

# AprendeR: Parte I

*The UIB-AprendeR team*

2019-09-09

## Presentación

Edición preliminar en línea de la 1a parte del libro *AprendeR*, producido por un grupo de profesores del Departamento de Ciencias Matemáticas e Informática de la UIB.

El libro está escrito en *R Markdown*, usando *RStudio* como editor de texto y el paquete **bookdown** para convertir los ficheros *markdown* en un libro.

Este trabajo se publica bajo licencia [Atribución-No Comercial-SinDerivadas 4.0](#)

# Lección 1 Logística de R

R es un entorno de programación para el análisis estadístico y gráfico de datos muy popular, cada día más utilizado en empresas y universidades. Su uso tiene muchas ventajas. Para empezar, es *software* libre. La elección de *software* libre es, en general, acertada por varios motivos. Por un lado, transmite valores socialmente positivos, como por ejemplo la libertad individual, el conocimiento compartido, la solidaridad y la cooperación. Por otro, nos aproxima al método científico, porque permite el examen y mejora del código desarrollado por otros usuarios y la reproducibilidad de los resultados obtenidos. Finalmente, pero no menos importante desde un punto de vista práctico, podemos adquirir de manera legal y gratuita copias del programa, sin necesidad de licencias personales o académicas.

Aparte de su faceta de *software* libre, R tiene algunas ventajas específicas: por ejemplo, su sintaxis básica es sencilla e intuitiva, con la que es muy fácil familiarizarse, lo que se traduce en un aprendizaje rápido y cómodo. Además, tiene una enorme comunidad de usuarios, estructurada alrededor de la *Comprehensive R Archive Network*, o *CRAN*, que desarrolla cada día nuevos paquetes que extienden sus funcionalidades y cubren casi todas las necesidades computacionales y estadísticas de un científico o ingeniero. Para que os hagáis una idea, en el momento de revisar estas notas (septiembre de 2019) el número de paquetes en el repositorio de la CRAN acaba de superar los 15000.

## 1.1 Cómo instalar R y *RStudio*

Instalar R es muy sencillo; de hecho, seguramente ya lo tenéis instalado en vuestro ordenador, pero es conveniente que dispongáis de su versión más reciente y que regularmente lo pongáis al día. Los pasos a realizar en Windows o Mac OS X para instalar su última versión son los siguientes:

- Si sois usuarios de Windows, acceded a la página web de la [CRAN](#) y pulsad sobre el enlace *Download R for Windows*. A continuación, entrad en el enlace *base*, descargad R y seguid las instrucciones de instalación del documento *Installation and other*

*instructions* que encontraréis en esa misma página.

- Si sois usuarios de Mac OS X, acceded a la página web de la [CRAN](#) y pulsad sobre el enlace *Download R for Mac OS X*. A continuación, descargad el fichero `.pkg` correspondiente y, una vez descargado, abridlo y seguid las instrucciones del Asistente de Instalación.
- Si trabajáis con Ubuntu o Debian, para instalar la última versión de R basta que ejecutéis en una terminal, estando conectados a Internet, la siguiente instrucción:

```
sudo aptitude install r-base
```

Cuando instaláis R para Windows o Mac OS X, con él también se os instala una interfaz gráfica que se abrirá al abrir la aplicación y en la que podréis trabajar. La instalación para Linux no lleva una interfaz por defecto, así que sus usuarios tienen que trabajar con R en la terminal (ejecutando R para iniciar una sesión) o instalar aparte una interfaz.

Independientemente de todas estas posibilidades, en este curso usaremos *RStudio* como interfaz gráfica de usuario de R para todos los sistemas operativos.

Propiamente hablando, *RStudio* es mucho más que una interfaz de R: se trata de todo un entorno integrado para utilizar y programar con R, que dispone de un conjunto de herramientas que facilitan el trabajo con este lenguaje. Para instalarlo, se ha de descargar del url <http://www.rstudio.com/products/rstudio/download/> la versión correspondiente al sistema operativo en el que se trabaja; en cada caso, escoged la versión gratuita de *RStudio Desktop*. Una vez descargado, si usáis Windows o Mac OS X ya lo podéis abrir directamente. En el caso de Linux, hay que ejecutar en una terminal la siguiente instrucción para completar su instalación:

```
sudo dpkg -i rstudio-<version>-i386.deb
```

donde `version` refiere a la versión concreta que hayáis descargado. Conviene recordar que *RStudio* no es R, ni tan sólo lo contiene: hay que instalar ambos programas. De hecho, las instalaciones de R y *RStudio* son independientes una de la otra, de manera que cuando se pone al día uno de estos programas, no se modifica el otro.

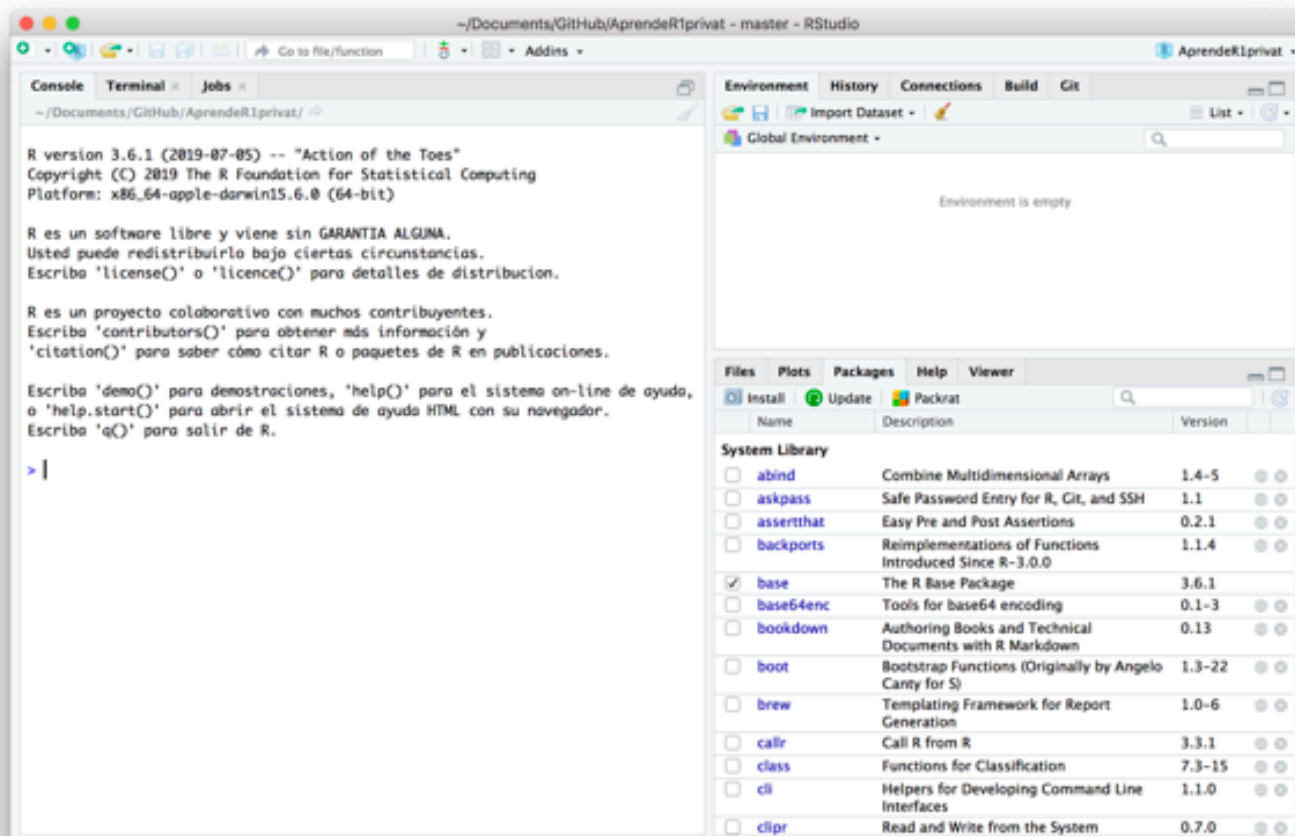


Figura 1.1: Ventana de RStudio para Mac OS X.

Cuando se abre *RStudio*, aparece una ventana similar a la que muestra la Figura 1.1: su apariencia exacta dependerá del sistema operativo, de la versión de *RStudio* e incluso de los paquetes que estemos usando. De momento, nos concentraremos en la ventana de la izquierda, llamada la **consola** de R (la pestaña *Console*). Observaréis que en el momento de abrir la aplicación, dicha ventana contiene una serie de información (versión, créditos etc.) y al final una línea en blanco encabezada por el símbolo `>`. Este símbolo es la **marca de inicio** e indica que R espera que escribáis alguna instrucción y la ejecutéis.

Durante la mayor parte de este curso, usaremos *RStudio* de manera interactiva:

1. Escribiremos una instrucción en la consola, a la derecha de la marca de inicio de su última línea.
2. La ejecutaremos pulsando la tecla *Entrar* ( $\leftarrow$ ).
3. R la evaluará y, si corresponde, escribirá el resultado en la línea siguiente de la consola (como veremos, no todas las instrucciones hacen que R escriba algo).
4. R abrirá una nueva línea en blanco encabezada por una marca de inicio, donde esperará una nueva instrucción.

Haced una prueba: escribid `1+1` junto a la marca de inicio y pulsad *Entrar*; R escribirá en la línea siguiente el resultado de la suma, `2`, y a continuación una nueva línea en blanco encabezada por la marca de inicio. Ya hablaremos en la Lección 4 del `[1]` que os habrá aparecido delante del 2 en el resultado. Hasta entonces, no os preocupéis por él. En los

bloques de código de este libro no incluimos la marca de inicio, para que podáis copiar tranquilamente el código y luego pegarlo y ejecutarlo en vuestra consola, y el resultado aparece precedido de `##`, para que si por descuido copiáis un resultado, no se ejecute: el símbolo `#` sirve para indicar a R que no ejecute lo que venga a continuación en la misma línea. Así, en este libro el cálculo anterior corresponde a:

```
1+1
```

```
## [1] 2
```

Para facilitarnos el trabajo, la consola dispone de un mecanismo para acceder a las instrucciones ya ejecutadas y modificarlas si queremos. Si situamos el cursor a la derecha de la marca de inicio de la línea inferior y pulsamos la tecla de la flecha vertical ascendente  $\uparrow$ , iremos obteniendo de manera consecutiva, en esa línea, las instrucciones escritas hasta el momento en la misma sesión; si nos pasamos, podemos usar la tecla  $\downarrow$  para retroceder dentro de esta lista; una vez alcanzada la instrucción deseada, podemos volver a ejecutarla o, con las teclas de flechas horizontales, ir al lugar de la instrucción que queramos y reescribir un trozo antes de ejecutarla. Otra posibilidad es usar la pestaña *History* de la ventana superior derecha de *RStudio*, que contiene la lista de todas las instrucciones que se han ejecutado en la sesión actual. Si seleccionamos una instrucción de esta lista y pulsamos el botón *To console* del menú superior de la pestaña, la instrucción se copiará en la consola y la podremos modificar o ejecutar directamente.

También podemos copiar instrucciones de otros ficheros y pegarlas a la derecha de la marca de inicio de la manera habitual en el sistema operativo de nuestro ordenador. Pero hay que ir con cuidado: las instrucciones copiadas de ficheros en formato que no sea texto simple pueden contener caracteres invisibles a simple vista que generen errores al intentar ejecutar la instrucción copiada. En particular, esto afecta a las instrucciones que podáis copiar de ficheros en formato PDF, procurad no hacerlo. En cambio, no hay ningún problema en copiar y pegar instrucciones de ficheros html como los de estas lecciones.

Volvamos a la ventana de *RStudio* de la Figura 1.1. Observaréis que está dividida a su vez en tres ventanas. La de la izquierda es la consola, donde trabajamos en modo interactivo. La ventana inferior derecha tiene algunas pestañas, entre las que destacamos:

- *Files*, que muestra el contenido de la carpeta de trabajo actual (véase la Sección 1.2).

Al hacer clic sobre un fichero en esta lista, se abrirá en la ventana de ficheros (véase la Sección 1.3).

- *Plots*, que muestra los gráficos que hayamos producido durante la sesión. Se puede navegar entre ellos con las flechas de la barra superior de la pestaña.
- *Packages*, que muestra todos los paquetes instalados y, marcados, los que están cargados en la sesión actual (véase la Sección 1.5).
- *Help*, donde aparecerá la ayuda que pidamos (véase la Sección 1.4).

Por lo que se refiere a la ventana superior izquierda, destacamos las dos pestañas siguientes:

- *Environment*, con la lista de los objetos actualmente definidos (véase la Lección 2).
- *History*, de la que ya hemos hablado, que contiene la lista de todas las instrucciones que hayamos ejecutado durante la sesión.

Aparte de estas tres ventanas, *RStudio* dispone de una cuarta ventana para ficheros, que se abre en el sector superior izquierdo, sobre la consola (véase la Sección 1.3).

Para cerrar *RStudio*, basta elegir *Quit RStudio* del menú *RStudio* o pulsar la combinación de teclas usual para cerrar un programa en vuestro sistema operativo.

## 1.2 Cómo guardar el trabajo realizado

Antes de empezar a utilizar R en serio, lo primero que tenéis que hacer es crear en vuestro ordenador una carpeta específica que será vuestra **carpeta de trabajo** con R. A continuación, en las *Preferencias* de *RStudio*, que podréis abrir desde el menú *RStudio*, tenéis que declarar esta carpeta como *Default working directory*. A partir de este momento, por defecto, todo el trabajo que realicéis quedará guardado dentro de esta carpeta, y *RStudio* buscará dentro de esta carpeta todo lo que queráis que lea. Si en un momento determinado queréis cambiar temporalmente de carpeta de trabajo, tenéis dos opciones:

- Podéis usar el menú *Session* → *Set Working Directory* → *Choose Directory...* para escoger una carpeta.
- Podéis abrir la pestaña *Files* de la ventana inferior derecha y navegar por el árbol de directorios que aparece en su barra superior hasta llegar a la carpeta deseada.



Tanto de una manera como de la otra, la carpeta que especifiquéis será la carpeta de trabajo durante lo que queda de sesión o hasta que la volváis a cambiar.

En cualquier momento podéis guardar la sesión en la que estéis trabajando usando el menú *Session* → *Save Workspace as....* Además, si no habéis modificado esta opción en las *Preferencias*, cuando cerréis *RStudio* se os pedirá si queréis guardar la sesión; si contestáis que sí, *RStudio* guardará en la carpeta de trabajo dos ficheros, `.RData` y `.RHistory`, que se cargarán automáticamente al volver a abrir *RStudio* y estaréis exactamente donde lo habíais dejado. Nuestro consejo es que digáis que no: normalmente, no os interesará arrastrar todo lo que hayáis hecho en sesiones anteriores. Y si queréis guardar algunas definiciones e instrucciones de una sesión, lo más práctico es guardarlas en un *guión* (véase la Sección [1.3](#)).

Los gráficos que generéis con *RStudio* aparecerán en la ventana inferior derecha, en la pestaña *Plots* que se activa automáticamente cuando se crea alguno. La apariencia del gráfico dependerá de las dimensiones de esta ventana, por lo que es conveniente que sea cuadrada si queréis que el gráfico no aparezca achatado o estirado. Si modificáis la forma de la ventana, las dimensiones del gráfico que aparezca en ella se modificarán de manera automática.

Para guardar un gráfico, hay que ir al menú *Export* de esta pestaña y seleccionar cómo queréis guardarlo: como una imagen en uno de los formatos estándares de imágenes (.png, .jpeg, .tiff, etc.) o en formato PDF. Entonces, se abrirá una ventana donde podéis darle nombre, modificar sus dimensiones y especificar el directorio donde queráis que se guarde, entre otras opciones.

## 1.3 Cómo trabajar con guiones y otros ficheros

R admite la posibilidad de crear y usar ficheros de instrucciones que se pueden ejecutar y guardar llamados **guiones** (*scripts*). Estos guiones son una alternativa muy cómoda a las sesiones interactivas, porque permiten guardar las versiones finales de las instrucciones usadas, y no toda la sesión con pruebas, errores y resultados provisionales, y facilitan la ejecución de secuencias de instrucciones en un solo paso. Además, un guión se puede guardar, volver a abrir más adelante, editar, etc. Como ya hemos comentado, el símbolo `#` sirve para indicar a R que omita todo lo que hay a su derecha en la misma línea, lo que permite añadir comentarios a un guión.

Para crear un guión con *RStudio*, tenéis que ir al menú *File* → *New File* → *R Script*. Veréis que os aparece una ventana nueva en el sector superior izquierdo de la ventana de *RStudio*, sobre la consola: la llamaremos **ventana de ficheros**. En ella podéis escribir, línea a línea, las instrucciones que queráis. Para ejecutar instrucciones de esta ventana, basta que las seleccionéis y pulséis el botón *Run* que aparece en la barra superior de esta ventana.

Para guardar un guión, basta pulsar el botón con el icono de un disquete de ordenador que aparece en la barra superior de su ventana. Otra posibilidad es usar el menú *File* → *Save*, o pulsar la combinación de teclas usual para guardar un fichero en vuestro sistema operativo, siempre y cuando la **ventana activa** de *RStudio* (donde esté activo el cursor en ese momento) sea la del guión. Al guardar un guión por primera vez, se abre una ventana de diálogo donde *RStudio* espera que le demos un nombre; la costumbre es usar para los guiones la extensión `.R`.

Podéis abrir un guión ya preexistente con *RStudio* usando el menú *File* → *Open File* de *RStudio* o pulsando sobre él en la pestaña *Files*. También podéis arrastrar el icono del guión sobre el de *RStudio* o (si habéis declarado que la aplicación por defecto para abrir ficheros con extensión `.R` sea *RStudio*) simplemente abrir el fichero de la manera usual en vuestro sistema operativo.

Además de guiones, con *RStudio* también podemos crear otros tipos de ficheros que combinen instrucciones de R con instrucciones de otro lenguaje. En este curso lo usaremos para crear ficheros *R Markdown*, que permiten generar de manera muy cómoda informes y presentaciones que incorporen instrucciones de R (o sólo sus resultados). Para crear un fichero *R Markdown*, tenéis que ir al menú *File* → *New File* → *R Markdown...*, donde os aparecerá una ventana que os pedirá el tipo de documento (*Document*, *Presentation...*), su título y el formato de salida. Una vez completada esta información, se abrirá el fichero en la ventana superior izquierda.

Por poner un ejemplo, supongamos que habéis elegido realizar un informe (*Document*) con formato de salida html (los formatos posibles son: PDF, HTML o Word); entonces, para generar un informe básico basta sustituir las palabras clave que ha generado *RStudio* en esta ventana. Probadlo: cambiad el título y el texto; a continuación, guardad el fichero con el nombre que queráis y extensión `.Rmd`, y pulsad el botón *Knit* situado en la barra



superior de la ventana; se generará un fichero HTML en la carpeta de trabajo, con el texto del fichero *R Markdown* y el mismo título cambiando la extensión `.Rmd` por `.html`, y se abrirá en una ventana aparte.

Aprender los primeros pasos de *R Markdown* es sencillo. Para ello, podéis consultar el manual de referencia rápida de *R Markdown* que encontraréis en el menú *Help* de *RStudio*, que para la mayoría de ejercicios de este curso es más que suficiente. También os puede ser útiles las “chuletas” de *R Markdown* siguientes:

- <https://www.rstudio.com/wp-content/uploads/2015/03/rmarkdown-reference.pdf>
- <https://github.com/rstudio/cheatsheets/raw/master/rmarkdown-2.0.pdf>
- <https://www.rstudio.com/wp-content/uploads/2015/02/rmarkdown-cheatsheet.pdf>

En la Lección ?? explicamos algunas técnicas para mejorar los ficheros resultantes.

## 1.4 Cómo obtener ayuda

Para conocer toda la información (qué hace, cuál es la sintaxis correcta, qué parámetros tiene, algunos ejemplos de uso...) sobre una función o un objeto, se puede usar el campo de búsqueda, marcado con una lupa, en la esquina superior derecha de la pestaña de **Ayuda** (*Help*), situada en la ventana inferior derecha de *RStudio*. Como alternativa, se pueden usar las instrucciones

```
help(nombre del objeto)
```

o, equivalentemente,

```
?nombre del objeto
```

Por ejemplo, si entramos en el campo de búsqueda de la pestaña de Ayuda la palabra `sum`, o si **entramos** en la consola (es decir, si escribimos a la derecha de la marca de inicio y a continuación pulsamos la tecla *Entrar*) la instrucción

```
help(sum)
```

obtenemos en la pestaña de Ayuda toda la información sobre la función `sum`.

Cuando hayamos avanzado un poco en este curso, la Ayuda os será muy útil. Aquí sólo veremos alguna aplicación simple de la mayoría de las funciones que estudiemos, con los parámetros más importantes y suficientes para nuestros propósitos, y necesitaréis consultar su Ayuda para conocer todos sus usos, todos sus parámetros u otra información relevante.

Si queremos pedir ayuda sobre un tema concreto, pero no sabemos el nombre exacto de la función, podemos entrar una palabra clave en el campo de búsqueda de la pestaña de Ayuda, o usar la función

```
help.search("palabra clave")
```

o, equivalentemente,

```
??"palabra clave"
```

(las comillas ahora son obligatorias). De esta manera, conseguiremos en la ventana de Ayuda una lista de las funciones que R entiende que están relacionadas con la palabra clave entrada. Entonces, pulsando en la función que nos interese de esta lista, aparecerá la información específica sobre ella. Como podéis imaginar, conviene que la palabra clave esté en inglés.

Además de la ayuda que incorpora el mismo R, siempre podéis acudir a foros y listas de discusión para encontrar ayuda sobre cualquier duda que podáis tener. Algunos recursos que nosotros encontramos especialmente útiles son los siguientes:

- La sección dedicada a R del foro [stackoverflow](#)
- El archivo de la lista de discusión [R-help](#)
- El grupo de Facebook [R project en español](#)

Si tenéis alguna dificultad, es muy probable que alguien ya la haya tenido y se la hayan resuelto en alguno de estos foros.

Existe también una comunidad muy activa de usuarios hispanos de R, en cuyo [portal web](#) encontraréis muchos recursos útiles para mejorar vuestro conocimiento de este lenguaje.

## 1.5 Cómo instalar y cargar paquetes

Muchas funciones y tablas de datos útiles no vienen con la instalación básica de R, sino que forman parte de **paquetes** (*packages*), que se tienen que instalar y cargar para poderlos usar. Por citar un par de ejemplos, el paquete **magic** lleva una función `magic` que crea **cuadrados mágicos** (tablas cuadradas de números naturales diferentes dos a dos tales que las sumas de todas sus columnas, de todas sus filas y de sus dos diagonales principales valgan todas lo mismo), y para usarla tenemos que instalar y cargar este paquete. De manera similar, el paquete **ggplot2** incorpora una serie de funciones para dibujar gráficos avanzados que no podemos usar si primero no instalamos y cargamos este paquete.

Podemos consultar en la pestaña *Packages* la lista de paquetes que tenemos instalados. Los paquetes que aparecen marcados en esta lista son los que tenemos cargados en la sesión actual. Si queremos cargar un paquete ya instalado, basta marcarlo en esta lista; podemos hacerlo también desde la consola, con la instrucción

```
library(nombre del paquete)
```

En caso de necesitar un paquete que no tengamos instalado, hay que instalarlo antes de poderlo cargar. La mayoría de los paquetes se pueden instalar desde el repositorio del CRAN; esto se puede hacer de dos maneras:

- Desde la consola, entrando la instrucción

```
install.packages("nombre del paquete", dep=TRUE)
```

(las comillas son obligatorias, y fijaos en el plural de `packages` aunque sólo queráis instalar uno). El parámetro `dep=TRUE` hace que R instale no sólo el paquete requerido, sino todos aquellos de los que dependa para funcionar correctamente.

- Pulsando el botón *Install* de la barra superior de la pestaña de paquetes; al hacerlo, *RStudio* abre una ventana dónde se nos pide el nombre del paquete a instalar. Conviene dejar marcada la opción *Install dependencies*, para que se instalen también los paquetes necesarios para su funcionamiento.

Así, supongamos que queremos construir cuadrados mágicos, pero aún no hemos cargado el paquete `magic`.

```
magic(10)
```

```
## Error in magic(10): could not find function "magic"
```

Así que instalamos y cargamos dicho paquete (también lo podríamos hacer desde la ventana *Packages*):

```
install.packages("magic", dep=TRUE)  
library(magic)
```

Ahora ya podemos usar la función `magic`:

```
magic(10)
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]  
## [1,]  34  35   6   7  98  99  70  71  42  43  
## [2,]  36  33   8   5 100  97  72  69  44  41  
## [3,]  11  10  83  82  75  74  47  46  39  38  
## [4,]  12   9  84  81  73  76  48  45  40  37  
## [5,]  87  86  79  78  51  50  23  22  15  14  
## [6,]  85  88  77  80  52  49  21  24  13  16  
## [7,]  63  62  55  54  27  26  19  18  91  90  
## [8,]  61  64  53  56  25  28  17  20  89  92  
## [9,]  59  58  31  30   3   2  95  94  67  66  
## [10,]  57  60  29  32   1   4  93  96  65  68
```

Cuando cerramos *RStudio*, los paquetes instalados en la sesión siguen instalados, pero los cargados se pierden; por lo tanto, si queremos volver a usarlos en otra sesión, tendremos que volver a cargarlos.

Hay paquetes que no se encuentran en el CRAN y que, por lo tanto, no se pueden instalar de la forma que hemos visto. Cuando sea necesario, ya explicaremos la manera de instalarlos y cargarlos en cada caso.

Para terminar, observad que a la derecha del nombre de cada paquete en la pestaña *Packages* aparecen dos símbolos. Al pulsar el primero, seis puntitos, se abrirá en el navegador la página de información del paquete, y al pulsar el segundo, una crucecita, desinstalamos el paquete. Asimismo, en la barra superior de la pestaña *Packages* encontraréis un botón *Update* que sirve para poner al día los paquetes instalados, obteniendo sus últimas versiones publicadas.

## 1.6 Guía rápida

- `help` o `?` permiten pedir información sobre una función. También se puede usar el campo de búsqueda de la pestaña *Help* en la ventana inferior derecha de *RStudio*.
- `help.search` o `??` permiten pedir información sobre una palabra clave (entrada entre comillas). De nuevo, también se puede usar el campo de búsqueda de la pestaña *Help* en la ventana inferior derecha de *RStudio*.
- `install.packages("paquete", dep=TRUE)` instala el `paquete` y todos los otros paquetes de los que dependa. También se puede usar el botón *Install* de la pestaña *Packages* en la ventana inferior derecha de *RStudio*.
- `library(paquete)` carga el `paquete`. También se puede cargar marcándolo en la ventana *Packages* de *RStudio*.

## Lección 2 La calculadora

Cuando se trabaja en modo interactivo en la consola de R, hay que escribir las instrucciones a la derecha de la marca de inicio `>` de la línea inferior (que omitimos en los bloques de código de este libro). Para evaluar una instrucción al terminar de escribirla, se tiene que pulsar la tecla *Entrar* ( $\hookrightarrow$ ); así, por ejemplo, si junto a la marca de inicio escribimos `2+3` y pulsamos *Entrar*, R escribirá en la línea siguiente el resultado, 5, y a continuación una nueva línea en blanco encabezada por la marca de inicio, donde podremos continuar entrando instrucciones.

```
2+3 #Y ahora aquí pulsamos Entrar
```

```
## [1] 5
```

Bueno, hemos hecho trampa. Como ya habíamos comentado en la lección anterior, se pueden escribir comentarios: R ignora todo lo que se escribe en la línea después de un signo `#`. También podéis observar que R ha dado el resultado en una línea que empieza con `[1]`; ya discutiremos en la Lección 4 qué significa este `[1]`.

Si la expresión que entramos no está completa, R no la evaluará y en la línea siguiente esperará a que la acabemos, indicándolo con la **marca de continuación**, por defecto un signo `+`. (En estas notas, y excepto en el ejemplo que damos a continuación, no mostraremos este signo `+` para no confundirlo con una suma.) Además, si cometemos algún error de sintaxis, R nos avisará con un mensaje de error.

```
2*(3+5 #Pulsamos Entrar, pero no hemos acabado  
+ ) #ahora sí
```

```
## [1] 16
```



```
2*3+5)
```

```
## Error: <text>:1:6: unexpected ')\n## 1: 2*3+5)\n##           ^
```

Como podemos ver, al ejecutar la segunda instrucción, R nos avisa de que el paréntesis no está en su sitio.

Se puede agrupar más de una instrucción en una sola línea separándolas con signos de punto y coma. Al pulsar la tecla *Entrar*, R las ejecutará todas, una tras otra y en el orden en el que las hayamos escrito.

```
2+3; 2+4; 2+5
```

```
## [1] 5
```

```
## [1] 6
```

```
## [1] 7
```

## 2.1 Números reales: operaciones y funciones básicas

La separación entre la parte entera y la parte decimal en los números reales se indica con un punto, no con una coma. Por consistencia, en el texto también seguiremos el convenio angloamericano de usar un punto en lugar de una coma como separador decimal.

```
2+2,5
```

```
## Error: <text>:1:4: unexpected ' , '
```

```
## 1: 2+2,
```

```
##      ^
```

```
2+2.5
```

```
## [1] 4.5
```

Las operaciones usuales se indican en R con los signos que damos en la lista siguiente. Por lo que se refiere a los dos últimos operadores en esta lista, recordad que si  $a$  y  $b$  son dos números reales, con  $b > 0$ , la **división entera** de  $a$  por  $b$  da como **cociente entero** el mayor número entero  $q$  tal que  $q \cdot b \leq a$ , y como **resto** la diferencia  $a - q \cdot b$ . Por ejemplo, la división entera de 29.5 entre 6.3 es  $29.5 = 4 \cdot 6.3 + 4.3$ , con cociente entero 4 y resto 4.3. (Cuando  $b < 0$ , R da como cociente entero el menor número entero  $q$  tal que  $q \cdot b \geq a$ , y como resto la diferencia  $a - q \cdot b$ , que en este caso es negativa.)

- **Suma:** `+`
- **Resta:** `-`
- **Multiplicación:** `*`
- **División:** `/`
- **Potencia:** `^`
- **Cociente entero:** `%/%`
- **Resto de la división entera:** `%%`

A continuación, damos algunos ejemplos de manejo de estas operaciones. Observad el uso natural de los paréntesis para indicar la precedencia de las operaciones.

```
2*3+5/2
```

```
## [1] 8.5
```

```
2*(3+5/2) #Aquí lo único que dividimos entre 2 es 5
```

```
## [1] 11
```

```
2*((3+5)/2)
```

```
## [1] 8
```

```
2/3+4 #Aquí el denominador de la fracción es 3
```

```
## [1] 4.666667
```

```
2/(3+4)
```

```
## [1] 0.2857143
```

```
2^3*5 #Aquí el exponente es 3
```

```
## [1] 40
```

```
2^(3*5)
```

```
## [1] 32768
```

```
2^-5 #En este caso no hacen falta paréntesis...
```

```
## [1] 0.03125
```

```
2^(-5) #Pero queda más claro si se usan
```

```
## [1] 0.03125
```

```
534/%7 #¿Cuántas semanas completas caben en 534 días?
```

```
## [1] 76
```

```
534%%7 #¿Y cuántos días sobran?
```

```
## [1] 2
```

```
534-76*7
```

```
## [1] 2
```

El objeto `pi` representa el número real  $\pi$ .

```
pi
```

```
## [1] 3.141593
```

¡Cuidado! No podemos omitir el signo `*` en las multiplicaciones.

```
2(3+5)
```

```
## Error in eval(expr, envir, enclos): attempt to apply non-function
```

```
2*(3+5)
```

```
## [1] 16
```

```
2pi
```

```
## Error: <text>:1:2: unexpected symbol
```

```
## 1: 2pi
```

```
##      ^
```

```
2*pi
```

```
## [1] 6.283185
```

Cuando un número es muy grande o muy pequeño, R emplea la llamada **notación científica** para dar una aproximación.

```
2^40
```

```
## [1] 1.099512e+12
```

```
2^(-20)
```

```
## [1] 9.536743e-07
```

En este ejemplo, `1.099512e+12` representa el número  $1.099512 \cdot 10^{12}$ , es decir, 1099512000000, y `9.536743e-07` representa el número  $9.536743 \cdot 10^{-7}$ , es decir, 0.0000009536743. Como muestra el ejemplo siguiente, no es necesario que un número

sea especialmente grande o pequeño para que R lo escriba en notación científica: basta que esté rodeado de otros números en esa notación.

```
c(2^40,2^(-20),17/3) #La función c sirve para definir vectores
```

```
## [1] 1.099512e+12 9.536743e-07 5.666667e+00
```

Este `5.666667e+00` representa el número  $5.666667 \cdot 10^0$ , es decir, 5.666667.

R dispone, entre muchas otras, de las funciones numéricas de la lista siguiente:

- **Valor absoluto**,  $|x|$ : `abs(x)`
- **Raíz cuadrada**,  $\sqrt{x}$ : `sqrt(x)`
- **Exponencial**,  $e^x$ : `exp(x)`
- **Logaritmo neperiano**,  $\ln(x)$ : `log(x)`
- **Logaritmo decimal**,  $\log_{10}(x)$ : `log10(x)`
- **Logaritmo binario**,  $\log_2(x)$ : `log2(x)`
- **Logaritmo en base  $a$** ,  $\log_a(x)$ : `log(x,a)`
- **Factorial**,  $n!$ : `factorial(n)`
- **Número combinatorio**,  $\binom{n}{m}$ : `choose(n,m)`
- **Seno**,  $\sin(x)$ : `sin(x)`
- **Coseno**,  $\cos(x)$ : `cos(x)`
- **Tangente**,  $\tan(x)$ : `tan(x)`
- **Arcoseno**,  $\arcsin(x)$ : `asin(x)`
- **Arcocoseno**,  $\arccos(x)$ : `acos(x)`
- **Arcotangente**,  $\arctan(x)$ : `atan(x)`



Recordad que el **valor absoluto**  $|x|$  de un número  $x$  se obtiene tomando  $x$  sin signo:  $|-8| = |8| = 8$ . Recordad también que el **factorial**  $n!$  de  $n$ , es el producto

$$n! = n \cdot (n - 1) \cdot (n - 2) \cdots 3 \cdot 2 \cdot 1$$

(con el convenio de que  $0! = 1$ ), y es igual al número de maneras posibles de ordenar una lista de  $n$  objetos diferentes (su número de **permutaciones**), y que el **número combinatorio**  $\binom{n}{m}$ , con  $m \leq n$ , es

$$\binom{n}{m} = \frac{n!}{m! \cdot (n - m)!} = \frac{n(n - 1)(n - 2) \cdots (n - m + 1)}{m(m - 1)(m - 2) \cdots 2 \cdot 1},$$

y es igual al número de maneras posibles de escoger un subconjunto de  $m$  elementos de un conjunto de  $n$  objetos diferentes.

Las funciones de R se aplican a sus argumentos introduciéndolos siempre entre paréntesis. Si la función se tiene que aplicar a más de un argumento, éstos se tienen que especificar en el orden que toque y separándolos mediante comas; R no tiene en cuenta los espacios en blanco alrededor de las comas. Veamos algunos ejemplos:

```
sqrt(4)
```

```
## [1] 2
```

```
sqrt(8)-8^(1/2)
```

```
## [1] 0
```

```
log10(8)
```

```
## [1] 0.90309
```

```
log(8)/log(10)
```

```
## [1] 0.90309
```

```
7^log(2,7) #7 elevado al logaritmo en base 7 de 2 es 2
```

```
## [1] 2
```

```
10! #R no entiende esta expresión
```

```
## Error: <text>:1:3: unexpected '!'
```

```
## 1: 10!
```

```
##      ^
```

```
factorial(10)
```

```
## [1] 3628800
```

```
exp(sqrt(8))
```

```
## [1] 16.91883
```

```
choose(5,3) #Núm. de subconjuntos de 3 elementos de un conjunto de 5
```

```
## [1] 10
```

```
choose(3,5) #Núm. de subconjuntos de 5 elementos de un conjunto de 3
```

```
## [1] 0
```

R entiende que los argumentos de las funciones `sin`, `cos` y `tan` están en radianes. Si queremos aplicar una de estas funciones a un número de grados, podemos traducir los grados a radianes multiplicándolos por  $\pi/180$ . De manera similar, los resultados de `asin`, `acos` y `atan` también están en radianes, y se pueden traducir a grados multiplicándolos por  $180/\pi$ .

```
cos(60) #Coseno de 60 radianes
```

```
## [1] -0.952413
```

```
cos(60*pi/180) #Coseno de 60 grados
```

```
## [1] 0.5
```

```
acos(0.5) #Arcocoseno de 0.5 en radianes
```

```
## [1] 1.047198
```

```
acos(0.5)*180/pi #Arcocoseno de 0.5 en grados
```

```
## [1] 60
```

```
acos(2)
```

```
## [1] NaN
```

Este último `NaN` (acrónimo de `Not a Number`) significa que el resultado no existe; en efecto,  $\arccos(2)$  no existe como número real, ya que  $\cos(x)$  siempre pertenece al intervalo  $[-1, 1]$ .

Ya hemos visto que R dispone del signo `pi` para representar el número real  $\pi$ . En cambio, no tiene ningún signo para indicar la constante de Euler  $e$ , y hay que emplear `exp(1)` .

```
2*exp(1) #2·e
```

```
## [1] 5.436564
```

```
exp(pi)-pi^exp(1) #e^pi-pi^e
```

```
## [1] 0.6815349
```

Para terminar esta sección, observad el resultado siguiente:

```
sqrt(2)^2-2
```

```
## [1] 4.440892e-16
```

R opera numéricamente con  $\sqrt{2}$ , no formalmente, y por eso no da como resultado de  $(\sqrt{2})^2 - 2$  el valor 0 exacto, sino el número pequeñísimo  $4.440892 \cdot 10^{-16}$ ; de hecho, R trabaja internamente con una precisión de aproximadamente 16 cifras decimales, por lo que no siempre podemos esperar resultados exactos. Si necesitáis trabajar de manera exacta con más cifras significativas, os recomendamos usar las funciones del paquete `Rmpfr` .

## 2.2 Cifras significativas y redondeos

En cada momento, R decide cuántas cifras muestra de un número según el contexto. También podemos especificar este número de cifras para toda una sesión, entrándolo en lugar de los puntos suspensivos en `options(digits=...)` . Hay que tener presente que ejecutar esta instrucción no cambiará la precisión de los cálculos, sólo cómo se muestran los resultados.

Si queremos conocer una cantidad específica  $n$  de cifras significativas de un número  $x$ , podemos emplear la función

```
print(x, n)
```

Observad su efecto:

```
sqrt(2)
```

```
## [1] 1.414214
```

```
print(sqrt(2), 20)
```

```
## [1] 1.4142135623730951455
```

```
print(sqrt(2), 2)
```

```
## [1] 1.4
```

```
2^100
```

```
## [1] 1.267651e+30
```

```
print(2^100, 15)
```

```
## [1] 1.26765060022823e+30
```

```
print(2^100, 5)
```

```
## [1] 1.2677e+30
```

El número máximo de cifras que podemos pedir con `print` es 22; si pedimos más, R nos dará un mensaje de error.

```
print(sqrt(2), 22)
```

```
## [1] 1.414213562373095145475
```

```
print(sqrt(2), 23)
```

```
## Error in print.default(sqrt(2), 23): invalid 'digits' argument
```

Por otro lado, hay que tener en cuenta que, como ya hemos comentado, R trabaja con una precisión de unas 16 cifras decimales y por lo tanto los dígitos más allá de esta precisión pueden ser incorrectos. Por ejemplo, si le pedimos las 22 primeras cifras de  $\pi$ , obtenemos el resultado siguiente:

```
print(pi, 22)
```

```
## [1] 3.141592653589793115998
```



En cambio,  $\pi$  vale en realidad 3.141592653589793**238462...**, lo que significa que el valor que da R es erróneo a partir de la decimosexta cifra decimal.

La función `print` permite indicar las cifras que queremos *leer*, pero no sirve para especificar las cifras decimales con las que queremos *trabajar*. Para *redondear* un número  $x$  a una cantidad específica  $n$  de cifras decimales, y trabajar solamente con esas cifras, hay que usar la función

```
round(x, n)
```

La diferencia entre los efectos de `print` y `round` consiste en que `print(sqrt(2), 4)` es igual a  $\sqrt{2}$ , pero R sólo muestra sus primeras 4 cifras, 1.414, mientras que `round(sqrt(2), 3)` es *igual a* 1.414. Veamos algunos ejemplos

```
print(sqrt(2), 4)
```

```
## [1] 1.414
```

```
print(sqrt(2), 4)^2
```

```
## [1] 1.414
```

```
## [1] 2
```

```
1.414^2
```

```
## [1] 1.999396
```

```
round(sqrt(2), 3)
```

```
## [1] 1.414
```

```
round(sqrt(2), 3)^2
```

```
## [1] 1.999396
```

En caso de empate, R redondea al valor que termina en cifra par, siguiendo la regla de redondeo en caso de empate recomendada por el estándar IEEE 754 para aritmética en coma flotante.

```
round(2.25, 1)
```

```
## [1] 2.2
```

```
round(2.35, 1)
```

```
## [1] 2.4
```

¿Qué pasa si no se indica el número de cifras en el argumento de `round` ?

```
round(sqrt(2))
```

```
## [1] 1
```

```
round(sqrt(2), 0)
```

```
## [1] 1
```

Al entrar `round(sqrt(2))` , R ha entendido que el número de cifras decimales al que queríamos redondear era 0. Esto significa que 0 es el **valor por defecto** de este parámetro. No es necesario especificar los valores por defecto de los parámetros de una función, y para saber cuáles son, hay que consultar su Ayuda. Así, por ejemplo, la Ayuda de `round` indica que su sintaxis es

```
round(x, digits=0)
```

donde el valor de `digits` ha de ser un número entero que indique el número de cifras decimales. Esta sintaxis significa que el valor por defecto del parámetro `digits` es 0.

Escribir `digits=` en el argumento para especificar el número de cifras decimales es optativo, siempre que mantengamos el orden de los argumentos indicado en la Ayuda: en este caso, primero el número y luego las cifras. Este es el motivo por el que podemos escribir `round(sqrt(2), 1)` en lugar de `round(sqrt(2), digits=1)` . Si cambiamos el orden de los argumentos, entonces sí que hay que especificar el nombre del parámetro, como muestra el siguiente ejemplo:

```
round(digits=3, sqrt(2))
```

```
## [1] 1.414
```

```
round(3, sqrt(2))
```

```
## [1] 3
```

En la lista de funciones ya vimos una función de dos argumentos que toma uno por defecto: `log` . Su sintaxis completa es `log(x, base=...)` , y si no especificamos la `base` , toma su valor por defecto,  $e$ , y calcula el logaritmo neperiano.

La función `round(x)` redondea  $x$  al valor entero más cercano (y en caso de empate, al que termina en cifra par). R también dispone de otras funciones que permiten redondear a números enteros en otros sentidos específicos:

- `floor(x)` redondea  $x$  a un número entero **por defecto**, dando el mayor número entero menor o igual que  $x$ , que denotamos por  $\lfloor x \rfloor$ .
- `ceiling(x)` redondea  $x$  a un número entero **por exceso**, dando el menor número entero mayor o igual que  $x$ , que denotamos por  $\lceil x \rceil$ .
- `trunc(x)` da la **parte entera** de  $x$ , eliminando la parte decimal: es lo que se llama **truncar**  $x$  a un entero.

```
floor(8.3) #El mayor entero menor o igual que 8.3
```

```
## [1] 8
```

```
ceiling(8.3) #El menor entero mayor o igual que 8.3
```

```
## [1] 9
```

```
trunc(8.3) #La parte entera de 8.3
```

```
## [1] 8
```

```
round(8.3) #El entero más cercano a 8.3
```

```
## [1] 8
```

```
floor(-3.7) #El mayor entero menor o igual que -3.7
```

```
## [1] -4
```

```
ceiling(-3.7) #El menor entero mayor o igual que -3.7
```

```
## [1] -3
```

```
trunc(-3.7) #La parte entera de -3.7
```

```
## [1] -3
```

```
round(-3.7) #El entero más cercano a -3.7
```

```
## [1] -4
```

## 2.3 Definición de variables

R funciona mediante **objetos**, estructuras de diferentes tipos que sirven para realizar diferentes tareas. Una **variable** es un tipo de objeto que sirve para guardar datos. Por ejemplo, si queremos crear una variable `x` que contenga el valor  $\pi^2$ , podemos escribir:

```
x=pi^2
```

Al entrar esta instrucción, R creará el objeto `x` y le asignará el valor que hemos especificado. En general, se puede crear una variable y asignarle un valor, o asignar un nuevo valor a una variable definida anteriormente, mediante la construcción

```
nombre_de_la_variable=valor
```

También se puede conectar el nombre de la variable con el valor por medio de una flecha `->` o `<-`, compuesta de un guión y un signo de desigualdad, de manera que el sentido de la flecha vaya del valor a la variable; por ejemplo, las tres primeras instrucciones

siguientes son equivalentes, y asignan el valor 2 a la variable  $x$ , mientras que las dos últimas son incorrectas:

```
x=2  
x<-2  
2->x
```

```
2=x
```

```
## Error in 2 = x: invalid (do_set) left-hand side to assignment
```

```
2<-x
```

```
## Error in 2 <- x: invalid (do_set) left-hand side to assignment
```

Nosotros usaremos sistemáticamente el signo `=` para hacer asignaciones.

Se puede usar como nombre de una variable cualquier palabra que combine letras mayúsculas y minúsculas (R las distingue), con acentos o sin (aunque os recomendamos que no uséis letras acentuadas, ya que se pueden importar mal de un ordenador a otro), dígitos (0,..., 9), puntos `.` y guiones bajos `_`, siempre que empiece con una letra o un punto. Aunque no esté prohibido, es muy mala idea redefinir nombres que ya sepáis que tienen significado para R, como por ejemplo `pi` o `sqrt`.

Como podéis ver en las instrucciones anteriores y en las que siguen, cuando asignamos un valor a una variable, R no da ningún resultado; después podemos usar el nombre de la variable para referirnos al valor que representa. Es posible asignar varios valores a una misma variable en una misma sesión: naturalmente, en cada momento R empleará el último valor asignado. Incluso se puede redefinir el valor de una variable usando en la nueva definición su valor actual.

```
x=5  
x^2
```



```
## [1] 25
```

```
x=x-2 #Redefinimos x como su valor actual menos 2  
x
```

```
## [1] 3
```

```
x^2
```

```
## [1] 9
```

```
x=sqrt(x) #Redefinimos x como la raíz cuadrada de su valor actual  
x
```

```
## [1] 1.732051
```

## 2.4 Definición de funciones

A menudo queremos definir alguna función. Para ello tenemos que usar, en vez de simplemente `=`, una construcción especial:

```
nombre_de_la_función=function(variables){definición}
```

Una vez definida una función, la podemos aplicar a valores de la variable o variables.

Veamos un ejemplo. Vamos a llamar  $f$  a la función  $x^2 - 2^x$ , usando  $x$  como variable, y a continuación la aplicamos a  $x = 30$ :

```
f=function(x){x^2-2^x}  
f(30)
```

```
## [1] -1073740924
```

Conviene que os acostumbréis a escribir la fórmula que define la función entre llaves `{...}` . A veces es necesario y a veces no, pero no vale la pena discutir cuándo.

El nombre de la variable se indica dentro de los paréntesis que siguen al `function` . En el ejemplo anterior, la variable era  $x$ , y por eso hemos escrito `=function(x)` . Si hubiéramos querido definir la función con variable  $t$ , habríamos usado `=function(t)` (y, naturalmente, habríamos escrito la fórmula que define la función con la variable  $t$ ):

```
f=function(t){t^2-2^t}
```

Se pueden definir funciones de dos o más variables con `function` , declarándolas todas. Por ejemplo, para definir la función  $f(x, y) = e^{(2x-y)^2}$ , tenemos que entrar

```
f=function(x, y){exp((2*x-y)^2)}
```

y ahora ya podemos aplicar esta función a pares de valores:

```
f(0, 1)
```

```
## [1] 2.718282
```

```
f(1, 0)
```

```
## [1] 54.59815
```

Las funciones no tienen por qué tener como argumentos o resultados sólo números reales: pueden involucrar vectores, matrices, tablas de datos, etc. Y se pueden definir por medio de secuencias de instrucciones, no sólo mediante fórmulas numéricas directas; en este caso, hay que separar las diferentes instrucciones con signos de punto y coma o escribir cada instrucción en una nueva línea. Ya iremos viendo ejemplos a medida que avance el curso.

En cada momento se pueden saber los objetos (por ejemplo, variables y funciones) que se han definido en la sesión hasta ese momento entrando la instrucción `ls()` o consultando la pestaña *Environment*. Para borrar la definición de un objeto, hay que aplicarle la función `rm`. Si se quiere hacer limpieza y borrar de golpe las definiciones de todos los objetos que se han definido hasta el momento, se puede emplear la instrucción `rm(list=ls())` o usar el botón con el icono de la escoba de la barra superior de la pestaña *Environment*.

```
rm(list=ls())    #Borramos todas las definiciones
f=function(t){t^2-2^t}
a=1
a
```

```
## [1] 1
```

```
ls()
```

```
## [1] "a" "f"
```

```
rm(a)
ls()
```

```
## [1] "f"
```

```
a
```

```
## Error in eval(expr, envir, enclos): object 'a' not found
```

## 2.5 Números complejos (opcional)

Hasta aquí, hemos operado con números reales. Con R también podemos operar con números complejos. Los signos para las operaciones son los mismos que en el caso real.

```
(2+5i)*3
```

```
## [1] 6+15i
```

```
(2+5i)*(3+7i)
```

```
## [1] -29+29i
```

```
(2+5i)/(3+7i)
```

```
## [1] 0.7068966+0.0172414i
```

Fijaos en que cuando entramos en R un número complejo escrito en forma binomial  $a + bi$ , *no* escribimos un `*` entre la `i` y su coeficiente; de hecho, *no hay que escribirlo* :

```
2+5*i
```

```
## Error in eval(expr, envir, enclos): object 'i' not found
```

Por otro lado, si el coeficiente de  $i$  es 1 o -1, hay que escribir el 1: por ejemplo,  $3 - i$  se tiene que escribir `3-1i` . Si no lo hacemos, R da un mensaje de error.

```
(3+i)*(2-i)
```

```
## Error in eval(expr, envir, enclos): object 'i' not found
```

```
(3+1i)*(2-1i)
```

```
## [1] 7-1i
```

Los complejos que tienen como parte imaginaria un número entero o un racional escrito en forma decimal se pueden entrar directamente en forma binomial, como lo hemos hecho hasta ahora. Para definir números complejos más... complejos, se puede usar la función

```
complex(real=..., imaginary=...)
```

Veamos un ejemplo:

```
1+2/3i #Esto en realidad es 1 más 2 partido por 3i
```

```
## [1] 1-0.666667i
```

```
1+(2/3)i
```

```
## Error: <text>:1:8: unexpected symbol
```

```
## 1: 1+(2/3)i
```

```
##      ^
```

```
complex(real=1,imaginary=2/3)
```

```
## [1] 1+0.666667i
```

```
z=1+sqrt(2)i
```

```
## Error: <text>:1:12: unexpected symbol
```

```
## 1: z=1+sqrt(2)i
```

```
##          ^
```

```
z=complex(real=1, imaginary=sqrt(2))
```

```
z
```

```
## [1] 1+1.414214i
```

Como sabéis, los números complejos se inventaron para poder trabajar con raíces cuadradas de números negativos. Ahora bien, por defecto, cuando calculamos la raíz cuadrada de un número negativo R no devuelve un número complejo, sino que se limita a avisarnos de que no existe.

```
sqrt(-3)
```

```
## Warning in sqrt(-3): NaNs produced
```

```
## [1] NaN
```

Si queremos que R produzca un número complejo al calcular la raíz cuadrada de un número negativo, tenemos que especificar que este número negativo es un número complejo. La mejor manera de hacerlo es declarándolo como complejo aplicándole la función `as.complex`

```
sqrt(as.complex(-3))
```

```
## [1] 0+1.732051i
```

La mayoría de las funciones que hemos dado para los números reales admiten extensiones para números complejos, y con R se calculan con la misma función. Ahora no entraremos a explicar cómo se definen estas extensiones, sólo lo comentamos por si sabéis qué hacen y os interesa calcularlas.

```
sqrt(2+3i)
```

```
## [1] 1.674149+0.895977i
```

```
exp(2+3i)
```

```
## [1] -7.31511+1.042744i
```

```
sin(2+3i)
```

```
## [1] 9.154499-4.168907i
```

```
acos(as.complex(2)) #EL arcocoseno de 2 es un número complejo
```

```
## [1] 0+1.316958i
```

La raíz cuadrada merece un comentario. Naturalmente, `sqrt(2+3i)` calcula un número complejo  $z$  tal que  $z^2 = 2 + 3i$ . Como ocurre con los números reales, todo número complejo diferente de 0 tiene dos raíces cuadradas, y una se obtiene multiplicando la otra por -1. R da como raíz cuadrada de un número real la positiva, y como raíz cuadrada de un complejo la que tiene parte real positiva, y si su parte real es 0, la que tiene parte imaginaria positiva.

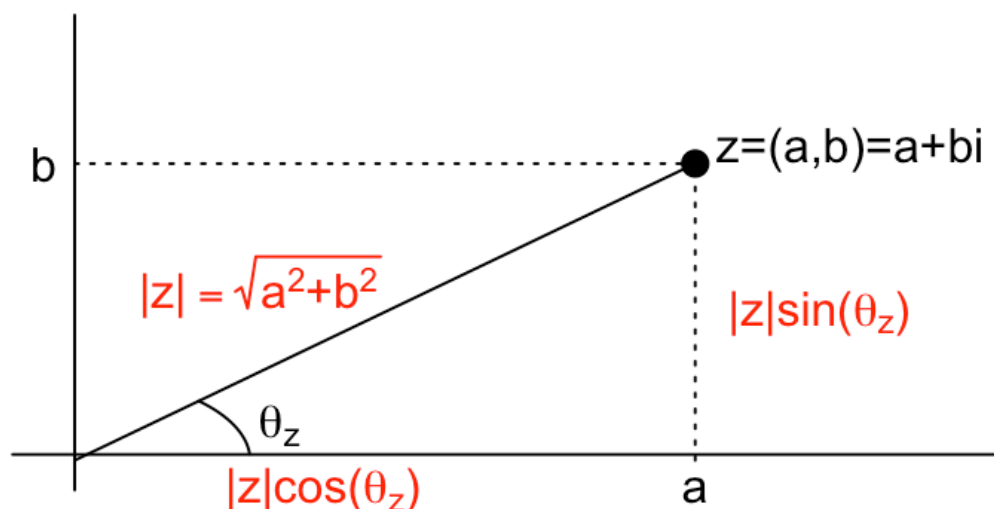


Figura 2.1: Interpretación geométrica de los números complejos.

Un número complejo  $z = a + bi$  se puede representar como el punto  $(a, b)$  del plano cartesiano  $\mathbb{R}^2$ . Esto permite asociarle dos magnitudes geométricas: véase la Figura 2.1

- El **módulo** de  $z$ , que denotaremos por  $|z|$ , es la distancia euclídea de  $(0, 0)$  a  $(a, b)$ :

$$|z| = \sqrt{a^2 + b^2}.$$

Si  $z \in \mathbb{R}$ , su módulo coincide con su valor absoluto; en particular, si  $z = 0$ , su módulo es 0, y es el único número complejo de módulo 0.

- El **argumento** de  $z$  (para  $z \neq 0$ ), que denotaremos por  $\theta_z$ , es el ángulo que forman el semieje positivo de abscisas y el vector que va de  $(0, 0)$  a  $(a, b)$ . Este ángulo está determinado por las ecuaciones

$$\cos(\theta_z) = \frac{a}{\sqrt{a^2 + b^2}}, \quad \sin(\theta_z) = \frac{b}{\sqrt{a^2 + b^2}}.$$

R sabe calcular módulos y argumentos de números complejos. Los argumentos los da en radianes y dentro del intervalo  $(-\pi, \pi]$ . En general, R dispone de las funciones básicas específicas para números complejos de la lista siguiente:

- **Parte real:** `Re`



- **Parte imaginaria:** Im
- **Módulo:** Mod
- **Argumento:** Arg
- **Conjugado:** Conj

Recordad que el **conjugado** de un número complejo  $z = a + bi$  es  $\bar{z} = a - bi$ . Veamos algunos ejemplos de uso de estas funciones:

```
Re(4-7i)
```

```
## [1] 4
```

```
Im(4-7i)
```

```
## [1] -7
```

```
Mod(4-7i)
```

```
## [1] 8.062258
```

```
Arg(4-7i)
```

```
## [1] -1.05165
```

```
Conj(4-7i)
```

```
## [1] 4+7i
```

El módulo y el argumento de un número complejo  $z \neq 0$  lo determinan de manera única, porque

$$z = |z| \left( \cos(\theta_z) + \sin(\theta_z)i \right).$$

Si queremos definir un número complejo mediante su módulo y argumento, no hace falta utilizar esta igualdad: podemos usar la instrucción

```
complex(modulus=..., argument=...)
```

Por ejemplo:

```
z=complex(modulus=3, argument=pi/5)
```

```
z
```

```
## [1] 2.427051+1.763356i
```

```
Mod(z)
```

```
## [1] 3
```

```
Arg(z)
```

```
## [1] 0.6283185
```

```
pi/5
```

```
## [1] 0.6283185
```

## 2.6 Guía rápida

- Signos de operaciones aritméticas:

- Suma: `+`
- Resta: `-`
- Multiplicación: `*`
- División: `/`
- Potencia: `^`
- Cociente entero: `/%`
- Resto de la división entera: `%%`

- Funciones numéricas:

- Valor absoluto: `abs`
- Raíz cuadrada: `sqrt`
- Exponencial de base e: `exp`
- Logaritmo neperiano: `log`
- Logaritmo decimal: `log10`
- Logaritmo binario: `log2`
- Logaritmo en base  $a$ : `log(...,base=a)`
- Factorial: `factorial`
- Número combinatorio: `choose`
- Seno: `sin`
- Coseno: `cos`
- Tangente: `tan`
- Arcoseno: `asin`
- Arcocoseno: `acos`
- Arcotangente: `atan`

- `pi` es el número  $\pi$ .
- `print(x, n)` muestra el valor de  $x$  con  $n$  cifras significativas.
- `round(x, n)` redondea el valor de  $x$  a  $n$  cifras decimales.
- `floor(x)` redondea  $x$  a un número entero por defecto.
- `ceiling(x)` redondea  $x$  a un número entero por exceso.
- `trunc(x)` da la parte entera de  $x$ .
- `variable=valor` asigna el `valor` a la `variable`. Otras construcciones equivalentes son `variable<-valor` y `valor->variable`.
- `función=function(variables){instrucciones}` define la `función` de variables las especificadas entre los paréntesis mediante las instrucciones especificadas entre las llaves.
- `ls()` nos da la lista de objetos actualmente definidos.
- `rm` borra la definición del objeto u objetos a los que se aplica.
- `rm(list=ls())` borra las definiciones de todos los objetos que hayamos definido.
- `complex` se usa para definir números complejos que no se puedan entrar directamente en forma binomial. Algunos parámetros importantes:
  - `real` e `imaginary` : sirven para especificar su parte real y su parte imaginaria.
  - `modulus` y `argument` : sirven para especificar su módulo y su argumento.
- `as.complex` convierte un número real en complejo.
- Funciones específicas para números complejos:
  - Parte real: `Re`
  - Parte imaginaria: `Im`
  - Módulo: `Mod`
  - Argumento: `Arg`
  - Conjugado: `Conj`

## 2.7 Ejercicios

### Test

En los tests, tenéis que entrar las respuestas sin dejar ningún espacio en blanco excepto los que se pidan explícitamente. Cuando os pidan que deis una instrucción de R, *no* tenéis que incluir la marca de inicio `>`. Del mismo modo, cuando os pidan que copiéis un resultado dado por R, *no* tenéis que incluir el `[1]`.

(1) Dad una expresión para calcular  $(2 + 7)8 + \frac{5}{2} - 3^6 + 8!$ , con las operaciones escritas exactamente en el orden dado y sin paréntesis innecesarios, y a continuación, separado por un único espacio en blanco, copiad exactamente el resultado que ha dado R al evaluarla.

(2) Dad una expresión para calcular  $|\sin(\sqrt{2}) - e^{\sqrt[5]{2}}|$ , con las operaciones y funciones escritas exactamente en el orden dado, y a continuación, separado por un único espacio en blanco, copiad exactamente el resultado que ha dado R al evaluarla.

(3) Dad una expresión para calcular  $\sin(37^\circ)$ , empleando la construcción explicada en esta lección para calcular funciones trigonométricas de ángulos dados en grados, y a continuación, separado por un único espacio en blanco, copiad exactamente el resultado que ha dado R al evaluarla.

(4) Dad una expresión para calcular  $3e - \pi$ , con las operaciones escritas exactamente en la orden dado, y a continuación, separado por un único espacio en blanco, copiad exactamente el resultado que ha dado R al evaluarla.

(5) Dad una expresión para calcular  $e^{2/3}$  redondeado a 3 cifras decimales y a continuación, separado por un único espacio en blanco, copiad exactamente el resultado que ha dado R al evaluarla.

(6) En una sola línea, definid  $x$  como  $\sqrt{2}$  e  $y$  como  $\cos(3\pi)$  y calculad  $\ln(x^y)$ ; separad las tres instrucciones con puntos y comas seguidos de un único espacio en blanco. A continuación, separado por un espacio en blanco (sin punto y coma), copiad exactamente el resultado que ha dado R al evaluar esta secuencia de instrucciones.

(7) Corresponde el número en notación científica `3.3333e10` al número 33333000000? Tenéis que contestar SI (sin acento) o NO.

# Ejercicio

Si hubiéramos empezado a contar segundos a partir de las 12 campanadas que marcaron el inicio de 2015, ¿qué día de qué año llegaríamos a los 250 millones de segundos?  
¡Cuidado con los años bisiestos!

## Respuestas al test

(1) `(2+7)*8+5/2-3^6+factorial(8)` 39665.5

(2) `abs(sin(sqrt(2))-exp(2^(1/5)))` 2.166319

También sería correcto `abs(sin(2^(1/2))-exp(2^(1/5)))` 2.166319

(3) `sin(37*pi/180)` 0.601815

(4) `3*exp(1)-pi` 5.013253

(5) `round(exp(2/3),3)` 1.948

(6) `x=sqrt(2); y=cos(3*pi); log(x^y)` -0.3465736

(7) SI

# Lección 3 Un aperitivo: Introducción a la regresión lineal

En muchos libros de texto y artículos científicos encontraréis gráficos donde una línea recta o algún otro tipo de curva se ajusta a una serie de observaciones representadas por medio de puntos en el plano. La situación en general es la siguiente. Supongamos que tenemos una serie de puntos del plano cartesiano  $\mathbb{R}^2$ ,

$$(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n),$$

$$(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n),$$

que representan pares de observaciones de dos variables numéricas: por ejemplo,  $x =$  año e  $y =$  población, o  $x =$  longitud de una rama e  $y =$  número de hojas en la rama. Queremos describir cómo depende la variable **dependiente**  $y$  de la variable **independiente**  $x$  a partir de estas observaciones. Para ello, buscaremos una función  $y = f(x)$  cuya gráfica se aproxime lo máximo posible a los puntos  $(x_i, y_i)_{i=1, \dots, n}$ . Esta función nos dará un modelo matemático del comportamiento de las observaciones realizadas que nos permitirá entender mejor los mecanismos que relacionan las variables estudiadas o hacer predicciones sobre futuras observaciones.

Una primera opción, y la más sencilla, es estudiar si los puntos  $(x_i, y_i)_{i=1, \dots, n}$  satisfacen una relación lineal. En este caso, se busca la recta de ecuación  $y = b_1x + b_0$ , con  $b_0, b_1 \in \mathbb{R}$ , que aproxime mejor los puntos dados, en el sentido de que la suma de los cuadrados de las diferencias entre los valores  $y_i$  y sus aproximaciones  $b_1x_i + b_0$ ,

$$\sum_{i=1}^n (y_i - (b_1x_i + b_0))^2,$$

$$\sum_{i=1}^n (y_i - (b_1x_i + b_0))^2,$$

sea mínima. A esta recta  $y = b_1x + b_0$  se la llama **recta de regresión por mínimos cuadrados**; para abreviar, aquí la llamaremos simplemente **recta de regresión**, porque es la única que estudiaremos por ahora.

El objetivo de esta lección es ilustrar el uso de R mediante el cálculo de esta recta de regresión. Para ello, introduciremos algunas funciones de R que ya explicaremos con más detalle en otras lecciones. Utilizaremos también transformaciones logarítmicas para tratar casos en los que los puntos dados se aproximen mejor mediante una función potencial o exponencial.

### 3.1 Cálculo de rectas de regresión

Consideremos la Tabla 3.1, que da la altura media de los niños a determinadas edades. Los datos se han extraído de [http://www.cdc.gov/growthcharts/clinical\\_charts.htm](http://www.cdc.gov/growthcharts/clinical_charts.htm). Queremos determinar a partir de estos datos si hay una relación lineal entre la edad y la altura media de los niños.

Tabla 3.1: Alturas medias de niños por edad.

edad (años)	altura (cm)
1	76.11
2	86.45
3	95.27
5	109.18
7	122.03
9	133.73
11	143.73
13	156.41

Cuando tenemos una serie de observaciones emparejadas como las de esta tabla, la manera natural de almacenarlas en R es mediante una **tabla de datos**, un **data frame** en el argot de R. Aunque en este ejemplo concreto no sería necesario, lo haremos así para



que empecéis a acostumbraros. La ventaja de tener los datos organizados en forma de *data frame* es que con ellos luego se pueden hacer muchas más cosas. Estudiaremos en detalle los *data frames* en la Lección [??](#).

Para crear este *data frame*, en primer lugar guardaremos cada fila de la Tabla [3.1](#) como un **vector**, es decir, como una lista ordenada de números, y le pondremos un nombre adecuado. Para definir un vector, podemos aplicar la función `c` a la secuencia ordenada de números, separados por comas:

```
edad=c(1,2,3,5,7,9,11,13)
altura=c(76.11,86.45,95.27,109.18,122.03,133.73,143.73,156.41)
edad
```

```
## [1]  1  2  3  5  7  9 11 13
```

```
altura
```

```
## [1]  76.11  86.45  95.27 109.18 122.03 133.73 143.73 156.41
```

Ahora vamos a construir un *data frame* de dos columnas, una para la edad y otra para la altura, y lo llamaremos `datos1`. Estas columnas serán las **variables** de nuestra tabla de datos. Para organizar diversos vectores de la misma longitud en un *data frame*, podemos aplicar la función `data.frame` a los vectores:

```
datos1=data.frame(edad,altura)
datos1
```

```
##      edad altura
## 1      1    76.11
## 2      2    86.45
## 3      3    95.27
## 4      5   109.18
## 5      7   122.03
## 6      9   133.73
## 7     11   143.73
## 8     13   156.41
```

Observad que las filas del *data frame* resultante corresponden a los pares (edad, altura) de la Tabla 3.1.

Al analizar unos datos, siempre es conveniente empezar con una representación gráfica que nos permita hacernos una idea de sus características. En este caso, lo primero que haremos será dibujar los pares (edad, altura) usando la función `plot`. Esta función tiene muchos parámetros que permiten mejorar el resultado, pero ya los veremos al estudiarla en detalle en la Lección ???. Por ahora nos conformamos con un gráfico básico de estos puntos que nos muestre su distribución.

Dada una familia de puntos  $(x_n, y_n)_{n=1, \dots, k}$ , si llamamos `x` al vector  $(x_n)_{n=1, \dots, k}$  de sus abscisas e `y` al vector  $(y_n)_{n=1, \dots, k}$  de sus ordenadas, podemos obtener el gráfico de los puntos  $(x_n, y_n)_{n=1, \dots, k}$  mediante la instrucción

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

Si los vectores `x` e `y` son, en este orden, la primera y la segunda columna de un *data frame* de dos variables, como es nuestro caso, es suficiente aplicar la función `plot` al *data frame*. Así, por ejemplo, para dibujar el gráfico de la Figura 3.1 de los puntos  $(\text{edad}_n, \text{altura}_n)_{n=1, \dots, 8}$ , basta entrar la siguiente instrucción:

```
plot(datos1)
```

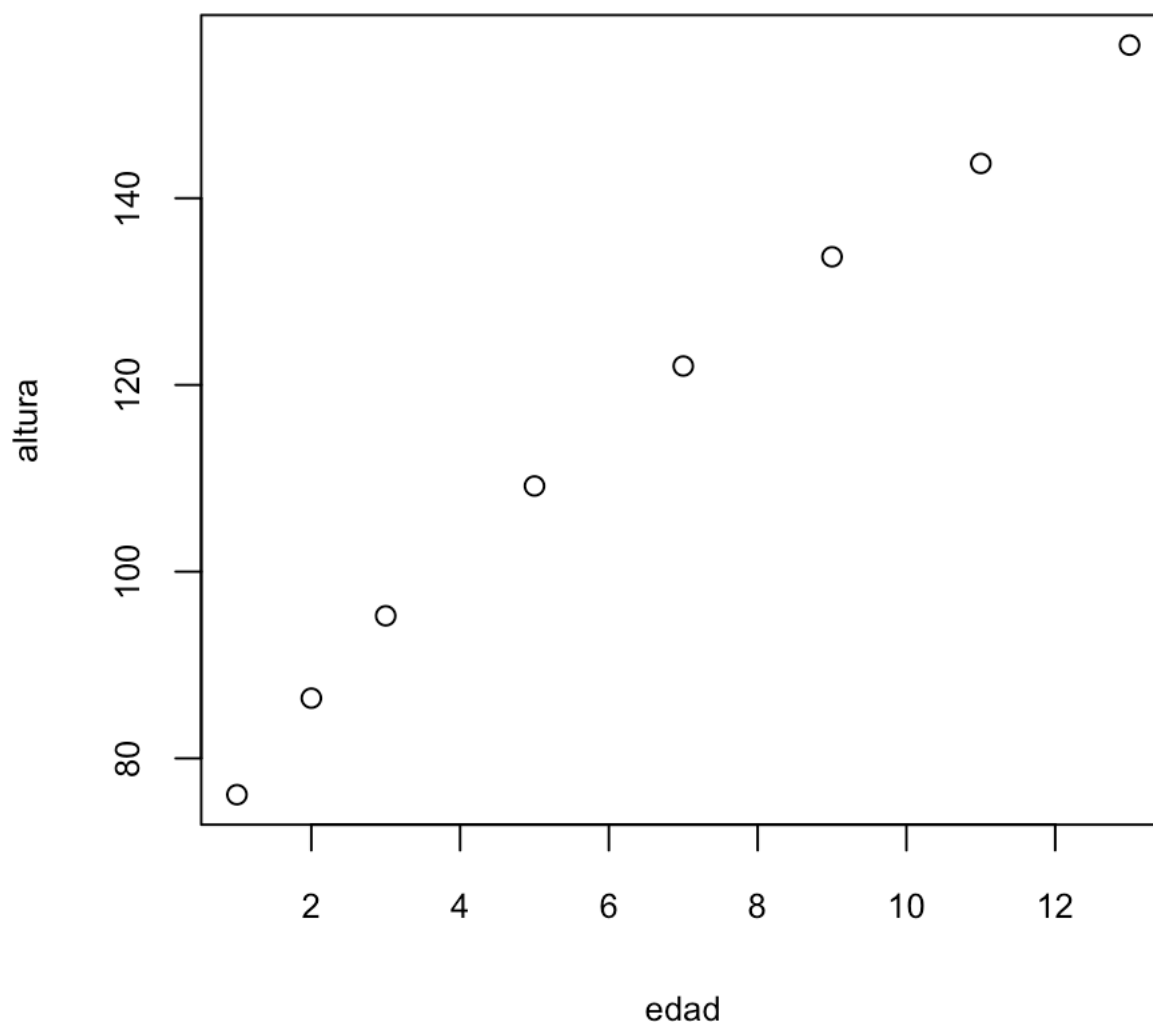


Figura 3.1: Representación gráfica de la altura media de los niños a determinadas edades.

Al ejecutar esta instrucción en la consola de `Rstudio`, el gráfico resultante se abrirá en la pestaña *Plots*, y en él se puede observar a simple vista que nuestros puntos siguen aproximadamente una recta. Vamos a calcular ahora su recta de regresión.

Dada una familia de puntos  $(x_n, y_n)_{n=1, \dots, k}$ , si llamamos `x` al vector  $(x_n)_{n=1, \dots, k}$  de sus abscisas e `y` al vector  $(y_n)_{n=1, \dots, k}$  de sus ordenadas, su recta de regresión se calcula con R por medio de la instrucción

```
lm(y~x)
```

Fijaos en la sintaxis: dentro del argumento de `lm`, primero va el vector `y`, seguido del vector `x` conectado a `y` por una tilde `~`. Para R, esta tilde significa **en función de**: es decir, `lm(y~x)` significa **la recta de regresión de  $y$  en función de  $x$** . Para obtener este signo, los usuarios de Windows y Linux tienen que pulsar Ctrl+Alt+4 seguido de un espacio en blanco y los de Mac OS X con teclado español pueden pulsar Alt+Ñ seguido de un espacio en blanco.

Si los vectores `y` y `x` son dos columnas de un *data frame*, para calcular la recta de regresión de  $yy$  en función de  $xx$  podemos usar la instrucción

```
lm(y~x, data=nombre del data frame)
```

Así pues, para calcular la recta de regresión de los puntos  $(\text{edad}_n, \text{altura}_n)_{n=1, \dots, 8}$   $(\text{edad}_n, \text{altura}_n)_{n=1, \dots, 8}$ , entramos la siguiente instrucción:

```
lm(altura~edad, data=datos1)
```

```
##  
## Call:  
## lm(formula = altura ~ edad, data = datos1)  
##  
## Coefficients:  
## (Intercept)      edad  
##      73.968      6.493
```

El resultado que hemos obtenido significa que la recta de regresión tiene término independiente 73.968 (el punto donde la recta *interseca* al eje de las  $yy$ ) y el coeficiente de  $xx$  es 6.493 (el coeficiente de la variable `edad`). Es decir, es la recta

$$y = 6.493x + 73.968.$$

$$y = 6.493x + 73.968.$$

Ahora la podemos superponer al gráfico anterior, empleando la función `abline`. Esta función permite añadir una recta al gráfico activo en la pestaña *Plots*. Por lo tanto, si no hemos cerrado el gráfico anterior, la instrucción

```
abline(lm(altura~edad, data=datos1))
```

le añade la recta de regresión, produciendo la Figura 3.2. Se ve a simple vista que, efectivamente, esta recta aproxima muy bien los datos.

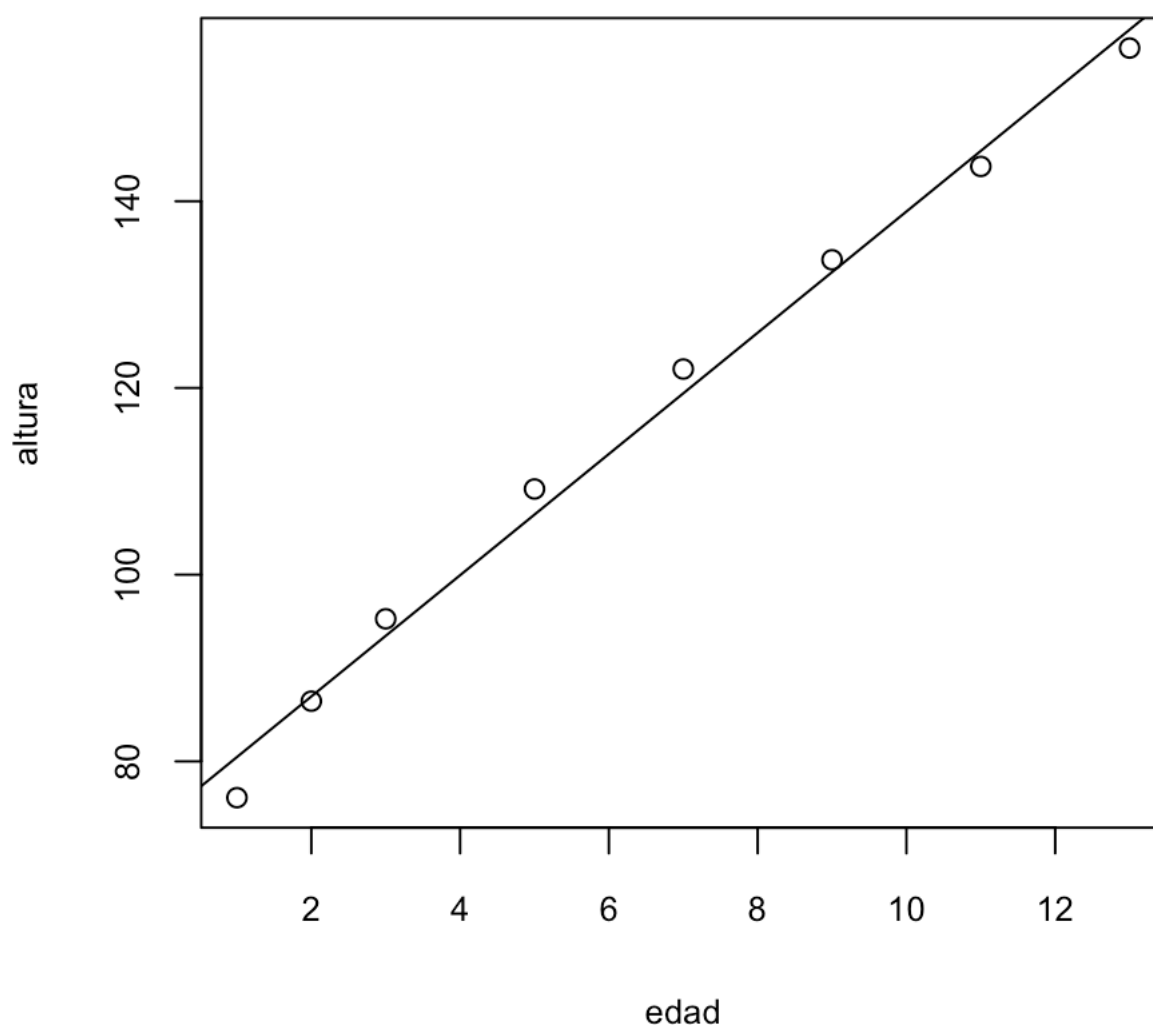


Figura 3.2: Ajuste mediante la recta de regresión de la altura media de los niños respecto de su edad.

Es importante tener presente que el análisis que hemos realizado de los pares de valores  $(\text{edad}_n, \text{altura}_n)_{n=1, \dots, 8}$  ha sido puramente descriptivo: hemos mostrado que estos datos son consistentes con una función lineal, pero *no hemos demostrado* que la altura media sea función aproximadamente lineal de la edad. Esto último requeriría una demostración matemática o un argumento biológico, no una simple comprobación numérica para una muestra pequeña de valores, que, al fin y al cabo, es lo único que hemos hecho.

Lo que sí que podemos hacer ahora es usar la relación lineal observada para predecir la altura media de los niños de otras edades. Por ejemplo, ¿qué altura media estimamos que tienen los niños de 10 años? Si aplicamos la regla

$$\text{altura} = 6.493 \cdot \text{edad} + 73.968,$$

$$\text{altura} = 6.493 \cdot \text{edad} + 73.968,$$

podemos predecir que la altura media a los 10 años es  $6.493 \cdot 10 + 73.968 = 138.898$ , es decir, de unos 139 cm.

Para evaluar numéricamente si la relación lineal que hemos encontrado es significativa o no, podemos usar el *coeficiente de determinación*  $R^2$ . No explicaremos aquí cómo se define, ya lo haremos en la Lección [??](#). Es suficiente saber que es un valor entre 0 y 1 y que cuanto más se aproxime la recta de regresión al conjunto de puntos, más cercano será a 1. Por el momento, y como regla general, si este coeficiente de determinación  $R^2$  es mayor que 0.9, consideraremos que el ajuste de los puntos a la recta es bueno.

Cuando R calcula la recta de regresión también obtiene este valor, pero no lo muestra si no se lo pedimos. Si queremos saber todo lo que ha calculado R con la función `lm`, tenemos que emplear la construcción `summary(lm(...))`. En general, la función `summary` aplicada a un objeto de R nos da un resumen de los contenidos de este objeto, resumen que depende de la clase de objeto que se trate.

Veamos cuál es el resultado de esta instrucción en nuestro ejemplo:

```
summary(lm(altura~edad, data=datos1))
```

```
##

## Call:
## lm(formula = altura ~ edad, data = datos1)
##

## Residuals:
##      Min       1Q   Median       3Q      Max
## -4.351 -1.743  0.408  2.018  2.745
##

## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept)  73.9681     1.7979   41.14 1.38e-08 ***
## edad         6.4934     0.2374   27.36 1.58e-07 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

##

## Residual standard error: 2.746 on 6 degrees of freedom
## Multiple R-squared:  0.992, Adjusted R-squared:  0.9907
## F-statistic: 748.4 on 1 and 6 DF, p-value: 1.577e-07
```

Por ahora podemos prescindir de casi toda esta información (en todo caso, observad que la columna `Estimate` nos da los coeficientes de la recta de regresión) y fijarnos sólo en el primer valor de la penúltima línea, `Multiple R-squared`. Éste es el coeficiente de determinación  $R^2$  que nos interesa. En este caso ha sido de 0.992, lo que confirma que la recta de regresión aproxima muy bien los datos.

Podemos pedir a R que nos dé el valor `Multiple R-squared` sin tener que obtener todo el `summary`, añadiendo el sufijo `$r.squared` a la construcción `summary(lm(...))`.

```
summary(lm(altura~edad, data=datos1))$r.squared
```

```
## [1] 0.9920466
```

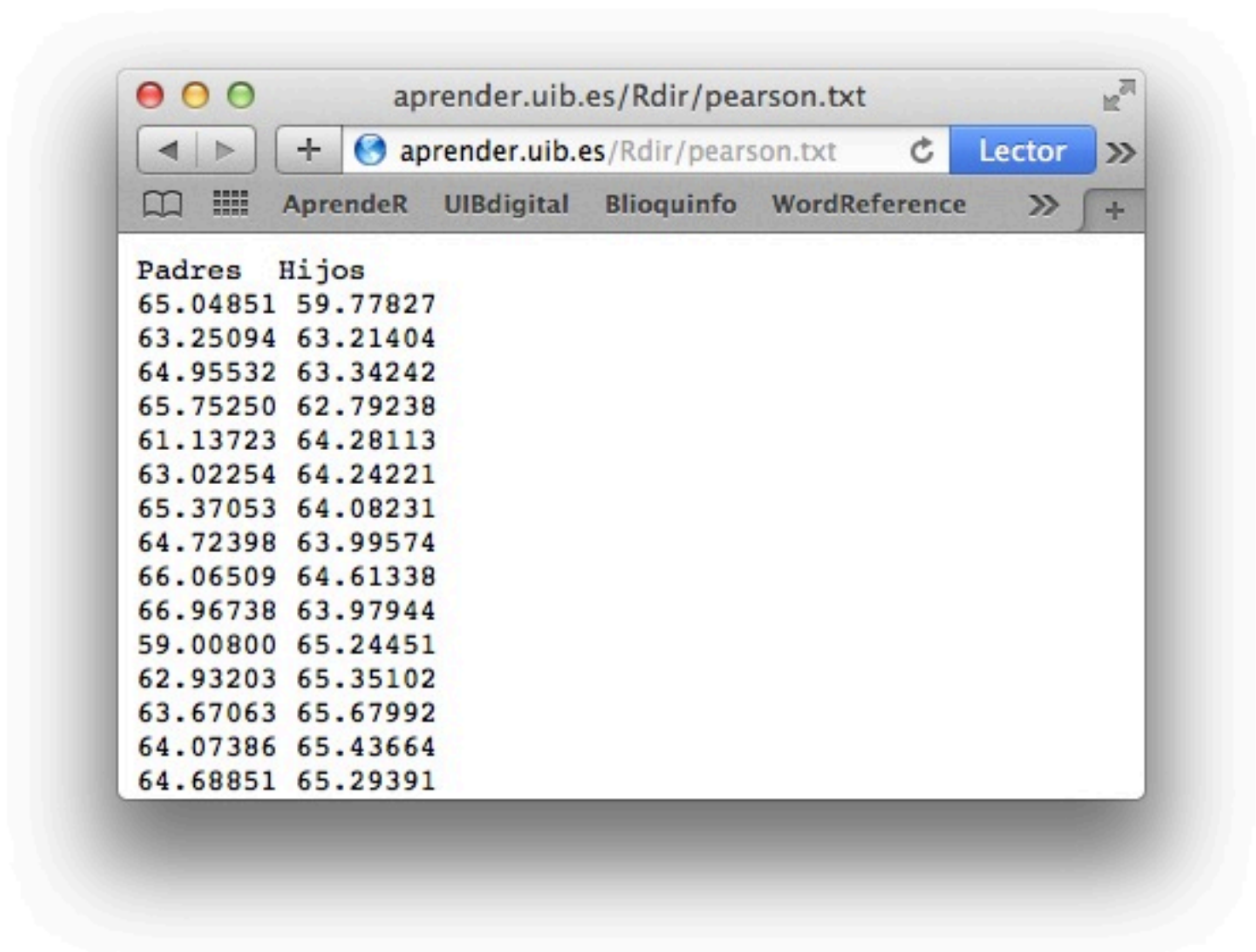
Los sufijos que empiezan con `$` suelen usarse en R para obtener componentes de un objeto. Por ejemplo, si al nombre de un *data frame* le añadimos el sufijo formado por `$` seguido del nombre de una de sus variables, obtenemos el contenido de esta variable.

```
datos1$edad
```

```
## [1] 1 2 3 5 7 9 11 13
```

Veamos otro ejemplo de cálculo de recta de regresión.

**Ejemplo 3.1** Karl Pearson recopiló en 1903 las alturas de 1078 parejas formadas por un padre y un hijo. Hemos guardado en el `url` <http://aprender.uib.es/Rdir/pearson.txt> un fichero que contiene estas alturas. Si lo abris en un navegador, veréis que es una tabla de dos columnas, etiquetadas `Padres` e `Hijos` (Figura 3.3). Cada fila contiene las alturas en pulgadas de un par Padre-Hijo.



Padres	Hijos
65.04851	59.77827
63.25094	63.21404
64.95532	63.34242
65.75250	62.79238
61.13723	64.28113
63.02254	64.24221
65.37053	64.08231
64.72398	63.99574
66.06509	64.61338
66.96738	63.97944
59.00800	65.24451
62.93203	65.35102
63.67063	65.67992
64.07386	65.43664
64.68851	65.29391

Figura 3.3: Vista en un navegador del fichero `pearson.txt`.



Vamos a usar estos datos para estudiar si hay dependencia lineal entre la altura de un hijo y la de su padre. Para ello, lo primero que haremos será cargarlos en un *data frame*. Esto se puede llevar a cabo de dos maneras:

- Usando el menú *Import Dataset* de la pestaña *Environment* de la ventana superior derecha de *RStudio*, sobre el que volveremos en la Lección [??](#). Al pulsar sobre este menú, se nos ofrece la posibilidad de importar un fichero de diferentes maneras; en este ejemplo, vamos a usar *From Text (readr)...*, que es la adecuada para importar tablas de Internet. Al seleccionarla, se nos pide el `url` del fichero y se nos dan a escoger una serie de opciones donde podemos especificar el nombre del *data frame* que queremos crear, si el fichero tiene o no una primera fila con los nombres de las columnas, cuál es el signo usado para separar columnas, etc. Pulsando el botón *Update* podremos ver en el campo *Data Preview* de esta ventana de diálogo el aspecto del *data frame* que obtendremos con las opciones seleccionadas; se trata entonces de escoger las opciones adecuadas para que se cree la versión correcta del *data frame*. En el caso concreto de esta tabla `pearson.txt`, se tiene que seleccionar la casilla de *First Row as Names* y escoger el valor *Whitespace* en *Delimiter* (Figura [3.4](#)). Al pulsar el botón *Import*, se importará el fichero en un *data frame* con el nombre especificado en el campo *Name* y se verá su contenido en la ventana de ficheros si se ha seleccionado la casilla *Open Data Viewer*.

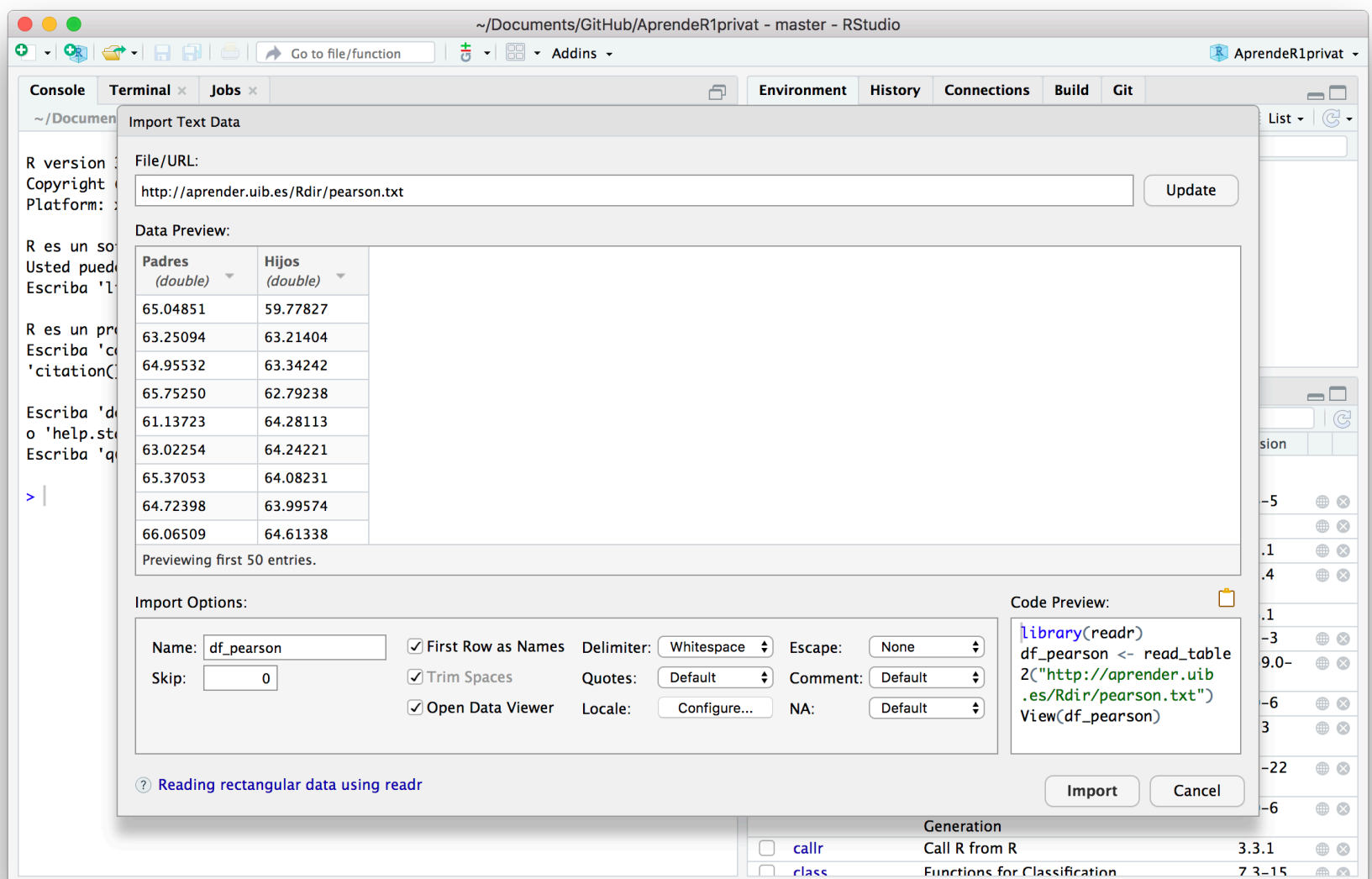


Figura 3.4: Opciones para guardar el fichero *pearson.txt* en un *data frame* llamado *df\_pearson* usando el menú *Import Dataset*.

- Usando la instrucción `read.table`, de la que también hablaremos en la Lección ??; por ahora simplemente hay que saber que se ha de aplicar al nombre del fichero entre comillas, si está en el directorio de trabajo, o a su `url`, también escrito entre comillas. Si además el fichero contiene una primera fila con los nombres de las columnas, hay que añadir el parámetro `header=TRUE`.

Así pues, para cargar esta tabla de datos concreta en un *data frame* llamado `df_pearson`, podemos usar el menú *Import Dataset*, o entrar la instrucción siguiente:

```
df_pearson=read.table("http://aprender.uib.es/Rdir/pearson.txt", header=TRUE)
```

En ambos casos, para comprobar que se ha cargado bien, podemos usar las funciones `str`, que muestra la estructura del *data frame*, y `head`, que muestra sus primeras filas.

```
str(df_pearson)
```

```
## 'data.frame':    1078 obs. of  2 variables:
##  $ Padres: num  65 63.3 65 65.8 61.1 ...
##  $ Hijos : num  59.8 63.2 63.3 62.8 64.3 ...
```

```
head(df_pearson)
```

```
##      Padres      Hijos
## 1 65.04851 59.77827
## 2 63.25094 63.21404
## 3 64.95532 63.34242
## 4 65.75250 62.79238
## 5 61.13723 64.28113
## 6 63.02254 64.24221
```

El resultado de `str(df_pearson)` nos dice que este *data frame* está formado por 1078 observaciones (filas) de dos variables (columnas) llamadas `Padres` e `Hijos`. El resultado de `head(df_pearson)` nos muestra sus primeras seis filas, que podemos comprobar que coinciden con las del fichero original mostrado en la Figura [3.3](#).

Calculemos la recta de regresión de las alturas de los hijos respecto de las de los padres: ahora las siguientes instrucciones:

```
lm(Hijos~Padres, data=df_pearson)
```

```
##
## Call:
## lm(formula = Hijos ~ Padres, data = df_pearson)
##
## Coefficients:
## (Intercept)      Padres
##    33.8866      0.5141
```

```
summary(lm(Hijos~Padres, data=df_pearson))$r.squared
```

```
## [1] 0.2513401
```

Obtenemos la recta de regresión

$$y = 33.8866 + 0.5141x,$$

$$y = 33.8866 + 0.5141x,$$

donde  $yy$  representa la altura de un hijo y  $xx$  la de su padre, y un coeficiente de determinación  $R^2 = 0.25$ , muy bajo. La regresión no es muy buena, como se puede observar en la Figura 3.5 que generamos con el código siguiente:

```
plot(df_pearson)
abline(lm(Hijos~Padres, data=df_pearson), col="red")
```

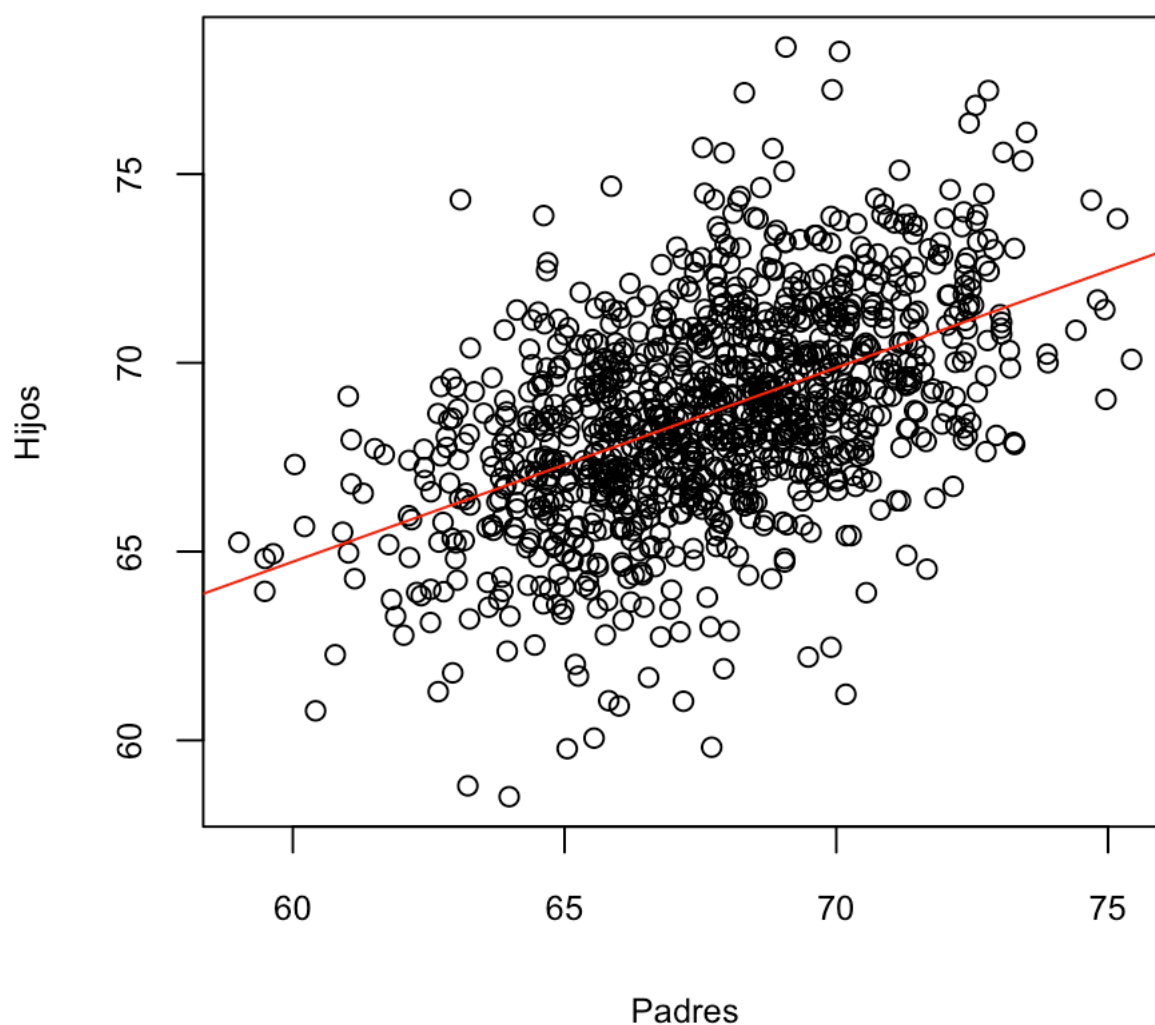


Figura 3.5: Representación gráfica de las alturas de los hijos en función de la de sus padres, junto con su recta de regresión.

Hemos usado el parámetro `col="red"` en el `abline` para que la recta de regresión sea roja y facilitar así su visualización en medio de la nube de puntos.

## 3.2 Rectas de regresión y transformaciones logarítmicas

La dependencia de un valor en función de otro no siempre es lineal. A veces podremos detectar otras dependencias (en concreto, exponenciales o potenciales) realizando un **cambio de escala** adecuado en el gráfico.

Cuando dibujamos un gráfico, lo normal es marcar cada eje de manera que la misma distancia entre marcas signifique la misma diferencia entre sus valores; por ejemplo, en el gráfico de la Figura 3.1, las marcas sobre cada uno de los ejes están igualmente espaciadas, de manera que entre cada par de marcas consecutivas en el eje de abscisas hay una diferencia de 2 años y entre cada par de marcas consecutivas en el eje de ordenadas hay una diferencia de 20 cm. Decimos entonces que los ejes están en **escala lineal**. Pero a veces es conveniente dibujar algún eje en **escala logarítmica**, situando las marcas de tal manera que la misma distancia entre marcas signifique el mismo *cociente* entre sus valores. Como el logaritmo transforma cocientes en restas, un eje en escala logarítmica representa el logaritmo de sus valores en escala lineal.

Decimos que un gráfico está en **escala semilogarítmica** cuando su eje de abscisas está en escala lineal y su eje de ordenadas en escala logarítmica. Salvo por los valores en las marcas sobre el eje de las  $y$ , esto significa que dibujamos en escala lineal el gráfico de  $\log(y)$  en función de  $x$ . Así pues, si al representar unos puntos  $(x, y)$  en escala semilogarítmica observamos que siguen aproximadamente una recta, esto querrá decir que los valores  $\log(y)$  siguen una ley aproximadamente lineal en los valores  $x$ , y, por lo tanto, que  $y$  sigue una ley aproximadamente exponencial en  $x$ . En efecto, si  $\log(y) = ax + b$ , entonces

$$y = 10^{\log(y)} = 10^{ax+b} = 10^{ax} \cdot 10^b = 10^b \cdot (10^a)^x = \beta \cdot \alpha^x,$$

$$y = 10^{\log(y)} = 10^{ax+b} = 10^{ax} \cdot 10^b = 10^b \cdot (10^a)^x = \beta \cdot \alpha^x,$$

donde  $\beta = 10^b$   $\beta = 10^b$  y  $\alpha = 10^a$   $\alpha = 10^a$ .

De manera similar, decimos que un gráfico está en **escala doble logarítmica** cuando ambos ejes están en escala logarítmica. Esto es equivalente, de nuevo salvo por los valores en las marcas sobre los ejes, a dibujar en escala lineal el gráfico de  $\log(y)\log(y)$  en función de  $\log(x)\log(x)$ . Por consiguiente, si al dibujar unos puntos  $(x, y)(x, y)$  en escala doble logarítmica observamos que siguen aproximadamente una recta, esto querrá decir que los valores  $\log(y)\log(y)$  siguen una ley aproximadamente lineal en los valores  $\log(x)\log(x)$ , y, por lo tanto, que  $yy$  sigue una ley aproximadamente potencial en  $xx$ . En efecto, si  $\log(y) = a \log(x) + b\log(y) = a\log(x) + b$ , entonces

$$y = 10^{\log(y)} = 10^{a \log(x) + b} = 10^{a \log(x)} \cdot 10^b = 10^b \cdot (10^{\log(x)})^a = 10^b \cdot x^a = \beta \cdot x^a,$$

$$y = 10^{\log (y)} = 10^{a \log (x) + b} = 10^{a \log (x)} \cdot 10^b = 10^b \cdot (10^{\log (x)})^a = 10^b \cdot x^a = \beta \cdot x^a,$$

donde  $\beta = 10^b$   $\beta = 10^b$ .

Veamos algunos ejemplos de regresiones lineales con cambios de escala.

**Ejemplo 3.2** La serotonina se asocia a la estabilidad emocional en el hombre. En un experimento (véase el artículo de B. Peskar y S. Spector “Serotonin: Radioimmunoassay” en *Science* 179 (1973), pp. 1340-1341) se midió, para algunas cantidades de serotonina (expresadas en *nanogramos*, la milmillonésima parte de un gramo), el porcentaje de inhibición de un cierto proceso bioquímico en el que se observaba su presencia. El objetivo era estimar la cantidad de serotonina presente en un tejido a partir del porcentaje de inhibición observado. Los datos que se obtuvieron son los de la Tabla 3.2.

Tabla 3.2: Porcentajes de inhibición de un cierto proceso bioquímico en presencia de serotonina.

serotonina (ng)	inhibición (%)
1.2	19
3.6	36
12.0	60
33.0	84

Como queremos predecir la cantidad de serotonina en función de la inhibición observada, consideraremos los pares (inhibición,serotonina). En esta ocasión, en vez de trabajar con un *data frame*, trabajaremos directamente con los vectores.

```
inh=c(19,36,60,84)
ser=c(1.2,3.6,12,33)
```

Con la instrucción siguiente obtenemos la Figura 3.6, donde vemos claramente que la cantidad de serotonina no es función lineal de la inhibición.

```
plot(inh,ser)
```

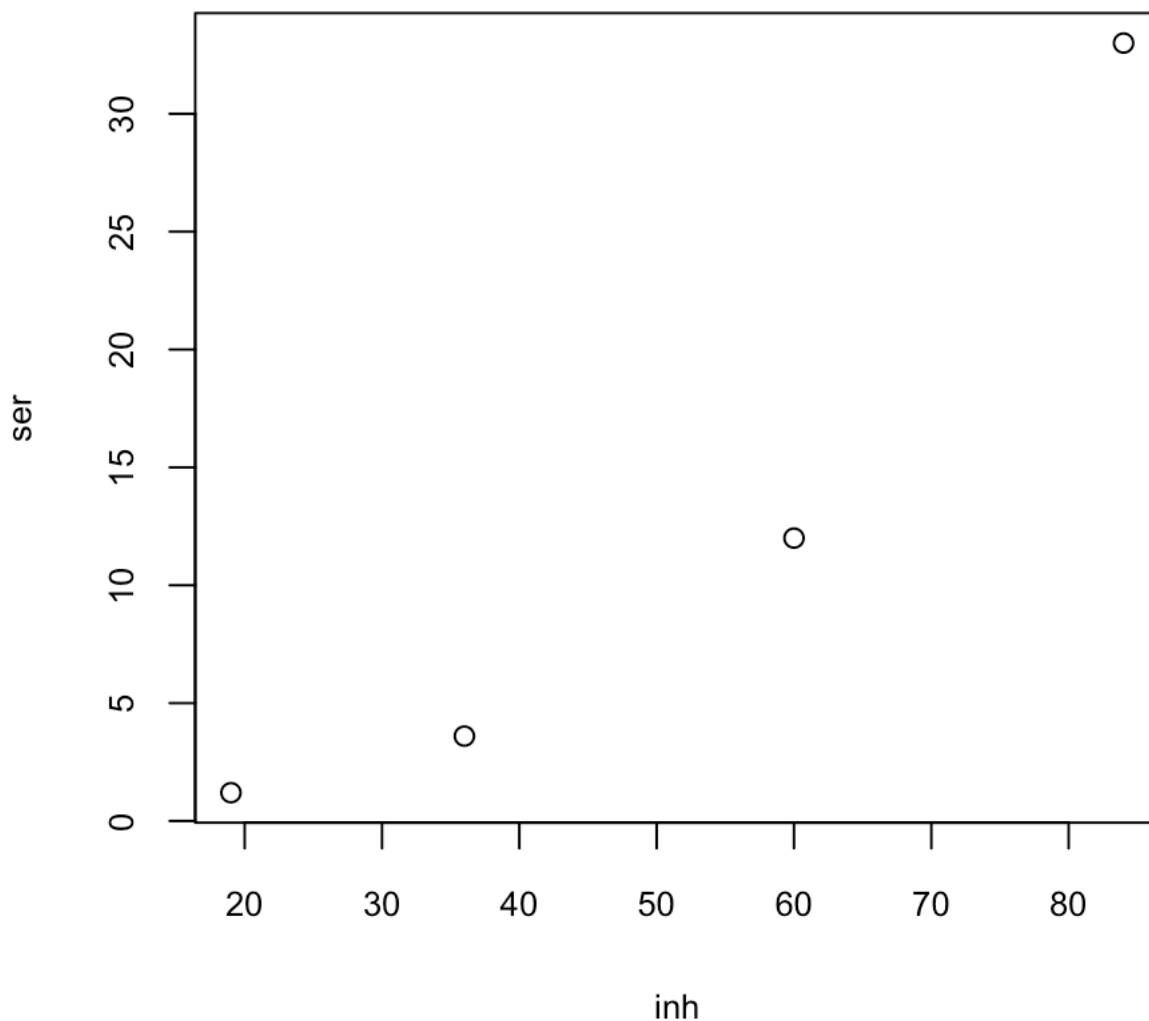


Figura 3.6: Representación gráfica en escala lineal del porcentaje de inhibición en función de la cantidad de serotonina.

Vamos a dibujar ahora el gráfico semilogarítmico de estos puntos, para ver si de esta manera quedan sobre una recta. Para ello, tenemos que añadir al argumento de `plot` el parámetro `log="y"`.

```
plot(inh, ser, log="y")
```

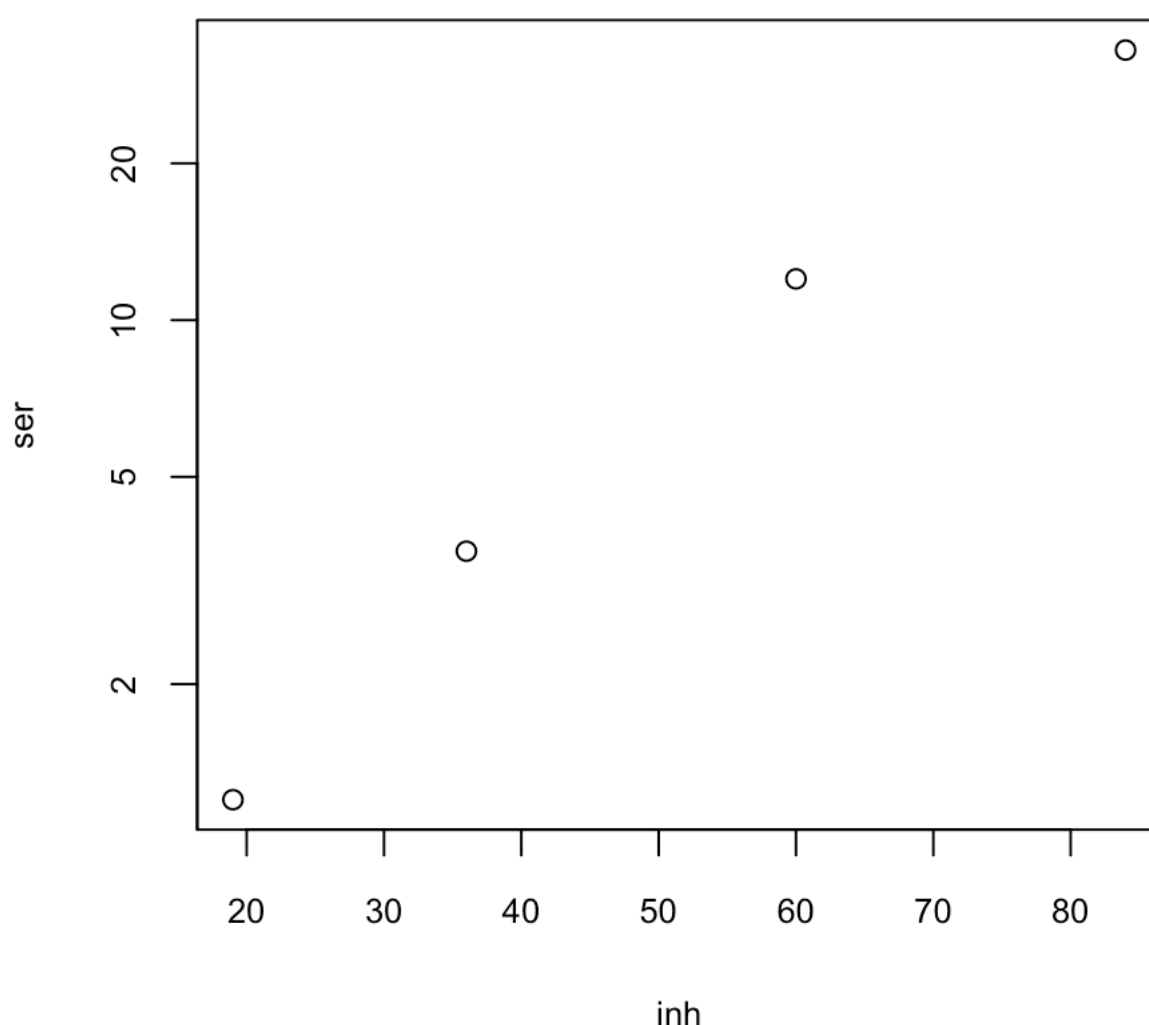


Figura 3.7: Representación gráfica en escala semilogarítmica del porcentaje de inhibición en función de la cantidad de serotonina.

Obtenemos la Figura 3.7. Observad cómo las marcas en el eje de ordenadas no están distribuidas de manera lineal: la distancia de 5 a 10 es la misma que de 10 a 20. Los puntos en este gráfico sí que parecen seguir una recta. Por lo tanto, parece que el logaritmo de la cantidad de serotonina es una función aproximadamente lineal del porcentaje de inhibición. Para confirmarlo, calcularemos la recta de regresión de los puntos

$$(\text{inhibición}_n, \log(\text{serotonina}_n))_{n=1, \dots, 4}.$$

$$(\text{inhibición}_n, \log(\text{serotonina}_n))_{n=1, \dots, 4}.$$

Para calcular los logaritmos en base 10 de todas las cantidades de serotonina en un solo paso, podemos aplicar la función `log10` directamente al vector `ser`.



```
log10(ser)
```

```
## [1] 0.07918125 0.55630250 1.07918125 1.51851394
```

```
lm(log10(ser)~inh)
```

```
##  
## Call:  
## lm(formula = log10(ser) ~ inh)  
##  
## Coefficients:  
## (Intercept)          inh  
##    -0.28427         0.02196
```

```
summary(lm(log10(ser)~inh))$r.squared
```

```
## [1] 0.9921146
```

El resultado indica que la recta de regresión de estos puntos es  $y = 0.02196x - 0.28427$ , con un valor de  $R^2$  de 0.992, muy bueno. Por lo tanto, podemos afirmar que, aproximadamente,

$$\log(\text{serotonina}) = 0.02196 \cdot \text{inhibición} - 0.28427.$$

Elevando 10 a cada uno de los lados de esta identidad, obtenemos

$$\begin{aligned}\text{serotonina} &= 10^{\log(\text{serotonina})} = 10^{-0.28427} \cdot 10^{0.02196 \cdot \text{inhibición}} \\ &= 0.52 \cdot 1.052^{\text{inhibición}}.\end{aligned}$$

Es decir, los puntos de partida siguen aproximadamente la función exponencial

$$y = 0.52 \cdot 1.052^x.$$

Vamos ahora a dibujar en un mismo gráfico los puntos ( $\text{inhibición}_n$ ,  $\text{serotonina}_n$ ) y esta función exponencial. Para añadir la gráfica de una función  $y = f(x)$  al gráfico activo en la pestaña *Plots* podemos emplear la función

```
curve(f(x), add=TRUE)
```

Así, el código siguiente produce la Figura 3.8; fijaos en cómo hemos especificado la función  $y = 0.52 \cdot 1.052^x$  dentro del `curve`.

```
plot(inh, ser)
curve(0.52*1.052^x, add=TRUE)
```

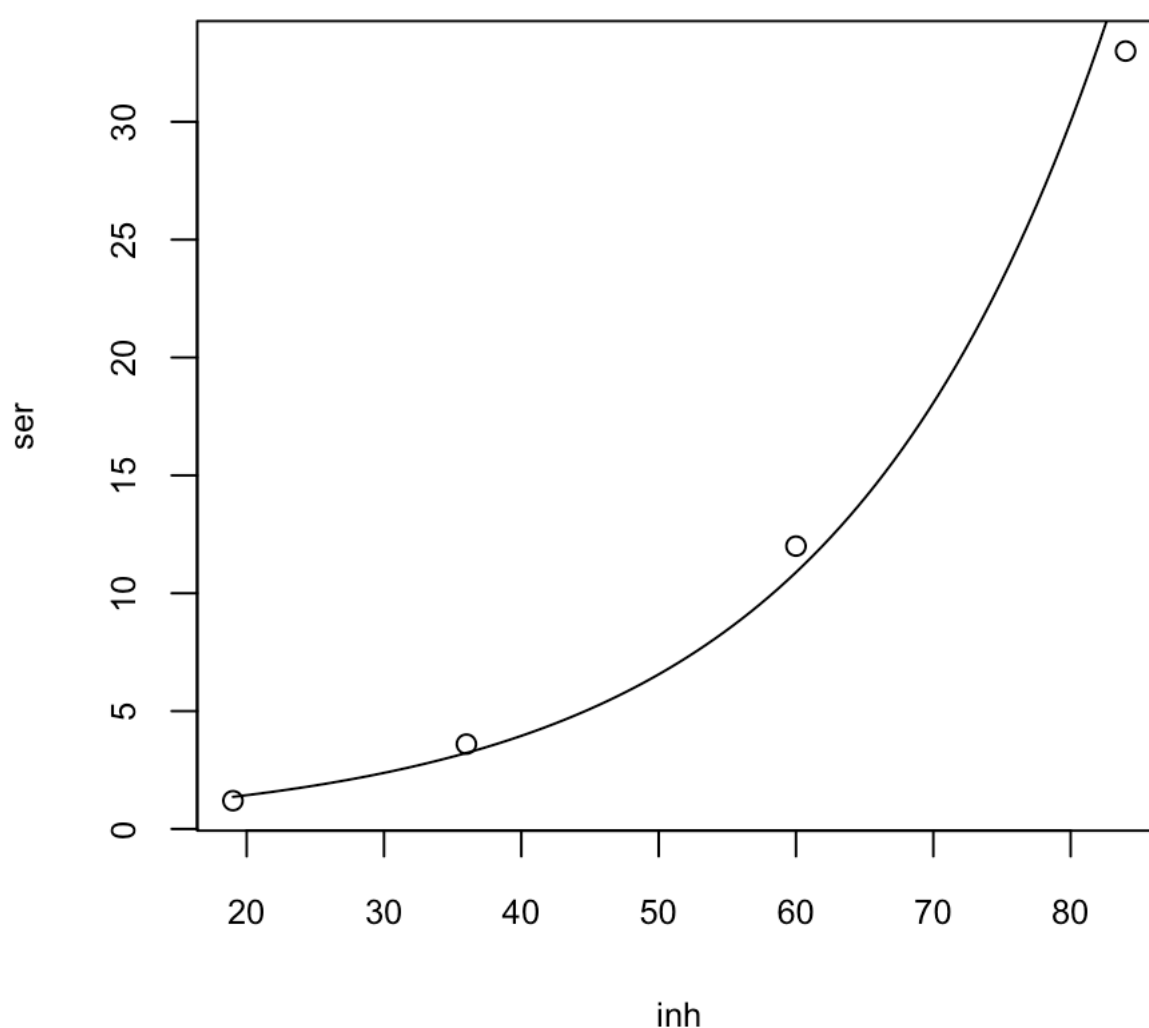


Figura 3.8: Representación gráfica en escala lineal del porcentaje de inhibición en función de la cantidad de serotonina, junto con la función  $y = 0.52 \cdot 1.052^x$ .

Ahora podemos usar la relación observada,

$$\text{serotonina} = 0.52 \cdot 1.052^{\text{inhibición}},$$

para estimar la cantidad de serotonina presente en el tejido a partir de una inhibición concreta. Por ejemplo, si hemos observado un 25% de inhibición, podemos estimar que la cantidad de serotonina es  $0.52 \cdot 1.052^{25} = 1.84$  ng

**Ejemplo 3.3** Consideremos ahora los datos de la Tabla 3.3. Se trata de los números acumulados de casos de SIDA en los Estados Unidos desde 1981 hasta 1992, extraídos del *HIV/AIDS Surveillance Report* de 1993 (<http://www.cdc.gov/hiv/topics/surveillance/resources/reports/index.htm>). *Acumulados* significa que, para cada año, se da el número de casos detectados *hasta* entonces.

Tabla 3.3: Números acumulados anuales de casos de SIDA en los Estados Unidos, 1981 a 1992.

año	casos
1981	97
1982	709
1983	2698
1984	6928
1985	15242
1986	29944
1987	52902
1988	83903
1989	120612
1990	161711
1991	206247
1992	257085

Queremos estudiar el comportamiento de estos números acumulados de casos en función del tiempo expresado en años a partir de 1980. Lo primero que hacemos es cargar los datos en un *data frame*. Fijaos en que la lista de años va a ser la secuencia de números

consecutivos entre 1 y 12. Para definir la secuencia de números consecutivos entre  $a$  y  $b$  podemos usar la construcción `a:b`. Esto nos ahorra trabajo y reduce las oportunidades de cometer errores al escribir los números.

```
tiempo=1:12  
SIDA_acum=c(97,709,2698,6928,15242,29944,52902,83903,120612,161711,206247,257085)  
df_SIDA=data.frame(tiempo, SIDA_acum)
```

Con la instrucción siguiente dibujamos estos datos:

```
plot(df_SIDA)
```

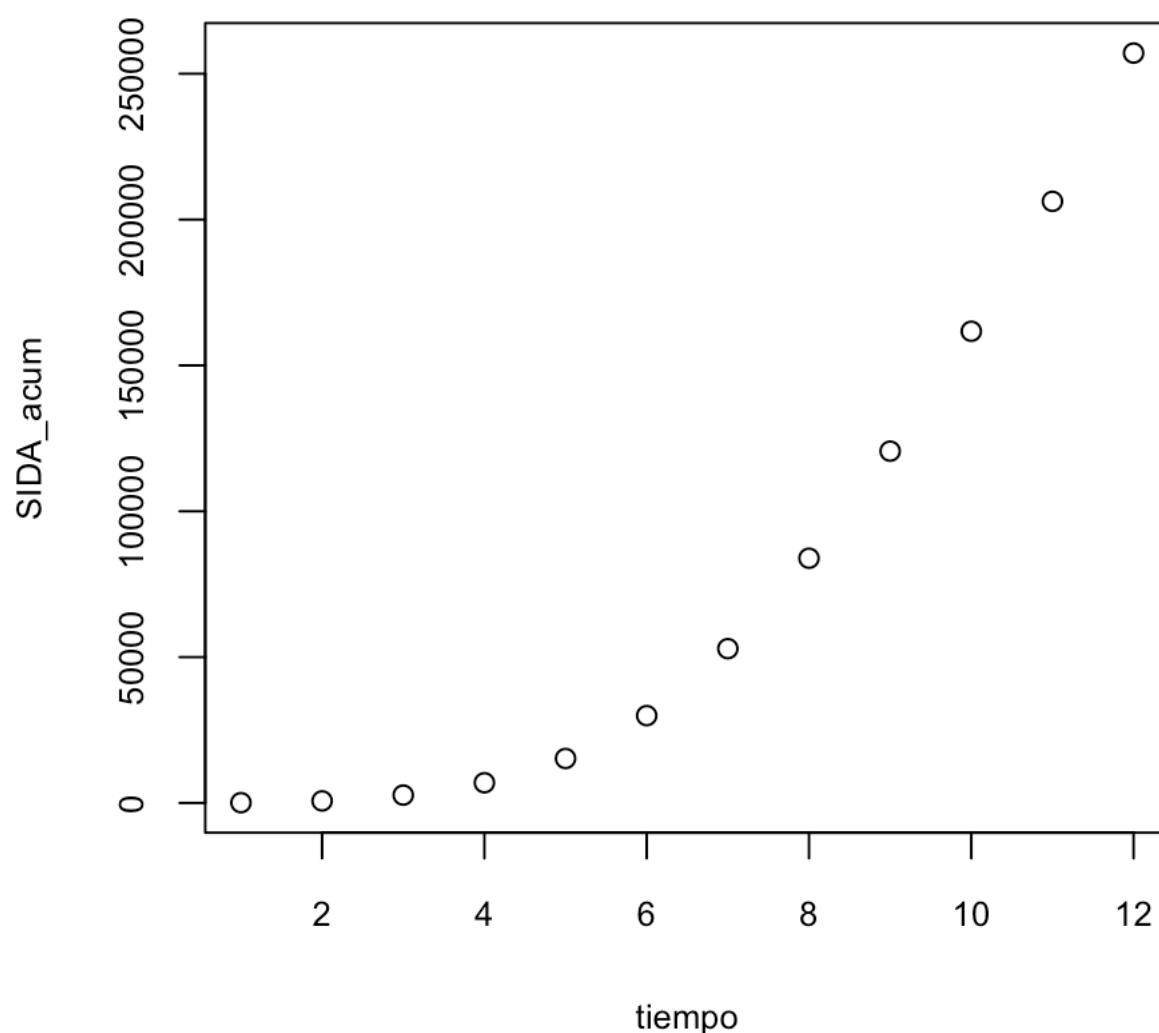


Figura 3.9: Representación gráfica en escala lineal del número acumulado de casos de SIDA en EEUU desde 1980 en función de los años transcurridos desde ese año.

Obtenemos el gráfico de la Figura 3.9, y está claro que los puntos  $(x_n, y_n)$ , donde  $x$  representa el año e  $y$  el número acumulado de casos de SIDA, no se ajustan a una recta. De hecho, a simple vista se diría que el crecimiento de  $y$  en función de  $x$  es exponencial.

Para confirmar este crecimiento exponencial, dibujamos el gráfico semilogarítmico:

```
plot(df_SIDA, log="y")
```

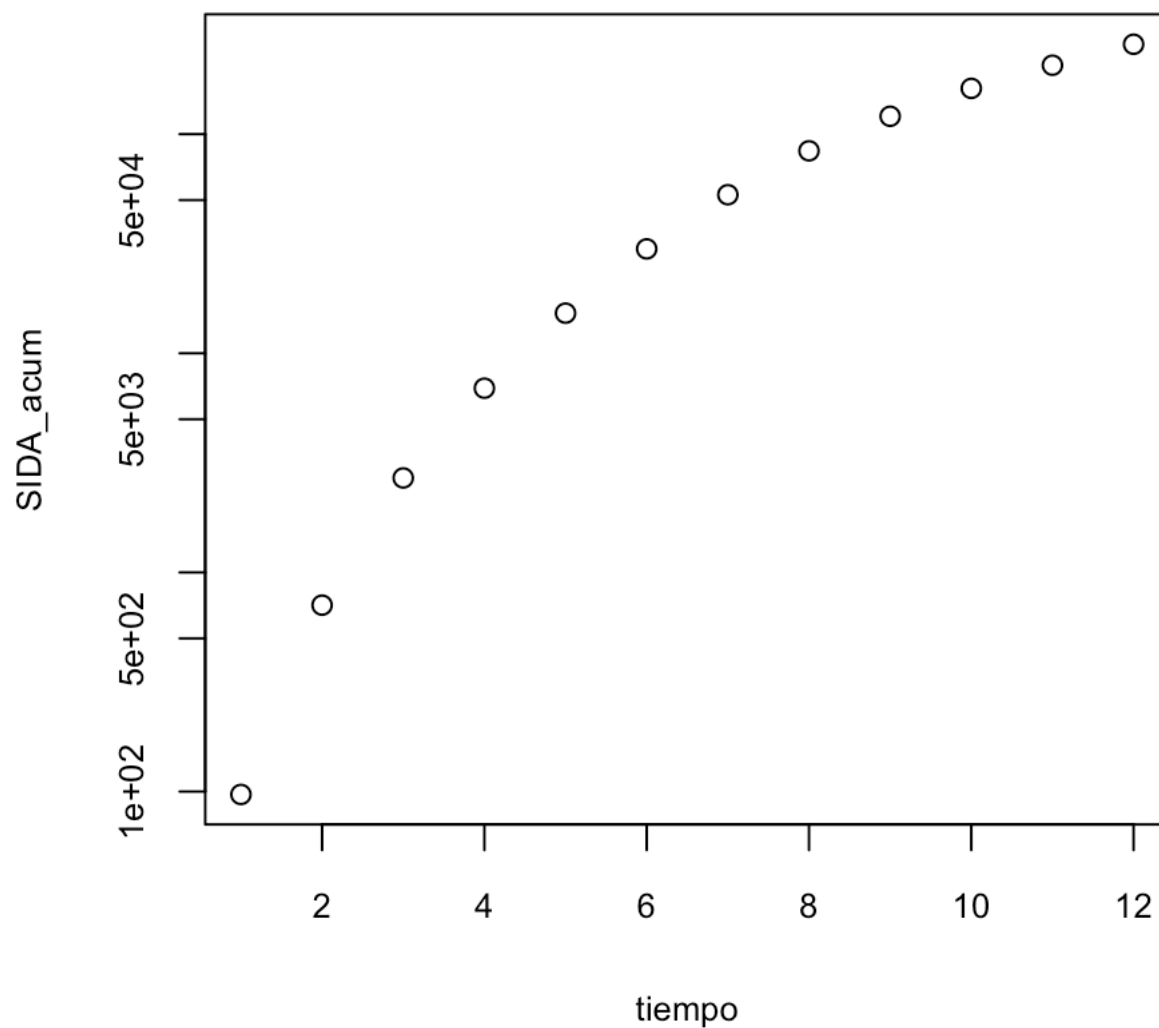


Figura 3.10: Representación gráfica en escala semilogarítmica del número acumulado de casos de SIDA en EEUU desde 1980 en función de los años transcurridos desde ese año.

Obtenemos el gráfico de la Figura 3.10, donde los puntos tampoco siguen una recta. Así pues, resulta que  $y$  tampoco parece ser función exponencial de  $x$ .

Vamos a ver si el crecimiento de  $y$  en función de  $x$  es potencial. Para ello, dibujaremos un gráfico doble logarítmico de los puntos  $(x_n, y_n)$ , especificando `log="xy"` dentro del argumento de `plot`.

```
plot(df_SIDA, log="xy")
```

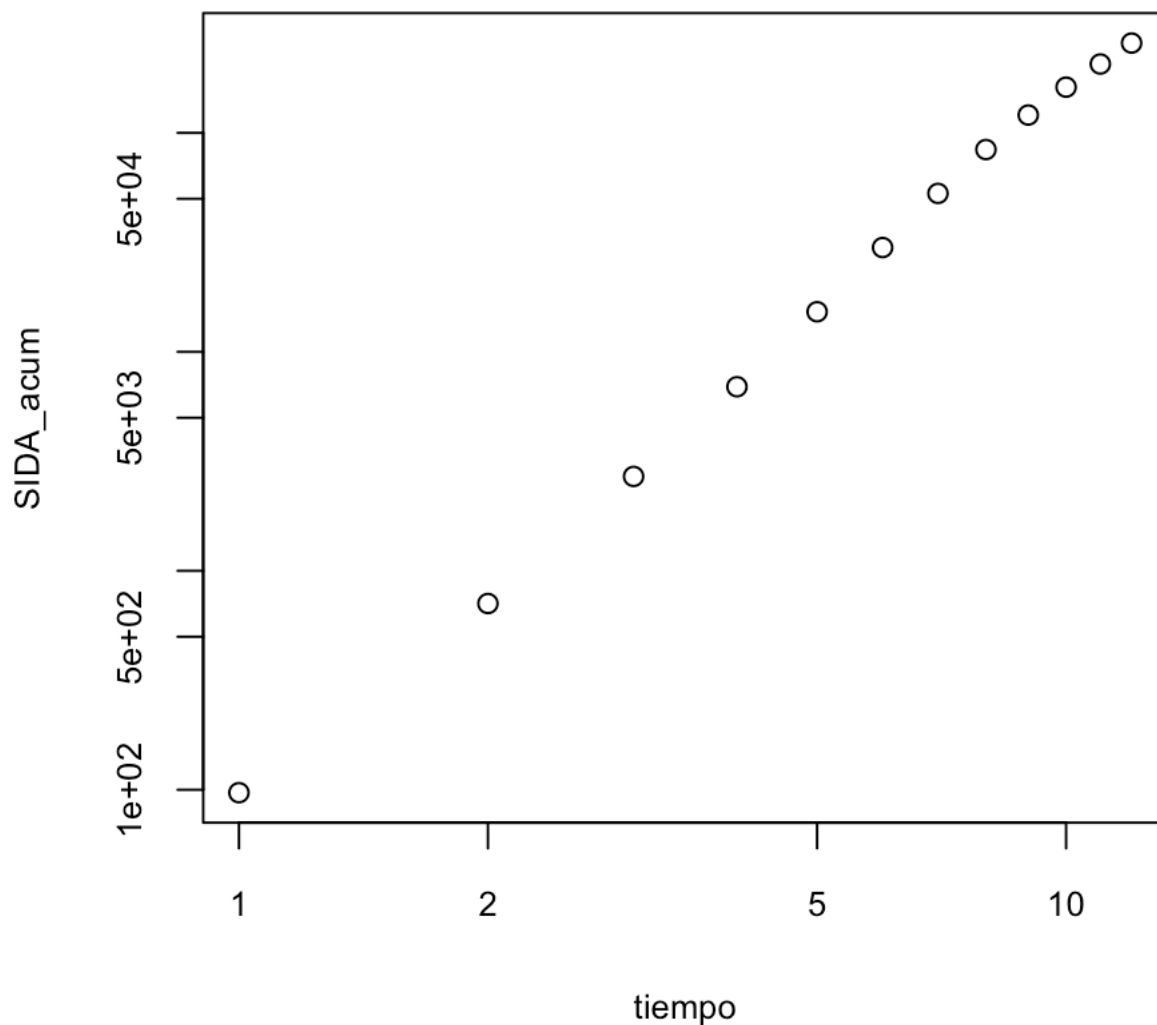


Figura 3.11: Representación gráfica en escala doble logarítmica del número acumulado de casos de SIDA en EEUU desde 1980 en función de los años transcurridos desde ese año.

Obtenemos el gráfico de la Figura 3.11, y ahora sí que parece lineal. Así que parece que los números acumulados de casos de SIDA crecieron potencialmente con el transcurso de los años.

Lo que haremos ahora será calcular la recta de regresión del logaritmo de `SIDA_acum` respecto del logaritmo de `tiempo` y mirar el coeficiente de determinación. Recordad que podemos aplicar una función a todas las entradas de un vector en un solo paso.

```
lm(log10(SIDA_acum)~log10(tiempo), data=df_SIDA)
```

```
##

## Call:
## lm(formula = log10(SIDA_acum) ~ log10(tiempo), data = df_SIDA)
##

## Coefficients:
##      (Intercept)      log10(tiempo)
##           1.918           3.274
```

```
summary(lm(log10(SIDA_acum)~log10(tiempo),data=df_SIDA))$r.squared
```

```
## [1] 0.9983866
```

La regresión que obtenemos es  $\log(y) = 1.918 + 3.274 \log(x)$ , con un valor de  $R^2$  de 0.998, muy alto. Elevando 10 a ambos lados de esta igualdad, obtenemos

$$\begin{aligned} y = 10^{\log(y)} &= 10^{1.918} \cdot 10^{3.274 \log(x)} = 10^{1.918} \cdot (10^{\log(x)})^{3.274} \\ &= 82.79422 \cdot x^{3.274}. \end{aligned}$$

Para ver si los puntos  $(\text{tiempo}_n, \text{SIDA\_acum}_n)_{n=1, \dots, 12}$  se ajustan bien a la curva

$$y = 82.79422 \cdot x^{3.274},$$

dibujaremos los puntos y la curva en un único gráfico (en escala lineal):

```
plot(df_SIDA)
curve(82.79422*x^3.274, add=TRUE)
```

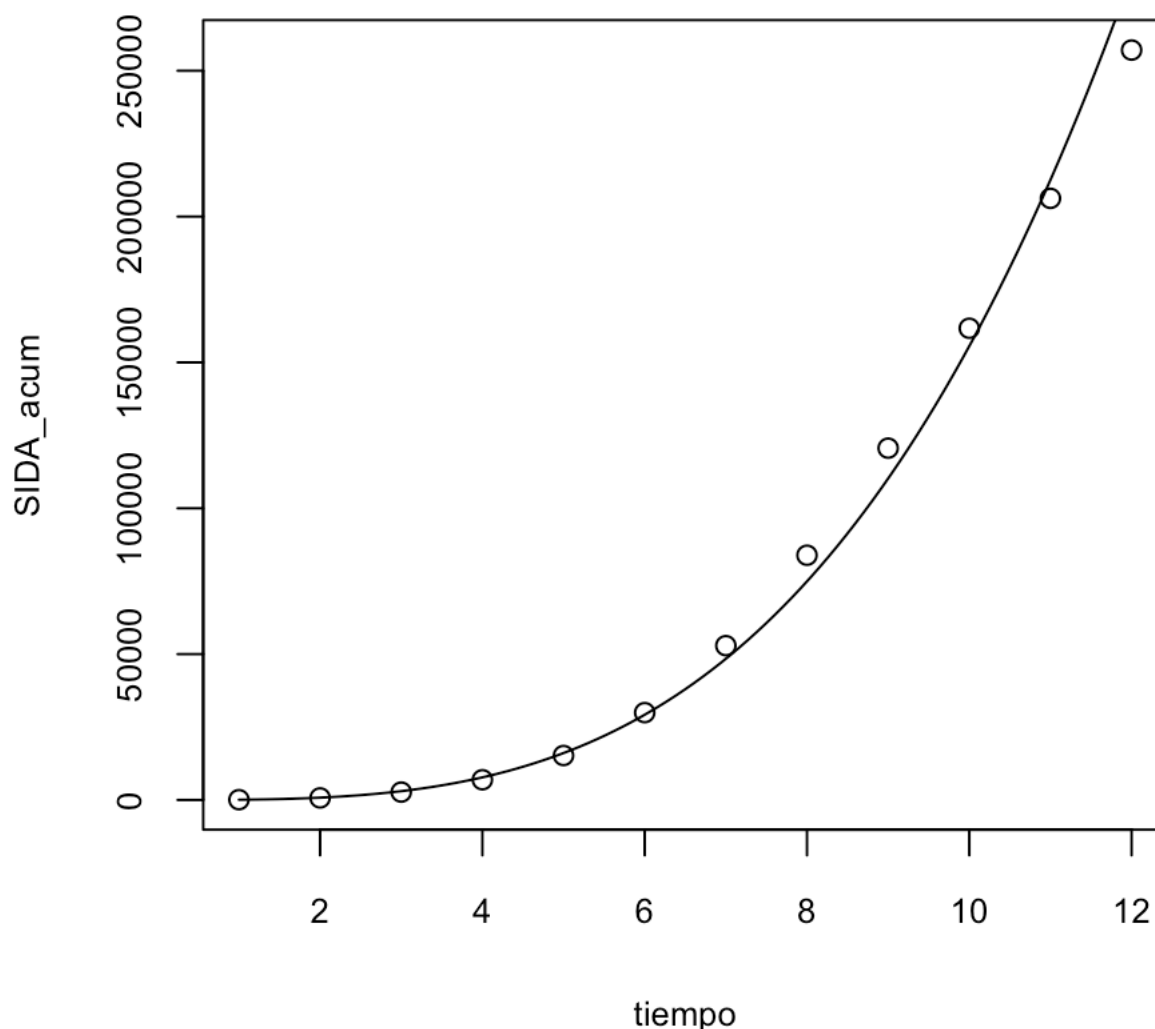


Figura 3.12: Representación gráfica en escala lineal de la cantidad acumulada de enfermos de SIDA en EEUU desde 1980 en función de los años transcurridos desde ese año, junto con su ajuste mediante la función potencial  $82.79422 \cdot x^{3.274}$ .

Obtenemos la Figura 3.12, donde vemos que la curva se ajusta bastante bien a los puntos.

Hay que mencionar aquí que se han propuesto modelos matemáticos que predicen que, cuando se inicia una epidemia de SIDA en una población, los números acumulados de casos en los primeros años son proporcionales al cubo del tiempo transcurrido desde el inicio; véase, por ejemplo, el artículo de S.A. Colgate, E. A. Stanley, J. M. Hyman, S. P. Layne y C. Qualls “Risk behavior-based model of the cubic growth of acquired immunodeficiency syndrome in the United States”, en *PNAS* 86 (1989), pp. 4793-4797. El resultado del análisis que hemos realizado es consistente con esta predicción teórica.

### 3.3 Guía rápida

- `c` sirve para definir vectores.
- `a:b`, con  $a < b$ , define un vector con la secuencia  $a, a+1, a+2, \dots, b$ .



- `data.frame` , aplicada a unos vectores de la misma longitud, define un *data frame* (el tipo de objetos de R en los que guardamos usualmente las tablas de datos) cuyas columnas serán estos vectores.
- `read.table` define un *data frame* a partir de un fichero externo. También se puede usar el menú *Import Dataset* de la pestaña *Environment* en la ventana superior derecha de *RStudio*.
- `lm(y~x)` calcula la recta de regresión del vector  $y$  respecto del vector  $x$ . Si  $x$  e  $y$  son dos columnas de un *data frame*, éste se ha de especificar en el argumento mediante el parámetro `data` igualado al nombre del *data frame*.
- `summary` sirve para obtener un resumen estadístico de un objeto. Este resumen depende del objeto. En el caso de una recta de regresión calculada con `lm` , muestra una serie de información estadística extra obtenida en dicho cálculo.
- `plot(x,y)` produce el gráfico de los puntos  $(x_n, y_n)$ . Si  $x$  e  $y$  son, respectivamente, la primera y la segunda columna de un *data frame* de dos columnas, se le puede entrar directamente el nombre del *data frame* como argumento. El parámetro `log` sirve para indicar los ejes que se desea que estén en escala logarítmica: `"x"` (abscisas), `"y"` (ordenadas) o `"xy"` (ambos).
- `abline` añade una recta al gráfico activo.
- `curve(función, add=TRUE)` añade la gráfica de la `función` al gráfico activo.

## 3.4 Ejercicios

### Ejercicio

Las larvas de *Lymantria dispar*, conocidas como *orugas peludas del alcornoque*, son una plaga en bosques y huertos. En un experimento se quiso determinar la capacidad de atracción de una cierta feromona sobre los machos de esta especie, con el objetivo de emplearla en trampas (véase el artículo de M. Beroza y E. F. Knipling “Gypsy moth control with the sex attractant pheromone” en *Science* 177 (1972), pp. 19-27). En la Tabla 3.4,  $x$  representa la cantidad de feromona empleada, en microgramos (la millonésima parte de un gramo) y  $N$  el número de machos atrapados en una trampa empleando esta cantidad de feromona para atraerlos.

Tabla 3.4: Cantidades de feromona empleadas en trampas y números de machos atrapados.

$x$	$N$
0.1	3
1.0	6
5.0	9
10.0	11
100.0	20

1. Decidid si, en los puntos  $(x, N)$  dados en la Tabla 3.4, el valor de  $N$  sigue una función aproximadamente lineal, exponencial o potencial en el valor de  $x$ .
2. En caso de ser una función de uno de estos tres tipos, calculadla.
3. Representad en un gráfico los puntos  $(x, N)$  de la Tabla 3.4 y la función que hayáis calculado en el apartado anterior, para visualizar la bondad del ajuste de la curva a los puntos.
4. Estimad cuánta feromona tenemos que usar en una trampa para atraer a 50 machos.

# Lección 4 Vectores y otros tipos de listas

Un **vector** es una secuencia ordenada de datos. R dispone de muchos tipos de datos, entre los que destacamos:

- `logical` (lógicos: `TRUE` , verdadero, o `FALSE` , falso)
- `integer` (números enteros)
- `numeric` o `double` (números reales)
- `complex` (números complejos)
- `character` (palabras)

Una restricción fundamental de los vectores en R es que todos sus objetos han de ser del mismo tipo: todos números, todos palabras, etc. Cuando queramos usar vectores formados por objetos de diferentes tipos, tendremos que usar **listas heterogéneas** (véase la Sección [4.5](#)).

## 4.1 Construcción de vectores

Para definir un vector con unos elementos dados, por ejemplo

1, 5, 6, 2, 5, 7, 8, 3, 5, 2, 1, 0

podemos aplicar la función `c` a estos elementos separados por comas.

```
x=c(1,5,6,2,5,7,8,3,5,2,1,0)
```

```
x
```

```
## [1] 1 5 6 2 5 7 8 3 5 2 1 0
```

Si queremos crear un vector de palabras con la instrucción `c` , tenemos que entrarlas obligatoriamente entre comillas. R también nos las muestra entre comillas.

```
nombres=c("Pep","Catalina","Joan","Pau")
```

```
nombres
```

```
## [1] "Pep"      "Catalina" "Joan"      "Pau"
```

Si nos olvidamos de las comillas:

```
nombres=c(Pep,Catalina,Joan,Pau)
```

```
## Error in eval(expr, envir, enclos): object 'Pep' not found
```

Hemos mencionado que todos los elementos de un vector han de ser del mismo tipo. Por este motivo, si juntamos datos de diferentes tipos en un vector, R automáticamente los convertirá a un tipo que pueda ser común a todos ellos. El orden de conversión entre los tipos que hemos explicado al principio de la lección es: `character` gana a `complex`, que gana a `numeric`, que gana a `integer`, que gana a `logical`. Así, cuando alguna entrada de un vector es de tipo palabra, R considera el resto de sus entradas como palabras (y las muestra entre comillas), como se puede ver en el siguiente ejemplo:

```
c(2,3.5,TRUE,"casa")
```

```
## [1] "2"      "3.5"    "TRUE"   "casa"
```

Otra posibilidad para crear un vector es usar la función `scan`. Si ejecutamos la instrucción `scan()` (así, con el argumento vacío), R abre en la consola un entorno de diálogo donde podemos ir entrando datos separados por espacios en blanco; cada vez que pulsemos la tecla *Entrar*, R importará los datos que hayamos escrito desde la vez anterior en que la pulsamos y abrirá una nueva línea donde esperará más datos; cuando hayamos acabado, dejamos la última línea en blanco (pulsando por última vez la tecla *Entrar*) y R cerrará el vector.

Por ejemplo, para crear un vector `x_scan` que contenga dos copias de

1, 5, 6, 2, 5, 7, 8, 3, 5, 2, 1, 0

una posibilidad sería primero entrar `scan()` , a continuación copiar esta secuencia con el editor de textos y pegarla dos veces en la última línea de la consola, pulsando *Entrar* después de cada vez, y finalmente pulsar *Entrar* por tercera vez en la última línea en blanco. Probadlo vosotros.

```
x_scan=scan()  #Y pulsamos Entrar
1: 1 5 6 2 5 7 8 3 5 2 1 0
13: 1 5 6 2 5 7 8 3 5 2 1 0
25:
Read 24 items
```

```
x_scan
```

```
## [1] 1 5 6 2 5 7 8 3 5 2 1 0 1 5 6 2 5 7 8 3 5 2 1 0
```

La función `scan` también se puede usar para copiar en un vector el contenido de un fichero de texto situado en el directorio de trabajo, o del que conozcamos su dirección en Internet. La manera de hacerlo es aplicando `scan` al nombre del fichero o a su *url*, entrados en ambos casos entre comillas. Por ejemplo, para definir un vector llamado `notas` con las notas de un examen que tenemos guardadas en el fichero <http://aprender.uib.es/Rdir/notas.txt> , sólo tenemos que entrar:

```
notas=scan("http://aprender.uib.es/Rdir/notas.txt")
notas
```

```
## [1] 4.1 7.8 5.8 6.5 4.8 6.9 1.3 6.4 4.6 6.9 9.4 3.0 6.8 4.8
## [15] 5.6 7.7 10.0 4.4 1.7 8.0 6.3 3.0 7.5 3.8 7.2 5.7 7.3 6.0
## [29] 5.7 4.7 5.1 1.5 7.0 7.0 6.0 6.6 7.2 5.0 3.5 3.3 4.7 5.4
## [43] 7.1 8.2 6.7 0.1 5.1 6.8 6.9 8.8 4.5 6.6 2.0 3.0 6.7 7.9
## [57] 7.7 6.4 3.0 5.3 5.1 5.3 5.1 5.4 3.0
```

Si primero descargamos este fichero, sin cambiarle el nombre, en el directorio de trabajo de R, para definir el vector anterior bastará entrar:

```
notas2=scan("notas.txt")
notas2
```

```
## [1] 4.1 7.8 5.8 6.5 4.8 6.9 1.3 6.4 4.6 6.9 9.4 3.0 6.8 4.8
## [15] 5.6 7.7 10.0 4.4 1.7 8.0 6.3 3.0 7.5 3.8 7.2 5.7 7.3 6.0
## [29] 5.7 4.7 5.1 1.5 7.0 7.0 6.0 6.6 7.2 5.0 3.5 3.3 4.7 5.4
## [43] 7.1 8.2 6.7 0.1 5.1 6.8 6.9 8.8 4.5 6.6 2.0 3.0 6.7 7.9
## [57] 7.7 6.4 3.0 5.3 5.1 5.3 5.1 5.4 3.0
```

Si usamos el menú *Import Dataset* de la pestaña *Environment* para importar un vector contenido en un fichero externo como explicamos en el Ejemplo 3.1, obtendremos en realidad un *data frame* de una sola columna, llamada `v1` . Para construir un vector con esta columna, podemos usar luego la instrucción

```
nombre_del_vector=nombre_del_dataframe$V1
```

Véase la Lección ?? para más detalles.

La función `scan` dispone de muchos parámetros, que podéis consultar en su Ayuda. Estos parámetros se entran entre los paréntesis de `scan()` . Los más útiles en este momento son los siguientes:

- `sep` : sirve para indicar el signo usado para separar entradas consecutivas si no son espacios en blanco. Para ello se ha de igualar `sep` a este signo, entrecomillado. Por ejemplo, si vamos a entrar las entradas separadas por comas (o si están así en el fichero que vamos a importar), tenemos que especificar `sep=","` .

```
x_scan2=scan()
1: 1,2,3,4
1:
```

```
## Error in scan() : scan() expected 'a real', got '1,2,3,4'
```

```
x_scan2=scan(sep=",")
```

```
1: 1,2,3,4
```

```
5:
```

```
Read 4 items
```

```
x_scan2
```

```
## [1] 1 2 3 4
```

- `dec` : sirve para indicar el separador decimal cuando no es un punto. Para ello hemos de igualar `dec` al separador decimal entre comillas. Por ejemplo, si queremos crear con `scan` un vector formado por los dos números reales 4,5 y 6,2 escritos exactamente de esta manera, tenemos que especificar `dec=","` .

```
x_scan3=scan()
```

```
1: 4,5 6,2
```

```
## Error in scan() : scan() expected 'a real', got '4,5'
```

```
x_scan3=scan(dec=",")
```

```
1: 4,5 6,2
```

```
3:
```

```
Read 2 items
```

```
x_scan3
```

```
## [1] 4.5 6.2
```

- `what` : sirve para indicar a R de qué tipo tiene que considerar los datos que se le entren. En particular, `what="character"` especifica que los valores que se le entran son palabras, aunque no estén entre comillas (si se entran entre comillas, no hace falta especificarlo).

```
x_scan4=scan()
```

```
1:  Pep Catalina Joan Pau
```

```
## Error in scan() : scan() expected 'a real', got 'Pep'
```

```
x_scan4=scan(what="character")
```

```
1: Pep Catalina Joan Pau
```

```
5:
```

```
Read 4 items
```

```
x_scan4
```

```
## [1] "Pep"      "Catalina" "Joan"      "Pau"
```

- `encoding` : sirve para indicar la codificación de alfabeto del fichero externo que se va a importar. Sólo es necesario especificarlo si dicho fichero contiene caracteres que no sean de 7 bits; o sea, letras acentuadas o caracteres especiales. En este caso, si su codificación no es la que espera nuestro ordenador y no la especificamos con este parámetro, estos caracteres se importarán mal. Sus dos posibles valores son `"latin1"` y `"UTF-8"` . Por ejemplo, si sois usuarios de Windows, seguramente vuestro ordenador espere que el fichero a importar esté codificado en `latin1` ; entonces, si está codificado en `utf8` y contiene letras acentuadas, no las entenderá a no ser que especifiquéis `encoding="UTF-8"` .

Para definir un vector constante podemos usar la función

```
rep(a, n)
```



que genera un vector que contiene el valor  $a$  repetido  $n$  veces.

```
rep(1, 6)
```

```
## [1] 1 1 1 1 1 1
```

```
rep("Palma", 5) #Las palabras, siempre entre comillas
```

```
## [1] "Palma" "Palma" "Palma" "Palma" "Palma"
```

La función `rep` también se puede usar para repetir vectores. Ahora bien, cuando decimos que queremos repetir cinco veces los valores 1, 2, 3, podemos referirnos a

1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3

o a

1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3.

Para especificar el tipo de repetición tenemos que usar el parámetro adecuado en el argumento de `rep`: si añadimos `times=5`, repetiremos el vector en bloque cinco veces (en el primer sentido), y si en cambio añadimos `each=5`, repetiremos cada valor cinco veces (en el segundo sentido).

```
rep(c(1,2,3), times=5)
```

```
## [1] 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3
```

```
rep(c(1,2,3), each=5)
```

```
## [1] 1 1 1 1 1 2 2 2 2 2 3 3 3 3 3
```

Si queremos repetir cada elemento de un vector un número diferente de veces, podemos especificarlo con el parámetro `times` igualado al vector de estas multiplicidades.

```
rep(c(1,2,3,4), times=c(2,3,4,5))
```

```
## [1] 1 1 2 2 2 3 3 3 3 4 4 4 4 4
```

Las progresiones aritméticas se pueden especificar de manera compacta usando la función `seq`. Una primera manera de hacerlo es mediante la instrucción

```
seq(a, b, by=p)
```

que especifica la progresión aritmética de paso  $p$  que empieza en  $a$ ,  $a$ ,  $a + p$ ,  $a + 2p$ , ..., hasta llegar a  $b$ . En concreto, si  $a < b$  y  $p > 0$ , la función `seq(a, b, by=p)` genera un vector con la secuencia creciente  $a$ ,  $a + p$ ,  $a + 2p$ , ... hasta llegar al último valor de esta sucesión menor o igual que  $b$ . Por ejemplo:

```
seq(3, 150, by=4.5)
```

```
## [1] 3.0 7.5 12.0 16.5 21.0 25.5 30.0 34.5 39.0 43.5 48.0
## [12] 52.5 57.0 61.5 66.0 70.5 75.0 79.5 84.0 88.5 93.0 97.5
## [23] 102.0 106.5 111.0 115.5 120.0 124.5 129.0 133.5 138.0 142.5 147.0
```

Si en cambio  $a > b$  y  $p < 0$ , entonces `seq(a, b, by=p)` genera un vector con la secuencia decreciente  $a$ ,  $a + p$ ,  $a + 2p$ , ... hasta parar en el último valor de esta sucesión mayor o igual que  $b$ . Por ejemplo:

```
seq(80, 4, by=-3.5)
```

```
## [1] 80.0 76.5 73.0 69.5 66.0 62.5 59.0 55.5 52.0 48.5 45.0 41.5 38.0 34.5
## [15] 31.0 27.5 24.0 20.5 17.0 13.5 10.0 6.5
```

Finalmente, si el signo de  $p$  no es el correcto, obtenemos un mensaje de error:

```
seq(80, 4, by=3.5)
```

```
## Error in seq.default(80, 4, by = 3.5): wrong sign in 'by' argument
```

Como vimos en la lección anterior, la instrucción `seq` con paso  $\pm 1$  se puede abreviar con el signo `:`. La instrucción `a:b` define la secuencia de números consecutivos entre dos números  $a$  y  $b$ , es decir, la secuencia  $a, a + 1, a + 2, \dots$  hasta llegar a  $b$  (si  $a < b$ ), o  $a, a - 1, a - 2, \dots$  hasta llegar a  $b$  (si  $a > b$ ).

```
1:15
```

```
## [1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
```

```
2.3:12.5
```

```
## [1] 2.3 3.3 4.3 5.3 6.3 7.3 8.3 9.3 10.3 11.3 12.3
```

```
34:-5
```

```
## [1] 34 33 32 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12
```

```
## [24] 11 10 9 8 7 6 5 4 3 2 1 0 -1 -2 -3 -4 -5
```

Id con cuidado con los paréntesis y las operaciones al usar este operador:

```
-3:5
```

```
## [1] -3 -2 -1 0 1 2 3 4 5
```

```
-(3:5)
```

```
## [1] -3 -4 -5
```

```
2:3*4
```

```
## [1] 8 12
```

```
2:(3*4)
```

```
## [1] 2 3 4 5 6 7 8 9 10 11 12
```

La función `seq` también se puede usar para definir progresiones aritméticas de otras dos maneras. En primer lugar, la función

```
seq(a, b, length.out=n)
```

define la progresión aritmética de longitud  $n$  que va de  $a$  a  $b$ ; su paso es, por lo tanto,  $p = (b - a)/(n - 1)$  si  $n > 1$ ; si  $n = 1$  sólo produce el valor  $a$ .

```
seq(2, 10, length.out=10)
```

```
## [1] 2.000000 2.888889 3.777778 4.666667 5.555556 6.444444 7.333333  
## [8] 8.222222 9.111111 10.000000
```

Por otro lado,

```
seq(a, by=p, length.out=n)
```

define la progresión aritmética de longitud  $n$  y paso  $p$  que empieza en  $a$ :

$a, a + p, a + 2p, \dots, a + (n - 1)p$ .

```
seq(2, by=0.5, length.out=10)
```

```
## [1] 2.0 2.5 3.0 3.5 4.0 4.5 5.0 5.5 6.0 6.5
```

A estas alturas habréis observado que cuando el resultado de una instrucción es un vector, R comienza cada línea del resultado con un número entre corchetes `[ ]`. Este número indica la posición dentro del vector de la primera entrada de la línea correspondiente. De esta manera, en el resultado de `seq(2, 10, length.out=10)`, R nos indica que 2.000000 es el primer elemento de este vector y 8.222222 su octavo elemento.

La función `c` que hemos usado para crear vectores en realidad *concatena* sus argumentos en un vector (de ahí viene la `c`). Si la aplicamos a vectores, crea un nuevo vector concatenando sus elementos. Podemos mezclar vectores y datos en su argumento.

```
x=c(rep(1, 10), 2:10)
```

```
x
```

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

```
x=c(0,x,20,30)
```

```
x
```

```
## [1] 0 1 1 1 1 1 1 1 1 1 1 2 3 4 5 6 7 8 9 10 20 30
```

Esta última construcción, `x=c(0,x,20,30)`, muestra que la función `c` se puede usar para añadir valores al principio o al final de un vector sin cambiarle el nombre: en este caso, hemos redefinido `x` añadiéndole un 0 al principio y 20, 30 al final.

Un vector se puede modificar fácilmente usando el editor de datos que incorpora *Rstudio*. Para hacerlo, se aplica la función `fix` al vector que queremos editar. R abre entonces el vector en una nueva ventana de edición. Mientras esta ventana esté abierta, será la ventana activa de R y no podremos volver a nuestra sesión de R hasta que la cerremos. Los cambios que hagamos en el vector con el editor de datos se guardarán cuando cerremos esta ventana.

Probadlo. Cread un vector con R y abridlo en el editor. Por ejemplo:

```
x=c(rep(1, 10), 2:10)
fix(x)
```

Se abrirá entonces una ventana como la que mostramos en la Figura 4.1. Ahora, en esta ventana, podéis añadir, borrar y cambiar los datos que queráis. Por ejemplo, añadid un 0 al principio y 20, 30 al final y guardad el resultado (pulsando el botón Save en la ventana del editor). El contenido del vector `x` se habrá modificado, como podréis comprobar entrando `x` en la consola.

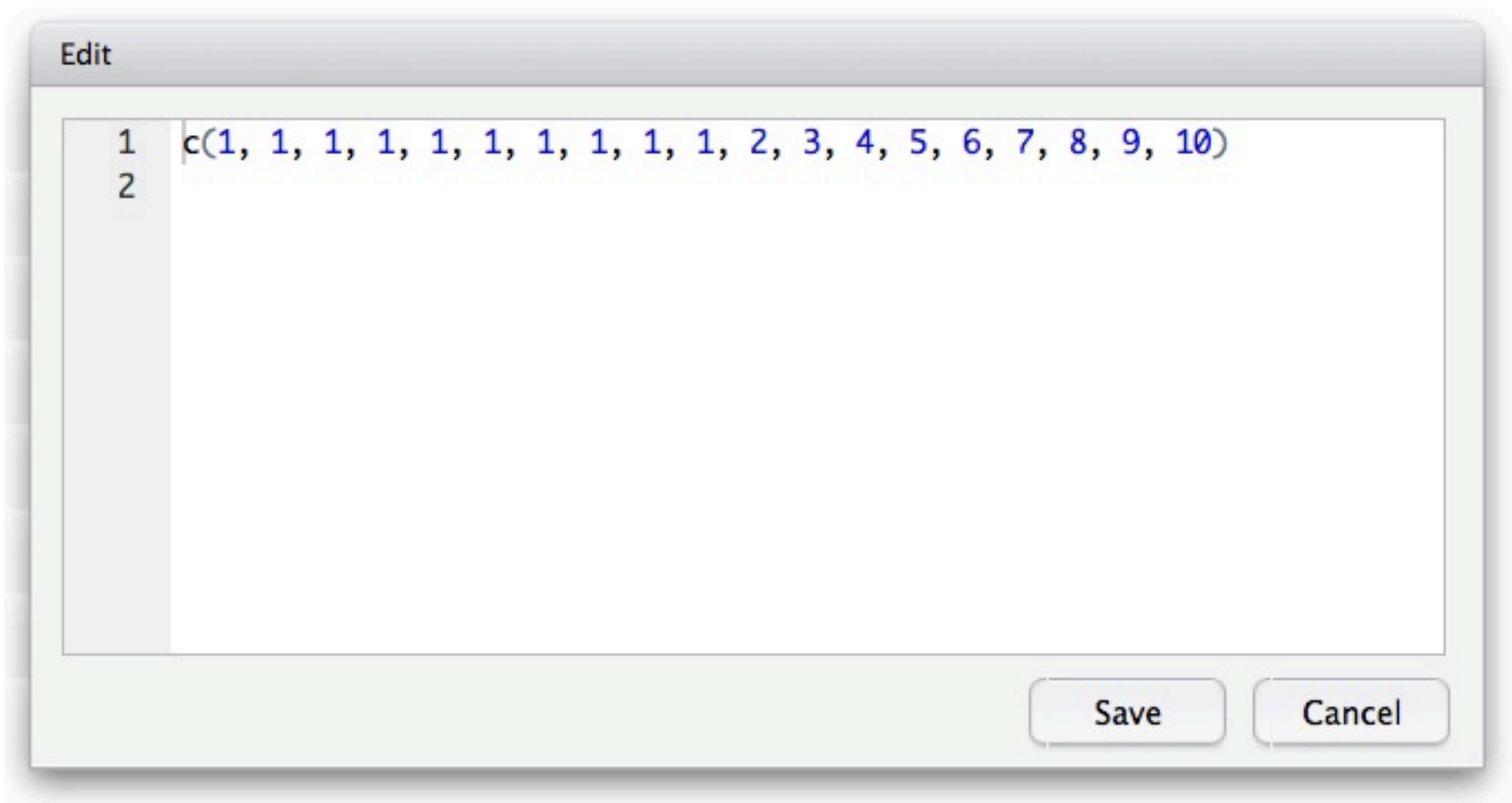


Figura 4.1: Ventana del editor de vectores de *RStudio* para Mac OS X.

## 4.2 Operaciones con vectores

El manejo de vectores con R tiene una propiedad muy útil, que ya observamos en la sección anterior al entrar `-(3:5)` o `2:3*4` : podemos aplicar una función a todos los elementos de un vector en un solo paso.

```
x=seq(2, 30, by=3)
```

```
x
```

```
## [1] 2 5 8 11 14 17 20 23 26 29
```

```
x+2.5
```

```
## [1] 4.5 7.5 10.5 13.5 16.5 19.5 22.5 25.5 28.5 31.5
```

```
2.5*x
```

```
## [1] 5.0 12.5 20.0 27.5 35.0 42.5 50.0 57.5 65.0 72.5
```

```
sqrt(x)
```

```
## [1] 1.414214 2.236068 2.828427 3.316625 3.741657 4.123106 4.472136
```

```
## [8] 4.795832 5.099020 5.385165
```

```
2^x
```

```
## [1] 4 32 256 2048 16384 131072 1048576
```

```
## [8] 8388608 67108864 536870912
```

```
x^2
```

```
## [1] 4 25 64 121 196 289 400 529 676 841
```

```
(1:4)^2
```

```
## [1] 1 4 9 16
```

```
1:4^2 #Cuidado con los paréntesis
```

```
## [1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
```

A veces no es posible aplicar una función concreta a todo un vector entrándolo dentro del argumento de la función, como hemos hecho en los ejemplos anteriores. En estos casos, podemos usar la instrucción `sapply(vector, FUN=función)`.

Por ejemplo, dentro de un rato veremos que la función `mean` calcula la media aritmética de un vector. Supongamos que definimos una función `F` que, aplicada a un número natural  $x$ , calcula la media de los números  $1, 2, \dots, x$ .

```
F=function(x){mean(1:x)}  
F(20)
```

```
## [1] 10.5
```

```
F(30)
```

```
## [1] 15.5
```

Resulta que no podemos aplicar esta función a todas las entradas de un vector  $x$  entrando simplemente `F(x)`.



```
F(20:30)
```

```
## [1] 10.5
```

En casos como este, podemos recurrir a la función `sapply` .

```
sapply(20:30, FUN=F)
```

```
## [1] 10.5 11.0 11.5 12.0 12.5 13.0 13.5 14.0 14.5 15.0 15.5
```

También podemos operar término a término las entradas de dos vectores de la misma longitud.

```
1:5+1:5 #Suma entrada a entrada
```

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

```
(1:5)*(1:5) #Producto entrada a entrada
```

```
## [1] 1 4 9 16 25
```

```
(1:5)^(1:5) #Potencia entrada a entrada
```

```
## [1] 1 4 27 256 3125
```

Esto nos permite calcular fácilmente vectores de la forma  $(x_n)_{n=p,\dots,q}$ , formados por los términos  $x_p, x_{p+1}, \dots, x_q$  de una sucesión  $(x_n)_n$ , a partir de la fórmula explícita de  $x_n$  como función del índice  $n$ : basta aplicar esta fórmula a `p:q` . Por ejemplo, para definir el vector

$$x = (3 \cdot 2^n - 20)_{n=1, \dots, 20}$$

podemos entrar lo siguiente:

```
n=1:20 #Secuencia 1,...,20, y la llamamos n por comodidad
x=3*2^n-20 #Aplicamos la fórmula a n=1,...,20
x
```

```
## [1] -14 -8 4 28 76 172 364 748
## [9] 1516 3052 6124 12268 24556 49132 98284 196588
## [17] 393196 786412 1572844 3145708
```

De manera similar, para definir el vector

$$y = \left( \frac{n}{n^2 + 1} \right)_{n=0, \dots, 20}$$

podemos usar lo siguiente:

```
n=0:20
y=n/(n^2+1)
y
```

```
## [1] 0.00000000 0.50000000 0.40000000 0.30000000 0.23529412 0.19230769
## [7] 0.16216216 0.14000000 0.12307692 0.10975610 0.09900990 0.09016393
## [13] 0.08275862 0.07647059 0.07106599 0.06637168 0.06225681 0.05862069
## [19] 0.05538462 0.05248619 0.04987531
```

En ambos casos, y para facilitar la visualización de la construcción, hemos creado el vector  $n$  con los índices de los términos de la sucesión, y después hemos obtenido el trozo de sucesión deseado aplicando la función que la define a  $n$ . También habríamos podido generar estos vectores escribiendo directamente la sucesión de índices en la fórmula que los define. Por ejemplo:

```
(0:20)/((0:20)^2+1)
```

```
## [1] 0.00000000 0.50000000 0.40000000 0.30000000 0.23529412 0.19230769
## [7] 0.16216216 0.14000000 0.12307692 0.10975610 0.09900990 0.09016393
## [13] 0.08275862 0.07647059 0.07106599 0.06637168 0.06225681 0.05862069
## [19] 0.05538462 0.05248619 0.04987531
```

R dispone de muchas funciones para aplicar a vectores, relacionadas principalmente con la estadística. Veamos algunas que nos pueden ser útiles por el momento, y ya iremos viendo otras a medida que avance el curso:

- `length` calcula la longitud del vector.
- `max` y `min` calculan sus valores máximo y mínimo, respectivamente.
- `sum` calcula la suma de sus entradas.
- `prod` calcula el producto de sus entradas.
- `mean` calcula la media aritmética de sus entradas.
- `diff` calcula el vector formado por las diferencias sucesivas entre entradas del vector original.
- `cumsum` calcula el vector de **sumas acumuladas** de las entradas del vector original: cada entrada de `cumsum(x)` es la suma de las entradas de `x` hasta su posición.
- `sort` ordena los elementos del vector en el orden natural creciente del tipo de datos que lo forman: el orden numérico, el orden alfabético, etc. Si lo queremos ordenar en orden decreciente, podemos incluir en su argumento el parámetro `dec=TRUE`.
- `rev` invierte el orden de los elementos del vector; por lo tanto, `rev(sort(...))` es otra opción para ordenar en orden decreciente.

Veamos algunos ejemplos:

```
x=c(1,5,6,2,5,7,8,3,5,2,1,0)
length(x)
```

```
## [1] 12
```

```
max(x)
```

```
## [1] 8
```

```
min(x)
```

```
## [1] 0
```

```
sum(x)
```

```
## [1] 45
```

```
prod(x)
```

```
## [1] 0
```

```
mean(x)
```

```
## [1] 3.75
```

```
cumsum(x)
```

```
## [1] 1 6 12 14 19 26 34 37 42 44 45 45
```

```
diff(x)
```

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

```
sort(x)
```

```
## [1] 0 1 1 2 2 3 5 5 5 6 7 8
```

```
sort(x, dec=TRUE)
```

```
## [1] 8 7 6 5 5 5 3 2 2 1 1 0
```

```
rev(x)
```

```
## [1] 0 1 2 5 3 8 7 5 2 6 5 1
```

La función `sum` es útil para evaluar sumatorios; por ejemplo, si queremos calcular

$$\sum_{n=0}^{200} \frac{1}{n^2 + 1}$$

sólo tenemos que entrar:

```
n=0:200
```

```
sum(1/(n^2+1))
```

```
## [1] 2.071687
```

La función `cumsum` permite definir sucesiones descritas mediante sumatorios; a modo de ejemplo, para definir la sucesión de los 20 primeros **números armónicos**

$$y = \left( \sum_{i=1}^n \frac{1}{i} \right)_{n=1, \dots, 20}$$

basta aplicar `cumsum` al vector  $x = (1/i)_{i=1, \dots, 20}$  de la manera siguiente:

```
i=1:20
x=1/i
y=cumsum(x)
y
```

```
## [1] 1.000000 1.500000 1.833333 2.083333 2.283333 2.450000 2.592857
## [8] 2.717857 2.828968 2.928968 3.019877 3.103211 3.180134 3.251562
## [15] 3.318229 3.380729 3.439553 3.495108 3.547740 3.597740
```

## 4.3 Entradas y trozos de vectores

Si queremos extraer el valor de una entrada concreta de un vector, o si queremos referirnos a esta entrada para usarla en un cálculo, podemos emplear la construcción

```
vector[i]
```

que indica la  $i$ -ésima entrada del `vector`. En particular, `vector[length(vector)-i]` es la  $(i + 1)$ -ésima entrada del `vector` empezando por el final: su última entrada es `vector[length(vector)]`, la penúltima es `vector[length(vector)-1]` y así sucesivamente.

Observad que para referirnos a elementos de un vector, empleamos corchetes `[ ]`, y no los paréntesis redondos usuales.

```
x=seq(2, 50, by=1.5)
x
```

```
## [1] 2.0 3.5 5.0 6.5 8.0 9.5 11.0 12.5 14.0 15.5 17.0 18.5 20.0 21.5
## [15] 23.0 24.5 26.0 27.5 29.0 30.5 32.0 33.5 35.0 36.5 38.0 39.5 41.0 42.5
## [29] 44.0 45.5 47.0 48.5 50.0
```

```
x(3) #¿La tercera entrada del vector?
```

```
## Error in x(3): could not find function "x"
```

```
x[3] #La tercera entrada del vector, ahora sí
```

```
## [1] 5
```

```
x[length(x)] #La última entrada del vector
```

```
## [1] 50
```

```
x[length(x)-5] #La sexta entrada del vector empezando por el final
```

```
## [1] 42.5
```

También podemos extraer subvectores de un vector. Una primera manera de obtener un subvector es especificando los índices de las entradas que lo han de formar:

- `vector[y]` , donde `y` es un vector (de índices), crea un nuevo vector con las entradas del `vector` original cuyos índices pertenecen a `y` .
- En particular, si  $a$  y  $b$  son dos números naturales, `vector[a:b]` crea un nuevo vector con las entradas del `vector` original que van de la  $a$ -ésima a la  $b$ -ésima.
- `vector[-y]` , donde `y` es un vector (de índices), es el complementario de `vector[y]` : sus entradas son las del `vector` original cuyos índices *no* pertenecen a `y` .
- En particular, `vector[-i]` borra la entrada  $i$ -ésima del `vector` original.

Veamos algunos ejemplos:

```
n=1:10
```

```
x=2*3^n-5*n^3*2^n
```

```
x
```

```
## [1] -4 -142 -1026 -4958 -19514 -67662 -215146
## [8] -642238 -1826874 -5001902
```

```
x[-3] #x sin la tercera entrada
```

```
## [1] -4 -142 -4958 -19514 -67662 -215146 -642238 -1826874
## [9] -5001902
```

```
x[3:7] #Los elementos tercero a séptimo de x
```

```
## [1] -1026 -4958 -19514 -67662 -215146
```

```
x[7:3] #Los elementos séptimo a tercero de x
```

```
## [1] -215146 -67662 -19514 -4958 -1026
```

```
x[seq(1, length(x), by=2)] #Los elementos de índice impar de x
```

```
## [1] -4 -1026 -19514 -215146 -1826874
```

```
x[seq(2, length(x), by=2)] #Los elementos de índice par
```



```
## [1]      -142      -4958      -67662      -642238      -5001902
```

```
x[-seq(1, length(x), by=2)] #Borramos los elementos de índice impar
```

```
## [1]      -142      -4958      -67662      -642238      -5001902
```

```
x[(length(x)-5):length(x)] #Los últimos 6 elementos de x
```

```
## [1]    -19514    -67662   -215146   -642238  -1826874  -5001902
```

```
x[length(x)-5:length(x)] #No os dejéis los paréntesis ...
```

```
## [1] -19514  -4958  -1026   -142    -4
```

Fijaos en las dos últimas instrucciones: si denotamos `length(x)` por  $n$ , entonces

`(length(x)-5):length(x)` es la secuencia de índices

$n-5, n-4, n-3, n-2, n-1, n$ , mientras que `length(x)-5:length(x)` es la secuencia  $n-(5, 6, 7, \dots, n) = n-5, n-6, n-7, \dots, 1, 0$ .

También podemos extraer las entradas de un vector (o sus índices) que satisfagan alguna condición. Los operadores lógicos que podemos usar para definir estas condiciones son los que damos en la lista siguiente:

- `==`: `==`
- `!=`: `!=`
- `<`: `<`
- `>`: `>`
- `<=`: `<=`
- `>=`: `>=`
- negación: `!`
- conjunción: `&`

- disjunción: |

Veamos un ejemplo (y observad su sintaxis). Vamos a extraer los elementos mayores que 3 de un vector `x`.

```
x=c(1,5,6,2,5,7,8,3,5,2,1)
x[x>3]
```

```
## [1] 5 6 5 7 8 5
```

En esta última instrucción, la construcción `x>3` define un vector que, en cada posición, contiene un `TRUE` si el elemento correspondiente del vector `x` es mayor que 3 y un `FALSE` si no lo es.

```
x>3
```

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

Entonces `x[x>3]` lo que nos da son las entradas del vector `x` correspondientes a los `TRUE` de este vector de valores lógicos.

Veamos otros ejemplos.

```
x[x>2 & x<=5] #Elementos mayores que 2 y menores o iguales que 5
```

```
## [1] 5 5 3 5
```

```
x[x!=2 & x!=5] #Elementos diferentes de 2 y de 5
```

```
## [1] 1 6 7 8 3 1
```

```
x[x>5 | x<=2] #Elementos mayores que 5 o menores o iguales que 2
```

```
## [1] 1 6 2 7 8 2 1
```

```
x[x>=4] #Elementos mayores o iguales que 4
```

```
## [1] 5 6 5 7 8 5
```

```
x[!x<4] #Elementos que NO son menores que 4; es equivalente a la anterior
```

```
## [1] 5 6 5 7 8 5
```

```
x[x%%4==0] #Elementos múltiplos de 4
```

```
## [1] 8
```

```
x[x>3]
```

```
## [1] 5 6 5 7 8 5
```

Esta construcción también permite extraer las entradas de un vector cuyos índices sean los de las entradas de otro vector que satisfagan una condición lógica. Por ejemplo:

```
x
```

```
## [1] 1 5 6 2 5 7 8 3 5 2 1
```

```
y=c(2,-3,0,1,2,-1,4,-1,-2,3,5)
```

```
x[y>0] #Entradas de x correspondientes a entradas positivas de y
```

```
## [1] 1 2 5 8 2 1
```

Para obtener los *índices* de las entradas del vector que satisfacen una condición dada, podemos usar la función `which`. Esta función, aplicada a un vector de valores lógicos, da los índices de las posiciones de los `TRUE`. Así, para saber los índices de las entradas de `x` que son mayores que 3, usamos `which(x>3)`, que nos dará los índices de las entradas `TRUE` del vector `x>3`.

```
x
```

```
## [1] 1 5 6 2 5 7 8 3 5 2 1
```

```
x[x>3] #Elementos mayores que 3
```

```
## [1] 5 6 5 7 8 5
```

```
which(x>3) #Índices de los elementos mayores que 3
```

```
## [1] 2 3 5 6 7 9
```

Veamos otros ejemplos:

```
which(x>2 & x<=5) #Índices de los elementos > 2 y <= 5
```

```
## [1] 2 5 8 9
```

```
which(x!=2 & x!=5) #Índices de los elementos diferentes de 2 y 5
```

```
## [1] 1 3 6 7 8 11
```

```
which(x>5 | x<=2) #Índices de los elementos > 5 o <= 2
```

```
## [1] 1 3 4 6 7 10 11
```

```
which(x%%2==0) #Índices de los elementos pares del vector
```

```
## [1] 3 4 7 10
```

La instrucción `which.min(x)` nos da la primera posición en la que el vector toma su valor mínimo; `which.max(x)` hace lo mismo, pero para el máximo. En cambio, con `which(x==min(x))` obtenemos todas las posiciones en las que el vector toma su valor mínimo y, con `which(x==max(x))`, aquellas en las que toma su valor máximo.

```
x
```

```
## [1] 1 5 6 2 5 7 8 3 5 2 1
```

```
which.min(x)
```

```
## [1] 1
```

```
which(x==min(x))
```

```
## [1] 1 11
```

Si un vector no contiene ningún término que satisfaga la condición que imponemos, obtenemos como respuesta un vector vacío. R lo indica con `numeric(0)` si es de números, `character(0)` si es de palabras, o `integer(0)` si es de índices de entradas de un vector. Estos vectores vacíos tienen longitud, naturalmente, 0.

```
x=2^(0:10)
```

```
x
```

```
## [1] 1 2 4 8 16 32 64 128 256 512 1024
```

```
x[20<x & x<30] #Elementos de x estrictamente entre 20 y 30
```

```
## numeric(0)
```

```
length(x[20<x & x<30]) #¿Cuántas entradas hay entre 20 y 30?
```

```
## [1] 0
```

```
which(x>1500) #Índices de elementos mayores que 1500
```

```
## integer(0)
```

Si R no sabe de qué tipo son los datos que faltan en un vector vacío, lo indica con `NULL`. También podemos usar este valor para definir un vector vacío.

```
x=c()
```

```
x
```

```
## NULL
```

```
z=NULL
```

```
z
```

```
## NULL
```

```
y=c(x, 2, z)
```

```
y
```

```
## [1] 2
```

Los operadores lógicos que hemos explicado también se pueden usar para pedir si una condición sobre unos números concretos se satisface o no. Por ejemplo:

```
exp(pi)>pi^(exp(1)) #¿Es mayor e^pi que pi^e?
```

```
## [1] TRUE
```

```
1234567%%9==0 #¿Es 1234567 múltiplo de 9?
```

```
## [1] FALSE
```

Podemos modificar algunas entradas de un vector simplemente declarando sus nuevos valores. Esto se puede hacer entrada a entrada, o para todo un subvector de golpe.

```
x=1:10
```

```
x
```

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

```
x[3]=15 #En la posición 3 escribimos 15  
x[11]=25 #Añadimos en la posición 11 un 25  
x
```

```
## [1] 1 2 15 4 5 6 7 8 9 10 25
```

```
x[2:4]=x[2:4]+10 #Sumamos 10 a las entradas en las posiciones 2 a 4  
x
```

```
## [1] 1 12 25 14 5 6 7 8 9 10 25
```

```
x[(length(x)-2):length(x)]=0 #Igualamos las últimas tres entradas a 0  
x
```

```
## [1] 1 12 25 14 5 6 7 8 0 0 0
```

Fijaos en la próxima instrucción:

```
x[length(x)+3]=2  
x
```

```
## [1] 1 12 25 14 5 6 7 8 0 0 0 NA NA 2
```

Hemos añadido al vector `x` el valor 2 tres posiciones más allá de su última entrada. Entonces, en las posiciones 12 y 13 ha escrito `NA` antes de añadir en la 14 el 2. Estos `NA`, de *Not Available*, indican que las entradas correspondientes del vector no existen.



Los `NA` serán muy importantes cuando usemos vectores en estadística descriptiva, donde podrán representar valores desconocidos, errores, etc. Serán importantes porque son molestos, puesto que, por norma general, una función aplicada a un vector que contenga algún `NA` da `NA` .

```
sum(x)
```

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

```
mean(x)
```

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

Afortunadamente, muchas de las funciones para vectores admiten un parámetro `na.rm` que, igualado a `TRUE` , hace que la función sólo tenga en cuenta las entradas definidas.

```
sum(x, na.rm=TRUE)
```

```
## [1] 80
```

```
mean(x, na.rm=TRUE)
```

```
## [1] 6.666667
```

Para especificar las entradas no definidas de un vector  $x$  no podemos usar la condición lógica `x==NA` , sino la función `is.na(x)` .

```
x
```

```
## [1] 1 12 25 14 5 6 7 8 0 0 0 NA NA 2
```

```
which(x==NA) #¿Índices de entradas NA?
```

```
## integer(0)
```

```
is.na(x)
```

```
## [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [12] TRUE TRUE FALSE
```

```
which(is.na(x)) #Índices de entradas NA
```

```
## [1] 12 13
```

```
y=x #Creamos una copia de x y la llamamos y
y[is.na(y)]=mean(y, na.rm=TRUE) #Cambiamos los NA de y por la media del resto de y
```

```
## [1] 1.000000 12.000000 25.000000 14.000000 5.000000 6.000000 7.000000
## [8] 8.000000 0.000000 0.000000 0.000000 6.666667 6.666667 2.000000
```

Naturalmente, podemos usar la negación de `is.na(x)` para obtener las entradas definidas de un vector `x` : formarán el vector `x[!is.na(x)]` .

```
x
```

```
## [1] 1 12 25 14 5 6 7 8 0 0 0 NA NA 2
```

```
x[!is.na(x)]
```

```
## [1] 1 12 25 14 5 6 7 8 0 0 0 2
```

```
sum(x[!is.na(x)])
```

```
## [1] 80
```

```
cumsum(x)
```

```
## [1] 1 13 38 52 57 63 70 78 78 78 78 NA NA NA
```

```
cumsum(x, na.rm=TRUE) #cumsum no admite na.rm
```

```
## Error in cumsum(x, na.rm = TRUE): 2 arguments passed to 'cumsum' which requires
```

```
cumsum(x[!is.na(x)])
```

```
## [1] 1 13 38 52 57 63 70 78 78 78 78 80
```

Las entradas no definidas de un vector también se pueden borrar aplicándole la función

```
na.omit .
```

```
na.omit(x)
```

```
## [1] 1 12 25 14 5 6 7 8 0 0 0 2
## attr(,"na.action")
## [1] 12 13
## attr(,"class")
## [1] "omit"
```

```
sum(na.omit(x))
```

```
## [1] 80
```

```
cumsum(na.omit(x))
```

```
## [1] 1 13 38 52 57 63 70 78 78 78 78 80
```

Observad el resultado de `na.omit(x)` . Contiene un primer vector formado por las entradas del vector original que no son `NA` , y luego una serie de información extra llamados **atributos**, e indicados por R con `attr` : los índices de las entradas que ha eliminado y el tipo de acción que ha llevado a cabo. Como podéis ver, estos atributos no interfieren para nada en las operaciones que se realicen con el primer vector, pero si os molestan, se pueden eliminar: la instrucción

```
attr(objeto , atributo)=NULL
```

borra el `atributo` del `objeto` .

```
x_sinNA=na.omit(x)
x_sinNA
```

```
## [1] 1 12 25 14 5 6 7 8 0 0 0 2
## attr(,"na.action")
## [1] 12 13
## attr(,"class")
## [1] "omit"
```

```
attr(x_sinNA, "na.action")=NULL
attr(x_sinNA, "class")=NULL
x_sinNA
```

```
## [1] 1 12 25 14 5 6 7 8 0 0 0 2
```

## 4.4 Factores

Un **factor** es como un vector, pero con una estructura interna más rica que permite usarlo para clasificar observaciones. Para ilustrar la diferencia entre vectores y factores, vamos a crear un vector `Ciudades` con los nombres de algunas ciudades, y a continuación un factor `Ciudades.factor` con el mismo contenido, aplicando a este vector la función `factor`.

```
Ciudades=c("Madrid","Palma","Madrid","Madrid","Barcelona","Palma","Madrid","Madrid")
Ciudades
```

```
## [1] "Madrid" "Palma" "Madrid" "Madrid" "Barcelona" "Palma"
## [7] "Madrid" "Madrid"
```

```
Ciudades.factor=factor(Ciudades)
Ciudades.factor
```

```
## [1] Madrid    Palma      Madrid    Madrid    Barcelona Palma      Madrid
## [8] Madrid
## Levels: Barcelona Madrid Palma
```

Observad la diferencia. El factor dispone de un atributo especial llamado `niveles` (*levels*), y cada elemento del factor es igual a un nivel; de esta manera, los niveles *clasifican* las entradas del factor. Podríamos decir, en resumen, que un factor es una lista formada por copias de etiquetas (los niveles), como podrían ser el sexo o la especie de unos individuos.

Cuando tengamos un vector que queramos usar para clasificar datos, conviene definirlo como un factor y así podremos hacer más cosas con él. Para crear un factor, hemos de definir un vector y transformarlo en factor por medio de una de las funciones `factor` o `as.factor`. La diferencia entre estas funciones es que `as.factor` *convierte* el vector en un factor, y toma como sus niveles los diferentes valores que aparecen en el vector, mientras que `factor` *define* un factor a partir del vector, y dispone de algunos parámetros que permiten modificar el factor que se crea, tales como:

- `levels`, que permite especificar los niveles e incluso añadir niveles que no aparecen en el vector.
- `labels`, que permite cambiar los nombres de los niveles.

De esta manera, con `as.factor` o con `factor` sin especificar `levels`, el factor tendrá como niveles los diferentes valores que toman las entradas del vector, y además aparecerán en su lista de niveles, `Levels`, ordenados en orden alfabético. Si especificamos el parámetro `levels` en la función `factor`, los niveles aparecerán en dicha lista en el orden en el que los entremos en él.

```
S=c("M","M","F","M","F","F","F","M","M","F")
Sex=as.factor(S)
Sex
```

```
## [1] M M F M F F F M M F
## Levels: F M
```

```
Sex2=factor(S)    #Esto definirá el mismo factor
```

```
Sex2
```

```
##  [1] M M F M F F F M M F
```

```
## Levels: F M
```

Ahora vamos a añadir un tercer nivel, `I` , que no está representado en `S` :

```
Sex3=factor(S, levels=c("F","M","I"))
```

```
Sex3
```

```
##  [1] M M F M F F F M M F
```

```
## Levels: F M I
```

Fijaos en que ahora R no ordena alfabéticamente los niveles, sino en el orden especificado en el `levels` .

Y ahora vamos a cambiar el orden de los niveles y su nombre:

```
Sex4=factor(S, levels=c("M","F","I"), labels=c("Masc.", "Fem.", "Indet."))
```

```
Sex4
```

```
##  [1] Masc. Masc. Fem.  Masc. Fem.  Fem.  Fem.  Masc. Masc. Fem.
```

```
## Levels: Masc. Fem. Indet.
```

Para obtener los niveles de un factor, podemos emplear la función `levels` .

```
levels(Sex)
```

```
## [1] "F" "M"
```

```
levels(Sex4)
```

```
## [1] "Masc." "Fem." "Indet."
```

La función `levels` también permite cambiar los nombres de los niveles de un factor.

```
Notas=as.factor(c(1,2,2,3,1,3,2,4,2,3,4,2))
```

```
Notas
```

```
## [1] 1 2 2 3 1 3 2 4 2 3 4 2
```

```
## Levels: 1 2 3 4
```

```
levels(Notas)=c("Muy.mal", "Mal", "Bien", "Muy.bien")
```

```
Notas
```

```
## [1] Muy.mal Mal Mal Bien Muy.mal Bien Mal
```

```
## [8] Muy.bien Mal Bien Muy.bien Mal
```

```
## Levels: Muy.mal Mal Bien Muy.bien
```

Observad que los niveles han heredado el orden del factor original.

Con la función `levels` también podemos agrupar varios niveles de un factor en uno solo, simplemente repitiendo nombres al especificarlos; por ejemplo, en el factor de notas anterior, vamos a agrupar los niveles “Muy mal” y “Mal” en uno solo, y lo mismo con los niveles “Muy bien” y “Bien”:

```
Notas_2niv=Notas
```

```
levels(Notas_2niv)=c("Mal", "Mal", "Bien", "Bien")
```

```
Notas_2niv
```



```
## [1] Mal Mal Mal Bien Mal Bien Mal Bien Bien Mal
## Levels: Mal Bien
```

Nos hemos referido varias veces al orden de los niveles. En realidad, hay dos tipos de factores: simples y ordenados. Hasta ahora sólo hemos considerado los factores **simples**, en los que el orden de los niveles realmente no importa, y si lo modificamos es sólo por razones estéticas o de comprensión de los datos; en este caso, la manera más sencilla de hacerlo es redefiniendo el factor con `factor` y modificando en el parámetro `levels` el orden de los niveles. Pero si el orden de los niveles es relevante para analizar los datos, entonces es conveniente definir el factor como **ordenado**. Esto se lleva a cabo con la función `ordered`, que dispone de los mismos parámetros que `factor`. Así, si queremos que nuestro factor `Notas` sea un factor ordenado, con sus niveles ordenados de “Muy mal” a “Muy bien”, hay que entrar lo siguiente:

```
Notas=ordered(Notas, levels=c("Muy.mal", "Mal", "Bien", "Muy.bien"))
Notas
```

```
## [1] Muy.mal Mal Mal Bien Muy.mal Bien Mal
## [8] Muy.bien Mal Bien Muy.bien Mal
## Levels: Muy.mal < Mal < Bien < Muy.bien
```

Observad que R indica el orden de los niveles de un factor ordenado mediante el signo `<`.

Aunque en la instrucción anterior hemos aplicado la función `ordered` a un factor, también se puede aplicar a un vector, como si usáramos `factor`.

## 4.5 Listas heterogéneas

Los vectores que hemos estudiado hasta el momento sólo pueden contener datos, y estos datos han de ser de un solo tipo. Por ejemplo, no podemos construir un vector que contenga simultáneamente palabras y números, o cuyas entradas sean a su vez vectores. Este problema se resuelve con las **listas heterogéneas**; para abreviar, las llamaremos por

su nombre en R: ***list***. Una *list* es una lista formada por objetos que pueden ser de clases diferentes. Así, en una misma *list* podemos combinar números, palabras, vectores, otras *list*, etc. En la Lección 3 ya aparecieron dos objetos de clase `list` : los resultados de `lm(...)` y `summary(lm(...))` .

Supongamos por ejemplo que queremos guardar en una lista un vector, su nombre, su media, y su vector de sumas acumuladas. En este caso, tendríamos que hacerlo en forma de lista heterogénea usando la función `list` .

```
x=c(1,2,-3,-4,5,6)
L=list(nombre="x",vector=x,media=mean(x),sumas=cumsum(x))
L
```

```
## $nombre
## [1] "x"
##
## $vector
## [1]  1  2 -3 -4  5  6
##
## $media
## [1] 1.166667
##
## $sumas
## [1]  1  3  0 -4  1  7
```

Observad la sintaxis de la función `list` : le hemos entrado como argumento los diferentes objetos que van a formar la lista heterogénea, poniendo a cada uno un nombre adecuado. Este nombre es interno de la *list*: por ejemplo, pese a que `L` contiene un objeto llamado `sumas` , en el entorno de trabajo de R no tenemos definida ninguna variable con ese nombre (a no ser que la hayamos definido previamente durante la sesión).

```
sumas
```

```
## Error in eval(expr, envir, enclos): object 'sumas' not found
```

Para referirnos o usar una componente concreta de una *list*, tenemos que añadir al nombre de la *list* el sufijo formado por un signo `$` y el nombre de la componente; recordad cómo extraíamos el valor de  $R^2$  de un `summary(lm(...))` en la Sección [3.1](#).

```
L$nombre
```

```
## [1] "x"
```

```
L$vector
```

```
## [1]  1  2 -3 -4  5  6
```

```
L$media
```

```
## [1] 1.166667
```

También podemos indicar el objeto por su posición en la *list* usando un par de dobles corchetes `[[ ]]`. Si usamos sólo un par de corchetes, como en los vectores, lo que obtenemos es una *list* formada por esa única componente, no el objeto que forma la componente.

```
L[[1]]
```

```
## [1] "x"
```

```
L[[4]] #Esto es un vector
```

```
## [1]  1  3  0 -4  1  7
```

```
3*L[[4]] #Y podemos operar con él
```

```
## [1] 3 9 0 -12 3 21
```

```
L[4] #Esto es una list, no un vector
```

```
## $sumas
```

```
## [1] 1 3 0 -4 1 7
```

```
3*L[4] #Y NO podemos operar con él
```

```
## Error in 3 * L[4]: non-numeric argument to binary operator
```

Para conocer la estructura interna de una *list*, es decir, los nombres de los objetos que la forman y su naturaleza, podemos usar la función `str`. Si sólo queremos saber sus nombres, podemos usar la función `names`. Si la *list* se obtiene con una función de R cuyo resultado sea una estructura de este tipo, como, por ejemplo, `lm`, es recomendable consultar la Ayuda de la función, ya que probablemente explique el significado de los objetos que la forman.

```
str(L)
```

```
## List of 4
```

```
## $ nombre: chr "x"
```

```
## $ vector: num [1:6] 1 2 -3 -4 5 6
```

```
## $ media : num 1.17
```

```
## $ sumas : num [1:6] 1 3 0 -4 1 7
```

```
names(L)
```

```
## [1] "nombre" "vector" "media" "sumas"
```

Finalmente, queremos comentar que la función `c` también se puede usar para concatenar *lists* o para añadir miembros a una *list*:

```
L=c(L,numero.pi=pi)
```

```
L
```

```
## $nombre
```

```
## [1] "x"
```

```
##
```

```
## $vector
```

```
## [1] 1 2 -3 -4 5 6
```

```
##
```

```
## $media
```

```
## [1] 1.166667
```

```
##
```

```
## $sumas
```

```
## [1] 1 3 0 -4 1 7
```

```
##
```

```
## $numero.pi
```

```
## [1] 3.141593
```

## 4.6 Guía rápida de funciones

- `c` sirve para definir un vector concatenando elementos o vectores. También sirve para concatenar *lists*.
- `scan` crea un vector importando datos que se entren en la consola o contenidos en un fichero. Algunos parámetros importantes:
  - `dec` : indica el separador decimal.
  - `sep` : indica el signo usado para separar las entradas.
  - `what` : indica el tipo de datos que se importan.

- `encoding` : indica la codificación de alfabeto del fichero externo; sus dos valores posibles son `"latin1"` y `"UTF-8"` .
- `rep` sirve para definir un vector repitiendo un valor o las entradas de otro vector.

Algunos parámetros importantes:

- `each` : cuando aplicamos la función a un vector, sirve para indicar cuántas veces queremos repetir cada entrada del vector.
- `times` : cuando aplicamos la función a un vector, sirve para indicar cuántas veces queremos repetir todo el vector en bloque.
- `seq` sirve para definir progresiones aritméticas. Sus tres usos principales son:
  - `seq(a, b, by=p)` define la progresión  $a, a + p, a + 2p, \dots, b$  (o parándose en el término inmediatamente anterior a  $b$ , si  $b$  no pertenece a la progresión).
  - `seq(a, b, length.out=n)` define la progresión  $\overbrace{a, a + p, a + 2p, \dots, b}^n$  de longitud  $n$ , tomando como paso  $p = (b - a) / (n - 1)$ .
  - `seq(a, by=p, length.out=n)` define la progresión  $a, a + p, a + 2p, \dots, a + (n - 1)p$
- `a:b` es sinónimo de `seq(a, b, by=1)` (si  $a < b$ ) o `seq(a, b, by=-1)` (si  $a > b$ ).
- `NULL` indica un vector vacío.
- `fix` abre un vector (o, en general, un objeto de datos: una matriz, un *data frame*...) en el editor de datos.
- Funciones para vectores:
  - `length` : calcula la longitud de un vector.
  - `max` : calcula el máximo de un vector.
  - `min` : calcula el mínimo de un vector
  - `sum` : calcula la suma de las entradas de un vector.
  - `prod` : calcula el producto de las entradas de un vector.
  - `mean` : calcula la media de las entradas de un vector.
  - `cumsum` : calcula el vector de sumas acumuladas de un vector
  - `diff` : calcula el vector de diferencias consecutivas de un vector.
  - `sort` : ordena en orden creciente las entradas de un vector.
  - `rev` : invierte el orden de un vector.

Las funciones `max` , `min` , `sum` , `prod` y `mean` admiten el parámetro `na.rm=TRUE` que impone que no se tengan en cuenta los valores `NA` del vector al calcularla.

- `sapply(vector, FUN=función)` aplica la `función` a todas las entradas del `vector` .
- `vector[...]` se usa para especificar un elemento o un subvector del `vector` . Las entradas que formarán el subvector pueden especificarse mediante el vector de sus índices o mediante una condición lógica sobre las entradas. Los signos de operadores lógicos que se pueden usar para definir condiciones lógicas son los siguientes:
  - `==` :  $=$
  - `!=` :  $\neq$
  - `<` :  $<$
  - `>` :  $>$
  - `<=` :  $\leq$
  - `>=` :  $\geq$
  - `!` : negación
  - `&` : conjunción
  - `|` : disjunción
- `which` sirve para obtener los índices de las entradas de un vector que satisfacen una condición lógica.
- `which.min` y `which.max` dan la primera posición en la que el vector toma su valor mínimo o máximo, respectivamente.
- `is.na` es la alternativa correcta a la condición `==NA` .
- `na.omit` elimina las entradas `NA` de un vector.
- `as.factor` transforma un vector en un factor.
- `factor` crea un factor a partir de un vector. Algunos parámetros importantes:
  - `levels` : sirve para especificar los niveles. `labels` : sirve para cambiar los nombres de los niveles.
- `ordered` crea un factor ordenado a partir de un vector o un factor; sus parámetros son los mismos que los de `factor` .
- `levels` sirve para obtener los niveles de un factor, y también para cambiar sus nombres.

- `list` construye listas heterogéneas, *lists*.
- `str` sirve para obtener la estructura de una *list*.
- `names` sirve para conocer los nombres de las componentes de una *list*.
- `list$componente` sirve para referirnos al objeto que forma la `componente` de la *list*.
- `list[[i]]` sirve para referirnos al objeto que forma la *i*-ésima componente de la *list*.

## 4.7 Ejercicios

### Test

(1) Dad el valor del elemento decimocuarto de la sucesión de números consecutivos entre -25 y 71.

(2) Dad la instrucción que crea, usando la función `c`, un vector llamado `Pueblos` formado por los nombres Palma, Inca, Manacor, Binissalem.

(3) Tomad la progresión de números consecutivos que va de -7 a 20, cambiad el décimo elemento empezando por el final por un 30 y calculad la media. Dad su valor redondeado a 3 cifras decimales (el valor, no las instrucciones empleadas).

(4) Dad la instrucción que crea, usando la función `rep`, un vector llamado `As` formado por 100 copias de la letra A.

(5) Decid, indicándolo con SI (sin acento) o NO, si la igualdad  $2 \cdot 3^n - 4 \cdot 2^n = 1560$  es verdadera para algún número natural  $n$  entre 0 y 100.

(6) Dad el menor número natural  $n$  tal que  $2 \cdot 3^n - 4 \cdot 0.8^n \geq 10^6$ .

(7) Dad el valor de  $n$  en el que la secuencia  $(2 \cdot 3^n - 4 \cdot 2.5^n)_{n=0, \dots, 100}$  toma su valor mínimo.

(8) Decid, respondiendo SI (sin acento) o NO, si la secuencia  $(4^n - 3 \cdot 2^n)_{n=0, \dots, 200}$  es creciente o no.



(9) Decid si la secuencia  $(4^n - 7 \cdot 2^n)_{n=0, \dots, 200}$  es creciente, decreciente o ninguna de las dos cosas. La respuesta tiene que ser CRECIENTE, DECRECIENTE o NADA, según sea el caso.

(10) Dad el primer valor de  $n$  para el que

$$\sum_{y=0}^n \frac{e^y}{y+1}$$

es mayor o igual que  $10^6$ . Si no existe, tenéis que contestar NO.

(11) Dad el valor de la suma

$$\sum_{n=0}^{30} n \cdot e^{-n}$$

redondeado a 3 cifras decimales.

(12) Dad una instrucción que cambie los nombres de los niveles de un factor llamado `F` de 5 niveles a `S`, `A`, `N`, `E` y `MH` (en este orden).

(13) Dad una instrucción que defina un factor llamado `F01` a partir del vector  $(0, 1, 0, 0, 1, 0)$ , asignando al 0 y al 1 los niveles `No` y `Yes`, respectivamente.

(14) El quinto objeto de una *list* llamada `Datos` es un vector que además contiene valores `NA`. Dad una instrucción que calcule su media sin tener en cuenta los `NA`.

## Ejercicio

Tenemos las siguientes notas obtenidas por unos estudiantes en un examen:

```
7.9, 4.3, 5.5, 7.9, 9.8, 2.7, 4.7, 2.4, 8.3, 7.3, 6.8, 6.3, 4.8, 5.7, 3.8, 6.3, 5.4,
5.4, 80, 4.2, 8.3, 4.7, 6.0, 6.8, 5.7, 6.5, 4.6, 5.4, 3.7, 7.1, 5.5, 6.0, 6.7, 7.0,
7.3, 3.0, 6.6, 6.1, 2.4, 7.1, 9.4, 3.7, 4.5, 5.1, 5.9, 4.7, 5.5, 8.9, 8.1, 8.3, 4.
7.1, 9.3, 5.1, 6.1, 3.0, 5.7, 6.8, 3.1, 7.7, 7.3, 7.0, 6.2, 8.8, 5.3, 4.0.
```

1. Cread un vector con estas notas (podéis copiarlas de este documento y pegarlas) y ponedle un nombre adecuado.
2. ¿Cuántas notas contiene este vector? ¿Cuál es su valor medio?

3. ¡Vaya! El 80 ha sido un error, tenía que ser un 8.0. Cambiad el 80 del vector anterior por un 8.0, sin volver a entrar el resto de notas. Volved a calcular la media de las notas tras haber corregido este error.
4. ¿Cuál es la nota mínima obtenida por estos estudiantes? ¿Cuántos estudiantes la han sacado?
5. ¿Cuántos estudiantes han logrado un notable (entre 7 y 8.9)? ¿Qué porcentaje del total de estudiantes representan?
6. ¿Qué grupo es más numeroso: el de los estudiantes que han sacado entre 4 y 4.9, o el de los que han sacado entre 5 y 5.9?
7. Ordenad en orden creciente estas notas y obtened su `mediana` : una vez ordenado el vector, si tiene un número impar de entradas, su mediana es el valor central, y si tiene un número par de entradas, su mediana es la media aritmética de los dos valores centrales.
8. La mediana de un vector se puede calcular directamente con la función `median` . Calculad la del vector anterior con esta función. ¿Da lo mismo que el valor obtenido en el punto anterior?
9. ¿Cuántos notas diferentes hay en esta muestra? (Podéis emplear astutamente algunas funciones explicadas en esta lección, o podéis consultar `help.search("duplicated")` a ver si encontráis una función que elimine las entradas duplicadas de un vector.)

## Respuestas al test

(1) -12

(2) `Pueblos=c("Palma","Inca","Manacor","Binissalem")`

(3) 7.179

(4) `As=rep("A",100)`

(5) NO

(6) 12

(7) 3

(8) SI

(9) NADA

(10) 17

(11) 0.921

(12) `levels(F)=c("S","A","N","E","MH")`

(13) `F01=factor(c(0,1,0,0,1,0), labels=c("No","Yes"))`

(14) `mean(Datos[[5]],na.rm=TRUE)` . También sería correcto `mean(na.omit(Datos[[5]]))` .

# Lección 5 Matrices

Una **matriz de orden**  $n \times m$  es una tabla rectangular de números (o, en algunas situaciones específicas, algún otro tipo de datos: valores lógicos, etiquetas...) formada por  $n$  filas y  $m$  columnas. Una matriz es **cuadrada** cuando tiene el mismo número de filas que de columnas, es decir, cuando  $n = m$ ; en este caso, decimos que la matriz es de **orden**  $n$ . En matemáticas, es costumbre escribir las matrices rodeadas por paréntesis para marcar con claridad sus límites. Por ejemplo,

$$A = \begin{pmatrix} 2 & 1 & -3 \\ -4 & 3 & \sqrt{3} \end{pmatrix}$$

es una matriz  $2 \times 3$ .

## 5.1 Construcción de matrices

Disponemos de dos maneras básicas de definir una matriz en R. En primer lugar, la instrucción

```
matrix(vector , nrow=n, byrow=valor_lógico)
```

define una matriz de  $n$  filas (rows) formada por las entradas del `vector`. Si se entra `byrow=TRUE`, la matriz se construye por filas, mientras que con `byrow=FALSE` se construye por columnas; este último es el valor por defecto, por lo que no hace falta especificarlo. En vez de emplear `nrow`, se puede indicar el número de columnas con `ncol`. Veamos algunos ejemplos:

```
matrix(1:6, nrow=2)
```

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

```
matrix(1:6, nrow=3)
```

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

```
matrix(1:6, nrow=2, byrow=TRUE)
```

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

```
matrix(1:6, nrow=3, byrow=TRUE)
```

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

Observad cómo muestra R las matrices: indica las filas con `[i,]`, donde  $i$  es el índice de la fila, y las columnas con `[,j]`, donde  $j$  es el índice de la columna.

Para construir una matriz de  $n$  filas o  $n$  columnas, es conveniente que la longitud del vector al que se aplica `matrix` sea múltiplo de  $n$ . Si no es así, R rellena la última fila o columna con entradas del principio del vector y emite un mensaje de advertencia.

```
matrix(1:6, nrow=4)
```

```
## Warning in matrix(1:6, nrow = 4): data length [6] is not a sub-multiple or  
## multiple of the number of rows [4]
```

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

En particular, se puede definir una matriz constante aplicando la función `matrix` a un número. En este caso, se han de usar los parámetros `nrow` y `ncol` para especificar el orden de la matriz. Por ejemplo, la siguiente instrucción define una matriz de orden  $2 \times 3$  con todas sus entradas iguales a 1:

```
matrix(1, nrow=2, ncol=3)
```

```
##      [,1] [,2] [,3]  
## [1,]    1    1    1  
## [2,]    1    1    1
```

Otra manera posible de definir matrices es combinando filas o columnas. La instrucción

```
rbind(vector1, vector2, ....)
```

construye la matriz de filas *vector1*, *vector2*... (que han de tener la misma longitud) en este orden. Si en lugar de `rbind` se usa `cbind`, se obtiene la matriz cuyas columnas son los vectores a los que se aplica.

```
rbind(c(1,0,2),c(2,3,6),c(1,2,0))
```

```
##      [,1] [,2] [,3]
## [1,]    1    0    2
## [2,]    2    3    6
## [3,]    1    2    0
```

```
cbind(c(1,0,2),c(2,3,6),c(1,2,0))
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    1
## [2,]    0    3    2
## [3,]    2    6    0
```

Con las funciones `cbind` o `rbind` también podemos añadir columnas, o filas, a una matriz; en concreto, si  $A$  es una matriz de orden  $n \times m$  y  $v$  es un vector de longitud  $n$ , la instrucción `cbind(A, v)` define la matriz de orden  $n \times (m + 1)$  que tiene como primeras  $m$  columnas las de  $A$  y como columna  $m + 1$  el vector  $v$ ; de manera similar, `cbind(v, A)` define la matriz de orden  $n \times (m + 1)$  que tiene como primera columna el vector  $v$  y después las columnas de  $A$ . Con `cbind` también podemos concatenar por columnas dos matrices con el mismo número de filas. La instrucción `rbind` es similar a `cbind`, pero actúa por filas en vez de por columnas: permite añadir filas arriba o abajo de una matriz ya existente, y, en general, concatenar por filas dos matrices con el mismo número de columnas.

Veamos algunos ejemplos de uso de estas dos funciones.

```
A=matrix(c(1,2,3,4), nrow=2)
```

```
A
```

```
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
```

```
cbind(c(7,8),c(5,6),A)
```

```
##      [,1] [,2] [,3] [,4]  
## [1,]    7    5    1    3  
## [2,]    8    6    2    4
```

```
cbind(A,A)
```

```
##      [,1] [,2] [,3] [,4]  
## [1,]    1    3    1    3  
## [2,]    2    4    2    4
```

```
rbind(A,c(10,12))
```

```
##      [,1] [,2]  
## [1,]    1    3  
## [2,]    2    4  
## [3,]   10   12
```

```
rbind(A,A)
```

```
##      [,1] [,2]  
## [1,]    1    3  
## [2,]    2    4  
## [3,]    1    3  
## [4,]    2    4
```

Como pasaba con los vectores, todas las entradas de una matriz en R han de ser del mismo tipo de datos, y si una matriz contiene datos de diferentes tipos, automáticamente los convierte a un tipo que pueda ser común a todos ellos.



Si los vectores que concatenamos con `rbind` o `cbind` tienen nombres, las filas o columnas correspondientes de la matriz los heredan.

```
x=c(1,2,3)
y=c(0,1,-1)
rbind(x,y)
```

```
##      [,1] [,2] [,3]
## x      1      2      3
## y      0      1     -1
```

```
cbind(x,y)
```

```
##      x  y
## [1,] 1  0
## [2,] 2  1
## [3,] 3 -1
```

Se puede también poner nombres a las filas y las columnas de una matriz con la instrucción

```
dimnames(matriz)=list(vector de nombres de filas, vector de nombres de columnas)
```

Si las filas o las columnas no han de tener nombres, se declara su vector de nombres como `NULL` en esta `list` .

```
A=matrix(1:6,nrow=3)
A
```

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

```
dimnames(A)=list(c("X1", "X2", "X3"),c("Y1", "Y2"))
```

A

```
##      Y1 Y2
## X1    1  4
## X2    2  5
## X3    3  6
```

```
dimnames(A)=list(NULL,c("Palma", "Barcelona"))
```

A

```
##      Palma Barcelona
## [1,]    1          4
## [2,]    2          5
## [3,]    3          6
```

Los nombres de las filas y columnas de una matriz pueden servir para hacer más clara la información contenida en la misma.

## 5.2 Acceso a entradas y submatrices

La **entrada**  $(i, j)$  de una matriz es el elemento situado en su fila  $i$  y su columna  $j$ . Por ejemplo, las entradas  $(1,2)$  y  $(2,1)$  de la matriz

$$A = \begin{pmatrix} 2 & 1 & -3 \\ -4 & 3 & \sqrt{3} \end{pmatrix}$$

son, respectivamente, 1 y -4.

El acceso a las entradas de una matriz se realiza como en los vectores, sólo que ahora en las matrices podemos especificar la fila y la columna:

- `M[i, j]` indica la entrada  $(i, j)$  de la matriz `M`.
- `M[i, ]` indica la fila  $i$ -ésima de `M`.
- `M[, j]` indica la columna  $j$ -ésima de `M`.

En los dos últimos casos, el resultado es un vector. Si queremos que el resultado sea una matriz de una sola fila o de una sola columna, respectivamente, tenemos que añadir el parámetro `drop=FALSE` dentro de los corchetes.

```
M=matrix(c(1,3,5,2,3,6,2,9,8,4,2,5), nrow=3, byrow=TRUE)
M
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    3    5    2
## [2,]    3    6    2    9
## [3,]    8    4    2    5
```

```
M[2,4] #Entrada (2,4)
```

```
## [1] 9
```

```
M[3,1] #Entrada (3,1)
```

```
## [1] 8
```

```
M[1, ] #Fila 1
```

```
## [1] 1 3 5 2
```

```
M[1, , drop=FALSE] #ATENCIÓN: fijaos en las dos comas
```

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

```
M[ ,3] #Columna 3
```

```
## [1] 5 2 2
```

```
M[ ,3, drop=FALSE]
```

```
##      [,1]  
## [1,]    5  
## [2,]    2  
## [3,]    2
```

Estas construcciones sirven también para definir submatrices, y no sólo entradas, filas o columnas. Naturalmente, para indicar más de una fila o más de una columna tenemos que usar vectores de índices.

```
M[c(1,2),c(1,3)] #Submatriz de filas 1, 2 y columnas 1, 3
```

```
##      [,1] [,2]  
## [1,]    1    5  
## [2,]    3    2
```

```
M[c(1,3), ] #Submatriz de filas 1, 3 y todas las columnas
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    3    5    2
## [2,]    8    4    2    5
```

```
M[, c(2,3,4)] #Submatriz de columnas 2, 3, 4 y todas las filas
```

```
##      [,1] [,2] [,3]
## [1,]    3    5    2
## [2,]    6    2    9
## [3,]    4    2    5
```

Si las filas o las columnas de una matriz tienen nombres, se pueden usar para especificar trozos de la misma.

```
A=matrix(1:9,nrow=3)
dimnames(A)=list(c("X1","X2","X3"),c("Y1","Y2","Y3"))
```

A

```
##      Y1 Y2 Y3
## X1    1  4  7
## X2    2  5  8
## X3    3  6  9
```

```
A[c(1,3),2]
```

```
## X1 X3
##  4  6
```

```
A[c("X1", "X3"), "Y2"]
```

```
## X1 X3
##  4  6
```

```
A[c("X1", "X3"), c("Y1", "Y2")]
```

```
##      Y1 Y2
## X1   1  4
## X3   3  6
```

La diagonal principal de una matriz cuadrada (la que va de la esquina superior izquierda a la esquina inferior derecha) se obtiene con la función `diag` . Si la matriz no es cuadrada, `diag` produce el vector de entradas (1, 1), (2, 2) ... hasta que se para en la última fila o la última columna.

```
A=matrix(1:9, nrow=3, byrow=TRUE)
A
```

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

```
diag(A)
```

```
## [1] 1 5 9
```

```
B=matrix(1:10, nrow=2, byrow=TRUE)
```

```
B
```

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

```
diag(B)
```

```
## [1] 1 7
```

## 5.3 Algunas funciones para matrices

Las **dimensiones** de una matriz, es decir, sus números de filas y de columnas, se obtienen con las funciones `nrow` y `ncol`, respectivamente. Si queremos un vector formado por las dos dimensiones, podemos emplear la función `dim`.

```
X=matrix(c(1,2,4,3,5,1,4,6,7,1,6,4), byrow=TRUE, nrow=2)
```

```
X
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    1    2    4    3    5    1
## [2,]    4    6    7    1    6    4
```

```
nrow(X)
```

```
## [1] 2
```

```
ncol(X)
```

```
## [1] 6
```

```
dim(X)
```

```
## [1] 2 6
```

La mayoría de las funciones numéricas para vectores se pueden aplicar a matrices. Por ejemplo, podemos usar las funciones `sum`, `prod` o `mean` para obtener la suma, el producto o la media, respectivamente, de *todas* las entradas de una matriz.

```
A=matrix(c(1,2,1,3,-1,3), nrow=2, byrow=TRUE)
```

```
A
```

```
##      [,1] [,2] [,3]  
## [1,]    1    2    1  
## [2,]    3   -1    3
```

```
sum(A)
```

```
## [1] 9
```

```
mean(A)
```

```
## [1] 1.5
```

En estadística a veces es necesario calcular la suma o la media por filas o por columnas de una matriz. Esto se puede llevar a cabo con las instrucciones siguientes:



- `colSums` produce un vector con las sumas de las columnas.
- `rowSums` produce un vector con las sumas de las filas.
- `colMeans` produce un vector con las medias de las columnas.
- `rowMeans` produce un vector con las medias de las filas.

```
A=rbind(c(1,2,3,2),c(2,5,3,1),c(4,1,2,4))
```

```
A
```

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

```
colSums(A)  #Sumas de columnas
```

```
## [1]  7  8  8  7
```

```
rowSums(A)  #Sumas de filas
```

```
## [1]  8 11 11
```

```
colMeans(A)  #Medias de columnas
```

```
## [1] 2.333333 2.666667 2.666667 2.333333
```

```
rowMeans(A)  #Medias de filas
```

```
## [1] 2.00 2.75 2.75
```

Si queremos aplicar otras funciones a las filas o las columnas de una matriz, podemos emplear la función `apply`. Su estructura básica es

```
apply(A, MARGIN=..., FUN=función)
```

donde `A` es una matriz, la `función` es la que queremos aplicar, y el valor de `MARGIN` ha de ser `1` si la queremos aplicar por filas, `2` si la queremos aplicar por columnas, o `c(1, 2)` si la queremos aplicar entrada a entrada; como pasaba con los vectores, en muchas ocasiones podemos aplicar una función a todas las entradas de una matriz entrando la matriz en su argumento, pero a veces es necesario usar `apply` con `MARGIN=c(1,2)`.

Por ejemplo, vamos a calcular la **norma euclídea** de las filas de la matriz  $A$  anterior (la raíz cuadrada de la suma de los cuadrados de sus entradas):

```
f=function(x){sqrt(sum(x^2))}  
apply(A, MARGIN=1, FUN=f)
```

```
## [1] 4.242641 6.244998 6.082763
```

Vamos a ordenar cada columna de la matriz  $A$ :

```
A_ord=apply(A, MARGIN=2, FUN=sort) #Matriz con cada columna de A ordenada  
A_ord
```

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

Finalmente, vamos a calcular la matriz de raíces cuadradas de las entradas de  $A$  sin usar `apply` y usándola:

```
sqrt(A)
```

```
##           [,1]      [,2]      [,3]      [,4]
## [1,] 1.000000 1.414214 1.732051 1.414214
## [2,] 1.414214 2.236068 1.732051 1.000000
## [3,] 2.000000 1.000000 1.414214 2.000000
```

```
apply(A, MARGIN=c(1,2), FUN=sqrt)
```

```
##           [,1]      [,2]      [,3]      [,4]
## [1,] 1.000000 1.414214 1.732051 1.414214
## [2,] 1.414214 2.236068 1.732051 1.000000
## [3,] 2.000000 1.000000 1.414214 2.000000
```

## 5.4 Cálculo matricial

Las operaciones algebraicas usuales con matrices numéricas se indican de la manera siguiente:

- La **traspuesta** se obtiene con la función `t`.
- La **suma** de matrices se indica con el signo usual `+`.
- El **producto por un escalar** de una matriz se indica con el signo usual `*`.
- El **producto** de matrices se indica con `%*%`.

**¡Atención!** Si *multiplicáis* dos matrices con el signo `*`, no obtenéis el producto de las dos matrices, sino la matriz que tiene en cada entrada  $(i, j)$  el producto de las entradas  $(i, j)$  de cada una de las dos matrices. Esto a veces es útil, pero no es el producto de matrices. De manera similar, si  $M$  es una matriz y entráis `M^n`, el resultado no es la potencia  $n$ -ésima de  $M$ , sino la matriz que tiene en cada entrada la potencia  $n$ -ésima de la entrada correspondiente de  $M$ . De nuevo, esto a veces es útil, pero muy pocas veces coincide con la potencia  $n$ -ésima de  $M$ .

Veamos algunos ejemplos de operaciones matriciales:

```
A=matrix(c(1,2,1,3), nrow=2, byrow=TRUE)
```

A

```
##      [,1] [,2]
## [1,]    1    2
## [2,]    1    3
```

```
B=matrix(c(-2,4,3,1,0,2), nrow=3, byrow=TRUE)
```

B

```
##      [,1] [,2]
## [1,]   -2    4
## [2,]    3    1
## [3,]    0    2
```

```
C=matrix(c(1,0,1,2,1,0), nrow=2, byrow=TRUE)
```

C

```
##      [,1] [,2] [,3]
## [1,]    1    0    1
## [2,]    2    1    0
```

```
t(B) #Traspuesta
```

```
##      [,1] [,2] [,3]
## [1,]   -2    3    0
## [2,]    4    1    2
```

```
t(B)+C #Suma
```

```
##      [,1] [,2] [,3]
## [1,]  -1   3   1
## [2,]   6   2   2
```

5\*A *#Producto por escalar*

```
##      [,1] [,2]
## [1,]    5  10
## [2,]    5  15
```

C\*\*B *#Producto*

```
##      [,1] [,2]
## [1,]  -2   6
## [2,]  -1   9
```

(C\*\*B)\*\*A *#Producto*

```
##      [,1] [,2]
## [1,]    4  14
## [2,]    8  25
```

A^2 *#Esto no es elevar al cuadrado*

```
##      [,1] [,2]
## [1,]    1   4
## [2,]    1   9
```

```
A%%A #Esto sí
```

```
##      [,1] [,2]
## [1,]    3    8
## [2,]    4   11
```

Al multiplicar matrices por vectores, R trata por defecto estos últimos como vectores columna, pero si en alguna situación concreta la manera natural de entender un vector es como vector fila, lo hace sin ningún reparo. Veamos algunos ejemplos:

```
A=rbind(c(1,2),c(3,4))
A
```

```
##      [,1] [,2]
## [1,]    1    2
## [2,]    3    4
```

```
v=c(5,6)
```

El producto

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \cdot \begin{pmatrix} 5 \\ 6 \end{pmatrix}$$

se obtiene mediante

```
A%%v
```

```
##      [,1]
## [1,]   17
## [2,]   39
```

El producto

$$\begin{pmatrix} 5 & 6 \end{pmatrix} \cdot \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

se obtiene mediante

```
v**%A
```

```
##      [,1] [,2]  
## [1,]    23    34
```

El producto

$$\begin{pmatrix} 5 & 6 \end{pmatrix} \cdot \begin{pmatrix} 5 \\ 6 \end{pmatrix}$$

se obtiene mediante

```
v**%v
```

```
##      [,1]  
## [1,]    61
```

El producto

$$\begin{pmatrix} 5 \\ 6 \end{pmatrix} \cdot \begin{pmatrix} 5 & 6 \end{pmatrix}$$

se obtiene mediante

```
v**%t(v)
```

```
##           [,1] [,2]
## [1,]      25  30
## [2,]      30  36
```

La versión básica de R no lleva ninguna función para calcular potencias de matrices, y hay que cargar algún paquete adecuado para disponer de ella. Por ejemplo, el paquete `expm` dispone de la operación `%^%`. No obstante, hay que tener en cuenta que esta función, u otras parecidas, no calculan las potencias de manera exacta, sino que emplean algoritmos de cálculo numérico para aproximarlas a cambio de calcularlas rápido, y por lo tanto no siempre dan el resultado exacto.

Por ejemplo, para calcular

$$\begin{pmatrix} 1 & 2 \\ 1 & 3 \end{pmatrix}^{20}$$

podemos entrar

```
A=matrix(c(1,2,1,3), nrow=2, byrow=TRUE)
library(expm)
A%^%20
```

```
##           [,1]      [,2]
## [1,] 58063278153 158631825968
## [2,] 79315912984 216695104121
```

El determinante de una matriz cuadrada se calcula con la función `det`.

```
Y=rbind(c(1,3,2),c(2,3,5),c(-1,3,2))
Y
```



```
##      [,1] [,2] [,3]
## [1,]    1    3    2
## [2,]    2    3    5
## [3,]   -1    3    2
```

```
det(Y)
```

```
## [1] -18
```

Para ganar en rapidez, R calcula los determinantes usando un método numérico que a veces produce efectos no deseados como el siguiente:

```
A=matrix(c(3,10,30,100), nrow=2)
```

```
A
```

```
##      [,1] [,2]
## [1,]    3   30
## [2,]   10  100
```

```
det(A)
```

```
## [1] 3.552714e-14
```

Pero, de hecho,

$$\begin{vmatrix} 3 & 30 \\ 10 & 100 \end{vmatrix} = 3 \cdot 100 - 30 \cdot 10 = 0.$$

Por lo tanto, este determinante  $3.552714 \cdot 10^{-14}$  es en realidad 0.

El rango de una matriz  $A$  se puede calcular mediante la instrucción `qr(A)$rank` .

```
X=matrix(c(0,1,0,-7,3,-1,16,-3,4), nrow=3, byrow=TRUE)
```

```
X
```

```
##      [,1] [,2] [,3]
## [1,]    0    1    0
## [2,]   -7    3   -1
## [3,]   16   -3    4
```

```
det(X)
```

```
## [1] 12
```

```
qr(X)$rank
```

```
## [1] 3
```

```
Y=rbind(rep(0,3),rep(1,3))
```

```
Y
```

```
##      [,1] [,2] [,3]
## [1,]    0    0    0
## [2,]    1    1    1
```

```
qr(Y)$rank
```

```
## [1] 1
```

Podemos calcular la inversa de una matriz invertible con la instrucción `solve` . Por ejemplo, para calcular la inversa  $A^{-1}$  de la matriz

$$A = \begin{pmatrix} 1 & 3 & 4 \\ 0 & 2 & -1 \\ 2 & 1 & 2 \end{pmatrix}$$

podemos entrar lo siguiente:

```
A=matrix(c(1,3,4,0,2,-1,2,1,2), nrow=3, byrow=TRUE)
solve(A)
```

```
##           [,1]      [,2]      [,3]
## [1,] -0.2941176  0.1176471  0.64705882
## [2,]  0.1176471  0.3529412 -0.05882353
## [3,]  0.2352941 -0.2941176 -0.11764706
```

Obtenemos

$$A^{-1} = \begin{pmatrix} -0.2941176 & 0.1176471 & 0.64705882 \\ 0.1176471 & 0.3529412 & -0.05882353 \\ 0.2352941 & -0.2941176 & -0.11764706 \end{pmatrix}.$$

Comprobemos si esta matriz es realmente la inversa de  $A$ :

```
A%%solve(A)
```

```
##           [,1] [,2]      [,3]
## [1,] 1.000000e+00  0  0.000000e+00
## [2,] 5.551115e-17  1 -2.775558e-17
## [3,] 0.000000e+00  0  1.000000e+00
```

```
solve(A)%%A
```

```
##      [,1]      [,2]      [,3]
## [1,]      1 0.000000e+00 0.000000e+00
## [2,]      0 1.000000e+00 1.110223e-16
## [3,]      0 2.775558e-17 1.000000e+00
```

Los productos `A*%solve(A)` y `solve(A)*%A` no han dado exactamente la matriz identidad, como deberían, pero la diferencia está en la decimosexta cifra decimal. Recordad que R no trabaja con precisión infinita, a veces los errores de redondeo son inevitables.

La función `solve` también sirve para resolver sistemas de ecuaciones lineales

$$\left. \begin{array}{l} a_{1,1}x_1 + \cdots + a_{1,n}x_n = b_1 \\ a_{2,1}x_1 + \cdots + a_{2,n}x_n = b_2 \\ \vdots \\ a_{n,1}x_1 + \cdots + a_{n,n}x_n = b_n \end{array} \right\}$$

cuya **matriz del sistema**

$$A = \begin{pmatrix} a_{1,1} & \cdots & a_{1,n} \\ \vdots & \ddots & \vdots \\ a_{n,1} & \cdots & a_{n,n} \end{pmatrix}$$

sea cuadrada e invertible. Para ello, se usaría la instrucción

```
solve(A, b)
```

donde `A` es la matriz  $A$  del sistema y `b` es el vector de términos independientes

$$b = (b_1, \dots, b_n).$$

Por ejemplo, para resolver el sistema

$$\left. \begin{array}{l} x + 6y - 3z = 7 \\ 2x - y + z = 2 \\ x + y - z = 3 \end{array} \right\},$$

que escrito en forma matricial es

$$\begin{pmatrix} 1 & 6 & -3 \\ 2 & -1 & 1 \\ 1 & 1 & -1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 7 \\ 2 \\ 3 \end{pmatrix},$$

podemos entrar

```
A=matrix(c(1,6,-3,2,-1,1,1,1,-1), nrow=3, byrow=TRUE)
```

```
A
```

```
##      [,1] [,2] [,3]
## [1,]    1    6   -3
## [2,]    2   -1    1
## [3,]    1    1   -1
```

```
b=c(7,2,3)
solve(A, b)
```

```
## [1]  1.6666667  0.4444444 -0.8888889
```

y obtenemos que la solución del sistema (redondeada a 7 cifras decimales) es

$$x = 1.6666667, y = 0.4444444, z = -0.8888889.$$

## 5.5 Valores y vectores propios

La función básica para calcular valores y vectores propios es `eigen`, por el hecho que, en inglés, los valores y vectores propios se llaman `eigenvalues` y `eigenvectors`, respectivamente. (Si necesitáis repasar las definiciones de vector y valor propio de una matriz y de descomposición canónica de una matriz diagonalizable, y por qué son importantes, podéis consultar las entradas correspondientes de la *Wikipedia*:

[http://es.wikipedia.org/wiki/Vector\\_propio\\_y\\_valor\\_propio](http://es.wikipedia.org/wiki/Vector_propio_y_valor_propio) y  
[http://es.wikipedia.org/wiki/Matriz\\_diagonalizable](http://es.wikipedia.org/wiki/Matriz_diagonalizable).)

Supongamos, por ejemplo, que queremos calcular los valores propios de la matriz

$$A = \begin{pmatrix} 2 & 6 & -8 \\ 0 & 6 & -3 \\ 0 & 2 & 1 \end{pmatrix}.$$

```
A=matrix(c(2,6,-8,0,6,-3,0,2,1), nrow=3, byrow=TRUE)
```

```
A
```

```
##      [,1] [,2] [,3]
## [1,]    2    6   -8
## [2,]    0    6   -3
## [3,]    0    2    1
```

```
eigen(A)
```

```
## eigen() decomposition
## $values
## [1] 4 3 2
##
## $vectors
##      [,1]      [,2] [,3]
## [1,] 0.2672612 -0.8164966    1
## [2,] 0.8017837  0.4082483    0
## [3,] 0.5345225  0.4082483    0
```

El resultado de `eigen` es una `list` con dos objetos: `values` y `vectors` .

```
str(eigen(A))
```

```
## List of 2
## $ values : num [1:3] 4 3 2
## $ vectors: num [1:3, 1:3] 0.267 0.802 0.535 -0.816 0.408 ...
## - attr(*, "class")= chr "eigen"
```

```
eigen(A)$values
```

```
## [1] 4 3 2
```

```
eigen(A)$vectors
```

```
##           [,1]      [,2] [,3]
## [1,] 0.2672612 -0.8164966    1
## [2,] 0.8017837  0.4082483    0
## [3,] 0.5345225  0.4082483    0
```

El objeto `values` es un vector con los valores propios, y el objeto `vectors` es una matriz cuyas columnas son vectores propios: la primera columna es un vector propio del primer valor propio del vector `values`, la segunda lo es del segundo, y así sucesivamente. De este modo, del resultado anterior deducimos que los valores propios de  $A$  son 2, 3 y 4 y que

$$\begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} -0.8164966 \\ 0.4082483 \\ 0.4082483 \end{pmatrix}, \begin{pmatrix} 0.2672612 \\ 0.8017837 \\ 0.5345225 \end{pmatrix}$$

son vectores propios de  $A$  de valores propios 2, 3 y 4, respectivamente (o, para ser precisos, los dos últimos son vectores propios de valor propio 3 y 4 redondeados a 7 cifras decimales).

Es importante tener en cuenta algunas propiedades de la función `eigen`:

- Da los valores propios en orden decreciente de su valor absoluto (o de su módulo, si hay valores propios complejos) y repetidos tantas veces como su multiplicidad.
- Si hay algún valor propio con multiplicidad mayor que 1, da tantos vectores de este valor propio como su multiplicidad. Además, en este caso procura que estos vectores propios sean linealmente independientes. Por lo tanto, cuando da vectores propios

repetidos de algún valor propio es porque para este valor propio no existen tantos vectores propios linealmente independientes como su multiplicidad y, por consiguiente, la matriz no es diagonalizable.

Del resultado de `eigen(A)` se puede obtener una **descomposición canónica**

$$A = P \cdot D \cdot P^{-1}$$

de una matriz diagonalizable  $A$ : basta tomar como  $D$  la matriz diagonal que tiene como diagonal principal el vector `eigen(A)$values` y como  $P$  la matriz `eigen(A)$vectors`.

Para construir una matriz diagonal cuya diagonal principal sea un `vector` dado, podemos usar la instrucción `diag(vector)`. Si aplicamos `diag` a un número  $n$ , produce la matriz identidad de orden  $n$ .

```
diag(c(2,5,-1))
```

```
##      [,1] [,2] [,3]
## [1,]    2    0    0
## [2,]    0    5    0
## [3,]    0    0   -1
```

```
diag(3)
```

```
##      [,1] [,2] [,3]
## [1,]    1    0    0
## [2,]    0    1    0
## [3,]    0    0    1
```

La función `diag` ya había salido en la Sección [5.2](#): si se aplica a una matriz, se obtiene el vector formado por sus entradas (1,1), (2,2) ...; ahora vemos que si se aplica a un vector, produce una matriz diagonal.

```
B=matrix(1:10, nrow=2, byrow=TRUE)
```

```
B
```



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

```
diag(B)
```

```
## [1] 1 7
```

```
diag(diag(B))
```

```
##      [,1] [,2]
## [1,]    1    0
## [2,]    0    7
```

Veamos un ejemplo de uso de `eigen` para calcular una descomposición canónica. Como hemos visto, la matriz

$$A = \begin{pmatrix} 2 & 6 & -8 \\ 0 & 6 & -3 \\ 0 & 2 & 1 \end{pmatrix}$$

es de orden 3 y tiene sus tres valores propios diferentes. Por lo tanto, es diagonalizable y las matrices de una descomposición canónica son las siguientes:

- La matriz diagonal  $D$  de valores propios es

```
D=diag(eigen(A)$values)
```

```
D
```

```
##      [,1] [,2] [,3]
## [1,]    4    0    0
## [2,]    0    3    0
## [3,]    0    0    2
```

- La matriz  $P$  de vectores propios es

```
P=eigen(A)$vectors
P
```

```
##      [,1]      [,2] [,3]
## [1,] 0.2672612 -0.8164966    1
## [2,] 0.8017837  0.4082483    0
## [3,] 0.5345225  0.4082483    0
```

Por consiguiente, una descomposición canónica de  $A$  es (redondeando)

$$A = \begin{pmatrix} 0.2673 & -0.8165 & 1 \\ 0.8019 & 0.4082 & 0 \\ 0.5345 & 0.4082 & 0 \end{pmatrix} \cdot \begin{pmatrix} 4 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 2 \end{pmatrix} \cdot \begin{pmatrix} 0.2673 & -0.8165 & 1 \\ 0.8019 & 0.4082 & 0 \\ 0.5345 & 0.4082 & 0 \end{pmatrix}^{-1}.$$

Comprobemos que, efectivamente,  $A = P \cdot D \cdot P^{-1}$ :

```
P%%D%%solve(P)
```

```
##      [,1] [,2] [,3]
## [1,]    2    6   -8
## [2,]    0    6   -3
## [3,]    0    2    1
```

```
A
```

```
##           [,1] [,2] [,3]
## [1,]      2    6   -8
## [2,]      0    6   -3
## [3,]      0    2    1
```

Veamos otro ejemplo. Queremos decidir si la matriz

$$B = \begin{pmatrix} 0 & 1 & 0 \\ -7 & 3 & -1 \\ 16 & -3 & 4 \end{pmatrix}$$

es diagonalizable y, en caso afirmativo, obtener una descomposición canónica.

```
B=matrix(c(0,1,0,-7,3,-1,16,-3,4), nrow=3, byrow=TRUE)
eigen(B)
```

```
## eigen() decomposition
## $values
## [1] 3 2 2
##
## $vectors
##           [,1]      [,2]      [,3]
## [1,] -0.1301889 -0.1825742 -0.1825742
## [2,] -0.3905667 -0.3651484 -0.3651484
## [3,]  0.9113224  0.9128709  0.9128709
```

Da dos veces el mismo vector propio de valor propio 2. Esto significa que  $B$  no tiene dos vectores propios linealmente independientes de este valor propio, y, por lo tanto, no es diagonalizable.

## 5.6 Matrices complejas (opcional)

La mayoría de las instrucciones explicadas en esta lección para operar con matrices numéricas sirven sin ningún cambio para operar con matrices de entradas números complejos. Por ejemplo, para elevar al cuadrado la matriz

$$\begin{pmatrix} 3 - 2i & 5 + 3 \\ 1 + 2i & 2 - i \end{pmatrix},$$

podemos entrar:

```
A=matrix(c(3-2i,5+3i,1+2i,2-1i), nrow=2, byrow=TRUE)
```

```
A
```

```
##      [,1] [,2]  
## [1,] 3-2i 5+3i  
## [2,] 1+2i 2-1i
```

```
A%%A
```

```
##      [,1] [,2]  
## [1,] 4+1i 34+0i  
## [2,] 11+7i 2+9i
```

Para calcular sus valores y vectores propios, podemos entrar:

```
eigen(A)
```

```
## eigen() decomposition
## $values
## [1] 4.902076+1.101916i 0.097924-4.101916i
##
## $vectors
##           [,1]           [,2]
## [1,] 0.8483705+0.000000i 0.8519823+0.000000i
## [2,] 0.4695014+0.244614i -0.5216168-0.045189i
```

Y para resolver el sistema de ecuaciones

$$\left. \begin{aligned} (3 - 2i)x + (5 + 3i)y &= 2 - i \\ (1 + 2i)x + (2 - i)y &= 3 \end{aligned} \right\}$$

podemos entrar:

```
A=matrix(c(3-2i,5+3i,1+2i,2-1i), nrow=2, byrow=TRUE)
b=c(2-1i,3)
solve(A, b)
```

```
## [1] 0.4705882-0.7176471i 0.4823529+0.1294118i
```

La excepción más importante son los determinantes.

```
det(A)
```

```
## Error in determinant.matrix(x, logarithm, ...): 'determinant' not currently defi
```

Pero resulta que el determinante de una matriz es igual al producto de sus valores propios, incluyendo repeticiones. Por lo tanto, para calcular el determinante de una matriz compleja

$A$  podemos usar `prod(eigen(A)$values)` .

```
A=matrix(c(3-2i, 5+3i, 1+2i, 2-1i), nrow=2, byrow=TRUE)
prod(eigen(A)$values)
```

```
## [1] 5-20i
```

## 5.7 Guía rápida de funciones

- `matrix` sirve para construir una matriz a partir de un vector. Algunos parámetros importantes:
  - `byrow` : un parámetro lógico para indicar si la matriz se construye por filas (igualado a `TRUE` ) o por columnas (valor por defecto).
  - `nrow` : el número de filas.
  - `ncol` : el número de columnas.
- `cbind` concatena vectores y matrices por columnas.
- `rbind` concatena vectores y matrices por filas.
- `dimnames` permite poner nombres a las filas y las columnas de una matriz.
- `matriz[...]` se usa para especificar un elemento, una fila, una columna o una submatriz de la `matriz` . Si extraemos una fila o una columna con el parámetro `drop=FALSE` , el resultado es una matriz y no un vector.
- `diag` tiene dos usos:
  - aplicada a un vector, construye una matriz diagonal
  - aplicada a una matriz, extrae su diagonal principal.
- `nrow` da el número de filas de una matriz.
- `ncol` da el número de columnas de una matriz.
- `dim` da un vector con las dimensiones de una matriz.
- `sum` calcula la suma de las entradas de una matriz.
- `prod` calcula el producto de las entradas de una matriz.
- `mean` calcula la media aritmética de las entradas de una matriz.
- `colSums` y `rowSums` calculan, respectivamente, las sumas de las entradas de cada una de las columnas y de cada una de las filas de una matriz.
- `colMeans` y `rowMeans` calculan, respectivamente, las medias de cada una de las columnas y de cada una de las filas de una matriz.

- `apply(matriz, MARGIN=..., FUN=función)` aplica la `función` a las filas (`MARGIN=1`), a las columnas (`MARGIN=2`) o a todas las entradas (`MARGIN=c(1, 2)`) de la `matriz`.
- Signos de operaciones del álgebra matricial:
  - `+` : Suma
  - `*` : Producto por escalar
  - `%%` : Producto de matrices
  - `^%` del paquete `expm` : Potencia
  - `t` : Traspuesta
- `det` calcula el determinante de una matriz.
- `qr(matriz)$rank` calcula el rango de la `matriz`.
- `solve`, aplicada a una matriz invertible  $A$ , calcula su inversa  $A^{-1}$ , y aplicada a una matriz invertible  $A$  y un vector  $b$ , calcula  $A^{-1} \cdot b$ .
- `eigen` calcula los valores y vectores propios de una matriz. El resultado es una `list` con dos componentes:
  - `values` : un vector con los valores propios.
  - `vectors` : una matriz cuyas columnas son vectores propios de los correspondientes valores propios.

## 5.8 Ejercicios

### Test

(1) Dad una instrucción, empleando la función `matrix` con el parámetro `nrow`, que

construya por filas la matriz  $\begin{pmatrix} 1 & 5 & 3 \\ 2 & 3 & 9 \end{pmatrix}$ .

(2) Dad una instrucción que use la función `rbind` para construir la matriz  $\begin{pmatrix} 1 & 5 & 3 \\ 2 & 3 & 9 \end{pmatrix}$ .

(3) Dad la instrucción que indica la entrada (2,3) de una matriz llamada M.

(4) Dad la instrucción que da una matriz de una sola columna formada por la octava columna de una matriz llamada M.

(5) Emplead la función `diag` para construir la matriz diagonal con diagonal principal  $(1, -1, 3, 4)$ .

(6) Dad el valor del determinante de la matriz

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 8 & 7 & 6 \\ 5 & 4 & 3 & 2 \end{pmatrix}.$$

(7) Dad la entrada (2,2) de  $A \cdot (A + A^t) \cdot A$ , donde  $A = \begin{pmatrix} 1 & 3 \\ 2 & 4 \end{pmatrix}$ . Si no existe, tenéis que responder NO.

(8) Dad la entrada (2,3) de la matriz

$$\begin{pmatrix} 1 & 5 & 3 \\ 2 & 3 & 9 \\ 4 & 1 & 1 \end{pmatrix}^{-1}$$

como un número real redondeado a 4 cifras decimales. Si no existe, tenéis que responder NO.

(9) Dad el valor de  $y$ , redondeado a 4 cifras decimales, en la solución del sistema

$$\left. \begin{array}{lcl} x + 5y + 3z & = & 1 \\ 2x + 3y + 9z & = & 1 \\ 4x + y + z & = & 1 \end{array} \right\}$$

Si no existe o no es único, tenéis que contestar NO.

(10) Dad los valores propios, separados por un espacio en blanco, en orden decreciente de su valor absoluto y repetidos tantas veces como su multiplicidad, de la matriz

$$\begin{pmatrix} 2 & 4 & -6 \\ 0 & 0 & 3 \\ 0 & -2 & 5 \end{pmatrix}.$$

(11) Dad, redondeado a 3 cifras decimales, el vector propio de valor propio 4 de la matriz

$$\begin{pmatrix} -48 & 35 & -12 \\ -134 & 95 & -32 \\ -194 & 133 & -44 \end{pmatrix}$$

que da R. Dad sus entradas separadas por exactamente un espacio en blanco. Si no existe, responded NO.



(12) Dad el rango de la matriz

$$\begin{pmatrix} -2 & -8 & -2 & 3 \\ -3 & -6 & -1 & 2 \\ -9 & -22 & -3 & 7 \\ -18 & -44 & -8 & 15 \end{pmatrix}.$$

## Ejercicio

Sean  $x$  e  $y$  dos cantidades de las cuales efectuamos una serie de  $k$  observaciones conjuntas, de forma que obtenemos una secuencia de pares de valores

$$(x_1, y_1), (x_2, y_2), \dots, (x_k, y_k).$$

Como veíamos en la Lección 3, si queremos encontrar la recta  $y = ax + b$  que aproxime mejor estas observaciones, una posibilidad es calcular los valores  $a, b \in \mathbb{R}$  tales que

$$\sum_{i=1}^k (ax_i + b - y_i)^2$$

sea mínimo. De este modo encontraríamos la **recta de regresión por mínimos cuadrados**. Resulta (no lo demostraremos aquí) que los coeficientes  $a$  y  $b$  de esta recta de regresión se obtienen por medio de la fórmula

$$\begin{pmatrix} b \\ a \end{pmatrix} = (D_2 \cdot D_2^t)^{-1} \cdot D_2 \cdot w,$$

donde

$$w = \begin{pmatrix} y_1 \\ \vdots \\ y_k \end{pmatrix}, \quad D_2 = \begin{pmatrix} 1 & 1 & \dots & 1 \\ x_1 & x_2 & \dots & x_k \end{pmatrix}.$$

- Calculad de este modo los valores de  $a$  y  $b$  cuando las observaciones son las de la Tabla 5.1 (es la Tabla 3.1 de la Lección 3), y comprobad que obtenéis el mismo resultado que obteníamos en su momento con la función `lm`.

Tabla 5.1: Alturas medias de niños por edad.

edad (años)	altura (cm)
1	76.11
2	86.45
3	95.27
5	109.18
7	122.03
9	133.73
11	143.73
13	156.41

De manera similar, si queremos obtener una función cuadrática  $y = ax^2 + bx + c$  que aproxime los pares  $(x_i, y_i)_{i=1, \dots, k}$ , podemos buscar los coeficientes  $a, b, c$  que minimicen el valor de

$$\sum_{i=1}^k (ax_i^2 + bx_i + c - y_i)^2.$$

Estos coeficientes se obtienen de manera similar, por medio de la fórmula

$$\begin{pmatrix} c \\ b \\ a \end{pmatrix} = (D_3 \cdot D_3^t)^{-1} \cdot D_3 \cdot w,$$

donde  $w$  es como antes y ahora

$$D_3 = \begin{pmatrix} 1 & 1 & \dots & 1 \\ x_1 & x_2 & \dots & x_k \\ x_1^2 & x_2^2 & \dots & x_k^2 \end{pmatrix}.$$

- b. Calculad los valores de  $a, b, c$  para los pares  $(x, y)$  de la Tabla 5.1.
- c. Con R, podemos calcular estos coeficientes de la manera siguiente: si definimos un nuevo vector  $z$  con los cuadrados de los valores de  $x$ , y aplicamos la función

```
lm(y~x+z)
```

obtenemos los coeficientes de la función lineal  $y = c + bx + az$  que mejor se ajusta a las ternas  $(x_i, z_i, y_i)$ , en el sentido de que minimiza la suma de las diferencias al cuadrado entre los valores  $y_i$  y los correspondientes  $c + bx_i + az_i$ ; substituyendo entonces  $z$  por  $x^2$  en esta relación, obtenemos la función  $y = c + bx + ax^2$  que buscábamos.

Comprobad que, efectivamente, coincide con la calculada en el apartado anterior.

## Respuestas al test

(1) `matrix(c(1,5,3,2,3,9),nrow=2,byrow=TRUE)`

(2) `rbind(c(1,5,3),c(2,3,9))`

(3) `M[2,3]`

(4) `M[,8,drop=FALSE]`

(5) `diag(c(1,-1,3,4))`

(6) 0

(7) 240

(8) -0.0224

(9) 0.1493

(10) 3 2 2

(11) 0.371 0.743 0.557

(12) 3