



Índice general

1. R Un entorno de análisis y programación estadísticos	1
1.1. Introducción	1
1.2. Instalación	1
1.3. Primer contacto	2
1.4. Conceptos iniciales	2
1.4.1. Teclas de edición	4
1.4.2. Redireccionamiento	4
1.4.3. Edición de un objeto	5
2. Algunas clases de objetos comunes	7
2.1. Vector	7
2.1.1. Operaciones con vectores	9
2.2. Factor	11
2.3. Matriz	12
2.3.1. Operaciones con matrices	14
2.4. Variable multiindexada	17
2.5. Lista	18
2.6. Hoja de datos	20
2.7. Comprobación y cambio de tipos de objetos	22
3. Funciones	24
3.1. Definición de una función	25
3.1.1. Metodología de definición de una función	27
3.2. Algunas funciones elementales	27
3.3. Funciones .First y .Last	29
3.4. Algunos elementos útiles en programación	29
3.4.1. Operadores de relación	30
3.4.2. Estructuras condicionales	30
3.4.3. Estructuras de repetición definida e indefinida	32
3.4.4. invisible	33
3.4.5. Recursividad	34
3.4.6. Depuración de errores	34
3.5. Ejemplos de funciones	35
3.5.1. Factorial de un número	36
3.5.2. Progresión aritmética	37
3.5.3. Progresión geométrica	39
3.5.4. Sucesión de Fibonacci	39
3.5.5. Simulación de una extracción de cartas de una baraja	41

3.5.6.	Función Suma de potencias	47
3.5.7.	Función en una cuadrícula	48
3.5.8.	Procesos no lineales	49
3.6.	Función <code>missing</code>	53
3.7.	Función <code>menu</code>	53
3.8.	Lista de búsqueda y entornos	54
3.9.	Entradas y salidas mediante archivos	56
4.	Uso de archivos externos	58
4.1.	Función <code>scan</code>	58
4.1.1.	Función <code>readLines</code>	61
4.1.2.	Función <code>writeLines</code>	61
4.1.3.	Función <code>unlink</code>	62
4.1.4.	Gestión de archivos	62
4.1.5.	Conexiones	62
4.1.6.	Personalización de archivos de un proyecto	62
4.2.	Lectura de hojas de datos	63
4.3.	Gestión de objetos	66
4.3.1.	<code>dump</code>	66
4.3.2.	<code>write</code>	67
4.3.3.	<code>dput</code> y <code>dget</code>	67
4.3.4.	<code>save</code> y <code>load</code>	67
4.4.	Creación de archivos temporales	69
4.5.	Función <code>library</code>	71
5.	Gráficos	74
5.1.	Dispositivos gráficos	74
5.1.1.	Función <code>x11</code>	74
5.1.2.	Función <code>win.print</code>	74
5.1.3.	Función <code>pictex</code>	75
5.1.4.	Funciones <code>bmp</code> , <code>jpeg</code> y <code>png</code>	75
5.1.5.	Función <code>win.metafile</code>	76
5.1.6.	Función <code>postscript</code>	76
5.2.	Función <code>graphics.off</code>	79
5.3.	Función <code>plot</code>	79
5.4.	Función <code>text</code>	79
5.5.	Función <code>symbols</code>	80
5.6.	Función <code>fourfoldplot</code>	81
5.7.	Función <code>hist</code>	82
5.8.	Función <code>polygon</code>	82
5.9.	Función <code>curve</code>	83
5.10.	Funciones <code>lines</code> y <code>points</code>	84
5.11.	Función <code>par</code>	86
5.12.	Desplazamiento entre dispositivos gráficos	89
5.13.	Impresión de gráficos	90
5.14.	Identificación interactiva	90
5.15.	Función <code>locator</code>	90
5.16.	Función <code>barplot</code>	91
5.17.	Función <code>boxplot</code>	91
5.18.	Función <code>pairs</code>	92

Índice general

5.19. Gráficos múltiples	94
5.20. Funciones de uso de pantalla	95
5.21. Diagrama de sectores	97
5.22. Diagramas de estrella	98
5.23. Representaciones tridimensionales	101
6. Fórmulas y Modelos	104
6.1. Fórmulas	104
6.2. La función <code>lm</code>	107

Capítulo 1

R Un entorno de análisis y programación estadísticos

1.1. Introducción

R es un entorno de análisis y programación estadísticos que, en su aspecto externo, es similar a S y además es gratuito¹.

Aunque comenzar a utilizar R es más complejo que comenzar a utilizar algunos programas manejados mediante menú, no es excesivamente difícil y sin embargo tiene muchas ventajas sobre ellos. A lo largo del texto veremos cómo utilizar R para realizar análisis convencionales, como los que se encuentran en SAS, SPSS o STATGRAPHICS, y dejaremos patente sus posibilidades de análisis más complejos, así como el desarrollo de nuevos análisis.

No olvide que una de las mejores fuentes de información es Google. Cuando tenga dudas que no pueda resolver directamente con los manuales, acuda a <http://google.com>.

1.2. Instalación

El primer paso para la utilización de R es instalarlo en el ordenador. R puede conseguirse a partir de <http://www.r-project.org>, en varias direcciones de Internet. Una de ellas² es la primaria y el resto son imágenes especulares (**mirrors**) de la misma. En cualquiera de estas direcciones se encuentra todo el proyecto R. Este proyecto incluye las fuentes para construir R, lo que permite saber completamente cómo está construido el programa, así como las direcciones donde se pueden conseguir las herramientas necesarias para hacerlo. Muchas personas no desearán realizar este trámite, por lo que también existen distribuciones binarias de R³. Además se encuentra allí documentación, programas en

¹En realidad es algo más que gratuito. Se distribuye de acuerdo a **GNU GENERAL PUBLIC LICENSE** como puede consultarse en el propio programa escribiendo `?license`. Esta gratuidad implica, contra lo que pudiera parecer a primera vista, que un gran número de personas colaboran desarrollándolo.

²El proyecto se denomina CRAN, Comprehensive R Archive Network. Hay varios juegos de palabras en las denominaciones, como la del propio R.

³A lo largo de este texto haremos referencia a la versión de Windows

1.3. Primer contacto

R, referencias a otras direcciones de interés, etc.

Si descarga el archivo, la instalación puede hacerla sin más que ejecutarlo. La instalación sólo realiza cambios mínimos en Windows y copia los archivos necesarios en un directorio. Su desinstalación es sencilla y completa.

El directorio habitual donde realizar la instalación es **C:/Archivos de programa**. El programa crea en él un subdirectorio, **R**, y por cada versión un subdirectorio de este último donde copia todos los archivos, por ejemplo **R-3.2.2** para la versión 3.2.2.

En la instalación es conveniente seleccionar todas las opciones que se ofrecen, de tal modo que siempre disponga de todas las ayudas posibles.

El programa de instalación puede crear en el escritorio un acceso directo al programa, que se encuentra en el directorio de instalación, en **bin/Rgui.exe**. También se encuentra allí otro programa, **bin/Rterm.exe**, utilizado para ejecución asíncrona del programa.

1.3. Primer contacto

Aunque es posible leer estas notas sin un ordenador, es conveniente tener uno con el programa instalado e ir comprobando cada una de las explicaciones. Si tiene instalado el programa y lo ejecuta, puede escribir la orden **demo()**, en la que no debe olvidar los paréntesis. El programa le contesta con una serie de opciones que puede seleccionar. Si, por ejemplo, escribe **demo(graphics)**, verá algunos de los gráficos que R es capaz de construir. Cuando termine, cierre el programa pulsando la cruz de la esquina superior derecha y seleccione la opción **No**.

1.4. Conceptos iniciales

R es un lenguaje interpretado. Esto quiere decir que puede escribir órdenes y R las ejecutará inmediatamente. Así, por ejemplo, puede escribir **2+2** o **7*2** y obtendrá los resultados esperados. De aquí se deduce que **2**, **7**, **+** y ***** son conceptos que el lenguaje entiende. De hecho puede utilizar R como una calculadora, ya que es capaz de manejar las operaciones elementales fácilmente. Incluso⁴ puede utilizar valores lógicos, **T** (true=verdadero) y **F** (false=falso), o números complejos, sin más que utilizar la letra **i** para la parte imaginaria, y teniendo en cuenta el uso de paréntesis. Las operaciones de suma, resta, multiplicación, división, exponenciación, división entera y módulo se realizan mediante los símbolos **+**, **-**, *****, **/**, **^**, **%%** y **%/**.

```
> 2+3i*1-1i
[1] 2+2i
> (2+3i)*(1-1i)
[1] 5+1i
```

Tenga en cuenta que R no evalúa una expresión hasta que tiene la certeza de que se ha terminado su escritura. Así, **si la expresión comienza con un paréntesis, podrá utilizar varias líneas para su escritura**. También puede utilizar llaves, **{ }**,

⁴Los tipos de datos soportados incluyen **logical**, **integer**, **single**, **double**, **complex** y **character**. El modo **double** se denomina también **numeric**.

1.4. Conceptos iniciales

con un sentido más amplio, ya que todo lo que incluyen se considera una sola acción.

```
> (2
+
+
+
+ *
+
+ 3)
[1] 6
```

R es un lenguaje funcional, esto es, realiza las tareas a través de funciones. La primera función para estudiar⁵ es `help`. Si escribes `help` aparece la definición de la función `help`. Si quiere obtener el resultado de la ejecución de la función, debe escribir entre paréntesis el conjunto de argumentos que quiere suministrar a la misma, que podrá ser vacío en algún caso. Así, por ejemplo, `help()`. En este caso aparecerá la ayuda referida precisamente a la función `help()` que ha solicitado.

En general, para cualquier función, su escritura sin los paréntesis indica al lenguaje que debe devolver la definición, en tanto que al incluir paréntesis, le indica que intente ejecutarla.

También puede llegar a la ayuda seleccionando la opción `Help` en el menú.

En la ayuda se encuentran a menudo ejemplos de uso. Puede copiar una parte de ellos y pegarla en la ventana de órdenes para probar su ejecución. Si desea ejecutarlos todos hasta con escribir la función `example` cuyo argumento es el nombre de la función de la que desea ejecutar los ejemplos. Así, para la función `mean` puede ejecutar los ejemplos escribiendo `example(mean)`.

```
> example(mean)

mean> x <- c(0:10, 50)

mean> xm <- mean(x)

mean> c(xm, mean(x, trim = 0.1))
[1] 8.75 5.50

mean> mean(USArrests, trim = 0.2)
  Murder  Assault UrbanPop   Rape
    7.42   167.60   66.20   20.16
```

Si no conoce el nombre exacto de una función, puede pedir ayuda utilizando la función `apropos` que localiza los objetos cuyo nombre coincide parcialmente con el argumento que se le suministra.

```
> apropos("hel")
[1] "contr.helmert" ".helpForCall" "help" "help.search"
[5] "help.start" "link.html.help" "shell" "shell.exec"
```

⁵En R son distintas las letras mayúsculas y las minúsculas. Si escribes `Help()` obtendrá un error, ya que no existe esa función.

1.4. Conceptos iniciales

La segunda función para estudiar es `q`. Si escribe `q` verá su definición.

```
function (save = "default", status = 0, runLast = TRUE)
.Internal(quit(save, status, runLast))
<environment: namespace:base>
```

Si escribe `help(q)` obtendrá la página de ayuda referente a la función, en donde podrá leer que permite terminar la sesión de trabajo. Si escribe `q()` la ejecutará, por lo que, en este caso, intentará terminar la sesión de trabajo. Si la escribe, R preguntará si desea salvar el espacio de trabajo, esto es, si se desea que los cambios realizados en la definición de objetos se hagan permanentes. Conteste afirmativamente si desea hacerlo y negativamente en caso contrario.

1.4.1. Teclas de edición

Puede obtener ayuda sobre las teclas de edición seleccionando en el menú la opción **Help** y a continuación **Console**. La mayoría son las comunes de **Windows**.

Para repetir cualquier orden, puede marcar esta en la pantalla y copiarla, tras lo cual puede pegarse, bien en la propia ventana de órdenes bien en cualquier procesador desde el que posteriormente se seguirá el camino inverso.

1.4.2. Redireccionamiento

De modo análogo a **Unix** y a **MS-DOS**, si escribe una expresión, puede redireccionar su salida, de tal modo que no aparezca escrita en pantalla, sino que vaya dirigida a otro sitio. Así, si escribe `7*2->kk`, no obtendrá ningún resultado en pantalla, sino que este se almacena en un *objeto* de nombre `kk`. Puede recuperar este objeto y utilizarlo en cualquier momento sin más que escribir su nombre⁶. Así `kk` devuelve el valor `14` que además puede ser utilizado en cualquier expresión, como en `kk^3`, que devolverá el valor `2744`.

Este redireccionamiento puede escribirse en la dirección en que se ha escrito o en la contraria, en la cual es más intuitivo que la parte de la izquierda se está definiendo. Así, `kk <- -23*5`, almacena el resultado de la operación en el objeto `kk`, borrando el anterior valor almacenado. También puede usar el símbolo **igual**, `=`, para realizar esta asignación, como en `kk = -23*5`.

¿Dónde se almacenan los objetos? ¿Cómo se almacenan? Encontrará la respuesta a estas preguntas posteriormente, al estudiar la función `search`. Por ahora conténtese con saber que se almacenan en el disco duro, por lo que los objetos son permanentes, en el sentido de que aún terminando la sesión y comenzando posteriormente otra, los objetos siguen estando a disposición del usuario, siempre que salve el espacio de trabajo. El espacio de trabajo puede salvarse en cualquier momento, para prevenir por ejemplo los efectos de un corte de luz, eligiendo la opción **File** y a continuación **Save Workspace...**

R dispone de dos funciones que permiten gestionar los objetos, que son⁷ `ls` y `rm`, y corresponden, respectivamente, a observar qué objetos existen y a eliminar un objeto concreto. Ellas, unidas al redireccionamiento, que permite la creación de un objeto, son suficientes para la gestión completa de los objetos.

⁶Recuerde que al terminar la sesión de trabajo, si elige la opción No, perderá todos los cambios realizados, luego, en dicho caso, `kk` no estaría disponible en una sesión posterior.

⁷Estas funciones, como otras muchas, existen en Unix con significados similares.

Los parámetros de una función pueden darse por dos métodos: Por posición o por nombre. En el primer procedimiento se hace coincidir cada uno de los argumentos con el que ocupa la misma posición en la definición de la función. En el segundo se precede cada argumento con el nombre al que corresponde⁸. En casos elementales es más cómodo el primero, pero el segundo no sólo es más cómodo en casos más complejos, como cuando se desea dar valor al octavo parámetro sin dárselo a los anteriores, sino que es más claro, pues especifica a qué parámetro corresponde, e incluso en algunos casos, como verá más adelante con la función `plot`, es obligatorio, pues hay argumentos que pueden repetirse. Cuando se utiliza el primer método debe incluir las comas para indicar que desea saltar algún argumento. Así, si la función `ls` admite los argumentos (`name`, `pos`, `envir`, `all.names`, `pattern`), independientemente de su significado, serían equivalentes las expresiones siguientes:

```
ls(,2)  ls(pos=2)  ls(,pos=2)  ls(po=2)  ls(pos=2,"")
```

Cuando los argumentos se dan por nombre, su orden es irrelevante, por lo que, suponiendo que la siguiente sea una orden correcta,

```
matrix(data=0, nrow=4, ncol=5)
```

será equivalente a

```
matrix(ncol=5,data=0, nrow=4)
```

Sin que se preocupe por ahora de la función `objects` ni del segundo argumento de la función `ls`, se puede afirmar que la función `ls` da una lista de los objetos que se encuentran en la posición `pos=1`.

En efecto, si escribe `ls()`, obtendrá una lista de nombres, como por ejemplo la siguiente,

```
[1] "kk"
```

en donde aparece el objeto `kk`. La función `rm` aplicada a dicho objeto, `rm(kk)`, lo eliminará, por lo que la función `ls` ya no lo devolverá ni podrá utilizar este objeto en ningún cálculo posterior.

1.4.3. Edición de un objeto

Si desea editar un objeto, puede escribir su definición en pantalla, marcarla, copiarla y pegarla en un editor; allí editarla y luego, con el paso inverso, pegarla en la ventana de órdenes, con lo cual se actualiza. Este es el método de trabajo general de *Windows*, pero en otro entorno sería diferente. Es posible hacerlo con un método directo, utilizando la función `edit`⁹.

La sintaxis¹⁰ es `edit(name, file, title, editor)`, donde `name` es el nombre de un objeto cualquiera que se desea editar, si no se especifica ninguno se utilizará el contenido de (`file`), `file` es el nombre del archivo donde debe guardarse el objeto durante la edición y que puede omitirse siempre, puesto que el sistema suministra uno, y, por último, `editor` es una variable de tipo cadena de caracteres que especifica el nombre de un editor. Si no se especifica ninguno, `ls` usa el predeterminado de *Windows*, *Notepad*. En este último caso, (`title`) se utilizará como identificador en la cabecera.

⁸Habitualmente basta con indicar las letras que identifiquen el parámetro de modo único

⁹Incluso es posible utilizar *vi* o *emacs*

¹⁰Recuerde que, para todas las funciones que aparezcan en el texto, debería pedir la ayuda correspondiente, e incluso, más adelante, leer su definición.

1.4. Conceptos iniciales

Es importante tener en cuenta que esta función no modifica el objeto, sino que el objeto editado, en su caso, es devuelto al sistema, por lo que debe redirigirse a un objeto, que puede ser otro o el mismo que se está editando, para que quede almacenado.

Así, la orden `Tarta<-edit(Tarta,,"q")` edita el objeto **Tarta** con el editor **q**¹¹ y guarda el resultado en el propio objeto **Tarta**.

Si, después de ejecutar la función, escribe `edit()`, recuperará el último texto escrito. Ello es útil, por ejemplo, cuando se produce un error en la escritura de una función, ya que no se almacena la misma, y esta opción permite recuperar todo lo que se haya escrito.

```
> Malo <- 1
> Malo <- edit(Malo)
```

Si en el editor escribe `function(x) x+`, al terminar la edición obtendrá

```
Error in edit(name, file, title, editor) :
  An error occurred on line 3
use a command like
x <- edit()
to recover
```

y, consecuentemente, no se altera el objeto,

```
> Malo
[1] 1
```

pero puede recuperar lo escrito en el editor sin más que escribir

```
> Malo <- edit()
```

Si ahora escribe en el editor `function(x) x+2`, al terminar la edición obtendrá

```
> Malo
function(x) x+2
```

Si el objeto a editar es una matriz o una hoja de datos, estructuras que se introducirán más adelante, el editor conduce a la función `data.entry`.


¹¹El nombre del editor se construirá añadiendo, si es necesario, cualquiera de las extensiones admitidas por Windows, por ejemplo, podrá ser `q.com`, `q.exe` o `q.bat`, por el orden en que estén definidas en Windows, y se buscará en el directorio de trabajo y en la lista de búsqueda definida por Windows

Capítulo 2

Algunas clases de objetos comunes

2.1. Vector

El primer tipo de objeto que puede manejar es el vector. Por ejemplo, el símbolo de dos puntos, situado entre dos números, construye un vector de modo sencillo, tanto en orden ascendente como descendente:

```
> 1:10
[1] 1 2 3 4 5 6 7 8 9 10
> 10:1
[1] 10 9 8 7 6 5 4 3 2 1
> 1:3.5 
[1] 1 2 3
> 1.9:3.5
[1] 1.9 2.9
```

El vector comienza en el primer número suministrado y finaliza en el segundo o en un número anterior sin sobrepasarlo, tanto en orden ascendente como descendente.

Es ilustrativo observar que el vector `1:1` es precisamente el número 1, por lo que se puede afirmar que, en realidad, todos los números son considerados como vectores, lo que ayuda a comprender las operaciones relacionadas con vectores.

Pero los vectores no están limitados a ser sucesiones equiespaciadas de números, ni siquiera a serlo de números. Podrá construir vectores fácilmente con la función de concatenación, `c`. Verá sus propiedades con más detalle posteriormente, pero ahora le basta con saber que admite como argumento varios vectores¹ y los concatena en uno solo. Así

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

¹Recuerde que un sólo número es equivalente a un vector

2.1. Vector

```
> c(1:10,2,7,1,6,2:-5)
[1]  1  2  3  4  5  6  7  8  9 10  2
[12]  7  1  6  2  1  0 -1 -2 -3 -4 -5
> c()
NULL
```

En el último caso, el resultado es el objeto **NULL**. Si los elementos van entrecomillados, serán cadenas de caracteres y obtendrá un vector de este tipo. Estos vectores son útiles, por ejemplo, para dar nombres a las variables utilizadas en un análisis o dar nombre a los meses del año², como hacemos a continuación.

```
> nombre.mes = c("Enero","Febrero","Marzo","Abril","Mayo",
,"Junio","Julio","Agosto","Septiembre","Octubre"
,"Noviembre","Diciembre")
> c("Hola", "Uno", "Dos")
[1] "Hola" "Uno"  "Dos"
```

Un vector está formado por elementos del mismo tipo, así pues, no pueden mezclarse números y cadenas de caracteres, ya que se transformarán en cadenas de caracteres, como ocurre en el siguiente ejemplo:

```
> c("Hola", "Uno", "Dos",3)
[1] "Hola" "Uno"  "Dos"  "3"
```

Del mismo modo, si mezcla números reales y complejos, se convertirán todos a complejos:

```
> c(1,1+2i)
[1] 1+0i 1+2i
```

Puede asignar nombre a los elementos de un vector mediante la función **names**. De hecho, esta función permite dar nombre a los elementos de cualquier objeto. Aplicada a un vector sería así:

```
> x<-1:5
> x
[1] 1 2 3 4 5
> names(x)<-c("I","II","III","IV","V")
> x
  I  II III IV  V
1  2  3  4  5
```

La operación **seq** es un caso particular de la función **seq** que permite construir vectores que son sucesiones equiespaciadas. Consultando la ayuda, **help(seq)**, verá que posee los siguientes argumentos:

from Valor inicial de la sucesión.

to Valor final de la sucesión.

by Espaciado entre los valores.

²La función **month.name** contiene estos nombres en inglés.

2.1. Vector

length Longitud del valor resultante.

along Un objeto cuya longitud se usará para el objeto a construir.

Es un error utilizar todos los argumentos simultáneamente. Sólo debe dar los necesarios, el resto se calculan a partir de los dados. Si no indica ninguno, todos valen 1.

Puede crear un vector con la función **vector**. Su forma es

vector(mode="logical", length=0)


donde **mode** indica el modo o tipo del vector y **length** es su longitud.

mode devuelve o modifica el tipo de un objeto. Los modos posibles incluyen los siguientes: **logical**, **numeric**, **complex**, **character**, **null**, **list**, **function**, **graphics**, **expression**, **name**, **frame**, **raw** y **unknown**. El tipo **numeric** engloba los diferentes tipos numéricos correspondientes a números reales.

```
> x<-1:5
> mode(x)
[1] "numeric"
> mode(x)<-"complex"
> x
[1] 1+0i 2+0i 3+0i 4+0i 5+0i
> x<-vector("numeric",5)
> x
[1] 0 0 0 0 0
```

2.1.1. Operaciones con vectores

Puede realizar operaciones con vectores directamente. Así,

```
> (1:10)*2 
[1] 2 4 6 8 10 12 14 16 18 20
> (1:10)*(1:10)
[1] 1 4 9 16 25 36 49 64 81 100
```

donde observará, en primer lugar, el uso de paréntesis, que garantiza que las operaciones se realizan en el orden deseado (Si no los utiliza, R sigue un orden de precedencias en las operaciones. Compruebe $(1:10)^2$ y $1:10^2$.) y, en segundo lugar, que la multiplicación se realiza componente a componente. Del mismo modo actuaría cualquier otra operación. En realidad, la primera de las dos operaciones anteriores, que parece más intuitiva a primera vista, lo es menos. Lo que R ha hecho es replicar el vector 2 (2:2) tantas veces como ha hecho falta (10) para que la operación coincidiese con una como la segunda. Si en esa replicación hubiese llegado a sobrar algún elemento se hubiese indicado mediante un mensaje. Los dos ejemplos siguientes ilustran este concepto, que será muy utilizado, por lo que debe asimilarlo correctamente.

```
> (1:10)+(1:5)
[1] 2 4 6 8 10 7 9 11 13 15
> (1:10)+(2:5)
[1] 3 5 7 9 7 9 11 13 11 13
Warning message:
longer object length is not a multiple
of shorter object length in: (1:10) + (2:5)
```

2.1. Vector

Esto es, $(1:10)^2$ equivale a $(1:10)^{(2:2)}$

```
> (1:10)^(2:2)
[1] 1 4 9 16 25 36 49 64 81 100
```

Compare con $(1:10)^{(1:3)}$

```
> (1:10)^(1:3)
[1] 1 4 27 4 25 216 7 64 729 10
```

Warning message:

```
longer object length is not a multiple of
shorter object length in: (1:10)^(1:3)
```

La longitud de un vector puede obtenerse con la función `length`, como puede ser `length(1:10)`. Puede consultar la ayuda sobre esta función con la función `help(length)` donde encontrará que sirve también para *definir* la longitud del vector, haciéndolo más largo o más corto en su caso. En caso de hacerlo más largo, se completará con `NA`³, esto es con valores faltantes.

```
> kk<-1
> kk
[1] 1
> length(kk)<-10
> kk
[1] 1 NA NA NA NA NA NA NA NA NA
> kk<-"Pepe"
> kk
[1] "Pepe"
> length(kk)<-10
> kk
[1] "Pepe" NA NA NA NA NA NA NA NA NA
```

Puede referirse al elemento que ocupa la posición `i` de un vector, `x`, mediante `x[i]`. También puede referirse a un subconjunto mediante un subconjunto de subíndices. Si los subíndices son números naturales, hacen referencia a las posiciones dentro del vector. Si los subíndices son valores lógicos, se corresponden con los mismos elementos del vector, pero sólo hacen referencia a los que tienen un subíndice con valor lógico TRUE.

```
> x<-(1:10)^2
> x
[1] 1 4 9 16 25 36 49 64 81 100
> x[3]
[1] 9
> x[c(3,5,9)]
[1] 9 25 81
> x[seq(1,9,by=2)]
[1] 1 9 25 49 81
```

³Iniciales de Not Available. También encontrará en otros casos el valor NaN, Not a Number, en casos en que el resultado no puede ser calculado.

2.2. Factor

El ejemplo anterior crea un objeto, **x**, que contiene los cuadrados de los números naturales de 1 a 10. Luego se ha referido, en primer lugar, al tercer elemento; en segundo lugar, a los elementos tercero, quinto y noveno; y, por último, a los elementos impares. En el siguiente caso se hace referencia a los elementos que cumplen cierta condición.

```
> sexo
Error: Object "sexo" not found
> sexo<-c("H","H","M","M","H")
> sexo
[1] "H" "H" "M" "M" "H"
> nombre
Error: Object "nombre" not found
> nombre<-c("Pedro","Juan","Ana","Lola","Rafael")
> nombre
[1] "Pedro" "Juan" "Ana" "Lola" "Rafael"
> sexo=="H"
[1] T T F F T
> nombre[sexo=="H"]
[1] "Pedro" "Juan" "Rafael"
```

La referencia a los elementos impares de **x** puede hacerla también así:

```
> x[1:10%2==1]
[1] 1 9 25 49 81
```

2.2. Factor

Un **factor** es un vector de cadenas de caracteres que se interpreta de modo especial mediante el atributo **levels** que indica los valores numéricos posibles, cada uno de los cuales se asocia a cada cadena diferente.

La forma de creación es mediante la función **factor**, cuya utilización es

```
factor(x, levels=, labels=, exclude= , ordered=)
```

siendo los argumentos los siguientes:

x son los datos a partir de los cuales se genera el factor

levels es un vector de niveles opcional. Los valores de **x** que no correspondan a uno de los indicados se sustituyen por NA. El valor predeterminado de este parámetro es la lista ordenada de valores distintos de **x**.

labels es un vector de valores que se usará como etiqueta de los niveles. El valor predeterminado es **as.character(levels)**.

exclude es un vector de valores que deben excluirse de la formación de niveles y sustituirse por NA.

ordered es un valor lógico que indica si lo es el factor.

Un factor corresponde a una variable nominal u ordinal, dividida en categorías, y que se utiliza, por ejemplo, para dividir una población en grupos. Es un concepto muy importante que debe asimilar a través de la práctica.

Ejemplo

```
> factor(c("A","B","A"))
[1] A B A
Levels: A B
> levels(factor(c("A","B","A")))
[1] "A" "B"
```

2.3. Matriz

El siguiente tipo de objeto que puede utilizar es la matriz, considerada como una variable indexada mediante dos índices. Para crearlas puede utilizar la función `matrix`. Los parámetros de esta función son:

data Vector que contiene los valores que formarán la matriz. Debe tener en cuenta que si no es suficientemente grande, se repetirá las veces que sea necesario.

nrow Número de filas. Si especifica el número de columnas, y no da el de filas, se calcula este mediante `ceiling(length(data)/ncol)`. Si no especifica ninguno, se toma `ncol=1`.

ncol Número de columnas. Si especifica el número de filas, y no da el de columnas, se calcula este mediante `ceiling(length(data)/nrow)`.

byrow Variable lógica que indica si la matriz debe construirse por filas o por columnas. El valor predeterminado es F.

dimnames Lista⁴ de longitud 2 con los nombres de las filas y las columnas. Cada componente de la lista debe tener longitud 0 o ser un vector de cadenas de caracteres con la misma longitud que la dimensión correspondiente de la matriz.

Ejemplos

`matrix <- matrix(0, 4, 5)` Crea una matriz de 4×5 cuyos elementos son ceros (ya que el cero se repite tantas veces como haga falta para crearla).

Si ahora escribe:

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

R crea la matriz que aparece, que como ha ordenado tiene cinco filas y por tanto $(10/5)$ dos columnas, y se rellena por columnas. El mismo resultado obtendría con `matrix(data=1:10, ncol=2, nrow=5, byrow=F)`.

Si quiere dar nombres a las columnas (o a las filas) puede hacerlo asignando valores al parámetro `dimnames`

⁴Verá el concepto de lista a continuación

2.3. Matriz

```
> matrix(1:10,5,dim=list(c(),c("Primero","Segundo")))
      Primero Segundo
[1,]      1       6
[2,]      2       7
[3,]      3       8
[4,]      4       9
[5,]      5      10
```

Cuando las longitudes no son las requeridas exactamente, se repetirá el vector si es necesario, pudiendo recibir un mensaje de advertencia si alguna de las veces no se utiliza completo.

```
> matrix(1:10,3)
      [,1] [,2] [,3] [,4]
[1,]     1     4     7    10
[2,]     2     5     8     1
[3,]     3     6     9     2
Warning message:
data length [10] is not a sub-multiple or multiple
of the number of rows [3] in matrix
```

Puede hacer referencia a una submatriz, en particular a un elemento de la matriz, sin más que indicar los índices a que hace referencia. Si dispone de los datos de peso, altura y edad de un grupo de personas puede escribir:

```
> datos<-c(
77 , 1.63 , 23,
58 , 1.63 , 23,
89 , 1.85 , 26,
55 , 1.62 , 23,
47 , 1.60 , 26,
60 , 1.63 , 26,
54 , 1.70 , 22,
58 , 1.65 , 23,
75 , 1.78 , 26,
65 , 1.70 , 24,
82 , 1.77 , 28,
85 , 1.83 , 42,
75 , 1.74 , 25,
65 , 1.65 , 26)
> matriz<-matrix(datos,ncol=3,
+ dimnames=list(c(),c("Peso","Altura","Edad")), byrow=T)
> matriz[,1]
[1] 77 58 89 55 47 60 54 58 75 65 82 85 75 65
> matriz[, "Peso"]
[1] 77 58 89 55 47 60 54 58 75 65 82 85 75 65
> matriz[4,3]
Edad
23
> matriz[1,]
Peso Altura Edad
77 1.63 23
```


2.3. Matriz

Las operaciones realizadas, en primer lugar, guardan en el objeto **matriz** la matriz de datos construida a partir del vector de datos. Puede hacer referencia a la columna (vector) de pesos mediante **matriz[,1]** o **matriz[, "Peso"]**, a un elemento concreto con **matriz[4,3]**, que da el valor de la tercera variable en el cuarto individuo, o a un individuo concreto, como **matriz[1,]** que da las observaciones del primer individuo.

Por otra parte, además de valores naturales, los subíndices pueden tomar también valores lógicos. En ese caso, se está haciendo referencia sólo a las posiciones cuyo subíndice tome el valor **TRUE**.

```
> x=1:3
> x
[1] 1 2 3
> x[c(T,F,T)]
[1] 1 3
```

2.3.1. Operaciones con matrices

Los operadores aritméticos pueden utilizarse sin problemas. Recuerde que su uso es análogo al caso de los vectores, esto es, componente a componente.

```
> A1<-matrix(1:10,2,5)
> A1
      [,1] [,2] [,3] [,4] [,5]
[1,]     1     3     5     7     9
[2,]     2     4     6     8    10
> A2<-matrix((1:10)^2,2,5)
> A2
      [,1] [,2] [,3] [,4] [,5]
[1,]     1     9    25    49    81
[2,]     4    16    36    64   100
> A1*A2
      [,1] [,2] [,3] [,4] [,5]
[1,]     1    27   125   343   729
[2,]     8    64   216   512  1000
> A2/A1
      [,1] [,2] [,3] [,4] [,5]
[1,]     1     3     5     7     9
[2,]     2     4     6     8    10
> A1%%3
      [,1] [,2] [,3] [,4] [,5]
[1,]     1     0     2     1     0
[2,]     2     1     0     2     1
> A1*(2:4)
      [,1] [,2] [,3] [,4] [,5]
[1,]     2    12    15    14    36
[2,]     6     8    24    24    20
> A1*(2:6)
      [,1] [,2] [,3] [,4] [,5]
[1,]     2    12    30    21    45
[2,]     6    20    12    32    60
```



2.3. Matriz

También es posible realizar el producto matricial mediante el operador `%*%`.

```
> B<-matrix(1:10,ncol=2)
> B
      [,1] [,2]
[1,]    1    6
[2,]    2    7
[3,]    3    8
[4,]    4    9
[5,]    5   10
> A1%*%B
      [,1] [,2]
[1,]   95  220
[2,]  110  260
```

La función `crossprod` devuelve el producto matricial cruzado de dos matrices, esto es, la traspuesta de la primera matriz, multiplicada por la segunda. Si no especifica segunda matriz, R la toma igual a la primera, con lo que obtiene el conocido producto $X'X$.

```
> crossprod(A1)
      [,1] [,2] [,3] [,4] [,5]
[1,]    5   11   17   23   29
[2,]   11   25   39   53   67
[3,]   17   39   61   83  105
[4,]   23   53   83  113  143
[5,]   29   67  105  143  181
```

La función `outer` realiza el producto exterior de dos matrices (o vectores) sobre una función dada. La forma de uso generalizada es

```
outer(X, Y, FUN="*", ...)
```

donde **X** e **Y** son los argumentos de la función **FUN**, que es cualquier función que acepte al menos dos vectores como argumentos y devuelva un valor único. Después de la función pueden pasarse otros argumentos adicionales que sean necesarios para la función, lo que se indica mediante los puntos suspensivos.

El resultado es una variable multiindexada (*array*), cuyo vector de dimensiones es la concatenación de los vectores de dimensión de **X** e **Y** y donde **FUN(X[i,j,k,...], Y[a,b,c,...])** es el valor del elemento **[i,j,k,...,a,b,c,...]**.

También puede usarse en forma de operador mediante `%o%`, en cuyo caso la función utilizada es el producto.

```
> x<-1:3
> y<-1:4
> outer(x,y)
      [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    2    4    6    8
[3,]    3    6    9   12
```

t calcula la matriz traspuesta.

2.3. Matriz

```
> t(B)
      [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3    4    5
[2,]    6    7    8    9   10
```

solve invierte una matriz, si sólo se le da un argumento, y también resuelve sistemas lineales de ecuaciones, cuando se le dan dos. El primer argumento es una matriz que debe ser cuadrada no singular, y el segundo argumento, opcional, es un vector o matriz de coeficientes.

Las siguientes órdenes crean una matriz cuadrada, $B'B$, la invierten, y comprueban que el producto de la matriz y su inversa es la matriz identidad (salvo errores de redondeo).

```
> crossprod(B)
      [,1] [,2]
[1,]   55  130
[2,]  130  330
> solve(crossprod(B))
      [,1] [,2]
[1,]  0.264 -0.104
[2,] -0.104  0.044
> solve(crossprod(B))%*%crossprod(B)
      [,1] [,2]
[1,]  1.000000e+00 8.604228e-15
[2,] -3.538836e-15 1.000000e+00
```

Para resolver el sistema de ecuaciones:

$$3x + 2y = 5, x - y = 0$$

que en forma matricial se escribe

$$\begin{pmatrix} 3 & 2 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 5 \\ 0 \end{pmatrix}$$

escriba

```
> solve(matrix(c(3,2,1,-1),ncol=2,byrow=T),c(5,0))
[1] 1 1
```

siendo la solución, por tanto, $x = 1, y = 1$.

eigen calcula los autovalores y autovectores de una matriz cuadrada. El primer argumento de esta función, **x**, es la matriz. El segundo, optativo, es **symmetric**. Es de tipo lógico y se utiliza para indicar que la matriz es simétrica (o hermitiana en el caso complejo), en caso contrario se inspecciona para comprobarlo. El tercer argumento, optativo, es **only.values=F**. También es de tipo lógico y en caso de ser **T** no se calculan los autovectores.

El resultado es una lista: El primer elemento es **values**, que es el vector de autovalores, y el segundo es **vectors**, que es la matriz de autovectores por columnas. A partir de los autovalores puede escribir una función para calcular fácilmente el determinante de la matriz:

```
determinante <- function(x) prod(eigen(x)$values)
```

2.4. Variable multiindexada

No es necesario crear esta función, ya que R suministra las funciones `det` y `determinant` que lo realizan directamente.

2.4. Variable multiindexada

La generalización de los vectores y matrices son las variables multiindexadas, denominadas **arrays**, y de las cuales son casos particulares los vectores y matrices. Para crear una variable multiindexada se utiliza la función

```
array(data, dim, dimnames)
```

donde **dim** es un vector de dimensiones.

Así, por ejemplo, la siguiente orden crea una variable triindexada. Puede hacer referencia a cualquier subconjunto de la misma, de modo similar a las matrices.

```
> Tri <- array(1:(2*3*5), c(2,3,5))
```

```
> Tri
```

```
, , 1
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

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

```
, , 3
     [,1] [,2] [,3]
[1,]   13   15   17
[2,]   14   16   18
```

```
, , 4
     [,1] [,2] [,3]
[1,]   19   21   23
[2,]   20   22   24
```

```
, , 5
     [,1] [,2] [,3]
[1,]   25   27   29
[2,]   26   28   30
```

```
> Tri[1,1,1]
```

```
[1] 1
```

```
> Tri[1,,]
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    7   13   19   25
[2,]    3    9   15   21   27
[3,]    5   11   17   23   29
```

2.5. Lista

aperm permite transponer una variable multiindexada indicando la propia variable como primer argumento y el vector de permutaciones de índices como segundo.

```
> x <- array(1:6, 2:3)
> x
      [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
> xt <- aperm(x, c(2,1))
> xt
      [,1] [,2]
[1,]    1    2 [2,]    3    4
[3,]    5    6
> xt <- aperm(x, c(2,1),F)
> xt
      [,1] [,2] [,3]
[1,]    1    5    4
[2,]    3    2    6
```

2.5. Lista

Los tipos anteriores corresponden a variables de un solo tipo, aunque indexadas. Si necesita disponer de variables de varios tipos, puede recurrir a las listas. Una lista se construye con la función **list** que devuelve un objeto de tipo lista con tantos componentes como argumentos se le suministren y es el más utilizado para devolver el resultado de una función.

```
> list(A=1:10,B=nombre.mes)
$A:
 [1]  1  2  3  4  5  6  7  8  9 10

$B:
 [1] "Enero"      "Febrero"    "Marzo"      "Abril"
 [5] "Mayo"       "Junio"      "Julio"      "Agosto"
 [9] "Septiembre" "Octubre"    "Noviembre"  "Diciembre"
```

Puede referirse a cada uno de los elementos de la lista de dos formas distintas: Si tiene nombre, como en este caso, mediante el nombre de la lista, el símbolo \$, y el nombre del elemento. En cualquier caso, siempre puede referirse a él mediante el índice de posición entre dobles corchetes.

```
> list(A=1:10,B=nombre.mes)$A
 [1]  1  2  3  4  5  6  7  8  9 10
> list(A=1:10,B=nombre.mes)[[1]]
 [1]  1  2  3  4  5  6  7  8  9 10
```

La segunda forma pone más de relieve la estructura de la lista. Además, si el elemento referido es un objeto indexado podrá de nuevo hacerse referencia a índices para designar sus componentes.

2.5. Lista

```
> lista1<-list(A=1:10,B=nombre.mes)
> lista2<-list(lista1,C="Otro nombre",Matriz=matriz)
> lista2
[[1]]:
[[1]]$A:
 [1]  1  2  3  4  5  6  7  8  9 10

[[1]]$B:
 [1] "Enero"      "Febrero"    "Marzo"      "Abril"
 [5] "Mayo"       "Junio"      "Julio"      "Agosto"
 [9] "Septiembre" "Octubre"    "Noviembre"  "Diciembre"

$C:
 [1] "Otro nombre"

$Matriz:
      Peso Altura Edad
 [1,]   77    1.63   23
 [2,]   58    1.63   23
 [3,]   89    1.85   26
 [4,]   55    1.62   23
 [5,]   47    1.60   26
 [6,]   60    1.63   26
 [7,]   54    1.70   22
 [8,]   58    1.65   23
 [9,]   75    1.78   26
[10,]   65    1.70   24
[11,]   82    1.77   28
[12,]   85    1.83   42
[13,]   75    1.74   25
[14,]   65    1.65   26

> lista2[[3]][2,3]
      Edad
      23
> lista2[[1]][[1]][2]
 [1] 2
```

La diferencia fundamental entre las tres formas, `[`, `[[` y `$` es que la primera permite seleccionar varios elementos, en tanto que las dos últimas solo permiten seleccionar uno. Además, `$` no permite utilizar índices calculados. El operador `[[` necesita que se le indiquen todos los índices (ya que debe seleccionar un sólo elemento) en tanto que `[` permite obviar índices, en cuyo caso se seleccionan todos los valores posibles.

Si se aplican a una lista, `[[` devuelve el elemento de la lista especificado y `[` devuelve una lista con los elementos especificados.

2.6. Hoja de datos

La adaptación de la matriz de datos al uso habitual en Estadística es el objeto `data.frame`, que se puede traducir como *Hoja de datos*, aunque puede ser preferible mantener el nombre original por no existir una traducción estándar. La diferencia fundamental con la matriz de datos es que este objeto no tiene por qué estar compuesto de elementos del mismo tipo. Los objetos pueden ser vectores, factores, matrices, listas e incluso hojas de datos. Las matrices, listas y hojas de datos, contribuyen con tantas variables como columnas tengan. Los vectores numéricos y los factores se incluyen directamente y los vectores no numéricos se fuerzan como factores. Si no desea la expansión o la conversión, debe utilizar la función `I`. En caso de duda, debe tener en cuenta que el cambio se hace utilizando la función `as.data.frame`. Puede referirse a cualquier elemento de la hoja mediante dos índices, de modo similar a una matriz.

Ejemplo

Las órdenes siguientes crean una hoja de datos con dos vectores, a continuación crean una nueva hoja uniendo la anterior con un vector de caracteres que representa el sexo.

```
> hoja1<-data.frame(Peso=c(90,87,60), Altura=c(1.85,1.87,1.63))
> hoja1
  Peso Altura
1   90  1.85
2   87  1.87
3   60  1.63
> hoja2<-data.frame(hoja1,Sexo=c("H","H","M"))
> hoja2
  Peso Altura Sexo
1   90  1.85    H
2   87  1.87    H
3   60  1.63    M
> codes(hoja2[,3])
[1] 1 1 2
```



Este vector, como se ha indicado, se transforma en un factor, como puede comprobar pidiendo sus códigos. Adviértase que el uso de corchetes individuales o dobles implica diferencia en los resultados.

```
> is.vector(hoja1)
[1] FALSE
> is.matrix(hoja1)
[1] FALSE
> is.data.frame(hoja1)
[1] TRUE
> is.factor(hoja2[3])
[1] FALSE
> is.factor(hoja2[[3]])
[1] TRUE
> hoja2[3]
```

2.6. Hoja de datos

```
Sexo
1    H
2    H
3    M
> hoja2[[3]]
[1] H H M
Levels: H M
> hoja2[,3]
[1] H H M
Levels: H M
> hoja2[[,3]]
Error in "[.data.frame"(hoja2, , 3) : invalid subscript type
```

Para introducir un vector de nombres como tales, sin transformarlo en factores, debe utilizar la función `I`, como se muestra a continuación:

```
> Nombres=c("Pepe", "Juan", "Antonia")
> Nombres
[1] "Pepe"    "Juan"    "Antonia"
> Hoja3=data.frame(hoja2, Nombre=I(Nombres))
> Hoja3
  Peso Altura Sexo  Nombre
1   90   1.85   H    Pepe
2   87   1.87   H    Juan
3   60   1.63   M  Antonia
```

Las estructuras de datos que más se utilizan, son precisamente las más complejas: La hoja de datos es la más apropiada para describir unos datos a analizar⁵, y la lista es la más apropiada para devolver datos desde una función, ya que permite devolver datos de varios tipos y, además, de modo estructurado, lo que facilita utilizar el resultado de una función como entrada de otra.

Si desea seleccionar un subconjunto de una hoja de datos, puede hacerlo con la función `subset`, función que puede utilizar también en vectores.

La sintaxis en vectores es `subset(x, subset, ...)` y en hojas de datos `subset(x, subset, select, drop = FALSE, ...)` donde los argumentos son

x El objeto al que se aplica la selección, generalmente una hoja de datos.

subset La condición que se aplica para seleccionar un subconjunto. En una hoja de datos pueden utilizarse los nombres de las variables (columnas) de la hoja.

select Columnas que se desea conservar en una hoja de datos.

drop Este argumento se pasa al método de indexación de hojas de datos.

Para realizar modificaciones en una hoja de datos es muy útil la función `transform` que es de la forma `transform(x, ...)`, donde `x` es la hoja de datos en la que se van a realizar transformaciones y el resto de argumentos son transformaciones posibles definidas de la forma `variable=expresión`. Si el nombre

⁵Ver attach

2.7. Comprobación y cambio de tipos de objetos

de la variable coincide con una de las columnas de la hoja de datos se almacena allí la transformación, en caso contrario se añade una columna con el nombre de variable indicado.

Los siguientes ejemplos son ilustrativos de las situaciones comentadas.

```
> Hoja3
  Peso Altura Sexo  Nombre
1   90   1.85    H    Pepe
2   87   1.87    H    Juan
3   60   1.63    M Antonia
> subset(Hoja3,select=c(Sexo,Nombre))
  Sexo  Nombre
1    H    Pepe
2    H    Juan
3    M Antonia
> subset(Hoja3,subset=(Sexo=="H"))
  Peso Altura Sexo  Nombre
1   90   1.85    H    Pepe
2   87   1.87    H    Juan
> transform(Hoja3,Peso=log(Peso))
  Peso Altura Sexo  Nombre
1 4.499810   1.85    H    Pepe
2 4.465908   1.87    H    Juan
3 4.094345   1.63    M Antonia
> transform(Hoja3,LogPeso=log(Peso))
  Peso Altura Sexo  Nombre  LogPeso
1   90   1.85    H    Pepe 4.499810
2   87   1.87    H    Juan 4.465908
3   60   1.63    M Antonia 4.094345
```

2.7. Comprobación y cambio de tipos de objetos

Hay una serie de funciones que permiten comprobar si un objeto es de un tipo determinado, todas comienzan por **is.**, o cambiarlo a un tipo concreto, que comienzan por **as.**. Así podrá (en un programa, claro está) querer dilucidar si un objeto es un vector o no. Puede hacerlo mediante la función **is.vector** que, aplicada al objeto, indica si es o no un vector, esto es, si es cierto (T) o falso (F) que es un vector. Del mismo modo puede comprobar si un número es entero, si es un valor lógico, etc.

Si necesita que el objeto sea de un tipo concreto puede transformarlo (si se puede) a ese tipo. Así, un vector puede transformarse en una hoja de datos mediante la función **as.data.frame**.

Utilizando la ayuda puede encontrar todas las funciones que comienzan con las palabras **is.** y **as.**, utilizando la función **apropos**. Así, las siguientes:

```
is.R is.array is.atomic is.call is.character is.complex
is.data.frame is.double is.element is.empty.model
is.environment is.expression is.factor is.finite
is.function is.infinite is.integer is.language is.list
is.loaded is.logical is.matrix is.na is.na.data.frame
```

2.7. Comprobación y cambio de tipos de objetos

is.name is.nan is.null is.numeric is.object is.ordered
is.pairlist is.qr is.real is.recursive is.single is.symbol is.ts is.vector

La función `str` muestra de forma compacta la estructura interna de un objeto.

Ejemplo

```
> x<-1:5
> str(x)
  int [1:5] 1 2 3 4 5
> is.vector(x)
[1] T
> is.data.frame(x)
[1] F
> x<-as.data.frame(x)
> is.data.frame(x)
[1] T
> str(x)
'data.frame':  5 obs. of  1 variable:
 $ x: int  1 2 3 4 5
```

Capítulo 3

Funciones

Como ya se indicó anteriormente, las funciones permiten realizar las diferentes acciones. Existen muchas funciones ya definidas, algunas incluso pueden ser modificadas, incluso accidentalmente¹, pero una de las capacidades más interesante es la posibilidad de crear nuevas funciones que realicen tareas que no estaban definidas en el momento de instalar el programa. Estas nuevas funciones se incorporan al lenguaje y, desde ese momento, se utilizan como las previamente existentes. De este modo, cualquier persona que escribe una función puede difundirla y puede ser incorporada por otras personas. La lectura de la definición de las diferentes funciones incorporadas en origen es una tarea muy formativa, ya que están escritas por especialistas en el lenguaje y, además, sufren un proceso de depuración a través de las quejas de los usuarios, por lo que se puede decir que, en general, están muy bien escritas. Para leer la definición, como ya sabe, basta con escribir el nombre de la función sin paréntesis. Por ejemplo, la definición de la función que realiza el cálculo de la varianza, se obtiene escribiendo `var`. Antes de estudiar la definición conviene obtener ayuda sobre qué es lo que hace la función, mediante la función `help`. Si necesita ejecutar un programa del sistema operativo puede utilizar la función `system`, que permite incluso salir al sistema operativo con la orden `system("cmd")`. También puede utilizar la orden `shell`, como por ejemplo en `shell("dir")`.

Las funciones tradicionales en cualquier lenguaje de uso matemático y estadístico, están definidas. Entre ellas se encuentran, entre otras, `abs`, `sqrt`, `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `exp`, `log`, `log10`, `min`, `max`, `sum`, `prod`, `length`, `mean`, `range`, `median`, `var`, `cov`, `summary`, `sort`, `rev`, `order`; cuyas definiciones abreviadas puede ver en el apéndice y ampliadas mediante la orden `help` en el propio programa, y que puede utilizar inmediatamente.

```
> sin(1:10)
[1] 0.8414710 0.9092974 0.1411200 -0.7568025 -0.9589243
[8] -0.2794155 0.6569866 0.9893582 0.4121185 -0.5440211
> sin(pi)
[1] 1.224606e-16
> cos(pi)
[1] -1
> sum(1:10)
```

¹Ver la sección 3.8 correspondiente a la función `search`

```
[1] 55
> prod(1:10)
[1] 3628800
```

3.1. Definición de una función

Una función se define asignando a un objeto la palabra **function** seguida de los argumentos que desee dar a la función, escritos entre paréntesis y separados por comas, seguida de la orden, entre llaves si son varias órdenes, que desee asignar a la misma. Entre los argumentos destaca que si utiliza tres puntos seguidos, ..., ello indica que el número de argumentos es arbitrario.

Por ejemplo, la siguiente instrucción, define una función, llamada **funcionsuma**, que calcula la suma de dos números que se le pasan como argumentos, aunque si alguno de ellos no se indica se le asigna un valor predeterminado. Esta suma se devuelve como resultado de la función. Advierta que el objeto aparece en la lista de objetos si se utiliza la función **ls**, y que su definición puede verse como la de cualquier otra función. Antes de asignar una función a un objeto, como **funcionsuma**, es conveniente intentar obtener su definición, por si ya existiese, ya que en caso afirmativo la nueva definición sustituiría a la antigua, puede que incluso irreparablemente.

```
> funcionsuma
Error: Object "funcionsuma" not found
> funcionsuma<-function(A=1,B=2) A+B
> funcionsuma()
[1] 3
> funcionsuma(5,7)
[1] 12
> funcionsuma
function(A = 1, B = 2)
A + B
```

Debemos destacar, en primer lugar, que la función que acabamos de definir no se refiere a la suma de dos números, sino a la suma de dos objetos, que podrán ser vectores, matrices, variables indexadas, etc.

Las funciones que tienen prevista la actuación de modo diferente según el tipo o el número de argumentos que se les suministre se denominan **genéricas** y son de importancia fundamental.

```
> funcionsuma(1:9,2:4)
[1] 3 5 7 6 8 10 9 11 13
```

En segundo lugar, la función lo que realiza es la orden que tiene a continuación (una sola) por lo que para escribir varias órdenes deberán agruparse entre llaves, {}, y el valor que devuelve la función es el correspondiente a la última que ejecuta.

Compruebe los siguientes ejemplos, variaciones sobre este ejemplo elemental:

```
> f1 = function(A=1,B=2)
+ {
```

3.1. Definición de una función

```
+ #Devuelve A+B
+ 46
+ 25
+ A+B
+ }
> f2 = function(A=1,B=2)
+ {
+   #Devuelve 25
+   A+B
+   46
+   25
+ }
> f3 = function(A=1,B=2)
+ {
+   #Devuelve B, porque A termina la línea por sí misma y +B es B
+   A
+   +
+   B
+ }
> f4 = function(A=1,B=2)
+ {
+   #Devuelve en una lista A+B y (A+B)^2
+   list(Suma= A+B,Cuadrado= (A+B)^2)
+ }
> f5 = function(A=1,B=2)
+ {
+   #Devuelve A+B
+   A-B
+   A+B
+ }
```

Si desea poner de manifiesto el valor devuelto, puede utilizar la función **return** que termina la función y devuelve su argumento. Cuando se encuentra esta función se detiene la ejecución y se devuelven los valores indicados, por lo que es muy útil en programación cuando existen bifurcaciones.

```
> f1 = function(A=1,B=2)
+ {
+   #Devuelve A+B
+   return(A+B)
+ }
```

Es posible acceder a los argumentos de una función mediante la función **formals** y al cuerpo de la misma mediante la función **body**.

```
> f1
function (A = 1, B = 2)
{
  A + B
}
> #Almacena los valores actuales en argumentos
```

3.2. Algunas funciones elementales

```
> formals(f1) -> argumentos
> #Modifica los valores
> formals(f1)=alist(X=,Y=-1)
> f1
function (X, Y = -1)
{
    A + B
}
> #Recupera los valores almacenados
> formals(f1) = argumentos
> f1
function (A = 1, B = 2)
{
    A + B
}
> #Almacena los valores actuales en cuerpo
> body(f1) -> cuerpo
> #Modifica los valores
> body(f1) <- expression(A*B)
> f1
function (A = 1, B = 2)
A * B
> #Recupera los valores almacenados
> body(f1) = cuerpo
> f1
function (A = 1, B = 2)
{
    A + B
}
>
```

3.1.1. Metodología de definición de una función

Para definir una función, por ejemplo f , es aconsejable realizar los siguientes pasos:

1. Escriba f . Confirme que aparece el mensaje de que no existe esta función
2. Cree la función vacía con `f=function(){}`
3. Edite la función con la orden `f=edit(f)` y guarde sin hacer ninguna modificación
4. Utilice la orden `f=edit(f)` tantas veces como sea necesario guardando las modificaciones, hasta obtener la función final deseada

3.2. Algunas funciones elementales

A continuación aparecen algunas funciones elementales, para comprender los rudimentos de la programación. Estas funciones están construidas de modo lineal, esto es, sin bifurcaciones, y utilizando solamente aspectos elementales.

Función media

En primer lugar, se incluye una definición alternativa a la definición de media incluida en R. Es una función de un solo argumento, que si no se especifica será sustituido por el valor **NA**, con lo que, en ese caso, el resultado sería **NA**. En caso de que se le suministre un argumento, no se comprueba si es válido o no, sino que suponiendo que es un vector, se eliminan del mismo los elementos correspondientes a **NA**. Para ello se utiliza la negación **!** y la condición de ser un valor no disponible. Del resto se calcula la media mediante la definición trivial, $\sum x/n$.

```
media<- function(x=NA)
{
  x<-x[!is.na(x)]
  return(sum(x)/length(x))
}


> media(c(2,3,7))
[1] 4
> media(c(2,3,7,NA))
[1] 4
```

Advierta que el objeto **x** utilizado dentro del cuerpo de la función es temporal² y desaparece al terminar de ejecutarse la misma. Además no coincide con ningún objeto del mismo nombre y ya existente³ por lo que no debe preocuparse de si podrá interferir con objetos preexistentes.



Ejercicio. Modifique la función para que devuelva también cuantos elementos había y cuantos ha quitado.

Función varianza

Si quiere calcular la varianza de un vector, puede utilizar la *perversa* fórmula derivada de la relación entre los momentos, $\mu_2 = m_2 - m_1^2$, para definir, de modo análogo al anterior, y usando la función **media** ya existente, la varianza. 

```
> koning<- function(x=NA)
{
  return(media(x^2)-(media(x))^2)
}

> koning(c(2,3,7))
[1] 4.666667
> koning(c(2,3,7,NA))
[1] 4.666667
```

Es preferible utilizar directamente la definición de la varianza para obtener la siguiente función:

```
> varianza<- function(x=NA)
{
```

²En Visual Basic corresponde a un argumento pasado **byval**

³Se crea en un entorno correspondiente a la función, es como si el objeto **o** dentro de la función **f** fuese **f.o**

3.3. Funciones `.First` y `.Last`

```
      y<-media(x)
      return(media((x-y)^2))
}
> varianza(c(2,3,7))
[1] 4.666667
> varianza(c(2,3,7,NA))
[1] 4.666667
```

que suministra, en los ejemplos considerados, los mismos resultados, pero es más estable en otros casos.

Por supuesto, la escritura de la función puede hacerse más o menos compacta y de modo equivalente:

```
> varianza2<- function(x=NA) media((x-media(x))^2)
> varianza2(c(2,3,7))
[1] 4.666667
```

Ejercicio. Escriba una función que calcule los coeficientes de asimetría y curtosis de Pearson. Sus definiciones son:

$$\gamma_1 = \mu_3/\sigma^3 \quad \gamma_2 = \mu_4/\sigma^4$$

3.3. Funciones `.First` y `.Last`

Estas funciones se ejecutan, si existen, al iniciar una sesión de trabajo y al terminarla, respectivamente. Puede utilizarlas para personalizar su método de trabajo. Los objetos cuyo nombre comienza con un punto, como es el caso de estas dos funciones, no se muestran al utilizar la orden `ls` salvo que se incluya el parámetro `all.names=T`

En R el mecanismo de inicio, a grandes rasgos, es el siguiente: Si existe el archivo **.Renviron** se utiliza como un conjunto de órdenes que se ejecutan, salvo que la orden de ejecución incluya la opción **—no-environ**. A continuación comprueba si existe el archivo **.Rprofile** y también lo ejecuta. A continuación carga el archivo **.RData** salvo que se especifique la opción **—no-restore**. A continuación, si existe la función **.First**, se ejecuta.

Así, la siguiente definición de la función **.First**, que utiliza la función `cat` que escribe un texto en pantalla, obliga a que, al inicio de cada sesión de trabajo, se escriba el mensaje **Hola.Bienvenido a R** en pantalla.

```
.First<-
function()
{
  cat("Hola.Bienvenido a R\n")
}
```

3.4. Algunos elementos útiles en programación

La programación sin posibilidad de estructuras de programación es muy limitada. No es este el caso, sino que existen gran cantidad de estas estructuras. A continuación encontrará un breve repaso de las mismas.

3.4.1. Operadores de relación

Con **!** se indica la negación, con **&** la conjunción y con **|** la disyunción. Estos dos últimos, si se escriben repetidos tienen el mismo significado, pero se evalúa primero la parte de la izquierda y, si ya se sabe el resultado (suponiendo que se pudiera calcular la expresión de la derecha) no se sigue evaluando, por lo que pueden ser más rápidos y eliminar errores. **<, >, <=, >=, ==** son respectivamente los símbolos de menor, mayor, menor o igual, mayor o igual, e igual. Advierta que este último se escribe con dos signos de igualdad.

3.4.2. Estructuras condicionales

Son aquellas que, según el resultado de una comparación, realizan una u otra acción. Recuerde que un conjunto de acciones, entre llaves, se interpretan como una sola acción desde el punto de vista lógico. La primera estructura es

if (condición) acción1 [else acción2]

Esta estructura es **escalar**, ya que si la condición está formada por más de un elemento, sólo considera el primero⁴. Por ello en el ejemplo siguiente se incluye la orden **min(x)**, ya que si se aplicase directamente a **x** sólo consideraría la primera componente del vector de comparación.

Ejemplo

```
> raiz<-function(x)
{
  if (is.numeric(x) && min(x)>0)
    # Este símbolo hace que el resto de la línea
    # se considere un comentario
    # Advierta que no se intenta aplicar
    # la función min si el argumento
    # no es numérico, previniendo un error
    {sqrt(x)}
    else {stop("0 x no es numerico o no es positivo")}
}

> raiz("Pepe")
Error in raiz("Pepe"): 0 x no es numerico o no es positivo
> raiz(-1)
Error in raiz(-1): 0 x no es numerico o no es positivo
> raiz(7)
[1] 2.645751
```

Observe que **acción1** y **acción2** son sendas acciones individuales. Si desea que se realicen varias acciones debe incluirlas entre llaves, para que aparezcan como una sola desde el punto de vista lógico. En el ejemplo se han incluido llaves pero podrían eliminarse.

La segunda estructura es

⁴Y además presentaría un mensaje de advertencia

3.4. Algunos elementos útiles en programación

ifelse(condición, acción en caso cierto, acción en caso falso)

Esta estructura es **vectorial**, ya que si la condición está formada por más de un elemento, considera cada uno de ellos.

Así, por ejemplo, suponiendo que el argumento **x** es numérico, tendríamos:

```
> Inverso<-function(x) ifelse(x==0,NA,1/x)
> Inverso(-1:1)
[1] -1 NA 1
> Inverso(-2:2)
[1] -0.5 -1.0 NA 1.0 0.5
```

Por último, la tercera estructura es

switch(expresión,[valor-1=]acción-1,...,[valor-n=]acción-n)

Esta función es **escalar** y además **expresión** debe devolver un vector de longitud 1. La función se comporta de modo distinto según que la expresión evalúe a numérico o a carácter. Cuando la expresión devuelve un valor numérico, se transforma el resultado en un entero, **i**, y si está entre 1 y **n**, se evalúa la acción **i**, tenga o no un valor asociado, y se devuelve el resultado, en caso contrario se devuelve NULL. Cuando la expresión devuelve una cadena de caracteres, si este coincide exactamente con uno de los valores, se evalúa la acción correspondiente y se devuelve el resultado. Si no coincide con ninguno y una acción (normalmente la última se reserva para ello) no tiene un valor asociado, se ejecuta esta y se devuelve su resultado; pero si todas tienen valor asociado, se devuelve NULL. Si hay varias acciones sin valor asociado se devuelve la primera de ellas.

Aunque en el ejemplo no se hace para no distraer la atención, hay que comprobar que el argumento es del tipo que se desea. Si en este ejemplo utilizamos **Decide(1)** obtendríamos el mismo resultado que con **Decide("m")**. Una solución intermedia consiste en sustituir **x** por **as.character(x)** como argumento en la función **switch**.

```
> n<-2
> switch(n,"Uno","Dos","Tres")
[1] "Dos"
> n<-3
> switch(n,"Uno","Dos","Tres")
[1] "Tres"
> n<-4
> switch(n,"Uno","Dos","Tres")
NULL
> Decide<-function(x)
switch(x,m=cat("Has dicho eme minúscula\n"),
M=cat("Has dicho eme mayúscula\n"),cat("Dime m o M\n"))
> Decide
function(x)
switch(x,
      m = cat("Has dicho eme minúscula\n"),
      M = cat("Has dicho eme mayúscula\n"),
      cat("Dime m o M\n"))
> Decide("m")
Has dicho eme minúscula
```

3.4. Algunos elementos útiles en programación

```
> Decide("M")
Has dicho eme mayúscula
> Decide("P")
Dime m o M
> n<-"Dos"
> switch(n,"Uno"]=1,"Dos"]=2,"Tres"]=3,"Distinto")
[1] 2
> n<-"Cuatro"
> switch(n,"Uno"]=1,"Dos"]=2,"Tres"]=3,"Distinto")
[1] "Distinto"
```

3.4.3. Estructuras de repetición definida e indefinida

Hay tres estructuras, que son:

for (variable in valores) acción

while (condición) acción

repeat acción

La estructura **for** asigna a **variable** cada uno de los **valores** y realiza la **acción** para cada uno.

La estructura **while** evalúa la **condición** y mientras esta es cierta se evalúa la **acción**.

La estructura **repeat** evalúa la **acción** indefinidamente.

En los tres casos, el valor del ciclo completo es el de la última evaluación de la acción.

Las expresiones pueden contener algún condicional como **if** asociado con las funciones **next** o **break**.

La estructura **next** indica que debe terminarse la iteración actual y pasar a la siguiente.

La estructura **break** indica que debe terminarse el ciclo actual.

Los tres ejemplos siguientes muestran una estructura **for** que recorre la sucesión **-5:5**. En el primer caso la recorre completa, en el segundo, salta el número 0, y en la tercera, se detiene al llegar a cero. En los tres casos, si a continuación hubiera otra orden seguiría ejecutándola a continuación.

```
> for (i in -5:5) {cat(i,"\t",i^2,"\n")}
-5      25
-4      16
-3       9
-2       4
-1       1
0        0
1        1
2        4
3        9
4       16
5       25
> for (i in -5:5) {if (i==0) next;cat(i,"\t",i^2,"\n")}
-5      25
-4      16
-3       9
-2       4
-1       1
1        1
2        4
3        9
4       16
5       25
```

3.4. Algunos elementos útiles en programación

```
-5      25
-4      16
-3       9
-2       4
-1       1
1        1
2        4
3        9
4       16
5       25
> for (i in -5:5) {if (i==0) break;cat(i,"\t",i^2,"\n")}
```

Por último, las dos estructuras siguientes recorren la sucesión **5:1** mediante **while** y **repeat**. Téngase en cuenta que están escritas sólo a modo de ilustración, si se desea recorrer esas secuencias es preferible utilizar un ciclo **for**.

En esta se añade la acción (**i=i-1**) que modifica la condición (**i>0**).

```
> i<-5;while (i>0) {print(i);i=i-1}
> [1] 5
[1] 4
[1] 3
[1] 2
[1] 1
```

En esta se añade la comprobación (**i==0**) de haber alcanzado el final.

```
> i<-5;repeat{print(i);i=i-1;if(i==0) break}
> [1] 5
[1] 4
[1] 3
[1] 2
[1] 1
```

3.4.4. invisible

La función **invisible** indica que un objeto no debe mostrarse. La sintaxis es

```
invisible(x)
```

siendo **x** cualquier objeto que es devuelto a su vez por la función. Esta función se usa muy a menudo para no presentar directamente informaciones devueltas por funciones que, sin embargo, pueden ser utilizadas.

Cualquier acción en R devuelve un resultado. Incluso cuando se realiza una asignación se devuelve un resultado, aunque se hace de modo invisible. Para hacerlo visible basta con utilizar paréntesis (¡que son una función!). Sin embargo, una agrupación de órdenes entre llaves (¡que no es una función!) no tiene este

3.4. Algunos elementos útiles en programación

efecto. Vea los siguientes ejemplos, especialmente el último, en que se asigna 3 a la variable **y**, lo que devuelve de modo invisible este valor, que se asigna a continuación a **x**.

```
> x<-1
> (x<-1)
[1] 1
> {x<-1}
> {x<-1;y<-2;z<-3}
> ({x<-1;y<-2;z<-3})
[1] 3
> x<-y<-3
> x
[1] 3
> y
[1] 3
```

3.4.5. Recursividad

Es posible que la definición de una función haga referencia a la propia función, lo que permite construir funciones que desarrollan estructuras definidas fácilmente mediante recursividad y muy complejamente por otros medios; como por ejemplo los fractales. Sin embargo debe recordar que este tipo de estructura consume muchos recursos del sistema, por lo que debe reservarse para los casos en que sea estrictamente necesaria. A continuación, cuando lea la definición de factorial de un número, encontrará una definición utilizando recursividad, aunque repito que no es la mejor solución, ya que en ese caso es preferible una estructura de repetición.

3.4.6. Depuración de errores

En caso de error en la utilización de una función es de gran utilidad la función **traceback**. Por ejemplo, con la función **Dibuja** que está definida posteriormente, la función **traceback** permite reconstruir el camino seguido por R hasta devolver el error.

```
> Dibuja()
Error in plot.window(xlim, ylim, log, asp, ...) :
  invalid xlim
> traceback()
[1] "plot.window(xlim, ylim, log, asp, ...)"
[2] "plot.default(0, 0, xlim = xlim, ylim = ylim,"
[3] "type = \"n\", xaxs = xaxs, "
[4] "      yaxs = yaxs, xlab = xlab, ylab = ylab, ...)"
[5] "plot(0, 0, xlim = xlim, ylim = ylim,"
[6] "type = \"n\", xaxs = xaxs, "
[7] "      yaxs = yaxs, xlab = xlab, ylab = ylab, ...)"
[8] "image(x, y, outer(x, y, f))"
[9] "Dibuja()"
```

3.5. Ejemplos de funciones

De gran utilidad son las funciones `debug` y `undebug`. La primera de ellas, aplicada a una función determinada, hace que esta se realice paso a paso, pudiendo incluso observar los valores de las variables que en ella intervienen en cada momento. Hasta aplicarle la segunda función, `undebug`, permanecerá en este estado.

```
> debug(raiz)
> raiz(3)
debugging in: raiz(3)
debug: {
  if (is.numeric(x) && min(x) > 0)
    sqrt(x)
  else stop("0 x no es numerico o no es positivo")
}
Browse[1]>
debug: if (is.numeric(x) && min(x) > 0) sqrt(x)
else stop("0 x no es numerico o no es positivo")
Browse[1]>
exiting from: raiz(3)
[1] 1.732051
> undebug(raiz)
```

Advierta que la función `traceback` devuelve el proceso que generó el último error, por lo que no tiene sentido aplicarla cuando este no existe.

```
> raiz(3)
[1] 1.732051
> traceback()
[1] "plot.window(xlim, ylim, log, asp, ...)"
[2] "plot.default(0, 0, xlim = xlim, ylim = ylim,"
[3] "type = \"n\", xaxs = xaxs, "
[4] "    yaxs = yaxs, xlab = xlab, ylab = ylab, ...)"
[5] "plot(0, 0, xlim = xlim, ylim = ylim,"
[6] "type = \"n\", xaxs = xaxs, "
[7] "    yaxs = yaxs, xlab = xlab, ylab = ylab, ...)"
[8] "image(x, y, outer(x, y, f))"
[9] "Dibuja()"
```

3.5. Ejemplos de funciones

A continuación se presentan algunas funciones para dejar claro el procedimiento de escritura de las mismas. Estas funciones sufrirán ciertas depuraciones, por lo que aparecen varias versiones de algunas.

3.5.1. Factorial de un número

A continuación, se define el factorial⁵ de un número (natural⁶) utilizando tanto un ciclo determinado como uno indeterminado

```
> Factorial.d<-function(n)
{
#####
# Valor inicial de factorial el neutro del producto  #
# Estructura for en funcion de n positivo            #
# Se devuelve factorial                             #
#####
    factorial <- 1
    if(n>1)
        for (i in 1:n)
            factorial <- factorial * i
    return(factorial)
}
> Factorial.d(3)
[1] 6
> Factorial.d(100)
[1] 9.332622e+157
> Factorial.d(0)
[1] 1

> Factorial.i<-function(n)
{
#####
# Valor inicial de factorial el neutro del producto  #
# Estructura while en funcion de n positivo          #
# Se añade el factor n y se disminuye el valor de n #
# Se devuelve factorial                             #
#####
    factorial <- 1
    while(n > 0)
    {
        factorial <- factorial * n
        n <- n - 1
    }
    return(factorial)
}
> Factorial.i(3)
[1] 6
> Factorial.i(100)
[1] 9.332622e+157
> Factorial.i(0)
[1] 1
```

⁵De hecho, existe la función `factorial(x)`, definida como `gamma(x+1)`. Aquí presentamos un ejercicio.

⁶No se ha incluido comprobación de que el argumento es correcto, `is.integer`, por lo que la función puede terminar con error si el argumento es de tipo carácter, por ejemplo.

3.5. Ejemplos de funciones

A continuación se presentan dos procedimientos alternativos que son, en este caso, peores que los anteriores: El primero consume recursos debido a la recursividad y el segundo los consume porque genera todos los factores antes de multiplicarlos, consumiendo memoria.

```
> Factorial.r<-function(n)
{
  #####
  # La función se llama a sí misma modificando el argumento #
  #####
  if(n > 1)
    factorial <- n * Factorial.r(n - 1)
  else factorial <- 1
  return(factorial)
}
> Factorial.r(3)
[1] 6
> F.r(300)
[1] Inf
> Factorial.r(1000)
Error: evaluation nested too deeply:
infinite recursion / options(expression=)?
```

y observe que no puede calcularse el factorial de 1000 porque no hay bastantes recursos en el modo predeterminado.

```
> Factorial.m<-function(n)
{
  if(n > 1)
    return(prod(n:1))
  else return(1)
}
> Factorial.m(3)
[1] 6
> Factorial.m(100)
[1] 9.332622e+157
```

En este último caso no se llegan a agotar los recursos, pero se han consumido más que en los dos primeros, ya que es necesario generar el vector **n:1** y almacenarlo antes de calcular el producto de sus elementos, aunque es más eficiente desde el punto de vista de tiempo de cálculo. Advierta que, además, en unos casos se utilizan los productos desde 1 a n y en otros al contrario, lo que numéricamente no es equivalente.

3.5.2. Progresión aritmética

La progresión aritmética es una sucesión definida recurrentemente a partir del primer elemento, a_1 , y la diferencia, d , mediante la relación $a_{n+1} = a_n + d$. Se dispone además de un resolvente explícito, $a_{n+1} = a_1 + d * n$.

Construimos tres funciones para implementarla: **Ar.e**, **Ar.r** y **Ar.r.v**. La primera corresponde a la forma explícita, la segunda a la forma recursiva y la tercera a la forma recursiva vectorial.

3.5. Ejemplos de funciones

```
> Ar.e = function(n=1,a1=1,d=1) a1+d*(n-1)
> Ar.r = function(n=1,a1=1,d=1)
{
  if (n>1)
    {return(Ar.r(n-1,a1,d)+d)}
  else
    {return(a1)}
}
> Ar.r.v = function(n=1,a1=1,d=1)
{
  A=1:n
  A[1]=a1
  for (i in 2:n) A[i]=A[i-1]+d
  return(A[n])
}
```

La que he denominado *forma recursiva vectorial*, consiste en la construcción de un objeto que represente la estructura que queremos desarrollar, en este caso un vector (de ahí el adjetivo de vectorial) representa perfectamente a los n primeros términos de una sucesión, y construir todos los elementos de la estructura comenzando desde los primeros hasta llegar al final. Puede comprobar que, por ejemplo, obtener un millón de elementos en este caso es factible y se realiza bastante rápidamente, en tanto que obtener simplemente 1000 términos en el caso recursivo no es factible.

Es posible incluir un medidor de tiempo y tener en cuenta que, en el caso en que n vale 1, la anterior definición falla⁷. Definimos la función del siguiente modo:

```
> Ar.r.v = function(n=1,a1=1,d=1)
{
  tiempoinicial=Sys.time()
  if (n==1)
  {
    tiempofinal=Sys.time()
    return(list(a1,tiempofinal-tiempoinicial))
  }
  A=1:n
  A[1]=a1
  for (i in 2:n) A[i]=A[i-1]+d
  tiempofinal=Sys.time()
  return(list(A[n],tiempofinal-tiempoinicial))
}
> Ar.r.v(1000000)
[[1]]
[1] 1e+06

[[2]]
Time difference of 10.313 secs
```

⁷Pruebe más adelante con la función debug a descubrir dónde y por qué falla.

3.5.3. Progresión geométrica

La progresión geométrica es una sucesión definida recurrentemente a partir del primer elemento, a_1 , y la razón, r , mediante la relación $a_{n+1} = a_n * r$. Se dispone además de un resolvente explícito, $a_{n+1} = a_1 * r^n$.

Podemos construir tres funciones para implementarla: `Ge.e`, `Ge.r` y `Ge.r.v`. La primera corresponde a la forma explícita, la segunda a la forma recursiva y la tercera a la forma recursiva vectorial.


```
> Ge.e = function(n=1,a1=1,r=1) a1*r^(n-1)
> Ge.r = function(n=1,a1=1,r=1)
{
  if (n>1)
    {Ge.r(n-1,a1,r)*r}
  else
    {a1}
}
> Ge.r.v = function(n=1,a1=1,r=1)
{
  if (n==1) return(a1)

  A=1:n
  A[1]=a1
  for (i in 2:n) A[i]=A[i-1]*r
  return(A[n])
}
```

3.5.4. Sucesión de Fibonacci

Estas funciones devuelven el elemento enésimo de una sucesión de **Fibonacci**, caracterizada porque los dos primeros elementos valen 1 y el resto de los elementos se calculan como la suma de los dos que le preceden. Como no tenemos una forma explícita, utilizamos sólo dos procedimientos: El primero, recursivo puro, y el segundo recursivo con almacenamiento de los resultados en un vector.

```
> Fibonacci.r = function(n=1) if (n>2)
  {Fibonacci.r(n-1)+Fibonacci.r(n-2)} else {1}

> Fibonacci.r.v=function(n=1)
{
  if (n<3) return(1)
  x=vector()
  x[1]=1
  x[2]=1
  for (i in 3:n) x[i]=x[i-1]+x[i-2]
  return(x[n])
}
```

La forma recurrente pura consume muchísimos más recursos y tiempo que la recurrente vectorial, como ya se ha indicado anteriormente (Pruébela con $n=30$ y con $n=300$, añadiéndole un medidor de tiempo).

3.5. Ejemplos de funciones

En la definición de **Fibonacci.r.v** se ha aprovechado que los dos primeros valores son iguales para finalizar los cálculos al comprobar que n es menor que 3. Ello se hace mediante la función **return** que devuelve los argumentos y finaliza la ejecución. La siguiente función utiliza una expresión más compleja pero más apropiada para la construcción de funciones similares en que los primeros elementos no son idénticos.

```
> Fibonacci.vv = function(n=1,a1=1,a2=1)
{
  if (n==1) return(a1)
  if (n==2) return(a2)
  x=vector(1=n)
  x[1]=a1
  x[2]=a2
  for (i in 3:n) x[i]=x[i-1]+x[i-2]
  return(x[n])
}
```

También puede escribir la función utilizando **switch** en vez de dos **if**.

```
> Fibonacci.vv.c = function(n=1,a1=1,a2=1)
{
  N=as.character(n)
  # Se convierte en un carácter para que
  # la tercera opción haga el papel de else
  switch(N,
    "1"=return(a1),
    "2"=return(a2),
    {
      x=vector(1=n)
      x[1]=a1
      x[2]=a2
      for (i in 3:n) x[i]=x[i-1]+x[i-2]
      return(x[n])
    }
  )
}
```

Por último se presenta una modificación de la función en que se devuelve una lista con tres elementos: El elemento n -ésimo de la sucesión, todos los elementos de la sucesión hasta el n -ésimo y la suma de los mismos.

```
> Fibonacci.vv.l = function(n=1,a1=1,a2=1)
{
  N=as.character(n)
  # Se convierte en un carácter para que
  # la tercera opción haga el papel de else
  switch(N,
    "1"=return(list(xn=a1,v=a1,suma=a1)),
    "2"=return(list(xn=a2,c(a1,a2),suma=a1+a2)),
    # en cualquier otro caso
```

3.5. Ejemplos de funciones

```
{
  x=vector(l=n)
  x[1]=a1
  x[2]=a2
  for (i in 3:n) x[i]=x[i-1]+x[i-2]
  return(list(xn=x[n],v=x,suma=sum(x)))
}
)

> Fibonacci.vv.l(5)
$xn
[1] 5

$v
[1] 1 1 2 3 5

$suma
[1] 12
> Fibonacci.vv.l(15)$xn
[1] 610
> Fibonacci.vv.l(25)$suma
[1] 196417
```



3.5.5. Simulación de una extracción de cartas de una baraja

En este ejemplo se realiza una simulación del siguiente experimento: Realizar un muestreo con reemplazamiento de una baraja de 52 cartas, hasta obtener los cuatro ases e indicar el número de extracciones necesarias. Se utiliza la función `sample` para realizar cada extracción. Utilice `help(sample)` para estudiar la función.

```
> CuatroAses
function(Mostrar = F,Maximo=1000)
{
#####
# Pone los cuatro ases en 0                                     #
# Saca una carta de la baraja (52)                             #
# Suma uno al contador                                         #
# Si no es un as (1) vuelve al principio del ciclo            #
# El as correspondiente se pone en 1                           #
# Si los cuatro ases son 1 termina el ciclo                    #
#####
  Extracciones = 0
  Resultado = 1:Maximo
  Ases = c(0,0,0,0)
  repeat
  {
    if (Maximo <= Extracciones )
    {
```

3.5. Ejemplos de funciones

```
    if(Mostrar)
    {
        cat("No he podido obtener cuatro ases en ",
            Extracciones,
            "extracciones. \n")
    }
    return(list(E = NA, R = Resultado, Conseguido=F))
}

Extracciones = Extracciones + 1
SacoUna = sample(52, 1)
Resultado[Extracciones] = SacoUna
if(SacoUna %% 13 != 1) next
# %% es el módulo
Ases[(SacoUna -1) %% 13 + 1] = 1
# %% es la división entera
if(sum(Ases)==4)
break
}
length(Resultado)=Extracciones
if(Mostrar)
{
    cat("He necesitado", Extracciones,
        "extracciones para obtener cuatro ases.\n")
}
return(list(E = Extracciones, R = Resultado, Conseguido=T))
}
> CuatroAses(,10)
$E
[1] 10

$R
[1] 47 31 48 42 30 39 31 21 22 34

$Conseguido
[1] FALSE

> CuatroAses(T,10)
No he podido obtener cuatro ases en 11 extracciones.
$E
[1] NA

$R
[1] 24 35 41 24 10 14 39 13 48 42

$Conseguido
[1] FALSE

> CuatroAses()
$E
[1] 205
```

3.5. Ejemplos de funciones

```
$R
[1] 44 33 35 36 42 ....
```

```
$Conseguido
[1] TRUE
```

```
> CuatroAses(T)
He necesitado 61 extracciones para obtener cuatro ases.
$E
[1] 61
```

```
$R
[1] 3 19 2 40 15 47 ....
```

```
$Conseguido
[1] TRUE
```

```
>
```

Entre las alternativas a la escritura de esta función, debe destacarse que es posible obtener la muestra de tamaño **Maximo** de una sola vez con la orden **sample(52,Maximo,T)** e ir explorando posteriormente los valores.



Por otra parte, si se asignara a los ases los números 1, 2, 3 y 4 sería más sencillo realizar la programación.

Advierta también que cada vez que ejecuta la función obtiene el resultado de una simulación basada en números pseudoaleatorios, por lo que no es esperable obtener los mismos resultados en dos ejecuciones sucesivas. Se puede utilizar el resultado de la función como entrada de otra función, por ejemplo para repetir varias veces la simulación y estudiar la distribución de la misma.

```
> DistriAses = function(n = 5,Maximo=1000)
{
  Saco = vector(length=n)
  for(i in 1:n)
    Saco[i] = CuatroAses(F,Maximo)$E
  Saco
}
> Distribucion
[1] 174 89 76 259 140 98 99 ...
[487] 123 72 83 159 120 22 67 109 84 81 80 76 118 68
> summary(Distribucion)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  15.00  63.75   96.00  106.40  134.00  361.00
> hist(Distribucion)
> DistriAses(5,50)
[1] 33 NA NA 29 NA
```

Si desea escribir una función que realice una simulación de la extracción de cartas, pero sin reemplazamiento, deberá generar en primer lugar la muestra

3.5. Ejemplos de funciones

completa de la extracción y comprobar posteriormente cuándo ocurre el resultado que espera.

```
> CuatroAses.Sin = function(Mostrar = F)
{
#####
# Pone el contador de ases en 0                                     #
# Obtiene una permutación de las cartas de la baraja               #
# Si no es un as (1) pasa a la siguiente carta                     #
# Suma 1 al contador de ases                                       #
# Si hay cuatro ases termina el ciclo                               #
#####
  Ases = 0
  Resultado = sample(52)
  for (i in 1:52)
  {
    if(Resultado[i] %% 13 != 1) {next}
    # %% es el módulo
    Ases = Ases+1
    if(Ases==4) {break}
  }
  if(Mostrar)
  {
    cat("He necesitado", i,
        "extracciones para obtener cuatro ases\n")
  }
  return(list(E = i,R = Resultado[1:i]))
}

> CuatroAses.Sin()
$E
[1] 16

$R
[1] 29 51 20 21  1 26 40 42  6 12  5 43 15 14  9 27

> CuatroAses.Sin(T)
He necesitado 22 extracciones para obtener cuatro ases
$E
[1] 22


$R
[1] 49 33 40 34 14 22  3 36  8 44 18 48
[13] 26 52 24 35 32 50 46  1 39 27
```

Igual que en el caso anterior, podrá realizar una simulación

```
> DistriAses.Sin = function(n = 5)
{
  Saco = vector(length=n)
  for(i in 1:n)
```

3.5. Ejemplos de funciones

```
        {Saco[i] = CuatroAses.Sin()$E}
      return(Saco)
    }

> Distribucion = DistriAses.Sin(2000)
> Distribucion
      [1] 42 30 49 49 41 ...
[1996] 52 47 52 30 38
> summary(Distribucion)
      Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
      8.00  38.00  45.00  42.43  49.00  52.00
> hist(Distribucion) 
```

Se podrían unir las dos opciones en una de la forma

```
CuatroAses(Mostrar, Maximo, Reemplazamiento=T)
```

y utilizar en su definición la orden

```
if (Reemplazamiento==F) {Maximo=52}
```

La siguiente versión devuelve además el nombre de la carta que se obtiene en cada extracción. Para crear una sola cadena de caracteres a partir de varias, utiliza la función `paste`.

```
> CuatroAses.Nombres = function(Mostrar = F,Maximo=1000)
{
#####
# Pone los cuatro ases en 0                                     #
# Saca una carta de la baraja (52)                             #
# Suma uno al contador                                         #
# Si no es un as (1) vuelve al principio del ciclo             #
# El as correspondiente se pone en 1                           #
# Si los cuatro ases son 1 termina el ciclo                     #
#####
  Palos = c("Oros","Copas","Espadas","Bastos")
  Cartas = c("As","Dos","Tres","Cuatro","Cinco",
             "Seis","Siete","Ocho","Nueve","Diez",
             "Sota","Caballo","Rey")
  Extracciones = 0
  Resultado = 1:Maximo
  Nombres=vector(length=Maximo, mode="character")
  Ases = c(0,0,0,0)
  repeat
  {
    Extracciones = Extracciones + 1
    if (Maximo <= Extracciones )
    {
      if(Mostrar)
      {
        cat("No he podido obtener cuatro ases en",
```


3.5. Ejemplos de funciones

```
      Extracciones,
      "extracciones. \n")
    }
    return(list(E = NA, R = Resultado,
               N=Nombres, Conseguido=F))
  }
  SacoUna = sample(52, 1)
  Resultado[Extracciones] = SacoUna
  Palo = ((SacoUna-1) %% 13 )+ 1
  Carta = ((SacoUna-1) %% 13)+1
  Nombres[Extracciones] =
    paste(Cartas[Carta],"de", Palos[Palo])
  if(SacoUna %% 13 != 1) {next}
  # %% es el módulo
  Ases[(SacoUna -1) %% 13 + 1] = 1
  # %% es la división entera
  if(sum(Ases)==4) {break}
}
length(Resultado)=Extracciones
length(Nombres)=Extracciones
if(Mostrar)
{
  cat("He necesitado", Extracciones,
      "extracciones para obtener cuatro ases.\n")
}
return(list(E = Extracciones, R = Resultado, N=Nombres, Conseguido=T))
}
> CuatroAses.Nombres(T,10)
No he podido obtener cuatro ases en 10 extracciones.
$E
[1] NA

$R
[1] 49 21 30 10 46 43 17 11 25

$N
[1] "Diez de Bastos" "Ocho de Copas" ...
[9] "Caballo de Copas"

$Conseguido
[1] FALSE

> CuatroAses.Nombres()
$E
[1] NA

$R
[1] 33 37 52 25 14  3 15 16  1 29 40 38 49 26 19 27

$N
```

3.5. Ejemplos de funciones

```
[1] "Siete de Espadas" "Sota de Espadas" ...  
[16] "As de Espadas"
```

```
$Conseguido  
[1] TRUE
```

3.5.6. Función Suma de potencias

En primer lugar aparece una función que calcula la suma de los cuadrados y cubos de las componentes de un vector, y los devuelve en un vector.

```
> sumpot=function(x=NA)  
{  
  s2=sum(x^2)  
  s3=sum(x^3)  
  return(c(s2,s3))  
}  
> sumpot(1:10)  
[1] 385 3025
```

También puede presentar el resultado como una lista con nombre para cada uno de los componentes:

```
> sumpot=function(x)  
{  
  s2 = sum(x^2)  
  s3 = sum(x^3)  
  return(list(cuadrados=s2,cubos=s3))  
}  
> sumpot(1:10)  
$cuadrados:  
[1] 385  
  
$cubos:  
[1] 3025  
  
> sumpot(1:10)$cubos  
[1] 3025  
> sumpot(1:10)[[2]]  
[1] 3025
```

Por supuesto, esta definición ha sustituido a la anterior, ya que el nombre es el mismo. También podría escribir la función de modo más compacto, como sigue, aunque tal vez no quede tan claro si los cálculos fuesen más complejos:

```
sumpot=function(x)  
{  
  list(cuadrados=sum(x^2),cubos=sum(x^3))  
}
```

A continuación se modifica la definición, añadiendo un segundo argumento que, predeterminadamente, da el mismo resultado, pero que permite obtener las sumas de otras potencias.

3.5. Ejemplos de funciones

```
> sumpot2=function(x,potencias=2:3)
{
    L=length(potencias)
    resultado=vector("list",length=L)
    for (i in 1:L) resultado[[i]]=sum(x^potencias[i])
    return(resultado)
}
> sumpot2(1:10,c(3,5,7))
[[1]]:
[1] 3025

[[2]]:
[1] 220825

[[3]]:
[1] 18080425
```

Aunque parezca indiferente devolver un vector o una lista, debe tenerse en cuenta que todas las componentes de un vector son del mismo tipo, por lo que, por ejemplo, si se mezclan números reales y complejos en el resultado se presentarían todos como complejos. Por ello suele utilizarse la lista para devolver resultados.

3.5.7. Función en una cuadrícula

La siguiente función genera una cuadrícula sobre un conjunto de ordenadas (x) y abscisas (y) y, para cada punto de esa cuadrícula, calcula la distancia hasta el origen de coordenadas.

```
> distancia.origen = function(x,y)
{
    puntos=matrix(-1,length(x),length(y))
    for (i in 1:length(x))
    {
        for (j in 1:length(y))
        {
            puntos[i,j]=sqrt(x[i]^2+y[j]^2)
        }
    }
    return(puntos)
}
> distancia.origen(1:3,-2:2)
      [,1] [,2] [,3] [,4] [,5]
[1,] 2.236068 1.414214 1 1.414214 2.236068
[2,] 2.828427 2.236068 2 2.236068 2.828427
[3,] 3.605551 3.162278 3 3.162278 3.605551
```

Es posible escribir una definición alternativa utilizando la forma general del producto exterior, que es menos didáctica que la anterior, pero que, curiosamente, es más rápida en ejecución. Para esta función se utiliza un nombre distinto,

3.5. Ejemplos de funciones

ya que la palabra ORIGEN está escrita en mayúsculas. Ello implica que se crea un **nuevo** objeto, que no sustituye al anterior, sino que coexisten.



```
> distancia.ORIGEN=function(x,y)
{
  outer(x,y,function(x,y) sqrt(x^2+y^2))
}
> distancia.ORIGEN(1:3,-2:2)
      [,1]      [,2] [,3]      [,4]      [,5]
[1,] 2.236068 1.414214  1 1.414214 2.236068
[2,] 2.828427 2.236068  2 2.236068 2.828427
[3,] 3.605551 3.162278  3 3.162278 3.605551
> ls(patt="dis*")
[1] "dist.ORIGEN" "dist.origen"
```

También podría escribir la función así:

```
> dista = function(x,y) sqrt(x^2+y^2)
> distancia.ORI = function(x,y)
{
  outer(x,y,"dista")
}
> distancia.ORI(1:3,-2:2)
      [,1]      [,2] [,3]      [,4]      [,5]
[1,] 2.236068 1.414214  1 1.414214 2.236068
[2,] 2.828427 2.236068  2 2.236068 2.828427
[3,] 3.605551 3.162278  3 3.162278 3.605551
> persp( distancia.ORIGEN(-40:40,-40:40))
```

3.5.8. Procesos no lineales

Las tres funciones siguientes corresponden a procesos no lineales descritos en Dong que, de forma sencilla, definen estructuras complejas. En el apartado de gráficos serán utilizadas para su representación. Para utilizarlas debe tenerse en cuenta que el parámetro **numero** debe ser un número natural mayor que 1.

```
 > dong1=function(numero = 100)
{
  x = vector(mode = "numeric", length = numero)
  y = vector(mode = "numeric", length = numero)
  x[1] = 1
  y[1] = 1
  for(i in 2:numero)
  {
    if(sample(2,1) == 2) 
      {m = 1}
    else
      {m = -1}
    x[i] = 0.5 * x[i - 1] + 0.5 * y[i - 1] + m
    y[i] = -0.5 * x[i - 1] + 0.5 * y[i - 1] + m
  }
}
```

3.5. Ejemplos de funciones

```
    return(list(x = x[2:numero], y = y[2:numero]))
}
```



```
> dong2=function(numero = 100)
{
  x = vector(mode = "numeric", length = numero)
  y = vector(mode = "numeric", length = numero)
  x[1] = 1
  y[1] = 1
  for(i in 2:numero)
  {
    a = sample(3,1)
    if(a == 1)
    {
      m = 0
      n = 0
    }
    else
    {
      if(a == 2)
      {
        m = 0.5
        n = 0
      }
      else
      {
        m = 0.25
        n = 0.5
      }
    }
    x[i] = 0.5 * x[i - 1] + m
    y[i] = 0.5 * y[i - 1] + n
  }
  return(list(x = x[2:numero], y = y[2:numero]))
}

> dong3=function(numero = 100)
{
  x = vector(mode = "numeric", length = numero)
  y = vector(mode = "numeric", length = numero)
  x[1] = 1
  y[1] = 1
  for(i in 2:numero)
  {
    a = sample(100,1)
    if(a == 1)
    {
      x[i] = 0
      y[i] = 0.25 * y[i - 1]
    }
  }
}
```

3.5. Ejemplos de funciones

```
else
{
if(a <= 86)
{
x[i] = 0.85 * x[i - 1] + 0.04 * y[i - 1]
y[i] = -0.04 * x[i - 1] + 0.85 * y[i - 1] + 1.6
}
else
{
if(a <= 93)
{
x[i] = 0.2 * x[i - 1] - 0.26 * y[i - 1]
y[i] = 0.26 * x[i - 1] + 0.22 * y[i - 1]
}
else
{
x[i] = -0.15 * x[i - 1] + 0.28 * y[i - 1]
y[i] = 0.26 * x[i - 1] + 0.24 * y[i - 1] + 1
}
}
}
}
return(list(x = x[2:numero], y = y[2:numero]))
}
```

Es posible modificar estas definiciones en muchos aspectos, para conseguir más claridad o robustez o velocidad. Por ejemplo, la siguiente modificación de la primera función, incluye el tiempo que tarda en realizarse y genera todos los elementos aleatorios en una sola ejecución de la función `sample`. La inclusión del tiempo permite analizar con diversas ejecuciones cual de las diferentes formas es más apropiada.

```
dong1.tiempo = function(numero = 100)
{
  Inicio=Sys.time()
  x = vector(mode = "numeric", length = numero)
  y = vector(mode = "numeric", length = numero)
  m = vector(mode = "numeric", length = numero)
  x[1] = 1
  y[1] = 1
  m=sample(c(-1,1),numero,T)
  for(i in 2:numero)
  {
    x[i] = 0.5 * x[i - 1] + 0.5 * y[i - 1] + m[i]
    y[i] = -0.5 * x[i - 1] + 0.5 * y[i - 1] + m[i]
  }
  Final=Sys.time()
  tiempo=Final-Inicio
  return(list(x = x[2:numero], y = y[2:numero],tiempo=tiempo))
}
```

3.5. Ejemplos de funciones

Las estructuras if anidadas son complejas, por lo que pueden modificarse los programas utilizando vectores auxiliares como hacemos a continuación:

```
dong2.v = function(numero = 100)
{
  x = vector(mode = "numeric", length = numero)
  y = vector(mode = "numeric", length = numero)
  x[1] = 1
  y[1] = 1
  XX = c(0, 0.5, 0.25)
  YY = c(0, 0, 0.5)
  for(i in 2:numero)
  {
    a = sample(3,1)
    x[i] = 0.5 * x[i - 1] + XX[a]
    y[i] = 0.5 * y[i - 1] + YY[a]
  }
  return(list(x = x[2:numero], y = y[2:numero]))
}
```

También es mejorable el uso de la función sample para generar números no equiprobables, como hacemos a continuación:

```
dong3.v=function(numero = 100)
{
  x = vector(mode = "numeric", length = numero)
  y = vector(mode = "numeric", length = numero)
  x[1] = 1
  y[1] = 1
  for(i in 2:numero)
  {
    a = sample(4,1, p=c(1,85,7,7))
    switch(a,
    {# Caso a=1
    x[i] = 0
    y[i] = 0.25 * y[i - 1]
    },
    {# Caso a=2
    x[i] = 0.85 * x[i - 1] + 0.04 * y[i - 1]
    y[i] = -0.04 * x[i - 1] + 0.85 * y[i - 1] + 1.6
    },
    {# Caso a=3
    x[i] = 0.2 * x[i - 1] - 0.26 * y[i - 1]
    y[i] = 0.26 * x[i - 1] + 0.22 * y[i - 1]
    },
    {# Caso a=4
    x[i] = -0.15 * x[i - 1] + 0.28 * y[i - 1]
    y[i] = 0.26 * x[i - 1] + 0.24 * y[i - 1] + 1
    }
    )
  }
}
```

3.6. Función missing

```
return(list(x = x[2:numero], y = y[2:numero]))
}
```

3.6. Función missing

La función `missing` permite comprobar si se ha dado valor a un parámetro. Debe utilizarse al comenzar el cuerpo de la función. Así la siguiente función detecta si se han incluido los argumentos y reacciona a su carencia. (No se incluyen las salidas gráficas.)

```
> dibuja = function(x,y)
{
  if(missing(x))
  {
    stop("El argumento 'x' es obligatorio")
  }
  if(missing(y))
  {
    y = x
    x = 1:length(y)
  }
  plot(x,y)
}
> dibuja()
Error in dibuja() : El argumento 'x' es obligatorio
> a=-10:10
> dibuja(a)
> dibuja(a,a^2)
> dibuja(,a^2)
Error in dibuja(, x^2) : El argumento 'x' es obligatorio
> dibuja(a^2)
```

3.7. Función menu

La función `menu` permite elegir, de modo interactivo, entre un conjunto de opciones. Su sintaxis es

```
menu(choices, graphics=F, title="")
```

choices es un vector de caracteres correspondientes a cada una de las opciones del menú.

graphics No se utiliza.

title es el título del menú

devolviendo el número correspondiente al elemento seleccionado o 0 si no se elige ninguna. Si se elige cualquier otra opción no la admite. Esta función es especialmente útil cuando se escribe un programa para que sea utilizado por personas sin experiencia en R.

3.8. Lista de búsqueda y entornos

Hasta ahora ha definido objetos, ha podido ver los existentes con `ls` y ha eliminado los superfluos con `rm`. Pero esos objetos se encuentran en un directorio concreto. También ha utilizado funciones que no aparecían al escribir `ls`. Puede encontrar todos los lugares donde se encuentran objetos con las funciones `search` y `searchpaths`.

```
> search()
[1] ".GlobalEnv"          "package:methods"    "package:stats"
[4] "package:graphics"    "package:grDevices"  "package:utils"
[7] "package:datasets"    "Autoloads"          "package:base"
> searchpaths()
[1] ".GlobalEnv"
[2] "C:/ARCHIV~1/R/rw2000/library/methods"
[3] "C:/ARCHIV~1/R/rw2000/library/stats"
[4] "C:/ARCHIV~1/R/rw2000/library/graphics"
[5] "C:/ARCHIV~1/R/rw2000/library/grDevices"
[6] "C:/ARCHIV~1/R/rw2000/library/utils"
[7] "C:/ARCHIV~1/R/rw2000/library/datasets"
[8] "Autoloads"
[9] "C:/ARCHIV~1/R/rw2000/library/base"
>
```

Estas funciones devuelven un vector con la lista de **libros** (*packages*) y objetos de R (usualmente hojas de datos) que están en uso actualmente (*attached*). Comienzan por *.GlobalEnv* y terminan con el libro *base* que está siempre cargado en R. La diferencia entre ambas es que la primera devuelve los nombres internos y la segunda los caminos desde donde se han cargado los libros y hojas de datos.

Esta lista, que se denomina lista de búsqueda, es similar a la orden **path** de DOS. Cada vez que R debe buscar un objeto, comienza a buscarlo en el primer elemento de la lista. Si no lo encuentra en esa posición, pasa al siguiente y así sucesivamente. Ello quiere decir que es posible que existan dos objetos con el mismo nombre — que se encuentren en posiciones distintas, claro está — pero el que esté en una posición posterior no aparecerá pues será eclipsado por el que se encuentre en la posición anterior.

Puede ver los objetos que se encuentran en una posición de la lista con la función `ls`. Como indica la ayuda, esta función, equivalente a `objects`, tiene cinco argumentos: **name**, **pos**, **envir**, **all.names** y **pattern**.

name Se denomina así por compatibilidad con versiones anteriores, pero corresponde al **entorno** en que se buscan los objetos. Su valor predeterminado es el entorno actual. Puede definirse el entorno por varios procedimientos:

1. Mediante un número entero, *n*. Si *n* es positivo, indica una posición, `search()[n]`. Si *n* = -1 indica `search()[2, 3, ...]`
2. El nombre de un elementos de `search()`
3. Un entorno explícito, incluyendo el uso de la función `sys.frame` para acceder a entornos de ejecución de funciones

3.8. Lista de búsqueda y entornos

pos Se mantiene por compatibilidad con versiones anteriores, pero es una forma alternativa de especificar el entorno.

envir Se mantiene por compatibilidad con versiones anteriores, pero es una forma alternativa de especificar el entorno.

all.names Es una variable lógica. Si es **T** entonces la función devuelve todos los objetos, en caso contrario no devuelve los que comienzan con un punto. Este último es el valor predeterminado.

pattern Es una frase que corresponde a una **expresión regular**, de tal modo que si se especifica sólo se devolverán los nombres de objetos que se correspondan con ella, en caso contrario se devolverán todos. Puede utilizar **?regex** para ver todas las posibilidades de las expresiones regulares en R.

Cuando se usa sin argumentos en el nivel superior (la ventana de órdenes) muestra los datos y funciones definidos por el usuario. Si se hace dentro de una función, devuelve los nombres de las variables locales de la función.

Como resultado se obtiene un vector con los nombres de los objetos. Así, los objetos que se encuentran en la posición **9** y cuyo nombre contiene la frase **log1** se obtienen con

```
> ls(9,pa="log1")
[1] "log10" "log1p"
```

Además de estos entornos que podemos considerar «horizontales» existen otros «verticales» que comienzan sobre el primero y son los correspondientes a cada función que se ejecuta. Solicite ayuda sobre la función **sys.parent** para obtener mayor información. La siguiente función aclara qué objetos existen dentro del entorno de una función.

```
> Entorno = function(x,y)
{
  MM=1
  ls()
}
> Entorno()
[1] "MM" "x"  "y"
```

Por ejemplo, si desea ejecutar los ejemplos de una función (**example**), puede hacerlo de dos formas según que el parámetro **local** tome el valor **T** o el valor **F**. En el primer caso los ejemplos se realizan de modo local a la función, con lo que no aparecen en el entorno global, en tanto que en el segundo caso si aparecerán.

Pruebe con

```
x
example(log10, local=T)
x
example(log10, local=F)
x
```



3.9. Entradas y salidas mediante archivos

Si desea saber si existe o no un objeto, fundamentalmente en programación, puede usar la función `exists(nombre,where)`, siendo **where** el lugar desde donde buscarlo, esto es, en esa posición o cualquiera superior.

```
> exists("xedit",5)
[1] TRUE
> exists("xedit",6)
[1] TRUE
> exists("xedit",7)
[1] FALSE
```

Si desea recuperar un objeto definido en una posición de la lista de búsqueda y oculto por otro con el mismo nombre en una posición anterior, puede utilizar la función `get(nombre,pos)`.

```
> get("matplot",4) -> MMM
```

Una vez recuperado el objeto (puede que con el mismo nombre) puede modificarlo si lo desea.

La lista de búsqueda puede ampliarse o reducirse⁸ añadiendo o eliminando objetos (*database*) y libros (*package*). Para añadir objetos se usa la función `attach` que tiene tres argumentos, **what**, **pos** y **name**. El primero especifica el nombre de un objeto, que se añadirá a la lista de búsqueda. Puede ser una hoja de datos, una lista o un archivo creado con la orden **save**. El segundo especifica la posición que debe ocupar en la lista, que por defecto es la segunda. Nunca debe elegirse la primera posición porque esta es especial y, por tanto, tiene un tratamiento especial. El tercer argumento corresponde al nombre con que se desea que aparezca en la lista. Si el nombre no se especifica, se construye con el nombre del objeto. Personalmente, no uso nunca la función `attach`.

Para añadir un libro se utiliza la función `library` que veremos posteriormente en 4.5.

Cuando no desee seguir utilizando un objeto, elimínelo de la lista con `detach` indicando bien el nombre o la posición del objeto que se desea eliminar de la lista. Si no se indica este valor se toma la segunda posición, que coincide con la de `attach`. Esta función no permite eliminar la primera posición ni la última, correspondiente al libro **base**.

3.9. Entradas y salidas mediante archivos

Si se dispone de una serie de órdenes escritas en un archivo, `archivo.R`, en el directorio de trabajo, es posible ejecutarlas mediante la función `source`, del siguiente modo:

```
source("archivo.R")
```

El nombre del archivo puede especificarse como un camino absoluto o relativo. El camino absoluto será de la forma `esquema//máquina/carpeta/archivo`. Debe prestarse atención al hecho de utilizar la barra como separador. Si desea

⁸No debería reducir la lista original, sólo debe reducir una lista ampliada sobre la original.

3.9. Entradas y salidas mediante archivos

utilizar la barra hacia atrás (la utilizada habitualmente en Windows) debe escribirla dos veces (la primera indica que lo siguiente es un carácter especial y la segunda indica que ese carácter es la barra hacia atrás).



Por ejemplo, puede usar un camino absoluto como

```
source("http://www.ugr.es/local/andresgc/archivo.R",print.eval=T)
```

o relativo, como

```
source("../archivo.R",print.eval=T)
```

Así, si realiza una sesión de trabajo, puede guardar todas las órdenes dadas⁹ en un archivo, depurarlas mediante un procesador de textos y posteriormente ejecutarlas de una sola vez. Si va a realizar este trabajo a menudo, es más eficiente crear una o varias funciones.

Los caminos relativos se establecen a partir del **directorio de trabajo**. Puede consultar el directorio de trabajo con la orden `getwd()`. También puede modificarlo con la orden `setwd(nombre del directorio)`

La función `sink` permite redirigir la salida a un archivo. Admite cuatro parámetros:

file Si especifica un archivo, los resultados se dirigen a ese archivo, hasta que se especifique la orden de nuevo, dirigiendo los resultados a otro archivo o devolviéndolos al valor anterior, que en el nivel 0 es la pantalla, si no se especifica ninguno.

append El parámetro **append** toma los valores lógicos **TRUE** y **FALSE**, según deban añadirse los resultados al archivo o borrarlo previamente. Por defecto el valor es **FALSE**, lo que indica que los resultados NO se añaden a los existentes en el archivo.

type Puede tomar los valores **output** y **message** indicando si se desea redirigir la salida o los mensajes. Esta última opción debe realizarse con precaución.

split Es una variable lógica. Si es **TRUE** entonces se envían los resultados tanto a la pantalla como al archivo, en caso contrario solo se envían al archivo. El valor predeterminado es **FALSE**.

El ejemplo siguiente dirige los resultados de las órdenes a `archivo.txt` y después de nuevo a la pantalla (Suponiendo que esta era la salida estándar en el momento anterior).

```
sink("archivo.txt")
Órdenes.....
sink()
```

⁹También puede copiarlas de pantalla, o revisar el archivo de historial de órdenes, **.Rhistory**

Capítulo 4

Uso de archivos externos

4.1. Función scan

Una de las funciones más versátiles, y por tanto más complejas, para leer datos de un archivo¹ es la función **scan**, que lee datos desde un archivo de texto o de la entrada estándar. Los argumentos son los siguientes:

file Es una cadena de caracteres que especifica un archivo. Si el nombre es relativo, comenzará en el directorio de trabajo. Debe tener en cuenta que el separador entre directorios es / y si desea utilizar el símbolo \ debe escribirlo doble, \\, ya que este símbolo se utiliza para definir caracteres especiales. Si especifica la cadena vacía, los datos se leen desde la entrada estándar, esto es, el teclado y el final de la entrada de datos se realiza mediante una línea en blanco o con el símbolo de fin de archivo (Ctrl+Z en Windows).

what Es un vector de modo numérico, carácter o complejo o una lista de vectores de dichos tipos, de tal modo que los datos del archivo se interpretan como de ese tipo. Si no se indica nada los datos se consideran numéricos. Si es una lista, se considera que cada registro tiene **length(what)** campos y el modo de cada uno es el correspondiente en la lista.

nmax Es el máximo número de datos a leer o si **what** es una lista, el máximo número de registros. Si se omite se lee hasta el final del archivo.

n Es el máximo número de datos a leer. Si se omite no existe límite.

sep El carácter utilizado como separador entre campos. Puede utilizar el tabulador, \t, o el cambio de línea, \n. El valor predeterminado es uno o más blancos, o cambios de línea.

quote Una cadena de caracteres que corresponde a los caracteres que sirven para delimitar cadenas de caracteres.

dec El separador decimal.

¹Al hacer referencia a un archivo debe entenderse que el mismo puede residir en el propio ordenador o en una dirección accesible mediante red. En este último caso suele indicarse que se trata de una conexión o de una dirección URI.

4.1. Función scan

skip Número de líneas que deben saltarse al inicio del archivo antes de comenzar a leer los datos.

nlines Número máximo de líneas para leer.

na.strings Caracteres que deben ser sustituidos, tras su lectura, por NA.

flush Si el valor es TRUE, una vez leídos el número de campos indicados, se descarta el resto de la línea.

fill Si el valor es TRUE, si una línea contiene menos datos de los esperados se rellenan con vacíos.

strip.white Vector de valores lógicos correspondientes a los elementos de **what** para indicar si deben eliminarse los espacios blancos al inicio y final de cada campo. En campos numéricos, siempre se eliminan.

quiet Valor lógico que indica si es FALSE, que es el valor predeterminado, que imprima el número de campos leídos.

blank.lines.skip Valor lógico que indica si es TRUE que se ignoren las líneas en blanco.



multi.line Valor lógico que se usa solamente si **what** es una lista y que indica si es FALSE que una línea lógica no puede ocupar más de una línea física.

comment.char Un carácter que indica que el resto de la línea es un comentario. El valor predeterminado es la cadena vacía, lo que indica que no hay comentarios.

El resultado es un vector o una lista si se utiliza **what** y un vector numérico en caso contrario. Como regla general, debe tener en cuenta que, lo más cómodo, es preparar los datos para que su lectura no resulte complicada.

Si en un campo numérico se encuentra la palabra **NA** se entiende que es un valor no disponible.

Hay que destacar que cada vez que se lee un registro debe hacerse una llamada a la asignación de memoria para hacer sitio. Si se sabe previamente el tamaño de los vectores que se van a leer y se asigna este tamaño, se ahorra gran cantidad de memoria, lo que permite leer archivos mayores.

Para leer datos *bien preparados* la función **read.table**   es más conveniente.

Cuando el nombre de un archivo se da como un camino relativo y no absoluto, el nombre completo se resuelve completándolo desde el directorio de trabajo. Consúltelo con **getwd** —usualmente se encuentra en **C:/Archivos de programa/rw2000/bin**— o cámbielo con la función **setwd**.

Ejemplos

Peso <- **scan()** lee datos del teclado, hasta que se deje una línea en blanco.

Nombres <- **scan("personas.txt", what="")** Lee un vector de cadenas de caracteres del archivo "personas.txt"

Si desea leer un archivo y que los datos correspondan a una matriz puede hacerlo combinando la función **matrix** con **scan**. Por ejemplo, para los datos incluidos en el archivo **datos.txt**, la orden de lectura sería



4.1. Función scan

```
> matrix(scan("c:/datos/datos.txt"),ncol=3,byrow=T)
      [,1] [,2] [,3]
[1,]   77 1.63   23
[2,]   58 1.63   23
[3,]   89 1.85   26
[4,]   55 1.62   23
[5,]   47 1.60   26
[6,]   60 1.63   26
[7,]   54 1.70   22
[8,]   58 1.65   23
[9,]   75 1.78   26
[10,]  65 1.70   24
[11,]  82 1.77   28
[12,]  85 1.83   42
[13,]  75 1.74   25
[14,]  65 1.65   26
```

Si quiere dar nombre a las columnas puede hacerlo asignando los valores a una lista, **nombres**, y luego pasándola como parámetro a la función **matrix**.

```
> nombres <-list(c(),c("Peso","Altura","Edad"))
> matrix(scan("c:/datos/datos.txt"),ncol=3,
  dimnames=nombres, byrow=T)
      Peso Altura Edad
[1,]   77   1.63   23
[2,]   58   1.63   23
[3,]   89   1.85   26
[4,]   55   1.62   23
[5,]   47   1.60   26
[6,]   60   1.63   26
[7,]   54   1.70   22
[8,]   58   1.65   23
[9,]   75   1.78   26
[10,]  65   1.70   24
[11,]  82   1.77   28
[12,]  85   1.83   42
[13,]  75   1.74   25
[14,]  65   1.65   26
```


También podría haberlo escrito directamente en una sola orden, aunque del modo anterior queda más claro.

```
> matrix(scan("c:/datos/datos.txt"),ncol=3,
  dimnames=list(c(),c("Peso","Altura","Edad")), byrow=T)
      Peso Altura Edad
[1,]   77   1.63   23
[2,]   58   1.63   23
[3,]   89   1.85   26
[4,]   55   1.62   23
[5,]   47   1.60   26
[6,]   60   1.63   26
```

4.1. Función scan

```
[7,] 54 1.70 22
[8,] 58 1.65 23
[9,] 75 1.78 26
[10,] 65 1.70 24
[11,] 82 1.77 28
[12,] 85 1.83 42
[13,] 75 1.74 25
[14,] 65 1.65 26
```

4.1.1. Función readLines

Para leer un archivo por líneas completas puede utilizar la función `readLines`  cuyos argumentos son

con El nombre del archivo

n Un entero que indica el máximo número de líneas para leer. Si se indica un valor negativo se leerá hasta alcanzar el final del archivo.

ok Valor lógico que indica si se espera alcanzar el final del archivo antes de leer el número de líneas especificado como máximo, si no es así se genera un error.

El resultado es un vector de tipo carácter cuyos elementos son las líneas leídas. A continuación escribimos cuatro líneas en el archivo `prueba.txt` mediante la función `cat`. Los cambios de línea vienen marcados por la secuencia `\n`. Seguidamente leemos el archivo por líneas.

```
> cat("Hola", 12, 13, "Soy\n", "yo\n",
"y\n", "te espero\n", file="prueba.txt")
> readLines("prueba.txt", n=-1)
[1] "Hola 12 13 Soy"
[2] " yo"
[3] " y"
[4] " te espero"
```

4.1.2. Función writeLines

Esta función escribe líneas de texto en un archivo. Sus argumentos son

text Un vector de tipo carácter

con El nombre de un archivo. Su valor predeterminado es la salida estándar.

sep Una cadena de caracteres que se escribe tras cada línea de texto. El valor predeterminado es `\n`.

Además de las funciones anteriores que consideran los archivos en modo texto es posible tratarlos en modo binario con las funciones `readBin`, `writeBin`, `readChar` y `writeChar`.

4.1. Función scan

4.1.3. Función unlink

Esta función permite eliminar un archivo o un directorio. Sus argumentos son

x Un vector de caracteres con los nombres de los archivos y directorios que se desea borrar. Es posible utilizar abreviaturas para especificar conjuntos.

recursive Valor lógico que indica si los directorios deben borrarse recursivamente o no. El valor predeterminado es **FALSE**, por lo que no se borran incluso aunque estén vacíos.

Así, si se desea borrar el archivo antes creado puede utilizar la orden

```
> unlink("prueba.txt")
```

4.1.4. Gestión de archivos

La gestión de archivos puede realizarse con un grupo de funciones dedicadas: **file.create**, crea los archivos que se le indican, **file.exists**, indica si existen, **file.remove**, elimina los archivos, **file.rename**, modifica el nombre de un archivo, **file.append**, añade un archivo al final de otro, **file.copy**, copia un archivo en otro con la opción de añadirlo, **file.symlink**, establece un enlace a un directorio, **dir.create**, crea un directorio, **file.info**, devuelve información sobre el archivo, **file.path**, devuelve el camino absoluto correspondiente al archivo, **file.access**, devuelve los permisos disponibles sobre un archivo, **file.show**, muestra el contenido de los archivos, **list.files**, muestra los nombres de los archivos de un directorio que se corresponden con una abreviatura especificada, **path.expand**, expande un camino especificado relativamente mediante una tilde inicial que indica el directorio inicial del usuario, **basename** y **dirname**, devuelven respectivamente la parte de nombre de archivo y nombre de directorio correspondientes al camino que especifica un archivo, **file.choose**, muestra un cuadro de diálogo para seleccionar un archivo, y **choose.files**, para seleccionar varios.

4.1.5. Conexiones

Además de la gestión de archivos, que incluye subsidiariamente conexiones, es posible realizar la gestión de las mismas directamente, lo que permite un control mayor sobre las mismas. Para ello se utilizan diversas funciones: **file**, gestiona completamente el manejo de archivos, **gzfile**, abre archivos comprimidos con *gzip*, **unz**, lee archivos comprimidos con *zip*, **bzfile**, abre archivos comprimidos con *bzip2*, **url**, abre una conexión en lectura, **socketConnection**, abre una conexión en lectura y escritura, **open** y **close** son procedimientos genéricos, **isOpen** e **isIncomplete** comprueban si está abierta una conexión y si se ha completado.

4.1.6. Personalización de archivos de un proyecto

Es posible (y aconsejable) personalizar cada proyecto. Conviene, como es lógico, mantener los archivos de un proyecto estadístico separados de los de otros proyectos. Para evitar posibilidades de errores, puede hacerse del siguiente modo:

1. Cree un directorio para el proyecto
2. Copie el enlace a R en el directorio (Ese icono que hay en el escritorio y que permite ejecutar R)
3. Modifique en esa copia el directorio de trabajo cambiándolo por actual, que será el específico del proyecto

Desde ese momento, al pulsar en ese enlace a R personalizado, todas las acciones se referirán al directorio de ese proyecto

4.2. Lectura de hojas de datos

Existen diversas funciones que permiten leer datos de un archivo y almacenarlos directamente en una hoja de datos. Son las funciones que más se utilizan para leer datos *bien preparados* y proceder a su análisis. Los datos del archivo deben estar en forma de tabla y se creará una hoja de datos con el mismo número de variables que campos o columnas haya en el archivo y con el mismo número de filas que líneas haya en el archivo. Estas funciones son `read.table`, `read.csv`, `read.csv2`, `read.delim` y `read.delim2`. La más completa es la primera, siendo el resto modificaciones de la misma simplificadas para leer datos en casos concretos.

La sintaxis es la siguiente:

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

Los parámetros son los siguientes

```
read.table(file, header, sep, quote, dec , row.names,  
           col.names,as.is, na.strings, skip)
```



4.2. Lectura de hojas de datos

con el siguiente significado:

file es el nombre del archivo a leer, que debe tener una línea por individuo. Si el nombre es relativo, se construye a partir del directorio de trabajo.

header es un valor lógico. Si es **TRUE**, la primera línea del archivo contiene los nombres de las variables. Si es **FALSE**, y la primera línea tiene un campo menos que las demás, también se interpreta como nombres de variables. En caso contrario se interpreta como valores del primer individuo. Es conveniente especificar este campo explícitamente.

sep Separador entre campos. El valor predeterminado es uno o varios espacios en blanco.

dec es el separador decimal.

quote es el conjunto de caracteres que delimitan cadenas de caracteres.

row.names permite especificar nombres para las filas. Puede hacerlo de dos modos: Dando un vector de cadenas de caracteres de la misma longitud que el número de filas o mediante un nombre o número que indica un campo del archivo que contiene los nombres. Si lo deja en blanco, los nombres dependen de la existencia o no de un campo de caracteres con valores distintos. Si lo hay, los utiliza como nombres, en caso contrario utiliza como nombres los números de orden de cada fila. Esta última opción puede forzarse dando a este parámetro el valor **NULL**.

col.names permite especificar nombres para las columnas o variables. Si no los indica aquí, ni están incluidos en la primera fila del archivo, se construyen con la letra **V** y el número de columna.

as.is vector de valores lógicos que indica cómo tratar los campos no numéricos. El valor predeterminado es **F** con el cual dichos campos se transforman en factores, salvo que se utilicen como nombres de filas. Si este vector es de longitud inferior al número de campos no numéricos se repetirá las veces necesarias. También puede ser un vector numérico que indica que columnas deben conservarse como caracteres.

na.strings es un vector de caracteres y controla con qué valor se indica un valor **NA**. Así, podrá utilizar ***** para leer este tipo de datos procedentes de BMDP.

colClasses Un vector de caracteres que indica la clase de cada columna.

nrows Número máximo de filas que se leerán.

skip indica el número de líneas del principio del archivo que deben saltarse sin leer.

check.names De tipo lógico, si es **TRUE** se comprueba que los nombres de variables son correctos y que no hay duplicados.

fill De tipo lógico, si es **TRUE** se completan las filas con blancos si es necesario.

4.2. Lectura de hojas de datos

strip.white De tipo lógico, si es `TRUE` y se ha definido un separador, se eliminan los espacios en blanco al principio y final de los campos de tipo carácter.

blank.lines.skip De tipo lógico, si es `TRUE` se ignoran las líneas en blanco.

comment.char Indica un carácter a partir del cual no se lee la línea, interpretándose como un comentario.

Aunque esta función es menos versátil que la función `scan` es la que recomendamos que se utilice, **preparando** los archivos de datos para que puedan ser leídos con ella de modo **sencillo**.

Las restantes funciones coinciden con `read.table` excepto en los valores predeterminados. Están construidas para leer archivos delimitados por comas² o tabuladores, o la variante utilizada en países, como España, en que la coma se sigue utilizando como separador decimal y el punto y coma como separador entre números. Advierta que en estas funciones se presupone que `header=TRUE`, esto es, que la primera línea contiene los nombres de las variables.

En la versión actual de R esta función consume bastante memoria, ya que primero carga todos los datos en memoria con `scan` y luego hace los cambios necesarios.

La lectura de hojas de datos puede realizarse desde archivos con formato fijo mediante la función `read.fwf`³ que tiene la sintaxis

```
read.fwf(  
  file, widths, header = FALSE, sep = "\t",  
  as.is = FALSE, skip = 0, row.names, col.names,  
  n = -1, buffersize = 2000, ...)
```

donde los parámetros especiales son:

widths Si los datos están en una sola línea, será un vector de enteros que contiene las anchuras de los campos. Si los datos ocupan varias líneas, una lista de vectores enteros, uno por cada línea.

buffersize Máximo número de líneas a leer en cada ocasión.

Por otra parte, la función `write.table` escribe una hoja de datos en un archivo con un formato compatible con su lectura con `read.table`.

Ejemplo

La siguiente orden lee el archivo `c:/datos/datos2.txt` y produce una hoja de datos, que se almacena en la hoja de datos **h.datos2**

```
> h.datos2 <-  
+ read.table("c:/datos/datos2.txt", header=T, as.is=T)  
      Peso Altura Edad Sexo  
Juana Garcia    77   1.63   23    M  
Silvia Lopez    58   1.63   23    M  
Andres Garces   89   1.85   26    H
```


²csv es el acrónimo de Comma Separated Values

³fwf es el acrónimo de Fixed Width Format

4.3. Gestión de objetos

Laura Perez	55	1.62	23	M
Adela	47	1.60	26	M
Yolanda Lopez	60	1.63	26	M
Lola Martinez	54	1.70	22	M
Alberto Garcia	58	1.65	23	H
Pedro Vera	75	1.78	26	H
Diego Moreno	65	1.70	24	H
Julio Angulo	82	1.77	28	H
Juan Trucha	85	1.83	42	H
Rafael Perez	75	1.74	25	H
Monica Sanchez	65	1.65	26	M


Las siguientes órdenes también leen el archivo `c:/datos/datos2.txt` y producen la hoja de datos `h.datos2`, pero cambiando el directorio de trabajo

```
> opar <- getwd()   
> setwd("c:/datos")  
> h.datos2 <- read.table("datos2.txt",header=T,as.is=T)  
> setwd(opar)
```

4.3. Gestión de objetos

Existen diferentes funciones que permiten almacenar y recuperar objetos, unas son binarias (rápidas y exactas) y otras textuales (legibles por el usuario y compatibles entre plataformas y versiones del lenguaje).

4.3.1. dump

`dump` produce representaciones⁴ en modo texto de un grupo de objetos. Su sintaxis es 

```
dump(list, file = "dumpdata.R", append = FALSE,  
      control = "all", envir = parent.frame(), evaluate = TRUE)
```

donde `list` es un vector⁵ de nombres de objetos (de tipo carácter) y `file` es el nombre del archivo en que se guardarán, que si viene dado por un nombre relativo se construirá desde el directorio de trabajo. `append` indica si se añade al contenido del vector o no. La función devuelve el vector de nombres de objetos.

Las representaciones realizadas incluyen el nombre de los objetos y pueden recuperarse con la orden `source` aunque es posible que el modo de los datos numéricos sea modificado y, además, si se recuperan objetos grandes pueden necesitarse grandes cantidades de memoria. En caso de objetos complejos puede que plantee problemas.

Mediante este procedimiento podrá exportar objetos con la finalidad de enviarlos a R en otro ordenador, posiblemente con otro sistema operativo.

⁴Utilizando `sink`, por lo que anula cualquier ejecución anterior de esta función

⁵se conserva la palabra `list` por compatibilidad, pero no es una lista

4.3.2. write

La función **write** permite escribir un vector o una matriz en un archivo. Solamente escribe los elementos. La matriz necesita ser traspuesta para que coincida con la representación interna de R. Esta orden es cómoda para leer posteriormente los datos con **scan** o desde otro programa. Su parámetros son:

x El objeto a escribir (vector o matriz)

file El nombre del archivo

ncolumns Número de columnas para escribir

append Valor lógico que indica si los datos deben añadirse a los existentes en el archivo.

4.3.3. dput y dget

La función **dput** también permite almacenar en un archivo externo, en ASCII, la definición de un objeto. La diferencia fundamental con **dump** es que no conserva el nombre del objeto, sino sólo la definición del mismo. La recuperación se realiza con **dget**. La sintaxis de ambas funciones es

```
dput(x, file = "", control = "showAttributes")
dget(file)
```

4.3.4. save y load

La función **save** almacena los objetos en representación binaria, mediante una representación XDR idéntica en todos los sistemas operativos, por lo que pueden transportarse sin dificultad. La recuperación de los datos se realiza con la función **load**. Si desea almacenar todos los objetos, puede utilizar la forma abreviada **save.image**.

La sintaxis es

```
save(..., list = character(0),
      file = stop("'file' must be specified"),
      ascii = FALSE, version = NULL, envir = parent.frame(),
      compress = FALSE)

save.image(file = ".RData", version = NULL, ascii = FALSE,
           compress = FALSE, safe = TRUE)

load(file, envir = parent.frame())

loadURL(url, envir = parent.frame(), quiet = TRUE, ...)
```

donde los argumentos de **save** son

... Los nombres de los objetos que se desea salvar y que se buscaran en el entorno

list Vector de tipo carácter con los nombres de los objetos que se desea salvar y que se buscaran en el entorno

4.3. Gestión de objetos

file Nombre del archivo donde se almacenarán los datos

ascii Valor lógico, si es `codeTRUE` se almacenará en ASCII, aunque no es recomendable.

version La del espacio de trabajo. En la versión actual es 2.

envir Entorno en el que se buscan los objetos que se van a almacenar

compress Valor lógico que indica si los datos deben comprimirse.

safe Valor lógico, si es `codeTRUE` se salvan los datos en un fichero temporal y si todo va bien, luego se cambia el nombre al mismo. En caso contrario se trabaja directamente con el archivo.

name nombre del archivo

quiet Valor lógico que indica si debe imprimirse algún mensaje (`FALSE`) o no (`TRUE`)

La recuperación de los objetos salvados se realiza con la función `load` que, de modo predeterminado, recupera los datos en el entorno actual. Si desea recuperarlos en otro entorno, debe especificarlo. Por ejemplo, `envir=globalenv()`, los recupera en el entorno global.

Ejemplos

La siguiente orden guarda todos los objetos de la primera posición de la lista de búsqueda, en el archivo *Total.R* del directorio de trabajo (Posiblemente el archivo resultante sea muy grande).

```
dump(ls(), "Total.R")
```

Las siguientes órdenes crean el objeto `x`, lo muestran en pantalla, lo salvan, lo eliminan, comprueban que no existe en el entorno de trabajo, y vuelven a recuperarlo.

```
> x <- (1:10)^2
> x
[1] 1 4 9 16 25 36 49 64 81 100
> dump("x")
> rm(x)
> x
Error: Object "x" not found
> source("dumpdata")
> x
[1] 1 4 9 16 25 36 49 64 81 100
```

El objeto ha sido salvado en ASCII en el archivo **dumpdata** con el siguiente contenido:

```
"x"<- c(1, 4, 9, 16, 25, 36, 49, 64, 81, 100)
```

Las siguientes órdenes realizan las mismas acciones

4.4. Creación de archivos temporales

```
> x <- (1:10)^2
> x
[1] 1 4 9 16 25 36 49 64 81 100
> dput(x,file="x.txt")
> rm(x)
> x
Error: Object "x" not found
> x <- dget("x.txt")
> x
[1] 1 4 9 16 25 36 49 64 81 100
```

El objeto ha sido salvado en ASCII en el archivo **x.txt** con el siguiente contenido:

```
c(1, 4, 9, 16, 25, 36, 49, 64, 81, 100)
```

Por último, realizamos el mismo conjunto de acciones con **write**:

```
> x <- (1:10)^2
> x
[1] 1 4 9 16 25 36 49 64 81 100
> write(x,file="x.txt")
> rm(x)
> x
Error: Object "x" not found
> x <- scan("x.txt")
> x
[1] 1 4 9 16 25 36 49 64 81 100
```

El objeto ha sido salvado en ASCII en el archivo **x.txt** con el siguiente contenido:

```
1 4 9 16 25
36 49 64 81 100
```

4.4. Creación de archivos temporales

Durante la ejecución de una función es posible necesitar crear un archivo, del que se garantice que no existe en el momento de la ejecución. Para ello R dispone de la función **tempfile** que permite disponer de uno o varios nombres de archivo que no existen en el momento de ejecutar la función y de la función **tempdir** para obtener el directorio temporal que utiliza el sistema operativo.

La sintaxis es la siguiente:

```
tempfile(pattern = "file", tmpdir = tempdir())
```

La función devuelve tantos nombres como longitud tenga el vector **pattern**, nombres que comenzarán respectivamente con la cadena de caracteres que se le suministra en cada componente del vector.

4.4. Creación de archivos temporales

```
> tempfile(c("A","B","PEPE"))
[1] "C:\\DOCUME~1\\Andres\\CONFIG~1\\Temp\\Rtmp460\\A23520"
[2] "C:\\DOCUME~1\\Andres\\CONFIG~1\\Temp\\Rtmp460\\B25402"
[3] "C:\\DOCUME~1\\Andres\\CONFIG~1\\Temp\\Rtmp460\\PEPE24824"
> tempdir()
[1] "C:\\DOCUME~1\\Andres\\CONFIG~1\\Temp\\Rtmp460"
>
```

Las órdenes siguientes realizan las siguientes acciones: Se calcula un nombre de archivo inexistente y se almacena en la variable **Temp**. Se dirige la salida estándar a dicho archivo, se obtiene una lista de objetos (que se almacena en el archivo citado), y de nuevo se dirige la salida estándar a pantalla. Se borra el archivo, dirigiendo la salida a la variable **SeBorra**, se comprueba que se ha borrado (la variable vale 0) y se eliminan las dos variables usadas.

```
> Temp<- tempfile(pa="Temp")
> sink(Temp)
> ls()
> sink()
> SeBorra <- unlink(Temp)
> SeBorra
[1] 0
> rm(Temp)
> rm(SeBorra)
```

Ejemplos

A continuación, se crean los objetos **x** y **f**, que se salvan y posteriormente se eliminan. Se recuperan desde el entorno interactivo, y también desde dos funciones, para comprobar las diferencias de almacenamiento en el entorno. En la función **g** se recuperan en el entorno de la función y, al terminar esta, desaparecen. En la función **gg** se recuperan en el entorno global, por lo que al terminar la función siguen existiendo. En este último caso, si existiesen objetos con los mismos nombres en el entorno global, habrían sido sustituidos por los nuevos.

```
> x <- 1:10
> f <- function(x) x^3
> save(x,f,file="Salvado")
> rm(x)
> rm(f)
> load(file="Salvado")
> x
[1] 1 2 3 4 5 6 7 8 9 10
> f
function(x) x^3
> rm(x)
> rm(f)
> g <- function()
+ {
+   load(file="Salvado")
```

4.5. Función library

```
+ return(x,f)
+ }
> g()
$x
[1] 1 2 3 4 5 6 7 8 9 10

$f
function(x) x^3

> x
Error: Object "x" not found
> f
Error: Object "f" not found
> gg <- function()
+ {
+ load(file="Salvado",globalenv())
+ return(x,f)
+ }
> gg()
$x
[1] 1 2 3 4 5 6 7 8 9 10

$f
function(x) x^3

> x
[1] 1 2 3 4 5 6 7 8 9 10
> f
function(x) x^3
```

4.5. Función library

Esta es una función **fundamental**. `library` gestiona los libros⁶ de la biblioteca, dando información sobre los existentes y cargándolos en memoria o descargándolos de la misma. Un libro está formado por funciones, datos y documentación; todos ellos contenidos en un directorio que contiene varios subdirectorios, con informaciones diversas, de acuerdo con un formato establecido. De hecho, al cargar R, se carga al menos un primer libro denominado **base**.

La sintaxis es

```
library(package, help = NULL, lib.loc = .lib.loc,
         character.only = FALSE, logical.return = FALSE,
         warn.conflicts = TRUE)
```

cuyo significado es:

package es el nombre del libro.

help si se le da el nombre de un libro, devuelve información sobre los componentes del mismo.

⁶En R se denominan *packages*.

4.5. Función `library`

lib.loc es una cadena de caracteres que describe la posición de las bibliotecas para buscar en ellas.

logical.return es un valor lógico que devuelve si ha tenido éxito o no la acción.

warn.conflicts es un valor lógico que indica que se impriman los mensajes sobre conflictos al conectar el libro, salvo que este contenga un objeto `'conflicts.OK'`.

quietly es un valor lógico que indica que no se dé mensaje si no existe el libro en la biblioteca.

libname es una cadena de caracteres que contiene el nombre del directorio donde se encuentra el libro.


pkgname es una cadena de caracteres que contiene el nombre del libro.

all.available es un valor lógico que indica que se devuelva en un vector los nombres de todos los libros disponibles en **lib.loc**.

Si se llama a la función sin argumentos, devuelve una breve descripción de los diferentes libros disponibles en cada biblioteca de **lib.loc**.

Ejemplos

library() devuelve una lista de los libros que hay en la biblioteca predefinida de R.

 **library(help="base")** devuelve una pequeña ayuda sobre el libro `base` (Que siempre está en memoria, consulte **search()**).

En la ventana que aparece con la información puede observar una lista de los objetos que componen el libro. Puesto que este libro en concreto está en memoria, basta que utilice la ayuda sobre uno de los objetos, por ejemplo **help(Arithmetic)**, para acceder a información detallada sobre el mismo, y si está en Microsoft Windows ® y ha seleccionado la opción predeterminada en la instalación, al archivo compilado de ayuda, en el que encuentra toda la información sobre el libro.

Si consulta la lista de búsqueda observará los libros que hay actualmente en memoria, por ejemplo,

```
> search()
[1] ".GlobalEnv"          "package:stats"      "package:graphics"
[4] "package:grDevices"   "package:utils"      "package:datasets"
[7] "package:methods"     "Autoloads"          "package:base"
```

Si desea añadir el libro `cluster` (que suponemos está en la biblioteca y que ha aparecido al ejecutar la orden **library()** anterior) basta con que escriba la orden **library(cluster)** y podrá observar que se ha incluido en la lista ocupando, como es habitual, la segunda posición:

```
> search()
[1] ".GlobalEnv"          "package:cluster"    "package:stats"
[4] "package:graphics"    "package:grDevices" "package:utils"
[7] "package:datasets"    "package:methods"    "Autoloads"
[10] "package:base"
```

4.5. Función `library`

Para obtener una pequeña ayuda sobre el contenido del libro escriba

```
library(help=cluster)
```

(aunque no esté cargado en memoria, basta con que esté en la biblioteca). El primer objeto que aparece en esta ayuda es **agnes**. Por tanto, una vez cargado el libro en memoria, puede utilizar la orden `help(agnes)` para acceder a la información correspondiente a este libro.

Cuando ya no necesite utilizar el libro **cluster**, descárguelo con la orden `detach()`. Recuerde que esta orden elimina la segunda componente de la lista, así que no la use una segunda vez, ya que eliminaría el siguiente libro (**stats** en este ejemplo). Yo, personalmente, prefiero terminar la sesión de R cuando ya no necesito un libro concreto.

La instalación de un libro en el sistema consiste en copiar el directorio que lo contiene en un lugar accesible. El método más sencillo consiste en descomprimir el archivo **zip** en que se distribuye en el directorio **library**, dentro de la estructura de R.

Si lo desea puede instalar un libro en cualquier directorio accesible desde el ordenador y luego, cada vez que desee utilizarlo, indicar el camino de dicho directorio, que será entendido como una biblioteca. Así, por ejemplo, si descomprime el archivo **random_0.1.2.zip** en el directorio **C:\lib**, se creará un directorio denominado **random**. Si desea cargar este libro en memoria puede hacerlo con la orden⁷

```
library(random,lib.loc="C:/lib")
```

Del mismo modo, si desea saber qué libros hay en esa biblioteca, puede utilizar la orden

```
library(lib.loc="C:/lib")
```

Si observa el directorio **C:\lib\random**, encontrará el archivo de ayuda, **random.chm** y diversa documentación sobre este libro. La estructura de un libro puede encontrarla en el manual *Writing R Extensions* que se suministra con R.

La función `.libPaths` permite ver o modificar la lista de bibliotecas disponibles, que es un vector de cadenas de caracteres. La variable **.Library** contiene el camino a la biblioteca predeterminada del sistema.

⁷Recuerde utilizar la barra hacia adelante como separadora de directorios

Capítulo 5

Gráficos

Antes de generar un gráfico y representarlo, es necesario tener abierto un dispositivo que los admita (y que éste sea el dispositivo al que van a dirigirse los resultados gráficos, lo que se denomina como dispositivo gráfico actual, de entre los varios que pueden estar abiertos).

5.1. Dispositivos gráficos

La orden `?Devices` indica qué dispositivos gráficos están disponibles en el sistema en que trabajamos. En la versión de Microsoft Windows®, puede utilizar las funciones `x11`¹, `win.print`, `pictex`, `png`, `jpeg`, `bmp`, `win.metafile` y `postscript`. En cada caso, se abre el dispositivo gráfico y no se devuelve nada a R.

5.1.1. Función `x11`

La función `x11` abre una (nueva) ventana gráfica a la que irán dirigidos los resultados gráficos desde ese momento². Los argumentos de la función, todos opcionales, son `width`, `height` y `pointsize`.

width corresponde a la anchura lógica del eje X, medida en pulgadas, que automáticamente se hace corresponder a la anchura de la ventana. Ello importa para el momento en que este gráfico se pasa a impresora.

height hace el mismo papel respecto de la altura.

pointsize hace referencia al tamaño del punto de la fuente.

5.1.2. Función `win.print`

La función `win.print` genera gráficos directamente sobre el administrador de impresión del sistema. Podrá elegir en qué impresora de las instaladas y de qué modo se realizará la impresión. Así pues, si se ha instalado una impresora Postscript, podrá también realizar la impresión en un archivo Postscript por este procedimiento.

¹Son sinónimas las funciones ‘windows’, ‘win.graph’ y ‘X11’.

²Salvo que haya varios dispositivos gráficos abiertos y se cambie a otro

5.1.3. Función `pictex`

La función `pictex` genera gráficos que pueden utilizarse en \TeX y en \LaTeX . Su forma es

```
pictex(file = "Rplots.tex", width = 5, height = 4,  
        debug = FALSE, bg = "white", fg = "black")
```

Los argumentos son los siguientes:

file es el nombre del archivo donde se almacenará el gráfico. Su valor predeterminado es `Rplots.tex`.

width es la anchura, medida en pulgadas. Su valor predeterminado es 5.

height es la altura, medida en pulgadas. Su valor predeterminado es 4.

debug indica si debe imprimirse información para depuración. Su valor predeterminado es `FALSE`.

bg es el color del fondo. Su valor predeterminado es `"white"`.

fg es el color de la tinta. Su valor predeterminado es `"black"`.

Para utilizarlo en \LaTeX , debe incluir la biblioteca `pictex`, escribiendo la orden

```
\usepackage{pictex}
```

tras lo cual puede incluir el gráfico, por ejemplo, como una figura flotante con las órdenes

```
\begin{figure}[htp]  
  \centerline{\input{Rplots.tex}}  
\end{figure}
```

Si desea utilizarlo en \TeX , deberá incluir la biblioteca `pictex`, escribiendo la orden

```
\input pictex
```

tras lo cual puede incluir el gráfico, por ejemplo, centrado, con la orden

```
$$ \input Rplots.tex $$
```

5.1.4. Funciones `bmp`, `jpeg` y `png`

Cada una de estas funciones genera un gráfico en el formato que indica su nombre. Debe tener en cuenta que para el formato JPEG, R realiza una compresión del archivo, de tal modo que el gráfico resulta modificado, por lo que debe comprobar siempre si el mismo es correcto.

La forma de las funciones es

5.1. Dispositivos gráficos

```
bmp(filename="Rplot.bmp", width=480, height=480,
     pointsize=12)
jpeg(filename="Rplot.jpg", width=480, height=480,
     pointsize=12, quality=75)
png(filename="Rplot.png", width=480, height=480,
     pointsize=12)
```

filename es el nombre del archivo donde se almacenará el gráfico.

width es la anchura del gráfico, medida en pixels.

height es la altura del gráfico, medida en pixels.

pointsize es el tamaño del punto para dibujar textos.

quality sólo se aplica a la función `jpeg`. En este formato se realiza una compresión y este parámetro indica, mediante un porcentaje, la calidad del gráfico. Cuanto menor es el valor, más se comprime la imagen, y previsiblemente se obtiene peor calidad.

5.1.5. Función `win.metafile`

La función `win.metafile` imprime el gráfico en un archivo. Su forma es

```
win.metafile(filename = "", width = 7,
             height = 7, pointsize = 12)
```

filename es el nombre del archivo de salida, que contendrá un archivo en formato 'Microsoft Windows metafile', y cuya extensión habitualmente es '.emf' o '.wmf'. Si el valor de este argumento es una cadena vacía, el gráfico se pasa al portapapeles, desde donde podrá copiarse a otro programa.

width corresponde a la anchura lógica del eje X, medida en pulgadas, que automáticamente se hace corresponder a la anchura de la ventana. Ello importa para el momento en que este gráfico se pasa a impresora.

height hace el mismo papel respecto de la altura.

pointsize hace referencia al tamaño del punto de la fuente.

5.1.6. Función `postscript`

La función `postscript` genera gráficos que pueden utilizarse en cualquier dispositivo que acepte el lenguaje `Postscript`. En particular es útil para impresoras `Postscript`, y para incluir el gráfico en un procesador de textos. También puede utilizarlo en un intérprete `Postscript`, como `GHOSTSCRIPT` y `GSVIEW`.

Sus argumentos son

file es el nombre del archivo donde se almacenará el gráfico

paper es el tamaño del papel. Las posibilidades son "a4", "letter", "legal", "executive", y "special". En este último caso, será necesario definir el tamaño (anchura y altura) del papel.

5.1. Dispositivos gráficos

horizontal es la orientación del papel. Si es T, valor predeterminado, la orientación será apaisada; si es F, será vertical.

width, height son la anchura y altura, medidas en pulgadas. Los valores predeterminados son las medidas de la página, menos un borde de 0.25 pulgadas.

family es la familia de fuentes a utilizar. Puede seleccionar "AvantGarde", "Bookman", "Courier", "Helvetica", "Helvetica-Narrow", "NewCenturySchoolbook", "Palatino" o "Times".

pointsize es el tamaño de punto a utilizar.

bg es el color del fondo.

fg es el color de la tinta.

onefile si es T, valor predeterminado, permite almacenar varias figuras en un archivo, si es F, sólo permite almacenar una, y lo hace en formato EPS, que es el que debe utilizar para poder insertar la imagen en un procesador de textos.

pagecentre si es T, valor predeterminado, centra el gráfico en la página.

Ejemplos

Las siguientes órdenes crean tres gráficos, en tres ventanas distintas, con las realizaciones de tres procesos no lineales a través de las correspondientes funciones estudiadas anteriormente. Los resultados se muestran en las figuras 5.1, 5.2 y 5.3.

```
> dong1.datos<-dong1(10000)
> dong2.datos<-dong2(10000)
> dong3.datos<-dong3(10000)
> win.graph()
> plot(dong1.datos,col="yellow")
> win.graph()
> plot(dong1.datos,col="blue")
> win.graph()
> plot(dong3.datos,col="red")
```

Las órdenes siguientes crean dos gráficos, en formato metafile, y los almacenan en sendos archivos con los nombres **fig1.wmf** y **fig2.wmf**.

```
> win.metafile (file="fig1.wmf")
> plot(x)
> win.metafile (file="fig2.wmf")
> plot(y)
```


Figura 5.1: Realización del proceso Dong1

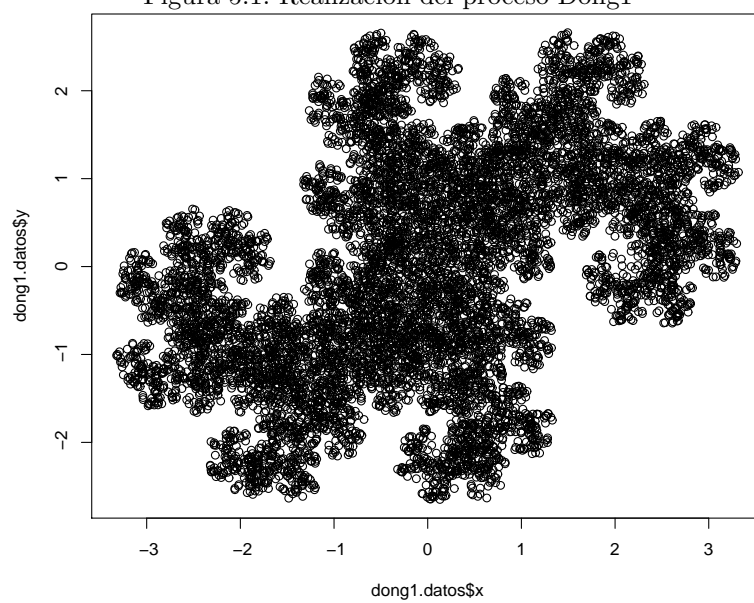
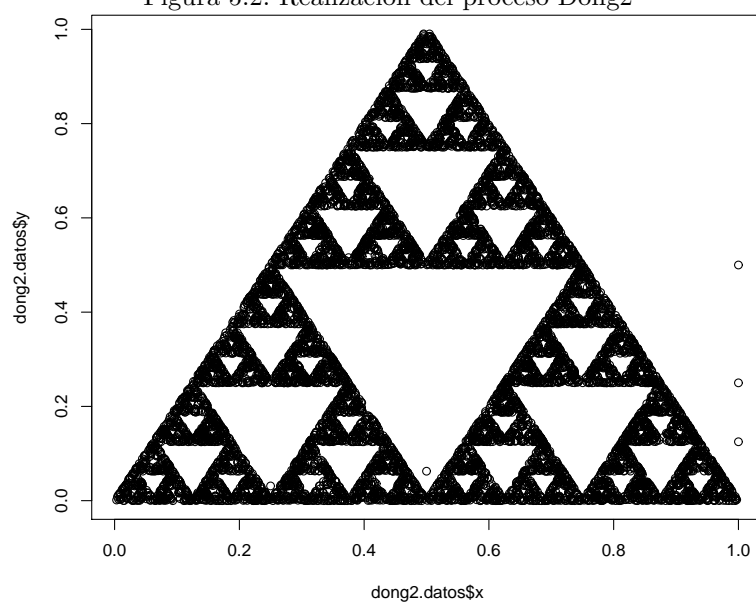
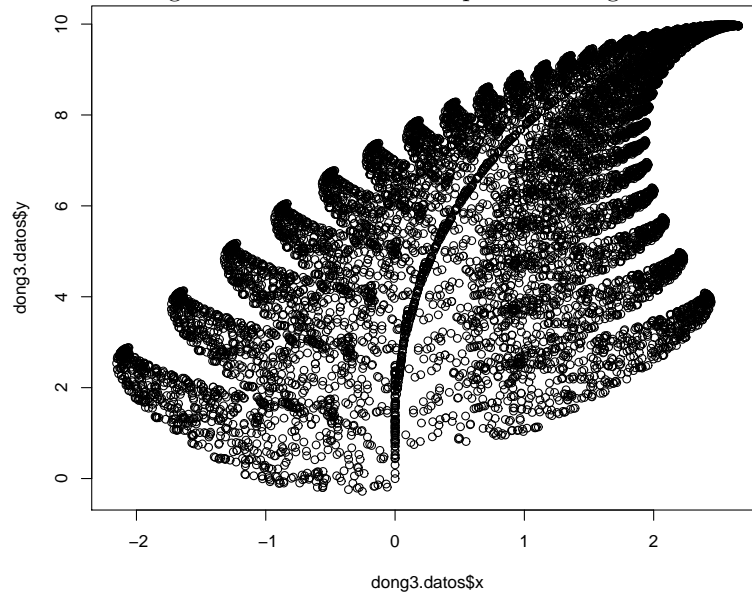


Figura 5.2: Realización del proceso Dong2



5.2. Función `graphics.off`

Figura 5.3: Realización del proceso Dong3



5.2. Función `graphics.off`

La función `graphics.off()` cierra todos los dispositivos gráficos. Esta función, como efecto adicional, termina de dibujar los gráficos pendientes, como puede ocurrir al mandar un gráfico a `win.print`.

A continuación veremos algunas funciones de las muchas disponibles en el libro `graphics`.

5.3. Función `plot`

`plot` es una función genérica que crea un gráfico en el dispositivo gráfico actual. Además existen funciones específicas en las que funciona de modo especial, como por ejemplo para `data.frame`, `lm`, etc. La forma de uso habitual es `plot(x, ...)`, donde `x` es uno o varios objetos. Además es posible utilizar diferentes parámetros, lo que se hace con la función `par`.

Por ejemplo, las dos órdenes siguientes representan, respectivamente, un tramo de la parábola $y = x^2$, y todas las parejas de gráficos bidimensionales de las variables incluidas en la hoja de datos `h.datos2`, definida en la página 65.

```
> win.graph()
> plot(1:100, (1:100)^2, type="l")
> win.graph()
> plot(h.datos2)
```

5.4. Función `text`

Esta función añade texto a un gráfico existente. Su sintaxis es

5.5. Función `symbols`

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

siendo los argumentos más importantes los siguientes:

x e **y** son vectores numéricos de coordenadas en que deben ser escritas las etiquetas (**labels**) de texto.

labels es un vector de tipo carácter o una expresión que especifica el texto que se desea escribir.

adj contiene uno o dos valores del intervalo $[0,1]$ que especifican el ajuste de las etiquetas en x e y respectivamente. El valor predeterminado es `c(0.5,0.5)` que significa centrado en ambas coordenadas.

pos toma valores 1,2,3 o 4 para indicar que el texto debe estar debajo, a la izquierda, encima o a la derecha de las coordenadas especificadas.

cex es el factor de expansión de los caracteres.

col y **font** son el color y la fuente.

El texto puede rotarse respecto del centro definido por **adj** mediante el parámetro gráfico **srt**.

Por ejemplo, las órdenes siguientes producen el gráfico de la figura 5.4.

```
curve(sin(x), -pi, pi, col="blue")  
text(-3, 0.5, "sen(x)", col="blue", cex=4, adj=0)
```

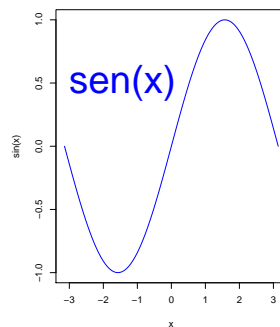


Figura 5.4: Curva y texto

5.5. Función `symbols`

Esta función permite dibujar círculos, cuadrados, rectángulos, estrellas, termómetros y cajas en una posición determinada de un gráfico, indicando además el tamaño que deben tener. Su sintaxis es

```
symbols(x, y = NULL, circles, squares, rectangles, stars,  
        thermometers, boxplots, inches = TRUE, add = FALSE,  
        fg = par("col"), bg = NA, xlab = NULL, ylab = NULL,  
        main = NULL, xlim = NULL, ylim = NULL, ...)
```

5.6. Función fourfoldplot

Los valores de `circles`, `squares`, `rectangles`, `stars`, `thermometers`, `boxplots`, son las medidas de los símbolos que se desea representar, `fg` y `bg` corresponden, respectivamente, al color de tinta y el de fondo.

Las siguiente función genera `n` valores de la distribución uniforme entre 0 y 1 y los representa mediante círculos de colores, como puede observarse en la figura 5.5 obtenida mediante `Globos()`.

```
Globos=function(n=10)
{
  x = 1:n
  palette(rainbow(n))
  z = runif(n)
  symbols(x,z,circles=z,xlim=c(-1,n+2),ylim=c(0,1.5),bg=1:n)
}
```

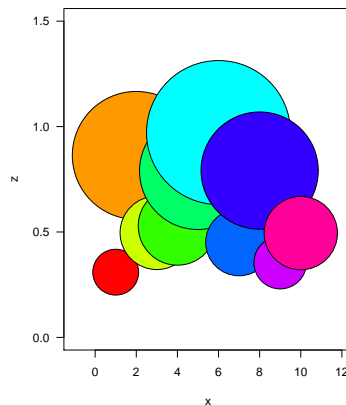


Figura 5.5: Símbolos

5.6. Función fourfoldplot

Esta función realiza una representación de tipo *fourfold* de k tablas de contingencia 2×2 . La sintaxis es

```
fourfoldplot(x, color = c("#99CCFF", "#6699CC"),
  conf.level = 0.95,
  std = c("margins", "ind.max", "all.max"),
  margin = c(1, 2), space = 0.2, main = NULL,
  mfrow = NULL, mfcoll = NULL)
```

5.7. Función hist

Esta función genera y devuelve un histograma de los datos suministrados³ y además lo dibuja si se indica el parámetro **plot=TRUE**. La sintaxis es

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

x es el vector que contiene los valores para los que se realizará el histograma.

breaks son los puntos de corte que definen los intervalos y pueden venir dados directamente, o por el número de intervalos, o por una cadena de caracteres que indican uno de los varios algoritmos existentes para construir los intervalos, o por una función que calcule el número de intervalos.

freq es una variable de tipo lógico que indica si se representan frecuencias absolutas (TRUE) o relativas (FALSE).

include.lowest es una variable de tipo lógico que indica en que intervalo se incluyen los puntos que coincidan con el extremo del intervalos.

right es una variable de tipo lógico que indica si los intervalos son cerrados por la derecha.

density es el número de líneas por pulgada para hacer sombreados.

angle es la pendiente en grados para las líneas de sombreado.

col es el color con que se rellenan las barras.

border es el color con que se dibujan las barras.

Las siguientes órdenes generan una muestra pseudoaleatoria de una normal tipificada, representan el histograma de los valores obtenidos y superponen el gráfico de la normal tipificada, como puede observarse en la figura 5.6.

```
hist(rnorm(100), col="red", freq=F, xlim=c(-5,5),
     ylim=c(0, dnorm(0)*1.2), xlab="",
     ylab="Densidad", main="Muestra y Población")

curve(dnorm(x), -5, 5, add=T, col="blue")
```

5.8. Función polygon

Esta función dibuja polígonos definidos por sus vértices, sobre un gráfico ya existente. La sintaxis de la orden es

```
polygon(x, y = NULL, density = NULL, angle = 45,
        border = NULL, col = NA, lty = par("lty"), ...)
```

³Un histograma es un objeto de la clase *histogram* que es una lista que, entre otros, contiene **breaks**, **counts**, etc.

5.9. Función curve

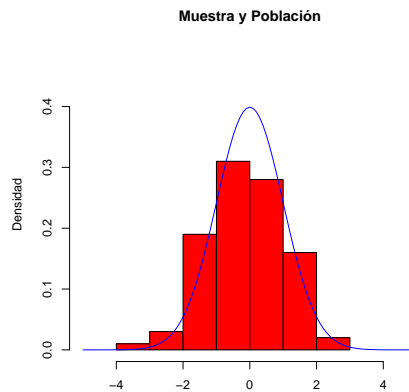


Figura 5.6: Histograma de una muestra de la Normal

donde **x** e **y** son vectores que contienen las coordenadas de los vértices del polígono. El polígono se cierra uniendo el último punto con el primero. En las coordenadas puede haber valores NA en cuyo caso se realizarán varios polígonos.

density es el número de líneas por pulgada para realizar el sombreado.

angle es la pendiente en grados de las líneas de sombreado.

border es el color con que se dibuja el borde del polígono. Si es **NA** no se dibujan.

col es el color del relleno del polígono.

lty es el tipo de línea que se usa.

Por ejemplo, las órdenes siguientes producen el gráfico de la figura 5.7.

```
x=c(1,9)
plot(x, x, type="n")
polygon(c(2,4,4,2,NA,6,8,8,6), c(2,2,4,4,NA,6,6,8,8),
        density=c(10, 20), angle=c(-45, 45),col=c("red","blue"))
```

5.9. Función curve

Esta función dibuja la curva correspondiente a una función dada en un intervalo concreto. Sus argumentos son:

expr es una expresión que corresponde a una función de x o el nombre de una función.

from, **to** determinan el intervalo en que se realizará la gráfica.

n = 101 es el número de puntos que se dibujará.

add = FALSE indica si el gráfico se añade o no al gráfico actual.

type = "l" indica el tipo de gráfico, predeterminadamente es líneas.

xlim = NULL es un vector numérico de longitud 2. Si se especifica indica los límites del gráfico.

... son argumentos adicionales que se pasan

También pueden pasarse etiquetas de los ejes y parámetros gráficos.

Así, por ejemplo, las órdenes siguientes producen el gráfico de la figura 5.8.

5.10. Funciones `lines` y `points`

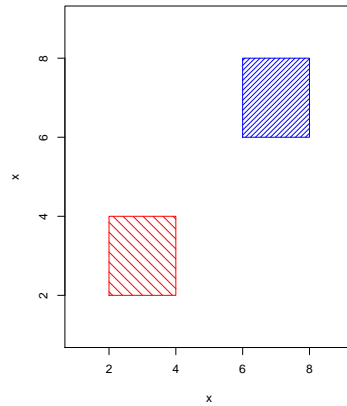


Figura 5.7: Polígonos

```
curve(sin(x),-pi,pi,col="blue")
curve(cos(x),-pi,pi,add=TRUE, col="red")
```

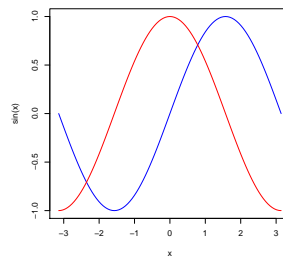


Figura 5.8: Representación de dos curvas

5.10. Funciones `lines` y `points`

Con estas funciones genéricas, puede añadir puntos o líneas al gráfico actual. Recuerde que debe haber realizado un gráfico, el cual define las medidas, aunque puede hacerlo con `type=n` si no desea representar ningún punto. La forma de uso es

```
lines(x, y, type="l")
points(x, y, type="p")
```

donde `x` e `y` son las coordenadas de unos puntos. Las coordenadas pueden expresarse mediante dos vectores, una matriz de dos columnas, una lista con dos componentes llamados `x` e `y`, un vector complejo, o una serie temporal univariante.

5.10. Funciones `lines` y `points`

Si alguno de los puntos a representar es **NA**, no se dibuja. Si está dibujando líneas, se producirá una ruptura.

También puede pasar parámetros gráficos de la función `par`. Por ejemplo, con el parámetro `pch`, puede seleccionar símbolos especiales para representar cada punto.

Ejemplos

A continuación generamos 30 valores pseudoaleatorios de una normal tipificada, los representamos y añadimos líneas, y líneas y puntos.

```
x=rnorm(30)
opar=par(mfrow=c(1,3))
plot(x,xlab="Tiempo",ylab="Ruido")
plot(x,xlab="Tiempo",ylab="Ruido")
lines(x,col="blue")
plot(x,xlab="Tiempo",ylab="Ruido")
lines(x,col="blue")
points(x,col="red",pch=4)
par(opar)
```

El resultado puede observarse en la figura 5.9.

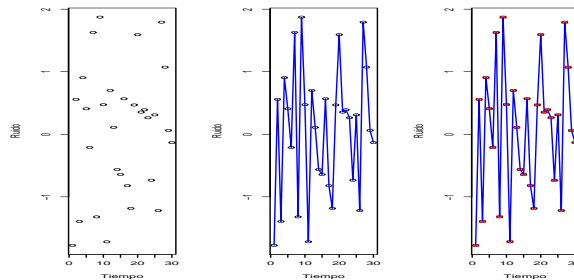


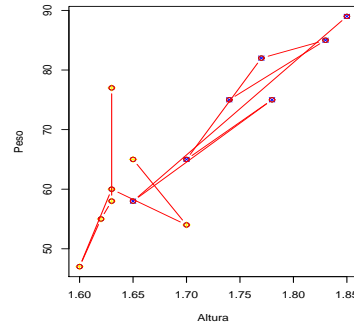
Figura 5.9: Puntos y Líneas

En este segundo ejemplo, utilizando los datos almacenados en el objeto **h.datos2**, seleccionamos en primer lugar las mujeres. A continuación representamos los valores de altura frente a peso, marcamos de modo distinto los hombres y mujeres y unimos con tramos rectos los elementos de cada grupo. El resultado se muestra en la figura 5.10.

```
> attach(h.datos2)
> Mujer<-Sexo=="M"
> plot(Altura,Peso)
> points(Altura[Mujer], Peso[Mujer],
+   type="p",pch=3,col="yellow")
> points(Altura[!Mujer],Peso[!Mujer],
+   type="p",pch=4,col="blue")
> points(c(Altura[Mujer],NA,Altura[!Mujer]),
+   c(Peso[Mujer], NA,Peso[!Mujer]),type="b",col="red")
> detach(h.datos2)
```


5.11. Función `par`

Figura 5.10: Mujeres y Hombres



5.11. Función `par`

Mediante la función `par` se controlan los parámetros gráficos que afectan al resultado de un gráfico. Todos sus argumentos son optativos y, por tanto, deben darse mediante el nombre. Si los argumentos de esta función se especifican en el interior de una de las funciones gráficas de nivel alto que hemos estudiado, sus efectos son temporales, en tanto que utilizados en la función `par`, son permanentes. La función `par` devuelve los valores actuales de sus parámetros. Por tanto, estos pueden almacenarse para ser recuperados posteriormente.

Puede comprobar como las órdenes siguientes producen dos gráficos distintos, representados en las figuras 5.11 y 5.12.

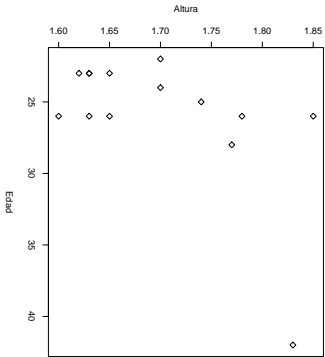
```
> attach(h.datos2)
> x11()
> plot(Edad,Altura)
> x11()
> par.antiguo <- par(pch=3)
> # Almacena los valores originales
> # y selecciona un nuevo símbolo
> # para representar los puntos
> plot(Edad,Altura)
> par(par.antiguo)
> # Restablece los valores originales
> detach(h.datos2)
```

Existen muchos argumentos, que puede consultar utilizando `?par`, pero los más comunes son los siguientes. Puede conocer los valores predeterminados con la orden `par()`.

no.readonly Existen dos tipos de parámetros, unos modificables y otros no. Si `no.readonly` es el único argumento y vale `FALSE`, se devuelven todos los parámetros, si es `TRUE`, sólo se devuelven los modificables.

ask Si debe preguntar antes de borrar un gráfico. Los valores son `TRUE` y `FALSE`.

Figura 5.11: pch original



5.11. Función `par`

bg Color de fondo del gráfico. Vea el parámetro `col`.

col Indica el color. Puede hacerse mediante el nombre del color. La función `colors` devuelve⁴ los nombres de colores disponibles. También puede definirse mediante una cadena de caracteres de la forma "`#RRGGBB`" donde cada pareja, 'RR', 'GG', 'BB', son dos dígitos hexadecimales que varían de '00' a 'FF', definiendo un color posible.⁵

fg Color de la tinta del gráfico. Vea el parámetro `col`.

font Fuente del texto. Si es posible, el 1 corresponde a letra normal, el 2 a negrita, el 3 a cursiva, y el 4 a negrita cursiva.

lwd Anchura de las líneas.

lty Es el tipo de línea. Los tipos de línea se pueden indicar, en su forma elemental, mediante un entero (0=blanco, 1=continuo, 2=guiones, 3=puntos, 4=punto y guión, 5=guión largo, 6=dos guiones) o mediante una cadena de caracteres, respectivamente, de las siguientes: "blank", "solid", "dashed", "dotted", "dotdash", "longdash", o "twodash". El tipo 0 corresponde a no dibujar la línea. Además es posible especificar una cadena de hasta ocho caracteres, tomados de entre "0123456789ABCDEF", para indicar la longitud, en hexadecimal, de los segmentos que se dibujan y saltan, alternativamente, para realizar la línea. Por ejemplo, "44", corresponde a guiones, y "13", a puntos.

new Valor lógico que indica si la siguiente orden gráfica de alto nivel debe comenzar un nuevo gráfico o añadirse al existente.

pch Símbolo con que se dibuja. Puede ser un entero (que indica el número del carácter especial a utilizar) o un carácter (que se utiliza literalmente).

ps Es un entero que indica el tamaño en puntos de los símbolos.

type Tipo de dibujo para representar los puntos. Los valores son:

p puntos

l líneas

b puntos y líneas, sin superponerse

o puntos y líneas, superponiéndose

n nada

s, **S** líneas en escalera, comenzando hacia la derecha o hacia arriba.

h líneas verticales desde el eje X

xlab Cadena de caracteres para etiquetar el eje X.

xlog Si es TRUE, se utiliza una escala logarítmica para el eje X, en caso contrario, se utiliza una escala lineal.

⁴La orden `colores<-colors()`, almacena los nombres en un vector, desde el que podrá utilizarlos fácilmente, como por ejemplo `par(col=colores[552])` para obtener el color rojo (red).

⁵También puede definir un color mediante un número de 0 a 8, para compatibilidad con S.

ylab Cadena de caracteres para etiquetar el eje Y.

ylog Si es TRUE, se utiliza una escala logarítmica para el eje Y, en caso contrario, se utiliza una escala lineal.

Ejemplos

Si escribe, `par(ask=T)`, desde ese momento, cada vez que se genera un gráfico, aparece un cuadro de diálogo antes de borrar el anterior.

Si escribe

```
> win.graph()
> plot(1:10,(1:10)^2)
> win.graph()
> plot(1:10,(1:10)^2,xlab="Eje X",ylab="Y=X^2")
```

el gráfico se genera, en el segundo caso, con nuevas etiquetas en los ejes.

5.12. Desplazamiento entre dispositivos gráficos

Cuando existen varios dispositivos gráficos, es posible saber cual es el activo, cerrar uno concreto (con lo que se termina de lanzar el gráfico correspondiente, si está pendiente) o pasar de uno a otro. Para ello se utilizan las siguientes funciones:

dev.cur devuelve el número y nombre del dispositivo gráfico actual.

dev.list devuelve el número y nombre de todos los dispositivos gráficos activos.

dev.next devuelve el número y nombre del siguiente dispositivo gráfico de la lista de activos.

dev.prev devuelve el número y nombre del anterior dispositivo gráfico de la lista de activos.

dev.set hace que sea activo el dispositivo indicado y, además, devuelve el número y nombre del mismo. Si no se le da valor, cambia al siguiente.

dev.off cierra el dispositivo actual, pasa al siguiente, y devuelve su número.

Siempre hay un dispositivo gráfico abierto con el número 1, *null device*, aunque no sirve para realizar ningún gráfico efectivo.

Ejemplo

Considere las siguientes órdenes:

```
> graphics.off()
> dev.cur()
null device
      1
> x11()
> x11()
> dev.cur()
windows
      3
> dev.next()
```

5.13. Impresión de gráficos

```
windows
  2
> dev.set(dev.next())
windows
  2
> dev.prev()
windows
  3
> dev.set(dev.prev())
windows
  3
> plot(1:10)
```

En primer lugar, cierra cualquier posible dispositivo gráfico abierto. Por tanto, el dispositivo actual será el 1; a continuación abre dos ventanas, por tanto, el dispositivo actual será el 3. El siguiente, será el 2. A continuación hace que sea este el dispositivo actual. El anterior será el 3, y si lo hace actual, la orden última realizará un gráfico sobre el mismo.


5.13. Impresión de gráficos

Para imprimir un gráfico, ya generado, puede utilizar la función `dev.print(device=postscript, ...)` que copia el gráfico del dispositivo actual en el dispositivo creado por la función especificada en `device` y a continuación lo cierra.

Para generar un gráfico directamente sobre el administrador de impresión, puede utilizar la función `win.print`.

5.14. Identificación interactiva

Cuando se ha realizado un gráfico, es posible identificar interactivamente puntos con la función `identify`. Al marcar un punto con el botón primario se escribe en pantalla el valor del identificativo. El botón secundario sale de este modo y la función devuelve los puntos marcados.

```
> plot(h.datos2[,3],h.datos2[,2])
> identify(h.datos2[,3],h.datos2[,2],h.datos2[,4]) 
```

[1] 11 13 9 12 3 1 10 7 14 8

5.15. Función locator

La función `locator` devuelve las coordenadas de una posición de un gráfico. Su sintaxis es

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

El argumento `n` indica el máximo número de puntos que leer, aunque puede finalizar antes pulsando el botón secundario. El argumento `type` permite dibujar utilizando las posiciones seleccionadas, con el mismo significado que en las otras

5.16. Función `barplot`

funciones gráficas, por tanto, su valor predeterminado es no dibujar. La función devuelve las coordenadas de las posiciones seleccionadas, en una lista con dos componentes, `x` e `y`.

Esta función suele utilizarse sin argumentos. Es particularmente útil para seleccionar interactivamente posiciones para elementos gráficos, tales como etiquetas, en que es difícil conocer previamente dónde deben colocarse. Por ejemplo, para colocar un texto informativo en una posición arbitraria de un gráfico, puede utilizar

```
> text(locator(1), "Aquí", adj=0)
```

5.16. Función `barplot`

Esta función genera gráficos de barras en muy diversas formas. Utilice la función `example(barplot)` para hacerse una idea de sus posibilidades. De allí hemos tomado el siguiente ejemplo, que produce el gráfico de la figura 5.13. Se generan 100 números pseudoaleatorios de una distribución de Poisson de parámetro $\lambda = 5$, se realiza un conteo de frecuencias y se representa con barras de diferentes colores.

```
tN <- table(Ni <- rpois(100, lambda=5))  
r <- barplot(tN, col=rainbow(20))
```

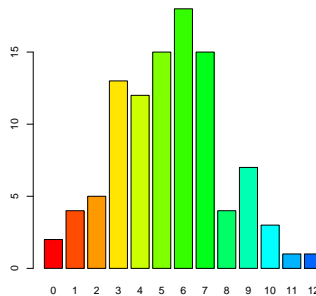


Figura 5.13: Diagrama de barras en una muestra de Poisson

5.17. Función `boxplot`

Esta función realiza gráficos de caja con bigotes (*box-and-whisker*). Utilice la función `example(boxplot)` para hacerse una idea de sus posibilidades.

El siguiente ejemplo produce el gráfico de la figura 5.14.

```
attach(hoja)  
boxplot(Peso[Sexo=="M"], Peso[Sexo=="H"],  
notch=T, names=c("Mujer", "Hombre"),  
ylab="Peso", col=c("blue", "red"))
```

5.18. Función `pairs`

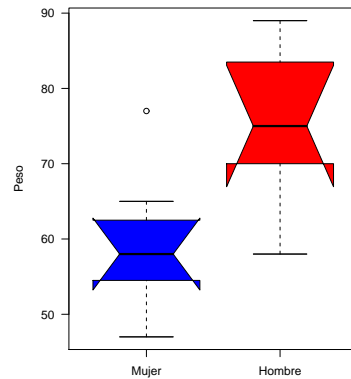


Figura 5.14: Caja con bigotes con melladuras

5.18. Función `pairs`

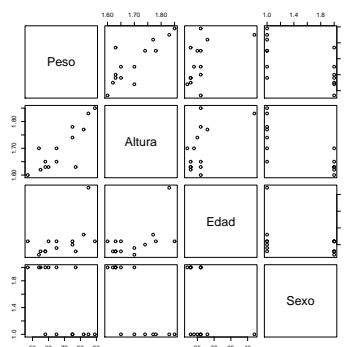
Esta función, genérica, crea una figura que contiene todos los diagramas de dispersión de cada variable frente a las restantes. La función está definida para las clases `matrix`, `data.frame` y `formula`. La sintaxis es

```
pairs(x, (x, labels = colnames(x), panel = points, ...))
```

donde `x` es un objeto, `labels` son caracteres que etiquetan a las variables, y `panel` es `'function(x,y,...)'`, que es la función que se utiliza para representar cada panel del gráfico. También pueden darse parámetros gráficos generales. Es posible referirse a la diagonal, el triángulo inferior y el superior del conjunto de paneles mediante `diag.panel`, `lower.panel` y `upper.panel` respectivamente.

Un ejemplo de utilización es `pairs(h.datos2)`, que produce la figura 5.15.

Figura 5.15: `pairs(h.datos2)`



Puesto que puede definir la función de representación, puede construir gráficos con modificaciones a partir de esta función, como se indica en la ayuda de la función. De este modo, las órdenes siguientes permiten representaciones como

5.18. Función pairs

las de las figuras 5.16 y 5.17.

Figura 5.16: pairs con histogramas

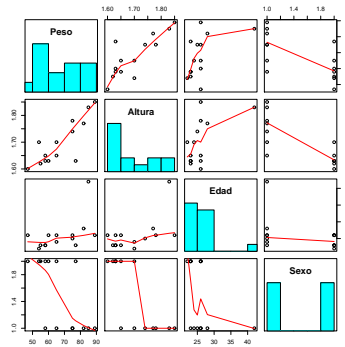
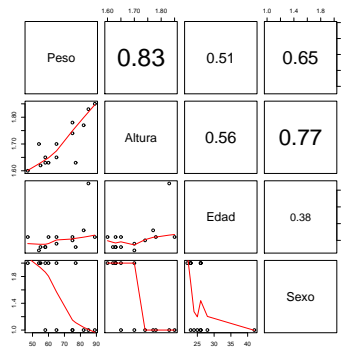


Figura 5.17: pairs con correlaciones



```
> ## incluye histogramas en la diagonal
> panel.hist <- function(x, ...)
> {
>     usr <- par("usr"); on.exit(par(usr))
>     par(usr = c(usr[1:2], 0, 1.5) )
>     h <- hist(x, plot = FALSE)
>     breaks <- h$breaks; nB <- length(breaks)
>     y <- h$counts; y <- y/max(y)
>     rect(breaks[-nB], 0, breaks[-1], y, col="cyan", ...)
> }
> pairs(h.datos2[1:4], panel=panel.smooth,
+       diag.panel=panel.hist, cex.labels=1.5,
+       font.labels=2)
> ## escribe correlaciones absolutas en los
> ## paneles superiores, de tamaño proporcional
> ## a los valores.
```



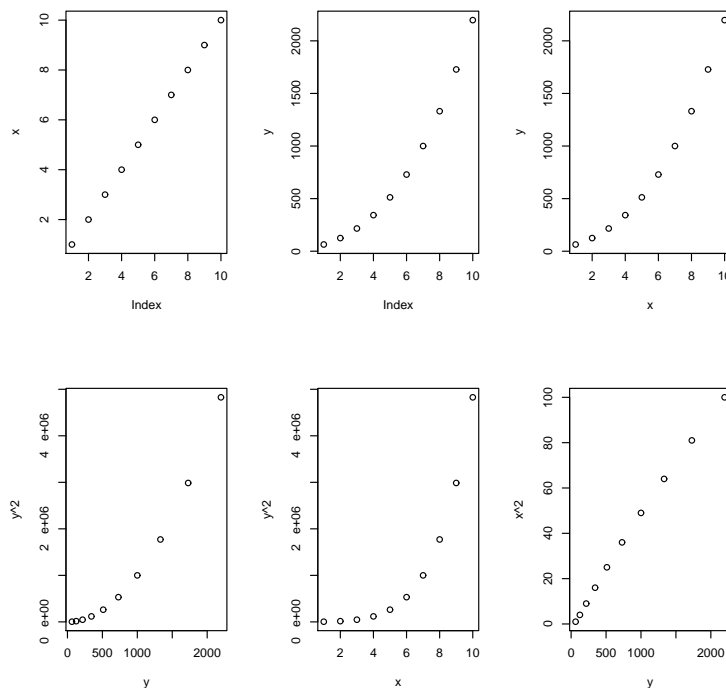
```
> panel.cor <- function(x, y, digits=2, prefix="", cex.cor)
> {
>   usr <- par("usr"); on.exit(par(usr))
>   par(usr = c(0, 1, 0, 1))
>   r <- abs(cor(x, y))
>   txt <- format(c(r, 0.123456789), digits=digits)[1]
>   txt <- paste(prefix, txt, sep="")
>   if(missing(cex.cor)) cex <- 0.8/strwidth(txt)
>   text(0.5, 0.5, txt, cex = cex * r)
> }
> pairs(h.datos2, lower.panel=panel.smooth,
+       upper.panel=panel.cor)
```

5.19. Gráficos múltiples

Entre los argumentos de la función `par` hay algunos que permiten presentar gráficos múltiples. Así, el argumento `mfcol=c(m,n)`, divide el dispositivo gráfico en $m \times n$ partes iguales, que se rellenan por columnas. Significado análogo tiene `mfrow=c(m,n)`, aunque en este caso se rellenan por filas.

Si escribe las órdenes siguientes, el dispositivo gráfico actual se dividirá en seis partes y se realizarán seis gráficos, en el orden que se indica, por filas, como se refleja en la figura 5.18.

Figura 5.18: Gráfico múltiple



```
> par(mfrow=c(2,3))
> x<-(1:10)
> y<-(4:13)^3
> plot(x)
> plot(y)
> plot(x,y)
> plot(y,y^2)
> plot(x,y^2)
> plot(y,x^2)
```

Si desea rellenarlo por columnas tendría que sustituir la primera orden del ejemplo anterior por

```
par(mfcol=c(2,3))
```

5.20. Funciones de uso de pantalla

Una forma versátil de dividir un dispositivo gráfico en partes es utilizar la función `split.screen`, que divide el dispositivo gráfico en partes, que pueden, hasta cierto punto, ser tratadas como dispositivos gráficos distintos. Estas partes pueden a su vez dividirse, lo que permite crear gráficos complejos. Las funciones `screen`, `erase.screen` y `close.screen`, permiten, respectivamente, seleccionar la pantalla actual, borrarla o cerrarla. La sintaxis de estas funciones es

```
split.screen(figs, screen =, erase = TRUE)
screen(n =, new = TRUE)
erase.screen(n =)
close.screen(n =, all = TRUE)
```

donde los parámetros se interpretan así:

figs es, bien un vector de la forma `c(filas,columnas)` que indica la división inicial de la pantalla, bien una matriz $N \times 4$, donde N es el número de pantallas y cada fila especifica la posición de la pantalla correspondiente dentro del dispositivo gráfico en términos de coordenadas relativas, considerando que el dispositivo total se representa por el vector `c(0,1,0,1)`. Estas coordenadas se dan así: las dos primeras componentes representan las dos abscisas (izquierda y derecha) y las dos segundas las dos ordenadas (inferior y superior).

n es el número de pantalla, utilizado tanto por `screen`, `erase.screen`, como `close.screen`. El valor predeterminado es el correspondiente a la pantalla activa.

screen es el número de la pantalla que se va a dividir. El valor predeterminado es la pantalla activa, que inicialmente es el dispositivo completo, numerado como 0.

erase valor lógico que indica si debe borrarse la pantalla en que se va a dibujar.

new valor lógico que indica si debe borrarse la pantalla.

all valor lógico que indica si deben cerrarse todas las pantallas.

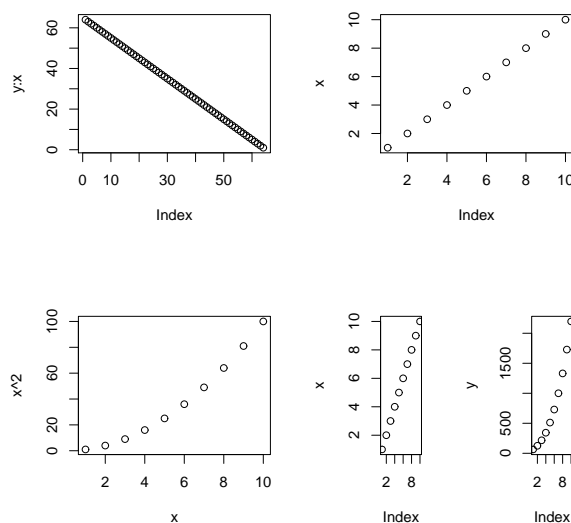
Ejemplos

Las siguientes órdenes dividen el dispositivo gráfico en 3×2 partes, numeradas por filas de 1 a 6. La primera función `plot` dibuja en la parte 1. La segunda dibuja en la misma parte y, por tanto, borra el gráfico anterior y además limpia todo el dispositivo gráfico. La función `screen` confirma que el gráfico se sigue produciendo en esa parte. El resto de parejas, `screen`, `plot`, seleccionan diferentes partes y realizan un gráfico en ellas.

```
> x<-(1:10)
> y<-(4:13)^3
> split.screen(figs=c(3,2))
[1] 1 2 3 4 5 6
> plot(y:x)
> plot(x)
> screen()
[1] 1
> screen(2)
> plot(x)
> screen(6)
> plot(x)
```

A continuación escribimos las siguientes órdenes, que producen como resultado que la pantalla se divide en cuatro partes, numeradas de 1 a 4 y que la cuarta se subdivide en dos partes, numeradas de 5 a 6, que podemos referir individualmente e incluso cerrar alguna de ellas. El resultado se muestra en la figura 5.19.

Figura 5.19: Subdivisiones múltiples



```
> x<-(1:10)
> y<-(4:13)^3
> graphics.off()
> x11()
> split.screen(figs=c(2,2))
> screen(4)
> split.screen(figs=c(1,2))
> screen(1)
> plot(y:x)
> screen(2)
> plot(x)
> screen(5)
> plot(x)
> screen(6)
> plot(y)
> screen(3)
> plot(x,x^2)
```

5.21. Diagrama de sectores

La función `pie` permite crear un diagrama de sectores. Su sintaxis es

```
pie(x, labels=names(x), shadow=FALSE,
    edges=200, radius=0.8, fill=NULL, main=NULL, ...)
```

siendo los parámetros los siguientes:

x es un vector de valores proporcionales al tamaño de cada sector.

labels es un vector de etiquetas de los sectores.

shadow es un vector lógico que indica qué efecto de sombreado debe aplicarse a un gráfico con colores.

edges número de vértices de un polígono que aproxima al círculo.

radius radio del círculo respecto del tamaño del gráfico.

col vector de colores para cada sector.

main título del gráfico.

... parámetros gráficos adicionales de la función `par`.

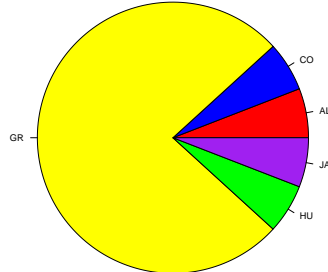
Ejemplos

Considere el siguiente vector de caracteres que representa las provincias de nacimiento de las personas de **h.datos2**. Puede realizar un diagrama de sectores mediante la función `pie` aplicada al conteo de frecuencias⁶ y obtendrá la figura 5.20.

⁶Función `table`.

```
> graphics.off()
> x11()
> opar<- par(mai=c(0,0,0.2,0))
> Provincia<-c("GR","CO","GR","GR","HU","GR","GR",
+ "GR","AL","GR","GR","GR","GR","GR","JA","GR","GR")
> Provincia
[1] "GR" "CO" "GR" "GR" "HU" "GR"
[7] "GR" "GR" "AL" "GR" "GR" "GR"
[13] "GR" "GR" "JA" "GR" "GR"
> table(Provincia)
Provincia
AL CO GR HU JA
 1  1 13  1  1
> pie(table(Provincia),
      col=c("red","blue","yellow","green","purple"))
> title("Provincias de nacimiento")
> par <- opar
```

Figura 5.20: Diagrama de sectores
Provincias de nacimiento



5.22. Diagramas de estrella

La función `stars` realiza un diagrama de estrellas, una por individuo, con información de todas las variables. La sintaxis es

```
stars(x, full = TRUE, scale = TRUE, radius = TRUE,
      labels = dimnames(x)[[1]], locations = NULL,
      xlimit = NULL, ylimit = NULL, len = 1, colors = NULL,
      key.loc = NULL, key.labels = NULL,
      draw.segments = FALSE, draw.axes = FALSE, ...)
```

cuyos argumentos más importantes tienen el siguiente significado:

`x` es una matriz de datos.

5.22. Diagramas de estrella

full valor lógico que indica si los símbolos deben ocupar un círculo completo o sólo el semicírculo superior.

scale valor lógico que indica si las columnas deben homogeneizarse al intervalo (0,1).

radius valor lógico que indica si deben dibujarse los radios.

labels etiquetas de los dibujos. El valor predeterminado es el nombre del individuo.

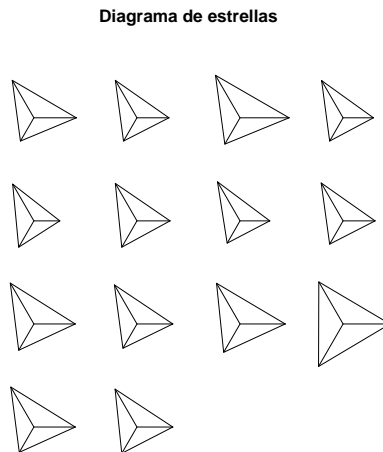
... parámetros gráficos adicionales de la función **par**.

Ejemplos

Las siguientes órdenes crean dos diagramas de estrella, recogidos en los gráficos 5.21 y 5.22.

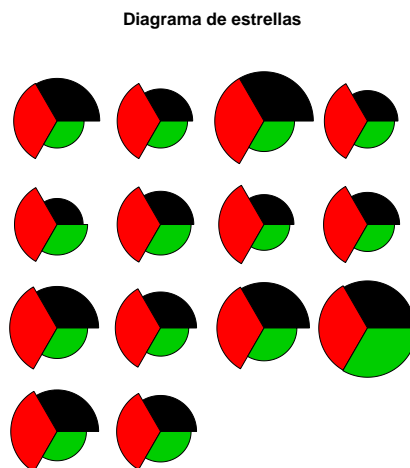
```
> attach(h.datos2)
> stars(matrix(c(Peso,Altura,Edad),ncol=3),
+   main="Diagrama de estrellas")
> stars(matrix(c(Peso,Altura,Edad),ncol=3),
+   main="Diagrama de estrellas", draw.segments = TRUE)
```

Figura 5.21: Diagrama de estrella



A continuación se define la función **Tarta**, para eliminar la necesidad de realizar el conteo de frecuencias. Para ilustrar la realización de varios gráficos, se repite el gráfico cuatro veces.

Figura 5.22: Diagrama de estrella sectorial



```
> Tarta<-function(var = "")
+ {
+   if(var == "")
+     stop("Necesitas una variable")
+   x11()
+   opar<-par(mfrow = c(2, 2))
+   for(i in 1:(2 * 2))
+     piechart(table(var),
+       col=c("red","blue","yellow","green","purple"))
+   par<-opar
+ }
> Tarta(Provincia)
```

Se puede modificar la función para que represente un número arbitrario de veces el gráfico, así como para comprobar que los parámetros son válidos.

```
> Tarta<-function(var = "",filas=2,columnas=2)
+ {
+   if(var == "")
+     stop("Necesitas una variable")
+   filas <-as.integer(filas)
+   columnas<-as.integer(columnas)
+   if(filas<1) stop("El numero de filas es < 1")
+   if(columnas<1) stop("El numero de columnas es < 1")
+   x11()
+   opar<-par(mfrow = c(filas, columnas))
+   for(i in 1:(filas * columnas))
```

```
+         piechart(table(var),
+         col=c("red","blue","yellow","green","purple"))
+     par<-opar
+ }
> graphics.off()
> Tarta(Provincia,3)
> Tarta(Provincia,3,3)
> Tarta(Provincia,,1)
> Tarta(Provincia,1,1)
> Tarta(Provincia,-1,1)
Error in Tarta(Provincia, -1, 1) : El numero de filas es < 1
```

5.23. Representaciones tridimensionales

Es posible realizar gráficos tridimensionales mediante las funciones **persp**, **contour** e **image**. Las tres utilizan los siguientes parámetros:

x vector de puntos del eje X, en orden ascendente, para los que se hará la representación.

y vector de puntos del eje Y, en orden ascendente, para los que se hará la representación.

z matriz de valores de la función a representar. Si este es el primer valor se calculan **x** e **y**.

La función **persp** crea un gráfico en perspectiva. Su sintaxis es

```
persp(x = seq(0, 1, len = nrow(z)),
      y = seq(0, 1, len = ncol(z)),
      z, xlim = range(x), ylim = range(y),
      zlim = range(z, na.rm = TRUE),
      xlab = NULL, ylab = NULL, zlab = NULL,
      theta = 0, phi = 15, r = sqrt(3), d = 1,
      scale = TRUE, expand = 1, col = NULL, border = NULL,
      ltheta = -135, lphi = 0, shade = NA, box = TRUE,
      axes = TRUE, nticks = 5, ticktype = "simple", ...)
```

Los parámetros más utilizados son:

x, y posiciones de la rejilla definida por puntos del eje X y del eje Y, en orden ascendente, para los que se ha medido el valor de 'z'. El valor predeterminado son valores equiespaciados entre 0 y 1. Si **x** es una lista, deberá tener dos componentes, **x\$x** y **x\$y**, que se utilizarán como valores de *x* e *y* respectivamente.

z matriz de valores de la función a representar.

xlim, ylim, zlim Extremos de *x*, *y* y *z*.

xlab, ylab, zlab Etiquetas de los ejes.

theta, phi Ángulos que definen la dirección de visión.

5.23. Representaciones tridimensionales

r Distancia desde el lugar de punto de vista al centro del gráfico.

box Valor lógico que indica si debe dibujarse la caja que delimita el dibujo. Su valor predeterminado es TRUE.

... parámetros gráficos adicionales.

La función `contour` realiza un mapa de nivel. Su sintaxis es

```
contour(x = seq(0, 1, len = nrow(z)),
        y = seq(0, 1, len = ncol(z)), z,
        nlevels = 10, levels = pretty(zlim, nlevels), labels = NULL,
        xlim = range(x, finite = TRUE),
        ylim = range(y, finite = TRUE),
        zlim = range(z, finite = TRUE),
        labcex = 0.6, drawlabels = TRUE, method = "flattest",
        vfont = c("sans serif", "plain"),
        col = par("fg"), lty = par("lty"), lwd = par("lwd"),
        add = FALSE, ...)
```

Los parámetros más utilizados son:

x, y posiciones de la rejilla definida por puntos del eje X y del eje Y, en orden ascendente, para los que se ha medido el valor de 'z'. El valor predeterminado son valores equiespaciados entre 0 y 1. Si x es una lista, deberá tener dos componentes, `x$x` y `x$y`, que se utilizarán como valores de *x* e *y* respectivamente.

z matriz de valores de la función a representar.

nlevels número de niveles de contorno, sólo en el caso de que no se especifique `levels`.

levels vector numérico de los niveles en que se dibujarán líneas de contorno.

labels vector de etiquetas de las líneas de contorno. Si su valor es 'NULL' se utilizan los propios niveles como etiquetas.

drawlabels Valor lógico que indica si deben etiquetarse las líneas de contorno.

xlim, ylim, zlim Extremos de x, y y z.

col Color de las líneas.

lty Tipo de línea que se dibujará.

lwd Anchura de las líneas.

add Valor lógico que indica si debe añadirse al gráfico actual.

... parámetros gráficos adicionales.

Por último, la función `image` dibuja una representación utilizando un código de color o una escala de grises. La sintaxis es

```
image(x, y, z, zlim=range(z), add=F)
```

zlim vector que contiene el mínimo y el máximo de z.

add valor lógico que indica si la imagen debe añadirse al gráfico actual.

También es posible añadir parámetros gráficos.

Ejemplos

La función **Dibuja** realiza los tres tipos de representación. A continuación la utilizamos con dos funciones diferentes, una definida por su nombre y otra definida directamente.

```
> Seno<-function(x,y) sin(x*y)
> Dibuja <- function(x = NA, y = NA, f = Seno)
+ {
+     close.screen(all = T)
+     win.graph()
+     split.screen(c(1, 3))
+     screen(1)
+     image(x, y, outer(x, y, f))
+     screen(2)
+     contour(x, y, outer(x, y, f))
+     screen(3)
+     persp(x, y, outer(x, y, f))
+ }
> x<- -5:5
> Dibuja(x,x)
> Dibuja(x,x,function(x,y) sin(x)*cos(y))
```

Capítulo 6

Fórmulas y Modelos

Los modelos son objetos que imitan las propiedades de los objetos reales en una forma más simple y conveniente. Con los modelos se realizan inferencias que se aplican a los objetos reales. Las diferencias entre los modelos y la realidad se denominan residuos y son una parte fundamental. Así, un mapa de carreteras es un modelo de una parte de la superficie terrestre que intenta imitar la posición relativa de las ciudades, carreteras y otros aspectos. Un buen modelo reproduce de modo preciso propiedades relevantes del objeto real al tiempo que es sencillo de utilizar.

6.1. Fórmulas

Una fórmula es una expresión simbólica de la estructura de un modelo y es utilizada por las funciones de ajuste para estimar el modelo concreto. Las fórmulas fueron introducidas por Wilkinson y Rogers [6] en 1973 y aquí se utilizan con algunas ampliaciones. Una fórmula está constituida por dos partes relacionadas: La respuesta o variable dependiente y los términos o variables independientes o predictores. Ambas partes se relacionan a través de la función `~` que, en forma de operador, permite crear un objeto de tipo `formula`. Es posible que la respuesta no esté incluida. Las expresiones que aparecen en una fórmula se interpretan como otras expresiones cualquiera excepto para los siguientes operadores:

`+` `-` `*` `/` `:` `%in%` `^`

cuyo significado iremos viendo a continuación.

Expresión	Significado
$T \sim F$	T se modeliza como F
$F_a + F_b$	Incluye ambos elementos
$F_a - F_b$	Incluye F_a excepto F_b
$F_a * F_b$	$F_a + F_b + F_a : F_b$
F_a / F_b	$F_a + F_b \%in\% (F_a)$
$F_a : F_b$	Interacciones
$F_a \%in\% F_b$	Interacciones
$F \sim m$	Todos los términos de F cruzados hasta el orden m

La siguiente, es un ejemplo de fórmula:

`Peso ~ Altura + Edad.`

6.1. Fórmulas

que es un modelo del **Peso** como combinación lineal de la **Altura** y la **Edad**, y que representa el modelo

$$\text{Peso} = \beta_0 + \beta_1 \text{Altura} + \beta_2 \text{Edad} + \varepsilon$$

donde las tres variables son vectores numéricos.

El operador $+$ permite combinar los términos. Estos no se limitan a nombres, sino que pueden ser cualquier expresión que, cuando se evalúe, se interpretará como una variable. Si queremos expresar el modelo anterior de dependencia, pero entre logaritmos de las variables, bastaría escribir

$$\log(\text{Peso}) \sim \log(\text{Altura}) + \log(\text{Edad}).$$

De hecho los términos de una fórmula pueden ser:

- un vector numérico, al que corresponde un coeficiente,
- un factor, al que corresponde un coeficiente por nivel,
- una matriz, a la que corresponde un coeficiente por columna, y
- cualquier expresión que, al evaluarse, corresponda a uno de los tres tipos anteriores.

Si alguna de las variables que utilizamos es un factor, su interpretación es distinta de la de un vector numérico. Así, el modelo

$$\text{Peso} \sim \text{Altura} + \text{Sexo}.$$

es la forma reducida de escribir el modelo

$$\text{Peso}_i = \mu + \beta \text{Altura}_i + \left\{ \begin{array}{ll} \alpha_H & \text{si } \text{Sexo} = H \\ \alpha_M & \text{si } \text{Sexo} = M \end{array} \right\} + \varepsilon_i$$

siendo α_H y α_M dos parámetros que representan los dos niveles de la variable **Sexo**. El modelo es equivalente a crear dos variables ficticias, por ejemplo, **Hombre**, que vale 1 en los hombres y 0 en las mujeres, y **Mujer**, con valores opuestos, y considerar el modelo

$$\text{Peso} \sim \text{Altura} + \text{Hombre} + \text{Mujer}.$$

A menudo en este tipo de modelos no pueden determinarse todos los coeficientes de modo único. Por ejemplo, en este caso, puede modificarse arbitrariamente μ mediante un factor aditivo, sin más que restarlo a los valores de α .

Si una variable lógica forma parte de un modelo, se considera un factor con niveles **TRUE** y **FALSE**. Una variable de caracteres también se interpreta como un factor cuyos niveles son los diferentes valores.

También es posible utilizar una matriz, entendiéndose en ese caso que cada una de sus columnas forma parte del modelo, aunque la matriz completa se considera un solo término.

El operador $:$ permite expresar interacciones entre dos o más términos, como ocurre cuando el efecto de una variable en el modelo puede ser diferente según el nivel de un factor. Los casos que se pueden dar son tres:

6.1. Fórmulas

factor:factor representa un término de la forma γ_{ij} que es un conjunto de $I \times J$ constantes, una para cada pareja de combinaciones de los factores,

factor:numeric representa un término de la forma $\beta_j x$ que se interpreta como una pendiente diferente para cada valor del factor, y

numeric:numeric representa un término de la forma βxy .

Así, si las variables **Sexo** y **Provincia** pueden interactuar, la fórmula

Sexo + Provincia + Sexo:Provincia.

expresa que se desean ajustar coeficientes para cada nivel de **Sexo** y **Provincia** y para cada una de sus combinaciones. El operador ***** permite escribir este tipo de fórmula más abreviadamente, así

Sexo * Provincia.

que se expande a

1 + Sexo + Provincia + Sexo:Provincia.

ya que 1 se incluye por defecto en cualquier fórmula, salvo que se elimine expresamente.

Si un modelo incluye términos anidados, como podría ser el caso de las variables **Comarca** y **Provincia**, puede escribirse la fórmula

1 + Provincia + Comarca %in% Provincia

que puede escribirse resumidamente como

Provincia / Comarca

El operador **-** se utiliza para eliminar términos, lo que permite escribir de forma más compacta un modelo complejo en el que queremos eliminar un sólo término.

Por ejemplo,

A * B * C - A:B:C

correspondería al modelo

A + B + C + A:B + A:C + B:C

y del mismo modo

Peso ~ Altura - 1

define un modelo sin término independiente.

La función **model.matrix** permite construir la matriz del diseño correspondiente a un modelo concreto, aunque también podemos trabajar directamente con esta matriz si el diseño es complejo y no sabemos expresarlo mediante una fórmula.

Si une una fórmula a unos datos concretos, obtendrá un modelo que puede ajustar.

6.2. La función `lm`

Esta función ajusta modelos lineales. La sintaxis es

```
lm(formula, data, subset, weights, na.action,  
   method = "qr", model = TRUE, x = FALSE, y = FALSE, qr = TRUE,  
   singular.ok = TRUE, contrasts = NULL, offset, ...)
```

cuyos argumentos son

formula es un objeto de la clase `formula` que describe el modelo que se desea ajustar.

data es una hoja de datos que contiene las variables que se utilizan en el modelo. Si no se especifica o no se encuentran las variables, se toman del entorno desde el que se llama a la función.

subset Es un vector opcional que indica el subconjunto de datos que se utilizará.

weights Es un vector opcional que indica los pesos de cada variable que se utilizará en el modelo. Si no se especifica se usa mínimos cuadrados ordinarios.

na.action es una función que indica qué decisión tomar cuando hay datos faltantes.

method indica el método que se utilizará. Para ajuste sólo puede especificarse `qr`.

model, **x**, **y**, **qr** son variables lógicas que indican si se desea o no obtener como resultado, respectivamente, el marco del modelo, la matriz del modelo, la respuesta y la descomposición QR.

singular.ok es una variable lógica que indica si puede aceptarse un ajuste singular o debe considerarse un error.

contrasts es una lista opcional de contrastes.

offset especifica componentes conocidas a priori que deben incluirse durante el ajuste.

lm devuelve un objeto de tipo `lm` y, en el caso de respuestas múltiples, uno de tipo `c(mlm, lm)`. Es posible aplicar las funciones `summary` y `anova` a estos objetos para obtener un resumen o un análisis de varianza.

Bibliografía

- [1] Becker R.A., Chambers J.M., Wilks A.R. (1988) *The New S Language: A Programming Environment for Data Analysis and Graphics*, Wadsworth & Brooks/Cole.
- [2] Chambers J.M., Hastie T.J. (1992) *Statistical models in S*, Wadsworth & Brooks/Cole.
- [3] Crawley (2012), *The R Book, 2nd ed.*, Wiley
- [4] *Google.com*
- [5] Steel G.L., Sussman G.J. (1975), *Scheme: An Interpreter for the Extended Lambda Calculus*, Memo 349, MIT Artificial Intelligence Laboratory.
- [6] Wilkinson, G.N. Rogers, C.E. (1973) Symbolic description of factorial models for analysis of variance *Applied Statistics*, **22**, 392-399.
- [7] Xie (2015) *Dynamic Documents with R and knitr, 2nd ed*, CRC Press