en los paquetes `graphics` y `lattice`, respectivamente. Por ejemplo, si queremos dibujar el histograma de unos datos y no sabemos cómo hacerlo, escribiremos

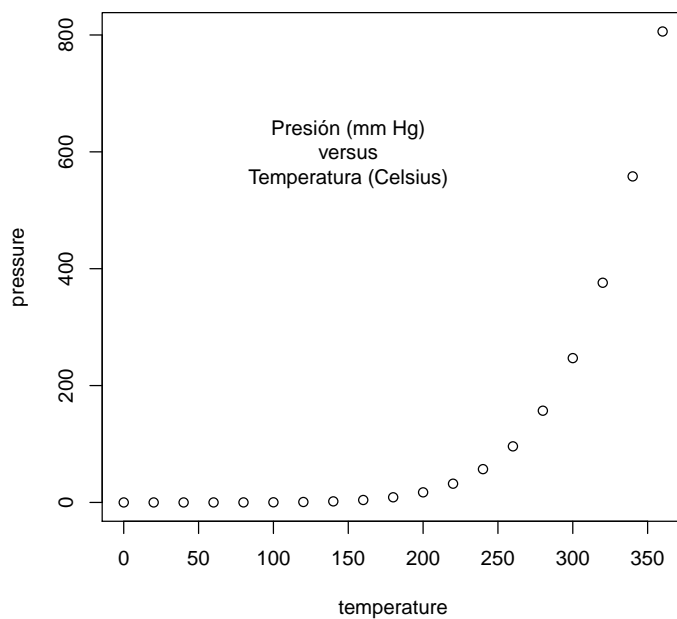
```
> help.search("histogram")
```

El resultado de nuestra búsqueda es una serie de funciones que contienen la palabra `histogram` y que pertenecen a distintos paquetes. Algunas funciones serán de gráfico mientras que otras, tal vez, sean funciones estadísticas o de otro tipo. Entre ellas podemos ver la función `hist` (del paquete `graphics`) y la función `histogram` (del paquete `lattice`, que se construye sobre el sistema de gráficos de malla).

### 10.3. Formato de gráficos

El siguiente código da un ejemplo de cómo producir un gráfico usando R para unos datos de presión y temperatura (el archivo de datos se llama `pressure`)

```
> plot(pressure)
> text(150, 600, "Presión (mm Hg)\nversus\nTemperatura (Celsius)")
```



La función `plot()` produce un gráfico de dispersión de la presión frente a la temperatura, incluyendo ejes, rótulos y un rectángulo que delimita el gráfico. La llamada a la función gráfica de bajo nivel `text()` añade un rótulo al gráfico y lo posiciona dentro del texto.

Cuando se usa R de esta forma, el resultado es un gráfico dibujado en la pantalla. Sin embargo, existe también la posibilidad de producir un archivo que contenga el gráfico, por ejemplo, como un documento \*.jpeg. Se pueden producir gráficos en una gran cantidad de formatos.

En la terminología de R, la salida gráfica se dirige a un determinado tipo de dispositivo gráfico y esto determina el formato de salida del archivo. Un dispositivo debe ser creado para poder recibir salida gráficas y, para aquellos dispositivos que crean un archivo en el disco, el dispositivo debe también cerrarse. Por ejemplo, para producir un archivo \*.jpeg, R tiene una función `jpeg()` que abre un archivo \*.jpeg que recibe la salida gráfica que nosotros ordenemos. La función `dev.off()` cierra este dispositivo.

```
> jpeg(file = "grafico1.jpeg")
> plot(pressure)
> dev.off()
```

Para producir la misma salida gráfica en formato \*.pdf, el código sería

```
> pdf(file = "grafico1.pdf")
> plot(pressure)
> dev.off()
```

La siguiente tabla proporciona una lista de las funciones que abren dispositivos gráficos y los formatos de salida gráfica con los que se corresponden.

Función	Formato gráfico
<i>Dispositivos de ventana</i>	
<code>x11()</code> , <code>X11()</code> , <code>windows()</code>	Ventana
<i>Dispositivos de archivo</i>	
<code>postscript()</code>	Archivo de Adobe postscript
<code>pdf()</code>	Archivo de Adobe pdf
<code>pictex()</code>	Archivo LATEX pictex
<code>xfig()</code>	Archivo XFIG
<code>bitmap()</code>	Archivo de mapa de bits
<code>png()</code>	Archivo de mapa de bits PNG
<code>jpeg()</code>	Archivo de mapa de bits JPEG
<code>win.metafile()</code>	Archivo de Windows Metafile
<code>bmp()</code>	Archivo de Windows BMP

Todas estas funciones ofrecen una serie de argumentos que permiten al usuario especificar cosas como el tamaño físico de la ventana o del documento creado. La documentación para cada una de ellas debe ser consultada para ver la descripción específica de cada uno de sus argumentos. Por ejemplo, si queremos cambiar el color de fondo y el tamaño del archivo \*.pdf que vamos a crear, primero consultamos la documentación de la función `pdf()`



```
> help(pdf)
```

Y luego especificamos los argumentos que nos interesan para cambiar estos parámetros:

```
> pdf(file = "grafico1.pdf", bg = "yellow", width = 3, height = 5)
> plot(pressure)
> dev.off()
```

Es posible tener más de un dispositivo gráfico abierto al mismo tiempo, aunque sólo uno puede estar activo y recibirá la salida gráfica. Si hay varios dispositivos gráficos abiertos al mismo tiempo, hay algunas funciones que nos van a permitir controlar cuál dispositivo está activo, como por ejemplo las funciones `dev.set()`, `dev.cur()` o `dev.list()`.

Algunos dispositivos de archivo permiten múltiples páginas. Por ejemplo, PDF y PostScript permiten múltiples páginas, pero PNG no. A veces es posible, especialmente para dispositivos de archivo que no permiten múltiples páginas, especificar que cada página de la salida gráfica produzca un archivo diferente. Esto se consigue especificando un patrón para el nombre del archivo, del tipo `file="grafico%03d"` de manera que `%03d` es sustituido por un número de tres dígitos indicando el número de página para cada archivo creado. Un ejemplo

```
> jpeg(file = "grafico%03d.jpeg")
> plot(pressure)
> plot(Orange)
> dev.off()
```

En los archivos que sí permiten múltiples páginas habrá que especificar además el argumento `onefile=FALSE` cuando se abre el dispositivo.

```
> pdf(file = "grafico%03d.pdf", onefile = FALSE)
> plot(pressure)
> plot(Orange)
> dev.off()
```

En caso de no especificar este argumento, todas las salidas gráficas se almacenarán en el mismo fichero.

Por último, es necesario saber que los gráficos enviados a dispositivos de ventana pueden ser posteriormente enviados a dispositivos de archivo utilizando la interfaz de RGui (Archivo → Guardar como), aunque no están todos los formatos disponibles y no hay opción de cambiar las especificaciones del archivo (tamaño, tipo de letra, color de fondo, etc.). La interfaz de 'R Commander' (paquete `Rcmdr`) también permite crear gráficos en R utilizando un formato más parecido a los softwares habituales (p.e. SPSS, Statistica o Excel). Sin embargo, las posibilidades que ofrece 'R Commander' son mucho más limitadas que el manejo del código fuente en R, por lo que se aconseja el uso de este último.

## 10.4. El paquete **graphics**

El conjunto de funciones que constituyen el sistema de gráficos tradicional están contenidas en el paquete **graphics**, que se carga automáticamente en una instalación estándar de R. Por tanto, no hace falta abrir este paquete en cada sesión, ya que se abrirá por defecto. Esta sección menciona todas las funciones de alto nivel que se encuentran en el paquete **graphics**, pero no describe todos los posibles usos de dichas funciones. Para obtener más información sobre cada una de las funciones, se puede consultar las páginas individuales de ayuda usando la función `help()`.

```
> help(barplot)
```

Otra manera de aprender cómo funciona una determinada función es utilizando el comando `example()`. Esta función corre el código contenido en la sección de ejemplos (bajo el epígrafe **Examples**) de la página de ayuda de la función que indiquemos.

```
> par(ask = TRUE)
> example(barplot)
```

La función `par(ask=TRUE)` es importante para asegurarnos de que el usuario sea preguntado antes de correr un nuevo bloque de código, ya que muchas funciones cuentan con más de un ejemplo<sup>5</sup>. Otra manera de hacer lo mismo es ir a la página de ayuda e ir cortando y pegando en R el código que aparece en la sección de ejemplos.

### 10.4.1. Funciones gráficas de alto nivel: Representación de una o dos variables

El sistema tradicional de gráficos ofrece una variedad de tipos de gráficos básicos: la función `plot()` produce gráficos de dispersión, la función `barplot()` produce gráficos de barra, la función `hist()` produce histogramas, la función `boxplot()` produce gráficos de caja, y la función `pie()` produce gráficos de tarta.

R no hace distinción entre, por ejemplo, un gráfico de dispersión que dibuja sólo los símbolos de los datos en cada coordenada (x, y) y un gráfico de dispersión que dibuja una línea que conecta todos los puntos en (x, y). Estos son meramente variaciones de la misma función controladas por el argumento `type` (tipo de gráfico). Así, por ejemplo, el siguiente código produce cuatro tipos diferentes de gráficos que varían solamente en el argumento `type`

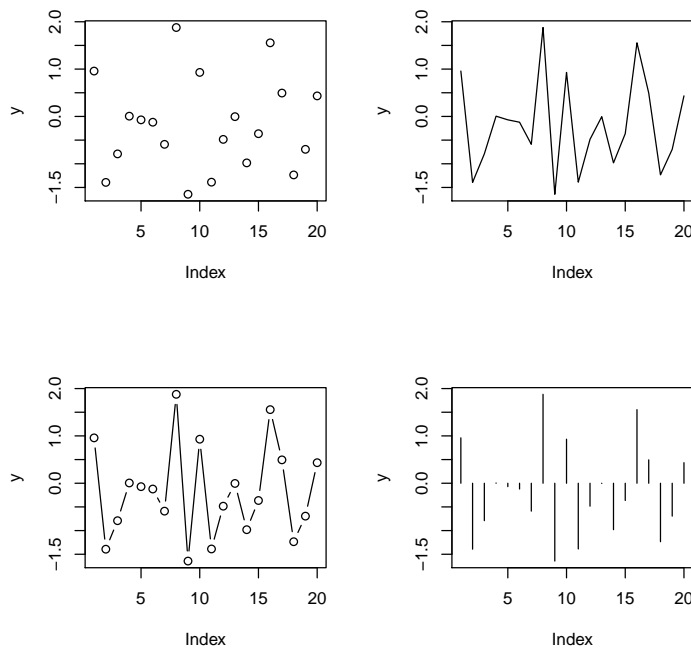
---

<sup>5</sup>La función `par()` controla parámetros gráficos adicionales.

```

> y <- rnorm(20)
> par(mfrow = c(2, 2))
> plot(y, type = "p")
> plot(y, type = "l")
> plot(y, type = "b")
> plot(y, type = "h")

```



El primer argumento -y, generalmente, el único estrictamente necesario- en las funciones gráficas de alto nivel son los datos que se van a dibujar. Este argumento se puede describir de varias formas. Las más comunes se describen a continuación

```

> plot(pressure)
> plot(pressure$temperature, pressure$pressure)
> plot(pressure ~ temperature, data = pressure)

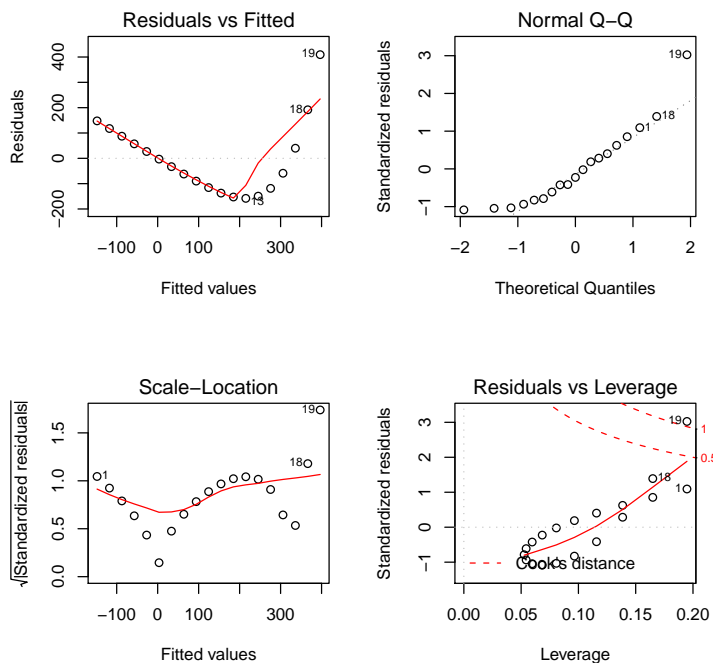
```

En el primer caso, todos los datos a dibujar están especificados en un único arreglo de datos (`data.frame`). En el segundo caso, se especifican las variables que se quieren dibujar en el eje x e y. El símbolo `$` se utiliza para especificar columnas de datos que están dentro de un arreglo de datos (ver sección 7). En el tercer y último caso, las variables que se quieren dibujar se especifican mediante una fórmula, en donde y es función de x.

Todas las funciones gráficas básicas son **genéricas**. Esto significa que el comportamiento de la función depende de la clase a la que pertenezca el primer argumento de la función. Para una función genérica, suele haber una

serie de métodos diferentes, en donde cada método es una función que se corresponde con la acción que se debe emprender para cada clase de argumento. Esto es especialmente relevante para la función `plot()`, en donde obtendremos un tipo de gráfico u otro, dependiendo del tipo de argumento que introduzcamos en primer lugar. Así, si la variable `x` es un factor, la función `plot()` producirá un gráfico de cajas (equivalente a la función `boxplot`). Si el primer argumento es, por el contrario, un objeto `lm` (el ajuste de un modelo lineal), entonces la función `plot()` se puede usar para dibujar los gráficos de los residuos de dicho modelo.

```
> lm1 <- lm(pressure ~ temperature, data = pressure)
> par(mfrow = c(2, 2))
> plot(lm1)
```

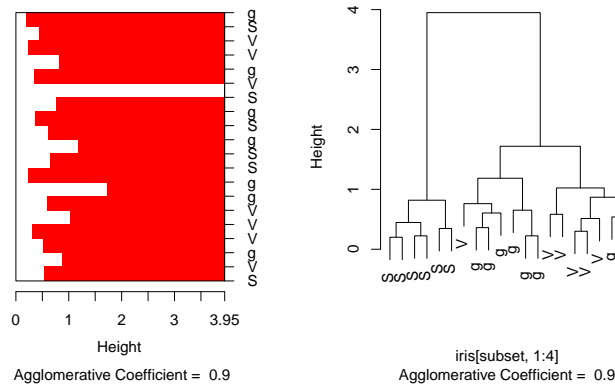


La mayoría de los paquetes proveen nuevos métodos para la función `plot()`. Por ejemplo, el paquete `cluster` ha creado un nuevo método para la función `plot()` que dibuja los resultados de un agrupamiento jerarquizado (el principal argumento es por tanto el resultado de un agrupamiento jerarquizado realizado con la función `agnes()`). Este método produce un gráfico de barras y un dendrograma de los datos. El siguiente código ofrece un ejemplo de ello (la primera expresión carga el paquete `cluster`, las siguientes cinco expresiones simplemente preparan los datos; las tres últimas crean un objeto con el agrupamiento jerarquizado y lo dibujan).

```

> library(cluster)
> subset <- sample(1:150, 20)
> cS <- as.character(Sp <- iris$Species[subset])
> cS[Sp == "setosa"] <- "S"
> cS[Sp == "versicolor"] <- "V"
> cS[Sp == "virginica"] <- "g"
> ai <- agnes(iris[subset, 1:4])
> par(mfrow = c(1, 2))
> plot(ai, labels = cS, main = "")

```



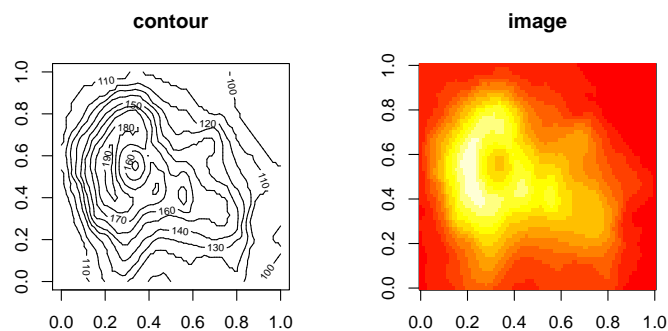
#### 10.4.2. Funciones gráficas de alto nivel: Representación de múltiples variables

El sistema tradicional de gráficos en R también dispone de un amplio número de funciones para visualizar datos multidimensionales. Para gráficos de tres dimensiones las siguientes funciones están disponibles: `contour()` y `filled.contour()` que producen contornos que representan los valores de una tercera variable; `persp()` que produce gráficos en 3D; e `image()` que produce una malla de rectángulos y representa con colores los valores de la tercera variable (esta última es muy parecida a la función `contour()` pero representa píxeles en vez de vectores). Una aplicación muy directa para el uso de estas funciones es con datos geográficos, donde las coordenadas *x* e *y* son coordenadas geográficas y la coordenada *z* representa el valor de otra variable (p.e. altitud, precipitación, temperatura). Para todas estas funciones los datos han de estar representados en forma matricial.

```
> str(volcano)

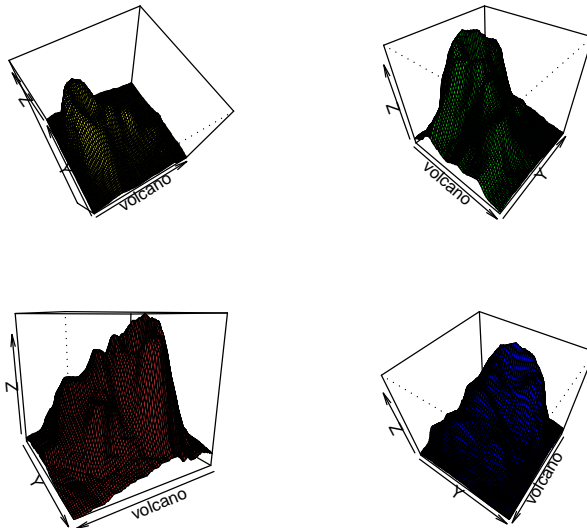
num [1:87, 1:61] 100 101 102 103 104 105 105 106 107 108 ...

> par(mfrow = c(1, 2))
> contour(volcano, main = "contour")
> image(volcano, main = "image")
```



La salida gráfica de estas funciones puede mejorarse considerablemente cambiando los argumentos según convenga. Por ejemplo, para la función `persp()` es interesante conocer los argumentos `theta` y `phi` que modifican los ángulos de visión del gráfico. También se pueden modificar los colores. No es necesario conocer exactamente qué hace cada argumento, basta con probar valores aleatorios e ir probando qué gráfico resulta más apropiado para nuestros objetivos. También puede resultar conveniente utilizar la función `example()` para ver cuál es el potencial de cada función.

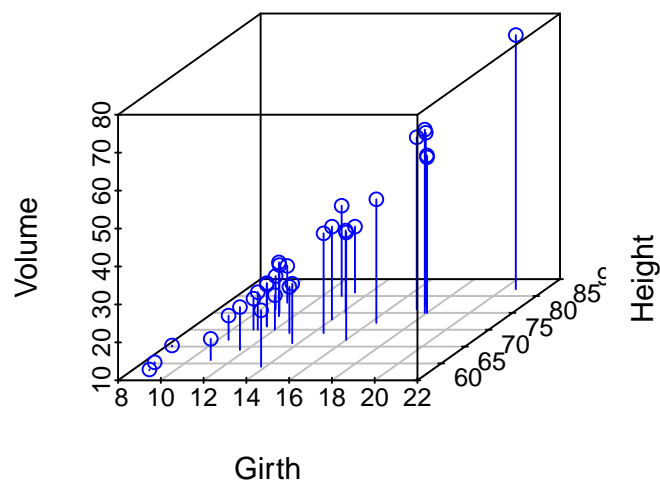
```
> par(mfrow = c(2, 2), mar = c(1, 1, 3, 1))
> persp(volcano, theta = 330, phi = 65, col = "yellow")
> persp(volcano, theta = 40, phi = 40, col = "green3")
> persp(volcano, theta = 150, phi = 15, col = "brown")
> persp(volcano, theta = 125, phi = 50, col = "blue")
```



Existen otros paquetes que incluyen funciones para la representación de datos multidimensionales. Para gráficos en 3D, hay un paquete llamado `scatterplot3d` y otro llamado `rgl`. El último tiene acceso a las capacidades de visualización del lenguaje OpenGL, por lo que tiene algunas ventajas a la hora de visualizar datos multidimensionales, como la rotación de gráficos o la inclusión de efectos de luz.

```
> install.packages(c("scatterplot3d", "rgl"), dep = T)
```

```
> library(scatterplot3d)
> scatterplot3d(trees, type = "h", color = "blue")
```



```
> library(rgl)
> demo(rgl)
> z <- 3 * volcano
> x <- 10 * (1:nrow(z))
> y <- 10 * (1:ncol(z))
> surface3d(x, y, z, color = "green", back = "lines")
```

#### 10.4.3. Funciones gráficas de bajo nivel

R posee un conjunto de funciones gráficas que afectan a una gráfica ya existente: funciones de graficación de bajo nivel. Estas son las principales:



Función	Utilidad
<code>points(x, y)</code>	Agrega puntos (se puede usar el argumento <code>type</code> )
<code>lines(x, y)</code>	Igual a la anterior pero con líneas
<code>text(x, y, labels, ...)</code>	Agrega el texto dado por el argumento <code>labels</code> en las coordenadas (x, y)
<code>mtext(text, side=3, line=0, ...)</code>	Agrega el texto dado por el argumento <code>text</code> en el margen especificado por el argumento <code>side</code> ; <code>line</code> especifica la línea del gráfico donde va a aparecer el texto
<code>segments(x0, y0, x1, y1)</code>	Dibuja una línea desde el punto (x0, y0) hasta el punto (x1, y1)
<code>arrows(x0, y0, x1, y1, angle= 30, code=2)</code>	Igual que el anterior pero con flechas; si <code>code=1</code> , la flecha se dibuja en (x0, y0), si <code>code=2</code> , la flecha se dibuja en (x1, y1); y si <code>code=3</code> , la flecha se dibuja en ambos extremos; <code>angle</code> indica el ángulo de la flecha
<code>abline(a, b)</code>	Dibuja una línea con pendiente 'b' e intercepto 'a'
<code>abline(h=y)</code>	Dibuja una línea horizontal en la ordenada y
<code>abline(v=x)</code>	Dibuja una línea vertical en la coordenada x
<code>abline(lm.obj)</code>	Dibuja una línea de regresión dada por <code>lm.obj</code> (el ajuste de una recta de regresión)
<code>rect(x1, y1, x2, y2)</code>	Dibuja un rectángulo donde las esquinas izquierda, derecha, superior e inferior están dadas por x1, x2, y1 e y2, respectivamente
<code>polygon(x, y)</code>	Dibuja un polígono uniendo los puntos dados por x e y
<code>legend(x, y, legend)</code>	Agrega la leyenda en el punto (x, y) con símbolos dados por <code>legend</code>
<code>title()</code>	Agrega un título y, opcionalmente, un subtítulo
<code>locator(n, type="n")</code>	Devuelve las coordenadas (x, y) después de que el usuario haya hecho click <i>n</i> veces en el gráfico con el ratón
<code>identify(x)</code>	Añade etiquetas a los símbolos representados en un gráfico tras hacer click con el ratón

En los gráficos tradicionales, existen algunas posibilidades de interaccionar con las salidas gráficas, por medio de funciones de bajo nivel. Una de ellas es la función `locator()` que permite al usuario hacer un click dentro de un gráfico y obtener las coordenadas del punto seleccionado con el ratón. La función `identify()` puede ser utilizada para añadir etiquetas a los símbolos de un gráfico.

```
> plot(pressure)
> locator()
> identify(pressure)
```

## 10.5. Ejercicios

1. Crea un único archivo de PDF que contenga los gráficos de presión y temperatura (`pressure`) en donde los puntos se representen con diez tipos de símbolos distintos. Para ayudarte a dibujar los diez gráficos, puedes utilizar el comando `plot(pressure, pch = i)` dentro de una secuencia definida por el comando `for()`. Haz ahora lo mismo pero creando diez archivos PDF distintos, uno para cada tipo de símbolo. Para saber dónde se guardan las gráficas utiliza el comando `getwd()` o cambia el directorio de trabajo con el comando `setwd()`.
2. Utilizando la base de datos sobre defoliación en la Sierra de los Filabres (disponible en <http://tinyurl.com/ylnry47>) representa una gráfica de dispersión del porcentaje de defoliación (eje y) con la insolación (eje x). A esta gráfica hay que añadirle:
  - Los nombres de las variables representadas en los ejes X e Y respectivamente en mayúsculas.
  - Un título general para la gráfica, por ejemplo “Defoliación en Filabres”.
  - Los puntos del diagrama de dispersión con un tamaño que indique la densidad de pinos de esa parcela (cuánto más grande más densidad de pinos). Se puede utilizar el argumento `cex` en la función `plot()` o la función de bajo nivel `points()`. En este último caso, cuando se dibuje la función hay que especificar que el argumento `type = "n"` para que sólo se dibujen los ejes.

## 11. Citando R en los trabajos científicos

Otra pregunta importante es “Quiero usar R en un análisis que voy publicar, pero ¿cómo lo cito?” La respuesta te la da R cuando escribes `citation()`.

```
> citation()
```

To cite R in publications use:

```
R Development Core Team (2009). R: A language and environment for
statistical computing. R Foundation for Statistical Computing,
Vienna, Austria. ISBN 3-900051-07-0, URL http://www.R-project.org.
```

A BibTeX entry for LaTeX users is

```
@Manual{,
  title = {R: A Language and Environment for Statistical Computing},
  author = {{R Development Core Team}},
  organization = {R Foundation for Statistical Computing},
  address = {Vienna, Austria},
  year = {2009},
```

```
note = {{ISBN} 3-900051-07-0},  
url = {http://www.R-project.org},  
}
```

We have invested a lot of time and effort in creating R, please cite it when using it for data analysis. See also ‘citation("pkgname")’ for citing R packages.

El mensaje es claro. R es un proyecto académico. Su uso debe ser respetado por revisores de revistas. Si usas métodos no convencionales la ventaja de R es que en el archivo de cada paquete hay citas disponibles para respaldar tu análisis.

R no debe ser sólo citado como un todo. También debemos citar los paquetes adicionales que utilizemos. Para saber cómo citarlos, se puede usar el comando `citation()` incluyendo dentro el nombre del paquete.

```
> citation("vegan")
```

## 12. Referencias

Hay un gran número de referencias disponibles. Muchos manuales de ayuda, a los que se puede acceder a través de la página de ayuda de R, no tienen nada que envidiar a algunos de los libros que se citan a continuación, con la ventaja de que son gratuitos.

Posiblemente la obra más completa escrita hasta el momento sea:

- Crawley, M.J. 2007. The R Book. Wiley.

No obstante, hay algunos otros libros que pueden también resultar interesantes, en particular:

- Crawley, M.J. 2005. Statistics. An introduction using R. Wiley.
- Dalgaard, P. 2008. Introductory statistics with R (Statistics and Computing). Springer-Verlag.
- Faraway, J.J. 2005. Linear models with R. Chapman & Hall/CRC Press, Florida, USA.
- Faraway, J.J. 2006. Extending the linear model with R. Generalized linear, mixed effects and non parametric regression models. Chapman & Hall/CRC Press, Florida, USA.
- Verzani, J. 2005. Using R for introductory statistics. Chapman & Hall/CRC Press, Florida, USA.

De todas formas, cada año se publica cerca de una docena de manuales de R, por lo que la lista podría alargarse varias hojas.