R para Principiantes

Emmanuel Paradis

Institut des Sciences de l'Évolution
Universit Montpellier II
F-34095 Montpellier cdex 05
France

E-mail: *paradis@isem.univ-montp2.fr*

traducido por

Jorge A. Ahumada

RCUH/ University of Hawaii &
USGS/ National Wildlife Health Center
E-mail: jahumada@usgs.gov

Quiero agradecerle a Julien Claude, Christophe Declercq, Élodie Gazave, Friedrich Leisch y Mathieu Ros por sus comentarios y sugerencias en versiones anteriores de este documento. También estoy muy agradecido con los miembros del grupo nuclear de programadores de R por sus esfuerzos considerables en el desarrollo de R y su ánimo en la lista de discusión ‘rhelp’. Gracias a todos los usuarios de R cuyas preguntas y comentarios me ayudaron a escribir ’R para prinicpiantes’.

Índice

1. Prólogo

El objetivo de este documento es proporcionar un punto de partida para personas interesadas en comenzar a utilzar R. He escogido hacer énfasis en el funcionamiento de R, con el objeto de que se pueda usar de una manera básica. Dado que R ofrece una amplia gama de posibilidades, es útil para el principiante adquirir algunas nociones y conceptos y asi avanzar progresivamente. He tratado de simplificar las explicaciones al máximo para hacerlas lo más comprensivas posibles, pero al mismo tiempo proporcionando detalles útiles, algunas veces con la ayuda de tablas.

R es un sistema para ánalisis estadísticos y gráficos creado por Ross Ihaka y Robert Gentleman[1]. R tiene una naturaleza doble de programa y lenguaje de programación y es considerado como un dialecto del lenguaje S creado por los Laboratorios AT&T Bell. S está disponible como el programa S-PLUS comercializado por Insightful[2]. Existen diferencias importantes en el diseño de R y S: aquellos interesados en averiguar más sobre este tema pueden leer el artículo publicado por Ihaka & Gentleman (1996) o las Preguntas Más Frecuentes en R[3], que también se distribuyen con el programa.

R se distribuye gratuitamente bajo los términos de la *GNU General Public Licence*[4]; su desarrollo y distribución son llevados a cabo por varios estadísticos conocidos como el *Grupo Nuclear de Desarrollo de R.*

R está disponible en varias formas: el código fuente escrito principalmente en C (y algunas rutinas en Fortran), esencialmente para máquinas Unix y Linux, o como archivos binarios pre-compilados para Windows, Linux (Debian, Mandrake, RedHat, SuSe), Macintosh y Alpha Unix.

Los archivos necesarios para instalar R, ya sea desde las fuentes o binarios pre-compilados, se distribuyen desde el sitio de internet *Comprehensive R Archive Network* (CRAN)[5] junto con las instrucciones de instalación. Para las diferentes distribuciones de Linux (Debian, . . .), los binarios están disponibles generalmente para las versiones más actualizadas de éstas *y* de R; visite el sitio CRAN si es necesario.

R posee muchas funciones para análisis estadísticos y gráficos; estos últimos pueden ser visualizados de manera inmediata en su propia ventana y ser guardados en varios formatos (jpg, png, bmp, ps, pdf, emf, pictex, xfig; los formatos disponibles dependen del sistema operativo). Los resultados de análisis estadísticos se muestran en la pantalla, y algunos resultados intermedios (como valores *P*-, coeficientes de regresión, residuales, . . .) se pueden guardar, exportar a un archivo, o ser utilizados en análisis posteriores.

El lenguaje R permite al usuario, por ejemplo, programar bucles (’loops’ en inglés) para analizar conjuntos sucesivos de datos. También es posible combinar en un solo programa diferentes funciones estadísticas para realizar análisis más complejos. Usuarios de R tienen a su disponibilidad un gran número de programas escritos para S y disponibles en la red;[6] la mayoría de estos pueden ser utilzados directamente con R.

Al principio, R puede parecer demasiado complejo para el no-especialista. Esto no es cierto necesariamente. De hecho, una de las características más sobresalientes de R es su enorme flexibilidad. Mientras que programas más clásicos muestran directamente los resultados de un análisis, R guarda estos resultados como un “objeto”, de tal manera que se puede hacer un análisis sin necesidad de mostrar su resultado inmediatamente. Esto puede ser un poco extraño para el usuario, pero esta característica suele ser muy útil. De hecho, el usuario puede extraer solo aquella parte de los resultados que le interesa. Por ejemplo, si uno corre una serie de 20 regresiones y quiere

[1] Ihaka R. & Gentleman R. 1996. R: a language for data analysis and graphics. *Journal of Computational and Graphical Statistics* 5: 299–314.

[2] véase http://www.insightful.com/products/splus/default.html para más información

[3] http://cran.r-project.org/doc/FAQ/R-FAQ.html

[4] para mayor información: http://www.gnu.org/

[5] http://cran.r-project.org/

[6] por ejemplo: http://stat.cmu.edu/S/

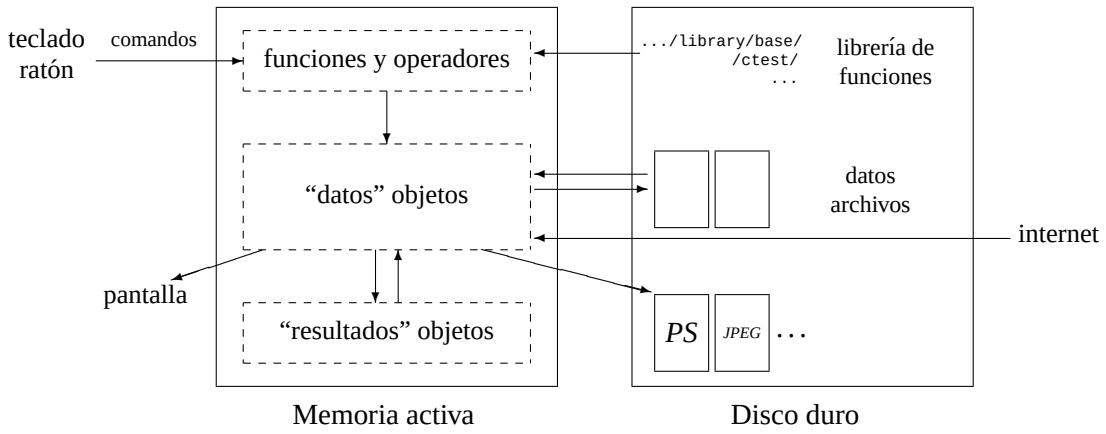


Figura 1: Una visión esquemática del funcionamiento de R.

comparar los coeficientes de regresión, R le puede mostrar únicamente los coeficientes estimados: de esta manera los resultados se pueden resumir en una sola línea, mientras que un programa clásico le puede abrir 20 ventanas de resultados. Más adelante, veremos otros ejemplos que ilustran y comparan la flexibilidad de R con programas de estadística más tradicionales.

2. Algunos conceptos antes de comenzar

Una vez instale R en su computador, el programa se puede iniciar corriendo el archivo ejecutable correspondiente. El cursor, que por defecto es el símbolo '`>`', indica que R está listo para recibir un comando. En Windows, algunos comandos pueden ser ejecutados a través de los menus interactivos (por ej. buscar ayuda en línea, abrir archivos, ...). En este punto, un nuevo usuario de R probablemente estará pensando "Y ahora que hago?". De hecho, cuando se utiliza R por primera vez, es muy útil tener una idea general de como funciona y eso es precisamente lo que vamos a hacer ahora. Como primera medida, veremos brevemente como funciona R. Posteriormente, describiré el operador "asignar" el cual permite crear objetos en R, miraremos como manejar estos objetos en memoria, y finalmente veremos cómo usar la ayuda en línea, la cual a diferencia de las ayudas en otros programas estadísticos, es bastante útil e intuituva.

2.1. Cómo funciona R

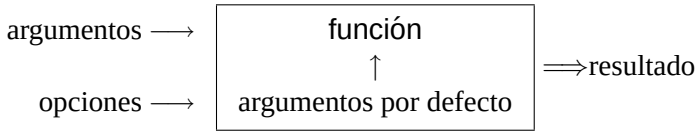
R es un lenguaje *Orientado a Objetos*: bajo este complejo término se esconde la simplicidad y flexibilidad de R. El hecho que R es un lenguaje de programación puede desaminar a muchos usuarios que piensan que no tienen "alma de programadores". Esto no es necesariamente cierto por dos razones. Primero R es un lenguaje interpretado (como Java) y no compilado (como C, C++, Fortran, Pascal, ...), lo cual significa que los comandos escritos en el teclado son ejecutados directamente sin necesidad de construir ejecutables.

Como segunda medida, la sintáxis de R es muy simple e intuitiva. Por ejemplo, una regresión lineal se puede ejecutar con el comando `lm(y ~x)`. Para que una función sea ejecutada en R debe estar *siempre* acompañada de paréntesis, inclusive en el caso que no haya nada dentro de los mismos (por ej., `ls()`). Si se escribe el nombre de la función sin los paréntesis, R mostrará el contenido (código) mismo de la función.

En este documento, se escribirán los nombres de las funciones con paréntesis para distinguirlas de otros objetos, a menos que se indique lo contrario en el texto.

Orientado a Objetos significa que las variables, datos, funciones, resultados, etc., se guardan en la memoria activa del computador en forma de *objetos* con un *nombre* específico. El usuario puede modificar o manipular estos objetos con *operadores* (aritméticos, lógicos, y comparativos) y *funciones* (que a su vez son objetos).

El uso y funcionamiento de los operadores es relativamente intuitivo, y veremos los detalles más adelante (p. 22). Una función en R se puede delinear de la siguiente manera:



Los argumentos pueden ser objetos ("datos", fórmulas, expresiones, ...), algunos de los cuales pueden ser definidos por defecto en la función; sin embargo estos argumentos pueden ser modificados por el usuario con opciones. Una función en R puede carecer totalmente de argumentos, ya sea porque todos están definidos por defecto (y sus valores modificados con opciones), o porque la función realmente no tiene argumentos. Veremos más tarde en detalle como usar y construir funciones (p. 57). Por ahora esta corta descripción es suficiente para entender el funcionamiento básico de R.

Todas las acciones en R se realizan con objetos que son guardados en la memoria activa del ordenador, sin usar archivos temporales (Fig. 1). La lectura y escritura de archivos solo se realiza para la entrada y salida de datos y resultados (gráficas, ...). El usuario ejecuta las funciones con la ayuda de comandos definidos. Los resultados se pueden visualizar directamente en la pantalla, guardar en un objeto o escribir directamente en el disco (particularmente para gráficos). Debido a que los resultados mismos son objetos, pueden ser considerados como datos y analizados como tal. Archivos que contengan datos pueden ser leidos directamente desde el disco local o en un servido remoto a través de la red.

Las funciones disponibles están guardadas en una librería localizada en el directorio R_HOME/library (R_HOME es el directorio donde R está instalado). Este directorio contiene *paquetes* de funciones, las cuales a su vez están estructuradas en directorios. El paquete denominado `base` constituye el núcleo de R y contiene las funciones básicas del lenguaje para leer y manipular datos, algunas funciones gráficas y algunas funciones estadísticas (regresión lineal y análisis de varianza). Cada paquete contiene un directorio denominado R con un archivo con el mismo nombre del paquete (por ejemplo, para el paquete `base`, existe el archivo R_HOME/library/base/R/base). Este archivo está en formato ASCII y contiene todas las funciones del paquete.

El comando más simple es escribir el nombre de un objeto para visualizar su contenido. Por ejemplo, si un objeto `n` contiene el valor 10:

```
> n
[1] 10
```

El dígito 1 indica que la visualización del objeto comienza con el primer elementp de `n`. Este comando constituye un uso implicito de la función `print`, y el ejemplo anterior es similar a `print(n)` (en algunas situaciones la función `print` debe ser usada explicitamente, como por ejemplo dentro de una función o un bucle).

El nombre de un objeto debe comenzar con una letra (A-Z and a-z) y puede incluir letras, dígitos (0-9), y puntos (.). R discrimina entre letras mayúsculas y minúsculas para el nombre de un objeto, de tal manera que `x` y `X` se refiere a objetos diferentes (inclusive bajo Windows).

2.2. Creación, listado y remoción de objetos en memoria

Un objeto puede ser creado con el operador "assignar" el cual se denota como una flecha con el signo menos y el símbolo "$>$" o "$<$" dependiendo de la dirección en que asigna el objeto:

```
> n <- 15
> n
[1] 15
> 5 -> n
> n
[1] 5
> x <- 1
> X <- 10
> x
[1] 1
> X
[1] 10
```

Si el objeto ya existe, su valor anterior es borrado después de la asignación (la modificación afecta solo objetos en memoria, no a los datos en el disco). El valor asignado de esta manera puede ser el resultado de una operación y/o de una función:

```
> n <- 10 + 2
> n
[1] 12
> n <- 3 + rnorm(1)
> n
[1] 2.208807
```

La función `rnorm(1)` genera un dato al azar muestrado de una distribución normal con media 0 y varianza 1 (p. 16). Note que se puede escribir una expresión sin asignar su valor a un objeto; en este caso el resultado será visible en la pantalla pero no será guardado en memoria:

```
> (10 + 2) * 5
[1] 60
```

La asignación será omitida de los ejemplos si no es necesaria para la comprensión del mismo.

La función `ls` simplemente lista los objetos en memoria: sólo se muestran los nombres de los mismos.

```
> name <- "Carmen"; n1 <- 10; n2 <- 100; m <- 0.5
> ls()
[1] "m"    "n1"   "n2"   "name"
```

Note el uso del punto y coma para separar comandos diferentes en la misma línea. Si se quiere listar solo aquellos objetos que contengan un caracter en particular, se puede usar la opción `pattern` (que se puede abreviar como `pat`):

```
> ls(pat = "m")
[1] "m"    "name"
```

Para restringir la lista a aquellos objetos que comienzan con este caracter:

```
> ls(pat = "^m")
[1] "m"
```

La función `ls.str()` muestra algunos detalles de los objetos en memoria:

```
> ls.str()
m :  num 0.5
n1 :  num 10
n2 :  num 100
name :  chr "Carmen"
```

La opción `pattern` se puede usar de la misma manera con `ls.str()`. Otra opción útil en esta función es `max.level` la cual especifica el nivel de detalle para la visualización de objetos compuestos. Por defecto, `ls.str()` muestra todos los detalles de los objetos en memoria, incluyendo las columnas de los marcos de datos ("data frames"), matrices y listas, lo cual puede generar una gran cantidad de información. Podemos evitar mostrar todos estos detalles con la opción `max.level = -1`:

```
> M <- data.frame(n1, n2, m)
> ls.str(pat = "M")
M : 'data.frame':       1 obs. of  3 variables:
 $ n1: num 10
 $ n2: num 100
 $ m : num 0.5
> ls.str(pat="M", max.level=-1)
M : 'data.frame':       1 obs. of  3 variables:
```

Para borrar objetos en memoria, utilizamos la función `rm()`: `rm(x)` elimina el objeto `x`, `rm(x,y)` elimina ambos objetos `x` y `y`, y `rm(list=ls())` elimina todos los objetos en memoria; las mismas opciones mencionadas para la función `ls()` se pueden usar para borrar selectivamente algunos objetos: `rm(list=ls(pat="^m"))`.

2.3. La ayuda en línea

La ayuda en línea de R proporciona información muy útil de cómo utilizar las funciones. La ayuda se encuentra disponible directamente para una función dada. Por ejemplo:

```
> ?lm
```

mostrará dentro de R, ayuda para la función `lm()` (*modelo lineal*). El comando `help(lm)` o `help("lm")` tiene el mismo efecto. Esta última función se debe usar para acceder a la ayuda con caracteres no-convencionales:

```
> ?*
Error: syntax error
> help("*")
Arithmetic               package:base               R Documentation

Arithmetic Operators
...
```

Al llamar la ayuda, se abre una ventana o página (esto depende del sistema operativo) con información general sobre la función en la primera línea, tal como el nombre del paquete donde se encuentra la función u operador. Después viene el título, seguido de secciones con información específica acerca de la misma.

Description: descripción breve.

Usage: para una función, proporciona el nombre de la misma con todos sus argumentos y los posibles valores por defecto (opciones); para un operador describe su uso típico.

Arguments: para una función, describe en detalle cada uno de sus argumentos.

Details: descripción detallada.

Value: si se aplica, el tipo de objeto retornado por la función o el operador.

See Also: otras páginas de ayuda con funciones u operadores similares.

Examples: algunos ejemplos que generalmente pueden ser ejecutados sin abrir la ayuda con la función `examples()`.

Para aquellos que hasta ahora están comenzando en R, es muy útil estudiar la sección **Examples:**. También es útil leer cuidadosamente la sección **Arguments:**. Otras secciones que pueden estar presentes son **Note:** (notas adicionales), **References:** (bibliografía que puede ser útil) o **Author(s):** (nombre del autor o autores).

Por defecto, la función `help` sólo busca en los paquetes que están cargados en memoria. La opción `try.all.packages`, que por defecto tiene el valor `FALSE` (falso), permite buscar en todos los paquetes disponibles si su valor se cambia a `TRUE` (verdadero):

```
> help("bs")
Error in help("bs") : No documentation for ‘bs’ in specified
packages and libraries:
  you could try ‘help.search("bs")’
> help("bs", try.all.packages=TRUE)
  topic ‘bs’ is not in any loaded package
  but can be found in package ‘splines’ in library ‘D:/rw1041/library’
```

Para ver la ayuda en formato html (por ejemplo a través de Netscape) escriba el comando:

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

Con esta ayuda en html es posible realizar búsquedas usando palabras clave. La sección **See Also:** contiene referencias en hipertexto a otras páginas de ayuda. También se pueden realizar búsquedas por palabra clave con la función `help.search` pero esto está aún en estado experimental (versión 1.5.0 de R).

La función `apropos` encuentra todas aquellas funciones cuyo nombre contiene la palabra dada como argumento para los paquetes cargados en memoria:

```
> apropos(help)
[1] "help"           "help.search"    "help.start"
[4] "link.html.help"
```

3. Manejando Datos con R

3.1. Objetos

Hemos visto que R trabaja con objetos los cuales tienen nombre y contenido, pero también *atributos* que especifican el tipo de datos representados por el objeto. Para entender la utilidad de estos atributos, consideremos una variable que toma los valores 1, 2, o 3: tal variable podría ser un número entero (por ejemplo, el número de huevos en un nido), o el código de una variable categórica (por ejemplo, el sexo de los individuos en una población de crustáceos: macho, hembra, o hermafrodita).

Es claro que los resultados de un análisis estadístico de esta variable no será el mismo en ambos casos: con R, los atributos del objeto proporcionan la información necesaria. En general, y hablando un poco más técnicamente, la acción de una función sobre un objeto depende de los atributos de este último.

Todo objeto tiene dos atributos *intrínsecos*: *tipo* y *longitud*. El tipo se refiere a la clase básica de los elementos en el objeto; existen cuatro tipos principales: numérico, caracter, complejo[7], y lógico (`FALSE` [Falso] or `TRUE` [Verdadero]). Existen otros tipos, pero no representan datos como tal (por ejemplo funciones o expresiones). La longitud es simplemente el número de elementos en el objeto. Para ver el tipo y la longitud de un objeto se pueden usar las funciones `mode` y `length`, respectivamente:

```
> x <- 1
> mode(x)
[1] "numeric"
> length(x)
[1] 1
> A <- "Gomphotherium"; compar <- TRUE; z <- 1i
> mode(A); mode(compar); mode(z)
[1] "character"
[1] "logical"
[1] "complex"
```

Cuando un dato no está disponible se representa como `NA` (*del inglés 'not available'*) independientemente del tipo del dato. Datos numéricos que son muy grandes se pueden expresar en notación exponencial:

```
> N <- 2.1e23
> N
[1] 2.1e+23
```

R representa correctamente valores numéricos no-finitos como $\pm\infty$ con `Inf` y `-Inf`, o valores que no son numéricos con `NaN` (*del inglés 'not a number'*).

```
> x <- 5/0
> x
[1] Inf
> exp(x)
[1] Inf
> exp(-x)
[1] 0
```

[7] El tipo complejo no será muy mencionado en este documento.

```
> x - x
[1] NaN
```

Variables que necesitan ser representadas como caracteres se delimitan con comillas `"`. Es posible incluir la comilla misma dentro de la variable si está precedida por el símbolo \. Los dos caracteres juntos \" pueden ser usados por funciones como `cat` para visualización en pantalla, o `write.table` para escritura en archivos (p. 13, véase la opción `qmethod` de esta función).

```
> cit <- "Ella dijo: \"Las comillas se pueden incluir en textos en R.\""
> cit
[1] "Ella dijo: \"Las comillas se pueden incluir en textos en R.\""
> cat(cit)
Ella dijo: "Las comillas se pueden incluir en textos en R."
```

La siguiente tabla resume los tipos de objetos y los datos que representan.

objeto	**tipos**	**varios tipos posibles en el mismo objeto?**
vector	numérico, caracter, complejo *o* lógico	No
factor	numérico *o* caracter	No
arreglo	numérico, caracter, complejo *o* lógico	No
matriz	numérico, caracter, complejo *o* lógico	No
data.frame	numérico, caracter, complejo *o* lógico	Si
ts	numérico, caracter, complejo *o* lógico	Si
lista	numérico, caracter, complejo, lógico función, expresión, ...	Si

Un vector es una variable en el significado comunmente asumido. Un factor es una variable categórica. Un arreglo es una tabla de dimensión k, y una matriz es un caso particular de un arreglo donde $k = 2$. Note que los elementos en un arreglo o una matriz son del mismo tipo. Un 'data.frame' (marco o base de datos) es una tabla compuesta de uno o más vectores y/o factores de la misma longitud pero que pueden ser de diferentes tipos. Un 'ts' es una serie temporal y como tal contiene atributos adicionales tales como frecuencia y fechas. Finalmente, una lista puede contener cualquier tipo de objeto incluyendo otras listas!

Para un vector, su tipo y longitud son suficientes para describirlo. Para otros objetos es necesario usar información adicional que es proporcionada por atributos *no-intrínsecos*. Dentro de estos atributos se encuentran por ejemplo *dim*, que corresponde a las dimensiones del objeto. Por ejemplo, una matriz con 2 filas y 2 columnas tiene como dim la pareja de valores [2, 2], pero su longitud es 4.

3.2. Leyendo datos desde un archivo

R utiliza el directorio de trabajo para leer y escribir archivos. Para saber cual es este directorio puede utilizar el comando `getwd()` (*get working directory*) Para cambiar el directorio de trabajo, se utiliza la fucnión `setwd()`; por ejemplo, `setwd("C:/data")` o `setwd("/home/paradis/R")`. Es necesario proporcionar la dirección ('path') completa del archivo si este no se encuentra en el directorio de trabajo.[8]

[8] En Windows, es útil crear un alias de Rgui.exe, editar sus propiedades, y cambiar el directorio en el campo "Comenzar en:" bajo la lengueta "Alias": este directorio será entonces el nuevo directorio de trabajo cuando R se ejecuta usando el alias.

R puede leer datos guardados como archivos de texto (ASCII) con las siguientes funciones: `read.table` (con sus variantes, ver abajo), `scan` y `read.fwf`. R también puede leer archivos en otros formatos (Excel, SAS, SPSS, ...), y acceder a bases de datos tipo SQL, pero las funciones necesarias no están incluidas en el paquete base. Aunque esta funcionalidad es muy útil para el usuario avanzado, nos restringiremos a describir las funciones para leer archivos en formato ASCII únicamente.

La función `read.table` crea un marco de datos ('data frame') y constituye la manera más usual de leer datos en forma tabular. Por ejemplo si tenemos un archivo de nombre data.dat, el comando:

```
> misdatos <- read.table("data.dat")
```

creará un marco de datos denominado `misdatos`, y cada variable recibirá por defecto el nombre `V1`, `V2`, ... y puede ser accedida individualmente escribiendo `misdatos$V1`, `misdatos$V2`, ..., o escribiendo `misdatos["V1"]`, `misdatos["V2"]`, ..., o, también escribiendo `misdatos[, 1]`, `misdatos[,2 ]`, ...[9] Existen varias opciones con valores por defecto (aquellos usados por R si son omitidos por el usuario) que se detallan en la siguiente tabla:

```
read.table(file, header = FALSE, sep = "", quote = "\"'", dec = ".",
           row.names, col.names, as.is = FALSE, na.strings = "NA",
           colClasses = NA, nrows = -1,
           skip = 0, check.names = TRUE, fill = !blank.lines.skip,
           strip.white = FALSE, blank.lines.skip = TRUE,
           comment.char = "#")
```

`file`	el nombre del archivo (entre "" o como una variable de tipo caracter), posiblemente con su dirección si se encuentra en un directorio diferente al de trabajo (el símbolo \no es permitido y debe reemplazarse con /, inclusive en Windows), o una dirección remota al archivo tipo URL (http://...)
`header`	una variable lógica (`FALSE` (falso) o `TRUE` (verdadero)) indicando si el archivo contiene el nombre de las variables en la primera fila o línea
`sep`	el separador de campo usado en el archivo; por ejemplo `sep="\t"` si es una tabulación
`quote`	los caracteres usados para citar las variables en modo caracter
`dec`	el caracter usado para representar el punto decimal
`row.names`	un vector con los nombres de las líneas de tipo caracter o numérico (por defecto: `1`, `2`, `3`, ...)
`col.names`	un vector con los nombres de las variables (por defecto: `V1`, `V2`, `V3`, ...)
`as.is`	controla la conversión de variables tipo caracter a factores (si es `FALSE`) o las mantiene como caracteres (`TRUE`); `as.is` puede ser un vector lógico o numérico que especifique las variables que se deben mantener como caracteres
`na.strings`	el valor con el que se codifican datos ausentes (convertido a `NA`)
`colClasses`	un vector de caracteres que proporciona clases para las columnas
`nrows`	el número máximo de líneas a leer (se ignoran valores negativos)
`skip`	el número de líneas ignoradas antes de leer los datos
`check.names`	si es `TRUE`, chequea que el nombre de las variables sea válido para R
`fill`	si es `TRUE` y todas las filas no tienen el mismo número de variables, agrega "blancos"
`strip.white`	(condicional a `sep`) si es `TRUE`, borra espacios extra antes y despues de variables tipo caracter
`blank.lines.skip`	si es `TRUE`, ignora líneas en "blanco"
`comment.char`	un caracter que define comentarios en el archivo de datos; líneas que comienzen con este caracter son ignoradas en la lectura (para desactivar este argumento utilize `comment.char =""`)

[9] Existe una diferencia: `misdatos$V1` y `misdatos[, 1]` son vectores mientras que `misdatos["V1"]` es un marco de datos. Mas adelante veremos algunos detalles acerca de la manipulación de objetos (p. 17)

Las variantes de `read.table` son útiles ya que vienen con diferentes opciones por defecto:

```
read.csv(file, header = TRUE, sep = ",", quote="\"", dec=".",
         fill = TRUE, ...)
read.csv2(file, header = TRUE, sep = ";", quote="\"", dec=",",
          fill = TRUE, ...)
read.delim(file, header = TRUE, sep = "\t", quote="\"", dec=".",
           fill = TRUE, ...)
read.delim2(file, header = TRUE, sep = "\t", quote="\"", dec=",",
            fill = TRUE, ...)
```

La función `scan` es más flexible que `read.table`. A diferencia de esta última es posible especificar el modo de las variables:

```
> misdatos <- scan("data.dat", what = list("", 0, 0))
```

En este ejemplo `scan` lee tres variables del archivo data.dat; el primero es un caracter y los siguientes dos son numéricos. Otra distinción importante es la capacidad de `scan()` de crear diferentes objetos como vectores, matrices, marcos de datos, listas, … En el ejemplo anterior, `misdatos` es una lista de tres vectores. Por defecto, es decir si se omite el argumento `what`, `scan()` crea un vector numérico. Si los datos leidos no corresponden al modo (o modos) esperado (s) (ya sea por defecto o especificado a través de `what`), se genera un mensaje de error. Las opciones son las siguientes:

```
scan(file = "", what = double(0), nmax = -1, n = -1, sep = "",
     quote = if (sep=="\n") "" else "'\"", dec = ".",
     skip = 0, nlines = 0, na.strings = "NA",
     flush = FALSE, fill = FALSE, strip.white = FALSE, quiet = FALSE,
     blank.lines.skip = TRUE, multi.line = TRUE, comment.char = "#")
```

`file`	el nombre del archivo(entre “”), posiblemente incluyendo la dirección completa (el símbolo \no es permitido y debe ser reemplazado por /, inclusive bajo Windows), o acceso remoto del tipoURL (http://...); si `file=“”`, los datos deben ser introducidos desde el teclado (la entrada se termina con una línea en blanco)
`what`	especifica el tipo (s) de los datos (numérico por defecto)
`nmax`	el número máximo de datos a ser leido, o si `what` es una lista, el número de líneas por leer (por defecto, `scan` lee los datos hasta que encuentra el final del archivo)
`n`	el número de datos por leer (por defecto no hay limite)
`sep`	el separador de campos usado en el archivo
`quote`	los caracteres usados para citar las variables de tipo caracter
`dec`	el caracter usado para el punto decimal
`skip`	el número de líneas ignorado antes de empezar a leer datos
`nlines`	el número de líneas a leer
`na.string`	el valor asignado a datos ausentes (convertido a `NA`)
`flush`	si es `TRUE`, `scan` va a la siguiente línea una vez se han leido todas las columnas (el usuario puede agregar comentarios en el archivo de datos)
`fill`	agrega “blancos” si es `TRUE` y todas las líneas no tienen el mismo número de variables
`strip.white`	(condicional a `sep`) si es `TRUE`, elimina espacios extras antes y despues de variables de tipo caracter
`quiet`	si es `FALSE`, `scan` muestra una línea indicando los campos que han sido leidos
`blank.lines.skip`	si es `TRUE`, ignora líneas en blanco
`multi.line`	si `what` es una lista, especifica si las variables de un mismo individuo están en una sola línea en el archivo (`FALSE`)
`comment.char`	un caracter que define comentarios en el archivo; aquellas líneas que comiencen con este caracter son ignoradas

La función `read.fwf` puede usarse para leer datos en archivos en *formato fijo ancho*:

```
read.fwf(file, widths, sep="\t", as.is = FALSE,
         skip = 0, row.names, col.names, n = -1)
```

Las opciones son las mismas que para `read.table()` con excepción de `widths` que especifica la anchura de los campos. Por ejemplo, si un archivo de nombre datos.txt tiene los datos mostrados a la derecha, esto se puede leer con el comando:

```
A1.501.2
A1.551.3
B1.601.4
B1.651.5
C1.701.6
C1.751.7
```

```
> misdatos <- read.fwf("datos.txt", widths=c(1, 4, 3))
> misdatos
  V1   V2  V3
1  A 1.50 1.2
2  A 1.55 1.3
3  B 1.60 1.4
4  B 1.65 1.5
5  C 1.70 1.6
6  C 1.75 1.7
```

3.3. Guardando datos

La función `write.table` guarda el contenido de un objeto en un archivo. El objeto es típicamente un marco de datos (’data.frame’), pero puede ser cualquier otro tipo de objeto (vector, matriz,...). Los argumentos y opciones son:

```
write.table(x, file = "", append = FALSE, quote = TRUE, sep = " ",
            eol = "\n", na = "NA", dec = ".", row.names = TRUE,
            col.names = TRUE, qmethod = c("escape", "double"))
```

x	el nombre del objeto a exportar
file	el nombre del archivo (por defecto, el objeto se muestra en la pantalla)
append	si es TRUE anexa los datos al archivo sin borrar datos ya existentes en el mismo
quote	lógico o numérico : si es TRUE variables de tipo caracter y factores se escriben entre ; si es un vector numérico, este indica el número de las variables a ser es mostradas entre (en ambos casos los nombres de las variables se escriben entre pero no si quote = FALSE)
sep	el separador de campo utilizado en el archivo
eol	el caracter que indica el final de línea ("\n" es 'retorno')
na	el caracter a usarse para datos faltantes
dec	el caracter usado para el punto decimal
row.names	una opción lógica que indica si los nombres de las líneas se escriben en el archivo
col.names	identificación para los nombres de las columnas
qmethod	si es quote=TRUE, especifica la manera como se debe tratar las comillas dobles "en variables tipo caracter: si es "escape"(o "e", por defecto) cada "es reemplazada por \; si es "dçada "es reemplazada por

Una manera sencilla de escribir los contenidos de un objeto en un archivo es utilizando el comando `write(x, file="data.txt")`, donde `x` es el nombre del objeto (que puede ser un vector, una matrix, o un arreglo). Esta función tiene dos opciones: `nc` (o `ncol`) que define el número de columnas en el archivo (por defecto `nc=1` si `x` es de tipo caracter, `nc=5` para otros tipos), y `append` (lógico) que agrega los datos al archivo sin borrar datos ya existentes (`TRUE`) o borra cualquier dato que existe en el archivo (`FALSE`, por defecto).

Para guardar un grupo de objetos de cualquier tipo se puede usar el comando `save(x, y, z, file= "xyz.RData")`. Para facilitar la transferencia de datos entre diferentes máquinas se pueden utilizar la opción `ascii = TRUE` . Los datos (denominados ahora como un *workspace* o "*espacio de trabajo*" en terminología de R) se pueden cargar en memoria más tarde con el comando `load("xyz.RData")`. La función `save.image()` es una manera corta del comando `save(list=ls(all=TRUE), file=".RData")` (guarda todos los objetos en memoria en el archivo .RData).

3.4. Generación de datos

3.4.1. Secuencias regulares

Una secuencia regular de números enteros, por ejemplo de 1 hasta 30, se puede generar con:

```
> x <- 1:30
```

El vector resultante `x` tiene 30 elementos. El operador '`:`' tiene prioridad sobre otros operadores aritméticos en una expresión:

```
> 1:10-1
 [1] 0 1 2 3 4 5 6 7 8 9
> 1:(10-1)
[1] 1 2 3 4 5 6 7 8 9
```

La función `seq` puede generar secuencias de números reales:

```
> seq(1, 5, 0.5)
[1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
```

donde el primer número indica el principio de la secuencia, el segundo el final y el tercero el incremento que se debe usar para generar la secuencia. También se puede usar:

```
> seq(length=9, from=1, to=5)
[1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
```

También se pueden escribir los valores directamente usando la función `c`:

```
> c(1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5, 5)
[1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
```

Si se quiere, también es posible introducir datos directamente desde el teclado usando la función `scan` sin opciones:

```
> z <- scan()
1: 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
10:
Read 9 items
> z
[1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
```

La función `rep` crea un vector con elementos idénticos:

```
> rep(1, 30)
 [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

La función `sequence` crea una serie de secuencias de números enteros donde cada secuencia termina en el número (o números) especificado (s) como argumento (s):

```
> sequence(4:5)
[1] 1 2 3 4 1 2 3 4 5
> sequence(c(10,5))
 [1]  1  2  3  4  5  6  7  8  9 10  1  2  3  4  5
```

La función `gl` (*generador de niveles*) es muy útil porque genera series regulares de factores. La función tiene la forma `gl(k, n)` donde `k` es el número de niveles (o clases), y `n` es el número de réplicas en cada nivel. Se pueden usar dos opciones: `length` para especificar el número de datos producidos, y `labels` para especificar los nombres de los factores. Ejemplos:

```
> gl(3, 5)
 [1] 1 1 1 1 1 2 2 2 2 2 3 3 3 3 3
Levels:  1 2 3
> gl(3, 5, length=30)
 [1] 1 1 1 1 1 2 2 2 2 2 3 3 3 3 3 1 1 1 1 1 2 2 2 2 2 3 3 3 3 3
Levels:  1 2 3
> gl(2, 6, label=c("Macho", "Hembra"))
 [1] Macho  Macho  Macho  Macho  Macho  Macho
 [7] Hembra Hembra Hembra Hembra Hembra Hembra
Levels:  Macho Hembra
> gl(2, 10)
 [1] 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2
Levels:  1 2
> gl(2, 1, length=20)
 [1] 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2
Levels:  1 2
> gl(2, 2, length=20)
 [1] 1 1 2 2 1 1 2 2 1 1 2 2 1 1 2 2 1 1 2 2
Levels:  1 2
```

Finalmente, `expand.grid()` crea un marco de datos con todas las combinaciones de vectores o factores proporcionados como argumentos:

```
> expand.grid(a=c(60,80), p=c(100, 300), sexo=c("Macho", "Hembra"))
   a   p    sexo
1 60 100   Male
2 80 100   Male
3 60 300   Male
4 80 300   Male
5 60 100 Female
6 80 100 Female
7 60 300 Female
8 80 300 Female
```

Distribución/función	**función**
Gausse (normal)	`rnorm(n, mean=0, sd=1)`
exponencial	`rexp(n, rate=1)`
gamma	`rgamma(n, shape, scale=1)`
Poisson	`rpois(n, lambda)`
Weibull	`rweibull(n, shape, scale=1)`
Cauchy	`rcauchy(n, location=0, scale=1)`
beta	`rbeta(n, shape1, shape2)`
'Student' (t)	`rt(n, df)`
Fisher–Snedecor (F)	`rf(n, df1, df2)`
Pearson (χ^2)	`rchisq(n, df)`
binomial	`rbinom(n, size, prob)`
geométrica	`rgeom(n, prob)`
hypergeométrica	`rhyper(nn, m, n, k)`
logística	`rlogis(n, location=0, scale=1)`
lognormal	`rlnorm(n, meanlog=0, sdlog=1)`
binomial negativa	`rnbinom(n, size, prob)`
uniforme	`runif(n, min=0, max=1)`
Estadístico de Wilcoxon's	`rwilcox(nn, m, n)`, `rsignrank(nn, n)`

3.4.2. Secuencias aleatorias

La posibilidad de generar datos aleatorios es bastante útil en estadśtica y R tiene la capacidad de hacer esto para un gran número de funciones y distribuciones. Estas funciones son de la forma `r`*`func`*`(n, p1, p2, ...)`, donde *`func`* indica la disribución, `n` es el número de datos generado, y `p1`, `p2`, ... son valores que toman los parámetros de la distribución. La tabla anterior muestra los detalles de cada distribución, y los posibles valores por defecto de los parámetros (si no se indica, significa que el parámetro debe ser especificado por el usuario).

Todas estas funciones se pueden usar reemplazando la letra `r` con las letras `d`, `p` o `q` para obtener, la densidad de probabilidad (`d`*`func`*`(x, ...)`), la densidad de probabilidad acumulada (`p`*`func`*`(x, ...)`), y el valor del cuartil (`q`*`func`*`(p, ...)`, con $0 <$ `p` < 1) respectivamente.

3.5. Manipulación de objetos

3.5.1. Creación de objetos

En las secciones anteriores vimos diferentes maneras de crear objetos usando el operador de asignación; el tipo y clase de los objetos así creados se determina generalmente de manera implícita. Es posible, sin embargo, generar un objeto especificando su clase, tipo, longitud, etc. Esta aproximación es interesante desde el punto de vista de la manipulación de objetos. Por ejemplo, se puede crear un objeto 'vacío' y modificar de manera sucesiva sus elementos; esto puede ser más eficiente que colocar todos los elementos juntos usando `c()`. El sistema de indexado se puede usar en estas circunstancias, como veremos más adelante (p. 23).

También puede ser bastante conveniente crear nuevos objetos a partir de objetos ya existentes. Por ejemplo, si se quiere ajustar una serie de modelos, es fácil colocar las fórmulas en una lista, y despues extraer sucesivamente los elementos para insertarlos en una función `lm`.

En esta etapa de nuestro aprendizaje de R, la utilidad de aprender las siguientes funciones no es solo práctica sino didáctica. La construcción explícita de un objeto nos proporciona un mejor entendimiento de su estructura, y nos permite ahondar en algunas nociones mencionadas previamente.

Vector. La función `vector`, que tiene dos argumentos `mode` y `length`, crea un vector cuyos elementos pueden ser de tipo numérico, lógico o caracter dependiendo del argumento especificado en `mode` (0, `FALSE` o " " respectivamente). Las siguientes funciones tienen exactamente el mismo efecto y tienen un solo argumento (la longitud del vector): `numeric()`, `logical()`, y `character()`.

Factor. Un factor incluye no solo los valores correspondientes a una variable categórica, pero también los diferentes niveles posibles de esta variable (inclusive si están presentes en los datos). La función `factor` crea un factor con las siguientes opciones:

```
factor(x, levels = sort(unique(x), na.last = TRUE),
       labels = levels, exclude = NA, ordered = is.ordered(x))
```

`levels` especifica los posibles niveles del factor (por defecto los valores únicos de `x`), `labels` define los nombres de los niveles, `exclude` especifica los valores `x` que se deben excluir de los niveles, y `ordered` es un argumento lógico que especifica si los niveles del factor están ordenados. Recuerde que `x` es de tipo numérico o caracter. Ejemplos:

```
> factor(1:3)
[1] 1 2 3
Levels:  1 2 3
> factor(1:3, levels=1:5)
[1] 1 2 3
Levels:  1 2 3 4 5
> factor(1:3, labels=c("A", "B", "C"))
[1] A B C
Levels:  A B C
> factor(1:5, exclude=4)
[1] 1  2  3  NA 5
Levels:  1 2 3 5
```

La función `levels` extrae los niveles posibles de un factor:

```
> ff <- factor(c(2, 4), levels=2:5)
> ff
[1] 2 4
Levels:  2 3 4 5
> levels(ff)
[1] "2" "3" "4" "5"
```

Matriz. Una matriz es realmente un vector con un atributo adicional (dim) el cual a su vez es un vector numérico de longitud 2, que define el número de filas y columnas de la matriz. Una matriz se puede crear con la función `matrix`:

```
matrix(data = NA, nrow = 1, ncol = 1, byrow = FALSE,
       dimnames = NULL)
```

La opción `byrow` indica si los valores en `data` deben llenar las columnas sucesivamente (por defecto) o las filas (if `TRUE`). La opción `dimnames` permite asignar nombres a las filas y columnas.

```
> matrix(data=5, nr=2, nc=2)
     [,1] [,2]
[1,]    5    5
[2,]    5    5
> matrix(1:6, 2, 3)
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
> matrix(1:6, 2, 3, byrow=TRUE)
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
```

Otra manera de crear una matriz es dando los valores apropiados al atributo dim (que inicialmente tiene valor `NULL`):

```
> x <- 1:15
> x
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
> dim(x)
NULL
> dim(x) <- c(5, 3)
> x
     [,1] [,2] [,3]
[1,]    1    6   11
[2,]    2    7   12
[3,]    3    8   13
[4,]    4    9   14
[5,]    5   10   15
```

Marco de datos. Hemos visto que un marco de datos ('data.frame') se crea de manera implícita con la función `read.table`; también es posible hacerlo con la función `data.frame`. Los vectores incluidos como argumentos deben ser de la misma longitud, o si uno de ellos es más corto que los otros, es "reciclado" un cierto número de veces:

```
> x <- 1:4; n <- 10; M <- c(10, 35); y <- 2:4
> data.frame(x, n)
  x  n
1 1 10
2 2 10
3 3 10
4 4 10
> data.frame(x, M)
  x  M
1 1 10
2 2 35
3 3 10
4 4 35
> data.frame(x, y)
Error in data.frame(x, y) :
    arguments imply differing number of rows: 4, 3
```

Si se incluye un factor en un marco de datos, el factor debe ser de la misma longitud que el vector (o vectores). Es posible cambiar el nombre de las columnas con `data.frame(A1=x, A2=n)`. También se pueden dar nombres a las filas con la opción `row.names` que debe ser, por supuesto, un vector de modo caracter con longitud igual al número de líneas en el marco de datos. Finalmente, note que los marcos de datos tienen un atributo similar al dim de las matrices.

Lista. Una lista se crea de manera similar a un marco de datos con la función `list`. No existe ninguna limitación en el tipo de objetos que se pueden incluir. A diferencia de `data.frame()`, los nombres de los objetos no se toman por defecto; tomando los vectores `x` y `y` del ejemplo anterior:

```
> L1 <- list(x, y); L2 <- list(A=x, B=y)
> L1
[[1]]
[1] 1 2 3 4

[[2]]
[1] 2 3 4

> L2
$A
[1] 1 2 3 4

$B
[1] 2 3 4

> names(L1)
```

```
NULL
> names(L2)
[1] "A" "B"
```

Series de tiempo. La función `ts` crea un objeto de clase `"ts"` (serie de tiempo) a partir de un vector (serie de tiempo única) o una matriz (serie multivariada). Las opciones que caracterizan un objeto de este tipo son:

```
ts(data = NA, start = 1, end = numeric(0), frequency = 1,
   deltat = 1, ts.eps = getOption("ts.eps"), class, names)
```

`data`	un vector o una matriz
`start`	el tiempo de la primera observación ya sea un número o un vector con dos enteros (ver ejemplo más abajo)
`end`	el tiempo de la última observación especificado de la misma manera que `start`
`frequency`	el número de observaciones por unidad de tiempo
`deltat`	la fracción del periodo de muestreo entre observaciones sucesivas (ej. `1/12` para datos mensuales); unicamente se debe especificar o `frequency` o `deltat`
`ts.eps`	tolerancia para la comparación de series. Las frecuencias se consideran iguales si su diferencia es menor que `ts.eps`
`class`	clase que se debe asignar al objeto; por defecto es `"ts"` para una serie univariada, y `c("mts", "ts")` para una serie multivariada
`names`	para una serie multivariada, un vector de tipo caracter con los nombres de las series individuales; por defecto los nombres de las columnas de `data`, o `Serie 1`, `Serie 2`, ...

Algunos ejemplos de series de tiempo creadas con `ts()`:

```
> ts(1:10, start = 1959)
Time Series:
Start = 1959
End = 1968
Frequency = 1
 [1]  1  2  3  4  5  6  7  8  9 10
> ts(1:47, frequency = 12, start = c(1959, 2))
     Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
1959       1   2   3   4   5   6   7   8   9  10  11
1960  12  13  14  15  16  17  18  19  20  21  22  23
1961  24  25  26  27  28  29  30  31  32  33  34  35
1962  36  37  38  39  40  41  42  43  44  45  46  47
> ts(1:10, frequency = 4, start = c(1959, 2))
     Qtr1 Qtr2 Qtr3 Qtr4
1959         1    2    3
1960    4    5    6    7
1961    8    9   10
> ts(matrix(rpois(36, 5), 12, 3), start=c(1961, 1), frequency=12)
         Series 1 Series 2 Series 3
Jan 1961        8        5        4
```

```
Feb 1961        6        6        9
Mar 1961        2        3        3
Apr 1961        8        5        4
May 1961        4        9        3
Jun 1961        4        6       13
Jul 1961        4        2        6
Aug 1961       11        6        4
Sep 1961        6        5        7
Oct 1961        6        5        7
Nov 1961        5        5        7
Dec 1961        8        5        2
```

Expresión. Los objetos de clase expresión juegan un papel fundamental en R. Una expresión es una serie de caracteres que hace sentido para R. Todos los comandos válidos son expresiones. Cuando se escribe un comando directamente en el teclado, este es *evaluado* por R y ejecutado si es válido. En muchos casos, es útil construir una expresión sin evaluarla: esto es lo que la función `expression` hace. Por supuesto, es posible evaluar la expresión posteriormente con `eval()`.

```
> x <- 3; y <- 2.5; z <- 1
> exp1 <- expression(x / (y + exp(z)))
> exp1
expression(x/(y + exp(z)))
> eval(exp1)
[1] 0.5749019
```

Las expresiones se pueden usar, entre otras cosas, para incluir ecuaciones en gráficos (p. 34). Una expresión se puede crear desde una variable de tipo caracter. Algunas funciones utilizan expresiones como argumentos; por ejemplo `D()` que calcula derivadas parciales:

```
> D(exp1, "x")
1/(y + exp(z))
> D(exp1, "y")
-x/(y + exp(z))^2
> D(exp1, "z")
-x * exp(z)/(y + exp(z))^2
```

3.5.2. Conversión de objetos

Para el lector debe ser obvio que las diferencias entre algunos tipos de objetos son pequeñas; por lo tanto debe ser relativamente fácil convertir el tipo de un objeto cambiando algunos de sus atributos. Tal conversión se puede realizar usando una función `as.algo`. R (versión 1.5.1) tiene en su paquete `base`, 77 funciones de este tipo, asi que no ahondaremos mucho más en este tema.

El resultado de esta conversión depende obviamente de los atributos del objeto convertido. Generalmente, las conversiones siguen reglas muy intuitivas. La siguiente tabla resume la situación para la conversión de diferentes tipos.

Conversión a	Función	Reglas
numérico	`as.numeric`	`FALSE` → 0
		`TRUE` → 1
		`"1"`, `"2"`, ... → 1, 2, ...
		"A", ... → `NA`
lógico	`as.logical`	0 → `FALSE`
		otros números → `TRUE`
		`"FALSE"`, `"F"` → `FALSE`
		`"TRUE"`, `"T"` → `TRUE`
		otros caracteres → `NA`
caracter	`as.character`	1, 2, ... → `"1"`, `"2"`, ...
		`FALSE` → `"FALSE"`
		`TRUE` → `"TRUE"`

Existen funciones que convierten entre diferentes clases de objetos (`as.matrix`, `as.data.frame`, `as.ts`, `as.expression`, ...). Estas funciones pueden afectar atributos diferentes al *tipo* durante la conversión. De nuevo, los resultados de la conversión son generalmente intuitivos. Una situación frecuente es la conversión de factores a valores numéricos. En este caso, R realiza la conversión usando la *codificación numérica* de los niveles del factor (no los valores literales del factor):

```
> fac <- factor(c(1, 10))
> fac
[1] 1 10
Levels:  1 10
> as.numeric(fac)
[1] 1 2
```

Para realizar la conversión manteniendo los valores literales del factor, primero se debe convertir a caracter y despues a numérico.

```
> as.numeric(as.character(fac))
[1] 1 10
```

Este procedimiento puede ser bastante útil si en un archivo una variable numérica también tiene valores no-numéricos. Vimos anteriormente que en esta situación `read.table()` leerá la columna como un factor por defecto.

3.5.3. Operadores

Previamente mencionamos que existen tres tipos de operadores en R[10]. Esta es la lista.

[10] Los siguientes caracteres también son operadores en R: `$`, `[`, `[[`, `:`, `?`, `<-`.

Operadores					
Aritméticos		Comparativos		Lógicos	
`+`	adición	`<`	menor que	`! x`	NO lógico
`-`	substracción	`>`	mayor que	`x & y`	Y lógico
`*`	multiplicación	`<=`	menor o igual que	`x && y`	id.
`/`	división	`>=`	mayor o igual que	`x | y`	O lógico
`^`	potencia	`==`	igual	`x || y`	id.
`%%`	módulo	`!=`	diferente de	`xor(x, y)`	O exclusivo
`%/%`	división de enteros				

Los operadores aritméticos y comparativos actúan en dos elementos (`x + y`, `a <b`). Los operadores aritméticos actúan sobre variables de tipo numérico o complejo, pero también lógico; en este caso los valores lógicos son forzados a valores numéricos. Los operadores comparativos pueden actuar sobre cualquier tipo devolviendo uno o varios valores lógicos.

Los operadores lógicos pueden actuar sobre uno (`!`) o dos objetos de tipo lógico, y pueden devolver uno (o varios) valores lógicos. Los operadores "Y" y "O" existen en dos formas: uno sencillo donde cada operador actúa sobre cada elemento del objeto y devuelve un número de valores lógicos igual al número de comparaciones realizadas; otro doble donde cada operador actúa solamente sobre el primer elemento del objeto.

Es necesario usar el operator "AND" para especificar una desigualdad del tipo $0 < x < 1$ la cual puede ser codificada como `0 <x & x <1`. La expresión `0 <x <1` es válida, pero no devuelve el resultado esperado debido a que ambos operadores son del mismo tipo y se ejecutan sucesivamente de izquierda a derecha. La comparación `0 <x` se realiza primero y el valor retornado es comparado con 1 (`TRUE` o `FALSE <1`): en este caso, el valor lógico es impícitamente forzado a numérico (`1` o `0 <1`).

Los operadores comparativos actúan sobre *cada* elemento de los dos objetos que se están comparando (reciclando los valores de los más pequeños si es necesario), devolviendo un objeto del mismo tamaño. Para comparar "totalmente" dos objetos es necesario usar la función `identical`:

```
> x <- 1:3; y <- 1:3
> x == y
[1] TRUE TRUE TRUE
> identical(x, y)
[1] TRUE
```

3.5.4. Cómo acceder los valores de un objeto: el sistema de indexación

El sistema de indexación es una manera eficiente y flexible de acceder selectivamente elementos de un objeto, y puede ser *numérico* o *lógico*. Por ejemplo, para acceder al tercer elemento de un vector `x`, simplemente se escribe `x[3]`. Si `x` es una matriz o un marco de datos el valor de la *i*ésima fila y la *j*ésima columna se accede con `x[i, j]`. Para cambiar todos los valores de la tercera columna,

```
> x[, 3] <- 10.2
```

El no especificar la fila incluye a todas. El sistema de indexación se puede generalizar fácilmente para matrices con más de dos dimensiones (por ejemplo una matriz tridimensional: `x[i, j, k]`, `x[, , 3]`, ...). Es importante recordar que la indexación se realiza con corchetes rectangulares, mientras que los paréntesis se usan para especificar los argumentos de una función:

```
> x(1)
Error: couldn’t find function "x"
```

La indexación se puede usar para suprimir una o mas filas o columnas. Por ejemplo, `x[-1, ]` suprime la primera fila, y `x[-c(1, 15), ]` hará lo mismo con la primera y la quinceava filas.

Para vectores, matrices y otros arreglos, es posible acceder los valores de un elemento usando como índice una expresión comparativa:

```
> x <- 1:10
> x[x >= 5] <- 20
> x
 [1]  1  2  3  4 20 20 20 20 20 20
> x[x == 1] <- 25
> x
 [1] 25  2  3  4 20 20 20 20 20 20
```

Un uso práctico de la indexación lógica es por ejemplo, la posibilidad de seleccionar los números pares de un vector de enteros:

```
> x <- rpois(40, lambda=5)
> x
 [1]  5  9  4  7  7  6  4  5 11  3  5  7  1  5  3  9  2  2  5  2
[21]  4  6  6  5  4  5  3  4  3  3  3  7  7  3  8  1  4  2  1  4
> x[x %% 2 == 0]
 [1] 4 6 4 2 2 2 4 6 6 4 4 8 4 2 4
```

Por lo tanto, el sistema de indexación utiliza los valores lógicos devueltos por los operadores comparativos. Estos valores se pueden calcular con anterioridad y pueden ser reciclados si es necesario:

```
> x <- 1:40
> s <- c(FALSE, TRUE)
> x[s]
 [1]  2  4  6  8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40
```

En este caso al aplicar la condición en `s` al vector `x` el primer elemento no se selecciona (`FALSE`), el segundo si (`TRUE`), el tercero no (`FALSE`), etc. Esta es una manera muy compacta y poderosa de hacer indexación selectiva sin utilizar bucles.

La indexación lógica se puede usar con marcos de datos, con la dificultad adicional que las diferentes columnas del marco pueden ser de diferentes tipos.

Para listas, es fácil acceder a diferentes elementos (que pueden ser de cualquier tipo) usando corchetes rectangulares dobles; por ejemplo `my.list[[3]]` accede al tercer elemento de `my.list`. El resultado puede ser indexado a su vez como se explicó anteriormente para vectores, matrices, etc. Si este tercer objeto es un vector, sus valores pueden ser modificados con `my.list[[3]][i]`, si es una matriz en tres dimensiones con `my.list[[3]][i, j, k]`, y asi sucesivamente.

3.5.5. Accediendo a los valores de un objeto con nombres

Hemos visto en múltiples ocasiones el concepto de *nombres*. Los nombres son atributos, y existen diferentes tipos (*names*, *colnames*, *rownames*, *dimnames*). Nos limitaremos por ahora a algunas nociones muy simples sobre los nombres, particularmente lo que concierne a su utilización para acceder a elementos de un objeto.

Si los elementos de un objeto tienen nombres, se pueden extraer usándolos como índices. Al realizar la extracción de esta manera los atributos del objeto original se mantienen intactos. Por ejemplo, si un marco de datos `DF` contiene las variables `x`, `y`, y `z`, el comando `DF["x"]` extraerá un marco de datos que solamente contendrá `x`; `DF[c(''x'', ''y'')]` extraerá un marco de datos con ambas variables. Esto funciona con listas si los elementos en la misma tienen nombres.

Como el lector se habrá dado cuenta, el índice utilizado aqui es un vector de modo caracter. Al igual que los vectores numéricos o lógicos descritos previamente, este vector se puede definir previamente y ser utilziado después para la extracción.

Para extraer un vector o un factor de un marco de datos se puede usar el símbolo `$` (e.g., `DF$x`). Este procedimiento también es válido para listas.

3.5.6. El editor de datos

Es posible utilziar un editor gráfico similar a una hoja de cálculo para editar un objeto numérico. Por ejemplo, si `X` es una matriz, el comando `data.entry(X)` abrirá un editor gráfico que le permitirá cambiar los valores en la matriz o adcionar nuevas columnas y/o filas.

La función `data.entry` modifica directamente el objeto dado como argumento sin necesidad de asignar su resultado. Por el contrario, la función `de` devuelve una lista con los objetos dados como argumentos y posiblemente modificados. Este resultado es mostrado en pantalla por defecto, pero como en muchas otras funciones, puede ser asignado a otro objeto.

Los detalles del uso del editor de datos dependen del sistema operativo (no está aún implementado en todas las plataformas).

3.5.7. Funciones aritméticas simples

Existen muchas funciones en R para manipular datos. Hemos visto la más sencila, `c` que concatena los objetos listados entre paréntesis. Por ejemplo:

```
> c(1:5, seq(10, 11, 0.2))
 [1]  1.0  2.0  3.0  4.0  5.0 10.0 10.2 10.4 10.6 10.8 11.0
```

Los vectores pueden ser manipulados con expresiones aritméticas clásicas:

```
> x <- 1:4
> y <- rep(1, 4)
> z <- x + y
> z
[1] 2 3 4 5
```

Se pueden adicionar vectores con diferentes longitudes; en este caso el vector más corto se recicla. Ejemplos:

```
> x <- 1:4
> y <- 1:2
> z <- x + y
```

```
> z
[1] 2 4 4 6
> x <- 1:3
> y <- 1:2
> z <- x + y
Warning message:
longer object length
 is not a multiple of shorter object length in: x + y
> z
[1] 2 4 4
```

Note que R ha devuelto un mensaje preventivo y no un mensaje de error; por lo tanto la operación fue realizada. Si queremos agregar (o multiplicar) un mismo valor a todos los elementos de un vector:

```
> x <- 1:4
> a <- 10
> z <- a * x
> z
[1] 10 20 30 40
```

El número de funciones disponibles en R es demasiado grande para ser listado en este documento. Se pueden encontrar todas las funciones matemáticas simples (`log`, `exp`, `log10`, `log2`, `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `abs`, `sqrt`, ...), funciones especiales (`gamma`, `digamma`, `beta`, `besselI`, ...), así como diversas funciones útiles en estadística. Algunas de estas funciones se detallan en la siguiente tabla.

`sum(x)`	suma de los elementos de `x`
`prod(x)`	producto de los elementos de `x`
`max(x)`	valor máximo en el objeto `x`
`min(x)`	valor mínimo en el objeto `x`
`which.max(x)`	devuelve el índice del elemento máximo de `x`
`which.min(x)`	devuelve el índice del elemento mínimo de `x`
`range(x)`	rango de `x` o `c(min(x), max(x))`
`length(x)`	número de elementos en `x`
`mean(x)`	promedio de los elementos de `x`
`median(x)`	mediana de los elementos de `x`
`var(x)` o `cov(x)`	varianza de los elementos de `x` (calculada en $n-1$); si `x` es una matriz o un marco de datos, se calcula la matriz de varianza-covarianza
`cor(x)`	matriz de correlación de `x` si es una matriz o un marco de datos (1 si `x` es un vector)
`var(x, y)` o `cov(x, y)`	covarianza entre `x` y `y`, o entre las columnas de `x` y `y` si son matrices o marcos de datos
`cor(x, y)`	correlación lineal entre `x` y `y`, o la matriz de correlación si `x` y `y` son matrices o marcos de datos

Estas funciones devuelven un solo valor (o un vector de longitud 1), a excepción de `range()` que retorna un vector de longitud 2, y `var()`, `cov()`, y `cor()` que pueden devolver matrices. Las siguientes funciones pueden devolver vectores más complejos:

`round(x, n)`	redondea los elementos de `x` a `n` cifras decimales
`rev(x)`	invierte el orden de los elementos en `x`
`sort(x)`	ordena los elementos de `x` en orden ascendente; para hacerlo en orden descendente: `rev(sort(x))`
`rank(x)`	alinea los elementos de `x`
`log(x, base)`	calcula el logaritmo de `x` en base `"base"`
`scale(x)`	si `x` es una matriz, centra y reduce los datos; si se desea centrar solamente utilizar `scale=FALSE`, para reducir solamente usar `center=FALSE` (por defecto `center=TRUE, scale=TRUE`)
`pmin(x,y,...)`	un vector en el que el *i*avo elemento es el mínimo de `x[i]`, `y[i]`, ...
`pmax(x,y,...)`	igual que el anterior pero para el máximo
`cumsum(x)`	un vector en el que el *i*avo elemento es la suma desde `x[1]` a `x[i]`
`cumprod(x)`	igual que el anterior pero para el producto
`cummin(x)`	igual que el anterior pero para el mínimo
`cummax(x)`	igual que el anterior pero para el máximo
`match(x, y)`	devuelve un vector de la misma longitud que `x` con los elementos de `x` que están en `y` (`NA` si no)
`which(x == a)`	devuelve un vector de los índices de `x` si la operación es (`TRUE`) (en este ejemplo, los valores de `i` para los cuales `x[i] == a`). El argumento de esta función debe ser una variable de tipo lógico
`choose(n, k)`	calcula el número de combinaciones de *k* eventos en *n* repeticiones = $n!/[(n-k)!k!]$
`na.omit(x)`	elimina las observaciones con datos ausentes (`NA`) (elimina la fila correspondiente si `x` es una matriz o un marco de datos)
`na.fail(x)`	devuelve un mensaje de error si `x` contiene por lo menos un `NA`
`unique(x)`	si `x` es un vector o un marco de datos, devuelve un objeto similar pero suprimiendo elementos duplicados
`table(x)`	devuelve una tabla con el número de diferentes valores de `x` (típicamente para enteros o factores)
`subset(x, ...)`	devuelve una selección de `x` con respecto al criterio (`...`, típicamente comparaciones: `x$V1 <10`); si `x` es un marco de datos, la opción `select` proporciona las variables que se mantienen (o se ignoran con `-`)
`sample(x, size)`	remuestrea al azar y sin reemplazo `size` elementos en el vector `x`; la opción `replace = TRUE` permite remuestrear con reemplazo

3.5.8. Cálculos con Matrices

R posee facilidades para manipular y hacer operaciones con matrices. Las funciones `rbind()` y `cbind()` unen matrices con respecto a sus filas o columnas respectivamente:

```
> m1 <- matrix(1, nr = 2, nc = 2)
> m2 <- matrix(2, nr = 2, nc = 2)
> rbind(m1, m2)
     [,1] [,2]
[1,]    1    1
[2,]    1    1
[3,]    2    2
[4,]    2    2
> cbind(m1, m2)
     [,1] [,2] [,3] [,4]
[1,]    1    1    2    2
[2,]    1    1    2    2
```

El operador para el producto de dos matrices es '`%*%`'. Por ejemplo, considerando las dos matrices `m1` y `m2`:

```
> rbind(m1, m2) %*% cbind(m1, m2)
     [,1] [,2] [,3] [,4]
[1,]    2    2    4    4
[2,]    2    2    4    4
[3,]    4    4    8    8
[4,]    4    4    8    8
> cbind(m1, m2) %*% rbind(m1, m2)
     [,1] [,2]
[1,]   10   10
[2,]   10   10
```

La transposición de una matriz se realiza con la función `t`; esta función también funciona con marcos de datos.

La función `diag` se puede usar para extraer o modificar la diagonal de una matriz o para construir una matriz diagonal:

```
> diag(m1)
[1] 1 1
> diag(rbind(m1, m2) %*% cbind(m1, m2))
[1] 2 2 8 8
> diag(m1) <- 10
> m1
     [,1] [,2]
[1,]   10    1
[2,]    1   10
> diag(3)
     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    1    0
[3,]    0    0    1
> v <- c(10, 20, 30)
> diag(v)
     [,1] [,2] [,3]
[1,]   10    0    0
[2,]    0   20    0
[3,]    0    0   30
> diag(2.1, nr = 3, nc = 5)
     [,1] [,2] [,3] [,4] [,5]
[1,]  2.1  0.0  0.0    0    0
[2,]  0.0  2.1  0.0    0    0
[3,]  0.0  0.0  2.1    0    0
```

R también posee algunas fucniones para cálculos con matrices. Mencionamos aqui `solve()` para invertir una matriz, `qr()` para descomposición, `eigen()` para calcular valores y vectores propios, y `svd()` para descomposición singular.

4. Haciendo gráficas en R

R ofrece una increible variedad de gráficos. Para darse una idea, escriba el comando `demo(graphics)`. No nos es posible detallar aqui todas las posibilidades de R en términos de generación de gráficas. 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.

El *modus operandi* de las funciones gráficas es sustancialmente diferente del esquema esbozado al principio de este documento. Particularmente, el resultado de una función gráfica no puede ser asignado a un objeto[11] sino que es enviado a un *dispositivo gráfico*. Un dispositivo gráfico es una ventana gráfica o un archivo.

Existen dos tipos de funciones gráficas: las *funciones de graficación de alto nivel* que crean una nueva gráfica y las *funciones de graficación 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`.

Primero veremos como manejar gráficos y dispositivos gráficos; después veremos en detalle algunas funciones gráficas y sus parámetros. Veremos ejemplos prácticos del uso de estas funcionalidades en la producción de gráficos. Finalmente, veremos los paquetes grid y lattice cuyo funcionamiento es diferente a las funciones gráficas 'normales' de R.

4.1. Manejo de gráficos

4.1.1. Abriendo múltiples dispositivos gráficos

Al ejecutarse una función gráfica, R abre una ventana para mostrar el gráfico si no hay ningún dispositivo abierto. Un dispostivo gráfico se puede abrir con la función apropiada. El tipo de dispositivos gráficos disponibles depende del sistema operativo. Las ventanas gráficas se llaman `X11` bajo Unix/Linux, `windows` bajo Windows y `macintosh` bajo Mac. En Unix/Linux y Windows se puede abrir una nueva ventana gráfica con el comando `x11()` ya que en Windows existe un alias apuntando a `windows()`. Dispositivos gráficos que son archivos se pueden abrir con una función que depende del tipo de archivo que se quiere crear: `postscript()`, `pdf()`, `png()`, ... La lista de dispositivos gráficos disponibles se obtiene con el comando `?device`.

El último dispositivo en ser abierto, se convierte en el dispositivo activo sobre el cual se dibujan (o escriben) las gráficas generadas. La función `dev.list()` muestra una lista con los dispositivos abiertos:

```
> x11(); x11(); pdf()
> dev.list()
X11 X11 pdf
  2   3   4
```

Los números corresponden al número del dispositivo respectivo. Este se puede usar para cambiar el dispositivo activo. Para saber cual es el dispositivo activo:

```
> dev.cur()
pdf
  4
```

y para cambiar el dispositivo activo:

[11] Existen algunas excepciones: `hist()` y `barplot()` producen también resultados numéricos como listas o matrices.

```
> dev.set(3)
X11
  3
```

La función `dev.off()` cierra el dispositivo; por defecto se cierra el dispositivo activo, de lo contrario el correspondiente al número pasado en la función. R muestra el número del nuevo dispositivo activo:

```
> dev.off(2)
X11
  3
> dev.off()
pdf
  4
```

Vale la pena mencionar dos características específicas de la versión de R para Windows: 1) la función `win.metafile` abre un dispositivo meta-archivo de Windows , y 2) el menú "History" seleccionado cuando la ventana gráfica está activa, permite 'grabar' todas las gráficas durante una sesión (por defecto, esta opción está inactiva, pero el usuario la puede activar haciendo click en "Recording" en este menú).

4.1.2. Disposición de una gráfica

La función `split.screen` divide el dispositivo gráfico activo. Por ejemplo:

```
> split.screen(c(1, 2))
```

divide el dispositivo en dos partes que se pueden seleccionar con `screen(1)` o `screen(2)`; `erase.screen()` borra la última gráfica dibujada. Una parte de un dispositivo se puede dividir a su vez en partes más pequeñas con `split.screen()` permitiendo la posibilidad de configuraciones complejas.

Estas funciones son incompatibles con otras similares (como `layout()` o `coplot()`) y no se deben usar con multiples dispositivos gráficos. Su uso se debe limitar por ejemplo, a la exploración gráfica de datos.

La función `layout` divide el dispositivo activo en varias partes donde se colocarán las gráficas de manera sucesiva. Esta función tiene como argumento principal una matriz con números enteros indicando el número de las sub-ventanas. Por ejemplo, para dividir el dispositivo en cuatro partes iguales:

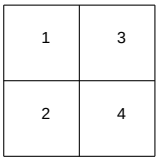
```
> layout(matrix(1:4, 2, 2))
```

Tambien es posible crear esta matriz previamente permitiendo una mejor visualización de la manera como se va a dividir el dispositivo:

```
> mat <- matrix(1:4, 2, 2)
> mat
     [,1] [,2]
[1,]    1    3
[2,]    2    4
> layout(mat)
```

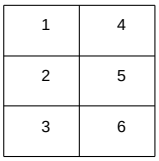
Para visualizar la partición creada, se puede usar la función `layout.show` con el número de sub-ventanas como argumento. En este ejemplo tenemos:

```
>layout.show(4)
```

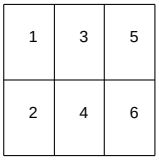


Los siguientes ejemplos demuestran algunas de las posibilidades ofrecidas por `layout()`.

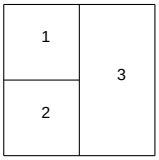
```
>layout(matrix(1:6, 3, 2))
>layout.show(6)
```




```
>layout(matrix(1:6, 2, 3))
>layout.show(6)
```




```
>m <- matrix(c(1:3, 3), 2, 2)
>layout(m)
>layout.show(3)
```

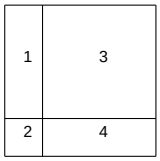


En ninguno de estos ejemplos hemos usado la opción `byrow` de `matrix()` (leer por filas), y por lo tanto las sub-ventanas se numeran a lo largo de las columnas; se puede especificar `matrix(..., byrow=TRUE)` para numerar las sub-ventanas a lo largo de las filas. Los números en la matriz se pueden dar en cualquier orden, por ejemplo `matrix(c(2, 1, 4, 3), 2, 2)`.

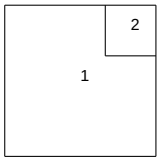
Por defecto, `layout()` divide el dispositivo en dimensiones regulares: esto se puede modificar con las opciones `widths` y `heights`. Estas dimensiones se dan de manera relativa[12]. Ejemplos:

[12] Aunque pueden ser proporcionadas en centímetros, ver `?layout`.

```
>m <- matrix(1:4, 2, 2)
>layout(m, widths=c(1, 3),
          heights=c(3, 1))
>layout.show(4)
```

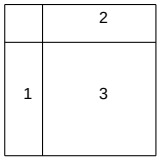



```
>m <- matrix(c(1,1,2,1),2,2)
>layout(m, widths=c(2, 1),
          heights=c(1, 2))
>layout.show(2)
```

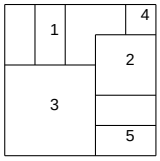


Finalmente, los números en la matriz pueden ser ceros dando la posibilidad de realizar arreglos complejos (o inclusive esotéricos).

```
>m <- matrix(0:3, 2, 2)
>layout(m, c(1, 3), c(1, 3))
>layout.show(3)
```




```
>m <- matrix(scan(), 5, 5)
1: 0 0 3 3 3 1 1 3 3 3
11: 0 0 3 3 3 0 2 2 0 5
21: 4 2 2 0 5
26:
Read 25 items
>layout(m)
>layout.show(5)
```



4.2. Funciones gráficas

La tabla a continuación resume algunas de las funciones gráficas en R.

`plot(x)`	graficar los valores de `x` (en el eje *y*) ordenados en el eje *x*
`plot(x, y)`	gráfico bivariado de `x` (en el eje *x*) y `y` (en el eje *y*)
`sunflowerplot(x, y)`	igual a `plot()` pero los puntos con coordenadas similares se dibujan como flores con el número de pétalos igual al número de puntos
`piechart(x)`	gráfico circular tipo 'pie'
`boxplot(x)`	gráfico tipo 'box-and-whiskers'
`stripplot(x)`	gráfico de los valores de `x` en una línea (como alternativa a `boxplot()` para pequeños tamaños de muestra)
`coplot(x~y | z)`	gráfico bivariado de `x` y `y` para cada valor o intervalo de valores de `z`
`interaction.plot (f1, f2, y)`	si `f1` y `f2` son factores, grafica el promedio de `y` (en el eje *y*) con respecto a los valores de `f1` (en el eje *x*) y de `f2` (curvas diferentes); la opción `fun` permite escoger un estadístico de `y` (por defecto, el promedio: `fun=mean`)
`matplot(x,y)`	gráfica bivariada de la primera columna de `x` *vs.* la primera columna de `y`, la segunda columna de `x` *vs.* la segunda columna de `y`, etc.
`dotplot(x)`	si `x` es un marco de datos, hace un gráfico de puntos tipo Cleveland (gráficos apilados fila por fila y columna por columna)

`fourfoldplot(x)`	utilizando cuartos de círculos, visualiza la asociación entre dos variables dicotómicas para diferentes poblaciones (`x` debe ser un arreglo de `dim=c(2, 2, k)`, o una matriz de `dim=c(2, 2)` si $k = 1$)
`assocplot(x)`	Gráfica 'amigable' de Cohen mostrando desviaciones de independencia de filas y columnas en una tabla de contingencia de dos dimensiones
`mosaicplot(x)`	gráfico 'mosaico' de los residuales de una regresión log-lineal de una tabla de contingencia
`pairs(x)`	si `x` es una matriz o un marco de datos, dibuja todas las posibles gráficas bivariadas entre las columnas de `x`
`plot.ts(x)`	si `x` es un objeto de clase `"ts"`, grafica `x` con respecto al tiempo. `x` puede ser multivariada pero las series deben tener la misma frecuencia y fechas
`ts.plot(x)`	igual a la anterior pero si `x` es multivariado, las series pueden tener diferentes fechas pero la misma frecuencia
`hist(x)`	histograma de las frecuencias de `x`
`barplot(x)`	histograma de los valores de `x`
`qqnorm(x)`	cuartiles de `x` con respecto a lo esperado bajo una distribución normal
`qqplot(x, y)`	cuartiles de `y` con respecto a los cuartiles de `x`
`contour(x, y, z)`	gráfico de contornos (los datos son interpolados para dibujar las curvas), `x` y `y` deben ser vectores, `z` debe ser una matriz tal que `dim(z)=c(length(x), length(y))` (`x` y `y` pueden ser omitidos)
`filled.contour (x, y, z)`	igual al anterior, pero las áreas entre contornos están coloreadas, y se dibuja una leyenda de colores
`image(x, y, z)`	igual al anterior pero con colores (se grafican los datos crudos)
`persp(x, y, z)`	igual al anterior pero en perspectiva (se grafican los datos crudos)
`stars(x)`	si `x` es una matriz o un marco de datos, dibuja una gráfica con segmentos o una estrella, donde cada fila de `x` es representada por una estrella, y las columnas son las longitudes de los segmentos
`symbols(x, y, ...)`	dibuja, en las coordenadas dadas por `x` y `y`, símbolos (círculos, cuadrados, rectángulos, estrellas, termómetros o cajas) cuyos tamaños, colores ... son especificados con argumentos adicionales
`termplot(mod.obj)`	gráfico de los efectos (parciales) de un modelo de regresión (`mod.obj`)

Las opciones y argumentos para cada una de estas opciones se pueden encontrar en la ayuda incorporada en R. Algunas de estas opciones son idénticas para varias funciones gráficas; éstas son las principales (con sus valores por defecto):

`add=FALSE`	si es `TRUE` superpone el gráfico en el ya existente (si existe)
`axes=TRUE`	si es `FALSE` no dibuja los ejes ni la caja del gráfico
`type="p"`	especifica el tipo de gráfico; `"p"`: puntos, `"l"`: líneas, `"b"`: puntos conectados por líneas, `".o"`: igual al anterior, pero las líneas están sobre los puntos, `"h"`: líneas verticales, `"s"`: escaleras, los datos se representan como la parte superior de las líneas verticales, `"S"`: igual al anterior pero los datos se representan como la parte inferior de las líneas verticales
`xlim=, ylim=`	especifica los límites inferiores y superiores de los ejes; por ejemplo con `xlim=c(1, 10)` o `xlim=range(x)`
`xlab=, ylab=`	tı'tulos en los ejes; deben ser variables de tipo caracter
`main=`	título principal; debe ser de tipo caracter
`sub=`	sub-título (escrito en una letra más pequeña)

4.3. Comandos de graficación de bajo nivel

R posee un conjunto de funciones gráficas que afectan una gráfica ya existente: *comandos de graficación de bajo nivel*. Estos son los principales:

`points(x, y)`	agrega puntos (se puede usar la opción `type=`)
`lines(x, y)`	igual a la anterior pero con líneas
`text(x, y, labels, ...)`	agrega texto dado por `labels` en las coordenadas (`x`,`y`); un uso típico: `plot(x, y, type="n"); text(x, y, names)`
`mtext(text, side=3, line=0, ...)`	agrega texto dado por `text` en el margen especificado por `side` (ver `axis()` más abajo); `line` especifica la linea del área de graficado
`segments(x0, y0, x1, y1)`	dibuja una línea desde el punto (`x0`,`y0`) hasta el punto (`x1`,`y1`)
`arrows(x0, y0, x1, y1, angle= 30, code=2)`	igual al anterior pero con flechas desde (`x0`,`y0`) si `code=2`, al punto (`x1`,`y1`) si `code=1`, o en ambos si `code=3`; `angle` controla el ángulo desde la base de la flecha hasta la punta de la misma
`abline(a,b)`	dibuja una línea con pendiente `b` e intercepto `a`
`abline(h=y)`	dibuja una línea horizontal en la ordenada `y`
`abline(v=x)`	dibuja una línea vertical en la abcisa `x`
`abline(lm.obj)`	dibuja la línea de regresión dada por `lm.obj` (ver sección 5)
`rect(x1, y1, x2, y2)`	dibuja un rectángulo donde las esquinas izquierda, derecha, superior e inferior están dadas por `x1`, `x2`, `y1`, y `y2`, respectivamente
`polygon(x, y)`	dibuja un polígono uniendo los puntos dados por `x` y `y`
`legend(x, y, legend)`	agrega la leyenda en el punto (`x`,`y`) con símbolos dados por `legend`
`title()`	agrega un título y opcionalmente un sub-título
`axis(side, vect)`	agrega un eje en la parte inferior (`side=1`), izquierda (2), superior (3), o derecha (4); `vect` (opcional) da la abscisa (u ordenada) donde se deben dibujar los marcadores ('tick marks') del eje
`rug(x)`	dibuja los datos `x` en el eje *x* como pequeñas líneas verticales
`locator(n, type="n", ...)`	devuelve las coordenadas (x,y) después que el usuario a hecho click `n` veces en el gráfico con el ratón; también dibuja símbolos (`type="p"`) o líneas (`type="l"`) con respecto a parámetros gráficos opcionales (`...`); por defecto no se dibuja nada (`type="n"`)
`identify(x, ...)`	similar a `locator()` con la diferencia que imprime en la gráfica el valor de x (u opcionalmente de una leyenda especificada en la opción `labels=`) más cercano al punto donde se hizo click. Util para identificar puntos en la gráfica que están asociados con nombres.

Note la posibilidad de agregar expresiones matemáticas en una gráfica con `text(x, y, expression(...))`, donde la función `expression` transforma su argumento en una ecuación matemática. Por ejemplo,

```
> text(x, y, expression(p == over(1, 1+eˆ-(beta*x+alpha))))
```

se verá en la gráfica como la siguiente ecuación en el punto de coordenadas (x,y):

$$\mathrm{p} = \frac{1}{1+\mathrm{e}^{-(\beta \mathrm{x}+\alpha)}}$$

Para incluir una variable en una expresión, se pueden utilizar las funciones `substitute` y `as.expression`; por ejemplo para incluir el valor de R^2 (calculado anteriomente y guardado en un objeto `Rsquared`):

```
> text(x, y, as.expression(substitute(Rˆ2==r, list(r=Rsquared))))
```

se verá en la gráfica en el punto con coordenadas (x,y):

$$\mathrm{R}^2 = 0{,}9856298$$

Para ver solo tres cifras decimales, podemos modificar el código de la siguiente manera:

```
> text(x, y, as.expression(substitute(R^2==r,
+                          list(r=round(Rsquared, 3)))))
```

que se verá como:

$$\mathrm{R}^2 = 0{,}986$$

Finalmente para escribir la R en cursivas:

```
> text(x, y, as.expression(substitute(italic(R)^2==r,
+                          list(r=round(Rsquared, 3)))))
```

$$R^2 = 0{,}986$$

4.4. Parámetros gráficos

Además de la utilización de comandos de graficación de bajo nivel, la presentación de gráficos se puede mejorar con parámetros gráficos adicionales. Estos se pueden utilizar como opciones de funciones gráficas (pero no funciona para todas), o usando la función `par` para cambiar de manera permanente parámetros gráficos; es decir gráficas subsecuentes se dibujarán con respecto a los parámetros especificados por el usuario. Por ejemplo, el siguiente comando:

```
> par(bg="yellow")
```

dará como resultado que todos los gráficos subsecuentes tendrán el fondo de color amarillo. Existen 68 parámetros gráficos y algunos tienen funciones muy similares. La lista completa de parámetros gráficos se puede ver con `?par`; en la siguiente tabla ilustramos solo los más usados.

`adj`	controla la justificación del texto (`0` justificado a la izquierda, `0.5` centrado, `1` justificado a la derecha)
`bg`	especifica el color del fondo (ej. : `bg="red"`, `bg="blue"`, ... La lista de los 657 colores disponibles se puede ver con `colors()`)
`bty`	controla el tipo de caja que se dibuja alrededor del gráfico: `"o"`, `"l"`, `"7"`, `"c"`, `"u"` o `"]"` (la caja se parece a su respectivo caracater); si `bty="n"` no se dibuja la caja
`cex`	un valor que controla el tamaño del texto y símbolos con respecto al valor por defecto; los siguientes parámetros tienen el mismo control para números en los ejes, `cex.axis`, títulos en los ejes, `cex.lab`, el título principal, `cex.main`, y el subtítulo, `cex.sub`
`col`	controla el color de los símbolos; como en `cex` estos son: `col.axis`, `col.lab`, `col.main` y `col.sub`
`font`	un entero que conrola el estilo del texto (`1`: normal, `2`: cursiva, `3`: negrilla, `4`: negrilla cursiva); como en `cex` existen: `font.axis`, `font.lab`, `font.main` y `font.sub`
`las`	un entero que controla la orientación de los caracteres en los ejes (`0`: paralelo a los ejes, `1`: horizontal, `2`: perpendicular a los ejes, `3`: vertical)
`lty`	un entero o caracter que controla el tipo de las líneas; (`1`: sólida, `2`: quebrada, `3`: punteada, `4`: punto-línea, `5`: línea larga-corta, `6`: dos líneas cortas), o una secuencia de hasta 8 caracteres (entre `"0"` y `"9"`) que especifica alternativamente la longitud en puntos o pixeles, de los elementos dibujados y los blancos; por ejemplo `lty="44"` tendrá el mismo efecto que `lty=2`
`lwd`	un número que controla la anchura de las líneas
`mar`	un vector con 4 valores numéricos que controla el espacio entre los ejes y el borde de la gráfica en la forma `c(inferior, izquierda, superior, derecha)`; los valores por defecto son `c(5.1, 4.1, 4.1, 2.1)`
`mfcol`	un vector del tipo `c(nr,nc)` que divide la ventana gráfica como una matriz con `nr` filas y `nc` columnas; las gráficas se dibujan sucesivamente en las columnas (véase la sección 4.1.2)
`mfrow`	igual al anterior, pero las gráficas se dibujan en las filas (ver sección 4.1.2)
`pch`	controla el tipo de símbolo, ya sea un entero entre 1 y 25, o un caracter entre " (Fig. 2)
`ps`	un entero que controla el tamaño (en puntos) de textos y símbolos
`pty`	un caracter que especifica el tipo de región a graficar, `"s"`: cuadrada, `"m"`: máxima
`tck`	un valor que especifica la longitud de los marcadores de eje como una fracción de la altura o anchura máxima del gráfico; si `tck=1` se dibuja una rejilla
`tcl`	un valor que especifica la longitud de los marcadores de eje como una fracción de la altura de una línea de texto (por defecto `tcl=-0.5`)
`xaxt`	si `xaxt="n"` el eje *x* se coloca pero no se muesttra (util en conjunción con `axis(side=1, ...)`)
`yaxt`	if `yaxt="n"` el eje *y* se coloca pero no se muesttra (util en conjunción con `axis(side=2, ...)`)

4.5. Un ejemplo práctico

Para ilustrar algunas de las funcionalidades gráficas de R, consideremos un ejemplo simple de una gráfica bivariada con 10 pares de coordenadas generadas al azar. Estos valores se generaron con:

```
> x <- rnorm(10)
> y <- rnorm(10)
```

La gráfica que queremos visualizar se puede obtener con `plot()`; simplemente se escribe el comando:

```
> plot(x, y)
```

y la gráfica será visible en el dispositivo gráfico activo. El resultado se puede ver en la Fig. 3. Por defecto, R dibuja gráficas de una manera "inteligente": los espacios entre los marcadores de los ejes, la ubicación de las etiquetas en los ejes, etc, son calculados automáticamente de tal manera que la gráfica resultante sea lo mas legible posible.

Sin embargo, el usuario puede cambiar la manera como se presenta la gráfica, por ejemplo, para ajustarse a un estilo editorial pre-definido o para darle un toque personal para una charla. La

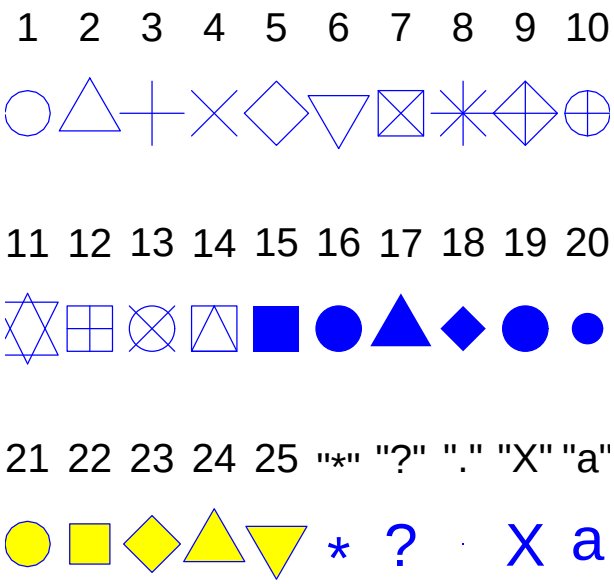


Figura 2: Los símbolos gráficos en R (`pch=1:25`). Los colores se obtuvieron con las opciones `col="blue", bg=" yellow"`; la segunda opción tiene efecto solo sobre los símbolos 21–25. Se puede usar cualquier caracter (`pch="*"`, `"?"`, `"."`, ...).

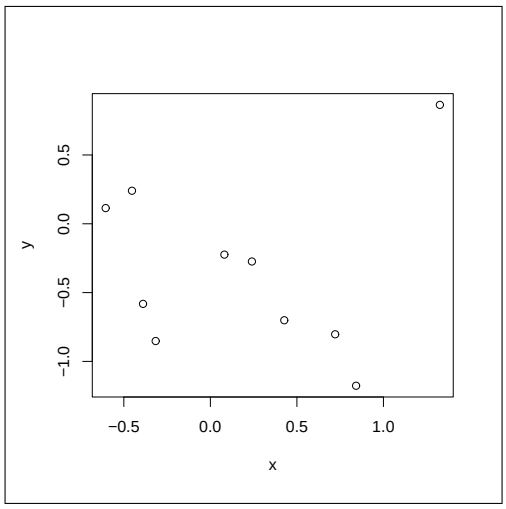


Figura 3: La función `plot` usada sin opciones.

Figura 4: La función `plot` usada con opciones.

manera más simple de cambiar un gráfico es a través de la adición de opciones que permiten modificar los argumentos dados por defecto. En nuestro ejemplo, podemos modificar significativamente la figura de la siguiente manera:

```
plot(x, y, xlab="Diez numeros al azar", ylab="Otros diez numeros",
     xlim=c(-2, 2), ylim=c(-2, 2), pch=22, col="red",
     bg="yellow", bty="l", tcl=0.4,
     main="Como personalizar un grafico en R", las=1, cex=1.5)
```

El resultado se ve en la Fig. 4. Miremos con detalle cada una de las opciones utilizadas. Primero, `xlab` y `ylab` cambian los títulos de los ejes, que por defecto son los nombres de las variables. Ahora, `xlim` y `ylim` nos permiten definir los límites en ambos ejes[13]. El parámetro gráfico `pch` es utilizado como una opción: `pch=22` especifica un cuadrado con contorno y fondo de diferentes colores dados respectivamente por `col` and `bg`. La tabla de parámetros gráficos explica el significado de las modificaciones hechas por `bty`, `tcl`, `las` and `cex`. Finalmente, adicionamos un título con `main`.

Los parámetros gráficos y las funciones de graficación de bajo nivel nos permiten modificar aún más la presentación de un gráfico. Como vimos anteriormente, algunos parámetros gráficos no se pueden pasar como argumentos en funciones como `plot`. Modificaremos algunos de estos parámetros con `par()`, y por ende es necesario escribir varios comandos. Cuando se cambian los parámetros gráficos es útil guardar sus valores iniciales previamente para poder restaurarlos posteriormente. Aqui están los comandos utilzados para obtener la Fig. 5.

```
opar <- par()
par(bg="lightyellow", col.axis="blue", mar=c(4, 4, 2.5, 0.25))
plot(x, y, xlab="Diez numeros al azar", ylab="Otros diez numeros",
     xlim=c(-2, 2), ylim=c(-2, 2), pch=22, col="red", bg="yellow",
     bty="l", tcl=-.25, las=1, cex=1.5)
title("Como personalizar un grafico en R (bis)", font.main=3, adj=1)
par(opar)
```

[13] Por defecto, R agrega 4 % a cada lado del límite del eje. Este comportamiento se puede alterar modificando los parámetros gráficos `xaxs="i"` y `yaxs="i"` (se pueden pasar como opciones a `plot()`).

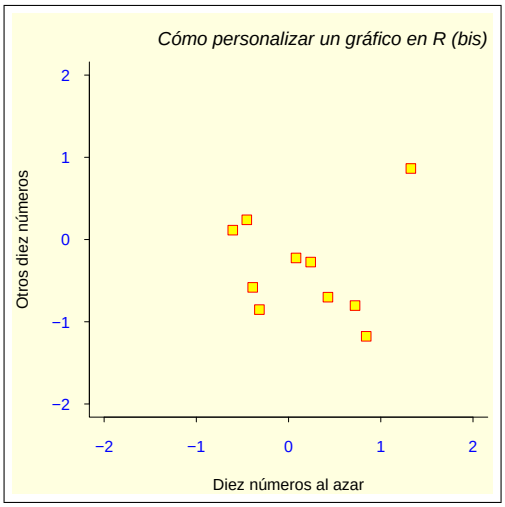


Figura 5: Las funciones `par`, `plot` y `title`.

Miremos con detalle las acciones resultantes de estos comandos. Primero, los parámetros gráficos por defecto se copian en una lista llamada `opar`. Modificaremos tres parámetros: `bg` el color del fondo, `col.axis` el color de los números en los ejes, y `mar` los tamaños de los márgenes en los bordes del gráfico. La gráfica se dibuja de una manera muy similar a como se hizo la Fig. 4. las modificaciones de los márgenes permiten utilizar el espacio alrededor del área de graficado. El título se añade con la función de bajo nivel `title` lo que permite agregar algunos parámetros como argumentos sin alterar el resto de la gráfica. Finalmente, los parámetros gráficos iniciales se restauran con el último comando.

Ahora, control total! En la Fig. 5, R todavía determina algunas cosas tales como el número de marcadores en los ejes, o el espacio entre el título y el área de graficado. Veremos ahora como controlar totalmente la presentación de la gráfica. La estrategia que vamos a usar aqui es dibujar primero una gráfica en blanco con `plot(..., type="n")`, y despues agregar puntos, ejes, etiquetas, etc., con funciones de graficación de bajo nivel. Incluiremos algunas novedades como cambiar el color del área de graficado. Los comandos se encuentran a continuación y la gráfica resultante se puede ver en la Fig. 6.

```
opar <- par()
par(bg="lightgray", mar=c(2.5, 1.5, 2.5, 0.25))
plot(x, y, type="n", xlab="", ylab="", xlim=c(-2, 2),
     ylim=c(-2, 2), xaxt="n", yaxt="n")
rect(-3, -3, 3, 3, col="cornsilk")
points(x, y, pch=10, col="red", cex=2)
axis(side=1, c(-2, 0, 2), tcl=-0.2, labels=FALSE)
axis(side=2, -1:1, tcl=-0.2, labels=FALSE)
title("Como personalizar un grafico en R (ter)",
      font.main=4, adj=1, cex.main=1)
mtext("Diez numeros al azar", side=1, line=1, at=1, cex=0.9, font=3)
mtext("Otros diez numeros", line=0.5, at=-1.8, cex=0.9, font=3)
mtext(c(-2, 0, 2), side=1, las=1, at=c(-2, 0, 2), line=0.3,
      col="blue", cex=0.9)
mtext(-1:1, side=2, las=1, at=-1:1, line=0.2, col="blue", cex=0.9)
par(opar)
```

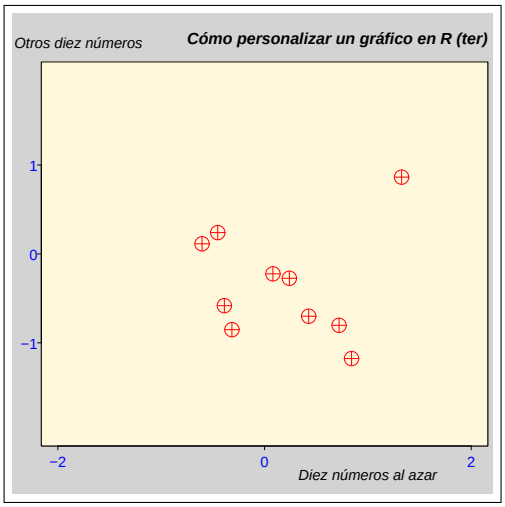


Figura 6: Una gráfica “hecha a mano”.

Como se hizo anteriormente los parámetros gráficos por defecto se guardan y el color del fondo y las márgenes se modifican. La gráfica se dibuja con `type="n"` para no colocar los puntos, `xlab=“”,ylab=“”` para no escribir títulos en los ejes, y `xaxt="n", yaxt="n"` para no dibujar los ejes. Esto resulta en que se dibuja solamente la caja alrededor del área de graficado y se definen los ejes con respecto a `xlim` y `ylim`. Note que hubiéramos podido usar la opción `axes=FALSE` pero en ese caso, ni los ejes ni la caja hubieran sido dibujados.

Los elementos se adicionan en la región de la gráfica con la ayuda de las funciones de bajo nivel. Antes de agregar los puntos el color dentro del área de graficado se cambia con `rect()`: el tamaño del rectángulo se escoge de tal manera que es substanciamente más grande que el área de graficado.

Los puntos se dibujan con `points()` utilizando un nuevo símbolo. Los ejes se agregan con `axis()`: el vector especificado como segundo argumento determina las coordenadas de los marcadores de los ejes. La opción `labels=FALSE` especifica que no se deben escribir anotaciones con los marcadores. Esta opción también acepta un vector de tipo caracter, por ejemplo `labels=c(‘‘A’’, ‘‘B’’, ‘‘C’’)`.

El título se adiciona con `title()`, pero el tipo de letra se cambia ligeramente. Las anotaciones en los ejes se escriben con `mtext()` (*texto marginal*). El primer argumento de esta función es un vector tipo caracter que proporciona el texto a ser escrito. La opción `line` indica la distancia al área de graficado (por defecto `line=0`), y `at` la coordenada. La segunda llamada a `mtext()` utiliza el valor por defecto de `side` (3). Las otras dos llamadas a `mtext()` pasan un vector numérico como el primer argumento: esto será convertido a un caracter.

4.6. Los paquetes grid y lattice

Los paquetes grid y lattice son la implementación de las gráficas Trellis de S-PLUS en R. Trellis es un método para visualizar datos multivariados y es particularmente apropiado para la exploración de relaciones e interacciones entre variables[14].

La idea principal detrás de lattice (y Trellis) es gráficas condicionales múltiples: una gráfica bivariada se divide en varias gráficas con respecto a los valores de una tercera variable. La función `coplot` usa una aproximación similar, pero grid ofrece mucho más flexibilidad y funcionalidad que `coplot`.

[14] http://cm.bell-labs.com/cm/ms/departments/sia/project/trellis/index.html

Las gráficas producidas por grid o lattice no se pueden combinar o mezclar con aquellas producidas por las funciones vistas anteriormente, porque estos paquetes usan un novedoso modo gráfico[15]. Este nuevo modo tiene su propio sistema de parámetros gráficos muy distinto a lo que hemos visto hasta ahora. Sin embargo, es posible usar ambos modos gráficos en la misma sesión y en el mismo dispositivo gráfico.

Desde un punto de vista práctico, grid contiene todas las funciones necesarias para el modo gráfico, mientras que lattice contiene las funciones más comúnmente usadas.

La mayor parte de las funciones en lattice toman una fórmula como argumento principal; por ejemplo, `y ~x`[16]. La fórmula `y ~x | z` significa que la gráfica de `y` con respecto a `x` será dibujada como diferentes gráficas con respecto a los valores de `z`.

La siguiente tabla resume las funciones principales en lattice. La fórmula proporcionada como argumento es la típicamente utilizada, pero todas estas funciones aceptan fórmulas condicionales (`y ~x | z`) como argumento principal; como se verá en los ejemplos más abajo, en este último caso se crea una gráfica múltiple con respecto a los valores de `z`.

`barchart(y ~x)`	histograma de los valores de `y` con respecto a los de `x`
`bwplot(y ~x)`	gráfico tipo "box-and-whiskers"
`densityplot(~x)`	gráfico de funciones de densidad
`dotplot(y ~x)`	gráfico de puntos tipo Cleveland (gráficos apilados línea por línea y columna por columna)
`histogram(~x)`	histograma de las frecuencias de `x`
`qqmath(~x)`	cuartiles de `x` de acuerdo a los esperado con una distribuión teórica
`stripplot(y ~x)`	gráfico uni-dimensional, `x` debe ser numérico, `y` puede ser un factor
`qq(y ~x)`	cuartiles para comparar dos distribuciones, `x` debe ser numérico, `y` puede ser numérico, caracter, o factor pero debe tener por lo menos dos niveles
`xyplot(y ~x)`	gráficos bivariados (con muchas funcionalidades)
`levelplot(z ~x*y)`	gráfico coloreado de los valores de `z` en las coordenadas dadas por `x` y `y` (`x`, `y` y `z` deben ser de la misma longitud)
`splom(~x)`	matriz de gráficos bivariados
`parallel(~x)`	gráfico de coordenadas paralelas

Algunas funciones en lattice tienen el mismo nombre que algunas de las funciones gráficas en el paquete base. Estas últimas se "esconden" cuando lattice es cargado en memoria.

Veamos algunos ejemplos para ilustrar algunos aspectos de lattice. Primero, el paquete debe ser cargado en memoria con el comando `library(lattice)` para poder acceder sus funciones.

Empecemos con los gráficos de funciones de densidad. Estos gráficos se pueden hacer simplemente con el comando `densityplot(~x)` resultando en una curva con la función de densidad empírica donde los puntos corresponden a las observaciones en el eje *x* (similar a `rug()`). Nuestro ejemplo será ligeramente más complicado con la superposición en cada gráfico de las curvas de densidad empíricas con las curvas esperadas bajo una distribución normal. Es necesario usar el argumento `panel` que define lo que se dibujará en cada gráfica. Los comandos son:

```
n <- seq(5, 45, 5)
x <- rnorm(sum(n))
y <- factor(rep(n, n), labels=paste("n =", n))
densityplot(~ x | y,
            panel = function(x, ...) {
```

[15] Este modo gráfico remedia algunas de las debilidades del paquete base tal como falta de interactividad con las gráficas.

[16] `plot()` tambien acepta fórmulas como argumento principal: si `x` y `y` son dos vectores de la misma longitud, `plot(y ~x)` y `plot(x, y)` resultan en gráficas idénticas

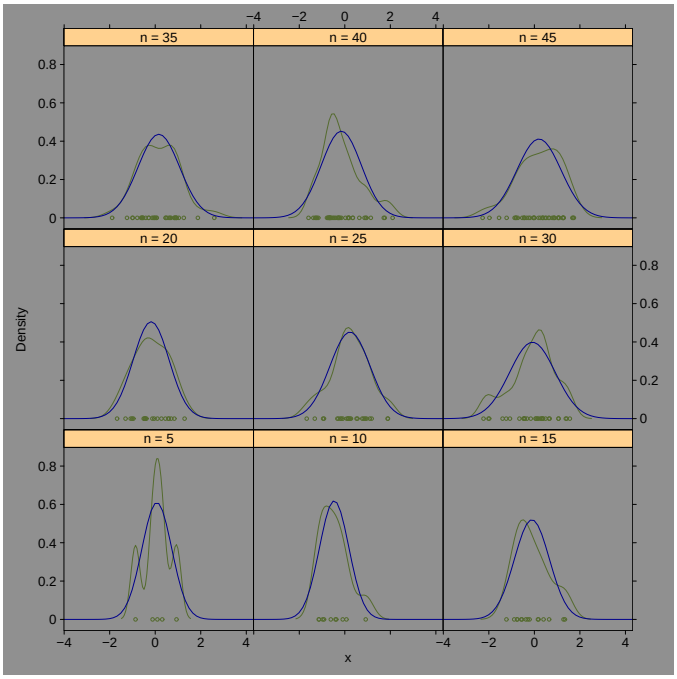


Figura 7: La función `densityplot`.

```
        panel.densityplot(x, col="DarkOliveGreen", ...)
        panel.mathdensity(dmath=dnorm,
                          args=list(mean=mean(x), sd=sd(x)),
                          col="darkblue")
})
```

Las primeras tres líneas generan una muestra aleatoria tomada de una distribución normal, la cual es submuestras de tamaño 5, 10, 15, ..., y 45. Después viene la llamada a la función `densityplot()` lo que produce una gráfica para cada sub-muestra. `panel` toma una fucnión como argumento. En este ejemplo, hemos definido una función que llama a otras dos funciones predefinidas en lattice: `panel.densityplot` que dibuja la función de densidad empírica, y `panel.mathdensity` que dibuja la función de densidad teórica asumiendo una distribución normal. La función `panel.densityplot` es llamada por defecto si no se especifica ningún argumento en `panel`: el comando `densityplot(~x | y)` hubiera dado como resultado la misma Fig. 7 pero sin las curvas azules.

Los siguientes ejemplos utilizan algunas de las conjuntos de datos disponibles en R: la localización de 1000 eventos sísmicos cerca de las islas Fiji, y algunas medidas florales tomadas para tres especies de iris.

Fig. 8 muestra la localización geográfica de los eventos sísmicos con respecto a su profundidad. Los comandos necesarios para crear esta gráfica son:

```
data(quakes)
mini <- min(quakes$depth)
maxi <- max(quakes$depth)
int <- ceiling((maxi - mini)/9)
inf <- seq(mini, maxi, int)
quakes$depth.cat <- factor(floor(((quakes$depth - mini) / int)),
                    labels=paste(inf, inf + int, sep="-"))
```

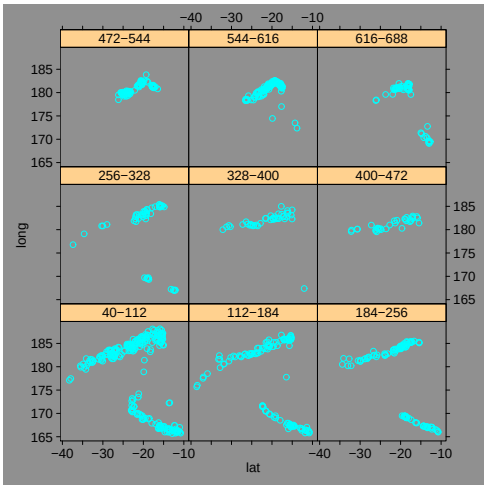


Figura 8: La función `xyplot` con los datos en “quakes”.

```
xyplot(lat ~ long | depth.cat, data = quakes)
```

El primer comando carga los datos de `quakes` en memoria. Los siguientes 5 comandos crean un factor dividiendo la profundidad (variable `depth`) en 9 intervalos equivalentes (con el mismo rango): los niveles del factor son enumerados con el límite superior e inferior de estos intervalos. Una vez definidas las variables, se llama a la función `xyplot` con su fórmula apropiada y un argumento `data` indicando donde se deben buscar las variables[17].

Con los datos en `iris`, la sobreposición entre diferentes especies es lo suficientemente baja y permite diferenciar claramente entre ellas en la figura (Fig. 9). Los comandos son:

```
data(iris)
xyplot(
  Petal.Length ~ Petal.Width, data = iris, groups=Species,
  panel = panel.superpose,
  type = c("p", "smooth"), span=.75,
  key = list(x=0.15, y=0.85,
    points=list(col=trellis.par.get()[["superpose.symbol"]]$col[1:3],
                pch = 1),
    text = list(levels(iris$Species)))
)
```

La llamada a la función `xyplot` es un poco más compleja aqui que en los ejemplos anteriores, y utiliza varias opciones que veremos en detalle. La opción `groups`, como su nombre lo sugiere, define grupos que serán usados por otras opciones. Vimos anteriormente la opción `panel` que define como se respresentarán los diferentes grupos en la gráfica: aqui usamos una función predefinida `panel.superpose` para superponer los grupos en la misma gráfica. No pasamos opciones a `panel.superpose`, asi que se usarán los colores por defecto para distinguir los grupos. La opción `type`, como en `plot()`, especifica como representar los datos, pero aqui se pueden especificar varios argumentos en un vector: `"p"` para dibujar puntos y `"smooth"` para dibujar una curva que se ajuste a los datos con un grado de flexibilidad especificado por `span`. La opción `key` agrega una leyenda a la gráfica; la sintaxis es un poco complicada aqui, pero esto

[17] `plot()` no puede tomar argumentos tipo `data`; la localización de las variables se debe especificar explícitamente. Por ejemplo, `plot(quakes$long ~quakes$lat)`.

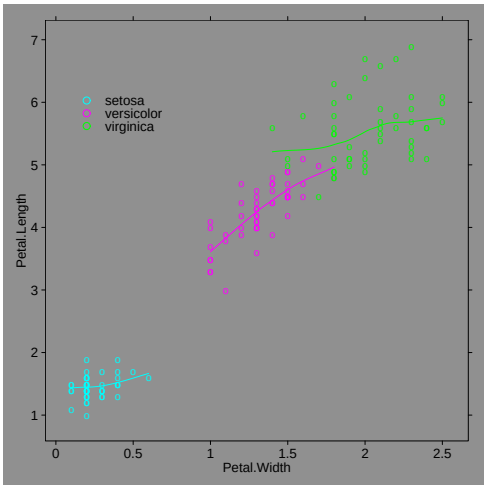


Figura 9: La función `xyplot` con los datos de “iris”.

será simplificado en versiones futuras de lattice a algo similar a la función `legend` utilizada en las gráficas estándar de R. `key` toma una lista como argumento: `x` y `y` indican la localización de la leyenda (si las coordenadas son omitidas la leyenda se coloca fuera de la región de dibujo); `points` especifica el tipo de símbolo, el cual es extraido de las definiciones por defecto (por eso la expresión ligeramente complicada); y `text` proporciona el texto de la leyenda, el cual es por supuesto, el nombre de la especie.

Veremos ahora la función `splom` con los mismos datos en `iris`. Los siguientes comandos se usaron para producir la Fig. 10:

```
splom(
  ~iris[1:4], groups = Species, data = iris, xlab = "",
  panel = panel.superpose,
  key = list(columns = 3,
    points = list(col=trellis.par.get()[["superpose.symbol"]]$col[1:3],
                  pch = 1),
    text = list(c("Setosa", "Versicolor", "Virginica")))
)
```

Esta vez, el argumento principal es una matriz (las primeras cuatro columnas de `iris`). El resultado es un conjunto de todos los posibles gráficos bi-variados entre las columnas de la matriz, como en la función estándar `pairs`. Por defecto, `splom` agrega el texto “Scatter Plot Matrix” bajo el eje *x*: para evitar esto, usamos la opción `xlab=“”`. Las otras opciones son similares a las del ejemplo anterior con excepción de `columns = 3` en `key` que se especifica así, para que la leyenda se presente en tres columnas.

La Fig. 10 se pudo haber hecho con `pairs()`, pero esta última no puede hacer gráficas condicionales como la de la Fig. 11. El código es relativamente simple:

```
splom(~iris[1:3] | Species, data = iris, pscales = 0,
      varnames = c("Sepal\nLength", "Sepal\nWidth", "Petal\nLength"))
```

Ya que las sub-gráficas son relativamente pequeñas, agregamos dos opciones para mejorar la legibilidad de la figura: `pscales = 0` omite los marcadores en los ejes (todas las sub-gráficas se dibujan en la misma escala), y los nombres de las variables se re-definieron para mostrarlos en dos líneas (`"\n"` codifica un caracter de retorno a nueva línea).

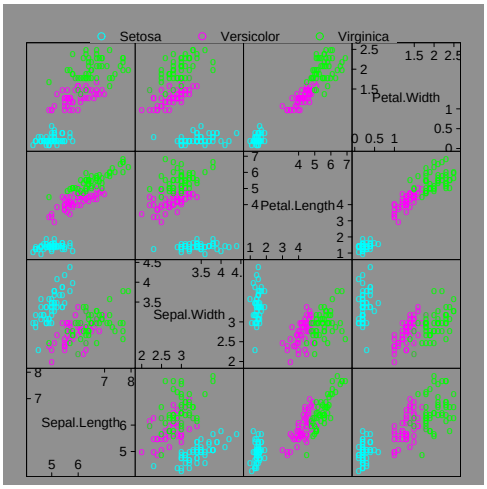


Figura 10: La función `splom` con los datos de “iris” (1).

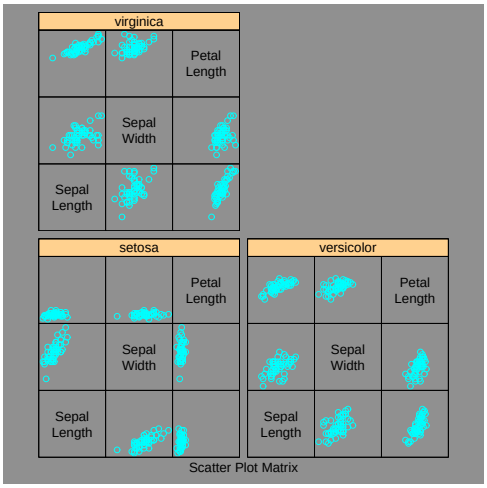


Figura 11: La función `splom` con los datos de “iris” (2).

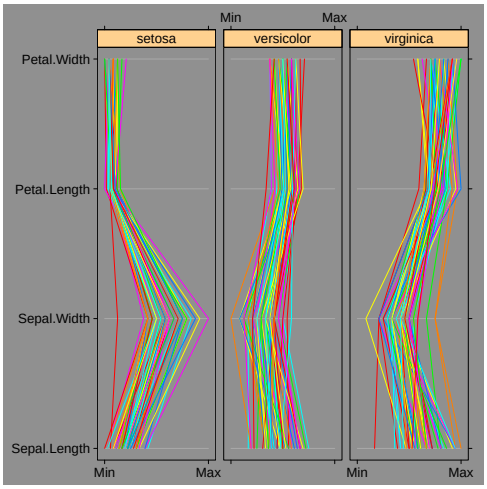


Figura 12: La función `parallel` con los datos de “iris”.

El último ejemplo utiliza el método de coordenadas paralelas para análisis exploratorio de datos multivariados. Las variables se disponen en uno de los ejes (por ejemplo, el eje *y*), y los valores observados se grafican en el otro eje (se estandarizan para hacerlas comparables). Los distintos valores de un individuo se unen por una línea. Con los datos de `iris`, se obtuvo la Fig. 12 usando el siguiente código:

```
parallel(~iris[, 1:4] | Species, data = iris, layout = c(3, 1))
```

5. Análisis estadísticos con R

Tal como con los gráficos, es imposible ahondar en detalles acerca de todas las posibilidades ofrecidas por R para realizar análisis estadísticos. Mi meta en esta sección es proporcionar una visión muy general para que el lector se lleve una idea de las características de R para realizar análisis de todo tipo.

Con expeción de las funciones en los paquetes grid y lattice, todas la funciones que hemos visto hasta ahora están localizadas en el paquete base. Algunas de las funciones para análisis de datos están en base pero la gran mayoría de los métodos estadísticos disponibles en R están distribuidos como paquetes *packages*. Algunos de estos paquetes vienen instalados junto con base, otros están dentro del grupo *recommended* ya que usan métodos comunmente utilizados en estadśtica, y finalmente mucho otros paquetes están dentro del grupo *contributed* y debe ser instalados por el usuario.

Para introducir el método general de realizar análisis de datos en R, empezaremos con un ejemplo simple que requiere solamente el paquete base . Posteriomente explicaremos algunos conceptos como *fórmulas* y *funciones genéricas*, que son útiles independientemente del tipo de análisis realizado. Concluiremos con un mirada rápida a los diferentes paquetes.

5.1. Un ejemplo simple de análisis de varianza

Existen tres funciones estadísticas principales en el paquete base: `lm`, `glm` y `aov` para realizar regresión lineal, modelos lineales generalizados y análisis de varianza, respectivamente. También mencionaremos `loglin` para modelos log-lineales, pero esta función toma una tabla de

contingencia como argumento en vez de una fórmula[18]. Para ver como hacer análisis de varianza tomemos unos datos que vienen incluidos con R: `InsectSprays` (insecticidas). Se probaron en el campo 6 diferentes tipos de insecticidas utilizando el número de insectos como la variable de respuesta. Cada insecticida se probó 12 veces, para un total de 72 observaciones. No haremos aqui una exploración gráfica de los datos, sino que nos enfocaremos en un análisis de varianza simple de la variable de respuesta como función del insecticida usado. Después de cargar los datos en memoria con la función `data`, el análisis se realiza con la función `aov` (después de transformar la respuesta):

```
> data(InsectSprays)
> aov.spray <- aov(sqrt(count) ~ spray, data = InsectSprays)
```

El argumento principal (y obligado) de `aov()` es una fórmula que especifica la respuesta en el lado izquierdo del símbolo `~` y la variable explicativa en el lado derecho. La opción `data = InsectSprays` inidica que las variables deben ser buscadas en el marco de datos `InsectSprays`. Esta sintaxis es equivalente a:

```
> aov.spray <- aov(sqrt(InsectSprays$count) ~ InsectSprays$spray)
```

o también, si conocemos el número de las columnas de las variables:

```
> aov.spray <- aov(sqrt(InsectSprays[, 1]) ~ InsectSprays[, 2])
```

la primera sintaxis es más clara y se presta a menos confusión.

Los resultados del análisis no se muestran ya que son asignados a un objeto llamado `aov.spray`. Usaremos entonces otras funciones para extraer los resultados, como por ejemplo `print()` para ver un resumen breve del análisis (más que todo los parámetros) y `summary()` para ver más detalles (incluyendo las pruebas estadísticas):

```
> aov.spray
Call:
   aov(formula = sqrt(count) ~ spray, data = InsectSprays)

Terms:
                   spray Residuals
Sum of Squares  88.43787  26.05798
Deg. of Freedom        5        66

Residual standard error: 0.6283453
Estimated effects may be unbalanced
> summary(aov.spray)
            Df Sum Sq Mean Sq F value    Pr(>F)
spray        5 88.438  17.688  44.799 < 2.2e-16 ***
Residuals   66 26.058   0.395
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

Recordemos que escribir el nombre del objeto como un comando, es similar al comando `print(aov.spray)`. Para ver una representación gráfica de los resultados podemos usar `plot()` o `termplot()`. Antes de escribir `plot(aov.spray)` dividiremos la ventana gráfica en cuatro partes de tal manera que las cuatro gráficas diagnósticas se dibujan en la misma ventana. Los comandos son:

[18] El paquete MASS tiene la función `loglm` que permite una interface de fórmula a `loglin`.

```
> opar <- par()
> par(mfcol = c(2, 2))
> plot(aov.spray)
> par(opar)
> termplot(aov.spray, se=TRUE, partial.resid=TRUE, rug=TRUE)
```

y los resultados se muestran en la Fig. 13 y la Fig. 14.

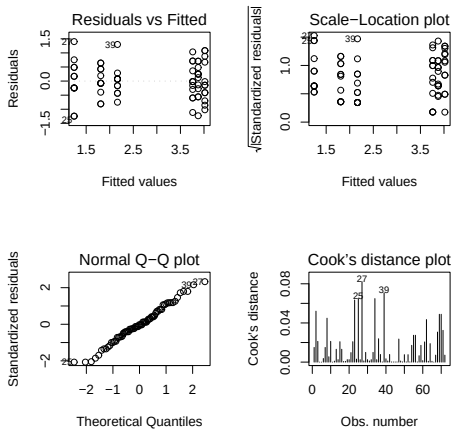


Figura 13: Representación gráfica de los resultados de `aov` usando `plot()`.

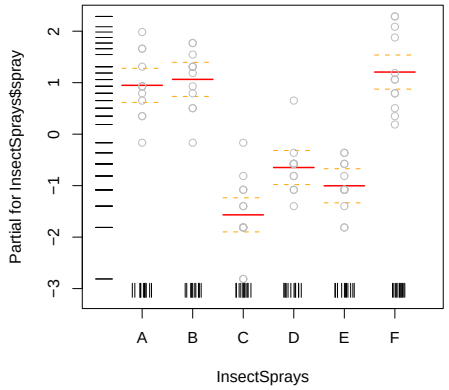


Figura 14: Representación gráfica de los resultados de la función `aov` usando `termplot()`.

5.2. Fórmulas

El uso de fórmulas es un elemento clave en análisis estadístico con R: la nomenclatura uilizada es la misma para (casi) todas las funciones. Una fórmula se escribe típicamente como `y ~modelo` donde `y` es la variable dependiente (o de respuesta) y `modelo` es un conjunto de términos para los cuales es necesario estimar una serie de parámetros. Estos términos se representan con símbolos aritméticos pero tienen un significado muy particular en R.

`a+b`	efectos de `a` y `b`
`X`	si `X` es una matriz, especifica un efecto aditivo para cada una de las columnas; por ejemplo `X[,1]+X[,2]+...+X[,ncol(X)]`; algunas de las columnas se pueden seleccionar con índices numéricos (por ej., `X[,2:4]`)
`a:b`	efecto interactivo entre `a` y `b`
`a*b`	efectos aditivos e interactivos entre `a` y `b` (idéntico a `a+b+a:b`)
`poly(a, n)`	polinomios de `a` hasta grado *n*
`^n`	incluye todas las itnteracciones hasta el nivel *n*, por ej., `(a+b+c)^2` es idéntico a `a+b+c+a:b+a:c+b:c`
`b%in% a`	los efectos de `b` están anidados en `a` (idéntico a `a+a:b` o `a/b`)
`a-b`	remueve el efecto de `b`, por ejemplo: `(a+b+c)^2-a:b` es idéntico a `a+b+c+a:c+b:c`
`-1`	`y~x-1` regresión a través del origen (igual para `y~x+0` o `0+y~x`)
`1`	`y~1` ajusta un modelo sin efectos (solo el intercepto)
`offset(...)`	agrega un efecto al modelo sin estimar los parámetros (e.g., `offset(3*x)`)

Vemos que los operadores aritméticos en las fórmulas tienen significados muy diferentes a los que tendrían en expresiones regulares. Por ejemplo la fórmula `y~x1+x2` define el modelo $y = \beta_1 x_1 + \beta_2 x_2 + \alpha$, en vez de (si el operador `+` tuviera su significado usual) $y = \beta(x_1 + x_2) + \alpha$. Para incluir operaciones aritméticas en una fórmula se puede usar la función `I()`: la fórmula `y~I(x1+x2)` define el modelo $y = \beta(x_1 + x_2) + \alpha$. De manera similar, para definir el modelo $y = \beta_1 x + \beta_2 x^2 + \alpha$, usaremos la fórmula `y ~poly(x, 2)` (y no `y ~x + x^2`).

Para análisis de varianza, `aov()` accepta una sintaxis particular para definir efectos aleatorios. Por ejemplo, `y ~a + Error(b)` significa los efectos aditivos de un término fijo (`a`) y uno aleatorio (`b`).

5.3. Funciones genéricas

Recordemos que las funciones de R actúan con respecto a los atributos de los objetos pasados como argumentos. Los objetos resultantes de un análisis poseen un atributo particular denominado *clase* que contiene la "firma" de la función usada para el análisis. Las funciones que usen posteriomente para extraer los resultados del análisis actuarán específicamente con respecto a la clase del objeto. Estas funciones se denominan genéricas.

Por ejemplo, la función más utilizada para extraer resultados de otros análisis es `summary` que muestra los resultados detallados de los mismos. Si el objeto dado como argumento es de clase `"lm"` (modelo lineal) o "`aov`" (análisis de varianza), suena obvio que la información mostrada no será la misma. La ventaja de estas funciones genéricas que su sintaxis es la misma para todos los análisis en que se usen.

El objeto que contiene los resultados de un análisis es generalmente una lista y su visualización está determinada por su clase. Hemos visto este concepto que la acción de una función depende del tipo de objeto proporcionado como argumento. Esta es una característica general de R[19]. La siguiente tabla muestra las principales funciones genéricas que se pueden usar para extraer información de objetos resultantes de un análisis. El uso típico de estas funciones es:

```
> mod <- lm(y ~ x)
> df.residual(mod)
[1] 8
```

[19] Existen más de 100 funciones genéricas en R.

`print`	devuelve un corto resúmen
`summary`	devuelve un resúmen detallado
`df.residual`	devuelve el número de grados de libertad
`coef`	devuelve los coeficientes estimados (algunas veces con sus errores estándar)
`residuals`	devuelve los residuales
`deviance`	devuelve la devianza
`fitted`	devuelve los valores ajustados
`logLik`	calcula el logaritmo de la verosimilitud y el número de parámetros
`AIC`	calcula el criterio de información de Akaike o AIC (depende de `logLik()`)

Funciones como `aov` o `lm` devuelven una lista donde los diferentes elementos corresponden a los resultados del análisis. Si tomamos nuestro ejemplo con el análisis de varianza de los datos en `InsectSprays`, podemos ver la estructura del objeto devuelto por `aov()`:

```
> str(aov.spray, max.level = -1)
List of 13
 - attr(*, "class")= chr [1:2] "aov" "lm"
```

Otra manera de ver esta estructura es visualizando los nombres del objeto:

```
> names(aov.spray)
 [1] "coefficients"  "residuals"     "effects"
 [4] "rank"          "fitted.values" "assign"
 [7] "qr"            "df.residual"   "contrasts"
[10] "xlevels"       "call"          "terms"
[13] "model"
```

Los elementos se pueden extraer como en cualquier otra lista:

```
> aov.spray$coefficients
(Intercept)      sprayB      sprayC      sprayD
  3.7606784   0.1159530  -2.5158217  -1.5963245
     sprayE      sprayF
 -1.9512174   0.2579388
```

`summary()` también crea una lista la cual, en el caso de `aov()`, es simplemente una tabla de pruebas:

```
> str(summary(aov.spray))
List of 1
 $ :Classes anova  and ‘data.frame’:    2 obs. of  5 variables:
  ..$ Df     : num [1:2] 5 66
  ..$ Sum Sq : num [1:2] 88.4 26.1
  ..$ Mean Sq: num [1:2] 17.688  0.395
  ..$ F value: num [1:2] 44.8   NA
  ..$ Pr(>F) : num [1:2] 0 NA
 - attr(*, "class")= chr [1:2] "summary.aov" "listof"
> names(summary(aov.spray))
NULL
```

Las funciones genéricas también se denominan *métodos*. Esquemáticamente, se construyen como *`method.foo`*, donde *`foo`* es la función de análisis. En el caso de `summary`, podemos ver las funciones que usan este método:

```
> apropos("^summary")
 [1] "summary"               "summary.aov"
 [3] "summary.aovlist"       "summary.connection"
 [5] "summary.data.frame"    "summary.default"
 [7] "summary.factor"        "summary.glm"
 [9] "summary.glm.null"      "summary.infl"
[11] "summary.lm"            "summary.lm.null"
[13] "summary.manova"        "summary.matrix"
[15] "summary.mlm"           "summary.packageStatus"
[17] "summary.POSIXct"       "summary.POSIXlt"
[19] "summary.table"
```

Podemos ver la diferencia para este método comparando como actúa en una regresión lineal vs. un análisis de varianza:

```
> x <- y <- rnorm(5);
> mod <- lm(y ~ x)
> names(mod)
 [1] "coefficients"  "residuals"     "effects"
 [4] "rank"          "fitted.values" "assign"
 [7] "qr"            "df.residual"   "xlevels"
[10] "call"          "terms"         "model"
> names(summary(mod))
 [1] "call"          "terms"         "residuals"
 [4] "coefficients"  "sigma"         "df"
 [7] "r.squared"     "adj.r.squared" "fstatistic"
[10] "cov.unscaled"
```

Los objetos devueltos por `aov()`, `lm()`, `summary()`, … son listas, pero no se visualizan como las listas “comunes y corrientes” que hemos visto anteriormente. De hecho, son métodos `print` de estos objetos (recuerde que escribir el nombre de un objeto como un comando equivale a usar `print()`):

```
> apropos("^print")
 [1] "print.pairwise.htest"   "print.power.htest"
 [3] "print"                  "print.anova"
 [5] "print.aov"              "print.aovlist"
 [7] "print.atomic"           "print.by"
 [9] "print.coefmat"          "print.connection"
[11] "print.data.frame"       "print.default"
[13] "print.density"          "print.difftime"
[15] "print.dummy.coef"       "print.dummy.coef.list"
[17] "print.factor"           "print.family"
[19] "print.formula"          "print.ftable"
[21] "print.glm"              "print.glm.null"
[23] "print.hsearch"          "print.htest"
[25] "print.infl"             "print.integrate"
[27] "print.libraryIQR"       "print.listof"
[29] "print.lm"               "print.lm.null"
[31] "print.logLik"           "print.matrix"
[33] "print.mtable"           "print.noquote"
```

```
[35] "print.octmode"          "print.ordered"
[37] "print.packageIQR"       "print.packageStatus"
[39] "print.POSIXct"          "print.POSIXlt"
[41] "print.recordedplot"     "print.rle"
[43] "print.SavedPlots"       "print.simple.list"
[45] "print.socket"           "print.summary.aov"
[47] "print.summary.aovlist"  "print.summary.glm"
[49] "print.summary.glm.null" "print.summary.lm"
[51] "print.summary.lm.null"  "print.summary.manova"
[53] "print.summary.table"    "print.table"
[55] "print.tables.aov"       "print.terms"
[57] "print.ts"               "print.xtabs"
```

Todos estos métodos de `print` permiten cierta visualización dependiendo del análisis.

La siguiente tabla muestra algunas funciones genéricas que realizan análisis suplementarios a un objeto resultante de un análisis, donde el argumento principal es este último, pero en algunos casos se necesitan argumentos adicionales (por ejemplo, para `predict` o `update`).

`add1`	prueba sucesivamente todos los términos que se pueden adicionar a un modelo
`drop1`	prueba sucesivamente todos los términos que se pueden remover de un modelo
`step`	selecciona un modelo con AIC (llama a `add1` y a `drop1`)
`anova`	calcula una tabla de análisis de varianza o devianza de uno o varios modelos
`predict`	calcula los valores predichos para datos nuevos de un modelo ya ajustado
`update`	re-ajusta un modelo con una nueva fórmula o nuevos datos

Existen tambien varias funciones utilitarias que extraen información de un objeto modelo o fórmula, tal como `alias()` que encuentra los términos linealmente dependientes en un modelo lineal especificado por una fórmula.

Finalmente, existen funciones gráficas como `plot` que hacen gráficos diagnósticos, o `termplot` (ver el ejemplo anterior), aunque esta última no es estrictamente genérica, pues llama a `predict()`.

5.4. Paquetes

La siguiente tabla muestra los paquetes que se distribuyen con base. Con excepción de ctest que se carga en memoria cuando R comienza, cada paquete puede ser utilizado después de haber sido cargado:

```
> library(eda)
```

La lista de las funciones disponibles en un paquete se puede ver escribiendo:

```
> library(help=eda)
```

o navegando la ayuda en formato html. La información relativa a cada función se puede acceder como lo vimos anteriormente (p. 7).

Paquete	Descripción
ctest	pruebas clásicas (Fisher, 'Student', Wilcoxon, Pearson, Bartlett, Kolmogorov-Smirnov, ...)
eda	métodos descritos en "Exploratory Data Analysis" por Tukey (solo ajuste lineal robusto y ajuste de medianas)
lqs	regresión resistente y estimación de covarianza
methods	definición de métodos y clases para objetos en R y herramientas de programación
modreg	regresión moderna (alisamiento y regresión local)
mva	análisis multivariado
nls	regresión no-lineal
splines	representaciones polinómicas
stepfun	funciones de distribución empíricas
tcltk	funciones para hacer interfase desde R a elementos de interfase gráfica Tcl/Tk
tools	herramientas para desarrollo y administración de paquetes
ts	análisis de series temporales

Muchos de los paquetes en la sección *contribuciones* complementan la lista de métodos estadísticos disponibles en R. Estos paquetes son distribuidos separademente, y deben ser instalados y cargados en R para su funcionamiento. Para ver una lista completa y descripción de los paquetes contribuidos visite la página web del CRAN[20]. Muchos de estos paquetes son *recomendados* ya que cubren métodos estadísticos usualmente usados en análisis de datos. (En Windows, los paquetes recomendados se distribuyen con la instalación base de R en el archivo SetupR.exe.) Los paquetes recomendados se describen brevemente en la siguiente tabla.

Paquete	Descripción
boot	métodos de remuestreo y "bootstraping"
class	métodos de clasificación
cluster	métodos de agregación
foreign	funciones para leer datos en diferentes formatos (S3, Stata, SAS, Minitab, SPSS, Epi Info)
KernSmooth	métodos para suavización nuclear y estimación de densidad (incluyendo núcleos bivariados)
MASS	contiene muchas funciones, herramientas y datos de las librerías de "Modern Applied Statistics with S-PLUS" por Venables & Ripley
mgcv	modelos aditivos generalizados
nlme	modelos lineales y non-linear con efectos mixos
nnet	redes neuronales y modelos multinomiales log-lineales
rpart	particionamiento recursivo
spatial	análisis espaciales ("kriging", covarianza espacial, ...)
survival	análisis de sobrevivencia

El procedimiento para instalar un paquete depende del sistema operativo y de la manera como se instaló R: desde el código fuente o archivos binarios pre-compilados. Si es esta útlima, es recomendado usar los paquetes pre-compilados disponibles en el sitio CRAN. En Windows, el archivo

[20] http://cran.r-project.org/src/contrib/PACKAGES.html

binario Rgui.exe tiene el menú “Packages” que permite la instalación de paquetes directamente desde el disco duro o a través de internet desde la página web CRAN.

Si R fue compilado localmente, se puede instalar un paquete directamente desde el código fuente el cual es distribuido normalmente como un archivo ‘.tar.gz’. Por ejemplo, si queremos instalar el paquete `gee`, primero es necesario bajar desde el internet el archivo gee_4.13-6.tar.gz (el número 4.13-6 indica la versión del paquete; generalmente solo existe una versión disponible en CRAN). Después es necesario escribir lo siguiente desde el sistema (no desde R):

```
R INSTALL gee_4.13-6.tar.gz
```

Existen varias funciones para manejar paquetes tales como `installed.packages()`, `CRAN.packages()` o `download.packages()`. También es bastante útil escribir el siguiente comando con cierta regularidad:

```
> update.packages()
```

esto chequea las versiones de los paquete instalados en el sistema y los compara con los disponibles en CRAN (este comando se puede llamar desde el menú “Packages” en Windows). De esta manera, el usuario puede actualizar sus paquetes con las versiones más recientes.

6. Programación práctica con R

Ya que hemos echado un vistazo general a la funcionalidad de R, volvamos por un momento al su uso como lenguaje de programación. Veremos ideas muy simples que son fáciles de implementar en la práctica.

6.1. Bucles y Vectorización

Una ventaja de R comparado con otros programas estadísticos con “menus y botones” es la posisbilidad de programar de una manera muy sencilla una serie de análisis que se puedan ejecutar de manera sucesiva. Esto es común a cualquier otro lenguaje de programación, pero R posee características muy particulares que hacen posible programar sin mcuhos conocimientos o experiencia previa en esta área.

Como en otros lenguajes, R posee *estructuras de control* que no son muy diferentes a las de un lenguaje de alto nivel como C. Supongamos que tenemos un vector `x`, y para cada elemento de `x` con valor igual a `b`, queremos asignar el valor 0 a otra variable `y`, o sino asignarle 1. Primero creamos un vector `y` de la misma longitud de `x`:

```
y <- numeric(length(x))
for (i in 1:length(x)) if (x[i] == b) y[i] <- 0 else y[i] <- 1
```

Se pueden usar corchetes para ejecutar varias instrucciones:

```
for (i in 1:length(x)) {
    y[i] <- 0
    ...
}

if (x[i] == b) {
    y[i] <- 0
    ...
}
```

Otra posibilidad es ejecutar una instrucción siempre y cuando se cumpla una cierta condición:

```
while (myfun > minimum) {
    ...
}
```

Sin embargo, este tipo de bucles y estructuras se pueden evitar gracias a una característica clave en R: *vectorización*. La vectorización hace los bucles implícitos en las expresiones y ya lo hemos visto en muchos casos. Por ejemplo, consideremos la suma de dos vectores:

```
> z <- x + y
```

Esta suma se hubiera podido escribir como un bucle, como se hace en muchos otros lenguajes:

```
> z <- numeric(length(x))
> for (i in 1:length(z)) z[i] <- x[i] + y[i]
```

En este caso, es necesario crear con anterioridad el vector `z` por la necesidad de indexar los elementos. Es fácil ver que este bucle explícito solo funciona si `x` y `y` son de la misma longitud: es necesario alterar el programa si esto es falso, mientras que la primera situación funcionará en todos los casos.

Las ejecuciones condicionales (`if ... else`) se pueden evitar con el uso de indexación lógica; volviendo al ejemplo anterior:

```
> y[x == b] <- 0
> y[x != b] <- 1
```

También existen varias funciones del tipo "`apply`" que evitan el uso de bucles. `apply()` actúa sobre las filas o columnas de una matriz, y su sintaxis es `apply(X, MARGIN, FUN, ...)`, donde `X` es una matriz, `MARGIN` indica si se van a usar las filas (`1`), las columnas (`2`), or ambas (`c(1, 2)`), `FUN` es una función (o un operador, pero en este caso debe especificarse en corchetes) a ser aplicada, y `...` son posibles argumentos opcionales de `FUN`. Veamos un ejemplo simple.

```
> x <- rnorm(10, -5, 0.1)
> y <- rnorm(10, 5, 2)
> X <- cbind(x, y)     # las columnas de X mantienen los nombres "x" y "y"
> apply(X, 2, mean)
        x         y
-4.975132  4.932979
> apply(X, 2, sd)
        x         y
0.0755153 2.1388071
```

La función `lapply()` actúa sobre una lista: su sintaxis es similar a la de `apply` y devuelve una lista.

```
> forms <- list(y ~ x, y ~ poly(x, 2))
> lapply(forms, lm)
[[1]]

Call:
```

```
FUN(formula = X[[1]])

Coefficients:
(Intercept)            x
     31.683        5.377


[[2]]

Call:
FUN(formula = X[[2]])

Coefficients:
(Intercept)  poly(x, 2)1  poly(x, 2)2
     4.9330       1.2181      -0.6037
```

La función `sapply()` es una variación más flexible de `lapply()` que puede tomar un vector o una matriz como argumento principal, y devuelve los resultados en una manera más amigable, generalmente en una tabla.

6.2. Escribiendo un programa en R

Típicamente, los programas en R se escriben en un archivo que se guarda en formato ASCII con terminación'.R'. Un programa se usa típicamente cuando uno quiere hacer una o varias operaciones muchas veces. En nuestro primer ejemplo, queremos hacer la misma gráfica para tres especies diferentes de aves y los datos están en tres archivos diferentes. Procederemos paso a paso y veremos diferentes maneras de programar este problema.

Primero, hagamos nuestro programa de la manera más intuitiva posible ejecutando sucesivamente los comandos necesarios, teniendo cuidado de particionar el dispositivo gráfico con anterioridad.

```
layout(matrix(1:3, 3, 1))              # particiona la ventana grafica
data <- read.table("Swal.dat")         # lee los datos
plot(data$V1, data$V2, type="l")
title("swallow")                       # agrega un titulo
data <- read.table("Wren.dat")
plot(data$V1, data$V2, type="l")
title("wren")
data <- read.table("Dunn.dat")
plot(data$V1, data$V2, type="l")
title("dunnock")
```

El carcater '#' se usa para agregar comentarios al programa y R los ignora durante la ejecución.

El problema del primer programa es que se puede volver bastante largo si queremos agregar otras especies. Más aún, algunos comandos se ejecutan varias veces, y por lo tanto, se pueden agrupar y ejecutar juntos cambiando algunos argumentos. La estrategia que usamos aqui es poner estos argumentos en vectores de tipo caracter, y después usar indexación para acceder a los diferentes valores.

```
layout(matrix(1:3, 3, 1))                # particionar la ventana gr'afica
species <- c("swallow", "wren", "dunnock")
```

```
file <- c("Swal.dat" , "Wren.dat", "Dunn.dat")
for(i in 1:length(species)) {
    data <- read.table(file[i])          # leer los datos
    plot(data$V1, data$V2, type="l")
    title(species[i])                    # agregar un titulo
}
```

Note que el argumento `file[i]` no se pone entre comillas en `read.table()` ya que este argumento ya es de tipo caracter.

Ahora nuestro programa es mucho más compacto. Es más fácil agregar otras especies ya que los nombres de las mismas están en vectores al principio del programa.

Los programas anteriores funcionarán correctamente siempre y cuando los archivos de datos '.dat' estén localizados en el directorio de trabajo de R; de lo contrario el usuario debe cambiar el directorio de trabajo o especificar la dirección completa en el programa (por ejemplo: `file <-"C:/data/Swal.dat"`). Si el programa está en el archivo Mybirds.R, es necesario primero cargarlo en memoria:

```
> source("Mybirds.R")
```

Como en el ejemplo anterior esto solo funciona si el archivo `Mybirds.R` se encuentra en el directorio de trabajo de R; de lo contrario es necesario especificar la dirección completa.

6.3. Creando sus propias funciones

Hemos visto que la mayor parte del trabajo en R se realiza a través de funciones con sus respectivos argumentos entre paréntesis. R permite al usuario escribir sus propias funciones, y estas tendrán las mismas propiedades de otras funciones.

Escribir sus propias funciones permite un uso flexible, eficiente y racional de R. Volvamos a nuestro ejemplo donde leemos unos datos y dibujamos una gráfica de los mismos. Si deseamos hacer esta operación en diferentes situaciones, puede ser una buena idea escribir una función:

```
mifun <- function(S, F)
{
    data <- read.table(F)
    plot(data$V1, data$V2, type="l")
    title(S)
}
```

Para que esta función pueda ser ejecutada, primero es necesario cargarla en memoria, y esto se puede hacer de varias maneras. Las líneas de la función se pueden escribir directamente desde el teclado, como cualquier otro comando, o ser copiada y pegada a un editor de texto. Si la función está guardada en un archivo ASCII, se puede cargar con `source()` como cualquier otro programa. Si el usuario desea que su función sea cargada cada vez que comienza R, se puede guardar en un archivo especial llamado "espacio de trabajo" (del inglés 'workspace') `.RData` que será cargado en memoria automáticamente si se encuentra en el directorio de trabajo de R. Otra posibilidad es configurar el archivo '.Rprofile' o 'Rprofile' (ver `?Startup` para más detalles). Finalmente, es posible crear un paquete, pero no discutiremos esta alternativa aqui (vea el manual "Writing R Extensions").

Una vez la función es cargada se puede ejecutar con un solo comando como por ejemplo, `mifun("swallow", "Swal.dat")`. Por lo tanto, tenemos ahora una tercera versión de nuestro programa:

```
layout(matrix(1:3, 3, 1))
mifun("swallow", "Swal.dat")
mifun("wren", "Wrenn.dat")
mifun("dunnock", "Dunn.dat")
```

También podemos usar `sapply()` creando una cuarta versión del programa:

```
layout(matrix(1:3, 3, 1))
species <- c("swallow", "wren", "dunnock")
file <- c("Swal.dat" , "Wren.dat", "Dunn.dat")
sapply(species, mifun, file)
```

En R, no es necesario declarar las variables usadas dentro de la función (a diferencia de otros lenguajes como C o Fortran). Cuando una función es ejecutada, R utiliza una regla llamada *ámbito lexicográfico* para decidir si un objeto es local a una función o global. Para entender mejor este mecanismos, consideremos la siguiente función:

```
> foo <- function() print(x)
> x <- 1
> foo()
[1] 1
```

El nombre `x` no está definido dentro de `foo()`, asi que R buscará `x` dentro del ámbito *circundante*, e imprimirá su valor (de lo contrario, se genera un mensaje de error y la ejecución se cancela).

Si `x` es utilizado como el nombre de un objeto dentro de la función, el valor de `x` en el ambiente global (externo a la función) no cambia.

```
> x <- 1
> foo2 <- function() { x <- 2; print(x) }
> foo2()
[1] 2
> x
[1] 1
```

Esta vez `print()` usa el objeto `x` definido dentro de su ambiente, es decir el ambiente de `foo2`.

La palabra "*circundante*" utilizada arriba es importante. En nuestras dos funciones ejemplo existen *dos* ambientes: uno global y el otro local a cada una de las funciones `foo` o `foo2`. Si existen tres o más ambientes anidados, la búsqueda de objetos se hace progresivamente desde un ambiente dado al ambiente circundante a este, y asi sucesivamente hasta llegar el ambiente global.

Existen dos maneras de especificar argumentos a una función: por sus posiciones o por sus nombres (también llamados *argumentos marcados*). Por ejemplo, consideremos una función con tres argumentos:

```
foo <- function(arg1, arg2, arg3) {...}
```

`foo()` se puede ejecutar sin isar los nombres `arg1`, ..., si los objetos correspondientes están colocados en la posición correcta; por ejemplo: `foo(x, y, z)`. Sin embargo, la posición no tiene ninguna importancia si se utilizan los nombres de los argumentos, por ejemplo, `foo(arg3 = z, arg2 = y, arg1 = x)`. Otra rasgo importante de las funciones en R es la posibilidad de usar valores por defecto en la definición. Por ejemplo:

```
foo <- function(arg1, arg2 = 5, arg3 = FALSE) {...}
```

Ambos comandos `foo(x)` y `foo(x, 5, FALSE)` producirán exactamente el mismo resultado. El uso de valores por defecto en la definición de una función es bastante útil y resulta es una mayor flexibilidad.

Otro ejemplo de la flexibilidad se ilustra con la siguiente función que simula el comportamiento de una población bajo el modelo de Ricker:

$$N_{t+1} = N_t \exp\left[r\left(1 - \frac{N_t}{K}\right)\right]$$

Este modelo es usado ampliamente en ecología de poblaciones, particularmente en estudios demográficos de peces. El objetivo es simular el modelo con respecto a la tasa de crecimiento r y el número inicial de individuos en la población N_0 (la capacidad de carga K es usualmente fijada en 1, y usaremos este como su valor por defecto); los resultados se mostrarán como una gráfica del número de individuos en función del tiempo. Agregaremos una opción para permitirle al usuario ver los resultados en los últimos pasos de la simulación (por defecto todos los resultados son graficados). La función abajo realiza este análisis numérico del modelo de Ricker.

```
ricker <- function(nzero, r, K=1, tiempo=100, desde=0, hasta=tiempo)
{
    N <- numeric(tiempo+1)
    N[1] <- nzero
    for (i in 1:tiempo) N[i+1] <- N[i]*exp(r*(1 - N[i]/K))
    Tiempo <- 0:tiempo
    plot(Tiempo, N, type="l", xlim=c(desde, hasta))
}
```

Utilizemos la función para explorar las propiedades del modelo:

```
> layout(matrix(1:3, 3, 1))
> ricker(0.1, 1); title("r = 1")
> ricker(0.1, 2); title("r = 2")
> ricker(0.1, 3); title("r = 3")
```

7. Literatura adicional sobre R

Manuales. R trae varios manuales que se instalan por defecto en R_HOME/doc/manual/ (donde R_HOME donde R está instalado). Todos estos manuales están en inglés:

- "An Introduction to R" [R-intro.pdf],
- "R Installation and Administration" [R-admin.pdf],
- "R Data Import/Export" [R-data.pdf],
- "Writing R Extensions" [R-exts.pdf],
- "R Language Definition" [R-lang.pdf].

Los archivos pueden estar en diferentes formatos (pdf, html, texi, ...) dependiendo del tipo de instalación.

FAQ. R también viene con su propio FAQ (*Preguntas más frecuentes*) localizadas en el directorio R_HOME/doc/html/. La versión de este R-FAQ es actualizada regularmente en el sitio CRAN: http://cran.r-project.org/doc/FAQ/R-FAQ.html.

Recursos en línea El sitio CRAN y la página web de R contienen varios documentos, recursos bibliográficos y enlaces a otros sitios. También se puede encontrar aqui una lista de publicaciones (libros y artículos) sobre R y métodos estadísticos en general[21], y algunos documentos y tutoriales escritos por usuarios de R[22].

Listas de correo. Existen tres listas de discusión en R; para suscribirse, mande un mensaje o lea los archivos en http://www.R-project.org/mail.html.

La lista de discusión general 'r-help' es una fuente interesante de información para usuarios de R (las otras dos listas están dedicadas a anuncios de nuevas versiones, nuevos paquetes, ..., y programadores). Muchos usuarios han enviado funciones y programas a 'r-help' los cuales pueden encontrarse en los archivos de la lista. Si se encuentra algún problema con R, es importante proceder en el siguiente orden antes de enviar un mensaje a 'r-help':

1. lea cuidadosamente la ayuda en línea (puede ser usando la máquina de búsqueda),
2. lea las preguntas más frecuentes (R-FAQ),
3. busque en los archivos de 'r-help' en la dirección proporcionada anteriormente o usando una de las máquinas de búsqueda disponibles en algunas páginas web[23].

R News. La revista electrónica *R News* tiene como objetivo llenar un vacío entre las listas de discusión electrónicas y publicaciones científicas tradicionales. El primer número fué publicado en enero 2001, y se producen tres números por año. Kurt Hornik y Friedrich Leisch son los editores[24].

Citando R en una publicación. Finalmente, si usted menciona a R en una publicación, debe citar el artículo original:

> Ihaka R. & Gentleman R. 1996. R: a language for data analysis and graphics. *Journal of Computational and Graphical Statistics* 5: 299–314.

[21] http://www.R-project.org/doc/bib/R-publications.html
[22] http://cran.r-project.org/other-docs.html. Aquí se pueden encontrar dos manuales más de R escritos en español.
[23] Las direcciones de estos sitios se encuentran en http://cran.r-project.org/search.html
[24] http://cran.r-project.org/doc/Rnews/