Introducción a R

Notas sobre R: Un entorno de programación para Análisis de Datos y Gráficos
Versión 1.0.1 (2000-05-16)

R Development Core Team

Índice General

Prólogo

Estas notas sobre R están escritas a partir de un conjunto de notas que describían los entornos S y S-PLUS escritas por Bill Venables y Dave Smith. Hemos realizado un pequeño número de cambios para reflejar las diferencias entre R y S.

R es un proyecto vivo y sus capacidades no coinciden totalmente con las de S. En estas notas hemos adoptado la convención de que cualquier característica que se vaya a implementar se especifica como tal en el comienzo de la sección en que la característica es descrita. Los usuarios pueden contribuir al proyecto implementando cualquiera de ellas.

Deseamos dar las gracias más efusivas a Bill Venables por permitir la distribución de esta versión modificada de las notas y por ser un defensor de R desde su inicio.

Cualquier comentario o corrección serán siempre bienvenidos. Dirija cualquier correspondencia a `R-core@r-project.org`.

Sugerencias al lector

La primera relación con R debería ser la sesión inicial del Apendice A [Ejemplo de sesion], página 84. Está escrita para que se pueda conseguir cierta familiaridad con el estilo de las sesiones de R y para comprobar que coincide con la versión actual.

Muchos usuarios eligen R fundamentalmente por sus capacidades gráficas. Si ese es su caso, debería leer antes o después el Capítulo 12 [Graficos], página 69, sobre capacidades gráficas y para ello no es necesario esperar a haber asimilado totalmente las secciones precedentes.

1 Introducción y Preliminares

1.1 El entorno R

R es un conjunto integrado de programas para manipulación de datos, cálculo y gráficos. Entre otras características dispone de:

- almacenamiento y manipulación efectiva de datos,
- operadores para cálculo sobre variables indexadas (Arrays), en particular matrices,
- una amplia, coherente e integrada colección de herramientas para análisis de datos,
- posibilidades gráficas para análisis de datos, que funcionan directamente sobre pantalla o impresora, y
- un lenguaje de programación bien desarrollado, simple y efectivo, que incluye condicionales, ciclos, funciones recursivas y posibilidad de entradas y salidas. (Debe destacarse que muchas de las funciones suministradas con el sistema están escritas en el lenguaje R)

El término "entorno" lo caracteriza como un sistema completamente diseñado y coherente, antes que como una agregación incremental de herramientas muy específicas e inflexibles, como ocurre frecuentemente con otros programas de análisis de datos.

R es en gran parte un vehículo para el desarrollo de nuevos métodos de análisis interactivo de datos. Como tal es muy dinámico y las diferentes versiones no siempre son totalmente compatibles con las anteriores. Algunos usuarios prefieren los cambios debido a los nuevos métodos y tecnología que los acompañan, a otros sin embargo les molesta ya que algún código anterior deja de funcionar. Aunque R puede entenderse como un lenguaje de programación, los programas escritos en R deben considerarse esencialmente efímeros.

1.2 Programas relacionados. Documentación

R puede definirse como una nueva implementación del lenguaje S desarrollado en AT&T por Rick Becker, John Chambers y Allan Wilks. Muchos de los libros y manuales sobre S son útiles para R.

La referencia básica es *The New S Language: A Programming Environment for Data Analysis and Graphics* de Richard A. Becker, John M. Chambers and Allan R. Wilks. Las características de la versión de agosto de 1991 de S están recogidas en *Statistical Models in S* editado por John M. Chambers y Trevor J. Hastie. Véase Apendice F [Referencias], página 100, para referencias concretas.

1.3 Estadística con R

En la introducción a R no se ha mencionado la palabra *estadística*, sin embargo muchas personas utilizan R como un sistema estadístico. Nosotros preferimos describirlo como un entorno en el que se han implementado muchas técnicas estadísticas, tanto clásicas como modernas. Algunas están incluidas en el entorno base de R y otras se acompañan en forma de *bibliotecas* (packages). El hecho de distinguir entre ambos conceptos es fundamentalmente una cuestión histórica. Junto con R se incluyen ocho bibliotecas (llamadas

bibliotecas estándar) pero otras muchas están disponibles a través de Internet en CRAN (http://www.r-project.org).

Como hemos indicado, muchas técnicas estadísticas, desde las clásicas hasta la última metodología, están disponibles en R, pero los usuarios necesitarán estar dispuestos a trabajar un poco para poder encontrarlas.

Existe una diferencia fundamental en la filosofía que subyace en R (o S) y la de otros sistemas estadísticos. En R, un análisis estadístico se realiza en una serie de pasos, con unos resultados intermedios que se van almacenando en objetos, para ser observados o analizados posteriormente, produciendo unas salidas mínimas. Sin embargo en SAS o SPSS se obtendría de modo inmediato una salida copiosa para cualquier análisis, por ejemplo, una regresión o un análisis discriminante.

1.4 R en un sistema de ventanas

La forma más conveniente de usar R es en una estación de trabajo con un sistema de ventanas. Estas notas están escritas pensando en usuarios de estas características. En particular nos referiremos ocasionalmente a la utilización de R en un sistema X-window, aunque normalmente se pueden aplicar a cualquier implementación del entorno R.

Muchos usuarios encontrarán necesario interactuar directamente con el sistema operativo de su ordenador de vez en cuando. En estas notas se trata fundamentalmente de la interacción con el sistema operativo UNIX. Si utiliza R bajo Microsoft Windows necesitará realizar algunos pequeños cambios.

El ajuste del programa para obtener el máximo rendimiento de las cualidades parametrizables de R es una tarea interesante aunque tediosa y no se considerará en estas notas. Si tiene dificultades busque a un experto cercano a usted.

1.5 Utilización interactiva de R

Cuando R espera la entrada de órdenes, presenta un símbolo para indicarlo. El símbolo predeterminado es '`>`', que en UNIX puede coincidir con el símbolo del sistema, por lo que puede parecer que no sucede nada. Si ese es su caso, sepa que es posible modificar este símbolo en R. En estas notas supondremos que el símbolo de UNIX es '`$`'.

Para utilizar R bajo UNIX por primera vez, el procedimiento recomendado es el siguiente:

1. Cree un subdirectorio, por ejemplo '`trabajo`', en el que almacenar los archivos de datos que desee analizar mediante R. Éste será el directorio de trabajo cada vez que utilice R para este problema concreto.

   ```
   $ mkdir trabajo
   $ cd trabajo
   ```

2. Inicie R con la orden

   ```
   $ R
   ```

3. Ahora puede escribir órdenes para R (como se hace más adelante).
4. Para salir de R la orden es

   ```
   > q()
   ```

R preguntará si desea salvar los datos de esta sesión de trabajo. Puede responder *yes* (Si), *no* (No) o *cancel* (cancelar) pulsando respectivamente las letras *y*, *n* o *c*, en cada uno de cuyos casos, respectivamente, salvará los datos antes de terminar, terminará sin salvar, o volverá a la sesión de R. Los datos que se salvan estarán disponibles en la siguiente sesión de R.

Volver a trabajar con R es sencillo:

1. Haga que 'trabajo' sea su directorio de trabajo e inicie el programa como antes:
```
$ cd trabajo
$ R
```
2. Dé las órdenes que estime convenientes a R y termine la sesión con la orden `q()`.

Bajo Microsoft Windows el procedimiento a seguir es básicamente el mismo: Cree una carpeta o directorio. Ejecute R haciendo doble click en el icono correspondiente. Seleccione **New** dentro del menú **File** para indicar que desea iniciar un nuevo problema (lo que eliminará todos los objetos definidos dentro del espacio de trabajo) y a continuación seleccione **Save** dentro del menú `File` para salvar esta imagen en el directorio que acaba de crear. Puede comenzar ahora los análisis y cuando salga de R, éste le preguntará si desea salvar la imagen en el directorio de trabajo.

Para continuar con este análisis posteriormente basta con pulsar en el icono de la imagen salvada, o bien puede ejecutar R y utilizar la opción **Open** dentro del menú **File** para seleccionar y abrir la imagen salvada.

1.6 Una sesión inicial

Se recomienda a los lectores que deseen ver cómo funciona R que realicen la sesión inicial dada en el Apendice A [Ejemplo de sesion], página 84.

1.7 Ayuda sobre funciones y capacidades

R contiene una ayuda similar a la orden `man` de UNIX. Para obtener información sobre una función concreta, por ejemplo `solve`, la orden es

```
> help(solve)
```

Una forma alternativa es

```
> ?solve
```

Con las funciones especificadas por caracteres especiales, el argumento deberá ir entre comillas, para transformarlo en una "cadena de caracteres":

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

Podrá utilizar tanto comillas simples como dobles, y cada una de ellas puede utilizarse dentro de la otra, como en la frase `"Dijo 'Hola y adiós' y se marchó"`. En estas notas utilizaremos dobles comillas.

En muchas versiones de R puede acceder a la ayuda escrita en formato HTML, escribiendo

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

que ejecuta un lector Web (`Netscape` en UNIX) para leer estas páginas como hipertextos. En UNIX, las peticiones de ayuda posteriores se envirán al sistema de ayuda basado en HTML.

Las versiones de Microsoft Windows de R poseen otros sistemas de ayuda opcionales. Utilice

```
> ?help
```

para obtener detalles adicionales.

1.8 Órdenes de R. Mayúsculas y minúsculas

Técnicamente hablando, R es un *lenguaje de expresiones* con una sintaxis muy simple. Consecuente con sus orígenes en UNIX, distingue entre mayúsculas y minúsculas, de tal modo que `A` y `a` son símbolos distintos y se referirán, por tanto, a objetos distintos.

Las órdenes elementales consisten en *expresiones* o en *asignaciones*. Si una orden consiste en una expresión, se evalúa, se imprime y su valor se pierde. Una asignación, por el contrario, evalúa una expresión, no la imprime y guarda su valor en una variable.

Las órdenes se separan mediante punto y coma, (‘`;`’), o mediante un cambio de línea. Si al terminar la línea, la orden no está sintácticamente completa, R mostrará un signo de continuación, por ejemplo

```
+
```

en la línea siguiente y las sucesivas y continuará leyendo hasta que la orden esté sintácticamente completa. El signo de continuación puede ser modificado fácilmente. En estas notas omitiremos generalmente el símbolo de continuación y lo indicaremos mediante un sangrado.

1.9 Recuperación y corrección de órdenes previas

Bajo muchas versiones de UNIX, R permite recuperar y ejecutar órdenes previas. Las flechas verticales del teclado puede utilizarse para recorrer el *historial de órdenes*. Cuando haya recuperado una orden con este procedimiento, puede utilizar las flechas horizontales para desplazarse por ella, puede eliminar caracteres con la tecla ⟨DEL⟩, o añadir nuevos caracteres. Más adelante se darán mayores detalles, véase Apendice C [El editor de ordenes], página 93.

La recuperación y edición en UNIX pueden ser fácilmente adaptadas. Para ello debe buscar en el manual de UNIX la información sobre **readline**.

También puede utilizar el editor de textos `emacs` para trabajar más cómodamente de modo interactivo con R, mediante **ESS**.[1]

1.10 Ejecución de órdenes desde un archivo y redirección de la salida

Si tiene órdenes almacenadas en un archivo del sistema operativo, por ejemplo `órdenes.R`, en el directorio de trabajo, `trabajo`, puede ejecutarlas dentro de una sesión de R con la orden

```
> source("órdenes.R")
```

En la versión de Microsoft Windows, **Source** también está disponible dentro del menú **File**. Por otra parte, la orden `sink`, como por ejemplo en

[1] Acrónimo en inglés de *Emacs Speaks Statistics*.

```
> sink("resultado.lis")
```

enviará el resto de la salida, en vez de a la pantalla, al archivo del sistema operativo, `resultado.lis`, dentro del directorio de trabajo. La orden

```
> sink()
```

devuelve la salida de nuevo a la pantalla.

Si utiliza nombres absolutos de archivo en vez de nombres relativos, los resultados se almacenaán en ellos, independientemente del directorio de trabajo.

1.11 Almacenamiento y eliminación de objetos

Las entidades que R crea y manipula se denominan *objetos*. Estos pueden ser de muchos tipos: Variables, Variables indexadas, Cadenas de caracteres, Funciones, etc. Incluso estructuras más complejas construidas a partir de otras más sencillas.

Durante una sesión de trabajo con R los objetos que se crean se almacenan por nombre (Discutiremos este proceso en la siguiente sección). La orden

```
> objects()
```

se puede utilizar para obtener los nombres de los objetos almacenados en R. Esta función es equivalente a la función `ls()`. La colección de objetos almacenados en cada momento se denomina *espacio de trabajo* (workspace).

Para eliminar objetos puede utilizar la orden `rm`, por ejemplo:

```
> rm(x, y, z, tinta, chatarra, temporal, barra)
```

Los objetos creados durante una sesión de R pueden almacenarse en un archivo para su uso posterior. Al finalizar la sesión, R pregunta si desea hacerlo. En caso afirmativo todos los objetos se almacenan en el archivo '`.RData`'[2] en el directorio de trabajo.

En la siguiente ocasión que ejecute R, se recuperarán los objetos de este archivo así como el historial de órdenes.

Es recomendable que utilice un directorio de trabajo diferente para cada problema que analice con R. Es muy común crear objetos con los nombres `x` e `y`, por ejemplo. Estos nombres tienen sentido dentro de un análisis concreto, pero es muy difícil dilucidar su significado cuando se han realizado varios análisis en el mismo directorio.

[2] El punto inicial del nombre de este archivo indica que es *invisible* en UNIX.

2 Cálculos sencillos. Números y vectores

2.1 Vectores (numéricos). Asignación

R utiliza diferentes *estructuras de datos*. La estructura más simple es el *vector*, que es una colección ordenada de números. Para crear un vector, por ejemplo `x`, consistente en cinco números, por ejemplo 10.4, 5.6, 3.1, 6.4 y 21.7, use la orden

```
> x <- c(10.4, 5.6, 3.1, 6.4, 21.7)
```

Esta es una *asignación* en la que se utiliza la *función* `c()` que, en este contexto, puede tener un número arbitrario de vectores como *argumento* y cuyo valor es el vector obtenido mediante la concatenación de todos ellos.[1]

Un número, por sí mismo, se considera un vector de longitud uno.

Advierta que el operador de asignación, ('`<-`'), **no** es el operador habitual, '`=`', que se reserva para otro propósito, sino que consiste en dos caracteres, '`<`' ('menor que') y '`-`' ('guión'), que obligatoriamente deben ir unidos y 'apuntan' hacia el objeto que recibe el valor de la expresión.[2]

La asignación puede realizarse también mediante la función `assign()`. Una forma equivalente de realizar la asignación anterior es

```
> assign("x", c(10.4, 5.6, 3.1, 6.4, 21.7))
```

El operador usual, `<-`, puede interpretarse como una abreviatura de la función `assign()`.

Las asignaciones pueden realizarse también con una flecha apuntando a la derecha, realizando el cambio obvio en la asignación. Por tanto, también podría escribirse

```
> c(10.4, 5.6, 3.1, 6.4, 21.7) -> x
```

Si una expresión se utiliza como una orden por sí misma, su valor se imprime y *se pierde*[3]. Así pues, la orden

```
> 1/x
```

simplemente imprime los inversos de los cinco valores anteriores en la pantalla (por supuesto, el valor de `x` no se modifica).

Si a continuación hace la asignación

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

creará un vector, `y`, con 11 elementos, consistentes en dos copias de `x` con un cero entre ambas.

---

[1] Con argumentos diferentes, por ejemplo `listas`, la acción de `c()` puede ser diferente. Vea Sección 6.2.1 [Concatenacion de listas], página 29.

[2] El símbolo de subrayado, '`_`', es un sinónimo del operador de asignación, pero no aconsejamos su utilización ya que produce un código menos legible.

[3] Aunque, de hecho, se almacena en `.Last.value` hasta que se ejecute otra orden.

2.2 Aritmética vectorial

Los vectores pueden usarse en expresiones aritméticas, en cuyo caso las operaciones se realizan elemento a elemento. Dos vectores que se utilizan en la misma expresión no tienen por qué ser de la misma longitud. Si no lo son, el resultado será un vector de la longitud del más largo, y el más corto será *reciclado*, repitiéndolo tantas veces como sea necesario (puede que no un número exacto de veces) hasta que coincida con el más largo. En particular, cualquier constante será simplemente repetida. De este modo, y siendo `x` e `y` los vectores antes definidos, la orden

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

genera un nuevo vector, `v`, de longitud 11, construido sumando, elemento a elemento, el vector `2*x` repetido 2.2 veces, el vector `y`, y el número `1` repetido 11 veces.

Los operadores aritméticos elementales son los habituales `+`, `-`, `*`, `/` y `^` para elevar a una potencia. Además están disponibles las funciones `log`, `exp`, `sin`, `cos`, `tan`, `sqrt`, bien conocidas. Existen muchas más funciones, entre otras, las siguientes: `max` y `min` que seleccionan respectivamente el mayor y el menor elemento de un vector; `range` cuyo valor es el vector de longitud dos, `c(min(x), max(x))`; `length(x)` que es el número de elementos o longitud de `x`; `sum(x)` que es la suma de todos los elementos de `x`; y `prod(x)` que es el producto de todos ellos.

Dos funciones estadísticas son `mean(x)`, que calcula la media, esto es,

```
sum(x)/length(x)
```

y `var(x)` que calcula la cuasi-varianza, esto es,

```
sum((x-mean(x))^2)/(length(x)-1)
```

Si el argumento de `var()` es una matriz $n \times p$, el resultado es la matriz de cuasi-covarianzas $p \times p$ correspondiente a interpretar las filas como vectores muestrales p-variantes.

Para ordenar un vector dispone de la función `sort(x)` que devuelve un vector del mismo tamaño que `x` con los elementos ordenados en orden creciente. También dispone de `order()` y de `sort.list()`, que produce la permutación del vector que corresponde a la ordenación.

Advierta que `max` y `min` seleccionan el mayor y el menor valor de sus argumentos, incluso aunque estos sean varios vectores. Las funciones *paralelas* `pmax` y `pmin` devuelven un vector (de la misma longitud del argumento más largo) que contiene en cada elemento el mayor y menor elemento de dicha posición de entre todos los vectores de entrada.

En la mayoría de los casos, el usuario no debe preocuparse de si los "números" de un vector numérico son enteros, reales o incluso complejos. Los cálculos se realizan internamente como números de doble precisión, reales o complejos según el caso.

Para trabajar con números complejos, debe indicar explícitamente la parte compleja. Así

```
sqrt(-17)
```

devuelve el resultado `NaN` y un mensaje de advertencia, pero

```
sqrt(-17+0i)
```

realiza correctamente el cálculo de la raíz cuadrada de este número complejo.

2.3 Generación de sucesiones

En R existen varias funciones para generar sucesiones numéricas. Por ejemplo, `1:30` es el vector `c(1,2, ...,29,30)`. El operador 'dos puntos' tiene máxima prioridad en una expresión, así, por ejemplo, `2*1:15` es el vector `c(2,4,6, ...,28,30)`. Escriba `n <- 10` y compare las sucesiones `1:n-1` y `1:(n-1)`.

La forma `30:1` permite construir una sucesión descendente.

La función `seq()` permite generar sucesiones más complejas. Dispone de cinco argumentos, aunque no se utilizan todos simultáneamente. Si se dan los dos primeros indican el comienzo y el final de la sucesión, y si son los únicos argumentos, el resultado coincide con el operador 'dos puntos', esto es, `seq(2,10)` coincide con `2:10`.

Los argumentos de `seq()`, y de muchas funciones de R, pueden darse además de por posición, por nombre, en cuyo caso, el orden en que aparecen es irrelevante. En esta función los dos primeros argumentos se pueden dar por nombre mediante `from=`*valor-inicial* y `to=`*valor-final*; por tanto `seq(1,30)`, `seq(from=1, to=30)` y `seq(to=30, from=1)` son formas equivalentes a `1:30`.

Los dos siguientes argumentos de `seq()` son `by=`*valor* y `length=`*valor*, y especifican el 'paso' y 'longitud' de la sucesión respectivamente. Si no se suministra ninguno, el valor predeterminado es `by=1` y `length` se calcula.

Por ejemplo

```
> seq(-5, 5, by=.2) -> s3
```

genera el vector `c(-5.0, -4.8, -4.6, ..., 4.6, 4.8, 5.0)` y lo almacena en `s3` . Similarmente

```
> s4 <- seq(length=51, from=-5, by=.2)
```

genera los mismos valores y los almacena en `s4`.

El quinto argumento de esta función es `along=`*vector*, y si se usa debe ser el único argumento, ya que crea una sucesión `1, 2, ..., length(`*vector*`)`, o la sucesión vacía si el vector es vacío (lo que puede ocurrir).

Una función relacionada con `seq` es `rep()`, que permite duplicar un objeto de formas diversas. Su forma más sencilla es

```
> s5 <- rep(x, times=5)
```

que coloca cinco copias de `x`, una tras otra, y las almacena en `s5`.

2.4 Vectores lógicos

R no solo maneja vectores numéricos, sino también lógicos. Los elementos de un vector lógico sólo pueden tomar dos valores: `FALSE` (falso) y `TRUE` (verdadero). Estos valores se representan también por `F` y `T`.

Los vectores lógicos aparecen al utilizar *condiciones*. Por ejemplo,

```
> temp <- x > 13
```

almacena en `temp` un vector de la misma longitud de `x` y cuyos valores serán, respectivamente, `T` o `F` de acuerdo a que los elementos de `x` cumplan o no la condición indicada: ser mayores que 13.

Los operadores lógicos son `<` (menor), `<=` (menor o igual), `>` (mayor), `>=` (mayor o igual), `==` (igual), y `!=` (distinto). Además, si `c1` y `c2` son expresiones lógicas, entonces `c1&c2` es su intersección (*"conjunción"*), `c1|c2` es su unión (*"disyunción"*) y `!c1` es la negación de `c1`.

Los vectores lógicos pueden utilizarse en expresiones aritméticas, en cuyo caso se transforman primero en vectores numéricos, de tal modo que `F` se transforma en `0` y `T` en `1`. Sin embargo hay casos en que un vector lógico y su correspondiente numérico no son equivalentes, como puede ver a continuación.

2.5 Valores faltantes

En ocasiones puede que no todas las componentes de un vector sean conocidas. Cuando falta un elemento, lo que se denomina 'valor faltante'[4], se le asigna un valor especial, `NA`[5]. En general, casi cualquier operación donde intervenga un valor `NA` da por resultado `NA`. La justificación es sencilla: Si no se puede especificar completamente la operación, el resultado no podrá ser conocido, y por tanto no estará disponible.

La función `is.na(x)` crea un vector lógico del tamaño de `x` cuyos elementos sólo valdrán `T` si el elemento correspondiente de `x` es `NA`, y `F` en caso contrario.

```
> z <- c(1:3,NA);  ind <- is.na(z)
```

Nótese que la expresión lógica `x == NA` es distinta de `is.na(x)` puesto que `NA` no es realmente un valor, sino un indicador de una cantidad que no está disponible. Por tanto `x == NA` es un vector de la misma longitud de `x` con *todos* sus elementos `NA` puesto que la expresión lógica es incompleta.

Además hay una segunda clase de valores "faltantes", producidos por el cálculo. Son los llamados valores `NaN` [6]. Este tipo de dato no existe en S, y por tanto se confunden con `NA` en S-Plus. Ejemplos de `NaN` son

```
> 0/0
```

o

```
> Inf - Inf
```

En resumen, `is.na(xx)` es `TRUE` *tanto* para los valores `NA` como para los `NaN`. Para diferenciar estos últimos existe la función `is.nan(xx)` que sólo toma el valor `TRUE` para valores `NaN`.

2.6 Vectores de caracteres

Las cadenas de caracteres, o frases, también son utilizadas en R, por ejemplo, para etiquetar gráficos. Una cadena de caracteres se construye escribiendo entre comillas la sucesión de caracteres que la define, por ejemplo, `"Altura"` o `"Resultados de la tercera iteración"`.

Los vectores de caracteres pueden concatenarse en un vector mediante la función `c()`.

---

[4] En la literatura estadística inglesa, "missing value"

[5] Acrónimo en inglés de "Not Available", no disponible.

[6] Acrónimo en inglés de "Not a Number", esto es, "No es un número".

Por otra parte, la función `paste()` une todos los vectores de caracteres que se le suministran y construye una sola cadena de caracteres. También admite argumentos numéricos, que convierte inmediatamente en cadenas de caracteres. En su forma predeterminada, en la cadena final, cada argumento original se separa del siguiente por un espacio en blanco, aunque ello puede cambiarse utilizando el argumento `sep="`*cadena*`"`, que sustituye el espacio en blanco por *cadena*, la cual podría ser incluso vacía.

Por ejemplo,

```
> labs <- paste(c("X","Y"), 1:10, sep="")
```

almacena, en `labs`, el vector de caracteres

```
c("X1", "Y2", "X3", "Y4", "X5", "Y6", "X7", "Y8", "X9", "Y10")
```

Recuerde que al tener `c("X", "Y")` solo dos elementos, deberá repetirse 5 veces para obtener la longitud del vector `1:10`.[7]

2.7 Vectores de índices. Selección y modificación de subvectores

Puede seleccionar un subvector de un vector añadiendo al nombre del mismo un *vector de índices* entre corchetes, `[` y `]`. En general podrá obtener un subvector de cualquier expresión cuyo resultado sea un vector, sin más que añadirle un vector de índices entre corchetes.

Los vectores de índices pueden ser de cuatro tipos distintos:

1. **Un vector lógico.** En este caso el vector de índices debe tener la misma longitud que el vector al que refiere. Sólo se seleccionarán los elementos correspondientes a valores `T` del vector de índices y se omitirá el resto. Por ejemplo,

   ```
   > y <- x[!is.na(x)]
   ```

   almacena en `y` los valores no-faltantes de `x`, en el mismo orden. Si `x` tiene valores faltantes, el vector `y` será más corto que `x`. Análogamente,

   ```
   > (x+1)[(!is.na(x)) & x>0] -> z
   ```

   almacena en `z` los elementos del vector `x+1` para los que el correspondiente elemento de `x` es no-faltante y positivo.

2. **Un vector de números naturales positivos.** En este caso los elementos del vector de índices deben pertenecer al conjunto {1, 2, . . . , `length(x)`}. El resultado es un vector formado por los elementos del vector referido que corresponden a estos índices y en el orden en que aparecen en el vector de índices. El vector de índices puede tener cualquier longitud y el resultado será de esa misma longitud. Por ejemplo, `x[6]` es el sexto elemento de `x`, y

   ```
   > x[1:10]
   ```

   es el vector formado por los diez primeros elementos de `x`, (supuesto que `length(x)` no es menor que `10`). Por otra parte,

[7] `paste(..., collapse=`*ss*`)` permite colapsar los argumentos en una sola cadena de caracteres separándolos mediante *ss*. Además existen otras órdenes de manipulación de caracteres. como `sub` y `substring`. Puede encontrar su descripción en la ayuda del programa.

```
> c("x","y")[rep(c(1,2,2,1), times=4)]
```

crea un vector de caracteres de longitud 16 formado por `"x"`, `"y"`, `"y"`, `"x"` repetido cuatro veces.

3. **Un vector de números naturales negativos.** En este caso, los índices indican los elementos del vector referido que deben *excluirse*. Así pues,

```
> y <- x[-(1:5)]
```

almacena en el vector `y` todos los elementos de `x` excepto los cinco primeros (suponiendo que `x` tiene al menos cinco elementos).

4. **Un vector de caracteres.** Esta opción solo puede realizarse si el vector posee el atributo `names` (nombres) para identificar sus componentes, en cuyo caso se comportará de modo similar al punto 2.

```
> fruta <- c(5, 10, 1, 20)
> names(fruta) <- c("naranja", "plátano", "manzana", "pera")
> postre <- fruta[c("manzana","naranja")]
```

La ventaja en este caso es que los *nombres* son a menudo más fáciles de recordar que los *índices numéricos*.Esta opción es especialmente útil al tratar de la estructura de "hoja de datos" (data frame) que veremos posteriormente.

La variable de almacenamiento también puede ser indexada, en cuyo caso la asignación se realiza *solamente sobre los elementos referidos*. La expresión debe ser de la forma `vector[`*vector_de_índices*`]` ya que la utilización de una expresión arbitraria en vez del nombre de un vector no tiene mucho sentido.

El vector asignado debe ser de la misma longitud que el vector de índices y, en el caso de un vector de índices lógico, debe ser de la misma longitud del vector que indexa. Por ejemplo,

```
> x[is.na(x)] <- 0
```

sustituye cada valor faltante de `x` por un cero. Por otra parte,

```
> y[y < 0] <- -y[y < 0]
```

equivale[8] a

```
> y <- abs(y)
```

2.8 Clases de objetos

Los vectores son el tipo básico de objeto en R, pero existen más tipos que veremos de modo formal posteriormente.

- Las *matrices* o, más generalmente, *variables indexadas* (Arrays) son generalizaciones multidimensionales de los vectores. De hecho, *son* vectores indexados por dos o más índices y que se imprimen de modo especial. Véase Capítulo 5 [Matrices y variables indexadas], página 20.
- Los *factores* sirven para representar datos categóricos. Véase Capítulo 4 [Factores], página 17.

[8] Tenga en cuenta que `abs()` no se comporta correctamente con números complejos, en ellos debería usar `Mod()`.

- Las *listas* son una forma generalizada de vector en las cuales los elementos no tienen por qué ser del mismo tipo y a menudo son a su vez vectores o listas. Las listas permiten devolver los resultados de los cálculos estadísticos de un modo conveniente. Véase Sección 6.1 [Listas], página 28.
- Las *hojas de datos* (data frames) son estructuras similares a una matriz, en que cada columna puede ser de un tipo distinto a las otras. Las hojas de datos son apropiadas para describir 'matrices de datos' donde cada fila representa a un individuo y cada columna una variable, cuyas variables pueden ser numéricas o categóricas. Muchos experimentos se describen muy apropiadamente con hojas de datos: los tratamientos son categóricos pero la respuesta es numérica. Véase Sección 6.3 [Hojas de datos], página 29.
- Las *funciones* son también objetos de R que pueden almacenarse en el espacio de trabajo, lo que permite extender las capacidades de R fácilmente. Véase Capítulo 10 [Escritura de funciones], página 46.

3 Objetos: Modos y atributos

3.1 Atributos intrínsecos: modo y longitud

Las entidades que manipula R se conocen con el nombre de *objetos*. Por ejemplo, los vectores de números, reales o complejos, los vectores lógicos o los vectores de caracteres. Este tipo de objetos se denominan estructuras 'atómicas' puesto que todos sus elementos son del mismo tipo o *modo*, bien sea *numeric*[1] (numérico), *complex* (complejo), *logical* (lógico) o *character* (carácter).

Los elementos de un vector deben ser *todos del mismo modo* y éste será el modo del vector. Esto es, un vector será, en su totalidad, de modo *logical*, *numeric*, *complex* o *character*. La única excepción a esta regla es que cualquiera de ellos puede contener el valor `NA`. Debe tener en cuenta que un vector puede ser vacío, pero pese a ello tendrá un modo. Así, el vector de caracteres vacío aparece como `character(0)` y el vector numérico vacío aparece como `numeric(0)`.

R también maneja objetos denominados *listas* que son del modo *list* (lista) y que consisten en sucesiones de objetos, cada uno de los cuales puede ser de un modo distinto. Las *listas* se denominan estructuras 'recursivas' puesto que sus componentes pueden ser asimismo listas.

Existen otras estructuras recursivas que corresponden a los modos *function* (función) y *expression* (expresión). El modo *function* está formado por las funciones que constituyen R, unidas a las funciones escritas por cada usuario, y que discutiremos más adelante. El modo *expression* corresponde a una parte avanzada de R que no trataremos aquí, excepto en lo mínimo necesario para el tratamiento de *fórmulas* en la descripción de modelos estadísticos.

Con el *modo* de un objeto designamos el tipo básico de sus constituyentes fundamentales. Es un caso especial de un *atributo* de un objeto. Los *atributos* de un objeto suministran información específica sobre el propio objeto. Otro atributo de un objeto es su *longitud*. Las funciones `mode(`*objeto*`)` y `length(`*objeto*`)` se pueden utilizar para obtener el modo y longitud de cualquier estructura.

Por ejemplo, si `z` es un vector complejo de longitud 100, entonces `mode(z)` es la cadena `"complex"`, y `length(z)` es `100`.

R realiza cambios de modo cada vez que se le indica o es necesario (y también en algunas ocasiones en que no parece que no lo es). Por ejemplo, si escribe

```
> z <- 0:9
```

y a continuación escribe

```
> digitos <- as.character(z)
```

el vector `digitos` será el vector de caracteres `("0", "1", "2", ..., "9")`. Si a continuación aplica un nuevo cambio de modo

```
> d <- as.integer(digitos)
```

R reconstruirá el vector numérico de nuevo y, en este caso, `d` y `z` coinciden.[2] Existe una colección completa de funciones de la forma `as.`*lo-que-sea()*, tanto para forzar el cambio de

[1] El modo *numérico* consiste realmente en dos modos distintos, *integer* (entero) y *double* (doble precisión).

[2] En general, al forzar el cambio de numérico a carácter y de nuevo a numérico, no se obtienen los mismos resultados, debido, entre otros, a los errores de redondeo.

modo, como para asignar un atributo a un objeto que carece de él. Es aconsejable consultar la ayuda para familiarizarse con estas funciones.

3.2 Modificación de la longitud de un objeto

Un objeto, aunque esté "vacío", tiene modo. Por ejemplo,

```
> v <- numeric()
```

almacena en `v` una estructura vacía de vector numérico. Del mismo modo, `character()` es un vector de caracteres vacío, y lo mismo ocurre con otros tipos. Una vez creado un objeto con un tamaño cualquiera, pueden añadirse nuevos elementos sin más que asignarlos a un índice que esté fuera del rango previo.Por ejemplo,

```
> v[3] <- 17
```

transforma `v` en un vector de longitud 3, (cuyas dos primeras componentes serán `NA`). Esta regla se aplica a cualquier estructura, siempre que los nuevos elementos sean compatibles con el modo inicial de la estructura.

Este ajuste automático de la longitud de un objeto se utiliza a menudo, por ejemplo en la función `scan()` para entrada de datos. (Véase Sección 7.2 [La funcion scan()], página 34.)

Análogamente, puede reducirse la longitud de un objeto sin más que realizar una nueva asignación. Si, por ejemplo, `alfa` es un objeto de longitud 10, entonces

```
> alfa <- alfa[2 * 1:5]
```

lo transforma en un objeto de longitud 5 formado por los elementos de posición par del objeto inicial.

3.3 Obtención y modificación de atributos

La función `attributes(`*objeto*`)` proporciona una lista de todos los atributos no intrínsecos definidos para el objeto en ese momento. La función `attr(`*objeto*`,` *nombre*`)` puede usarse para seleccionar un atributo específico. Estas funciones no se utilizan habitualmente, sino que se reservan para la creación de un nuevo atributo con fines específicos, por ejemplo, para asociar una fecha de creación o un operador con un objeto de R. Sin embargo, es un concepto muy importante que no debe olvidar.

La asignación o eliminación de atributos de un objeto debe realizarse con precaución, ya que los atributos forman parte del sistema de objetos utilizados en R.

Cuando se utiliza en la parte que recibe la asignación, puede usarse para asociar un nuevo atributo al *objeto* o para cambiar uno existente. Por ejemplo,

```
> attr(z,"dim") <- c(10,10)
```

permite tratar `z` como si fuese una matriz de 10×10.

3.4 Clases de objetos

Cada objeto pertenece a una *clase*, y ello permite utilizar en R programación dirigida a objetos.

Por ejemplo, si un objeto pertenece a la clase `"data.frame"`, se imprimirá de un modo especial; cuando le aplique la función `plot()` ésta mostrará un gráfico de un tipo especial;

y otras funciones genéricas, como `summary()`, producirán un resultado especial; todo ello en función de la pertenencia a dicha clase.

Para eliminar temporalmente los efectos de la clase puede utilizar la función `unclass()`. Por ejemplo, si `invierno` pertenece a la clase `"data.frame"`, entonces

```
> invierno
```

escribe el objeto en la forma de la clase, parecida a una matriz, en tanto que

```
> unclass(invierno)
```

lo imprime como una lista ordinaria. Sólo debe utilizar esta función en situaciones muy concretas, como, por ejemplo, si hace pruebas para comprender el concepto de clase y de función genérica.

Las clases y las funciones genéricas serán tratadas muy brevemente en la Sección 10.9 [Orientacion a objetos], página 54.

4 Factores Nominales y Ordinales

Un *factor* es un vector utilizado para especificar una clasificación discreta de los elementos de otro vector de igual longitud. En R existen factores *nominales* y factores *ordinales*.

4.1 Un ejemplo específico

Suponga que dispone de una muestra de 30 personas de Australia[1] de tal modo que su estado o territorio se especifica mediante un vector de caracteres con las abreviaturas de los mismos:

```
> estado <- c("tas", "sa",  "qld", "nsw", "nsw", "nt",  "wa",  "wa",
              "qld", "vic", "nsw", "vic", "qld", "qld", "sa",  "tas",
              "sa",  "nt",  "wa",  "vic", "qld", "nsw", "nsw", "wa",
              "sa",  "act", "nsw", "vic", "vic", "act")
```

Recuerde que, para un vector de caracteres, la palabra "ordenado" indica que está en orden alfabético.

Un *factor* se crea utilizando la función `factor()`:

```
> FactorEstado <- factor(estado)
```

La función `print()` trata a los factores de un modo distinto al de los vectores ordinarios:

```
> FactorEstado
 [1] tas sa  qld nsw nsw nt  wa  wa  qld vic nsw vic qld qld sa
[16] tas sa  nt  wa  vic qld nsw nsw wa  sa  act nsw vic vic act
Levels:  act nsw nt qld sa tas vic wa
```

Puede utilizar la función `levels()` para ver los niveles de un factor:

```
> levels(FactorEstado)
[1] "act" "nsw" "nt"  "qld" "sa"  "tas" "vic" "wa"
```

4.2 La función `tapply()`. Variables desastradas (ragged arrays)

Como continuación del ejemplo anterior, suponga que disponemos en otro vector de los ingresos de las mismas personas (medidos con unas unidades apropiadas)

```
> ingresos <- c(60, 49, 40, 61, 64, 60, 59, 54, 62, 69, 70, 42, 56,
                61, 61, 61, 58, 51, 48, 65, 49, 49, 41, 48, 52, 46,
                59, 46, 58, 43)
```

Para calcular la media muestral para cada estado podemos usar la función `tapply()`:

```
> MediaIngresos <- tapply(ingresos, FactorEstado, mean)
```

que devuelve el vector de medias con las componentes etiquetadas con los niveles:

[1] Para quienes no conocen la estructura administrativa de Australia, existen ocho estados y territorios en la misma: Australian Capital Territory, New South Wales, Northern Territory, Queensland, South Australia, Tasmania, Victoria, y Western Australia; y sus correspondientes abreviaturas son: act, nsw, nt, qld, sa, tas, vic, y wa.

```
> MediaIngresos
   act    nsw     nt    qld     sa    tas    vic     wa
44.500 57.333 55.500 53.600 55.000 60.500 56.000 52.250
```

La función `tapply()` aplica una función, en este ejemplo la función `mean()`, a cada grupo de componentes del primer argumento, en este ejemplo `ingresos`, definidos por los niveles del segundo argumento, en este ejemplo `FactorEstado`, como si cada grupo fuese un vector por sí solo. El resultado es una estructura cuya longitud es el número de niveles del factor. Puede consultar la ayuda para obtener más detalles.

Suponga que ahora desea calcular las desviaciones típicas de las medias de ingresos por estados. Para ello es necesario escribir una función en R que calcule la desviación típica de un vector. Aunque aún no se ha explicado en este texto cómo escribir funciones[2], puede admitir que existe la función `var()` que calcula la varianza muestral o cuasi-varianza, y que la función buscada puede construirse con la asignación:

```
> StdErr <- function(x) sqrt(var(x)/length(x))
```

Ahora puede calcular los valores buscados mediante

```
> ErrorTipicoIngresos <- tapply(ingresos, FactorEstado, StdErr)
```

con el siguiente resultado:

```
> ErrorTipicoIngresos
     act      nsw       nt      qld       sa      tas      vic       wa
1.500000 4.310195 4.500000 4.106093 2.738613 0.500000 5.244044 2.657536
```

Como ejercicio puede calcular el intervalo de confianza al 95% de la media de ingresos por estados. Para ello puede utilizar la función `tapply()`, la función `length()` para calcular los tamaños muestrales, y la función `qt()` para encontrar los percentiles de las distribuciones t de Student correspondientes.

La función `tapply()` puede utilizarse para aplicar una función a un vector indexado por diferentes categorías simultáneamente. Por ejemplo, para dividir la muestra tanto por el estado como por el sexo. Los elementos del vector se dividirán en grupos correspondientes a las distintas categorías y se aplicará la función a cada uno de dichos grupos. El resultado es una variable indexada etiquetada con los niveles de cada categoría.

La combinación de un vector[3] con un factor para etiquetarlo, es un ejemplo de lo que se llama *variable indexada desastrada* (ragged array) puesto que los tamaños de las subclases son posiblemente irregulares. Cuando estos tamaños son iguales la indexación puede hacerse implícitamente y además más eficientemente, como veremos a continuación.

4.3 Factores ordinales

Los niveles de los factores se almacenan en orden alfabético, o en el orden en que se especificaron en la función `factor` si ello se hizo explícitamente.

A veces existe una ordenación natural en los niveles de un factor, orden que deseamos tener en cuenta en los análisis estadísticos. La función `ordered()` crea este tipo de factores y su uso es idéntico al de la función `factor`. Los factores creados por la función `factor` los denominaremos nominales o simplemente factores cuando no haya lugar a confusión, y los

[2] La escritura de funciones será tratada en Capítulo 10 [Escritura de funciones], página 46.

[3] En general de una variable indexada

creados por la función `ordered()` los denominaremos ordinales. En la mayoría de los casos la única diferencia entre ambos tipos de factores consiste en que los ordinales se imprimen indicando el orden de los niveles. Sin embargo los contrastes generados por los dos tipos de factores al ajustar Modelos lineales, son diferentes.

5 Variables indexadas. Matrices

5.1 Variables indexadas (Arrays)

Una variable indexada (array) es una colección de datos, por ejemplo numéricos, indexada por varios índices. R permite crear y manipular variables indexadas en general y en particular, matrices.

Un vector de dimensiones es un vector de números enteros positivos. Si su longitud es k entonces la variable indexada correspondiente es k–dimensional. Los elementos del vector de dimensiones indican los límites superiores de los k índices. Los límites inferiores siempre valen 1.

Un vector puede transformarse en una variable indexada cuando se asigna un vector de dimensiones al atributo *dim*. Supongamos, por ejemplo, que `z` es un vector de 1500 elementos. La asignación

```
> dim(z) <- c(3,5,100)
```

hace que R considere a z como una variable indexada de dimensión $3 \times 5 \times 100$.

Existen otras funciones, como `matrix()` y `array()`, que permiten asignaciones más sencillas y naturales, como se verá en la Sección 5.4 [La funcion array()], página 22.

Los elementos del vector pasan a formar parte de la variable indexada siguiendo la regla[1] de que el primer índice es el que se mueve más rápido y el último es el más lento.

Por ejemplo, si se define una variable indexada, `a`, con vector de dimensiones `c(3,4,2)`, la variable indexada tendrá $3 \times 4 \times 2 = 24$ elementos que se formarán a partir de los elementos originales en el orden `a[1,1,1], a[2,1,1], ..., a[2,4,2], a[3,4,2]`.

5.2 Elementos de una variable indexada

Un elemento de una variable indexada puede referirse dando el nombre de la variable y, entre corchetes, los índices que lo refieren, separados por comas.

En general, puede referir una parte de una variable indexada mediante una sucesión de *vectores índices*, teniendo en cuenta que *si un vector índice es vacío, equivale a utilizar todo el rango de valores para dicho índice.*

Así, en el ejemplo anterior, `a[2,,]` es una variable indexada 4×2, con vector de dimensión `c(4,2)` y sus elementos son

```
c(a[2,1,1], a[2,2,1], a[2,3,1], a[2,4,1],
  a[2,1,2], a[2,2,2], a[2,3,2], a[2,4,2])
```

en ese orden. A su vez, `a[,,]` equivale a la variable completa, que coincide con omitir completamente los índices y utilizar simplemente `a`.

Para cualquier variable indexada, por ejemplo `Z`, el vector de dimensión puede referirse explícitamente mediante `dim(Z)` (en cualquiera de las dos partes de una asignación).

Asimismo, si especifica una variable indexada con *un solo índice o vector índice*, sólo se utilizan los elementos correspondientes del vector de datos, y el vector de dimensión se ignora. En caso de que el índice no sea un vector, sino a su vez una variable indexada, el tratamiento es distinto, como ahora veremos.

[1] Esta regla es la que se utiliza en el lenguaje Fortran

5.3 Uso de variables indexadas como índices

Una variable indexada puede utilizar no sólo un vector de índices, sino incluso una *variable indexada de índices*, tanto para asignar un vector a una colección irregular de elementos de una variable indexada como para extraer una colección irregular de elementos.

Veamos un ejemplo sobre una matriz, que es una variable indexada con dos índices. Puede construirse un índice matricial consistente en dos columnas y varias filas. Los elementos del índice matricial son los índices fila y columna para construir la matriz de índices. Supongamos que `X` es una variable indexada 4×5 y que desea hacer lo siguiente:

- Extraer los elementos `X[1,3]`, `X[2,2]` y `X[3,1]` con una estructura de vector, y
- Reemplazar dichos elementos de `X` con ceros.

Para ello puede utilizar una matriz de índices de 3×2 como en el siguiente ejemplo.

```
> x <- array(1:20,dim=c(4,5))    # Genera una variable indexada (4 × 5).
> x
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    5    9   13   17
[2,]    2    6   10   14   18
[3,]    3    7   11   15   19
[4,]    4    8   12   16   20
> i <- array(c(1:3,3:1),dim=c(3,2))
> i                              # i es una matriz de índices (3 × 2).
     [,1] [,2]
[1,]    1    3
[2,]    2    2
[3,]    3    1
> x[i]                           # Extrae los elementos.
[1] 9 6 3
> x[i] <- 0                      # Sustituye los elementos por ceros.
> x
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    5    0   13   17
[2,]    2    0   10   14   18
[3,]    0    7   11   15   19
[4,]    4    8   12   16   20
>
```

Un ejemplo algo más complejo consiste en generar la matriz de diseño de un diseño en bloques definido por dos factores, `bloques` (niveles `b`) y `variedades` (niveles `v`), siendo el número de parcelas `n`. Puede hacerlo del siguiente modo:

```
> Xb <- matrix(0, n, b)
> Xv <- matrix(0, n, v)
> ib <- cbind(1:n, bloques)
> iv <- cbind(1:n, variedades)
> Xb[ib] <- 1
> Xv[iv] <- 1
> X <- cbind(Xb, Xv)
```

Además, puede construir la matriz de incidencia, `N`, mediante

```
> N <- crossprod(Xb, Xv)
```

También puede construirla directamente mediante la función `table()`:

```
> N <- table(bloques, variedades)
```

5.4 La función `array()`

Una variable indexada no sólo puede construirse modificando el atributo `dim` de un vector, sino también directamente mediante la función `array`, que tiene la forma

```
> Z <- array(vector_de_datos,vector_de_dimensiones)
```

Por ejemplo, si el vector `h` contiene 24 números (o incluso menos), la orden

```
> Z <- array(h, dim=c(3,4,2))
```

usa `h` para almacenar en `Z` una variable indexada de dimensión $3 \times 4 \times 2$. Si el tamaño de `h` es exactamente 24, el resultado coincide con el de

```
> dim(Z) <- c(3,4,2)
```

Sin embargo, si `h` es más corto de 24, sus valores se repiten desde el principio tantas veces como sea necesario para obtener 24 elementos. (véase Sección 5.4.1 [Reciclado], página 22). El caso extremo, muy común, corresponde a un vector de longitud 1, como en este ejemplo

```
> Z <- array(0, c(3,4,2))
```

en que `Z` es una variable indexada compuesta enteramente de ceros.

Además, `dim(Z)`, el vector de dimensiones, es el vector `c(3,4,2)`, `Z[1:24]`, es un vector de datos que coincide con `h`, y tanto `Z[]`, con índice vacío, como `Z`, sin índices, son la variable indexada con estructura de variable indexada.

Las variables indexadas pueden utilizarse en expresiones aritméticas y el resultado es una variable indexada formada a partir de las operaciones elemento a elemento de los vectores subyacentes. Los atributos `dim` de los operandos deben ser iguales en general y coincidirán con el vector de dimensiones del resultado. Así pues, si `A`, `B` y `C` son variables indexadas similares, entonces

```
> D <- 2*A*B + C + 1
```

almacena en `D` una variable indexada similar, cuyo vector de datos es el resultado de las operaciones indicadas sobre los vectores de datos subyacentes a `A`, `B` y `C`. Las reglas exactas correspondientes a los cálculos en que se mezclan variables indexadas y vectores deben ser estudiadas con detenimiento.

5.4.1 Operaciones con variables indexadas y vectores. Reciclado.

Cuando se realizan operaciones que mezclan variables indexadas y vectores, se siguen los siguientes criterios:

- La expresión se analiza de izquierda a derecha.
- Si un vector es más corto que otro, se extiende repitiendo sus elementos (lo que se denomina reciclado) hasta alcanzar el tamaño del vector más largo.
- Si *sólo* hay variables indexadas y vectores más cortos, las variables indexadas deben tener el mismo atributo `dim`, o se producirá un error.
- Si hay un vector más largo que una variable indexada anterior, se produce un mensaje de error.

- Si hay variables indexadas y no se produce error, el resultado es una variable indexada del mismo atributo `dim` que las variables indexadas que intervienen en la operación.

5.5 Producto exterior de dos variables indexadas

Una operación de importancia fundamental entre variables indexadas es el *producto exterior*. Si `a` y `b` son dos variables indexadas numéricas, su producto exterior es una variable indexada cuyo vector de dimensión es la concatenación de los correspondientes de los operandos, en el orden de la operación, y cuyo vector de datos subyacente se obtiene mediante todos los posibles productos de los elementos de los vectores subyacentes de `a` y `b`. El producto exterior se obtiene mediante el operador `%o%`:

```
> ab <- a %o% b
```

o bien, en forma funcional, mediante `outer`:

```
> ab <- outer(a, b, "*")
```

La función "multiplicación" en esta última forma, puede reemplazarse por cualquier función de dos variables. Por ejemplo, para calcular la función $f(x, y) = cos(y)/(1 + x^2)$ sobre la retícula formada por todos los puntos obtenidos combinando las ordenadas y abscisas definidas por los elementos de los vectores `x` e `y` respectivamente, puede utilizar[2] las órdenes:

```
> f <- function(x, y) cos(y)/(1 + x^2)
> z <- outer(x, y, f)
```

En particular, el producto exterior de dos vectores, es una variable indexada con dos índices (esto es, una matriz, de rango 1 a lo sumo). Debe tener en cuenta que el producto exterior no es conmutativo.

Ejemplo: Distribución del determinante de una matriz de dígitos de 2×2

Un ejemplo apropiado se presenta en el cálculo del determinante de una matriz 2×2, $[a, b; c, d]$, en que cada elemento de la misma es un número natural entre 0 y 9. El problema planteado consiste en encontrar la distribución de los determinantes, $ad - bc$, de todas las matrices posibles de esta forma, y representarla gráficamente, supuesto que cada dígito se elige al azar de una distribución uniforme.

Para ello puede utilizar la función `outer()` dos veces:

```
> d <- outer(0:9, 0:9)
> fr <- table(outer(d, d, "-"))
> plot(as.numeric(names(fr)), fr, type="h",
       xlab="Determinante", ylab="Frecuencia")
```

Advierta cómo se ha forzado el atributo `names` de la tabla de frecuencias a numérico, para recuperar el rango de los valores de los determinantes. La forma aparentemente "obvia" de resolver este problema mediante iteraciones de tipo `for`, que se discutirán en el Capítulo 9 [Ciclos y ejecucion condicional], página 44, es tan ineficiente que es impracticable. Al observar el resultado, tal vez le sorprenda que aproximadamente una de cada veinte matrices sea singular.

[2] La definición de una función en R se estudia en el capítulo Capítulo 10 [Escritura de funciones], página 46.

5.6 Traspuesta generalizada de una variable indexada

La función `aperm(a, perm)` puede usarse para permutar la variable indexada `a`. El argumento `perm` debe ser una permutación de los enteros $\{1, \ldots, k\}$ siendo k el número de índices de `a`. El resultado es una variable indexada del mismo tamaño que `a` en la que la dimensión que en la original era `perm[j]` será ahora la dimensión `j`. Si `A` es una matriz, entonces

```
> B <- aperm(A, c(2,1))
```

almacena en `B` la matriz traspuesta de `A`. En el caso de matrices es más sencillo utilizar la función `t()`, y bastaría escribir `B <- t(A)`.

5.7 Operaciones con matrices

Como ya se ha indicado varias veces, una matriz es simplemente una variable indexada con dos índices. Ahora bien, su importancia es tal que necesita un apartado especial. R dispone de muchos operadores y funciones diseñados específicamente para matrices. Por ejemplo, acabamos de ver que `t(X)` es la matriz traspuesta de `X`. Las funciones `nrow` y `ncol` devuelven el número de filas y de columnas de una matriz.

5.7.1 Producto matricial. Inversa de una matriz. Resolución de sistemas lineales

El operador `%*%` realiza el producto matricial. Una matriz de $n \times 1$ o de $1 \times n$ puede ser utilizada como un vector n-dimensional en caso necesario. Análogamente R puede usar automáticamente un vector en una operación matricial convirtiéndolo en una matriz fila o una matriz columna cuando ello es posible (A veces la conversión no está definida de modo único, como veremos después).

Si, por ejemplo, `A` y `B`, son matrices cuadradas del mismo tamaño, entonces

```
> A * B
```

es la matriz de productos elemento a elemento, en tanto que

```
> A %*% B
```

es el producto matricial. Si `x` es un vector (de la dimensión apropiada) entonces

```
> x %*% A %*% x
```

es una forma cuadrática.[3]

La función `crossprod()` realiza el producto cruzado de matrices, esto es `crossprod(X,y)` suministra el mismo resultado que `t(X)%*%y`, pero la operación es más eficiente. Si omite el segundo argumento de la función `crossprod()`, ésta lo toma igual al primero.

También existe la función `diag()`. Si su argumento es una matriz, `diag(matriz)`, devuelve un vector formado por los elementos de la diagonal de la misma. Si, por el contrario,

[3] Si hubiese escrito `x %*% x` el resultado es ambiguo, pues tanto podría significar $\mathtt{x}'\mathtt{x}$ como $\mathtt{x}\mathtt{x}'$, donde `x` es la forma columna. En este tipo de casos, la interpretación corresponde a la matriz de menor tamaño, por lo que en este ejemplo el resultado es el escalar $\mathtt{x}'\mathtt{x}$. La matriz $\mathtt{x}\mathtt{x}'$ puede calcularse mediante `cbind(x) %*% x`, mediante `x %*% rbind(x)` o mediante `x %*% rbind(x)`, puesto que tanto el resultado de `rbind()` como el de `cbind()` son matrices.

su argumento es un vector (de longitud mayor que uno), `diag(vector)`, lo transforma en una matriz diagonal cuyos elementos diagonales son los del vector. Y, por último, si su argumento es un número natural, `n`, lo transforma en una matriz identidad de tamaño $n \times n$.

5.7.2 Autovalores y autovectores

Como ya hemos indicado, la función `eigen()` calcula los autovalores y autovectores de una matriz simétrica. El resultado es una lista de dos componentes llamados `values` y `vectors`. La asignación

```
> ev <- eigen(Sm)
```

almacenará esta lista en `ev`. Por tanto `ev$val` es el vector de autovalores de `Sm` y `ev$vec` es la matriz de los correspondientes autovectores. Si sólo quisiéramos almacenar los autovalores podríamos haber hecho la asignación:

```
> evals <- eigen(Sm)$values
```

y en este caso `evals` sólo contendría los autovalores, habiéndose descartado la segunda componente de la lista. Si se utiliza la directamente la expresión

```
> eigen(Sm)
```

se imprimen las dos componentes, con sus nombres, en la pantalla.

5.7.3 Descomposición en valores singulares. Determinantes

La función `svd` admite como argumento una matriz cualquiera, `M`, y calcula su descomposición en valores singulares, que consiste en obtener tres matrices, `U`, `D` y `V`, tales que la primera es una matriz de columnas ortonormales con el mismo espacio de columnas que `M`, la segunda es una matriz diagonal de números no negativos, y la tercera es una matriz de columnas ortonormales con el mismo espacio de filas que `M`, tales que `M=U%*%D%*%t(V)`. `D` se devuelve en forma de vector formado por los elementos diagonales. El resultado de la función es una lista de tres componentes cuyos nombres son `d`, `u` y `v`, y que corresponden a las matrices descritas.

Si `M` es una matriz cuadrada, es fácil ver que

```
> AbsDetM  <- prod(svd(M)$d)
```

calcula el valor absoluto del determinante de `M`. Si necesita este cálculo a menudo, puede definirlo como una nueva función en R:

```
> AbsDet <- function(M) prod(svd(M)$d)
```

tras lo cual puede usar `AbsDet()` como cualquier otra función de R. Se deja como ejercicio, trivial pero útil, el cálculo de una función, `tr()`, que calcule la traza de una matriz cuadrada. Tenga en cuenta que no necesita realizar ninguna iteración; estudie atentamente el código de la función anterior.

5.7.4 Ajuste por mínimos cuadrados. Descomposición QR

La función `lsfit()` devuelve una lista que contiene los resultados de un ajuste por mínimos cuadrados. Una asignación de la forma

```
> MinCua <- lsfit(X, y)
```

almacena los resultados del ajuste por mínimos cuadrados de un vector de observaciones, `y`, y una matriz de diseño, `X`. Para ver más detalles puede consultar la ayuda, y también la de la función `ls.diag()` para los diagnósticos de regresión. Tenga en cuenta que no necesita incluir un término independiente en `X`, ya que se incluye automáticamente.

Otras funciones estrechamente relacionadas son `qr()` y similares. Considere las siguientes asignaciones:

```
> Xplus <- qr(X)
> b <- qr.coef(Xplus, y)
> fit <- qr.fitted(Xplus, y)
> res <- qr.resid(Xplus, y)
```

que calculan la proyección ortogonal de `y` sobre `X` y la almacenan en `fit`, la proyección sobre el complemento ortogonal en `res` y el vector de coeficientes para la proyección en `b`[4].

No se presupone que `X` sea de rango completo. Se buscan las redundancias y, si existen, se eliminan.

Esta forma es la forma antigua, a bajo nivel, de realizar ajustes de mínimos cuadrados. Aunque sigue siendo útil en algún contexto, debería ser reemplazada por los modelos estadísticos, como se verá en el Capítulo 11 [Modelos estadisticos en R], página 55.

5.8 Submatrices. Funciones `cbind()` y `rbind()`.

Las funciones `cbind()` y `rbind()` construyen matrices uniendo otras matrices (o vectores), horizontalmente (modo columna) o verticalmente (modo fila), respectivamente.

En la asignación

```
> X <- cbind(arg_1, arg_2, arg_3, ...)
```

los argumentos pueden ser vectores de cualquier longitud o matrices con el mismo número de filas. El resultado es una matriz cuyas columnas son los argumentos concatenados, *arg_1*, *arg_2*, . . .

Si alguno de los argumentos de `cbind()` es un vector, y hay alguna matriz, el vector no puede ser más largo que el número de filas de las matrices presentes, y si es más corto, se recicla hasta alcanzar el número indicado. Si sólo hay vectores, los más cortos se reciclan hasta alcanzar el tamaño del mayor.

La función `rbind()` realiza el mismo papel, sustituyendo filas por columnas.

Supongamos que `X1` y `X2` tienen el mismo número de filas. Para combinar las columnas de ambas en una matriz, `X`, que tendrá el mismo número de filas, y añadirle una columna inicial de unos, puede escribir

```
> X <- cbind(1, X1, X2)
```

El resultado de `cbind()` o de `rbind()` siempre es una matriz y estas funciones constituyen, por tanto, la forma más sencilla para tratar un vector como una matriz columna o una matriz fila, respectivamente.

[4] `b` es esencialmente el resultado del operador "barra hacia atrás" de MATLAB.

5.9 La función de concatenación, `c()`, con variables indexadas

En tanto que `cbind()` y `rbind()` son funciones de concatenación que respetan el atributo `dim`, la función `c()` no lo hace, sino que despoja a los objetos numéricos de los atributos `dim` y `dimnames`, lo que, por cierto, puede ser útil en determinadas situaciones.

La forma *oficial* de transformar una variable indexada en el vector subyacente es utilizar la función `as.vector()`,

```
> vec <- as.vector(X)
```

Sin embargo, se obtiene un resultado análogo utilizando la función `c()` debido al efecto colateral citado:

```
> vec <- c(X)
```

Existen sutiles diferencias entre ambos resultados, pero la elección entre ambas es fundamentalmente una cuestión de estilo (personalmente preferimos la primera forma).

5.10 Tablas de frecuencias a partir de factores

Hemos visto que un factor define una tabla de entrada simple. Del mismo modo, dos factores definen una tabla de doble entrada, y así sucesivamente. La función `table()` calcula tablas de frecuencias a partir de factores de igual longitud. Si existen k argumentos categóricos, el resultado será una variable k-indexada, que contiene la tabla de frecuencias.

Vimos en un ejemplo anterior, que `FactorEstado` era un factor que indicaba el estado de procedencia. La asignación

```
> FrecEstado <- table(FactorEstado)
```

almacena en `FrecEstado` una tabla de las frecuencias de cada estado en la muestra. Las frecuencias se ordenan y etiquetan con los niveles del factor. Esta orden es equivalente, y más sencilla, que

```
> FrecEstado <- tapply(FactorEstado, FactorEstado, length)
```

Suponga ahora que `FactorIngresos` es un factor que define "tipos de ingresos", por ejemplo, mediante la función `cut()`:

```
> factor(cut(ingresos,breaks=35+10*(0:7))) -> FactorIngresos
```

Entonces, puede calcular una tabla de frecuencias de doble entrada del siguiente modo:

```
> table(FactorIngresos,FactorEstado)
              FactorEstado
FactorIngresos act nsw nt qld sa tas vic wa
       (35,45]   1   1  0   1  0   0   1  0
       (45,55]   1   1  1   1  2   0   1  3
       (55,65]   0   3  1   3  2   2   2  1
       (65,75]   0   1  0   0  0   0   1  0
```

La extensión a tablas de frecuencias de varias entradas es inmediata.

6 Listas y hojas de datos

6.1 Listas

En R, una *lista* es un objeto consistente en una colección ordenada de objetos, conocidos como *componentes*.

No es necesario que los componentes sean del mismo modo, así una lista puede estar compuesta de, por ejemplo, un vector numérico, un valor lógico, una matriz y una función. El siguiente es un ejemplo de una lista:

```
> Lst <- list(nombre="Pedro", esposa="María", no.hijos=3,
              edad.hijos=c(4,7,9))
```

Los componentes siempre están *numerados* y pueden ser referidos por dicho número. En este ejemplo, `Lst` es el nombre de una lista con cuatro componentes, cada uno de los cuales puede ser referido, respectivamente, por `Lst[[1]]`, `Lst[[2]]`, `Lst[[3]]` y `Lst[[4]]`. Como, además, `Lst[[4]]` es un vector, `Lst[[4]][1]` refiere su primer elemento.

La función `length()` aplicada a una lista devuelve el número de componentes (del primer nivel) de la lista.

Los componentes de una lista pueden tener *nombre*, en cuyo caso pueden ser referidos también por dicho nombre, mediante una expresión de la forma

nombre_de_lista`$`*nombre_de_componente*

Esta convención permite la obtención de una componente sin tener que recurrir a su número.

En el ejemplo anterior,

`Lst$nombre` coincide con `Lst[[1]]` y vale `"Pedro"`,

`Lst$esposa` coincide con `Lst[[2]]` y vale `"María"`,

`Lst$edad.hijos[1]` coincide con `Lst[[4]][1]` y vale 4.

También es posible utilizar los nombres de los componentes entre dobles corchetes, por ejemplo, `Lst[["nombre"]]` coincide con `Lst$nombre`. Esta opción es muy útil en el caso en que el nombre de los componentes se almacena en otra variable, como en

```
> x <- "nombre"; Lst[[x]]
```

Es muy importante distinguir claramente entre `Lst[[1]]` y `Lst[1]`. '`[[...]]`' es el operador utilizado para seleccionar un sólo elemento, mientras que '`[...]`' es un operador general de indexado. Esto es, `Lst[[1]]` es el *primer objeto de la lista* `Lst`, y si es una lista con nombres, el nombre *no* está incluido. Por su parte, `Lst[1]`, es una *sublista de la lista* `Lst` consistente en la primera componente. Si la lista tiene nombre, éste se transfiere a la sublista.

Los nombres de los componentes pueden abreviarse hasta el mínimo de letras necesarios para identificarlos de modo exacto. Así, en

```
> Lista <- list(coeficientes=c(1.3,4), covarianza=.87)
```

`Lst$coeficientes` puede especificarse mediante `Lista$coe`, y `Lista$covarianza` como `Lista$cov`.

El vector de nombres es un atributo de la lista, y como el resto de atributos puede ser manipulado. Además de las listas, también otras estructuras pueden poseer el atributo *names*.

6.2 Construcción y modificación de listas

La función `list()` permite crear listas a partir de objetos ya existentes. Una asignación de la forma

```
> Lista <- list(nombre_1=objeto_1,...,nombre_m=objeto_m)
```

almacena en `Lista` una lista de m componentes que son *objeto_1*, ..., *objeto_m*; a los cuales asigna los nombres *nombre_1*, ...,*nombre_m*; que pueden ser libremente elegidos[1]. Si omite los nombres, las componentes sólo estarán numeradas. Las componentes se *copian* para construir la lista y los originales no se modifican.

Las listas, como todos los objetos indexados, pueden ampliarse especificando componentes adicionales. Por ejemplo

```
> Lst[5] <- list(matriz=Mat)
```

6.2.1 Concatenación de listas

Al suministrar listas como argumentos a la función `c()` el resultado es una lista, cuyos componentes son todos los de los argumentos unidos sucesivamente.

```
> lista.ABC <- c(lista.A, lista.B, lista.C)
```

Recuerde que cuando los argumentos eran vectores, esta función los unía todos en un único vector. En este caso, el resto de atributos, como `dim`, se pierden.

6.3 Hojas de datos (Data frames)

Una *hoja de datos*[2] (Data frame) es una lista que pertenece a la clase `"data.frame"`. Hay restricciones en las listas que pueden pertenecer a esta clase, en particular:

- Los componentes deben ser vectores (numéricos, cadenas de caracteres, o lógicos), factores, matrices numéricas, listas u otras hojas de datos.
- Las matrices, listas, y hojas de datos contribuyen a la nueva hoja de datos con tantas variables como columnas, elementos o variables posean, respectivamente.
- Los vectores numéricos y los factores se incluyen sin modificar, los vectores no numéricos se fuerzan a factores cuyos niveles son los únicos valores que aparecen en el vector.
- Los vectores que constituyen la hoja de datos deben tener todos la *misma longitud*, y las matrices deben tener el mismo *tamaño de filas*

Las hojas de datos pueden interpretarse, en muchos sentidos, como matrices cuyas columnas pueden tener diferentes modos y atributos. Pueden imprimirse en forma matricial y se pueden extraer sus filas o columnas mediante la indexación de matrices.

6.3.1 Construcción de hojas de datos

Puede construir una hoja de datos utilizando la función `data.frame`:

[1] Aunque R permite lo contrario, deberían ser distintos entre sí

[2] Hemos utilizado esta traducción por analogía con la "hoja de cálculo"

```
> cont <- data.frame(dom=FactorEstado, bot=ingresos, dis=FactorIngresos)
```

Puede *forzar* que una lista, cuyos componentes cumplan las restricciones para ser una hoja de datos, realmente lo sea, mediante la función `as.data.frame()`

La manera más sencilla de construir una hoja de datos es utilizar la función `read.table()` para leerla desde un archivo del sistema operativo. Esta forma se tratará en el Capítulo 7 [Lectura de datos desde un archivo], página 33.

6.3.2 Funciones `attach()` y `detach()`

La notación `$` para componentes de listas, como por ejemplo `cont$dom`, no siempre es la más apropiada. En ocasiones, sería cómodo que los componentes de una lista o de una hoja de datos pudiesen ser tratados temporalmente como variables cuyo nombre fuese el del componente, sin tener que especificar explícitamente el nombre de la lista.

La función `attach()` puede tener como argumento el nombre de una lista o de una hoja de datos y permite conectar la lista o la hoja de datos directamente. Supongamos que `lentejas` es una hoja de datos con tres variables, `lentejas$u`, `lentejas$v` y `lentejas$w`. La orden

```
> attach(lentejas)
```

conecta la hoja de datos colocándola en la segunda posición de la trayectoria de búsqueda y, supuesto que no existen variables denominadas `u`, `v` o `w` en la primera posición; `u`, `v` y `w` aparecerán como variables por sí mismas. Sin embargo, si realiza una asignación a una de estas variables, como por ejemplo

```
> u <- v+w
```

no se sustituye la componente `u` de la hoja de datos, sino que se crea una nueva variable, `u`, en el directorio de trabajo, en la primera posición de la trayectoria de búsqueda, que enmascarará a la variable `u` de la hoja de datos. Para realizar un cambio en la propia hoja de datos, basta con utilizar la notación `$`:

```
> lentejas$u <- v+w
```

Este nuevo valor de la componente `u` no será visible de modo directo hasta que desconecte y vuelva a conectar la hoja de datos.

Para desconectar una hoja de datos, utilice la función

```
> detach()
```

Esta función desconecta la entidad que se encuentre en la segunda posición de la trayectoria de búsqueda. Una vez realizada esta operación dejarán de existir las variables `u`, `v` y `w` como tales, aunque seguirán existiendo como componentes de la hoja de datos. Las entidades que ocupan en la trayectoria de búsqueda posiciones superiores a la segunda, pueden desconectarse dando su posición o su nombre como argumento a la función `detach`. Personalmente preferimos la segunda opción, como por ejemplo `detach(lentejas)` o `detach("lentejas")`

> **Nota**: La trayectoria de búsqueda puede almacenar un número finito y pequeño de elementos, por tanto (puesto que además no es necesario) no debe conectar una misma hoja de datos más de una vez. Del mismo modo, es conveniente desconectar una hoja de datos cuando termine de utilizar sus variables.

Nota: En la versión actual de R sólo se pueden conectar listas y hojas de datos en la posición 2 o superior. No es posible asignar directamente en una lista u hoja de datos conectada (por tanto, en cierto sentido, son estáticas).

6.3.3 Trabajo con hojas de datos

Una metodología de trabajo para tratar diferentes problemas utilizando el mismo directorio de trabajo es la siguiente:

- Reúna todas las variables de un mismo problema en una hoja de datos y déle un nombre apropiado e informativo;
- Para analizar un problema, conecte, mediante `attach()`, la hoja de datos correspondiente (en la posición 2) y utilice el directorio de trabajo (en la posición 1) para los cálculos y variables temporales;
- Antes de terminar un análisis, añada las variables que deba conservar a la hoja de datos utilizando la forma `$` para la asignación y desconecte la hoja de datos mediante `detach()`;
- Para finalizar, elimine del directorio de trabajo las variables que no desee conservar, para mantenerlo lo más limpio posible.

De este modo podrá analizar diferentes problemas utilizando el mismo directorio, aunque todos ellos compartan variables denominadas `x`, `y` o `z`, por ejemplo.

6.3.4 Conexión de listas arbitrarias

La función `attach()` es una función genérica, que permite conectar en la trayectoria de búsqueda no sólo directorios y hojas de datos, sino también otros tipos de objetos, en particular cualquier lista, como en

```
> attach(cualquier.lista)
```

Posteriormente podrá desconectar el objeto utilizando la función `detach`, utilizando como argumento el número de posición o, preferiblemente, su nombre.

6.3.5 Gestión de la trayectoria de búsqueda

La función `search` devuelve la trayectoria de búsqueda actual y por tanto es la mejor manera de conocer qué hojas de datos, listas o bibliotecas han sido conectadas o desconectadas. Si no ha realiado ninguna conexión o desconexión su valor es

```
> search()
[1] ".GlobalEnv"   "Autoloads"    "package:base"
```

donde `.GlobalEnv` corresponde al espacio de trabajo.[3]

Una vez conectada la hoja de datos, `lentejas`, tendríamos

```
> search()
[1] ".GlobalEnv"   "lentejas"      "Autoloads"    "package:base"
> ls(2)
[1] "u" "v" "w"
```

[3] Consulte la ayuda sobre `autoload` para la descripción del significado del segundo término.

y, como vimos, `ls` (o `objects`) puede usarse para examinar los contenidos de cualquier posición en la trayectoria de búsqueda.

Por último, desconectamos la hoja de datos y comprobamos que ha sido eliminada de la trayectoria de búsqueda.

```
> detach("lentejas")
> search()
[1] ".GlobalEnv"   "Autoloads"    "package:base"
```

7 Lectura de datos de un archivo

Los datos suelen leerse desde archivos externos y no teclearse de modo interactivo. Las capacidades de lectura de archivos de R son sencillas y sus requisitos son bastante estrictos cuando no inflexibles. Se presupone que el usuario es capaz de modificar los archivos de datos con otras herramientas, por ejemplo con editores de texto[1], para ajustarlos a las necesidades de R. Generalmente esta tarea es muy sencilla.

La función `read.fwf()` puede utilizarse para leer un archivo con campos de anchura fija no delimitados.(Esta función utiliza un programa `perl` para transformar el archivo en otro adaptado para su lectura con `read.table`.) La función `count.fields()` cuenta el número de campos por línea de un archivo de campos delimitados. Estas dos funciones pueden resolver algunos problemas elementales, pero en la mayoría de los casos es mejor preparar el archivo a las necesidades de R antes de comenzar el análisis.

Si los datos se van a almacena en hojas de datos, método que recomendamos, puede leer los datos correspondientes a las mismas con la función `read.table()`. Existe también una función más genérica, `scan()`, que puede utilizar directamente.

7.1 La función `read.table()`

Para poder leer una hoja de datos directamente, el archivo externo debe reunir las condiciones adecuadas. La forma más sencilla es:

- La primera línea del archivo debe contener el *nombre* de cada variable de la hoja de datos.
- En cada una de las siguientes líneas, el primer elemento es la *etiqueta de la fila*, y a continuación deben aparecer los valores de cada variable.

Si el archivo tiene un elemento menos en la primera línea que en las restantes, obligatoriamente será el diseño anterior el que se utilice. A continuación aparece un ejemplo de las primeras líneas de un archivo, `datos.casas`, con datos de viviendas, preparado para su lectura con esta función.

Archivo de entrada con nombres de variables y etiquetas de filas:

```
     Precio   Superficie  Área   Habitaciones     Años     Calef
01   52.00    111.0        830     5              6.2         no
02   54.75    128.0        710     5              7.5         no
03   57.50    101.0       1000     5              4.2         no
04   57.50    131.0        690     6              8.8         no
05   59.75     93.0        900     5              1.9         si
...
```

Predeterminadamente, los elementos numéricos (excepto las etiquetas de las filas) se almacenan como variables numéricas; y los no numéricos, como `Calef`, se fuerzan como factores. Es posible modificar esta acción.

[1] En UNIX puede utilizar el programa `perl` o los editores `sed` y `awk`. Existen versiones para Microsoft Windows.

```
> PreciosCasas <- read.table("datos.casas")
```

A menudo no se dispone de etiquetas de filas. En ese caso, también es posible la lectura y el programa añadirá unas etiquetas predeterminadas. Así, si el archivo tiene la forma siguiente,

Archivo sin etiquetas de filas:

```
Precio   Superficie  Área   Habitaciones     Años     Calef
52.00    111.0        830     5              6.2        no
54.75    128.0        710     5              7.5        no
57.50    101.0       1000     5              4.2        no
57.50    131.0        690     6              8.8        no
59.75     93.0        900     5              1.9        si
...
```

podrá leerse utilizando un parámetro adicional

```
> PreciosCasas <- read.table("datos.casas", header=T)
```

donde el parámetro adicional, `header=T`, indica que la primera línea es una línea de cabeceras y que no existen etiquetas de filas explícitas.

7.2 La función `scan()`

Supongamos que el archivo `entrada.txt` contiene los datos correspondientes a tres vectores, de la misma longitud, el primero de tipo carácter y los otros dos de tipo numérico, escritos de tal modo que en cada línea aparecen los valores correspondientes de cada uno de ellos.

En primer lugar, utilizamos la función `scan()` para leer los tres vectores, del siguiente modo

```
> entrada <- scan("entrada.txt", list("",0,0))
```

El segundo argumento es una estructura de lista que establece el modo de los tres vectores que se van a leer. El resultado se almacena en `entrada`, que será una lista con tres componentes correspondientes a los vectores leídos. Puede referir cada uno de los vectores mediante la indexación:

```
> etiqueta <- entrada[[1]]; x <- entrada[[2]]; y <- entrada[[3]]
```

También podría haber utilizado nombres en la lista que define el modo de lectura, por ejemplo

```
> entrada <- scan("entrada.txt", list(etiqueta="",x=0,y=0))
```

En este caso, puede referir cada uno de los vectores con la notación $. Si desea acceder a las variables separadamente deberá, o bien asignarlas a variables del espacio de trabajo,

```
> etiqueta <- entrada$etiqueta; x <- entrada$x; y <- entrada$y
```

o bien conectar la lista completa en la posición 2 de la trayectoria de búsqueda (véase Sección 6.3.4 [Conexion de listas arbitrarias], página 31).

Si el segundo argumento hubiese sido un sólo elemento y no una lista, todos los elementos del archivo deberían ser del tipo indicado y se hubiesen leído en un sólo vector.

```
> X <- matrix(scan("light.dat", 0), ncol=5, byrow=TRUE)
```

La función `scan` permite realizar lecturas más complejas, como puede consultar en la ayuda.

7.3 Acceso a datos internos

En la distribución de R se incluyen unos cincuenta objetos con datos, y otros más se incluyen en las bibliotecas (incluyendo las estándar). Para utilizar estos datos, deben cargarse explícitamente[2] utilizando la función `data`. Para obtener una lista de los datos existentes en el sistema base utilice

```
data()
```

y para cargar uno, por ejemplo *infert*, debe suministrar dicho nombre como argumento de la función `data`.

```
data(infert)
```

Normalmente una orden de este tipo carga un objeto del mismo nombre que suele ser una hoja de datos. Sin embargo, también es posible que se carguen varios objetos, por lo que en cada caso debe consultar la ayuda interactiva sobre el objeto concreto para conocer exactamente la acción que realizará.

7.3.1 Acceso a datos de una biblioteca

Para acceder a los datos incluidos en una biblioteca, basta utilizar el argumento `package` en la función `data`. Por ejemplo,

```
data(package="nls")
data(Puromycin, package="nls")
```

Si la biblioteca ya ha sido conectada mediante la función `library`, sus datos habrán sido incluidos automáticamente en la trayectoria de búsqueda y no será necesario incluir el argumento `package`. Así,

```
library(nls)
data()
data(Puromycin)
```

presentará una lista de todos los datos de todas las bibliotecas conectadas en ese momento (que serán al menos la biblioteca **base** y la biblioteca **nls**) y posteriormente cargará los datos `Puromycin` de la primera librería en la trayectoria de búsqueda en que encuentre unos datos con dicho nombre.

Las librerías creadas por los usuarios son una fuente valiosa de datos. Por supuesto, las notas del Dr. Venables, fuente original de esta introducción, contienen un conjunto de datos que se encuentra disponible en CRAN en la biblioteca **Rnotes**.

[2] En S-Plus la carga explícita no es necesaria.

7.4 Edición de datos

Una vez creada una estructura de datos, la función `data.entry`, disponible en algunas versiones de R, permite modificarla. La orden

```
%> x.nuevo <- data.entry(x.viejo)
```

edita `x.viejo` utilizando un entorno similar a una hoja de cálculo. Al finalizar, el objeto se almacena en `x.nuevo`. `x.viejo`, y por tanto `x.nuevo`, puede ser una matriz, una hoja de datos, un vector o cualquier objeto atómico.

Si utiliza la función sin argumentos

```
> x.nuevo <- data.entry()
```

permite introducir datos desde una hoja vacía.

7.5 Cómo importar datos

En muchos casos es necesario importar datos desde bases de datos o, en general, desde archivos preparados para otros programas. Se están desarrollando varias bibliotecas para realizar estas tareas. En este momento existe la biblioteca **stataread** que lee y escribe archivos de Stata, y la biblioteca **foreign**, en fase experimental, que lee archivos de SAS, Minitab y SPSS. Otras bibliotecas permiten el acceso a bases de datos que soportan SQL[3], y se está terminando la biblioteca **RODBC** para acceder a bases de datos ODBC (tales como Access en Microsoft Windows).

[3] Acrónimo en inglés de Standard Query Language.

8 Distribuciones probabilísticas

8.1 Tablas estadísticas

R contiene un amplio conjunto de tablas estadísticas. Para cada distribución soportada, hay funciones que permiten calcular la función de distribución, $F(x) = P(X \leq x)$, la función de distribución inversa[1], la función de densidad y generar números pseudoaleatorios de la distribución. Las distribuciones son las siguientes:

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

Para construir el nombre de cada función, utilice el nombre de la distribución precedido de '`d`' para la función de densidad, '`p`' para la función de distribución, '`q`' para la función de distribución inversa, y '`r`' para la generación de números pseudoaleatorios. El primer argumento es `x` para la función de densidad, `q` para la función de distribución, `p` para la función de distribución inversa, y `n` para la función de generación de números pseudoaleatorios (excepto en el caso de `rhyper` y `rwilcox`, en los cuales es `nn`). En el momento de escribir este manual, el parámetro `ncp` sólo está disponible prácticamente en las funciones de distribución. Para conocer dónde puede usarlo utilice la ayuda interactiva.

Además de las anteriores, existen las funciones `ptukey` y `qtukey` para la distribución del rango estudentizado de muestras de una distribución normal.

Los siguientes ejemplos clarificarán estos conceptos:

```
> ## P valor a dos colas de la distribución t_13
> 2*pt(-2.43, df = 13)
> ## Percentil 1 superior de una distribución F(2, 7)
```

[1] Dado q, el menor x tal que $P(X \leq x) > q$

```
> qf(0.99, 2, 7)
```

8.2 Estudio de la distribución de unos datos

Dados unos datos (unidimensionales), su distribución puede estudiarse de muchas formas. La más sencilla es realizar un resumen estadístico, y ello puede obtenerse fácilmente con cualquiera de las funciones `summary` o `fivenum`; y también puede realizar un diagrama de tallo y hojas con la función `stem`.

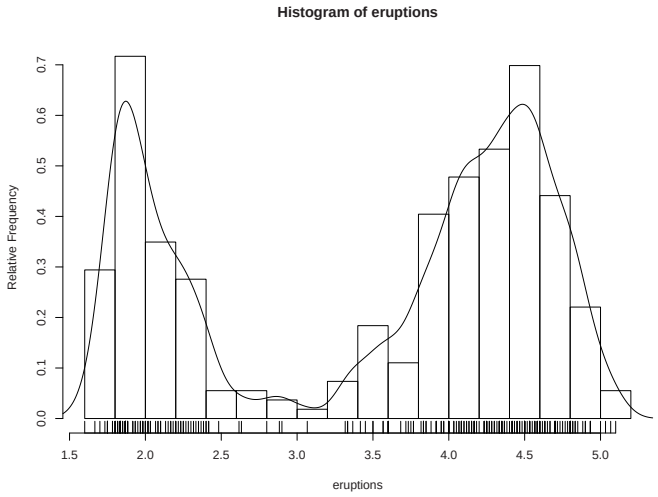
```
> data(faithful)
> attach(faithful)
> summary(eruptions)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  1.600   2.163   4.000   3.488   4.454   5.100
> fivenum(eruptions)
[1] 1.6000 2.1585 4.0000 4.4585 5.1000
> stem(eruptions)

  The decimal point is 1 digit(s) to the left of the |

  16 | 070355555588
  18 | 000022233333335577777777888822335777888
  20 | 00002223378800035778
  22 | 0002335578023578
  24 | 00228
  26 | 23
  28 | 080
  30 | 7
  32 | 2337
  34 | 250077
  36 | 0000823577
  38 | 2333335582225577
  40 | 000000335778888800233555577778
  42 | 033355557788002333335555777 8
  44 | 022223355577800000000233333577788 88
  46 | 0000233357700000023578
  48 | 00000022335800333
  50 | 0370
```

En vez del diagrama de tallo y hojas, puede representar el histograma utilizando la función `hist`.

```
> hist(eruptions)
# define intervalos menores y añade un gráfico de densidad
> hist(eruptions, seq(1.6, 5.2, 0.2), prob=TRUE)
> lines(density(eruptions, bw=0.1))
> rug(eruptions) # muestra los puntos
```

La función `density` permite realizar gráficos de densidad y la hemos utilizado para superponer este gráfico en el ejemplo. La anchura de banda, `bw`, ha sido elegida probando varias, ya que el valor predeterminado produce un gráfico mucho más suavizado. Si necesita

utilizar métodos automáticos de elección de ancho de banda, utilice las bibliotecas **MASS** y **KernSmooth**.

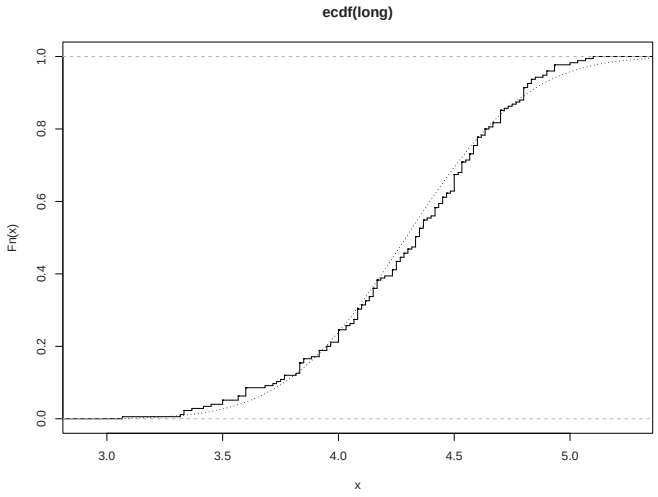


Podemos representar la función de distribución empírica mediante la función `ecdf` de la biblioteca estándar **stepfun**.

```
> library(stepfun)
> plot(ecdf(eruptions), do.points=FALSE, verticals=TRUE)
```

Esta distribución, obviamente, no corresponde a ninguna de las distribuciones estándar. Pero podemos estudiar qué ocurre con las erupciones de más de tres minutos. Vamos a seleccionarlas, ajustarles una distribución normal y superponer la distribución ajustada.

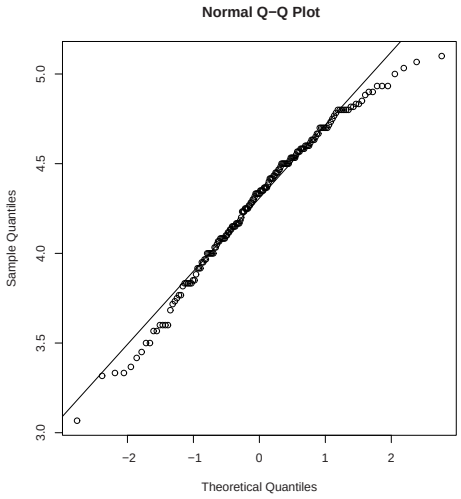
```
> long <- eruptions[eruptions > 3]
> plot(ecdf(long), do.points=FALSE, verticals=TRUE)
> x <- seq(3, 5.4, 0.01)
> lines(x, pnorm(x, mean=mean(long), sd=sqrt(var(long))), lty=3)
```



Los gráficos cuantil-cuantil ("Q-Q plots") pueden ayudarnos a examinar los datos más cuidadosamente.

```
> par(pty="s")
> qqnorm(long); qqline(long)
```

que muestran un ajuste razonable, aunque la cola de la derecha es más corta de lo que sería esperable en una distribución normal. Vamos a compararla con unos datos pseudoaleatorios tomados de una distribución t_5.




```
> x <- rt(250, df = 5)
> qqnorm(x); qqline(x)
```

que la mayoría de las veces (recuerde que es una muestra pseudo aleatoria) tendrá colas más largas de lo que sería esperable en una distribución normal. Podemos realizar un gráfico cuantil-cuantil de estos datos, pero frente a la distribución t_5, mediante

```
> qqplot(qt(ppoints(250), df=5), x, xlab="gráfico Q-Q de t_5")
> qqline(x)
```

Por último, realicemos un contraste de hipótesis para comprobar la normalidad. La biblioteca **ctest** permite realizar el contraste de Shapiro-Wilk

```
> library(ctest)
> shapiro.test(long)

        Shapiro-Wilk normality test

data:  long
W = 0.9793, p-value = 0.01052
```

y el contraste de Kolmogorov-Smirnov

```
> ks.test(long, "pnorm", mean=mean(long), sd=sqrt(var(long)))

        One-sample Kolmogorov-Smirnov test

data:  long
D = 0.0661, p-value = 0.4284
alternative hypothesis: two.sided
```

Hemos utilizado los datos como ejemplo de uso de las funciones, sin estudiar si el mismo es válido. En este caso no lo sería ya que se han estimado los parámetros de la distribución normal a partir de la misma muestra.

8.3 Contrastes de una y de dos muestras

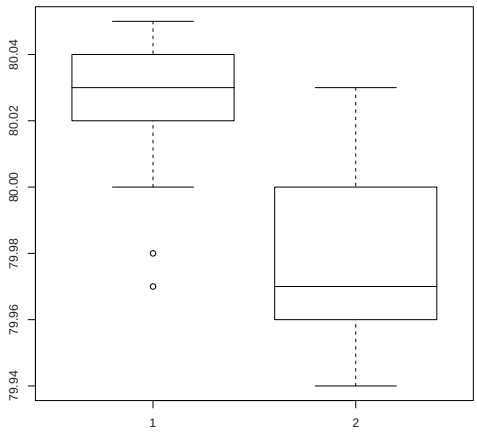
Acabamos de comparar una muestra con una distribución normal, pero es mucho más habitual comparar aspectos de dos muestras. Consideremos los siguientes datos, tomados de Rice (1995, p.490), del calor latente en la fusión del hielo expresados en *cal/gm*.

```
Método A: 79.98 80.04 80.02 80.04 80.03 80.03 80.04 79.97
          80.05 80.03 80.02 80.00 80.02
Método B: 80.02 79.94 79.98 79.97 79.97 80.03 79.95 79.97
```

Podemos comparar gráficamente las dos muestras mediante un diagrama de cajas.

```
> A <- scan()
79.98 80.04 80.02 80.04 80.03 80.03 80.04 79.97
80.05 80.03 80.02 80.00 80.02

> B <- scan()
80.02 79.94 79.98 79.97 79.97 80.03 79.95 79.97

> boxplot(A, B)
```

que muestra claramente que el primer grupo tiende a tener mayores resultados que el segundo.



Para contrastar la igualdad de medias de las dos poblaciones, se puede utilizar el contraste t de Student para *dos muestras independientes*, del siguiente modo:

```
> t.test(A, B)

        Welch Two Sample t-test

data:  A and B
t = 3.2499, df = 12.027, p-value = 0.00694
```

```
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 0.01385526 0.07018320
sample estimates:
mean of x mean of y
 80.02077  79.97875
```

que indica una diferencia significativa (Bajo las condiciones del modelo, incluida la normalidad). Esta función, de modo predeterminado, no presupone que las varianzas son iguales (al contrario que la función análoga de S-PLUS, `t.test`). Si desea contrastar la igualdad de varianzas, puede utilizar la función `var.test` de la biblioteca **ctest**.

```
> library(ctest)
> var.test(A, B)

        F test to compare two variances

data:  A and B
F = 0.5837, num df = 12, denom df =  7, p-value = 0.3938
alternative hypothesis: true ratio of variances is not equal to 1
95 percent confidence interval:
 0.1251097 2.1052687
sample estimates:
ratio of variances
         0.5837405
```

que no muestra evidencia de diferencias significativas (Bajo las condiciones del modelo, que incluyen normalidad). Si hubiésemos admitido esta hipótesis previamente, podríamos haber realizado un contraste más potente, como el siguiente:

```
> t.test(A, B, var.equal=TRUE)

        Two Sample t-test

data:  A and B
t = 3.4722, df = 19, p-value = 0.002551
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 0.01669058 0.06734788
sample estimates:
mean of x mean of y
 80.02077  79.97875
```

Como hemos indicado, una de las condiciones de aplicación de los contrastes anteriores es la normalidad. Si ésta falla, puede utilizar el contraste de dos muestras de Wilcoxon (o de Mann-Whitney) que solo presupone en la hipótesis nula que la distribución común es continua.

```
> library(ctest)
> wilcox.test(A, B)

        Wilcoxon rank sum test with continuity correction
```

```
data:  A and B
W = 89, p-value = 0.007497
alternative hypothesis: true mu is not equal to 0

Warning message:
Cannot compute exact p-value with ties in: wilcox.test(A, B)
```

Advierta el mensaje de advertencia (Warning . . .): Existen valores repetidos en cada muestra, lo que sugiere que los datos no proceden de una distribución continua (puede que ello ocurra debido al redondeo).

Además del diagrama de cajas, existen más métodos para comparar gráficamente dos muestras. Así, las órdenes siguientes:

```
> library(stepfun)
> plot(ecdf(A), do.points=FALSE, verticals=TRUE, xlim=range(A, B))
> plot(ecdf(B), do.points=FALSE, verticals=TRUE, add=TRUE)
```

representan las dos funciones de distribución empíricas. Por otra parte la función `qqplot` realizaría un gráfico cuantil-cuantil de las dos muestras.

El contraste de Kolmogorov-Smirnov, que sólo presupone que la distribución común es continua, también puede aplicarse:

```
> ks.test(A, B)

        Two-sample Kolmogorov-Smirnov test

data:  A and B
D = 0.5962, p-value = 0.05919
alternative hypothesis: two.sided

Warning message:
cannot compute correct p-values with ties in: ks.test(A, B)
```

siéndole de aplicación la misma precaución del contraste de Wilcoxon.

9 Ciclos. Ejecución condicional

9.1 Expresiones agrupadas

R es un lenguaje de expresiones, en el sentido de que el único tipo de orden que posee es una función o expresión que devuelve un resultado. Incluso una asignación es una expresión, cuyo resultado es el valor asignado[1] y que puede utilizarse en cualquier sitio en que pueda utilizarse una expresión. En particular es posible realizar asignaciones múltiples.

Las órdenes pueden agruparse entre `llaves`, `{`*expr_1*`;...;` *expr_m*`}`, en cuyo caso el valor del grupo es el resultado de la última expresión del grupo que se haya evaluado. Puesto que un grupo es por sí mismo una expresión, puede incluirse entre paréntesis y ser utilizado como parte de una expresión mayor. Este proceso puede repetirse si se considera necesario.

9.2 Órdenes de control

9.2.1 Ejecución condicional: la orden `if`

Existe una construcción condicional de la forma

`> if (`*expr_1*`)` *expr_2* `else` *expr_3*

donde *expr_1* debe producir un valor lógico, y si éste es verdadero, (T), se ejecutará *expr_2*. Si es falso, (F), y se ha escrito la opción `else`, que es opcional, se ejecutará *expr_3*.

A menudo suelen utilizarse los operadores `&&` y `||` como condiciones de una orden `if`. En tanto que `&` y `|` se aplican a todos los elementos de un vector, `&&` y `||` se aplican a vectores de longitud uno y sólo evalúan el segundo argumento si es necesario, esto es, si el valor de la expresión completa no se deduce del primer argumento.

Existe una versión vectorizada de la construcción `if/else`, que es la función `ifelse`, que tiene la forma `ifelse(condición, a, b)` y devuelve un vector cuya longitud es la del más largo de sus argumentos y cuyo elemento *i* es `a[i]` si `condición[i]` es cierta, y `b[i]` en caso contrario.

9.2.2 Ciclos: Órdenes `for`, `repeat` y `while`

Existe una construcción repetitiva de la forma

`> for (`*nombre* `in` *expr_1*`)` *expr_2*

donde *nombre* es la variable de control de iteración, *expr_1* es un vector (a menudo de la forma `m:n`), y *expr_2* es una expresión, a menudo agrupada, en cuyas sub-expresiones puede aparecer la variable de control, *nombre*. *expr_2* se evalúa repetidamente conforme *nombre* recorre los valores del vector *expr_1*.

Por ejemplo, suponga que `ind` es un vector de indicadores de clase y se quieren hacer gráficos de `y` sobre `x`, separados para cada clase. Una posibilidad es usar la función `coplot()`, que veremos más adelante, que produce una matriz de gráficos correspondientes a cada nivel del factor. Otra forma de hacerlo es usar la función `for`:

[1] La asignación devuelve el resultado de modo invisible. Basta escribirla entre paréntesis para comprobarlo.

```
> xc <- split(x, ind)
> yc <- split(y, ind)
> for (i in 1:length(yc)) {
    plot(xc[[i]], yc[[i]]);
    abline(lsfit(xc[[i]], yc[[i]]))
  }
```

La función `split()` produce una lista de vectores dividiendo un vector de acuerdo a las clases especificadas por un factor. Consulte la ayuda para obtener más detalles.

> **Nota**: En R, la función `for()` se utiliza mucho menos que en lenguajes tradicionales, ya que no aprovecha las estructuras de los objetos. El código que trabaja directamente con las estructuras completas suele ser más claro y más rápido.

Otras estructuras de repetición son

```
> repeat expr
```

y

```
> while (condición) expr
```

La función `break` se utiliza para terminar cualquier ciclo. Esta es la única forma (salvo que se produzca un error) de finalizar un ciclo `repeat`.

La función `next` deja de ejecutar el resto de un ciclo y pasa a ejecutar el siguiente[2].

Las órdenes de control se utilizan habitualmente en la escritura de *funciones*, que se tratarán en el Capítulo 10 [Escritura de funciones], página 46, donde se verán varios ejemplos.

[2] No existe equivalente para esta orden en Fortran o Basic

10 Escritura de nuevas funciones

Como hemos visto informalmente hasta ahora, R permite crear objetos del modo *function*, que constituyen nuevas funciones de R, que se pueden utilizar a su vez en expresiones posteriores. En este proceso, el lenguaje gana considerablemente en potencia, comodidad y elegancia, y aprender a escribir funciones útiles es una de las mejores formas de conseguir que el uso de R sea cómodo y productivo.

Debemos recalcar que muchas de las funciones que se suministran con R, como `mean`, `var` o `postscript`, están de hecho escritas en R y, por tanto, no difieren materialmente de las funciones que pueda escribir el usuario.

Para definir una función debe realizar una asignación de la forma

> *NombreDeFuncion* `<-function(`*arg_1*`, `*arg_2*`, ...)` *expresión*

donde *expresión* es una expresión de R (normalmente una expresión agrupada) que utiliza los argumentos *arg_i* para calcular un valor que es devuelto por la función.

El uso de la función es normalmente de la forma *NombreDeFuncion*`(`*expr_1*`,`*expr_2*`,...)` y puede realizarse en cualquier lugar en que el uso de una función sea correcto.

10.1 Ejemplos elementales

En primer lugar, consideremos una función que calcule el estadístico t de Student para dos muestras realizando "todos los pasos". Este es un ejemplo muy artificial, naturalmente, ya que hay otros modos, mucho más sencillos, de obtener el mismo resultado.

La función se define del siguiente modo:

```
> DosMuestras <- function(y1, y2) {
    n1  <- length(y1); n2  <- length(y2)
    yb1 <- mean(y1);   yb2 <- mean(y2)
    s1  <- var(y1);    s2  <- var(y2)
    s <- ((n1-1)*s1 + (n2-1)*s2)/(n1+n2-2)
    tst <- (yb1 - yb2)/sqrt(s2*(1/n1 + 1/n2))
    tst
  }
```

Una vez definida esta función, puede utilizarse para realizar un contraste de t de Student para dos muestras, del siguiente modo:

```
> tStudent <- DosMuestras(datos$hombre, datos$mujer); tStudent
```

En segundo lugar, considere por ejemplo la creación de una función[1] que calcule los coeficientes de la proyección ortogonal del vector y sobre el espacio de las columnas de la matriz X, esto es, la estimación de los coeficientes de regresión por mínimos cuadrados. Esta tarea se realizaría normalmente con la función `qr()`; sin embargo es algo compleja y compensa tener una función como la siguiente para usarla directamente.

Dado un vector, $y_{n\times 1}$, y una matriz, $X_{n\times p}$, entonces definimos $X\backslash y$ del siguiente modo: $(X'X)^{-}X'y$, donde $(X'X)^{-}$ es la inversa generalizada de $X'X$.

Podemos definir la función `Proyeccion` del siguiente modo

[1] En MATLAB, sería la orden \

```
> Proyeccion <- function(X, y) {
  X <- qr(X)
  qr.coef(X, y)
}
```

Una vez creada, puede utilizarla en cualquier expresión, como en la siguiente:

```
> CoefReg <- Proyeccion(matrizX, variabley)
```

La función `lsfit()` realiza la misma acción[2]. También utiliza las funciones `qr()` y `qr.coef()` en la forma anterior para realizar esta parte del cálculo. Por lo tanto puede ser interesante haber aislado esta parte en una función si se va a utilizar frecuentemente. Si ello es así, probablemente sería conveniente construir un operador binario matricial para que el uso sea más cómodo.

10.2 Cómo definir un operador binario

Si hubiésemos dado a la función `Proyeccion` un nombre delimitado por símbolos de porcentaje, `%`, por ejemplo de la forma

`%`*barra*`%`

podría utilizarse como un *operador binario* en vez de con la forma funcional. Suponga, por ejemplo, que elige[3] como nombre, entre los símbolos de porcentaje, el de `!`. La definición de la función debería comenzar así:

```
> "%!%" <- function(X, y) { ... }
```

donde hay que destacar la utilización de comillas. Una vez definida la función se utilizaría de la forma `X %!% y`.

Los operadores producto matricial, `%*%`, y producto exterior, `%o%`, son ejemplos de operadores binarios definidos de esta forma.

10.3 Argumentos con nombre. Valores predeterminados

Ya vimos en la Sección 2.3 [Generacion de sucesiones], página 9 que cuando los argumentos se dan por nombre, "*nombre*=*objeto*", el orden de los mismos es irrelevante. Además pueden utilizarse ambas formas simultáneamente: se puede comenzar dando argumentos por posición y después añadir argumentos por nombre.

Esto es, si la función `fun1` está definida como

```
> fun1 <- function(datos, hoja.datos, grafico, limite) {
    [aquí iría la definición]
  }
```

las siguientes llamadas a la función son equivalentes:

```
> resultado <- fun1(d, hd, T, 20)
> resultado <- fun1(d, hd, grafico=T, limite=20)
> resultado <- fun1(datos=d, limite=20, grafico=T, hoja.datos=hd)
```

[2] Vea también los métodos descritos en Capítulo 11 [Modelos estadisticos en R], página 55

[3] El uso del símbolo \ para el nombre, como en MATLAB, no es una elección conveniente, ya que presenta ciertos problemas en este contexto.

En muchos casos, puede suministrarse un valor predeterminado para algunos argumentos, en cuyo caso al ejecutar la función el argumento puede omitirse si el valor predeterminado es apropiado. Por ejemplo, si `fun1` estuviese definida como

```
> fun1 <- function(datos, hoja.datos, grafico=TRUE, limite=20) { ... }
```

la llamada a la función

```
> resultado <- fun1(d, hd)
```

sería equivalente a cualquiera de las tres llamadas anteriores. Tenga en cuenta que puede modificar los valores predeterminados, como en el caso siguiente:

```
> resultado <- fun1(d, hd, limite=10)
```

Es importante destacar que los valores predeterminados pueden ser expresiones arbitrarias, que incluso involucren otros argumentos de la misma función, y no están restringidos a ser constantes como en el ejemplo anterior.

10.4 El argumento '...'

Otra necesidad frecuente es la de que una función pueda pasar los valores de sus argumentos a otra función. Por ejemplo, muchas funciones gráficas, como `plot()`, utilizan la función `par()`, y permiten al usuario pasar los parámetros gráficos a `par()` para controlar el resultado gráfico. (Véase Sección 12.4.1 [La funcion par()], página 76 para detalles adicionales sobre la función `par()`.) Esta acción puede realizarse incluyendo un argumento adicional, "`...`", en la función, que puede ser traspasado. A continuación se incluye un bosquejo de ejemplo.

```
fun1 <- function(datos, hoja.datos, grafico=TRUE, limite=20, ...)  {
  [Algunas órdenes]
  if (grafico)
    par(pch="*", ...)
  [Más órdenes]
}
```

10.5 Asignaciones dentro de una función

Es fundamental tener en cuenta que *cualquier asignación ordinaria realizada dentro de una función es local y temporal y se pierde tras salir de la función*. Por tanto, la asignación `X <- qr(X)` no afecta al valor del argumento de la función en que se utiliza.

Para comprender completamente las reglas que gobiernan el ámbito de las asignaciones en R es necesario familiarizarse con la noción de *marco* (frame) de evaluación. Este es un tema complejo que no será tratado en este manual.

Si desea realizar asignaciones globales y permanentes dentro de una función, deberá utilizar el operador de 'superasignación', `<<-`, o la función `assign`. Puede encontrar una explicación más detallada consultando la ayuda.

El operador `<<-` es diferente en R y en S-Plus. Las diferencias serán tratadas en la Sección 10.7 [Ambito], página 51.

10.6 Ejemplos más complejos

10.6.1 Factores de eficiencia en diseño en bloques

Estudiaremos ahora un ejemplo de función más complejo: el cálculo de factores de eficiencia en un diseño en bloques. (Algunos aspectos de este problema ya han sido tratados en la Sección 5.3 [Variables indexadas utilizadas como indices], página 21.)

Un diseño en bloques está definido por dos factores, por ejemplo `bloques` (`b` niveles) y `variedades`, (`v` niveles). Si $R_{v\times v}$ y $K_{b\times b}$ son las matrices de *réplicas* y *tamaño de bloque*, y $N_{b\times v}$, es la matriz de incidencia, entonces los factores de eficiencia se definen como los autovalores de la matriz

$$E = I_v - R^{-1/2}N'K^{-1}NR^{-1/2} = I_v - A'A,$$

donde $A = K^{-1/2}NR^{-1/2}$.

Por ejemplo, la función podría escribirse así:

```
> EfiDisBlo <- function(bloques, variedades) {
    bloques <- as.factor(bloques)            # pequeña precaución
    b <- length(levels(bloques))
    variedades <- as.factor(variedades)      # pequeña precaución
    v <- length(levels(variedades))
    K <- as.vector(table(bloques))           # elimina el atributo dim
    R <- as.vector(table(variedades))        # elimina el atributo dim
    N <- table(bloques, variedades)
    A <- 1/sqrt(K) * N * rep(1/sqrt(R), rep(b, v))
    sv <- svd(A)
    list(eficiencia=1 - sv$d^2, cvbloques=sv$u, cvvariedad=sv$v)
}
```

Desde el punto de vista numérico, es levemente mejor trabajar con la función descomposición SVD en vez de con la función de los autovalores.

El resultado de esta función es una lista que contiene los factores de eficiencia como primera componente, y que además incluye dos contrastes, puesto que, a veces, suministran información adicional útil.

10.6.2 Cómo eliminar los nombres al imprimir una variable indexada

Para imprimir grandes matrices o variables indexadas en general, a menudo es interesante hacerlo en forma compacta sin los nombres de variables. La simple eliminación del atributo `dimnames` no es suficiente, sino que la solución consiste en asignar a dicho atributo cadenas de caracteres vacías. Por ejemplo, para imprimir la matriz `X` puede escribir

```
> temp <- X
> dimnames(temp) <- list(rep("", nrow(X)), rep("", ncol(X))
> temp; rm(temp)
```

Este resultado puede conseguirse fácilmente definiendo la función `SinNombres`, que aparece a continuación, que da un pequeño rodeo para conseguir el mismo resultado al

tiempo que ilustra el hecho de que las funciones pueden ser cortas y al mismo tiempo muy efectivas y útiles.

```
SinNombres <- function(a) {
  ## Elimina los nombres de dimensiones para impresión compacta.
  d <- list()
  l <- 0
  for(i in dim(a)) {
    d[[l <- l + 1]] <- rep("", i)
  }
  dimnames(a) <- d
  a
}
```

Una vez definida la función, para imprimir la matriz `X` en forma compacta basta con escribir

```
> SinNombres(X)
```

Esta función es de especial utilidad al imprimir variables indexadas de tipo entero y de gran tamaño, en que el interés real se centra más en los posibles patrones que en los valores en sí mismos.

10.6.3 Integración numérica recursiva

Las funciones pueden ser recursivas e, incluso, pueden definir funciones en su interior. Advierta, sin embargo, que dichas funciones, y por supuesto las variables, no son heredadas por funciones llamadas en marcos de evaluación superior, como lo serían si estuviesen en la trayectoria de búsqueda.

El ejemplo siguiente muestra una forma, un tanto ingenua, de realizar integración numérica unidimensional recursivamente. El integrando se evalúa en los extremos del intervalo y en el centro. Si el resultado de aplicar la regla del trapecio a un solo intervalo es bastante próxima al resultado de aplicarlo a los dos, entonces este último valor se considera el resultado. En caso contrario se aplica el procedimiento a cada uno de los dos intervalos. El resultado es un proceso de integración adaptativo que concentra las evaluaciones de la función en las regiones en que es menos lineal. Conlleva, sin embargo, un gran consumo de recursos, y la función solo es competitiva con otros algoritmos cuando el integrando es al tiempo suave y difícil de evaluar. El ejemplo es también un pequeño rompecabezas de programación en R.

```
area <- function(f, a, b, eps = 1.0e-06, lim = 10) {
  fun1 <- function(f, a, b, fa, fb, a0, eps, lim, fun) {
    ## La función 'fun1' sólo es visible dentro de 'area'
    d <- (a + b)/2
    h <- (b - a)/4
    fd <- f(d)
    a1 <- h * (fa + fd)
    a2 <- h * (fd + fb)
    if(abs(a0 - a1 - a2) < eps || lim == 0)
      return(a1 + a2)
    else {
      return(fun(f, a, d, fa, fd, a1, eps, lim - 1, fun) +
```

```
            fun(f, d, b, fd, fb, a2, eps, lim - 1, fun))
    }
  }
  fa <- f(a)
  fb <- f(b)
  a0 <- ((fa + fb) * (b - a))/2
  fun1(f, a, b, fa, fb, a0, eps, lim, fun1)
}
```

10.7 Ámbito

Este apartado es algo más técnico que otras partes de este documento. Sin embargo, pormenoriza una de las mayores diferencias entre S-Plus y R.

Los símbolos que tienen lugar en el cuerpo de una función se dividen en tres clases: parámetros formales, variables locales y variables libres. Los parámetros formales son los que aparecen en la lista de argumentos de la función y sus valores quedan determinados por el proceso de asignación de los argumentos de la función a los parámetros formales. Las variables locales son aquellas cuyos valores están determinados por la evaluación de expresiones en el cuerpo de las funciones. Las variables que no son parámetros formales ni variables locales se denominan variables libres. Las variables libres se transforman en variables locales si se les asigna valor. Para aclarar los conceptos, consideremos la siguiente función:

```
f <- function(x) {
  y <- 2*x
  print(x)
  print(y)
  print(z)
}
```

En esta función, `x` es un parámetro formal, `y` es una variable local y `z` es una variable libre.

En R la asignación de valor a una variable libre se realiza consultando el entorno en el que la función se ha creado, lo que se denomina *ámbito léxico*. En primer lugar definamos la función `cubo`:

```
cubo <- function(n) {
  sq <- function() n*n
  n*sq()
}
```

La variable `n` de la función `sq` no es un argumento para esta función. Por tanto es una variable libre y las reglas de ámbito deben utilizarse para determinar el valor asociado con ella. En un ámbito estático (como en S-Plus) el valor es el asociado con una variable global llamada `n`. En un ámbito léxico (como en R) es un parámetro para la función `cubo` puesto que hay una asignación activa para la variable `n` en el momento en que se define la función `sq`. La diferencia de evaluación entre R y S-Plus es que S-Plus intenta encontrar una variable global llamada `n` en tanto que R primero intenta encontrar una variable llamada `n` en el entorno creado cuando se activó `cubo`.

```
## primera evaluación en S
```

```
S> cubo(2)
Error in sq(): Object "n" not found
Dumped
S> n <- 3
S> cubo(2)
[1] 18
## la misma función evaluada en R
R> cubo(2)
[1] 8
```

El ámbito lexicográfico puede utilizarse para conceder a las funciones un *estado cambiante*. En el siguiente ejemplo mostramos cómo puede utilizarse R para simular una cuenta bancaria. Una cuenta bancaria necesita tener un balance o total, una función para realizar depósitos, otra para retirar fondos, y una última para conocer el balance.

Conseguiremos esta capacidad creando tres funciones dentro de `anota.importe` y devolviendo una lista que los contiene. Cuando se ejecuta `anota.importe` toma un argumento numérico, `total`, y devuelve una lista que contiene las tres funciones. Puesto que estas funciones están definidas dentro de un entorno que contiene a `total`, éstas tendrán acceso a su valor.

El operador de asignación especial, `<<-`, se utiliza para cambiar el valor asociado con `total`. Este operador comprueba los entornos creados desde el actual hasta el primero hasta encontrar uno que contenga el símbolo `total` y cuando lo encuentra, sustituye su valor en dicho entorno por el valor de la derecha de la expresión. Si se alcanza el nivel superior, correspondiente al entorno global, sin encontrar dicho símbolo, entonces lo crea en él y realiza la asignación. Para muchos usos, `<<-` crea una variable global y le asigna el valor de la derecha de la expresión[4]. Solo cuando `<<-` ha sido utilizado en una función que ha sido devuelta como el valor de otra función ocurrirá la conducta especial que hemos descrito.

```
anota.importe <- function(total) {
  list(
    deposito = function(importe) {
        if(importe <= 0)
            stop("Los depósitos deben ser positivos!\n")
          total <<- total + importe
          cat("Depositado",importe,". El total es", total, "\n\n")
    },
    retirada = function(importe) {
        if(importe > total)
            stop("No tiene tanto dinero!\n")
        total <<- total - importe
        cat("Descontado", importe,". El total es", total,"\n\n")
    },
    balance = function() {
        cat("El total es", total,"\n\n")
    }
  )
```

[4] En cierto sentido esto emula la conducta en S-Plus puesto que en S-Plus este operador siempre crea o asigna a una variable global.

```
}

Antonio <- anota.importe(100)
Roberto <- anota.importe(200)

Antonio$retirada(30)
Antonio$balance()
Roberto$balance()

Antonio$deposit(50)
Antonio$balance()
Antonio$retirada(500)
```

10.8 Personalización del entorno

El usuario de R puede adaptar el entorno de trabajo a sus necesidades de varias formas. Existe un archivo de inicialización del sistema y cada directorio puede tener su propio archivo de inicialización especial. Por último, puede usar las funciones especiales `.First` y `.Last`.

El archivo de inicialización del sistema se denomina `Rprofile` y se encuentra en el subdirectorio `library` del directorio inicial de R. Las órdenes contenidas en este archivo se ejecutan cada vez que se comienza una sesión de R, sea cual sea el usuario. Existe un segundo archivo, personal, denominado `.Rprofile`[5] que puede estar en cualquier directorio. Si ejecuta R desde un directorio que contenga este archivo, se ejecutarán las órdenes que incluya. Este archivo permite a cada usuario tener control sobre su espacio de trabajo y permite disponer de diferentes métodos de inicio para diferentes directorios de trabajo.

Si no existe el archivo `.Rprofile` en el directorio inicial, entonces R buscará si existe el archivo `.Rprofile` en el directorio inicial del usuario y, si existe, lo utilizará.

Si existe la función `.First()` (en cualquiera de los dos archivos de perfil o en el archivo de imagen '`.RData`') recibirá un tratamiento especial, ya que se ejecutará al comienzo de la sesión de R, y por tanto puede utilizarse para inicializar el entorno. Por ejemplo, la definición del siguiente ejemplo sustituye el símbolo de R para que sea `$` y establece otras características que quedan establecidas en el resto de la sesión.

En resumen, la secuencia en que se ejecutan los archivos es, '`Rprofile`', '`.Rprofile`', '`.RData`' y por último la función `.First()`. Recuerde que cualquier definición en un archivo posterior enmascarará las de un archivo precedente.

```
> .First <- function() {
    options(prompt="$ ", continue="+\t")
    # $ será el símbolo de sistema
    options(digits=5, length=999)
    # personaliza números y resultados
    x11()
    # abre una ventana para gráficos
    par(pch = "+")
    # carácter para realización de gráficos
```

[5] Por tanto, al comenzar su nombre con un punto, en UNIX, será un archivo oculto.

```
    source(paste(getwd(), "/R/MisOrdenes.R", sep = ""))
    # ejecuta las órdenes contenidas en MisOrdenes.R
    library(stepfun)
    # conecta la biblioteca stepfun
}
```

De modo análogo, si existe la función `.Last()`, se ejecutará al término de la sesión. A continuación se muestra un ejemplo de esta función.

```
> .Last <- function() {
    Graficos.off()
    # Una pequeña medida de seguridad.
    cat(paste(system.date(),"\nAdiós\n"))
    # Ya es la hora de irse.
}
```

10.9 Clases. Funciones genéricas. Orientación a objetos

La clase de un objeto determina de qué modo será tratado por lo que se conoce como funciones *genéricas*. Volviendo la oración por pasiva, una función será genérica si realiza una tarea o acción sobre sus argumentos *específica de la clase de cada argumento*. Si el argumento carece del atributo clase, o lo posee de uno no contemplado específicamente por la función genérica en cuestión, se suministra una *acción predeterminada*.

El mecanismo de clase ofrece al usuario la posibilidad de diseñar y escribir funciones genéricas para propósitos especiales. Entre otras funciones genéricas se encuentran `plot()`, para representar objetos gráficamente, `summary()`, para realizar análisis descriptivos de varios tipos, y `anova()`, para comparar modelos estadísticos.

El número de funciones genéricas que pueden tratar una clase de modo específico puede ser muy grande. Por ejemplo, entre las funciones que pueden tratar de modo específico objetos de la clase `"data.frame"` se encuentran

```
[      [[<-     any     as.matrix
[<-    model    plot    summary
```

Puede obtener la lista completa utilizando la función `methods`:

```
> methods(class="data.frame")
```

Como es esperable, el número de clases que una función genérica puede tratar también puede ser grande. Por ejemplo, la función `plot()` tiene variantes, entre otras, para las siguientes clases de objetos:

```
data.frame   default   density   factor
```

También en este caso puede obtener la lista completa actual utilizando la función `methods`:

```
> methods(plot)
```

11 Modelos estadísticos en R

En este apartado, suponemos al lector familiarizado con la terminología estadística, en particular con el análisis de regresión y el análisis de varianza. Posteriormente haremos algunas suposiciones más ambiciosas, particularmente el conocimiento de modelos lineales generalizados y regresión no lineal.

Los requisitos para el ajuste de modelos estadísticos están suficientemente bien definidos para hacer posible la construcción de herramientas generales de aplicación a un amplio espectro de problemas.

R contiene un conjunto de posibilidades que hace que el ajuste de modelos estadísticos sea muy simple. Como hemos mencionado en la introducción, la salida básica es mínima, y es necesario utilizar las funciones extractoras para obtener todos los detalles.

11.1 Definición de modelos estadísticos. Fórmulas

El ejemplo básico de un modelo estadístico es un modelo de regresión lineal con errores independientes y homoscedásticos

$$y_i = \sum_{j=0}^{p} \beta_j x_{ij} + e_i, \qquad e_i \sim \text{NID}(0, \sigma^2), \qquad i = 1, \ldots, n$$

En notación matricial puede escribirse

$$y = X\beta + e$$

donde y es el vector de respuesta, y X es la *matriz del modelo* o *matriz de diseño*, formada por las columnas $x_0, x_1, \ldots, x_p$, que son las variables predictoras. Muy a menudo x_0 será una columna de unos y definirá el *punto de corte* o *término independiente*.

Ejemplos

Antes de dar una definición formal, algunos ejemplos ayudarán a centrar las ideas.

Supongamos que `y`, `x`, `x0`, `x1`, `x2`, ... son variables numéricas, que `X` es una matriz y que `A`, `B`, `C`, ... son factores. Las fórmulas que aparecen en la parte izquierda de la siguiente tabla, especifican los modelos estadísticos descritos en la parte de la derecha.

`y ~ x`
`y ~ 1 + x` Ambos definen el mismo modelo de regresión lineal de y sobre x. El primero contiene el término independiente implícito y el segundo, explícito.

`y ~ 0 + x`
`y ~ -1 + x`
`y ~ x - 1` Regresión lineal de y sobre x sin término independiente, esto es, que pasa por el origen de coordenadas.

`log(y) ~ x1 + x2`
Regresión múltiple de la variable transformada, $\log(y)$, sobre x_1 y x_2 (con un término independiente implícito).

`y ~ poly(x,2)`
`y ~ 1 + x + I(x^2)`
: Regresión polinomial de y sobre x de segundo grado. La primera forma utiliza polinomios ortogonales y la segunda utiliza potencias de modo explícito.

`y ~ X + poly(x,2)`
: Regresión múltiple de y con un modelo matricial consistente en la matriz X y términos polinomiales en x de segundo grado.

`y ~ A` Análisis de varianza de entrada simple de y, con clases determinadas por A.

`y ~ A + x` Análisis de covarianza de entrada simple de y, con clases determinadas por A, y con covariante x.

`y ~ A*B`
`y ~ A + B + A:B`
`y ~ B %in% A`
`y ~ A/B` Modelo no aditivo de dos factores de y sobre A y B. Los dos primeros especifican la misma clasificación cruzada y los dos últimos especifican la misma clasificación anidada. En términos abstractos, los cuatro especifican el mismo subespacio de modelos.

`y ~ (A + B + C)^2`
`y ~ A*B*C - A:B:C`
: Experimento con tres factores con un modelo que contiene efectos principales e interacciones de dos factores solamente. Ambas fórmulas especifican el mismo modelo.

`y ~ A * x`
`y ~ A/x`
`y ~ A/(1 + x) - 1`
: Modelos de regresión lineal simple separados de y sobre x para cada nivel de A. La última forma produce estimaciones explícitas de tantos términos independientes y pendientes como niveles tiene A.

`y ~ A*B + Error(C)`
: Un experimento con dos factores de tratamiento, A y B, y estratos de error determinados por el factor C. Por ejemplo, un experimento split plot, con gráficos completos (y por tanto también subgráficos) determinados por el factor C.

El operador ~ se utiliza para definir una *fórmula de modelo* en R. La forma, para un modelo lineal ordinario es

respuesta ~ *op_1* *term_1* *op_2* *term_2* *op_3* *term_3* ...

donde

respuesta es un vector o una matriz (o una expresión que evalúe a un vector o matriz) que definen, respectivamente, la o las variables respuesta

op_i es un operador, bien `+`, bien `-`, que implica la inclusión o exclusión, respectivamente, de un término en el modelo. El primero, `+`, es opcional.

term_i es un término de uno de los siguientes tipos

- una expresión vectorial, una expresión matricial, o el número `1`; o
- un factor; o
- una *expresión de fórmula* consistente en factores, vectores o matrices conectados mediante *operadores de fórmula*.

En todos los casos, cada término define una colección de columnas que deben ser añadidas o eliminadas de la matriz del modelo. Un `1` significa un término independiente y está incluido siempre, salvo que se elimine explícitamente.

Los *operadores de fórmula* son similares a la notación de Wilkinson y Rogers utilizada en los programas Glim y Genstat. Un cambio inevitable es que el operador '`.`' se ha sustituido por '`:`' puesto que el punto es un carácter válido para nombres de objetos en R. Un resumen de la notación se encuentra en la siguiente tabla (basada en Chambers & Hastie, 1992, p.29).

$Y \sim M$ Y se modeliza como M.

M_1 `+` *M_2*
: Incluye *M_1* y *M_2*.

M_1 `-` *M_2*
: Incluye *M_1* exceptuando los términos de *M_2*.

M_1 `:` *M_2*
: El producto tensorial de *M_1* y *M_2*. Si ambos son factores, corresponde al factor "subclases".

M_1 `%in%` *M_2*
: Similar a *M_1*`:`*M_2*, pero con diferente codificación.

M_1 `*` *M_2*
: *M_1* `+` *M_2* `+` *M_1*`:`*M_2*.

M_1 `/` *M_2*
: *M_1* `+` *M_2* `%in%` *M_1*.

M`^`*n* Todos los términos de M junto a las "interacciones" hasta el orden n

`I(`*M*`)` Aísla *M*. Dentro de M todos los operadores tienen su sentido aritmético habitual y este término aparece en la matriz del modelo.

Advierta que, dentro de los paréntesis que habitualmente rodean los argumentos de una función, todos los operadores tienen su sentido aritmético habitual. La función `I()` es la función identidad, utilizada solamente para poder introducir términos en las fórmulas, definiéndolos mediante operadores aritméticos.

En particular, cuando las fórmulas especifican *columnas de la matriz del modelo*, la especificación de los parámetros es implícita. Este no es el caso en otros contextos, por ejemplo en la especificación de modelos no lineales.

11.1.1 Contrastes

Es necesario conocer, aunque sea someramente, el modo en que las fórmulas del modelo determinan las columnas de la matriz del modelo. Esto es sencillo si las variables son

continuas, ya que cada una constituirá una columna de dicha matriz. Del mismo modo, si el modelo incluye un término independiente, contribuirá con una columna de unos.

En el caso de un factor, `A`, con k niveles, la respuesta depende de si el factor es nominal u ordinal. En el caso de un factor nominal, se generan $k-1$ columnas correspondientes a los indicadores desde el segundo hasta el k-ésimo nivel del factor. (Por tanto, la parametrización implícita consiste en contrastar la respuesta del primer nivel frente a cada uno de los restantes niveles.) En el caso de un factor ordinal, las $k-1$ columnas son los polinomios ortogonales sobre `1, ..., k`, omitiendo el término constante.

Esta situación puede parecerle complicada, pero aún hay más. En primer lugar, si el término independiente se omite en un modelo que contiene algún término de tipo factor, el primero de dichos términos se codifica en k columnas correspondientes a los indicadores de todos los niveles del factor. En segundo lugar, todo este comportamiento puede cambiarse mediante el argumento `contrasts` de `options`. Los valores predeterminados son:

```
options(contrasts = c("contr.treatment", "contr.poly"))
```

La razón por la que se indican estos valores es que los valores predeterminados en R son distintos de los de S en el caso de factores nominales, ya que S utiliza los contrastes de Helmert. Por tanto, para obtener los mismos resultados que en S-PLUS, deberá escribir:

```
options(contrasts = c("contr.helmert", "contr.poly"))
```

Esta diferencia es deliberada, ya que entendemos que los contrastes predeterminados de R son más sencillos de interpretar para los principiantes.

Caben aún más posibilidades, ya que el esquema de contraste a utilizar puede fijarse para cada término del modelo utilizando las funciones `contrasts` y `C`.

Tampoco hemos considerado los términos de interacción, que generan los productos de las columnas introducidas por los términos de sus componentes.

Pese a que los detalles son complicados, las fórmulas de modelos en R generan habitualmente los modelos que un estadístico experto podría esperar, supuesto que se preserve la marginalidad. Por ejemplo, el ajuste de un modelo con interacción y, sin embargo, sin los correspondientes efectos principales conducirá en general a resultados sorprendentes, y debe reservarse sólo a los especialistas.

11.2 Modelos lineales

La función primaria para el ajuste de modelos múltiples ordinarios es `lm()` y una versión resumida de su uso es la siguiente:

```
> modelo.ajustado <- lm(formula.de.modelo, data=hoja.de.datos)
```

Por ejemplo

```
> fm2 <- lm(y ~ x1 + x2, data=produccion)
```

ajustará un modelo de regresión múltiple de y sobre x_1 y x_2 (con término independiente implícito).

El término `data=produccion`, pese a ser opcional, es importante y especifica que cualquier variable necesaria para la construcción del modelo debe provenir en primer lugar de la *hoja de datos* `produccion`, *y ello independientemente de que la hoja de datos* `produccion` *haya sido conectada a la trayectoria de búsqueda o no.*

11.3 Funciones genéricas de extracción de información del modelo

El valor de `lm()` es el objeto *modelo ajustado*; que consiste en una lista de resultados de clase `lm`. La información acerca del modelo ajustado puede imprimirse, extraerse, dibujarse, etc. utilizando funciones genéricas orientadas a objetos de clase `lm`. Algunas de ellas son:

```
add1     coef       effects   kappa   predict  residuals
alias    deviance   family    labels  print    step
anova    drop1      formula   plot    proj     summary
```

A continuación se incluye una breve descripción de las más utilizadas.

`anova(`*objeto_1*`, `*objeto_2*`)`
: Compara un submodelo con un modelo externo y produce una tabla de análisis de la varianza.

`coefficients(`*objeto*`)`
: Extrae la matriz de coeficientes de regresión.

  Forma reducida: `coef(`*objeto*`)`.

`deviance(`*objeto*`)`
: Suma de cuadrados residual, ponderada si es lo apropiado.

`formula(`*objeto*`)`
: Extrae la fórmula del modelo.

`plot(`*objeto*`)`
: Crea cuatro gráficos que muestran los residuos, los valores ajustados y algunos diagnósticos

`predict(`*objeto*`, newdata=`*hoja.de.datos*`)`
: La nueva hoja de datos que se indica debe tener variables cuyas etiquetas coincidan con las de la original. El resultado es un vector o matriz de valores predichos correspondiente a los valores de las variables de *hoja.de.datos*.

`print(`*objeto*`)`
: Imprime una versión concisa del objeto. A menudo se utiliza implícitamente.

`residuals(`*objeto*`)`
: Extrae la matriz de residuos, ponderada si es necesario.

  La forma reducida es `resid(`*objeto*`)`.

`step(`*objeto*`)`
: Selecciona un modelo apropiado añadiendo o eliminando términos y preservando las jerarquías. Se devuelve el modelo que en este proceso tiene el máximo valor de AIC[1].

`summary(`*objeto*`)`
: Imprime un resumen estadístico completo de los resultados del análisis de regresión.

[1] Acrónimo de Akaike's an Information Criterion

11.4 Análisis de varianza. Comparación de modelos

El análisis de varianza[2] es otra de las técnicas aquí recogidas.

La función de ajuste de modelo `aov(formula.de.modelo, data=`*hoja.de.datos*`)` opera en el nivel más simple de modo muy similar a la función `lm()`, y muchas de las funciones genéricas contenidas en la Sección 11.3 [Funciones genericas de extraccion de informacion del modelo], página 59, le son de aplicación.

Debemos destacar que, además, `aov()` realiza un análisis de modelos estratificados de error múltiple, tales como experimentos split plot, o diseños en bloques incompletos balanceados con recuperación de información inter-bloques. La fórmula

respuesta ∼ *formula.media* `+ Error(`*formula.estratos*`)`

especifica un experimento multiestrato con estratos de error definidos por *formula.estratos*. En el caso más sencillo, *formula.estratos* es un factor, cuando define un experimento con dos estratos, esto es, dentro de y entre los niveles de un factor.

Por ejemplo, si todas las variables predictoras son factores, un modelo como el siguiente:

```
> mf <- aov(cosecha ∼ v + n*p*k +
        Error(granjas/bloques), data=datos.granjas)
```

sería utilizable para describir un experimento con media del modelo `v + n*p*k` y tres estratos de error: "entre granjas", "dentro de granjas, entre bloques" y "dentro de bloques".

11.4.1 Tablas ANOVA

Debe tenerse en cuenta que la tabla ANOVA corresponde a una sucesión de modelos ajustados. Las sumas de cuadrados que en ella aparecen corresponden a la disminución en las sumas de cuadrados residuales como resultado de la inclusión de *un término* concreto en *un lugar* concreto de la sucesión. Por tanto el orden de inclusión sólo será irrelevante en experimentos ortogonales.

Para experimentos multiestrato el procedimiento consiste, en primer lugar, en proyectar la respuesta sobre los estratos de error, una vez más en secuencia, y, después, en ajustar la media del modelo a cada proyección. Para más detalles, consulte Chambers & Hastie (1992).

Una alternativa más flexible a la tabla ANOVA completa es comparar dos o más modelos directamente utilizando la función `anova()`.

`> anova(`*modelo.ajustado.1*`,` *modelo.ajustado.2*`, ...)`

El resultado es una tabla ANOVA que muestra las diferencias entre los modelos ajustados cuando se ajustan en ese orden preciso. Los modelos ajustados objeto de comparación constituyen por tanto una sucesión jerárquica. Este resultado no suministra información distinta a la del caso predeterminado, pero facilita su comprensión y control.

[2] Su acrónimo en inglés es ANOVA, de ANalysis Of Variance. En alguna literatura en español, el acrónimo se sustituye por ANDEVA, de ANálisis DE VArianza.

11.5 Actualización de modelos ajustados

La función `update()` se utiliza muy a menudo para ajustar un modelo que difiere de uno ajustado previamente en unos pocos términos que se añaden o se eliminan. Su forma es:

```
> modelo.nuevo <- update(modelo.anterior, fórmula.nueva)
```

En *fórmula.nueva* puede utilizarse un punto, '`.`', para hacer referencia a "la parte correspondiente de la fórmula del modelo anterior". Por ejemplo,

```
> fm05 <- lm(y ~ x1 + x2 + x3 + x4 + x5, data = produccion)
> fm6  <- update(fm05, . ~ . + x6)
> smf6 <- update(fm6, sqrt(.) ~ .)
```

ajusta una regresión múltiple con cinco variables, (procedentes si es posible) de la hoja de datos `produccion`; después ajusta un modelo adicional incluyendo un sexto regresor; y, por último, ajusta una variante del modelo donde se aplica una transformación raíz cuadrada a la variable predicha.

Advierta especialmente que si especifica el argumento `data` en la llamada original a la función de ajuste del modelo, esta información se pasa a su vez a través del objeto modelo ajustado a la función `update()` y sus asociadas.

El nombre "`.`" puede utilizarse también en otros contextos, pero con un significado levemente distinto. Por ejemplo,

```
> gra.completo <- lm(y ~ . , data=produccion)
```

ajustará un modelo con respuesta `y`, y como variables predictoras, *todas las de la hoja de datos* `produccion`.

Otras funciones que permiten explorar sucesiones crecientes de modelos son `add1()`, `drop1()` y `step()`. Consulte la ayuda para obtener información de las mismas.

11.6 Modelos lineales generalizados

Los modelos lineales generalizados constituyen una extensión de los modelos lineales, para tomar en consideración tanto distribuciones de respuestas no normales como transformaciones para conseguir linealidad, de una forma directa. Un modelo lineal generalizado puede describirse según las siguientes suposiciones:

- Existe una variable respuesta, y, y unas variables estímulo, x_1, x_2, ..., cuyos valores influyen en la distribución de la respuesta.
- Las variables estímulo influyen en la distribución de y mediante *una función lineal solamente*. Esta función lineal recibe el nombre de *predictor lineal*, y se escribe habitualmente
$$\eta = \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_p x_p,$$
por tanto x_i no influye en la distribución de y si y sólo si $\beta_i = 0$.
- La distribución de y es de la forma
$$f_Y(y; \mu, \varphi) = \exp\left[\frac{A}{\varphi}\left\{y\lambda(\mu) - \gamma\left(\lambda(\mu)\right)\right\} + \tau(y, \varphi)\right]$$
donde φ es un *parámetro de escala* (posiblemente conocido) que permanece constante para todas las observaciones, A representa una ponderación a priori que se supone

conocida pero que puede variar con las observaciones, y μ es la media de y. Por tanto, se supone que la distribución de y queda determinada por su media y tal vez un parámetro de escala.

- La media, μ, es una función inversible del predictor lineal:

$$\mu = m(\eta), \qquad \eta = m^{-1}(\mu) = \ell(\mu)$$

y esta función inversa, $\ell()$, se denomina *función de enlace*.

Estas suposiciones son suficientemente amplias para acomodar una amplia clase de modelos útiles en la práctica estadística y, al tiempo, suficientemente estrictas como para permitir el desarrollo de una metodología unificada de estimación e inferencia, al menos aproximadamente. Los interesados en este tema pueden consultar cualquiera de los trabajos de referencia sobre este tema, tales como McCullagh & Nelder (1989) o Dobson (1990).

11.6.1 Familias

La clase de modelos lineales generalizados que pueden ser tratados en R incluye las distribuciones de respuesta *gaussian* (normal), *binomial*, *poisson*, *inverse gaussian* (normal inversa) y *gamma* así como los modelos de *quasi-likelihood* (cuasi-verosimilitud) cuya distribución de respuesta no está explícitamente definida. En este último caso debe especificarse la *función de varianza* como una función de la media, pero en el resto de casos esta función está implícita en la distribución de respuesta.

Cada distribución de respuesta admite una variedad de funciones de enlace para conectar la media con el predictor lineal. La tabla siguiente recoge las que están disponibles automáticamente.

Nombre de la familia	Función de enlace
`binomial`	`logit`, `probit`, `cloglog`
`gaussian`	`identity`
`Gamma`	`identity`, `inverse`, `log`
`inverse.gaussian`	`1/mu^2`
`poisson`	`identity`, `log`, `sqrt`
`quasi`	`logit`, `probit`, `cloglog`, `identity`, `inverse`, `log`, `1/mu^2`, `sqrt`

La combinación de una distribución de respuesta, una función de enlace y otras informaciones que son necesarias para llevar a cabo la modelización se denomina *familia* del modelo lineal generalizado.

11.6.2 La función `glm`

Puesto que la distribución de la respuesta depende de las variables de estímulo *solamente* a través de una función lineal, se puede utilizar el mismo mecanismo de los modelos lineales para especificar la parte lineal de un modelo generalizado. Sin embargo la familia debe especificarse de modo distinto.

La función `glm()` permite ajustar un modelo lineal generalizado y tiene la forma siguiente

```
> modelo.ajustado <- glm(formula.de.modelo, family =
  generador.de.familia, data=hoja.datos)
```

La única característica nueva es *generador.de.familia* que es el instrumento mediante el que se describe la familia. Es el nombre de una función que genera una lista de funciones y expresiones que, juntas, definen y controlan el modelo y el proceso de estimación. Aunque puede parecer complicado a primera vista, su uso es bastante sencillo.

Los nombres de los generadores de familias estándar suministrados con R aparecen en la tabla de la Sección 11.6.1 [Familias], página 62 con la denominación de "Nombre de la familia". Si además debe seleccionar una función de enlace, debe indicarla como un parámetro, entre paréntesis, del nombre de la familia. En el caso de la familia `quasi`, la función de varianza puede especificarse del mismo modo.

Veamos algunos ejemplos.

La familia gaussiana (`gaussian`)

Una expresión de la forma

```
> mf <- glm(y ~ x1 + x2, family = gaussian, data = ventas)
```

obtiene el mismo resultado que

```
> mf <- lm(y ~ x1+x2, data=ventas)
```

pero con menor eficiencia. Tenga en cuenta que la familia gaussiana no dispone automáticamente de una serie de funciones de enlace, por lo que no admite parámetros. Si en un problema necesita utilizar la familia gaussiana con un enlace no estándar, la solución pasa por el uso de la familia `quasi`, como veremos posteriormente.

La familia binomial (`binomial`)

Consideremos el siguiente ejemplo artificial, tomado de Silvey (1970).

Los hombres de la isla de Kalythos, en el mar Egeo, sufren una enfermedad ocular congénita cuyos efectos se acrecientan con la edad. Se tomó una muestra de varios isleños de diferentes edades cuyos resultados se muestran a continuación:

Edad:	20	35	45	55	70
No. sujetos:	50	50	50	50	50
No. invidentes:	6	17	26	37	44

Consideramos el problema de ajustar un modelo logístico y otro probit a estos datos, y estimar en cada modelo el parámetro LD50, correspondiente a la edad en que la probabilidad de ceguera es del 50%.

Si y es el número de invidentes a la edad x, y n es el número de sujetos estudiados, ambos modelos tienen la forma

$$y \sim \mathrm{B}(n, F(\beta_0 + \beta_1 x))$$

donde, para el caso probit, $F(z) = \Phi(z)$ es la función de distribución normal (0,1), y en el caso logit (que es el predeterminado), $F(z) = e^z/(1 + e^z)$. En ambos casos, LD50 se define como

$$\mathrm{LD50} = -\beta_0/\beta_1$$

Esto es, el punto en que el argumento de la función de distribución es cero.

El primer paso es preparar los datos en una hoja de datos

```
> kalythos <- data.frame(x = c(20,35,45,55,70), n = rep(50,5),
                         y = c(6,17,26,37,44))
```

Para ajustar un modelo binomial utilizando `glm()` existen dos posibilidades para la respuesta:

- Si la respuesta es un *vector*, entonces debe corresponder a datos *binarios* y por tanto sólo debe contener ceros y unos.
- Si la respuesta es una *matriz de dos columnas*, la primera columna debe contener el número de éxitos y la segunda el de fracasos.

Aquí vamos a utilizar la segunda de estas convenciones, por lo que debemos añadir una matriz a la hoja de datos:

```
> kalythos$Ymat <- cbind(kalythos$y, kalythos$n - kalythos$y)
```

Para ajustar los modelos utilizamos

```
> fmp <- glm(Ymat ~ x, family = binomial(link=probit), data = kalythos)
> fml <- glm(Ymat ~ x, family = binomial, data = kalythos)
```

Puesto que la función de enlace logit es la predeterminada, este parámetro puede omitirse en la segunda expresión. Para ver los resultados de cada ajuste usaremos

```
> summary(fmp)
> summary(fml)
```

Ambos modelos se ajustan bien (demasiado bien). Para estimar LD50 podemos usar la siguiente función:

```
> ld50 <- function(b) -b[1]/b[2]
> ldp <- ld50(coef(fmp)); ldl <- ld50(coef(fmp)); c(ldp, ldl)
```

y obtendremos los valores 43.663 y 43.601 respectivamente.

Modelos de Poisson (`poisson`)

Para la familia `poisson` el enlace predeterminado es `log`, y en la práctica el uso fundamental de esta familia es ajustar modelos loglineales de Poisson a datos de frecuencias, cuya distribución en sí es a menudo multinomial. Este es un tema amplio e importante que no discutiremos aquí y que incluso constituye una parte fundamental de la utilización de modelos generalizados no gaussianos.

A veces surgen datos auténticamente poissonianos que, en el pasado se analizaban a menudo como datos gaussianos tras aplicarles una transformación logarítmica o de raíz cuadrada. Como alternativa al último, puede ajustarse un modelo lineal generalizado de Poisson, como en el siguiente ejemplo:

```
> fmod <- glm(y ~ A + B + x, family = poisson(link=sqrt),
              data = frec.gusanos)
```

Modelos de cuasi-verosimilitud (quasi)

En todas las familias, la varianza de la respuesta dependerá de la media, y el parámetro de escala actuará como un multiplicador. La forma de dependencia de la varianza respecto de la media es una característica de la distribución de respuesta, por ejemplo para la distribución de Poisson será $\mathrm{Var}[y] = \mu$.

Para estimación e inferencia de cuasi-verosimilitud, la distribución de respuesta precisa no está especificada, sino más bien sólo la función de enlace y la forma en que la función de varianza depende de la media. Puesto que la estimación de cuasi-verosimilitud utiliza formalmente las mismas técnicas de la distribución gaussiana, esta familia permite ajustar modelos gaussianos con funciones de enlace no estándar o incluso con funciones de varianza.

Por ejemplo, consideremos la regresión no lineal siguiente

$$y = \frac{\theta_1 z_1}{z_2 - \theta_2} + e$$

que puede escribirse también de la forma

$$y = \frac{1}{\beta_1 x_1 + \beta_2 x_2} + e$$

donde $x_1 = z_2/z_1$, $x_2 = -1/x_1$, $\beta_1 = 1/\theta_1$ y $\beta_2 = \theta_2/\theta_1$. Suponiendo que existe una hoja de datos apropiada, *bioquimica*, podemos ajustar este modelo mediante

```
> nlfit <- glm(y ~ x1 + x2 - 1,
               family = quasi(link=inverse, variance=constant),
               data = bioquimica)
```

Si desea mayor información, lea las ayudas correspondientes.

11.7 Modelos de Mínimos cuadrados no lineales y de Máxima verosimilitud

Ciertas formas de modelos no lineales pueden ajustarse mediante Modelos Lineales Generalizados, con la función `glm()`, pero en la mayoría de los casos será necesario utilizar optimización no lineal. La función que lo realiza en R es `nlm()`, que reemplaza aquí a las funciones `ms()` y `nlmin()` de S-Plus. Buscamos los valores de los parámetros que minimizan algún índice de falta de ajuste y `nlm()` lo resuelve probando varios parámetros iterativamente. Al contrario que en la regresión lineal, por ejemplo, no existe garantía de que el procedimiento converja a unos estimadores satisfactorios. La función `nlm()` necesita unos valores iniciales de los parámetros a estimar y la convergencia depende críticamente de la calidad de dichos valores iniciales.

11.7.1 Mínimos cuadrados

Una forma de ajustar un modelo no lineal es minimizar la suma de los cuadrados de los errores o residuos (SSE). Este método tiene sentido si los errores observados pueden proceder de una distribución normal.

Presentamos un ejemplo debido a Bates & Watts (1988), página 51. Los datos son:

```
> x <- c(0.02, 0.02, 0.06, 0.06, 0.11, 0.11, 0.22, 0.22, 0.56, 0.56,
         1.10, 1.10)
> y <- c(76, 47, 97, 107, 123, 139, 159, 152, 191, 201, 207, 200)
```

El modelo a ajustar es:

```
> fn <- function(p) sum((y - (p[1] * x)/(p[2] + x))^2)
```

Para realizar el ajuste necesitamos unos valores iniciales de los parámetros. Una forma de encontrar unos valores iniciales apropiados es representar gráficamente los datos, conjeturar unos valores de los parámetros, y dibujar sobre los datos la curva correspondiente a estos valores.

```
> plot(x, y)
> xajustado <- seq(.02, 1.1, .05)
> yajustado <- 200*xajustado/(.1+xajustado)
> lines(spline(xajustado,yajustado))
```

Aunque se podría tratar de encontrar unos valores mejores, los valores obtenidos, 200 y 0.1, parecen adecuados. Ya podemos realizar el ajuste:

```
> resultado <- nlm(fn,p=c(200,.1),hessian=TRUE)
```

Tras el ajuste, `resultado$minimum` contiene SSE, y `resultado$estimates` contiene los estimadores por mínimos cuadrados de los parámetros. Para obtener los errores típicos aproximados de los estimadores (SE) escribimos lo siguiente:

```
> sqrt(diag(2*resultado$minimum/(length(y) - 2) *
       solve(resultado$hessian)))
```

El número 2 en dicha expresión representa el número de parámetros. Un intervalo de confianza al 95% será: El estimador del parámetro $\pm$ 1.96 SE. Podemos representar el ajuste en un nuevo gráfico:

```
> plot(x,y)
> xajustado <- seq(.02,1.1,.05)
> yajustado <- 212.68384222*xajustado/(0.06412146+xajustado)
> lines(spline(xajustado,yajustado))
```

La biblioteca **nls** contiene muchas más posibilidades para ajustar modelos no lineales por mínimos cuadrados. El modelo que acabamos de ajustar es el modelo de Michaelis-Menten, por tanto podemos usar

```
> df <- data.frame(x=x, y=y)
> fit <- nls(y ~ SSmicmen(x, Vm, K), df)
> fit
Nonlinear regression model
  model:  y ~ SSmicmen(x, Vm, K)
   data:  df
         Vm            K
212.68370711   0.06412123
 residual sum-of-squares:  1195.449
> summary(fit)

Formula: y ~ SSmicmen(x, Vm, K)

Parameters:
    Estimate Std. Error t value Pr(>|t|)
```

```
Vm 2.127e+02  6.947e+00  30.615 3.24e-11
K  6.412e-02  8.281e-03   7.743 1.57e-05

Residual standard error: 10.93 on 10 degrees of freedom

Correlation of Parameter Estimates:
      Vm
K 0.7651
```

11.7.2 Máxima verosimilitud

El método de máxima verosimilitud es un método de ajuste de un modelo no lineal que puede aplicarse incluso cuando los errores no son normales. El método busca los valores de los parámetros que maximizan la log-verosimilitud o, lo que es igual, minimizan el valor de la -log-verosimilitud. El siguiente ejemplo, tomado de Dobson (1990), pp. 108-11, ajusta un modelo logístico a los datos de dosis-respuesta, que claramente también podría ajustarse utilizando la función `glm()`. Los datos son:

```
> x <- c(1.6907, 1.7242, 1.7552, 1.7842, 1.8113,
         1.8369, 1.8610, 1.8839)
> y <- c( 6, 13, 18, 28, 52, 53, 61, 60)
> n <- c(59, 60, 62, 56, 63, 59, 62, 60)
```

La -log-verosimilitud a minimizar es:

```
> fn <- function(p)
   sum( - (y*(p[1]+p[2]*x) - n*log(1+exp(p[1]+p[2]*x))
           + log(choose(n, y)) ))
```

Elegimos unos valores iniciales y realizamos el ajuste:

```
> out <- nlm(fn, p = c(-50,20), hessian = TRUE)
```

Tras lo cual, `resultado$minimum` contiene la -log-verosimilitud, y `resultado$estimates` contiene los estimadores de máxima verosimilitud. Para obtener los SE aproximados de los parámetros, escribimos:

```
> sqrt(diag(solve(out$hessian)))
```

El intervalo de confianza al 95% es: El estimador del parámetro $\pm$ 1.96 SE.

11.8 Algunos modelos no-estándar

Concluimos esta parte con una breve mención a otras posibilidades de R para regresión especial y análisis de datos.

- **Modelos mezclados.** La biblioteca creada por usuarios, **nlme**, contiene las funciones `lme` y `nlme` para modelos de efectos mezclados lineales y no lineales, esto es, regresiones lineales y no lineales en las cuales algunos coeficientes corresponden a efectos aleatorios. Estas funciones hacen un uso intensivo de las fórmulas para especificar los modelos.
- **Regresión con aproximación local.** La función `loess()` ajusta una regresión no paramétrica utilizando regresión polinomial localmente ponderada. Este tipo de regresión es útil para poner de relieve una tendencia en datos confusos o para reducir datos y obtener alguna luz sobre la estructura de grandes conjuntos de datos.

La biblioteca **modreg** contiene la función `loess` así como posibilidades de regresión "projection pursuit".

- **Regresión robusta.** Existen varias funciones para ajustar modelos de regresión resistentes a la influencia de valores anómalos (outliers) en los datos. La función `lqs` en la biblioteca del mismo nombre contiene los algoritmos más recientes para ajustes altamente resistentes. Otras funciones menos resistentes pero más eficientes estadísticamente se encuentran en otras bibliotecas creadas por usuarios, por ejemplo la función `rlm` en la biblioteca **MASS**.
- **Modelos aditivos.** Esta técnica intenta construir una función de regresión a partir de funciones aditivas suaves de las variables predictoras, habitualmente una por cada variable predicha. Las funciones `avas` y `ace` en la biblioteca **acepack** y las funciones `bruto` y `mars` de la biblioteca **mda** son ejemplos de estas técnicas contenidas en bibliotecas creadas por usuarios para R.
- **Modelos basados en árboles.** En vez de buscar un modelo lineal global explícito para predicción o interpretación, los modelos basados en árboles intentan bifurcar los datos, recursivamente, en puntos críticos de las variables predictoras con la finalidad de conseguir una partición de los datos en grupos tan homogéneos dentro del grupo y tan heterogéneos de un grupo a otro, como sea posible. Los resultados a menudo conducen a una comprensión de los datos que otros métodos de análisis de datos no suelen suministrar.

  Los modelos se especifican en la forma de un modelo lineal ordinario. La función de ajuste es `tree()`, y muchas funciones genéricas, como `plot()` y `text()` pueden mostrar los resultados de este ajuste de modo gráfico.

  Puede encontrar estos modelos en las bibliotecas creadas por usuarios **rpart** y **tree**.

12 Procedimientos gráficos

Las posibilidades gráficas son un componente de R muy importante y versátil. Es posible utilizarlas para mostrar una amplia variedad de gráficos estadísticos y también para construir nuevos tipos de gráficos.

Los gráficos pueden usarse tanto en modo interactivo como no interactivo, pero en la mayoría de los casos el modo interactivo es más productivo. Además al iniciar R en este modo, se activa un dispositivo para mostrar gráficos. Aunque este paso es automático, es útil conocer que la orden es `X11()`, aunque también puede usar `windows()` en Microsoft Windows.

Las órdenes gráficas se dividen en tres grupos básicos:

- **Alto nivel** Son funciones que crean un nuevo gráfico, posiblemente con ejes, etiquetas, títulos, etc..
- **Bajo nivel** Son funciones que añaden información a un gráfico existente, tales como puntos adicionales, líneas y etiquetas.
- **Interactivas** Son funciones que permiten interactuar con un gráfico, añadiendo o eliminando información, utilizando un dispositivo apuntador, como un ratón.

Además, en R existe una lista de *parámetros gráficos* que pueden utilizarse para adaptar los gráficos.

12.1 Funciones gráficas de nivel alto

Las órdenes gráficas de nivel alto están diseñadas para generar un gráfico completo a partir de unos datos pasados a la función como argumento. Cuando es necesario se generan automáticamente ejes, etiquetas o títulos (salvo que se especifique lo contrario). Estas órdenes comienzan siempre un nuevo gráfico, borrando el actual si ello es necesario.

12.1.1 La función `plot`

Una de las funciones gráficas más utilizadas en R es `plot`, que es una función *genérica*, esto es, el tipo de gráfico producido es dependiente de la *clase* del primer argumento.

`plot(`*x*`, `*y*`)`
`plot(`*xy*`)` Si *x* e *y* son vectores, `plot(`*x*`, `*y*`)` produce un diagrama de dispersión de *y* sobre *x*. El mismo efecto se consigue suministrando un único argumento (como se ha hecho en la segunda forma) que sea bien una lista con dos elementos, *x* e *y*, bien una matriz con dos columnas.

`plot(`*x*`)` Si *x* es una serie temporal, produce un gráfico temporal, si *x* es un vector numérico, produce un gráfico de sus elementos sobre el índice de los mismos, y si *x* es un vector complejo, produce un gráfico de la parte imaginaria sobre la real de los elementos del vector.

`plot(`*f*`)`
`plot(`*f*`, `*y*`)`
Sean *f* un factor, e *y* un vector numérico. La primera forma genera un diagrama de barras de *f*; la segunda genera diagramas de cajas de *y* para cada nivel de *f*.

`plot(`*hd*`)`
`plot(`$\sim$ *expr*`)`
`plot(`*y* $\sim$ *expr*`)`

Sean *hd*, una hoja de datos; *y*, un objeto cualquiera; y *expr*, una lista de nombres de objetos separados por símbolos '`+`' (por ejemplo, `a + b + c`). Las dos primeras formas producen diagramas de todas las parejas de variables de la hoja de datos *hd* (en el primer caso) y de los objetos de la expresión *expr* (en el segundo caso). La tercera forma realiza sendos gráficos de *y* sobre cada objeto nombrado en la expresión *expr* (uno para cada objeto).

12.1.2 Representación de datos multivariantes

R posee dos funciones muy útiles para representar datos multivariantes. Si `X` es una matriz numérica o una hoja de datos, la orden

```
> pairs(X)
```

produce una matriz de gráficos de dispersión para cada pareja de variables definidas por las columnas de `X`, esto es, cada columna de `X` se representa frente a cada una de las demás columnas, y los $n(n-1)$ gráficos se presentan en una matriz de gráficos con escalas constantes sobre las filas y columnas de la matriz.

Cuando se trabaja con tres o cuatro variables, la función `coplot` puede ser más apropiada. Si `a` y `b` son vectores numéricos y `c` es un vector numérico o un factor (todos de la misma longitud) entonces la orden

```
> coplot(a ~ b | c)
```

produce diagramas de dispersión de `a` sobre `b` para cada valor de `c`. Si `c` es un factor, esto significa que `a` se representa sobre `b` para cada nivel de `c`. Si `c` es un vector numérico, entonces se agrupa en *intervalos* y para cada intervalo se representa `a` sobre `b` para los valores de `c` dentro del intervalo. El número y tamaño de los intervalos puede controlarse con el argumento `given.values` de la función `coplot()`. La función `co.intervals()` también es útil para seleccionar intervalos. Asimismo, es posible utilizar dos variables *condicionantes* con una orden como

```
> coplot(a ~ b | c + d)
```

que produce diagramas de `a` sobre `b` para cada intervalo de condicionamiento de `c` y `d`.

Las funciones `coplot()` y `pairs()` utilizan el argumento `panel` para personalizar el tipo de gráfico que aparece en cada panel o recuadro. El valor predeterminado es `points()` para producir un diagrama de dispersión, pero si se introducen otras funciones gráficas de nivel bajo de los dos vectores `x` e `y` como valor de `panel`, se produce cualquier tipo de gráfico que se desee. Una función útil en este contexto es `panel.smooth()`.

12.1.3 Otras representaciones gráficas

Existen otras funciones gráficas de nivel alto que producen otros tipos de gráficos. Algunas de ellas son las siguientes:

`qqnorm(x)`
`qqline(x)`
`qqplot(x, y)`
: Gráficos de comparación de distribuciones. El primero representa el vector `x` sobre los valores esperados normales. El segundo le añade una recta que pasa por los cuartiles de la distribución y de los datos. El tercero representa los cuantiles de `x` sobre los de `y` para comparar sus distribuciones respectivas.

`hist(x)`
`hist(x, nclass=`*n*`)`
`hist(x, breaks=`*b*`, ...)`
: Produce un histograma del vector numérico `x`. El número de clases se calcula habitualmente de modo correcto, pero puede elegir uno con el argumento `nclass`, o bien especificar los puntos de corte con el argumento `breaks`. Si está presente el argumento `probability=TRUE`, se representan frecuencias relativas en vez de absolutas.

`dotplot(x, ...)`
: Construye un gráfico de puntos de `x`. En este tipo de gráficos, el eje y etiqueta los datos de `x` y el eje x da su valor. Por ejemplo, permite una selección visual sencilla de todos los elementos con valores dentro de un rango determinado.

`image(x, y, z, ...)`
`contour(x, y, z, ...)`
`persp(x, y, z, ...)`
: Gráficos tridimensionales. `image` representa una retícula de rectángulos con colores diferentes según el valor de `z`, `contour` representa curvas de nivel de `z`, y `persp` representa una superficie tridimensional de `z`.

12.1.4 Argumentos de las funciones gráficas de nivel alto

Existe una serie de argumentos que pueden pasarse a las funciones gráficas de nivel alto, entre otros los siguientes:

`add=TRUE` Obliga a la función a comportarse como una función a nivel bajo, de modo que el gráfico que genere se superpondrá al gráfico actual, en vez de borrarlo (sólo en algunas funciones)

`axes=FALSE`
: Suprime la generación de ejes. Útil para crear ejes personalizados con la función `axis`. El valor predeterminado es `axes=TRUE`, que incluye los ejes.

`log="x"`
`log="y"`
`log="xy"` Hace que el eje x, el eje y, o ambos ejes, sean logarítmicos. En algunos gráficos no tiene efecto.

`type=` Este argumento controla el tipo de gráfico producido, de acuerdo a las siguientes posibilidades:

`type="p"` Dibuja puntos individuales. Este es el valor predeterminado

`type="l"` Dibuja líneas

`type="b"` Dibuja puntos y líneas que los unen

`type="o"` Dibuja puntos y líneas que los unen, cubriéndolos

`type="h"` Dibuja líneas verticales desde cada punto al eje X

`type="s"`
`type="S"` Dibuja un gráfico de escalera. En la primera forma, la escalera comienza hacia la derecha, en la segunda, hacia arriba.

`type="n"` No se realiza ningún gráfico, aunque se dibujan los ejes (salvo indicación en contra) y se prepara el sistema de coordenadas de acuerdo a los datos. Suele utilizarse para crear gráficos en los que a continuación se utilizarán órdenes de nivel bajo.

`xlab=`*cadena*
`ylab=`*cadena*
Definen las etiquetas de los ejes x e y, en vez de utilizar las etiquetas predeterminadas, que normalmente son los nombres de los objetos utilizados en la llamada a la función gráfica de nivel alto.

`main=`*cadena*
Título del gráfico, aparece en la parte superior con tamaño de letra grande.

`sub=`*cadena*
Subtítulo del gráfico, aparece debajo del eje x con tamaño de letra pequeño.

12.2 Funciones gráficas de nivel bajo

A veces las funciones gráficas de nivel alto no producen exactamente el tipo de gráfico deseado. En este caso pueden añadirse funciones gráficas de nivel bajo para añadir información adicional (tal como puntos, líneas o texto) al gráfico actual.

Algunas de las funciones gráficas de nivel bajo más usuales son:

`points(x, y)`
`lines(x, y)`
Añaden puntos o líneas conectadas al gráfico actual. El argumento `type` de la función `plot()` puede pasarse a esta funciones (y su valor predeterminado es `"p"` para `points` y `"l"` para `lines`.)

`text(x, y, etiquetas, ...)`
Añade texto al gráfico en las coordenadas `x, y`. Normalmente, `etiquetas`, es un vector de enteros o de caracteres, en cuyo caso, `etiquetas[i]` se dibuja en el punto `(x[i], y[i])`. El valor predeterminado es `1:length(x)`.

Nota: Esta función se utiliza a menudo en la secuencia

```
> plot(x, y, type="n"); text(x, y, nombres)
```

El parámetro gráfico `type="n"` suprime los puntos pero construye los ejes, y la función `text` permite incluir caracteres especiales para representar los puntos, como se especifica en el vector de caracteres `nombres`.

`abline(a, b)`
`abline(h=`*y*`)`
`abline(v=`*x*`)`
`abline(`*lm.obj*`)`
: Añade al gráfico actual, una recta de pendiente `b` y ordenada en el origen `a`. La forma `h=`*y* representa una recta horizontal de altura `y`, y la forma `v=`*x*, una similar, vertical. En el cuarto caso, *lm.obj* puede ser una lista con un componente `coefficients` de longitud 2 (como el resultado de una función de ajuste de un modelo) que se interpretan como ordenada y pendiente, en ese orden.

`polygon(x, y, ...)`
: Añade al gráfico actual un polígono cuyos vértices son los elementos de (`x`,`y`); (opcionalmente) sombreado con líneas, o relleno de color si el periférico lo admite.

`legend(x, y, letrero, ...)`
: Añade al gráfico actual un letrero o leyenda, en la posición especificada. Los caracteres para dibujar, los estilos de líneas, los colores, etc. están identificados con los elementos del vector `letrero`. Debe darse al menos otro argumento más, *v*, un vector de la misma longitud que `letrero`, con los correspondientes valores de dibujo, como sigue:

  `legend( , fill=`*v*`)`
  : Colores para rellenar

  `legend( , col=`*v*`)`
  : Colores de puntos y líneas

  `legend( , lty=`*v*`)`
  : Tipos de línea

  `legend( , lwd=`*v*`)`
  : Anchura de línea

  `legend( , pch=`*v*`)`
  : Caracteres para dibujar (vector de caracteres)

`title(main, sub)`
: Añade un título, `main`, en la parte superior del gráfico actual, de tamaño grande, y un subtítulo, `sub`, en la parte inferior, de tamaño menor.

`axis(side, ...)`
: Añade al gráfico actual un eje en el lado indicado por el primer argumento (de 1 a 4, en el sentido de las agujas del reloj, siendo el 1 la parte inferior). Otros argumentos controlan la posición de los ejes, dentro o fuera del gráfico, las marcas y las etiquetas. Es útil para añadir ejes tras utilizar la función `plot()` con el argumento `axes=FALSE`.

Las funciones gráficas de nivel bajo necesitan normalmente alguna información de posición, como las coordenadas x e y, para determinar dónde colocar los nuevos elementos. Las coordenadas se dan en términos de *coordenadas de usuario*, las cuales están definidas

por las funciones gráficas de alto nivel previas y se toman en función de los datos suministrados a estas funciones.

Los dos argumentos, `x` e `y`, pueden sustituirse por un solo argumento de clase lista con dos componentes llamados `x` e `y`, o por una matriz con dos columnas. De este modo, funciones como `locator()`, que vemos a continuación, pueden usarse para especificar interactivamente posiciones en un gráfico.

12.2.1 Anotaciones matemáticas

En muchas ocasiones es conveniente añadir símbolos matemáticos y fórmulas a un gráfico. En R es posible hacerlo especificando una *expresión*, en vez de una cadena de caracteres, en cualquiera de las funciones `text`, `mtext`, `axis` o `title`. Por ejemplo, la orden siguiente dibuja la fórmula de la distribución binomial en la posición x, y:

```
> text(x, y, expression(paste(bgroup("(", atop(n, x), ")"),
                              p^x, q^{n-x})))
```

Puede obtener información detallada con las órdenes:

```
> help(plotmath)
> example(plotmath)
```

12.2.2 Fuentes vectoriales Hershey

Es posible escribir texto utilizando las fuentes vectoriales Hershey en las funciones `text` y `contour`. Existen tres razones para utilizar estas fuentes:

- Producen mejores resultados, especialmente en pantalla, con textos rotados o de pequeño tamaño.
- Contienen símbolos que pueden no estar disponibles en las fuentes ordinarias, como signos del zodiaco, cartográficos o astronómicos.
- Contienen caracteres cirílicos y japoneses (Kana y Kanji).

La información detallada, incluyendo las tablas de caracteres, puede obtenerla con las órdenes:

```
> help(Hershey)
> example(Hershey)
> help(Japanese)
> example(Japanese)
```

12.3 Funciones gráficas interactivas

R dispone de funciones que permiten al usuario extraer o añadir información a un gráfico utilizando el ratón. La más sencilla es la función `locator()`:

`locator(n, type)`

: Permite que el usuario seleccione posiciones del gráfico actual, fundamentalmente, utilizando el botón primario del ratón, hasta que haya seleccionado `n` puntos (el valor predeterminado son 512) o pulse el botón secundario. El argumento `type` permite dibujar utilizando los puntos seleccionados con el mismo significado que en las funciones de nivel alto. Su valor predeterminado es no

dibujar. La función `locator()` devuelve las coordenadas de los puntos seleccionados en una lista con dos componentes, `x` e `y`.

La función `locator()` suele utilizarse sin argumentos. Es particularmente útil para seleccionar interactivamente posiciones para elementos gráficos tales etiquetas cuando es difícil calcular previamente dónde deben colocarse. Por ejemplo, para colocar un texto informativo junto a un punto anómalo, la orden

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

puede ser útil. En el caso de que no se disponga de ratón, la función `locator()` aún puede ser utilizada; en este caso se preguntarán al usuario las coordenadas de x e y.

`identify(x, y, etiquetas)`
: Permite identificar cualquiera de los puntos definidos por `x` e `y` utilizando el botón primario del ratón, dibujando la correspondiente componente de `etiquetas` al lado (o el índice del punto si no existe `etiquetas`). Al pulsar el botón secundario del ratón, devuelve los índices de los puntos seleccionados.

A veces interesa identificar *puntos* particulares en un gráfico y no sus posiciones. Por ejemplo, si queremos que el usuario seleccione una observación de interés y, posteriormente, manipularla en algún modo. Dado un número de coordenadas, (x, y), en dos vectores numéricos, `x` e `y`, podríamos usar la función `identify()` del siguiente modo:

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

La función `identify()` no realiza ningún gráfico por sí misma sino que permite al usuario mover el puntero del ratón y pulsar junto a un punto. El punto más próximo al puntero (si está suficientemente próximo) se identificará dibujando junto a él su número índice, esto es, la posición que ocupa en los vectores `x` e `y`. Alternativamente podría haber utilizado una cadena de caracteres (como por ejemplo el nombre del caso) para la identificación, utilizando el argumento `etiquetas`, o inhabilitar la identificación utilizando el argumento `plot=FALSE`. Cuando se pulsa el botón secundario, la función devuelve los índices de los puntos seleccionados, que podrán ser utilizados para obtener los puntos correspondientes de los vectores `x` e `y`.

12.4 Uso de parámetros gráficos

Al crear gráficos, especialmente con fines de presentación o publicación, es posible que R no produzca de modo automático la apariencia exacta que se desea. Ahora bien, es posible personalizar cada aspecto del gráfico utilizando *parámetros gráficos.* R dispone de muchos parámetros gráficos que controlan aspectos tales como estilo de línea, colores, disposición de las figuras y justificación del texto entre otros muchos. Cada parámetro gráfico tiene un nombre (por ejemplo, '`col`', que controla los colores) y toma un valor (por ejemplo, `"blue"`, para indicar el color azul).

Por cada dispositivo gráfico activo, se mantiene una lista de parámetros gráficos, y cada dispositivo gráfico dispone de un conjunto predeterminado de parámetros cuando se inicializa. Los parámetros gráficos pueden indicarse de dos modos; bien de modo permanente, lo que afectará a todas las funciones gráficas que accedan al dispositivo gráfico, bien temporalmente, lo que sólo afecta a la función gráfica que lo utiliza en ese momento.

12.4.1 Cambios permanentes. La función `par()`

La función `par()` se utiliza para acceder a la lista de parámetros gráficos del dispositivo gráfico actual y para modificarla.

`par()` Sin argumentos, devuelve una lista de todos los parámetros gráficos y sus valores, para el dispositivo gráfico actual.

`par(c("col", "lty"))`
Con un vector de caracteres como argumento, devuelve una lista que contiene sólo los valores de los parámetros citados.

`par(col=4, lty=2)`
Con argumentos con nombre (o una lista como argumento) establece los valores de estos parámetros y devuelve (de modo invisible) una lista con los valores originales de los parámetros.

Al modificar cualquier parámetro con la función `par()`, la modificación es *permanente*, en el sentido de que cualquier llamada a una función gráfica (en el mismo dispositivo gráfico) vendrá afectada por dicho valor. Puede pensarse que este tipo de asignación equivale a modificar los valores predeterminados de los parámetros que utilizarán las funciones gráficas salvo que se les indique un valor alternativo.

Las llamadas a la función `par()` *siempre* afectan a los valores globales de los parámetros gráficos, incluso aunque la llamada se realice desde una función. Esta conducta a menudo no es satisfactoria; habitualmente desearíamos modificar algunos parámetros, realizar algunos gráficos y volver a los valores originales para no afectar a la sesión completa de R. Este trabajo recae en el usuario, que debe almacenar el resultado de `par()` cuando realice algún cambio y restaurarlo cuando el gráfico esté completo.

```
> par.anterior <- par(col=4,lty=2)
  ... varias órdenes gráficas ...
> par(par.anterior)
```

12.4.2 Cambios temporales. Argumentos de las funciones gráficas

Los parámetros gráficos también pueden pasarse a prácticamente todas las funciones gráficas como argumentos con nombre, lo que tiene el mismo efecto que utilizarlos en la función `par()`, excepto que los cambios sólo existen durante la llamada a la función. Por ejemplo:

```
> plot(x, y, pch="+")
```

realiza un diagrama de dispersión utilizando el signo de sumar, `+`, para representar cada punto, sin cambiar el carácter predeterminado para gráficos posteriores.

12.5 Parámetros gráficos habituales

A continuación se detallan muchos de los parámetros gráficos habituales. La ayuda de la función `par()` contiene un resumen más conciso; por lo que puede considerar esta descripción como una alternativa más detallada.

Los parámetros gráficos se presentarán en la siguiente forma:

nombre=*valor*
: Descripción del efecto del parámetro. *nombre* es el nombre del parámetro, esto es, el nombre del argumento que debe usar en la función `par()` o cualquier función gráfica. *valor* es un valor típico del parámetro.

12.5.1 Elementos gráficos

Los gráficos de R están formados por puntos, líneas, texto y polígonos (regiones rellenas). Existen parámetros gráficos que controlan cómo se dibujan los *elementos gráficos* citados, como los siguientes:

`pch="+"`
: Carácter que se utiliza para dibujar un punto. El valor predeterminado varía entre dispositivos gráficos, pero normalmente es '○'. Los puntos tienden a aparecer en posición levemente distinta de la exacta, salvo que utilice el carácter `"."` que produce puntos centrados.

`pch=4`
: Si `pch` toma un valor entre 0 y 18, ambos inclusive, se utiliza un símbolo especial para realizar los gráficos. Para saber cuales son los símbolos, utilice las órdenes

```
> plot(1,t="n")
> legend(locator(1), as.character(0:18), marks=0:18)
```

y pulse en la parte superior del gráfico.

`lty=2`
: Es el tipo de línea. Aunque algunos tipos no pueden dibujarse en determinados dispositivos gráficos, siempre, el tipo 1 corresponde a una línea continua, y los tipos 2 o superior corresponden a líneas con puntos, rayas o combinaciones de ambos.

`lwd=2`
: Es la anchura de línea, medida en múltiplos de la anchura "base". Afecta tanto a los ejes como a las líneas dibujadas con la función `lines()`, etc.

`col=2`
: Es el color que se utiliza para los puntos, líneas, texto, imágenes y relleno de zonas. Cada uno de estos elementos gráficos admite una lista de colores posibles y el valor de este parámetro es un índice para esta lista. Obviamente este parámetro solo tiene aplicación en algunos dispositivos.

`font=2`
: Es un entero que indica qué fuente se utilizará para el texto. Si es posible, `1` corresponde a texto normal, `2` a negrilla, `3` a itálica y `4` a itálica negrilla.

`font.axis`
`font.lab`
`font.main`
`font.sub`
: Es la fuente que se utilizará, respectivamente, para escribir en los ejes, para el etiquetado de x e y, y para el título y el subtítulo.

`adj=-0.1`
: Indica cómo debe justificarse el texto respecto de la posición de dibujo. Un `0` indica justificación a la izquierda, un `1` indica justificación a la derecha, y un `0.5` indica centrado. Puede usar cualquier otro valor que indicará la proporción de texto que aparece a la izquierda de la posición de dibujo, por tanto un valor de `-0.1` dejará un 10% de la anchura del texto entre el mismo y la posición de dibujo.

`cex=1.5`
: Es el carácter de expansión. Su valor indica el tamaño de los caracteres de texto (incluidos los caracteres de dibujo) respecto del tamaño predeterminado.

12.5.2 Ejes y marcas de división

Muchos de los gráficos de nivel alto en R tienen ejes, pero, además, siempre es posible construirlos con la función gráfica de nivel bajo, `axis()`. Los ejes tienen tres componentes principales: La *línea del eje*, cuyo estilo lo controla el parámetro `lty`, las *marcas de división*, que indican las unidades de división del eje, y las *etiquetas de división*, que indican las unidades de las marcas de división. Estas componentes pueden modificarse con los siguientes parámetros gráficos:

`lab=c(5, 7, 12)`
: Los dos primeros números indican el número de marcas del eje x y del eje y respectivamente, y el tercer número es la longitud de las etiquetas de los ejes medida en caracteres (incluido el punto decimal). Atención: Si elige un número muy pequeño puede que todas las etiquetas de división se redondeen al mismo número.

`las=1`
: Corresponde a la orientación de las etiquetas de los ejes. Un `0` indica paralelo al eje, un `1` indica horizontal, y un `2` indica perpendicular al eje.

`mgp=c(3, 1, 0)`
: Son las posiciones de las componentes de los ejes. La primera es la distancia desde la etiqueta del eje al propio eje, medida en líneas de texto. La segunda es la distancia hasta las etiquetas de división. La tercera y última es la distancia desde el eje a la línea del eje (normalmente cero). Los valores positivos indican que está fuera de la zona de dibujo, y los negativos indican que está dentro.

`tck=0.01`
: Es la longitud de las marcas de división, dada como una fracción de la zona de dibujo. Cuando `tck` es pequeño (menos de 0.5) las marcas de división de ambos ejes serán del mismo tamaño. Un valor de 1 hará que aparezca una rejilla. Si el valor es negativo, las marcas de división se harán por la parte exterior de la zona de dibujo. Utilice `tck=0.01` y `mgp=c(1,-1.5,0)` para obtener marcas de división internas.

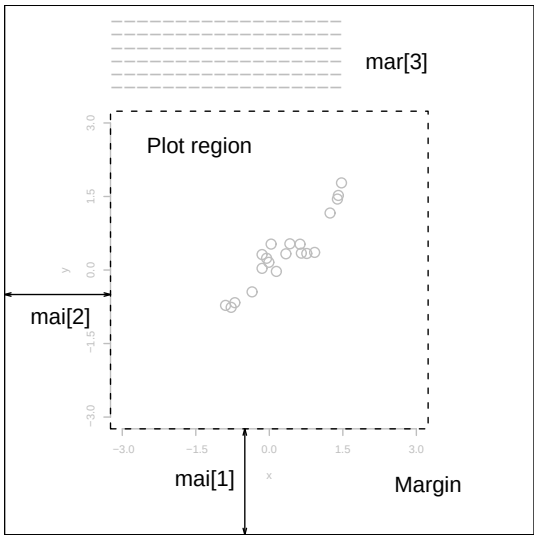
`xaxs="s"`
`yaxs="d"`
: Estilo de eje de los ejes x e y respectivamente. Con los estilos `"s"` (estándar) y `"e"` (extendido) tanto la mayor marca como la menor caen fuera del intervalo de los datos. Los ejes extendidos pueden ampliarse levemente si hay algún punto muy próximo al borde. Este estilo de ejes puede dejar a veces grandes zonas en blanco cerca de los bordes. Con los estilos `"i"` (interno) y `"r"` (predeterminado) las marcas de división siempre caen dentro del rango de los datos, aunque el estilo `"r"` deja un pequeño espacio en los bordes.

  Si selecciona el estilo `"d"` (directo) se procede a *bloquear* los ejes actuales y los utiliza para los siguientes gráficos hasta que el parámetro se cambie a otro de los valores. Este procedimiento es útil para generar series de gráficos con escalas fijas.

12.5.3 Márgenes de las figuras

Un gráfico de R se denomina `figura` y comprende una *zona de dibujo* rodeada de márgenes (que posiblemente contendrán etiquetas de ejes, títulos, etc.) y, normalmente, acotada por los propios ejes.

Una figura típica es



Los parámetros gráficos que controlan la disposición de la figura incluyen:

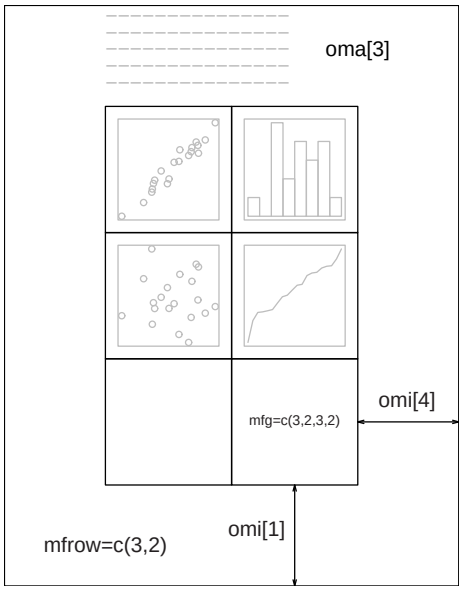
`mai=c(1, 0.5, 0.5, 0)`
: Anchuras de los márgenes inferior, izquierdo, superior y derecho respectivamente, medidos en pulgadas.

`mar=c(4, 2, 2, 1)`
: Similar a `mai`, pero medido en líneas de texto.

Los parámetros `mar` y `mai` están relacionados en el sentido de que un cambio en uno se refleja en el otro. Los valores predeterminados son a menudo demasiado grandes, el margen derecho se necesita raramente, igual que el superior si no se incluye título. Los márgenes inferior e izquierdo sólo necesitan el tamaño preciso para incluir las etiquetas de ejes y de división. Además el valor predeterminado no tiene en cuenta la superficie del dispositivo gráfico. Así, si utiliza el dispositivo `postscript()` con el argumento `height=4` obtendrá un gráfico en que la mitad del mismo son márgenes, salvo que explícitamente cambie `mar` o `mai`. Cuando hay figuras múltiples, como veremos después, los márgenes se reducen a la mitad, aunque suele ser insuficiente cuando varias figuras comparten la misma página.

12.5.4 Figuras múltiples

R permite la creación de una matriz de $n \times m$ figuras en una sola página. Cada figura tiene sus propios márgenes, y la matriz de figuras puede estar opcionalmente rodeada de un *margen exterior* , tal como se muestra en la siguiente figura:



Los parámetros gráficos relacionados con las figuras múltiples son los siguientes:

`mfcol=c(3, 2)`
`mfrow=c(2, 4)`
: Definen el tamaño de la matriz de figuras múltiples. En ambos, el primer valor es el número de filas, el segundo el de columnas. La diferencia entre los dos, es que con el primero, `mfcol`, la matriz se rellena por columnas, en tanto que con el segundo, `mfrow`, lo hace por filas. La distribución en la figura del ejemplo se ha creado con `mfrow=c(3,2)` y se ha representado la página en el momento en que se han realizado los cuatro primeros gráficos.

`mfg=c(2, 2, 3, 2)`
: Definen la posición de la figura actual dentro de la matriz de figuras múltiples. Los dos primeros valores indican la fila y columna de la figura actual, en tanto que los dos últimos son el número de filas y columnas de la matriz de figuras múltiples. Estos parámetros se utilizan para seleccionar cada una de las diferentes figuras de la matriz. Incluso, los dos últimos valores pueden ser distintos de los *verdaderos* valores, para poder obtener figuras de tamaños distintos en la misma página.

`fig=c(4, 9, 1, 4)/10`
: Definen la posición de la figura actual en la página. Los valores son las posiciones de los bordes izquierdo, derecho, inferior y superior respectivamente, medidos como la proporción de página desde la esquina inferior izquierda. El ejemplo correspondería a una figura en la parte inferior derecha de la página. Este parámetro permite colocar una figura en cualquier lugar de la página.

`oma=c(2, 0, 3, 0)`
`omi=c(0, 0, 0.8, 0)`
: Definen el tamaño de los márgenes exteriores. De modo similar a `mar` y `mai`, el primero está expresado en líneas de texto y el segundo en pulgadas, corresponden a los márgenes inferior, izquierdo, superior y derecho respectivamente.

Los márgenes exteriores son particularmente útiles para ajustar convenientemente los títulos, etc. Puede añadir texto en estos márgenes con la función `mtext()` sin más que

utilizar el argumento `outer=T`. R no crea márgenes exteriores predeterminadamente, pero pueden crearse utilizando las funciones `oma` y `omi`.

Es posible crear disposiciones de figuras múltiples más complejas utilizando las funciones `split.screen()` y `layout()`.

12.6 Dispositivos gráficos

R puede generar gráficos (con niveles de calidad diversos) en casi todos los tipos de pantalla o dispositivos de impresión. Ahora bien, en primer lugar es necesario informarle del tipo de dispositivo de que se trata. Esta acción se realiza iniciando un *controlador de dispositivo*. El propósito de un controlador de dispositivo es convertir las instrucciones gráficas de R (por ejemplo, 'dibuja una línea') en una forma que el dispositivo concreto pueda comprender.

Los controladores de dispositivo se inician llamando a una función de controlador de dispositivo. Existe una función para cada controlador de dispositivo y la lista completa puede conseguirse con `help(Devices)`. Por ejemplo, la orden

```
> postscript()
```

dirige cualquier salida gráfica a la impresora en formato PostScript. Algunos controladores de dispositivo habituales son:

`X11()` Para uso con los sistemas de ventanas X11 y Microsoft Windows.

`postscript()`
Para imprimir en impresoras PostScript o crear archivos con este formato.

`pictex()` Crea un archivo para LaTeX.

Al terminar de utilizar un dispositivo, asegúrese de finalizar el mismo utilizando la orden

```
> dev.off()
```

Esta orden asegura que el dispositivo finaliza correctamente; por ejemplo en el caso de una impresora asegura que cada página sea completada y enviada a la impresora.

12.6.1 Inclusión de gráficos PostScript en documentos

Si utiliza el argumento `file` en la función `postscript()`, almacenará los gráficos, en formato PostScript, en el archivo que desee. Tenga en cuenta que si el archivo ya existe, será previamente borrado. Ello ocurre aunque el archivo se haya creado previamente en la misma sesión de R. El gráfico tendrá orientación apaisada salvo que especifique el argumento `horizontal=FALSE`. El tamaño del gráfico se controla con los argumentos `width` (anchura) y `height` (altura) que se utilizan como factores de escala para ajustarlo a dichas dimensiones. Por ejemplo, la orden

```
> postscript("grafico.ps", horizontal=FALSE, height=5, pointsize=10)
```

producirá un archivo que contiene el código PostScript para una figura de cinco pulgadas de alto, que podrá ser incluido en un documento. A menudo dicha inclusión requiere que el archivo se almacene en formato EPS[1]. El archivo que produce R es de este tipo, pero sólo lo indicará expresamente si se utiliza el argumento `onefile=FALSE`. Esta notación es

[1] Acrónimo en inglés de Encapsulated PostScript

consecuencia de la compatibilidad con S e indica que la salida está constituida por una sola página. Por tanto para crear un gráfico que se pueda incluir sin problema en cualquier procesador, deberá utilizar una orden análoga a la siguiente:

```
> postscript("grafico.eps", horizontal=FALSE, onefile=FALSE,
             height=8, width=6, pointsize=10)
```

12.6.2 Dispositivos gráficos múltiples

La utilización avanzada de R implica a menudo tener varios dispositivos gráficos en uso simultáneamente. Naturalmente, sólo un dispositivo gráfico acepta las órdenes gráficas en cada momento, y se denomina *dispositivo actual*. Cuando se abren varios dispositivos, forman una sucesión numerada cuyos nombres determinan el tipo de dispositivo en cada posición.

Las principales órdenes utilizadas para trabajar con varios dispositivos y sus significados son las siguientes:

`X11()` Abre una ventana en Unix y Microsoft Windows

`windows()`
: Abre una ventana en Microsoft Windows

`postscript()`
`pictex()`
`...`
: Cada llamada a una función de controlador de dispositivo abre un nuevo dispositivo gráfico y, por tanto, añade un elemento a la lista de dispositivos, al tiempo que este dispositivo pasa a ser el dispositivo actual al que se enviarán los resultados gráficos. (En algunas plataformas es posible que existan otros dispositivos disponibles.)

`dev.list()`
: Devuelve el número y nombre de todos los dispositivos activos. El dispositivo de la posición 1 es siempre el *dispositivo nulo* que no acepta ninguna orden gráfica.

`dev.next()`
`dev.prev()`
: Devuelve el número y nombre del dispositivo gráfico siguiente o anterior, respectivamente, al dispositivo actual.

`dev.set(which=`k`)`
: Puede usarse para hacer que el dispositivo que ocupa la posición k en la lista de dispositivos sea el actual. Devuelve el número y nombre del dispositivo.

`dev.off(`k`)`
: Cierra el dispositivo gráfico que ocupa la posición k de la lista de dispositivos. Para algunos dispositivos, como los `postscript`, finalizará el gráfico, bien imprimiéndolo inmediatamente, bien completando la escritura en el archivo, dependiendo de cómo se ha iniciado el dispositivo.

`dev.copy(device, ..., which=`*k*`)`
`dev.print(device, ..., which=`*k*`)`
: realiza una copia del dispositivo *k*. Aquí, `device`, es una función de dispositivo, como `postscript`, con argumentos adicionales si es necesario, especificados por `...`. La función `dev.print` es similar, pero el dispositivo copiado se cierra inmediatamente, lo que finaliza las acciones pendientes que se realizan inmediatamente.

`graphics.off()`
: Cierra todos los dispositivos gráficos de la lista, excepto el dispositivo nulo.

12.7 Gráficos dinámicos

R no dispone (en este momento) de ninguna función de gráficos dinámicos, por ejemplo para rotar una nube de puntos o activar y desactivar puntos interactivamente. Sin embargo existen muchas de estas posibilidades disponibles en el sistema `XGobi` de Swayne, Cook y Buja, disponible en

`http://www.research.att.com/areas/stat/xgobi/`

a las que puede acceder desde R a través de la biblioteca **xgobi**.

`XGobi` está disponible actualmente para X Windows, tanto en Unix como en Microsoft Windows, y existen conexiones con R disponibles en ambos sistemas.

Apendice A Primera sesión con R

La siguiente sesión pretende presentar algunos aspectos del entorno R, utilizándolos. Muchos de estos aspectos le serán desconocidos e incluso enigmáticos al principio, pero esta sensación desaparecerá rápidamente. La sesión está escrita para el sistema UNIX, por lo que es posible que los usuarios de Microsoft Windows deban realizar pequeños cambios.

Conéctese e inicie el sistema de ventanas. Es necesario que disponga del archivo '`morley.data`' en su directorio de trabajo, por lo que, si es necesario, deberá copiarlo. Si no sabe hacerlo, consulte con un experto local. Si ya lo ha copiado, continúe.

`$ R` Ejecute R.

Comienza R y aparece el mensaje inicial.

(Dentro de R no mostraremos el símbolo de 'preparado' en la parte izquierda para evitar confusiones.)

`help.start()`

Muestra la ayuda interactiva, que está escrita en formato HTML, utilizando el lector WEB disponible en el ordenador. Si utiliza esta ayuda comprenderá mejor el funcionamiento de R.

Minimice la ventana de ayuda y continúe la sesión.

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

Genera dos vectores que contienen cada uno 50 valores pseudoaleatorios obtenidos de una distribución normal (0,1) y los almacena en x e y.

`plot(x, y)`

Aparecerá una ventana gráfica automáticamente. En ella se representan los puntos antes generados, tomando cada componente de x y de y como las coordenadas de un punto del plano.

`ls()` Presenta los nombres de los objetos existentes en ese momento en el espacio de trabajo de R.

`rm(x, y)` Elimina los objetos x e y.

`x <- 1:20` Almacena en x el vector $(1, 2, \ldots, 20)$.

`w <- 1 + sqrt(x)/2`

A partir del vector x, crea un vector ponderado de desviaciones típicas, y lo almacena en w.

```
hoja.de.datos <- data.frame(x=x, y= x + rnorm(x)*w)
hoja.de.datos
```

Crea una *hoja de datos* de dos columnas, llamadas x e y, y la almacena en *hoja.de.datos*. A continuación la presenta en pantalla.

```
regr <- lm(y ~ x, data=hoja.de.datos)
summary(regr)
```

Realiza el ajuste de un modelo de regresión lineal de y sobre x, lo almacena en *regr*, y presenta en pantalla un resumen del análisis.

`regr.pon <- lm(y ~ x, data=hoja.de.datos, weight=1/w^2)`
`summary(regr.pon)`
: Puesto que se conocen las desviaciones típicas, puede realizarse una regresión ponderada.

`attach(hoja.de.datos)`
: Conecta la hoja de datos, de tal modo que sus columnas aparecen como variables.

`regr.loc <- lowess(x, y)`
: Realiza una regresión local no paramétrica.

`plot(x, y)`
: Representa el gráfico bidimensional estándar.

`lines(x, regr.loc$y)`
: Le añade la regresión local.

`abline(0, 1, lty=3)`
: Le añade la verdadera recta de regresión (punto de corte 0, pendiente 1).

`abline(coef(regr))`
: Le añade la recta de regresión no ponderada.

`abline(coef(regr.pon), col = "red")`
: Le añade la recta de regresión ponderada.

`detach()` Desconecta la hoja de datos, eliminándola de la trayectoria de búsqueda.

```
plot(fitted(regr), resid(regr),
     xlab="Predichos",
     ylab="Residuos",
     main="Residuos / Predichos")
```
: Un gráfico diagnóstico de regresión para investigar la posible heteroscedasticidad.

`qqnorm(resid(regr), main="Residuos por rangos")`
: Gráfico en papel probabilístico normal para comprobar asimetría, aplastamiento y datos anómalos. (No es muy útil en este caso)

`rm(x,w,hoja.de.datos,regr,regr.pon,regr.loc)`
: Elimina los objetos creados.

`file.show("morley.data")`
: Presenta el contenido del archivo.

`mm <- read.table("morley.data")`
`mm`
: Lee los datos de Michaelson y Morley y los almacena en la hoja de datos `mm`, y la muestra en pantalla a continuación. Hay cinco experimentos (columna `Expt`) y cada uno contiene 20 series (columna `Run`) y la columna `Speed` contiene la velocidad de la luz medida en cada caso, codificada apropiadamente.

`mm$Expt <- factor(mm$Expt)`
`mm$Run <- factor(mm$Run)`
: Transforma `Expt` y `Run` en factores.

`attach(mm)`
Conecta la hoja de datos en la posición predeterminada: la 2.

`plot(Expt, Speed, main="Velocidad de la luz", xlab="No. Experimento")`
Compara los cinco experimentos mediante diagramas de cajas.

`fm <- aov(Speed ~ Run + Expt, data=mm)`
`summary(fm)`
Analiza los datos como un diseño en bloques aleatorizados, tomando las 'series' y los 'experimentos' como factores.

`fm0 <- update(fm, . ~ . - Run)`
`anova(fm0,fm)`
Ajusta el submodelo eliminando las 'series', y lo compara utilizando un análisis de la varianza.

`detach()`
`rm(fm, fm0)`
Desconecta la hoja de datos y elimina los objetos creados, antes de seguir adelante.

Consideraremos ahora nuevas posibilidades gráficas.

`x <- seq(-pi, pi, len=50)`
`y <- x` x es un vector con 50 valores equiespaciados en el intervalo $-\pi \leq x \leq \pi$. El vector y es idéntico a x.

`f <- outer(x, y, function(x, y) cos(y)/(1 + x^2))`
f es una matriz cuadrada, cuyas filas y columnas están indexadas por x e y respectivamente, formada por los valores de la función $\cos(y)/(1 + x^2)$.

`oldpar <- par(no.readonly = TRUE)`
`par(pty="s")`
Almacena los parámetros gráficos y modifica el parámetro `pty` (zona de dibujo) para que valga "s" (cuadrado).

`contour(x, y, f)`
`contour(x, y, f, nlevels=15, add=TRUE)`
Dibuja un mapa de curvas de nivel de f; y después le añade más líneas para obtener más detalle.

`fa <- (f-t(f))/2`
`fa` es la "parte asimétrica" de f. (`t(f)` es la traspuesta de `f`).

`contour(x, y, fa, nint=15)`
Dibuja un mapa de curvas de nivel,...

`par(oldpar)`
... y recupera los parámetros gráficos originales.

`image(x, y, f)`
`image(x, y, fa)`
Dibuja dos gráficos de densidad

`objects(); rm(x, y, f, fa)`
: Presenta los objetos existentes y elimina los objetos creados, antes de seguir adelante.

Con R también puede utilizar números complejos. `1i` se utiliza para representar la unidad imaginaria, i.

```
th <- seq(-pi, pi, len=100)
z <- exp(1i*th)
```

```
par(pty="s")
plot(z, type="l")
```
: La representación gráfica de un argumento complejo consiste en dibujar la parte imaginaria frente a la real. En este caso se obtiene un círculo.

```
w <- rnorm(100) + rnorm(100)*1i
```
: Suponga que desea generar puntos pseudoaleatorios dentro del círculo unidad. Un primer intento consiste en generar puntos complejos cuyas partes real e imaginaria, respectivamente, procedan de una normal (0,1) . . .

```
w <- ifelse(Mod(w) > 1, 1/w, w)
```
: . . . y sustituir los que caen fuera del círculo por sus inversos.

```
plot(w, xlim=c(-1,1), ylim=c(-1,1), pch="+",xlab="x", ylab="y")
lines(z)
```
: Todos los puntos están dentro del círculo unidad, pero la distribución no es uniforme.

```
w <- sqrt(runif(100))*exp(2*pi*runif(100)*1i)
plot(w, xlim=c(-1,1), ylim=c(-1,1), pch="+", xlab="x", ylab="y")
lines(z)
```
: Este segundo método utiliza la distribución uniforme. En este caso, los puntos presentan una apariencia más uniformemente espaciada sobre el círculo.

```
rm(th, w, z)
```
: De nuevo elimina los objetos creados, antes de seguir adelante.

`q()`
: Termina el programa R. Se le preguntará si desea salvar el espacio de trabajo. Puesto que esta sesión ha sido solamente de presentación, probablemente debería contestar que no.

Apendice B Ejecución de R

B.1 Ejecución de R en UNIX

La orden '`R`' se utiliza para ejecutar R con diferentes opciones, de la forma

`R` [*opciones*] [`<`*archivoentrada*] [`>`*archivosalida*],

o, a través del interfaz `R CMD`, para acceder a varias herramientas de R (por ejemplo, para procesar archivos de formato de documentación de R o manipular bibliotecas) que no están diseñadas para ser usadas "directamente".

Muchas opciones controlan lo que ocurre al comenzar y al terminar una sesión de R. El mecanismo de inicio (Utilice '`help(Startup)`' para información actualizada) es el siguiente:

- Salvo que se especifique la opción '`--no-environ`', R busca el archivo '`.Renviron`' en el directorio actual; si no lo encuentra, busca el archivo indicado en la variable de entorno `R_ENVIRON`, y si esta variable no existe, busca el archivo '`.Renviron`' en el directorio inicial del usuario. Si encuentra alguno de estos archivos, se ejecuta (como si se lo diese como argumento a la función `source` dentro de R) para dar valores a las variables de entorno. Las variables se exportan automáticamente, supuesto que se les dé valor en líneas de la forma '*nombre*=*valor*'. Algunas de estas variables son:

  `R_PAPERSIZE` (tamaño predeterminado del papel),

  `R_PRIMTCMD` (orden predeterminada de impresión),

  `R_LIBS` (lista de bibliotecas que deben buscarse), y

  `R_VSIZE` y `R_NSIZE` que describiremos después.
- A continuación, R busca el perfil de inicio global (para todos los usuarios) salvo que se haya especificado la opción '`--no-site-file`'. El nombre de este archivo se toma del valor de la variable de entorno `R_PROFILE`. Si esta variable no existe, el valor predeterminado es '`$R_HOME/etc/Rprofile`'.
- Una vez acabado el paso anterior, si no se ha especificado la opción '`--no-init-file`', R busca un archivo llamado '`.Rprofile`' en el directorio actual, y si no lo encuentra allí, lo busca en el directorio inicial del usuario. Si encuentra el archivo, lo ejecuta.
- Si existe un archivo llamado '`.RData`', que contenga una imagen de espacio de trabajo, lo utiliza para restaurar el espacio de trabajo, salvo que se haya especificado la opción '`--no-restore`'.
- Por último, si existe una función llamada `.First`, la ejecuta. (Del mismo modo, si existe una función llamada `.Last` será ejecutada al finalizar la sesión de R) La función puede definirse en los perfiles de inicio o residir en el archivo de imagen '`.RData`'.

Además de todo lo indicado, existen opciones para controlar la memoria disponible en una sesión de R (Utilice '`help(Memory)`' para disponer de información actualizada). R utiliza un modelo de memoria *estático*. Esto es, en el momento de iniciarse, el sistema operativo le reserva una cantidad fija de memoria, que no puede alterarse durante la sesión. Por tanto, pudiera ocurrir que no exista memoria suficiente en algún momento para realizar una acción, por ejemplo, para leer un archivo de datos muy grande. Las opciones '`--nsize`' y '`--vsize`' (o las variables de entorno `R_NSIZE` y `R_VSIZE`) controlan la memoria disponible para objetos de tamaños fijo y variable, respectivamente.

Las opciones que pueden darse al ejecutar R son las siguientes:

`--help`
`-h` Muestra un pequeño mensaje de ayuda y termina correctamente.

`--version`
Muestra la información de la versión y termina correctamente.

`RHOME` Muestra la trayectoria al "directorio inicial" de R y termina correctamente. Salvo la página de ayuda en UNIX y el archivo de ejecución, la instalación de R pone todos los archivos (ejecutables, bibliotecas, etc.) en este directorio.

`--save`
`--no-save`
Indica si debe salvar la imagen del entorno al terminar la sesión. En modo interactivo, si no ha especificado ninguna, le preguntará. En modo no interactivo es obligatorio usar una.

`--no-environ`
No lee ningún archivo para dar valor a las variables de entorno.

`--no-site-file`
No lee el perfil de inicio global al iniciar el programa.

`--no-init-file`
No lee el perfil de inicio de usuario al iniciar el programa.

`--restore`
`--no-restore`
Indica si la imagen salvada (archivo '`.Rdata`' en el directorio en que R se haya iniciado) debe ser recuperada o no. El valor predeterminado es recuperarla.

`--vanilla`
Combina las opciones '`--no-save`', '`--no-environ`' '`--no-site-file`', '`--no-init-file`', y '`--no-restore`'.

`--no-readline`
Desactiva la edición de órdenes a través de **readline**. Esta opción suele utilizarse cuando se ejecuta R desde Emacs utilizando la biblioteca ESS ("Emacs Speaks Statistics"). Véase Apendice C [El editor de ordenes], página 93, para ampliar esta información.

`--vsize=`*N*
Indica la cantidad de memoria utilizada para objetos de tamaño variable. *N* debe ser un entero, en cuyo caso la unidad de medida es el octeto o byte, o un entero seguido de una letra que indica la unidad de medida en octetos: '`M`', '`K`', o '`k`', que corresponden respectivamente a 'Mega' (2^20), 'Kilo' (2^10) (de ordenador), o 'kilo'decimal (1000).

`--nsize=`*N*
Indica la cantidad de memoria utilizada para objetos de tamaño fijo. Las consideraciones realizadas en el apartado anterior son válidas para *N*.

`--quiet`
`--silent`
`-q` No muestra ni el mensaje de copyright ni los mensajes de inicio.

`--slave` Ejecuta R con el mínimo de salidas posible. Esta opción se utiliza con programas que se sirven de R para realizar cálculos para ellos mismos.

`--verbose`
Muestra el máximo de salidas posible. Además modifica la opción `verbose` a `TRUE`. R utiliza esta opción para controlar si debe mostrar mensajes de diagnóstico.

`--debugger=`*depurador*
`-d` *depurador*
Ejecuta R desde el programa de depuración (debugger) *depurador*. En este caso, si existen otras opciones, se descartan. Cualquier otra opción debe darse al iniciar R desde el programa de depuración.

`--gui=`*tipo*
Utiliza *tipo* como interfaz gráfico (advierta que esto también incluye los gráficos interactivos). Los valores posibles de *tipo* son `X11` (predeterminado) y `GNOME`, supuesto que GNOME esté disponible.

La entrada y la salida pueden redirigirse de la manera habitual, utilizando '`<`' and '`>`'.

`R CMD` permite utilizar varias herramientas que son útiles en conjunción con R, pero que no están diseñadas para usarlas "directamente". La forma general es

`R CMD` *orden argumentos*

donde *orden* es el nombre de la herramienta y *argumentos* son los argumentos que se pasan a la misma.

Las herramientas disponibles son:

`BATCH` Ejecuta R en modo no interactivo.

`COMPILE` Compila archivos para uso con R.

`SHLIB` Construye bibliotecas compartidas del sistema operativo para carga dinámica.

`INSTALL` Instala bibliotecas añadidas.

`REMOVE` Elimina bibliotecas añadidas.

`build` Construye bibliotecas añadidas.

`check` Comprueba bibliotecas añadidas.

`Rdconv` Convierte desde formato Rd a otros formatos, incluyendo HTML, Nroff, LaTeX, texto ASCII sin formato, y formato de documentación S.

`Rd2dvi` Convierte desde formato Rd a DVI/PDF.

`Rd2txt` Convierte desde formato Rd a texto con formato.

`Rdindex` Extrae información de índices de archivos Rd.

`Sd2Rd` Convierte desde formato de documentación S a formato Rd.

Las cinco primeras herramientas (`BATCH`, `COMPILE`, `SHLIB`, `INSTALL`, y `REMOVE`) pueden ejecutarse "directamente" sin la opción `CMD`, esto es, en la forma `R` *orden argumentos*.

Utilice la orden

`R CMD` *herramienta* `--help`

para obtener información sobre cada una de las herramientas descritas.

B.2 Ejecución de R en Microsoft Windows

El procedimiento de inicio en Microsoft Windows es muy similar al de UNIX, pero no necesariamente idéntico. Existen dos versiones de R en este sistema operativo: Una basada en ventanas MDI, `RGui.exe`, y otra en ventanas SDI, `Rterm.exe`, pensada especialmente para uso no interactivo.

Muchas opciones controlan lo que ocurre al comenzar y al terminar una sesión de R. El mecanismo de inicio (Utilice '`help(Startup)`' para información actualizada) es el siguiente. Las referencias al "directorio inicial" deben ser aclaradas, ya que éste no siempre está definido en Microsoft Windows. Si se ha definido la variable de entorno `R_USER`, entonces su valor es el directorio inicial. En caso contrario, lo define la variable de entorno `HOME`; y si tampoco está definida, lo harán las variables `HOMEDRIVE` y `HOMEPATH` (que normalmente están definidas en Windows NT). Si ninguna de las variables mencionadas existe, el directorio inicial será el directorio de trabajo.

- Salvo que se especifique la opción '`--no-environ`', R busca el archivo '`.Renviron`' en el directorio actual; si no lo encuentra, busca el archivo '`.Renviron`' en el directorio inicial del usuario. Si encuentra alguno de estos archivos, se ejecuta (como si se lo diese como argumento a la función `source` dentro de R) para dar valores a las variables de entorno. Se asigna valor a las variables en líneas de la forma '*nombre*=*valor*'. Algunas de estas variables son

  `R_PAPERSIZE` (tamaño predeterminado del papel),

  `R_PRIMTCMD` (orden predeterminada de impresión),

  `R_LIBS` (lista de bibliotecas que deben buscarse), y

  `R_VSIZE` y `R_NSIZE` que describiremos después.

  Las variables de entorno también pueden indicarse como parejas de la forma '*nombre*=*valor*' en la propia línea de órdenes.
- A continuación, R busca el archivo de perfil de inicio en el ordenador, salvo que se indique la opción '`--no-site-file`'. El nombre del archivo se almacena en la variable de entorno `R_PROFILE` y si no existe se toma '`$R_HOME/etc/Rprofile`'.
- Si no se ha indicado '`--no-init-file`', R busca el archivo '`.Rprofile`' en el directorio actual o en el de inicio del usuario y ejecuta las órdenes en él contenidas.
- A continuación carga el archivo de imagen '`.RData`', si existe, salvo que se indique la opción '`--no-restore`'.
- Si existe la función `.First`, la ejecuta. Esta función, así como la función `.Last` que se ejecuta la finalizar la sesión de R, pueden definirse en los perfiles de inicio adecuados, o residir en el archivo '`.RData`'.

Además de todo lo indicado, existen opciones para controlar la memoria disponible en una sesión de R (Utilice '`help(Memory)`' para disponer de información actualizada). R utiliza un modelo de memoria *estático*. Esto es, en el momento de iniciarse, el sistema operativo le reserva una cantidad fija de memoria, que no puede alterarse durante la sesión. Por tanto, pudiera ocurrir que no exista memoria suficiente en algún momento para realizar una acción, por ejemplo, para leer un archivo de datos muy grande. Las opciones '`--nsize`' y '`--vsize`' (o las variables de entorno `R_NSIZE` y `R_VSIZE`) controlan la memoria disponible para objetos de tamaños fijo y variable, respectivamente.

Las opciones que pueden darse al ejecutar R en Microsoft Windows son las siguientes:

`--version`
: Muestra la información de la versión y termina correctamente.

`--mdi`
`--sdi`
`--no-mdi` Inidica si `Rgui` se comportará como un programa MDI (predeterminado), donde cada nueva ventana está contenida dentro de la ventana principal, o como un programa SDI, donde cada ventana aparece de modo independiente en el escritorio.

`--save`
`--no-save`
: Indica si debe salvar la imagen del entorno al terminar la sesión. En modo interactivo, si no ha especificado ninguna, le preguntará. En modo no interactivo es obligatorio usar una.

`--restore`
`--no-restore`
: Indica si la imagen salvada (archivo '`.Rdata`' en el directorio en que R se haya iniciado) debe ser recuperada o no. El valor predeterminado es recuperarla.

`--no-site-file`
: No lee el perfil de inicio global al iniciar el programa.

`--no-init-file`
: No lee el archivo '`.Rprofile`' del directorio del usuario (perfil de inicio de usuario) al iniciar el programa.

`--no-environ`
: No lee el archivo '`.Renviron`'.

`--vanilla`
: Combina las opciones `--no-save`, `--no-restore`, `--no-site-file`, `--no-init-file` y `--no-environ`.

`-q`
`--quiet`
`--silent` No muestra el mensaje de inicio.

`--slave` Ejecuta R con el mínimo de salidas posible.

`--verbose`
: Muestra el máximo de salidas posible.

`--ess` Prepara `Rterm` para uso en modo `R-inferior` en ESS.

Apendice C El editor de órdenes

C.1 Preliminares

Si la biblioteca de GNU, '`readline`', está disponible cuando se compila R en UNIX, se puede utilizar un editor interno de líneas de órdenes que permite recuperar, editar y volver a ejecutar las órdenes utilizadas previamente.

Este editor puede desactivarse (lo que permite utilizar ESS[1]) dando la opción de inicio '--no-readline'.

La versión de Microsoft Windows dispone de un editor más sencillo. Vea la opción '`Console`' en el menú '`Help`'.

Cuando use R con las posibilidades de **readline**, estarán disponibles las opciones que posteriormente se indican.

Tenga en cuenta las siguientes convenciones tipográficas usuales:

Muchas de las órdenes utilizan caracteres caracteres Control y Meta. Los caracteres Control, tales como *`Control-m`*, se obtienen pulsando la tecla ⟨CTRL⟩ y, sin soltarla, pulsando la tecla ⟨m⟩, y en la tabla los escribiremos como *`C-m`*. Los caracteres Meta, tales como *`Meta-b`*, se obtienen pulsando la tecla ⟨META⟩ y, después de soltarla, pulsando la tecla ⟨b⟩, y en la tabla los escribiremos como *`M-b`*. Si su teclado no tiene la tecla ⟨META⟩, puede obtenerlos mediante una secuencia de dos caracteres que comienza con *`ESC`*. Esto es, para obtener *`M-b`*, deberá escribir ⟨ESC⟩⟨b⟩. Estas secuencias *`ESC`* también puede utilizarlas aunque su teclado sí disponga de la tecla ⟨META⟩. Debe tener en cuenta que en los caracteres Meta se distingue entre mayúsculas y minúsculas, por lo que puede que sea distinto el resultado si se pulsa *`M-b`* o *`M-B`*.

C.2 Edición de acciones

R conserva un historial de las órdenes que se teclean, incluyendo las líneas erróneas, lo que permite recuperar las líneas del historial, modificarlas si es necesario, y volver a ejecutarlas como nuevas órdenes. Si el estilo de edición de órdenes es *emacs* cualquier carácter que teclee se inserta en la orden que se está editando, desplazando los caracteres que estén a la derecha del cursor. En el estilo *vi* el modo de inserción de caracteres se inicia con *`M-i`* o *`M-a`*, y se finaliza con ⟨ESC⟩.

Cuando pulse la tecla ⟨RET⟩, la orden completa será ejecutada.

Otras acciones posibles de edición se resumen en la tabla siguiente.

C.3 Resumen del editor de líneas de órdenes

[1] Corresponde al acrónimo del editor de textos, 'Emacs Speaks Statistics'; vea la dirección http://ess.stat.wisc.edu/

Recuperación de órdenes y movimiento vertical

`C-p` Recupera la orden anterior (retrocede en el historial).

`C-n` Recupera la orden posterior (avanza en el historial).

`C-r` *text* Recupera la última orden que contenga la cadena *texto*.

En muchos terminales, es posible utilizar las teclas de flecha hacia arriba y flecha hacia abajo, respectivamente, en vez de `C-p` y `C-n`.

Movimiento horizontal del cursor

`C-a` Va al principio de la línea.

`C-e` Va al final de la línea.

`M-b` Retrocede una palabra.

`M-f` Avanza una palabra.

`C-b` Retrocede un carácter.

`C-f` Avanza un carácter.

En muchos terminales, es posible utilizar las teclas de flecha hacia la izquierda y flecha hacia la derecha, respectivamente, en vez de `C-b` y `C-f`.

Edición

text Inserta *texto* en el cursor.

`C-f` *text* Añade *texto* tras el cursor.

⟨DEL⟩ Borra el carácter a la izquierda del cursor.

`C-d` Borra el carácter bajo el cursor.

`M-d` Borra el resto de la palabra bajo el cursor, y la guarda.

`C-k` Borra el resto de la línea desde el cursor, y lo guarda.

`C-y` Inserta el último texto guardado.

`C-t` Intercambia el carácter bajo el cursor con el siguiente.

`M-l` Cambia el resto de la palabra a minúsculas.

`M-c` Cambia el resto de la palabra a mayúsculas.

⟨RET⟩ Vuelve a ejecutar la línea.

Al pulsar ⟨RET⟩, se termina la edición de la línea.

Apendice D Índice de funciones y variables

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Apendice E Índice de conceptos

R

S

T

V

W

Apendice F Referencias

D. M. Bates and D. G. Watts (1988), *Nonlinear Regression Analysis and Its Applications.* John Wiley & Sons, New York.

Richard A. Becker, John M. Chambers and Allan R. Wilks (1988), *The New S Language.* Chapman & Hall, New York. This book is often called the "*Blue Book*".

John M. Chambers and Trevor J. Hastie eds. (1992), *Statistical Models in S.* Chapman & Hall, New York. This is also called the "*White Book*".

Annette J. Dobson (1990), *An Introduction to Generalized Linear Models*, Chapman and Hall, London.

Peter McCullagh and John A. Nelder (1989), *Generalized Linear Models.* Second edition, Chapman and Hall, London.

John A. Rice (1995), *Mathematical Statistics and Data Analysis.* Second edition. Duxbury Press, Belmont, CA.

S. D. Silvey (1970), *Statistical Inference.* Penguin, London.