Capítulo 1

Introducción a R y Rstudio

1.1. Introducción.

R (https://www.r-project.org) es un entorno y lenguaje de programación con un enfoque al análisis estadístico que puede analizar cualquier tipo de datos y una de cuyas ventajas principales es que es compatible con la mayoría de los formatos de datos, *csv, xls, sav, sas,* etc.

R es un lenguaje Orientado a Objetos y significa que las variables, datos, funciones, resultados, etc., se guardan en la memoria activa del ordenador en forma de objetos con un nombre específico. El usuario puede modificar o manipular estos objetos con operadores (aritméticos, lógicos, y comparativos) y funciones (que a su vez son objetos). El usuario ejecuta las funciones con la ayuda de comandos definidos y los resultados se pueden, visualizar directamente en la pantalla, guardar en un objeto o escribir directamente en el disco (particularmente para gráficos). Además, debido a que los resultados mismos son objetos, pueden ser considerados como datos y analizados como tal.

R también es un lenguaje interpretado (como Java) y no compilado (como C, C++, Fortran, Pascal, . . .), lo cual significa que los comandos escritos en el teclado son ejecutados directamente sin necesidad de construir ejecutables.

R es una implementación de software libre del lenguaje S pero con soporte de alcance estático. Se trata de uno de los lenguajes más utilizados en investigación por la comunidad estadística, siendo además muy popular en el campo de la minería de datos, la investigación biomédica, la bioinformática y las matemáticas financieras. A esto contribuye la posibilidad de cargar diferentes bibliotecas o paquetes con funcionalidades de cálculo o gráficos.

R es parte del sistema *GNU* y se distribuye bajo la licencia *GNU GPL*. Está disponible para los sistemas operativos Windows, OSX, Unix y GNU/Linux.

Fue desarrollado inicialmente por Robert Gentleman y Ross Ihaka del Departamento de Estadística de la Universidad de Auckland en 1993. Sin embargo, si nos remontamos a sus bases iniciales, puede decirse que se gestó en los *Bell Laboratories* de AT&T y ahora denominados *Alcatel-Lucent* en Nueva Jersey con el lenguaje *S*. Éste último, un sistema para el análisis de datos desarrollado por John Chambers, Rick Becker y diferentes colaboradores desde finales de 1970. La historia, desde este punto, es prácticamente la del lenguaje *S*. Los diseñadores iniciales, Gentleman y Ihaka, combinaron las fortalezas de dos lenguajes

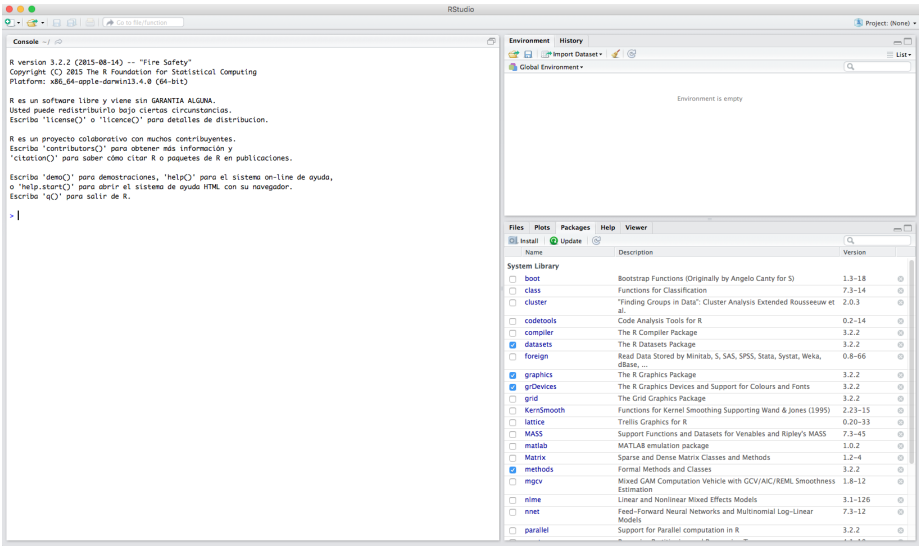


Figura 1.1: Bienvenido a Rstudio.

existentes, *S* y *Scheme*. En sus propias palabras: *El lenguaje resultante es muy similar en apariencia a S, pero en el uso de fondo y la semántica es derivado desde Scheme*. El resultado se llamó R y su desarrollo actual es responsabilidad del *R Development Core Team*.

Rstudio provee acceso a R en un navegador web. Esto tiene claras ventajas como no requerir instalación, interface amigable y el trabajo que se empieza en una máquina puede continuarse en otra máquina, posteriormente.

La URL de RStudio es **https://www.rstudio.com/** y una vez que se ejecuta nos aparece una imagen como la mostrada en la Figura 1.1.

donde se aprecia una ventana dividida en tres o cuatro paneles. Además, diversos paneles pueden a su vez subdividirse en múltiples tablas. El panel llamado *consola* es donde se teclean los comandos que R ejecuta.

1.2. Primeros pasos con R

Para obtener ayuda en R se pueden utilizar los siguientes comandos:

- help.start() → documentación online de R (HTML).
- help(func) o ?func → ayuda referente al comando *func*.
- apropos(comando, mode="function") → lista de funciones que contienen comando en el nombre.
- data() → muestra los datos de ejemplo que hay disponibles

El *workspace* es el espacio de trabajo en el que se incluyen todos los objetos definidos por el usuario y, cuando termina una sesión de R, pregunta si deseamos guardar los datos de dicha sesión. El directorio de trabajo o *working directory*

es el directorio en el cual trabaja R, donde guarda el *workspace* al finalizar la sesión y donde buscará un *workspace* guardado al inicio. Si quieres que R lea un fichero que no esté en *working directory* hay que especificar la ruta completa.

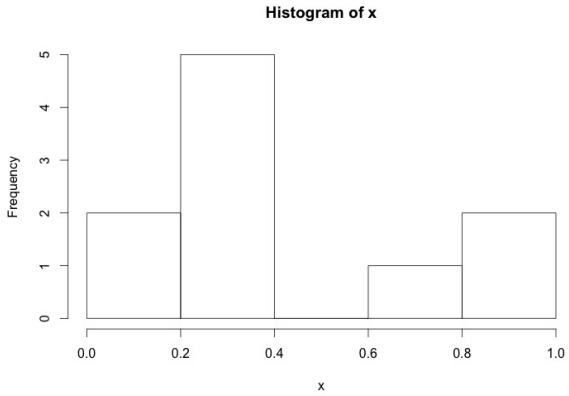
Algunos comandos interesantes para manejar y organizar el *workspace* son:

- save.image(miespacio.R) → guarda el *workspace* (en .RData).
- load(miespacio.R) → carga el *workspace* (miespacio.R).
- getwd() → muestra el *working directory*.
- setwd(otrodirectorio) → cambia el *working directory* al especificado.
- ls() o dir() → lista el contenido del *working directory*.
- history() → muestra los últimos comandos ejecutados.
- savehistory() → guarda el historial de comandos.
- loadhistory() → carga el historial de comandos.

Así por ejemplo

R 1 (Ejemplo de sesión en R).

```
> getwd() # directorio de trabajo actual
[1] "/users/Jose"

> setwd( "/users/Leandro") # nuevo directorio de trabajo

> set.seed(1) # semilla para valores entre 0 y 1

> x <- runif(10) # genera diez números aleatorios

> x

[1] 0.93470523 0.21214252 0.65167377 0.12555510
[5] 0.26722067 0.38611409 0.01339033 0.38238796
[9] 0.86969085 0.34034900

> summary(x) # resumen estadístico

Min.    1st Qu. Median    Mean   3rd Qu.  Max.
0.01339 0.22590 0.36140 0.41830 0.58530 0.93470

> hist(x) # histograma

> savehistory # guarda el historial

> save.image() # guarda la imagen
```



R considera todo lo que sigue al símbolo # como un comentario. Además, es un lenguaje de expresiones con una sintaxis simple que distingue entre mayúsculas y minúsculas, de tal modo que X y x representan dos objetos distintos. Las órdenes elementales consisten en expresiones o en asignaciones. Una expresión se evalúa, se imprime y su valor se pierde; en cambio, una asignación evalúa la expresión, no la imprime y guarda su valor en una variable. Las órdenes se separan mediante punto y coma o mediante un cambio de línea. Si al terminar la línea, la orden no está completa, R muestra un signo de continuación (+) en la línea siguiente y las sucesivas.

R permite recuperar y ejecutar órdenes previas. Las flechas verticales del teclado puede utilizarse para recorrer el historial de órdenes. Cuando se recupere una orden, se puede utilizar las flechas horizontales para desplazarse por ella, puede eliminar o añadir caracteres.

Si se tiene órdenes almacenadas en un archivo y en el directorio de trabajo, se pueden ejecutar con la función

R 2.

```
> source(‘‘ordenes.R")
```

Por otro lado, si se quiere volcar el resultado de una expresión en un archivo (en vez de en pantalla) se debe utilizar la orden

R 3.

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

La orden > sink() devuelve la salida de nuevo a la pantalla.

Las entidades que R crea y manipula se denominan objetos. Durante una sesión de trabajo con R los objetos que se crean se almacenan por nombre y para

ver los La orden objetos almacenados se puede utilizar la función > objects(). El conjunto de de objetos almacenados en cada momento se denomina **espacio de trabajo** (workspace). Para eliminar objetos puede utilizar la orden rm(), por ejemplo,

R 4.

```
> rm(x, y, z)
```

borra los objetos x, y, z.

Los objetos creados durante una sesión de R pueden almacenarse en un archivo para su uso posterior. Al finalizar la sesión, R pregunta si desea hacerlo. En caso afirmativo todos los objetos se almacenan en el archivo *.RData* en el directorio de trabajo. En la siguiente sesión se recuperarán los objetos de este archivo así como el historial de órdenes.

1.3. Uso de R como calculadora.

R constituye una calculadora con la que podemos hacer ciertas operaciones clásicas, como las que vamos a ver a continuación. Por ello esta sección aborda los operadores aritméticos, de comparación así como las funciones matemáticas más usuales ya predefinidas en R.

⊙ **Operadores Aritméticos**

Los operadores arítméticos que se pueden emplear en R son:

- Suma $+$
- Resta $-$
- Multiplicación $*$
- División $/$
- Potencia $\wedge$
- División entera $\%/\%$
- Resto o residuo $\%\%$

⊙ **Operadores de Comparación**

Los operadores de comparación que se pueden emplear en R son:

- Mayor que $>$
- Menor que $<$
- Igual que $==$
- Mayor o igual que $>=$
- Menor o igual que $<=$
- Distinto de $!=$

⊙ Funciones Matemáticas

Las funciones matemáticas predefinidas en R, se pueden englobar en cinco categorías: Funciones de Signo, Funciones Trigonométricas, Funciones Hiperbólicas, Funciones de Redondeo y Funciones Exponenciales/Logarítmicas.

1. Funciones de Signo.
   - Valor Absoluto $abs(a)$
   - Signo $sig(a)$
2. Funciones Trigonométricas.
   - Seno $sin(a)$
   - Coseno $cos(a)$
   - Tangente $tan(a)$
   - Arco-seno $asin(a)$
   - Arco-coseno $acos(a)$
   - Arco-tangente $atan(a)$
3. Funciones Hiperbólicas.
   - Seno hiperbólico $sinh(a)$
   - Coseno hiperbólico $cosh(a)$
   - Tangente hiperbólica $tanh(a)$
   - Arco-seno hiperbólico $asinh(a)$
   - Arco-coseno hiperbólico $acosh(a)$
   - Arco-tangente hiperbólica $atanh(a)$
4. Funciones de Redondeo.
   - Menor entero mayor o igual $ceiling(a)$
   - Menor entero mayor o igual $floor(a)$
   - Entero más cercano $trunc(a)$
5. Funciones Exponenciales y Logarítmicas.
   - Logaritmo en base e $log(a, base = exp(1))$
   - Logaritmo en base 10 $log10(a)$
   - Logaritmo en base 2 $log2(a)$
   - Raíz cuadrada $sqrt(a)$
   - Función exponencial $exp(a)$

R 5 (R como una calculadora)**.**

```
> 2+3
[1] 5
>log(1.2345)
[1] 0.210666
> 45.6 * 34.5
[1] 1573.2
```

Puedes conservar valores almacenándolos en variables para su posterior utilizando el operador de asignación $< -$. Hay que tener en cuenta que las variables se crean en el momento en que las asignas, no pudiéndose declarar con anterioridad o dejarlas vacías.

R 6 (R como una calculadora).

```
> producto1 <- 45.6 * 34.5       # almacena
> producto1                      # muestra
[1] 1573.2
> producto2 <- 45.6 * 34.5
> producto2
[1] 1573.2
> 45.6 * 34.5 -> producto3
> producto3
[1] 1573.2
```

Una vez que se ha definido una variable, puede ser referenciada con otros operadores y funciones.

R 7 (R como una calculadora).

```
> .5 * producto1
[1] 786.6
> log(producto1)
[1] 7.360867
> log10(producto1)
[1] 3.196784
> log(producto3,base=2)
[1] 10.61949
> 45.6 * 34.5 -> producto; producto
[1] 1573.2
> pi
[1] 3.141593
```

Mediante le comando getOption("digits") se puede ver cuántos dígitos decimales hay por defecto y para cambiar esté número se utiliza el comando options(digits =10).

R 8 (R como una calculadora).

```
> getOption("digits")
[1] 7

> options( digits = 12)
```

1.4. Objetos en R.

R trabaja con objetos, los cuales tienen nombre, contenido y atributos. Estos últimos especifican el tipo de datos representados por el objeto. Por ejemplo, consideremos una variable que toma los valores 1, 2, 3 o 4. Esta variable podría ser un número entero (por ejemplo, número de hijos de una pareja), o el código de una variable (por ejemplo, el tiempo de una prueba de atletismo). Los resultados de un análisis de esta variable no será el mismo en ambos casos: con R, los atributos del objeto proporcionan la información necesaria. En general, se puede decir que la acción de una función sobre un objeto depende de los atributos del objeto.

Todo objeto tiene dos atributos intrínsecos: tipo y longitud. El tipo se refiere a la clase de los elementos en el objeto y existen cuatro tipos principalmente: numérico, carácter, complejo y lógico (Falso o Verdadero).

Existen otros tipos, pero no representan datos como tal (por ejemplo funciones o expresiones). La longitud es simplemente el número de elementos en el objeto. Para ver el tipo y la longitud de un objeto se pueden usar las funciones mode() o typeof() y length(), respectivamente. Veamos un ejemplo.

R 9 (Tipos de datos, longitud).

```
> x <- 3
> mode (x)
[1] "numeric"
> length(x)
[1] 1
```

Cuando un dato no está disponible, se representa como NA independientemente del tipo de datos.

1.4.1. Tipos de datos.

Los tipos de datos que puede manejar R son:

- Número. La forma más básica de almacenar un número es hacer una asignación de un número simple.

R 10 (Numérico).

```
> x <- 3
> mode(a)
[1] "numeric"
```

- Caracteres. Se pueden almacenar caracteres o cadenas de caracteres utilizando las comillas simples.

R 11 (Carácter).

```
> a = "hello"
> b = "world"
> mode(a)
[1] "character"
> a
[1] "hello"
> d = c(a,b)
> d
[1] "hello" "world"
> d[1]
[1] "hello"
```

- Complejo. Se pueden almacenar tipos de datos complejos de la forma natural $a + bi$, donde a representa la parte entera del número complejo y b su parte imaginaria.

R 12 (Complejo).

```
> num1=3+2i
> num2=3+2i
> num1
[1] 3+2i
> num1 * num2
[1] 5+12i
```

- Lógico. Hay dos tipos predefinidos de variables, TRUE o FALSE

R 13 (Lógico).

```
> a = TRUE
> mode(a)
[1] "logical"
> b = FALSE
> typeof(b)
[1] "logical"
```

Los operadores lógicos que se pueden utilizar son:

- $<$ → menor que.
- $>$ → mayor que.
- $<=$ → menor o igual que.
- $>=$ → mayor o igual que.
- $==$ → igual a.
- $!=$ → no igual a.
- $|$ → o lógico.
- $!$ → negación.
- $\&$ → y lógico.
- xor(a,b) → o exclusivo.

R 14 (Ejemplo operadores lógicos).

```
> 3 > 2  # TRUE  (¿tres es mayor que dos?)
[1] TRUE
> 3 < 2  # FALSE (¿tres es menor que dos?)
[1] FALSE
> 3 == 2 # FALSE (¿tres es igual a dos?)
[1] FALSE
> 5 == 5 # TRUE (¿cinco es igual a cinco?)
[1] TRUE
> 5 != 5 # FALSE (¿cinco es distinto de cinco?)
[1] FALSE
```

1.4.2. Tipos de objetos.

Como ya hemos comentado, la información que manipulamos en R se estructura en forma de objetos que viene definido por una serie de atributos. Nos vamos a centrar solo en los objetos más utilizados: vectores, matrices, listas, data frames y factores.

Los tipos de objetos que puede manejar R son:

- *Vectores*. Un vector es una colección de uno o más objetos del mismo tipo.

Veamos algunos ejemplos de construcción de vectores.

R 15 (Vectores).

```
> w <- c(4,6,7)
> x <- c(0:5,11:16)
> y <- c(pi,2)
> z <- c(x,y)
> z
[1]  0.000000  1.000000  2.000000  3.000000
[5]  4.000000  5.000000 11.000000 12.000000
[9]  13.000000 14.000000 15.000000 16.000000
[13] 3.141593  2.000000
```

- *Matrices* y *arrays*. Las matriz son objetos de dos dimensiones, mientas que los arrays son objetos de más de dos dimensiones (todos los elementos deben ser del mismo tipo).

R 16 (Matrices).

```
> ma <- matrix(sample(9),3)
> ma
     [,1] [,2] [,3]
[1,]    9    1    7
[2,]   11    8    6
[3,]   12    4    3
[4,]    2   10    5
> ma[2,3]  # muestra segunda fila, tercera columna
```

R 17 (Arrays).

```
> arr <- array(1:12,c(3,2,2))
> arr
, , 1

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

, , 2

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

- *Listas*. Una lista es una colección de objetos donde cada uno puede tener una estructura distinta. Veamos algún ejemplo que ilustre este concepto en R.

R 18 (Listas).

```
> x <- matrix(sample(12),4,3)
> y <- "Mi primera lista de R"
> z <- c(T,F,T,T)
> lista1 <- list(x,y,z)
> lista1
[[1]]
     [,1] [,2] [,3]
[1,]   10    6    9
[2,]    8   11    4
[3,]   12    5    2
[4,]    7    1    3
[[2]]
[1] "Mi primera lista de R"
[[3]]
[1]  TRUE FALSE  TRUE  TRUE
```

- *Data frames*. Son una estructura de datos que generaliza a las matrices, en el sentido en que las columnas pueden ser de diferentes tipos de datos entre sí. Sin embargo, todos los elementos de una misma columna deben ser del mismo tipo. Al igual que una matriz, todos las filas y columnas de un data frame deben ser de la misma longitud.

Ejemplo:

R 19 (Data frames).

```
> x <- c(10,15,20,25)
> y <- c(9,16,19,26)
> z <- c("A","P","A","P")
> df1 <- data.frame(x,y,z)
> df1
   x  y z
1 10  9 A
2 15 16 P
3 20 19 A
4 25 26 P
```

- *Factores*. Un factor es un vector utilizado para especificar una clasificación discreta de los elementos de otro vector de igual longitud.

  Ejemplo:

R 20 (Factores).

```
> origen_estu <-c("Alicante","Elche","Calpe","Benidorm",
"Calpe","Alicante","Alicante","Elche","Calpe",
"Torrevieja","Denia","Torrevieja","Alicante","Elche")
> factororigen <- as.factor(origen_estu)
> factororigen
 [1] Alicante   Elche      Calpe      Benidorm   Calpe
 [6] Alicante   Alicante   Elche      Calpe  Torrevieja
 [11] Denia Torrevieja Alicante Elche
Levels: Alicante Benidorm Calpe Denia Elche Torrevieja
```

La cantidad de clases de objetos es muy grande y crece permanentemente, acompañando la creación de nuevos paquetes de R. Casi en cada paquete existen funciones que devuelven objetos de clases únicas que sólo esa función puede generar y cuya interpretación y funcionalidad es específica del paquete en cuestión o de otros creados posteriormente y que dependen del mismo.

Por ejemplo, las clases igraph o Spatial de los paquetes **igraph** y **sp** definen objetos asociados a trabajo con grafos y datos espaciales respectivamente.

En general, estas clases nuevas suelen ser sofisticadas y son creadas para manejar cantidades considerables de información y pueden ser entrada de funciones que necesitan este formato de datos para producir una salida particular.

1.4.3. Vectores en R.

Un vector no tiene dimensión en R, más allá de que en álgebra se considera que los vectores tienen dimensión 1. Es decir, en R y, a diferencia de otros lenguajes como Matlab u Octave, los vectores no son fila o columna, si no que simplemente son una secuencia de valores. Esto se refleja en el hecho de que al consultar por la dimensión de un vector, con el siguiente comando dim(1:10). El resultado es simplemente NULL, es decir, no tiene una dimensión asociada.

⊙ **Formas de introducir un vector**

Existen diversas formas de introducir vectores en R. La primera de ella es mediante la utilización de los dos puntos situado entre dos números. Esto genera un vector en orden ascendente o descendente según se indique. Veamos algunos ejemplos.

R 21 (Vectores mediante dos puntos)**.**

```
> 1:10
[1] 1 2 3 4 5 6 7 8 9 10
> 10:1
[1] 10 9 8 7 6 5 4 3 2 1
> x <- 1:3
> x
[1] 1 2 3
```

El comando dos puntos para la definición de vectores es un caso particular de la función seq, que nos permite construir vectores que son sucesiones equiespaciadas. La instrucción para este comando es

```
seq(from, to, by, length, along)
```

Sus argumentos son:

from Nos indica el valor inicial de la sucesión.

to Nos indica el valor final de la sucesión

by Marca el salto entre los valores.

length Indica la longitud del vector resultante.

along

Destacamos que no es necesario especificar todos los argumentos posibles, como se pone de manifiesto en los ejemplos siguientes.

R 22 (Vectores mediante la función *seq*)**.**

```
> seq(from=1, to=15, by=1)
[1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
> seq(from=1, to=15, by=3)
[1] 1 4 7 10 13
> seq(from=1, to=15, length=8)
[1] 1 3 5 7 9 11 13 15
```

La forma más común y sencilla para construir vectores de cualquier tipo se utiliza la función de concatenación c, que puede definir tanto números como cadenas de caracteres alfanuméricos. Vemos algunos ejemplos.

R 23 (Vectores mediante la función c).

```
> c(1,2,3,4,5)
[1] 1 2 3 4 5
> c(1,2,3,4,5, seq(1, 5, by=2))
[1] 1 2 3 4 5 1 3 5
> meses <- c("enero","febrero","marzo","abril","mayo","junio")
> meses
[1] "enero" "febrero" "marzo" "abril" "mayo" "junio"
```

Pero existe una gran cantidad de formas de introducir vectores. Veamos algunas otras.

R 24 (Vectores mediante la función rep).

```
x <- 1:7   #Igual que c(1,2,3,4,5,6,7)
x <- rep(1, 100)  #x es un vector con 100 unos.
x <- rep(1:2, times = 5)  #repite 1,2 cinco veces.
x <- rep(1:2, each = 2) # repite cada elemento dos veces.
```

⊙ Aritmética de los vectores

Además de definir vectores de diversas formas, R nos permite realizar operaciones básicas con ellos, como por ejemplo sumarlos, multiplicarlos, etc.

R 25 (Operaciones con vectores).

```
> x <- 1:5
> y <- c(1,2,3,4,5)
> x * y
[1]  1  4  9 16 25
> x+y
[1]  2  4  6  8 10
> x^2
[1]  1  4  9 16 25
> length(x)
[1] 5
```

Notemos que cuando dos vectores tienen la misma longitud, el producto de ambos se realiza componente a componente. En general, las operaciones aritméticas básicas $+, -, *, /$, así como las funciones matemáticas como $log(), exp(), sqrt()$, etc. están definidas para operar vectorialmente, componente a componente. Si las dimensiones de los vectores no coinciden, se repite el vector

de menor longitud hasta que ambas coinciden y se lleva a cabo la operación. Podemos conocer en todo momento, la longitud de un vector mediante la función length(), tal y como aparece en el ejemplo.

Análogamente, podemos referirnos a un elemento concreto de un vector de una forma sencilla. Supongamos que queremos saber el elemento que ocupa la tercera posición del vector $\boldsymbol{x}$ definido en el ejemplo. Para ello, se utiliza $\boldsymbol{x}[i]$. Si queremos obtener un subconjunto de elementos del vector $\boldsymbol{x}$, reemplazamos i por un vector de índices que nos indique las posiciones que tomamos.

R 26 (extraer elementos de un vector)**.**

```
> x[1]
[1] 1
> x[1:3]
[1] 1 2 3
> max(x)
[1] 5
> min(x)
[1] 1
> x[1] <- 6
[1] 6 2 3 4 5
```

Para realizar operaciones usuales con vectores se pueden utilizar las siguientes funciones:

- sum() suma los elementos del vector.
- prod() multiplica los elementos del vector.
- min() obtiene el mínimo de todos los elementos del vector.
- max() obtiene el máximo de todos los elementos del vector.
- mean() media aritmética de los elementos del vector.
- median() mediana de los elementos del vector.
- range() rango de los elementos del vector.
- sd() desviación típica de los elementos del vector.
- var() varianza de los elementos del vector.
- cov() covarianza de los elementos del vector.
- cor() coeficiente de correlación de los elementos del vector.
- sort() ordena los elementos del vector.
- length() retorna la longitud del vector.
- summary() devuelve un resumen estadístico de los elementos del vector.

1.4.4. Variables indexadas en R. Matrices y arrays.

Una variable indexada (matriz o array) es una colección de datos, por ejemplo numéricos, indexada por varios índices. R permite crear y manipular variables indexadas en general y, en particular, matrices. Un vector de dimensiones es un vector de números enteros positivos. Si su longitud es k entonces la variable indexada correspondiente es k-dimensional. Los elementos del vector de dimensiones indican los límites superiores de los k índices. Los límites inferiores siempre valen 1.

Un vector puede transformarse en una variable indexada cuando se asigna un vector de dimensiones al atributo dim(). Supongamos, por ejemplo, que **z** es un vector de 1500 elementos. La asignación $> dim(z) < -c(3, 5, 100)$ hace que R considere a **z** como una variable indexada de dimensión $3 \times 5 \times 100$.

Existen otras funciones, como matrix() y array(), que permiten asignaciones más sencillas.

⊙ **Matrices en R**

La definición y manipulación de matrices en álgebra lineal constituye uno de los problemas esenciales que debemos estudiar. Existen múltiples formas de definir una matriz en R. Una de las formas más habituales de construir una matriz es mediante la función **matrix** que vamos a analizar detenidamente.

```
matrix(data, nrow, ncol, byrow, dimnames)
```

Los argumentos asociados a esta función son los siguientes:

data: Constituye habitualmente una lista de elementos que llenará posteriormente la matriz.

nrow: Número de filas de la matriz.

ncol: Número de columnas de la matriz.

byrow: Variable lógica que nos indica si la matriz debe construirse por filas o por columnas.

dimnames: Lista de longitud 2 con los nombres de las filas y las columnas.

Al indicar *nrow* y *ncol*, a menudo, nos basta con indicar uno de los argumentos ya que si, por ejemplo, hay 20 elementos en la lista de datos y le indicamos que el número de filas es 5, entonces el programa automáticamente calcula el número de columnas, que es 4 en este caso.

Puede darse el caso en que el número de datos que tenemos no concuerde exactamente con las dimensiones de la matriz que estamos construyendo. En ese caso, R automáticamente va rellenando las posiciones que quedan repitiendo los datos. Lo vemos con el siguiente ejemplo.

R 27 (Construcción de una matriz a partir de datos).

```
> matrix(data=1:10,nrow=3)
[,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8    1
[3,]    3    6    9    2
Warning message:
In matrix(data = 1:10, nrow = 3) :
la longitud de los datos [10] no es un submúltiplo o
múltiplo del número de filas [3] en la matriz
```

En este ejemplo le indicamos un conjunto de datos que va del 1 al 10 y la matriz la definimos con 3 filas. Como 10 no es múltiplo de 3, R automáticamente completa la cuarta columna de la matriz repitiendo los dos primeros valores de la lista de datos.

A continuación, se muestran otros ejemplos.

R 28 (Construcción de matrices con *matrix*).

```
\footnotesize > matrix(1:5,2,5)
[,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    2    4
[2,]    2    4    1    3    5
> matrix(1:5,2,5,byrow=TRUE)
[,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3    4    5
[2,]    1    2    3    4    5
> datos <- c(3,18,4,22,2,12,4,25,3,20)
> matrix(datos,5,2,byrow=TRUE, dimnames = list(c(),c("cajas","peso")))
cajas peso
[1,]     3   18
[2,]     4   22
[3,]     2   12
[4,]     4   25
[5,]     3   20
```

De la misma forma que podíamos hacer referencia a elementos concretos de un vector, podemos hacer referencia a un elemento concreto de una matriz o, incluso, crear submatrices. La idea es similar a la empleada con los vectores.

- $A[,1]$ Nos muestra los elementos de A en la columna 1.
- $A[4,]$ Nos muestra los elementos de la cuarta fila de A.
- $A(1,1)$ Nos muestra el elemento que ocupa la primera fila y la primera columna.

R 29 (Extracción de una submatriz).

```
> datos <- matrix(datos,5,2,by="T", dim = list(c(),
c("cajas","peso")))

> datos[c(2,4), ]
      cajas   peso
[1,]        4       22
[2,]        4       25
```

En el ejemplo anterior, podemos ver cómo extraer una submatriz formada por los elementos que forman las filas 2 y 4 de la matriz *datos*.

⊙ **Norma de un vector en R**

Vemos cómo podemos calcular la norma de un vector en R. Debemos siempre especificar qué norma es la que debemos calcula, dentro del posible conjunto de normas como la euclídea, la norma máximo, la norma suma y otras.

Supongamos que creamos un vector $\boldsymbol{x_1} = [1, 2, 3]^T$ y que utilizamos la instrucción para la norma en R, que es *norm*. Si hacemos $norm(\boldsymbol{x_1})$, obtendremos un mensaje de error. Este error es debido a que debemos convertir el vector original a objeto matriz.

Además, debemos especificar el tipo de norma que queremos que nos calcule. La instrucción es:

```
norm(x, type=c("O","I","F","M","2"))
```

La opción que nos interesa para calcular la norma euclídea, que es la que habitualmente utilizaremos en álgebra lineal, es F o f, en minúsculas. Así, debemos escribir las siguientes instrucciones para que nos calcule la norma euclídea del vector $\boldsymbol{x_1}$:

R 30 (Cálculo de la norma de un vector).

```
> x1 <- 1:3
> norm(as.matrix(x1), "f")
[1] 3.741657
```

Si queremos normalizar un vector en R utilizando la norma euclídea, podemos proceder del siguiente modo:

R 31 (Cálculo de la norma de un vector).

```
> x <- c(1,2,3)
> x1 <- norm(as.matrix(x))
> xnor <- x/x1
```

Algunas operaciones matemáticas de la aritmética de matrices son:

- + - * / operaciones aritméticas elemento a elemento.
- %*% multiplicación de matrices.
- t() traspuesta de una matriz.
- solve() inversa de una matriz.
- det() determinante de una matriz.
- chol() descomposición de cholesky.
- eigen() valores y vectores propios.
- %x% producto de kronecker.

⊙ Arrays en R

Los arrays son otro tipo similar a las matrices pero pueden tener más de dos dimensiones. Los arrays en R son objetos con n dimensiones que se definen mediante el comando array(datos, dimensiones)

Por ejemplo, un array de 3 dimensiones puede definirse mediante la instrucción

R 32 (Array tri-dimensional).

```
x <- array(1:12, c(2, 2, 3))
```

donde los valores del array se van llenando siguiendo el orden de las dimensiones del mismo, al igual que ocurría con las matrices. El número de dimensiones determina la forma en que se usan los índices; en este caso, si son 3 dimensiones, hay que utilizar 3 índices.

R 33 (Arrays).

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

> x<-array(c(35,46,55,45,170,167,48,49,68,56,169,165)
,c(2,3,2))
> dimnames(x)<-list(c("hombres","mujeres")
,c("edad","peso","altura"),
c("villa uno","villa dos"))
> x
, , villa uno
        edad peso altura
hombres   45   65    170
mujeres   46   55    167
, , villa dos
        edad peso altura
hombres   48   68    169
mujeres   49   56    165
> x[,,"villa uno"]
        edad peso altura
hombres   45   65    170
mujeres   46   55    167
> x["hombres",,]
        villa uno villa dos
edad            45        48
peso            65        68
altura         170       169
> x[,"edad",]
         villa uno villa dos
hombres         45        48
mujeres         46        49
```

Las variables indexadas pueden utilizarse en expresiones aritméticas y el resultado es una variable indexada formada a partir de las operaciones elemento a elemento de los vectores correspondientes. Las dimensiones de los operandos deben ser iguales y coincidirán con el vector de dimensiones del resultado.

Así, si A, B y C son variables indexadas (vetores, matrices o arreglos) simi-

lares, entonces el cálculo $> D < -2 * A * B + C + 1$ almacena en D una variable indexada similar, cuyo vector de datos es el resultado de las operaciones indicadas sobre los datos A, B y C.

1.4.5. Listas en R.

Hasta ahora hemos visto vectores y matrices como las estructuras más sencillas para organizar la información dentro de R. Una restricción importante es que dichos objetos requieren que todos los elementos que los componen deben ser de la misma clase/tipo. En R la clase lista tiene una estructura que supera esta limitación.

Las listas sirven para concatenar objetos donde cada uno puede tener una estructura distinta. Esto no ocurre, por ejemplo, en los arrays, donde todos los elementos deben ser del mismo tipo (todos números, o todos carácter digamos).

En definitiva, una lista puede imaginarse como un vector que contiene varios objetos, donde cada objeto puede ser de un tipo/clase diferente. Además los componentes de la lista no tienen por qué tener la misma cantidad de elementos.

Una lista tiene una serie de componentes, a los que deberemos asignar un nombre. Para crear una lista puede utilizarse la función list().

R 34 (Listas).

```
> x <- matrix(1:4, 4, 7)   # matriz
> y <- "Ejemplo de lista"   # cadena de caracteres
> z <- c(T, F, T, NA)  # vector lógico
> miprimeralista <- list(x, y, z)

> miprimeralista
[[1]]:
 [,1] [,2] [,3] [,4] [,5] [,6] [,7]
[1,] 1 1 1 1 1 1 1
[2,] 2 2 2 2 2 2 2
[3,] 3 3 3 3 3 3 3
[4,] 4 4 4 4 4 4 4
[[2]]:
[1] "Ejemplo de lista"
[[3]]:
[1] TRUE FALSE  TRUE    NA

> class(miprimeralista[[1]])
[1] "matrix"
> length(miprimeralista[[1]])
[1] 28

> miprimeralista[2:3]
[[2]]
[1] "Ejemplo de lista"

[[3]]
 [1]  TRUE  FALSE   TRUE     NA
```

Los ejemplos anteriores son listas muy simples y carentes de nombres. Para facilitar la interacción con el usuario las listas pueden tener nombres para sus objetos.

R 35 (Listas con nombres).

```
> familia<-list(padre="Jose",madre="Maria",hijos=2,
nomb.hijos=c("Julia","Carlos"),
edades.hijos=c(22,19),ciudad="Alicante")
> familia
$padre
[1] "Jose"
$madre
[1] "Maria"
$hijos
[1] 2
$nomb.hijos
[1] "Julia"   "Carlos"
$edades.hijos
[1] 22 19
$ciudad
[1] "Alicante"

> names(familia)   # nombres de los objetos
[1] "padre"  "madre"  "hijos"
"nomb.hijos" "edades.hijos" "ciudad"

> familia$padre
[1] "juan"
> familia$hijos
[1] 3

> familia[[1]]   # igual que en el caso anterior
[1] "juan"
> familia[[3]]
[1] 3
```

como se ha comentado, las listas son útiles para agrupar varios tipos de datos en un sólo objeto, ya que es una clase de objeto no tan restrictiva como las matrices o arreglos. Por lo tanto, muchas funciones con salidas relativamente complejas utilizan estas estructuras para organizar la información como por ejemplo eigen().

1.4.6. *Data Frames* en R.

Los *data frames* son una estructura de datos que generaliza a las matrices, en el sentido en que las columnas (variables a menudo) pueden ser de diferente tipo entre sí (no todas numéricas, por ejemplo). Sin embargo, todos los elementos de una misma columna deben ser del mismo tipo. También podemos ver los *data frames* como un tipo especial de listas utilizados para almacenar conjuntos de datos en tablas. Podemos pensar que las filas son casos y que las columnas son las variables.

Al igual que las filas y columnas de una matriz, todos los elementos de un *data frame* deben ser de la misma longitud. De este modo, pueden utilizarse funciones tales como dimnames(), dim(), nrow() sobre un *data frame* como si se tratara de una matriz. Los datos de un *data frame* pueden ser accedidos como elementos de una matriz o de una lista.

Para crear data frames se usa la función

```
data.frame()
```

que funciona de forma muy similar a list(). Veamos un ejemplo.

R 36 (Data Frames).

```
> x <- c(10, 9, 8.3, 2)
> y <- c(9.5, 8.5, 8, 1)
> z <- c("A", "B", "B", "Z")
> notas <- data.frame(x, y, z)
> notas
     x    y z
1 10.0  9.5 A
2  9.0  8.5 B
3  8.3  8.0 B
4  2.0  1.0 Z

>class(notas$x)
[1] "numeric"

>dimnames(notas)
[[1]]
[1] "1" "2" "3" "4"

[[2]]
[1] "x" "y" "z"

> dim(notas)
[1] 4 3

> nrow(notas)  #o ncol
[1] 4

> summary(notas)
       x                y        z
 Min.   : 2.000   Min.   :1.00   A:1
 1st Qu.: 6.725   1st Qu.:6.25   B:2
 Median : 8.650   Median :8.25   Z:1
 Mean   : 7.325   Mean   :6.75
 3rd Qu.: 9.250   3rd Qu.:8.75
 Max.   :10.000   Max.   :9.50
```

En muchos sentidos un data.frame se comporta en forma similar a un objeto indexado. En particular, la forma de usar los índices es prácticamente idéntica, hay que especificar filas y columnas en este orden. Siguiendo con el ejemplo anterior:

R 37 (Listas).

```
> notas[1,]
   x   y z
1 10 9.5 A

>notas[2,4]
[1] B
Levels: A B Z
```

1.4.7. Factores en R.

Las variables pueden ser clasificadas como nominales, ordinales o numéricas. Las variables nominales son variables categóricas sin un orden definido, mientras que las variables ordinales implican orden pero no cantidad. En R se presta un interés especial a estas variables nominales y ordinales y las llamamos factores. Los factores son cruciales porque van a determinar cómo se analizarán los datos y cómo se mostrarán en un gráfico. Un factor es una variable nominal u ordinal con un pequeño número de valores (cadenas de caracteres) permitidos que se llaman niveles.

Una forma de comprender el concepto de factor es pensando en su uso dentro del diseño experimental: un factor es una variable cuyo efecto se busca entender; en un experimento se crean distintos tratamientos, cada uno con un nivel determinado para dicho factor. Estos tratamientos equivalen al concepto de niveles utilizado en los objetos tipo factor de R. Por ejemplo, la influencia del nitrógeno en el crecimiento de un cultivo. Tenemos un factor (N) con dos niveles: con N y sin N (que pueden ser denominados 0 y 1).

Podríamos decir que un factor es un tipo de vector para datos cualitativos.

R 38 (Listas)**.**

```
> persona <- c("Hugo", "Paco", "Luis", "Eva", "Maria",
"Carmen", "Pepe", "Rafaela", "Angeles")

> mes.nacimiento <- c("Dic", "Feb", "Oct", "Mar", "Feb",
"Nov", "Abr", "Dic", "Feb", "Oct", "Dic")


>Fmes.nacimiento <- as.factor(mes.nacimiento)
> Fmes.nacimiento
[1] Dic Feb Oct Mar Feb Nov Abr Dic Feb Oct Dic
Levels: Abr Dic Feb Mar Nov Oct
```

1.5. Funciones en R.

Hay miles y miles de funciones en R, agrupadas en paquetes, y cada día hay más paquetes, con lo que el número de funciones aumenta sin parar. La creación de funciones es la principal utilidad de un lenguaje programado aún en un lenguaje orientado como R, pero ¿para qué hacer funciones?

El primer motivo puede ser la practicidad. Esto es cierto en situaciones en las que se debe realizar un procedimiento que requiere la ejecución de comandos en un orden específico. En casos como estos, puede ahorrar tiempo escribir una función que contenga estos pasos y que podamos utilizar una y otra vez.

Otro motivo es la generalización, ya que, en la medida en que las funciones que escribimos nos resulten útiles, es importante poder pasar a utilizar variables que podamos modificar en cada caso. Por ejemplo, el cálculo del volumen de une esfera de radio r.

R 39 (Funciones. Volumen de una esfera).

```
> 4/3 * pi * 3^3
[1] 113.0973

> Vol_esfera <- function(r){4/3 * pi * r^3}

> Vol_esfera(3)
[1] 113.0973
```

Como se ha visto en el ejemplo, la sintaxis para la creación de una función es:

```
variable <- function(arg1, arg2, ..., argn){expresión}
```

Los argumentos se colocan siempre entre paréntesis, separados por comas y aunque la función no requiera argumentos, se sigue necesitando el paréntesis. Se trata de una asignación de un valor: la función, a una variable. A partir de esa definición, la variable se puede utilizar como el nombre de la función. En R, toda expresión tiene un valor, así que el valor de la expresión será lo que la función regresará cuando se aplique la función. Si te teclea el nombre de una función entre paréntesis, se visualizará el código para esa función.

R 40 (Código de una función).

```
> matrix
function (data = NA, nrow = 1, ncol = 1, byrow = FALSE,
dimnames = NULL)
{
    if (is.object(data) || !is.atomic(data))
        data <- as.vector(data)
    .Internal(matrix(data, nrow, ncol, byrow, dimnames,
missing(nrow),
        missing(ncol)))
}
<bytecode: 0x0000000009cdfa68>
<environment: namespace:base>
```

En R es posible editar los objetos (funciones) de una sesión con los comandos edit(), fix() o view(). La diferencia entre ambos es que edit() sólo permite editar objetos existentes y no modificar el contenido del objeto editado, mientras que las funciones fix() o view() permiten definir y modificar objetos.

Por ejemplo, si se teclea view() se puede editar una función ya existente.

Para que una función devuelva los resultados de más de un cálculo, hay que guardar los valores en una lista o utilizar las funciones print() y cat(). En caso contrario, la función devolverá solamente el resultado del último cálculo.

Así, por ejemplo:

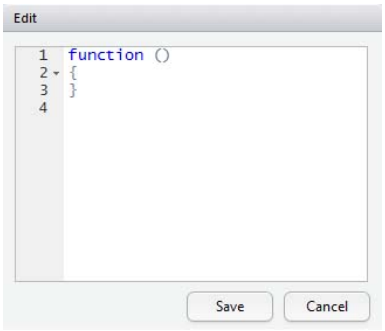


Figura 1.2: Editor.

R 41 (Retorno de una función 1).

```
> ejemplo1 <- function(x,y)
+{
+ var1 <- x^2+sqrt(y)
+ var2 <- sqrt(x)+y^2
+}

> x <-  9
> y <- 16
> ejemplo1(x,y)
[1] 85
```

R 42 (Retorno de una función 2).

```
> ejemplo2 <- function(x,y)
+{
+ var1 <- x^2+sqrt(y)
+ var2 <- sqrt(x)+y^2
+ print(var1)
+ cat("Variable 2:", var2,"\n")
+}
> x <- 9
> y <- 16
> ejemplo2(x,y)
[1] 85
variable 2: 259
```

R 43 (Retorno de una función 3).

```
> ejemplo3 <- function(x,y)
+{
+ var1 <- x^2 + sqrt(y)
+ var2 <- sqrt(x)+ y^2
+ list("Variable 1"=var1, "Variable 2"=var2)
+}

> x <- 9
> y <- 16
> ejemplo3(x,y)
$`Variable 1`
[1] 85

$`Variable 2`
[1] 259
```

Se les puede asignar valores por defecto a los argumentos de una función. Esto implica que, si no se les pasa el parámetro a la función, éste tomará el valor por defecto. Por ejemplo, la función log(),

R 44 (Retorno de una función 3).

```
> log
function (x, base = exp(1))  .Primitive("log")
```

calcula por defecto el logaritmo natural y para utilizar otra base hay que modificar este valor.

R 45 (Retorno de una función 3).

```
> log (32)
[1] 3.465736

> log(32, 2)
[1] 5
```

1.6. Scripts en R.

Un script es un conjunto de comandos de R que pueden ser guardados en un fichero y ejecutarse secuencialmente como si los teclearas en linea de comandos.

En Rstudio se puede crear y/o abrir un script en el menú **File**, submenú **New File**. Para poder ejecutar el contenido del script se utiliza el comando

source(). Esta función ejecuta los comandos que se encuentran en el fichero en la misma secuencia en la que están. Por otro lado, el ficheros-script tiene que estar en directorio de trabajo (working directory) y para ver en que directorio estamos trabajando se utiliza el comando getwd()).

Por ejemplo, se crea el fichero *Ejercicio.R* que contiene

R 46 (Script).

```
x <- rnorm(10)
mux = mean(x)
cat("La media del vector x es ",mean(x),"\n")

# resumn de los resultados
summary(x)
cat("El resumen de x es \n",summary(x),"\n")
print(summary(x))
```

Al ejecutar el comando source('Ejercicio.R') se obtiene:

R 47 (Ejecución de script).

```
> source('Ejercicio.R')
La media del vector x es  0.1691475
El resumen de x es
 -0.6759 -0.3743 0.122 0.1691 0.5466 1.729
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
-0.6759 -0.3743  0.1220  0.1691  0.5466  1.7290
```

1.7. Programación básica en R.

Como en cualquier otro lenguaje de programación, en R existen instrucciones para controlar el flujo de un programa, como por ejemplo, *for*, *if else*, *while*, *repeat*, y otras muchas.

1.7.1. Sentencia *if*

La ejecución condicional está disponible en R usando la sentencia *if* y la correspondiente *else*. A continuación mostramos un ejemplo.

R 48 (Sentencia *if*).

```
> x = 10
> if( x < 20)
  {
     x <- x + 1
     cat("incrementa el numero\n")
  }
incrementa el numero
> x
[1] 11
```

La sentencia *else* puede usarse para especificar una opción alternativa.

R 49 (Sentencia *if*).

```
> x = 20
> if ( x < 2)
 {
    x <- x + 1
    cat("incrementa el numero\n")
 } else
 {
    x <- x - 1
    cat("lo disminuye\n");
 }
lo disminuye
> x
[1] 19
```

Finalmente, la sentencia *if* puede ser encadenada junto con múltiples opciones. Así pues, se puede añadir después del *else*.

R 50 (Sentencia *if*).

```
> x = 10
> if ( x < 2)
 {
    x <- x + 1
    cat("incrementa el numero\n")
 } else if ( x < 15)
 {
   x <- 2*x
   cat("No es suficientemente grande\n")
 } else
 {
    x <- x - 1
    cat("lo disminuye\n");
 }
No es suficientemente grande
> x
[1] 20
```

El argumento de la sentencia *if* es una expresión lógica.

1.7.2. Sentencia *for*

El bucle *for* se usa para repetir un conjunto de instrucciones, y se utiliza cuando se conoce de antemano los valores de la variable del bucle. El formato básico de bucle *for* es *for(variable in secuencia) expresión*

Un ejemplo podría ser

R 51 (Sentencia *for*).

```
> x <- c(1,2,4,8,16)
> for (i in x)
 {
    cat("valor de i: ",i,"\n");
 }
valor de i:  1
valor de i:  2
valor de i:  4
valor de i:  8
valor de i:  16
```

Es imprescindible usar las funciones print() o cat() si se quiere que devuelva un resultado u objeto en cada iteración del bucle. Además, en la medida de lo posible, hay que evitar la utilización de los bucles *for* debido a su lentitud.

1.7.3. Sentencia *while*

El bucle *while* se usa para repetir una serie de instrucciones y se usa, frecuentemente, cuando no se conoce de antemano cuantas veces hay que ejecutar las instrucciones. El formato básico del bucle es *while(condición) expresión*

R 52 (Sentencia *while*).

```
> i <- 1;
> x <- 1
> while(x < 4)
 {
    x <- rnorm(1,mean=2,sd=3)
    cat("tratar este valor: ",x," (",i," veces)\n");
    i <- i + 1
 }
tratar este valor:  -4.163169  ( 1  veces)
tratar este valor:  3.061946  ( 2  veces)
tratar este valor:  2.10693  ( 3  veces)
tratar este valor:  -2.06527  ( 4 veces)
tratar este valor:  0.8873237  ( 5  veces)
tratar este valor:  3.145076  ( 6  veces)
tratar este valor:  4.504809  ( 7  veces)
```

1.7.4. Sentencia *repeat*

El bucle *repeat* es similar al bucle *while*. La diferencia es que siempre se ejecutará el bucle la primera vez. El bucle *while* solo empezará el bucle si se

cumple la condición la primera vez que es evaluado. Otra diferencia es que se tiene que especificar explicitamente cuando para el bucle utilizando el comando *break*.

R 53 (Sentencia *repeat*).

```
>  repeat
{
    x <- rnorm(1)
    if(x < -2) break
}
> x
[1] -2.300532
```

1.7.5. Sentencias *break* y *next*

La sentencia *break* se usa para parar la ejecución del bucle donde está incluida. La sentencia *next* se utiliza ara saltarse la ejecución de la sentencia que le sigue.

R 54 (Sentencias *break* y *next*).

```
> x <- rnorm(5)
> x
[1]  1.41699338  2.28086759 -0.01571884  0.56578443
0.60400784
> for(i in x)
 {
     if (i > 2.0)
         next

     if( (i<0.6) && (i > 0.5))
        break

    cat("El valor de i es ",i,"\n");
 }
El valor de i es  1.416993
El valor de i es  -0.01571884
```

1.7.6. Sentencia *switch*

La sentencia *switch* toma una expresión y devuelve un valor basado en los posibles resultados de la expresión. Esto hace que dependa del tipo d edatos de la expresión. La sintaxis básica es *switch(sentencia, item1, item2, item3,..., itemN)*.

Si el resultado de la expresión es un número, entonces devuelve el item de la lista que tenga el mismo índice.

R 55 (Sentencia *switch*).

```
> x <- as.integer(2)
> x
[1] 2
> z = switch(x,1,2,3,4,5)
> z
[1] 2
> x <- 3.5
> z = switch(x,1,2,3,4,5)
> z
[1] 3
```

Si el resultado de la expresión es una cadena de caracteres, entonces la lista de items tiene que tener la forma ?valorN?=resultadoN, y la sentencia devolverá el resultado que coincida con el valor.

R 56 (Sentencia *switch*).

```
> y <- rnorm(5)
> y
[1]  0.4218635 -0.8205637 -1.0191267 -0.6080061 -0.6079133
> x <- "sd"
> z <- switch(x,"mean"=mean(y),"median"=median(y),
"variance"=var(y),"sd"=sd(y))
> z
[1] 0.5571847
> x <- "median"
> z <- switch(x,"mean"=mean(y),"median"=median(y),
"variance"=var(y),"sd"=sd(y))
> z
[1] -0.6080061
```

1.7.7. Sentencia *scan*

El comando para leer entradas desde teclado es *scan*. Tiene una amplia variedad de opciones que pueden ser definidas dependiendo de las necesidades específicas. Espera la entrada del usuario y devuelve lo que tecleó.

Cuando se utiliza el comando sin un número determinado de líneas, continuará leyendo del teclado hasta que se introduzca una línea en blanco.

R 57 (Sentencia *scan*).

```
> help(scan)
> a <- scan(what=doble(0))
1: 3.5
2:
Leer 1 item
> a
[1] 3.5
> typeof(a)
[1] "doble"
>
> a <- scan(what=doble(0))
1: yo!
1:
Error in scan(file, what, nmax, sep, dec,
quote, skip, nlines, na.strings,  :
scan() expected 'a real', got 'yo!'
```

1.8. Datos en R.

La edición de datos desde R puede resultar difícil, en comparación con las hojas de cálculo, debido a que no presenta las mismas facilidades de visualización de tablas de datos que éstas. Si bien es posible editar datos como objetos de R (vectores, matrices o data.frames), es muy importante aprender a importar datos desde, por ejemplo, hojas de cálculo.

Los datos suelen leerse desde archivos externos y no teclearse. Las capacidades de lectura de archivos de R son sencillas y sus requisitos son bastante estrictos. Se presupone que el usuario es capaz de modificar los archivos de datos con otras herramientas para ajustarlos a las necesidades de R.

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

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

1.8.1. Importar datos.

Los archivos de texto plano, tales como los txt o csv, son la opción más sencilla para importar datos desde hojas de cálculo ya que desde cualquiera de ellas se pueden guardar archivos con estos formatos.

Por ejemplo, Excel permite guardar e importar archivos txt o csv y estos archivos pueden ser leídos desde R mediante la función read.table. Específicamente existe una función read.csv que lee datos de un fichero del tipo csv y cuyo

funcionamiento es muy similar al de read.table.

Antes de importar cualquier tipo de archivo que nuestros datos, se debe conocer el directorio de trabajo y el directorio donde se encuentran nuestros datos. Esto es muy importante puesto que muchas veces R trabaja en una carpeta por defecto y nuestros datos suelen encontrarse en otras carpetas. Para resolver estas cuestiones existen las funciones getwd() (informan del directorio de trabajo actual) y setwd() (permite cambiar dicho directorio).

Una vez situados en la carpeta donde se encuentran nuestros archivos de datos, se está en condiciones de importar los mismos a R.

⊙ **La función read.table**

Esta función importa los datos a un objeto de clase data.frame, con las mismas propiedades de cualquier objeto de esta clase y para mantener el objeto en el área de trabajo hay que asignarlo. Además, es importante entender los argumentos que tiene:

- **file**: es el nombre del archivo que queremos importar. Debe tener la extensión del archivo y dicho nombre debe ser escrito entre comillas.
- **header**: es un argumento lógico que cuando es TRUE indica que el nombre de las variables (columnas) está en nuestro archivo de calculo original. Debe tenerse en cuenta y no forman parte de las filas de la tabla importada.
- **sep**: es el argumento que indica como hemos separado los distintos campos (columnas) en nuestro archivo de texto. Dependiendo de que formato hayamos elegido para guardarlo será el valor correcto que debemos utilizar para que R reconozca en donde comienza y termina una columna. En el caso de los archivos txt los separadores suelen ser tabulaciones o incluso espacios en blanco de ancho variable (siendo esta última la opción por defecto de read.table). Para especificar la tabulación como separación se utiliza el simbolo `"\t"` Otra forma de guardar un archivo de cálculo es formato csv. En este caso el separador suele ser la coma *,* o el punto y coma *;*.
- *dec*: es el argumento que indica el símbolo de los decimales. La opción más adecuada depende de la configuración de idioma del programa con el que se creó el archivo de texto a importar. En el caso de archivos csv pueden darse confusiones entre separadores de columnas y decimales.

Existen otros argumentos en la función read.table como *row.names* (para poner nombre a las filas), *col.names* (para poner nombres a las columnas). Para conocer estos y otros argumentos utilice la ayuda o la función args, ?read.table o args(read.table)

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

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

A continuación aparece un ejemplo de las primeras líneas de un archivo llamado *datoscasas*, que contiene datos de viviendas, preparado para su lectura con esta función

R 58 (Fichero de datos).

```
Nombres de variables (columnas) y etiquetas (filas):
Precio Superficie Área Habitaciones Años Calefacción
01  52.00  111.0    830  5  8.2  no
02  54.75  128.0    710  5  6.5  no
03  67.50  101.0    978  5  3.2  si
04  57.50  131.0    690  6  7.8  no
05  69.75  103.0    900  5  1.5  si
...
```

Por defecto, los elementos numéricos (excepto las etiquetas de las filas) se almacenan como variables numéricas y los no numéricos, como Calef, se fuerzan como factores.

La utilización de la función sería. $>$ *Precio* $<-$ *read.table("datoscasas")*

A menudo no se dispone de etiquetas de filas. En ese caso, el programa añadirá unas etiquetas predeterminadas. Así, si el archivo tiene la forma:

R 59 (Fichero de datos).

```
Nombres de variables (columnas) y etiquetas (filas):
Precio Superficie Área Habitaciones Años Calefacción
52.00  111.0    830  5  8.2  no
54.75  128.0    710  5  6.5  no
67.50  101.0    978  5  3.2  si
57.50  131.0    690  6  7.8  no
69.75  103.0    900  5  1.5  si
...
```

Se podrá leer utilizando el parámetro header para indicarle que la primera línea es una línea de cabeceras y que no existen etiquetas de filas. $>$ *Precio* $<-$ *read.table("datoscasas",header=T)*

1.8.2. Exportar datos.

De la misma manera que se importan datos desde hojas de cálculo, puede interesar exportar datos en formatos que puedan ser leídos desde las mismas, ya sea como resultados de análisis o para visualizar y editar matrices de datos. La función en este caso es write.table.

Veamos el significado de los argumento con un ejemplo

write.table(ejemplo1, file = "ejemplo2.txt", sep = "\t ", eol = "\n ", dec = ".", row.names = TRUE, col.names = TRUE)

1. El primer argumento es el nombre del objeto que queremos guardar, sin comillas ni extensión. El archivo exportado se guardará en el directorio de trabajo actual.

2. **file**: es el archivo que queremos crear y en este caso si lleva comillas y extensión.

3. **sep**: es el mismo que para la función read.table.

4. **eol**: es un carácter especial que indica el final de línea y es importante cuando se trabaja con varios ordenadores con diferentes sistemas operativos.

5. **dec**: es el mismo que para la función read.table.

6. **row.names**: es una opción lógica que indica si los nombres de las filas se escriben en el archivo exportado.

7. **col.names**: opción lógica o vector con nombres de columnas. En el primer caso se guardan en el archivo los nombres de las columnas del archivo, mientras que el segundo simplemente asigna los nombres dados por el usuario en el momento de exportar.

También se puede colocar la información en el portapapeles (usando file = çlipboard") para después pegarlo en una hoja de cálculo, o incluso en otra sesión de R si así lo deseamos.

1.8.3. Datos internos.

En R se incluyen más de cien objetos con datos, y para utilizar estos datos, deben cargarse utilizando la función data. Para obtener una lista de los datos existentes en el sistema base se utiliza la funcióndata(), y para cargar uno, por ejemplo *euro*, debe suministrar dicho nombre como argumento de la función data, *data(euro)*

Normalmente una orden de este tipo carga un objeto del mismo nombre que suele ser una hoja de datos. Sin embargo, también es posible que se carguen varios objetos, por lo que es importante que, en cada caso, se consulte la ayuda interactiva sobre el objeto concreto, *help(euro)*.

1.8.4. Datos externos. Instalación de Packages.

Con la instalación simple de R tenemos muchísimas posibilidades, no obstante existen multitud de módulos opcionales que llamamos packages. Los paquetes son colecciones de funciones y datos, es decir, son extensiones que R necesita para poder hacer ciertas funciones. Existen más de 4000 paquetes disponibles para una infinidad de utilidades.

Para acceder a los datos incluidos en una biblioteca, hay que utilizar la función install.packages() y después cargar la biblioteca en memoria mediante library()

R 60 (Instalación de bibliotecas.)**.**

```
# Descarga e instalación de package
install.packages("igraph")

# cargar en memoria
library(igraph)
```

1.9. Algunos ejercicios.

Realiza los siguientes ejercicios.

Ejercicio 1.

Escribe una función que resuelva la ecuación de segundo grado. Prueba la función para resolver la ecuación

$$(x^2 + 5x - 16 = 0).$$

Ejercicio 2.

Supongamos que un ángulo α está dado como un número real positivo en grados. Escribe una función, denominada **Cuadrante**(α), que devuelva el cuadrante donde se encuentra el ángulo α.

Ejercicio 3.

Implementa una función que calcule la media y la desviación típica de un vector numérico que se le pase como parámetro.

Ejercicio 4.

Los primeros términos de la sucesión de Fibonacci son

$$1 \;\; 1 \;\; 2 \;\; 3 \;\; 5 \;\; 8 \;\; 13 \;\; 21 \cdots$$

Escribe una función, **Fibonacci(n)**, que calcule los n términos de esa sucesión. Es decir, el valor que devuelva **Fibonacci(n)** tiene que ser el n-ésimo término de la sucesión de Fibonacci.

Ejercicio 5.

Los primeros términos de la sucesión de números primos son

$$2 \;\; 3 \;\; 5 \;\; 7 \;\; 11 \;\; 13 \;\; 17 \;\; 19 \cdots$$

Escribe una función, **Primo(n)**, que calcule los términos de esa sucesión. Es decir, el valor que devuelva **Primo(n)** tiene que ser el n-ésimo término de la sucesión de números primos.

Ejercicio 6.

Supongamos que queremos encontrar una raíz de la función $f(x)$, esto es, deseamos encontrar un valor, x^*, tal que $f(x^*) = 0$. El método de Newton es un método iterativo que converge a una raíz. Empieza en un punto inicial x_0, y se aproxima linealmente a la función $f(x)$ alrededor de x_0. Encuentra la raíz de esta aproximación e itera estos pasos. La estructura del algoritmo es:

- Iniciar en x_0.
- Para $k = 1, 2, \cdots$ hacer

$$k_k = x_{k-1} - \frac{f(x_{k-1})}{f'(x_{k-1})}$$

  .

- Para un número determinado de iteraiones o para $|x_k - x_{k-1}| < \delta$.

Escribe una función en R con dos argumentos, *tolerancia* e *iteraciones*, de forma que para una ecuación dada devuelva la raíz.