





Práctica 1. Introducción a




Estadística
Grado en Ingeniería Informática

Índice


1. Introducción a 	3
2. Objetos y operaciones básicas	3
2.1. Primeros pasos en  . Asignación de valores	3
2.2.  como <i>calculadora científica</i>	4
2.3. Tipos de variables	5
2.4. Vectores	6
2.4.1. Construcción de vectores	6
2.4.2. <i>Length</i> , <i>sort</i> y <i>c</i>	7
2.4.3. Vectores lógicos, caracteres y factores	7
2.4.4. Selección de componentes	9
2.4.5. Operaciones dentro de un vector	10
2.4.6. Operaciones con vectores	10
2.4.7. Producto escalar	11
2.5. Matrices	11
2.5.1. Construcción de matrices	11
2.5.2. <i>Rownames</i> y <i>colnames</i>	12


2.5.3. Comandos para matrices	12
2.5.4. Operaciones con matrices	13
2.6. Factores	14
2.7. Listas y data frames	15
2.7.1. Listas	15
2.7.2. Data frames	16
3. Gràfics	17
3.1. Gràfics de alto nivel	18
3.1.1. La funció plot	18
3.1.2. Otros comandos gràfics	19
3.2. Gràfics de bajo nivel	20
4. Programació	21
4.1. Condicionales. La orden <i>if</i>	21
4.2. Repeticiones. Las ordenes <i>for</i> , <i>while</i> y <i>repeat</i>	22
4.3. Funciones	23
5. Librerías	24
6. Importación de datos	24
6.1. La función <i>read.table()</i>	25
6.2. La función <i>read.csv()</i>	25
6.3. Acceso a las bases de datos de 	26

1. Introducción a

 es un entorno de herramientas de software diseñado para la manipulación de datos, el cálculo matemático-estadístico y el tratamiento de gráficos. Toda la información acerca de  puede encontrarse en la página web <http://www.r-project.org/>, así como las opciones para su descarga (ver <http://cran.es.r-project.org/>, *espejo CRAN para España*) y diversos manuales para su utilización. Algunas de las principales características de  son:


- Efectividad en el manejo de datos y su almacenamiento.
- Entorno gráfico sencillo y potente.
- Herramientas de entrada/salida (lectura/escritura en archivos externos).
- Lenguaje de programación simple y bien estructurado (proviene del lenguaje C), con ejecución de comandos en línea: compilación y ejecución unidas en un mismo paso.
- Empleo de expresiones condicionales y bucles. Desarrollo por parte del usuario de funciones propias.
- Software libre y gratuito (S-Plus es su equivalente comercial).

Además,  crece y se desarrolla con el trabajo de sus propios usuarios. Dichos trabajos se almacenan en librerías (*packages*), que aglutinan funciones orientadas a un determinado objetivo: desde aplicaciones de diferentes métodos estadísticos, hasta la resolución de sudokus. Estas librerías pueden descargarse e instalarse a través del propio programa (desde el menú *Paquetes*).

Para más información, puede recurrirse bien a la ayuda de  que se instala con el programa (desde el menú *Ayuda*), o bien a los manuales disponibles en su página web (ver *Manuals* en el apartado *Documentation* en <http://www.r-project.org/>). También puede consultarse: Febrero, M. et al. (2008). *Estadística: Ingeniería Técnica en Informática de Sistemas*. USC (ver bibliografía básica de la asignatura).

2. Objetos y operaciones básicas

2.1. Primeros pasos en . Asignación de valores

Al ejecutar , se abre una consola en la que se pueden escribir directamente los comandos a ejecutar. Otra opción es crear un *script* (una pantalla de texto), escribir y guardar en él las funciones

o secuencias de comandos de interés. Luego, podrán ejecutarse copiándolas en la consola.

La operación más básica de es la de asignación de valores a distintas variables. El operador que nos permitirá realizar estas asignaciones es `<-`, el cual asigna a la variable que pongamos a su izquierda el valor que se encuentre a su derecha.

```
>a<-1; b<-'naranja'; c<-pi; d<-c
>a; b; c; d
[1] 1
[1] 'naranja'
[1] 3.141593
[1] 3.141593
```

En el ejemplo, hemos empleado el punto y coma (;) para poner todas las órdenes en la misma línea. Para saber qué variables hay creadas en el área de trabajo se usa el comando `ls()`. Para eliminar una variable existente se emplea `rm(nombre_variable)`.

2.2. como *calculadora científica*

Con se pueden realizar todas las operaciones de cálculo elemental. Los operadores utilizados son muy similares a los de otros lenguajes de programación: suma (+), resta (−), producto (*), división (/) y exponenciación (^). Si en una misma secuencia aparecen varios operadores binarios el *orden priorizado* de ejecución es exponenciación, producto/división y suma/resta. Para modificarlo, hemos de recurrir al uso de paréntesis.


```
>2+4*5-10/2^2
[1] 19.5
>(2+4)*5-(10/2)^2
[1] 5
```

Obsérvese que para la separación decimal, utiliza el punto. Este hecho puede dar lugar a problemas, por ejemplo, al importar datos de Excel donde se emplea la coma para separar la parte entera de la decimal.

Además de las operaciones elementales, pueden realizarse otro tipo de operaciones como logaritmos (`log`), exponenciales (`exp`), raíces cuadradas (`sqrt`), o funciones trigonométricas (`sin`, `cos`, `tan`, ...). Para estos casos, se deberá aplicar la correspondiente función sobre el valor con el que se opera.


```
>exp(2); sqrt(144); cos(pi/2)
[1] 7.389056
```

```
[1] 12  
[1] 6.123032e-17
```

En  los valores más infinito y menos infinito están representados por `Inf` y `-Inf`. Si llevamos a cabo operaciones para las cuales la matemática no tiene solución, el resultado es un *Not a Number*, `NaN`.

```
>1/0;-2*Inf;sqrt(-2);0/0  
[1] Inf  
[1] -Inf  
[1] NaN  
Warning message:  
In sqrt(-2) : Se han producido NaNs  
[1] NaN
```

2.3. Tipos de variables

En los ejemplos que hemos puesto hasta ahora, todas las variables han sido numéricas. Sin embargo, en  podemos trabajar con otros tipos de variable: `numeric`, `logical`, `character`, `list`, `matrix`, `array`, `factor` o `data.frame`. Para conocer el tipo o clase de una variable, se usa la función `class`.

```
>a<-2; b<-'variable'; c<-TRUE  
class(a); class(b); class(c)  
[1] 'numeric'  
[1] 'character'  
[1] 'logical'
```

Debemos tener en cuenta que el mismo operador/función puede tener efectos distintos según el tipo de objeto al que se le aplique. Por ejemplo, si hacemos `2*c`, la variable lógica se codifica primero como `TRUE=1/FALSE=0` y luego se opera como si fuera una variable numérica.

```
>2*c; class(2*c)  
[1] 2  
[1] 'numeric'
```

2.4. Vectores

opera con estructuras de datos. La más simple de ellas es el vector, que es una colección ordenada de números.

2.4.1. Construcción de vectores

En disponemos de varios mecanismos para construir vectores.

- La forma más común de crear un vector es usando el comando de concatenación numérica `c`.

```
>x<-c(3.4,23.1,5.9,6.7,7); x  
[1] 3.4 23.1 5.9 6.7 7.0
```

- Si deseamos crear un vector sin asignar de salida sus valores, podemos usar también `numeric` o `vector`, especificando en ambos casos la longitud del vector.

```
>y<-numeric(12); z<-vector(mode='numeric',length=5); y; z  
[1] 0 0 0 0 0 0 0 0 0 0 0 0  
[1] 0 0 0 0 0
```

- Para generar secuencias de número enteros podemos hacer lo siguiente.


```
>1:5; -3:3  
[1] 1 2 3 4 5  
[1] -3 -2 -1 0 1 2 3
```

- Cuando se busca tener secuencias equiespaciadas de números dentro de un intervalo, se utiliza la función `seq` indicando la longitud deseada de la secuencia o el espacio a dejar entre dos números consecutivos.

```
>seq(-5,5,length=5); seq(-5,5,by=2)  
[1] -5.0 -2.5 0.0 2.5 5.0  
[1] -5 -3 -1 1 3 5
```

- Si los vectores que deseamos generar toman en todas sus componentes el mismo valor (repeticiones) usamos el comando `rep`, indicando la longitud de vector deseada.

```
>rep(9,4)  
[1] 9 9 9 9
```

En cualquiera de los casos anteriores, para conocer qué parámetros y opciones tiene un comando podemos utilizar `args(nombre_comando)`. Si necesitamos más información sobre su uso, podemos recurrir a la ayuda de la que dispone  a través de `help(nombre_comando)`. Como ejemplo, podemos ver que argumentos tiene la función `numeric` mediante `args(numeric)` o acceder a la ayuda de `seq` empleando `help(seq)` o `?seq`.

2.4.2. *Length, sort y c*

Para conocer la longitud de un vector, se emplea la función `length`.

```
>length(x)
[1] 5
```

Si lo que nos interesa es ordenar los valores que conforman el vector de forma creciente o decreciente, podemos utilizar la función `sort`.

```
>sort(x); sort(x,decreasing=TRUE)
[1] 3.4 5.9 6.7 7.0 23.1
[1] 23.1 7.0 6.7 5.9 3.4
```

Para añadir más valores a un vector ya existente, podemos usar de nuevo el comando `c`.

```
>x<-c(x,a); x
[1] 3.4 23.1 5.9 6.7 7.0 2.0
```

2.4.3. *Vectores lógicos, caracteres y factores*

Además de numéricos, los vectores pueden ser lógicos, de tipo carácter o factores dependiendo del tipo de variables que los formen (`TRUE/FALSE`, caracteres o variables categóricas/nominales, respectivamente).

Vectores lógicos. Los vectores lógicos se generan normalmente al verificar una condición, por ejemplo, al comprobar si las componentes de un vector son negativas o si son distintas de 7.5.

```
>x<-c(4,1.3,-2.2,7.5,18,-3)
>x<0
[1] FALSE FALSE TRUE FALSE FALSE TRUE
>which(x<0)
[1] 3 6
```

```
>x!=7.5
[1] TRUE TRUE TRUE FALSE TRUE TRUE
```

En el ejemplo, el comando `which` indica qué elementos del vector cumplen la condición lógica de ser negativos.

Vectores con caracteres. Un vector de tipo carácter permite almacenar cualquier tipo de variable cuya respuesta no es numérica. Sus componentes estarán delimitadas por comillas. Por ejemplo, la respuesta a la pregunta ¿cuál es tu equipo favorito?

```
>y<-c('RM','FCB','RM','NIN','CEL','DEP','DEP','FCB','RM','RM','NIN')
>class(y)
[1] 'character'
>table(y)
CEL DEP FCB NIN RM
  1   2   2   2   4
```

En este caso, la función `table` sirve para resumir la información del vector. Si alguna de las personas preguntadas no responde a la pregunta, nos encontraríamos con un valor perdido. Estos se representan en con NA (*Not Available*).

```
>y[6]<-NA
>y
[1] 'RM' 'FCB' 'RM' 'NIN' 'CEL' NA 'DEP' 'FCB' 'RM' 'RM' 'NIN'
```

La función `paste` es muy útil para concatenar un determinado número de argumentos en secuencias de caracteres.

```
>paste('x',1:10,sep='')
'x1' 'x2' 'x3' 'x4' 'x5' 'x6' 'x7' 'x8' 'x9' 'x10'
>paste('limon','naranja',sep='-')
'limon-naranja'
```

Factores. Una variable como la de los equipos favoritos podría declararse también como tipo `factor` en función de las necesidades. Las variables `factor` permiten manejar de forma cómoda variables categóricas o agrupadas. La orden `as.factor` nos permite convertir una variable de cualquier otro tipo en `factor`, mientras que `levels` nos permite conocer las categorías de la variable.

```
>y<-as.factor(y)
>y
```



```
[1] RM FCB RM NIN CEL <NA>DEP FCB RM RM NIN
Levels: CEL DEP FCB NIN RM
>class(y)
'factor'
>levels(y)
[1] 'CEL' 'DEP' 'FCB' 'NIN' 'RM'
```

2.4.4. Selección de componentes

Diferentes subconjuntos de elementos de un vector pueden ser seleccionados incluyendo, entre corchetes después del nombre de vector, un vector índice: una expresión lógica o una secuencia numérica que indica los elementos a tomar.

```
>x[x<5&x>1]
[1] 4.0 1.3
>x[x<5|x>1]
[1] 4.0 1.3 -2.2 7.5 18.0 -3.0
>y[is.na(y)]
[1] <NA>
Levels: CEL DEP FCB NIN RM
>y[5:8]
[1] CEL <NA>DEP FCB
Levels: CEL DEP FCB NIN RM
```

Utilizando estos vectores índice, podemos reasignar los valores de un vector en aquellas posiciones que nos interesen. Por ejemplo, asignar un determinado valor a aquellas posiciones que presenten valores perdidos (NA).

```
>x[x<5&x>1]<-3
>x
[1] -3.0 -3.0 -2.2 7.5 18 -3.0
>z<-c(8.5,9,3,5,NA,6,6.5,1,NA,4.5)
>z[is.na(z)]<-1
>z
[1] 8.5 9.0 3.0 5.0 -1.0 6.0 6.5 1.0 -1.0 4.5
```


2.4.5. Operaciones dentro de un vector

Si tenemos un vector numérico, se pueden realizar sobre él numerosas operaciones: suma de sus componentes (`sum`), producto (`prod`), media (`mean`),...


```
>x<-c(2,3,4,5)
>sum(x)
[1] 14
>prod(x)
[1] 120
>mean(x)
[1] 3.5
>sum(x)/length(x)
[1] 3.5
>sum((x-mean(x))^2)
[1] 5
```

2.4.6. Operaciones con vectores

En general, las mismas operaciones que se podían llevar a cabo con números pueden ser realizadas con vectores.

Operaciones componente a componente. En  podemos realizar las operaciones básicas sobre vectores, componente a componente.

```
>x<-c(1,5,7,11);z<-c(1,-2,3,-4)
>x+z
[1] 2 3 10 7
>x*z
[1] 1 -10 21 -44
>x/z
[1] 1.000000 -2.500000 2.333333 -2.750000
>(2*x-z)^2
[1] 1 144 121 676
```

A diferencia de lo que ocurre con otros lenguajes de programación, es necesario estar muy atentos a que las longitudes de los vectores coincidan, pues en caso contrario  no devuelve un error (aunque si un mensaje de aviso) y seguirá operando como si nada hubiera ocurrido.

```
>t<-c(20,40,60)
>x+t
[1] 21 45 67 31
Warning message:
In x + t : longer object length is not a multiple of shorter object length
```

Y vemos que en la cuarta componente del vector suma ha reciclado la primera componente del vector t , realizando la operación $20 + 11 = 31$. Del mismo modo, se pueden llevar a cabo operaciones de vectores con números. En tal caso, la operación se aplica a cada componente del vector.

```
>x^2
[1] 1 25 49 121
>seq(-5,5,length=20)+5
[1] 0.0000000 0.5263158 1.0526316 1.5789474 2.1052632 2.6315789
[7] 3.1578947 3.6842105 4.2105263 4.7368421 5.2631579 5.7894737
[13] 6.3157895 6.8421053 7.3684211 7.8947368 8.4210526 8.9473684
[19] 9.4736842 10.0000000
```

2.4.7. Producto escalar

El producto escalar de dos vectores, tal y como se conoce en matemáticas $\sum_i x_i z_i$, se lleva a cabo con el operador `%*%`

```
>x%*%z
[1,] -32
```

2.5. Matrices


2.5.1. Construcción de matrices

Las matrices son las generalizaciones bidimensionales de los vectores. El comando `matrix` es el comúnmente utilizado a la hora de generar una matriz. En los parámetros de dicho comando es necesario indicar las componentes de la matriz y el número de filas y columnas. Otros comandos útiles para construir matrices son `rbind` y `cbind` que enlazan vectores por filas o por columnas para construir una matriz.

```
>A<-matrix(c(1,2,3,7),nrow=2,ncol=2)
>A
```

```
[1,] 1 3
[2,] 2 7
>B<-matrix(1:6,nrow=2,ncol=3)
>B
[1,] 1 3 5
[2,] 2 4 6
>A<-cbind(c(1,2),c(3,7))
>B<-rbind(c(1,3,5),c(2,4,6))
>A;B
```

2.5.2. *Rownames y colnames*

En las bases de datos con las que se suele trabajar tanto en Estadística como en otras ciencias experimentales, generalmente las filas representan a individuos y las columnas a las variables en estudio. Como consecuencia, puede ocurrir que, trabajando con matrices, nos interese asignar a determinadas filas o columnas nombres específicos. Esto se hace en  a través de los comandos `colnames` y `rownames`.

```
>colnames(A)
NULL
>colnames(A)<-c('Variable1','Variable2')
>A
Variable1 Variable2
[1,] 1 3
[2,] 2 7
```

2.5.3. *Comandos para matrices*

Existen multitud de comandos diseñados para trabajar sobre matrices: `diag` para extraer la diagonal, `det` para el determinante, `solve` que nos sirve para obtener la inversa (siempre y cuando la matriz sea no singular), ...

```
>diag(A)
[1] 1 7
>t(A)
Variable1 1 2
```

```
Variable2 3 7
>det(A)
[1] 1
>solve(A)
Variable1 7 -3
Variable2 -2 1
>A*%solve(A)
[1,] 1 0
[2,] 0 1
>solve(A)*%A
Variable1 1 0
Variable2 0 1
```

Además, `eigen` nos permite obtener autovalores y autovectores asociados a una determinada matriz.

```
>eigen(A)
$values
[1] 7.8729833 0.1270167
$vectors
[1,] -0.4000430 -0.9601733
[2,] -0.9164964 0.2794051
```

2.5.4. Operaciones con matrices

Operaciones componente a componente. Los operadores de cálculo comunes se pueden utilizar para operar componente a componente también con matrices. Aquí, a diferencia de lo que ocurría con los vectores, los desarreglos dimensionales serán reportados como errores.

```
>A+B
Error en A + B : arreglos de dimensión no compatibles
>C<-matrix(c(5,3,7,8),nrow=2,ncol=2)
>A+C
[1,] 6 10
[2,] 5 15
>A*C
[1,] 5 21
[2,] 6 56
```

Operaciones *matriciales*. A partir de un producto 'matricial' de vectores (puesto que un vector se puede considerar como una matriz con una única fila o columna) también se puede generar una matriz. El comando `t` se utiliza para obtener la traspuesta de una matriz (o de un vector).

```
>x<-c(1,2,3);y<-c(4,5,6)
>t(x)
[1,] 1 2 3
>x%*%t(x)
[1,] 1 2 3
[2,] 2 4 6
[3,] 3 6 9
>x%*%t(y)
[1,] 4 5 6
[2,] 8 10 12
[3,] 12 15 18
```

El producto de matrices comúnmente conocido en matemáticas se realiza, al igual que el producto vectorial que acabamos de ver, utilizando el operador `%*%`.

```
>A%*%B
[1,] 7 15 23
[2,] 16 34 52
```

2.6. Factores

Como ya anticipamos brevemente en un subapartado anterior, `factor` es la clase generalmente asignada en a las variables categóricas. El comando `factor` es el que proporciona para generar variables de este tipo. En el ejemplo que se muestra más abajo, generamos una variable categórica cuyas componentes son las letras de la palabra 'estadística' y los niveles, las letras del abecedario.

```
>letters
[1] 'a' 'b' 'c' 'd' 'e' 'f' 'g' 'h' 'i' 'j' 'k' 'l' 'm' 'n' 'o' 'p' 'q' 'r'
[20] 's' 't' 'u' 'v' 'w' 'x' 'y' 'z'
>substring("estadística",1:11,1:11)
```

```
[1] 'e' 's' 't' 'a' 'd' 'i' 's' 't' 'i' 'c' 'a'
>factor(substring('estadística',1:11,1:11),levels=letters)
[1] e s t a d i s t i c a
Levels: a b c d e f g h i j k l m n o p q r s t u v w x y z
```

A modo de ejemplo, señalar que las variables factor pueden estar formadas tanto por caracteres como por números.


```
>factor(sample(1:10,size=40,replace=TRUE),levels=1:10)
[1] 3 7 10 6 3 3 5 4 4 8 4 6 3 1 10 7 4 1 9 4 10 2 8 7 8
[26] 2 3 2 5 4 8 10 6 2 2 1 2 2 2 9
Levels: 1 2 3 4 5 6 7 8 9 10
```

El comando `levels` aplicado sobre una variable factor nos permite conocer sus niveles.

```
>levels(factor(sample(1:10,size=40,replace=TRUE),levels=1:10))
[1] '1' '2' '3' '4' '5' '6' '7' '8' '9' '10'
```

2.7. Listas y data frames

2.7.1. Listas

Una lista en  es un objeto consistente de una colección ordenada de otros objetos conocidos como componentes de la lista. Dichas componentes pueden ser de diferente clase, dimensión, El comando utilizado para generar una lista es `list` y sus parámetros, las componentes de dicha lista.

```
>Lista<-list(marido="Juan Pedro",esposa="Josefina",nhijos=3,
> + edadhijos=c(16,11,4),satisfaccion=matrix(c(8,7,5,4,4,3,5,10,7,7),
> + nrow=5,ncol=2))
```

Para acceder individualmente a las componentes de una lista existen dos vías. La primera es a través del número de componente, para lo cual tenemos que utilizar el doble corchete a continuación del nombre de la lista. La segunda es a través del nombre de la componente, escribiéndolo a continuación del nombre de la lista precedido por el carácter del dolar \$.

```
>Lista[[2]]
[1] 'Josefina'
>Lista[[5]]
[1,] 8 3
```

```
[2,] 7 5
[3,] 5 10
[4,] 4 7
[5,] 4 7
>Lista$marido
[1] 'Juan Pedro'
>Lista$edadhijos
[1] 16 11 4
```

2.7.2. Data frames

Otra clase de objetos con la que deberemos acostumbrarnos a trabajar son los data frames. Un data frame es una lista cuyas componentes han de cumplir una serie de requisitos. El comando general para la construcción de data frames es `data.frame`.

```
>cbind(1,1:5)
[1,] 1 1
[2,] 1 2
[3,] 1 3
[4,] 1 4
[5,] 1 5
>L3 <- LETTERS[1:3]
>L3
[1] 'A' 'B' 'C'
>data.frame(cbind(1,1:5),sample(L3,5,replace=TRUE))
  X1 X2 sample.L3..5..replace...TRUE.
1  1  1 C
2  1  2 A
3  1  3 A
4  1  4 A
5  1  5 C
```

Es común que, por exigencias de determinadas funciones de R, nos veamos en ocasiones obligados a convertir matrices en data frames. Análogamente a lo que ocurría con los factores, la orden a ejecutar es `as.data.frame`.

```
>matriz4x4<-matrix(1:16,nrow=4,ncol=4)
>as.data.frame(matriz4x4)
```



```
V1 V2 V3 V4  
1 1 5 9 13  
2 2 6 10 14  
3 3 7 11 15  
4 4 8 12 16
```

A pesar de que el entorno que nos ofrece el programa no es a primera vista tan atractivo como pueda ser el de otros programas comerciales, tales como S-Plus o SPSS, también nos ofrece la posibilidad de ver los datos de matrices o data frames en un formato de celdas, tal y como ocurre en estos últimos. Esto se consigue utilizando el comando `fix`.

```
>fix(matriz4x4)
```

Para que podamos continuar utilizando la consola de comandos, será necesario que de cada vez cerremos la pantalla con el formato de celdas.

3. Gráficos

Los gráficos representan una componente muy importante en R. Mediante gráficos es posible representar una gran cantidad de funciones estadísticas. Los comandos gráficos en se dividen básicamente en 3 grupos:

- **Gráficos de alto nivel.** Crean un nuevo gráfico en pantalla; opcionalmente incluyen etiquetas, títulos, leyenda, ...
- **Gráficos de bajo nivel.** Añaden información a gráficos ya existentes. Por ejemplo, puntos, líneas, ...
- **Gráficos interactivos.** Permiten al usuario añadir o eliminar información de un gráfico de forma interactiva, así como usar el ratón como puntero o localizador.

La totalidad de comandos y funciones gráficas que veremos aquí están incluidos en los paquetes que por defecto vienen instalados en R. Existen paquetes gráficos opcionales (y también gratuitos) que pueden ser instalados en R, tales como *maps*, *grid* o *lattice*, que proveen al usuario con funcionalidades más avanzadas.

3.1. Gráficos de alto nivel

Los gráficos de alto nivel están diseñados para generar un gráfico completo de los datos pasados como argumentos a la función. Ejes, etiquetas y títulos son generados automáticamente a no ser que el usuario exija lo contrario. Los gráficos de alto nivel siempre comienzan un nuevo gráfico, borrando el actual si no se les indica lo contrario. Si queremos abrir una pantalla gráfica nueva sin borrar la anterior, debemos usar la orden `windows()`.



3.1.1. La función `plot`

Sin duda la función gráfica más utilizada es `plot`. Es una función genérica, en la que el gráfico producido depende de la clase del primer argumento de la función. Veamos un ejemplo de su uso, donde se genera un gráfico de dispersión del peso en función de la altura en un grupo de 10 individuos.

```
>Altura<-c(175,177,192,159,166,188,180,182,177,163)
>Peso<-c(77,70,101,51,55,87,78,99,74,50)
>plot(Altura,Peso)
```

Si el primer dato que se le pasa a la función `plot` es un factor (por ejemplo, el sexo de cada uno de los individuos), el gráfico que se genera es un diagrama de caja (boxplot) del segundo dato (Altura o Peso) para cada nivel del factor Sexo. Veamos a continuación como se generarían dos boxplots de cada nivel de Sexo para la Altura y el Peso respectivamente.


```
>Sexo<-as.factor(c('H','M','H','M','M','H','H','H','M','M'))
>plot(Sexo,Altura,col=3)
>windows()
>plot(Sexo,Peso,col=4)
```

Hemos utilizado el parámetro `col`, para indicar a  el color en que queremos el gráfico. Esto se puede hacer numéricamente o bien indicando directamente el color con una orden del tipo `col='blue'`. Un catálogo de los colores disponibles en  puede encontrarse en: <https://r-charts.com/colors/>.

En ocasiones puede interesarnos generar una única pantalla gráfica que contenga multitud de gráficas diferentes. Supongamos, por ejemplo, que en la misma pantalla pretendemos incluir las tres gráficas generadas anteriormente y además una gráfica que represente la evolución de la estatura media en la última década (una serie temporal).

```
>par(mfrow=c(2,2))
```

```
>plot(Altura,Peso)
>plot(Sexo,Altura,col=3)
>plot(Sexo,Peso,col=4)
>EstatMedia<-c(166.8,168.2,168.1,168.9,170.5,170.2,170.6,172,173.1,174.2)
>plot(EstatMedia,type='l',lwd=2,col='cyan',lty=3)
```

Aquí, con la orden `par(mfrow=c(2,2))` estamos indicando a  que divida la pantalla de gráficos en 2 filas y 2 columnas, de modo que podamos incluir dentro $2 \times 2 = 4$ gráficos. Además, en el último de los gráficos, estamos usando parámetros que desconocíamos: `type='l'` nos sirve para indicar que queremos que los puntos del gráfico se unan a través de líneas, otras opciones son `type='p'` (para puntos) o `type='b'` (para puntos y líneas); el parámetro `lty` nos sirve para indicar el tipo de línea deseado, por ejemplo `lty=1` es la línea sólida habitual y `lty=3` es la línea discontinua; con el parámetro `lwd` indicamos el ancho de línea deseado.

3.1.2. Otros comandos gráficos

Aunque la función `plot` es la más importante y la más comúnmente usada, existen otras funciones que permiten llevar a cabo gráficos de alto nivel: `pairs` dibuja gráficos de dispersión 2 a 2 entre todas las columnas que conforman un data frame o una matriz numérica; `hist` dibuja el histograma de una variable; `image` crea un rectángulo a partir de una paleta de colores donde el color cambia en función de los datos del tercer argumento; `contour` crea un gráfico de contorno en función de los valores del tercer argumento y `persp` es el equivalente 3D al gráfico de dispersión 2D que dibujamos antes con `plot`. A continuación se muestran unos breves ejemplos de uso de dichos comandos.

```
>X<-matrix(sample(1:100,size=50*5,replace=TRUE),nrow=50,ncol=5)
>pairs(X)
>windows()
>hist(Peso)
>windows()
>x <- 10*(1:nrow(volcano))
>y <- 10*(1:ncol(volcano))
>image(x, y, volcano, col = terrain.colors(100),main = 'Volcan',
> + sub = 'Hawai, EEUU')
>windows()
>contour(x,y,volcano,col=2:3)
>windows()
>persp(x,y,volcano,theta=30,phi=40)
```

En el uso que aquí damos de la función `image` podemos encontrar un nuevo ejemplo de parámetro que se puede aplicar sobre gráficos de alto nivel: `main` sirve para dar un título al gráfico, mientras que de forma análoga un subtítulo puede aplicarse a través de `sub`; `theta` y `phi` son parámetros de `persp` que nos permiten cambiar el punto de vista desde el que observamos la gráfica.

3.2. Gráficos de bajo nivel


Algunas veces las funciones que crean gráficos de alto nivel no producen exactamente la clase de gráfico que el usuario desea. En tal caso, los comandos gráficos de bajo nivel pueden ser usados para añadir información extra al gráfico actual. En el siguiente ejemplo mostramos algunos de los gráficos de bajo nivel más comunes: la función `points` permite añadir a un gráfico uno o más puntos especificando sus coordenadas, el parámetro `pch` sirve para que el usuario indique cómo quiere señalar tales puntos (círculo sólido o hueco, diamante, triángulo, ...); `lines` permite añadir segmentos lineales a un gráfico preexistente, indicando las coordenadas de los puntos a unir; el comando `abline` permite incluir rectas dentro de un gráfico, indicando su vector director y el término independiente.

```
>plot(Altura,Peso)
>points(c(165,184),c(80,72),col='green',pch=2)
>lines(c(159,192),c(51,101),lwd=2.5,col=29)
>abline(a=72,b=0,col='red',lty=2,lwd=3)
```

Además de `points`, `lines` y `abline`, existen multitud de gráficos de bajo nivel en R. Aquí únicamente haremos referencia a dos más: `legend`, que permite incluir una leyenda en un gráfico preexistente, y `title`, que permite añadir títulos y subtítulos a un gráfico ya creado, siempre y cuando no se hubiera hecho antes.

```
>x<-seq(-pi,pi,length=65)
>plot(x,sin(x),type='l',ylim=c(-1.2,1.8),col= 3,lty=2)
>points(x,cos(x),pch=3,col=4)
>lines(x,tan(x),type='b',lty=1,pch=4,col=6)
>legend(-1,1.9,c('sin','cos','tan'),col=c(3,4,6),lty=c(2,-1,1),pch=c(-1,3,4))
>title(main='Funciones trigonométricas',sub='seno, coseno y tangente')
```

4. Programación

 es un lenguaje de expresión en el sentido de que sus comandos, funciones o expresiones propiamente dichas devuelven un resultado al usuario. Tal y como hemos visto, una secuencia de comandos pueden agruparse, bien estando separados por saltos de línea, bien por puntos y coma. Sin embargo, todo lenguaje de programación, para la resolución de problemas complejos, ha de estar en posesión de comandos que permitan una ejecución controlada en función de expresiones lógicas, tales como condicionales y repeticiones, que estudiaremos a continuación.

4.1. Condicionales. La orden *if*

Las sentencias condicionales se ejecutan a través de la orden *if* y, tal y como indica su nombre, ejecutan una determinada sentencia (que va entre llaves) siempre y cuando se cumpla una determinada expresión lógica (que va entre paréntesis justo después del *if*). Opcionalmente, pueden incluir un *else* y justo después una sentencia entre llaves que se ejecutará siempre y cuando la expresión lógica después del *if* no se cumpla.


```
>l<-sample(1:10,size=1)
>if (l<6) {print('entre 1 y 5')} else {print('entre 6 y 10')}
```

Aquí estamos simulando, con equiprobabilidad, un número entero entre 1 y 10, y, condicionalmente a si el número simulado está entre 1 y 5 o entre 6 y 10, lo imprimimos en pantalla, usando la orden *print*. Obviamente, las condicionales pueden hacerse más complejas según las necesidades específicas del usuario, usando *else if*

```
>l<-sample(1:10,size=1);t<-sample(1:10,size=1)
>if (l<t) {print(c(l,t,l*t))} else if (l==t) {print(c(l,t,l*t))}
>else {print(c(l,t,l/t))}
```


La complejidad de las condicionales está relacionada también con la complejidad de las expresiones lógicas que van dentro de los paréntesis. Supongamos que en un ejemplo como el anterior nos interesa imprimir en pantalla el producto de *l* por *t* siempre y cuando al menos uno de los dos está por debajo de 4 y el producto tome un valor superior a 20. La sentencia condicional sería como sigue:

```
>l<-sample(1:10,size=1);t<-sample(1:10,size=1)
>if ((l<4|t<4)&(l*t)>20) {print(l*t)}
```

Es muy común, mientras se programan secuencias de comandos, que el usuario quiera insertar comentarios, con el fin de dar explicaciones acerca de los pasos que se van dando en cada momento. Esto se consigue mediante el carácter #, de modo que , cada vez que encuentra en una línea dicho carácter, no tiene en cuenta todo lo que va después.

```
>#En esta línea no se ejecutará nada.  
>print(c(1,t)) #Esto es un comentario.
```

4.2. Repeticiones. Los ordenes *for*, *while* y *repeat*

En muchas ocasiones nos encontramos en situaciones en las que deseamos repetir una determinada secuencia de comandos un cierto número de veces o en función de si se da o no una determinada condición. En  existen 3 posibles ordenes para la repetición de secuencias de comandos: *for* se utiliza para la repetición finita de sentencias en un bucle; *while*, para la repetición de sentencias mientras se cumple una determinada expresión lógica; finalmente, *repeat*, se utiliza cuando se pretende repetir una serie de sentencias hasta que se llega a un *break* que detiene la ejecución.

Un ejemplo de uso de bucle *for* se muestra a continuación, donde se calculan los elementos de un vector (declarado previamente) mediante un bucle que se ejecuta a lo largo de la longitud de dicho vector.

```
>vec<-numeric(20)  
>for (i in 1:length(vec)) {  
> +   vec[i]<-sqrt((i*3-17)/2)  
> + }  
>vec
```

Para la orden *while*, tenemos que volver a echar mano de las expresiones lógicas, al igual que ocurría con *if*, debido al carácter condicional de la repetición.

```
>j<-0;k<-0  
>while(j<1e20|k<40) {j<-2**k;k<-k+1;print(c(j,k))}
```

Finalmente, la orden *repeat* repite una secuencia de comandos sin condiciones ni puntos de corte, de modo que la única forma de pararla es a través de un *break*. En el siguiente ejemplo, se calcula un vector cuyos componentes son las sucesivas potencias de 2, hasta que la ejecución se corta cuando las potencias son ya excesivamente grandes.

```
>l<-1;s<-1
>repeat {
> +   l<-c(l,2**s);s<-s+1;if (l[length(l)]>1e8) {break}
> +   }
>l
```

4.3. Funciones

Una de las ventajas del lenguaje de programación , y a la vez una de las claves para su crecimiento y el desarrollo de paquetes y librerías, es que permite al usuario crear objetos del tipo función. Las funciones son secuencias de comandos y órdenes que devuelven una o más salidas, y que pueden ser usadas en expresiones posteriores. De esta forma, el lenguaje de programación crece en potencia y elegancia, y su uso se hace más agradable y productivo.

A modo de ejemplo, mostramos una función cuyos parámetros de entrada son dos vectores y_1 y y_2 , y que devuelve como salida la media y la desviación típica de la resta de dichos vectores.

```
>calculo<-function(y1,y2) {
> +   Ry1y2<-y1-y2
> +   return(list('media'=mean(Ry1y2),'desv'=sd(Ry1y2)))
> + }
```

Puede observarse que uno de los comandos fundamentales en el interior de la función es `return`, que indica los parámetros de salida de la función. Para llamar a la función y obtener la salida, el procedimiento sería el siguiente.


```
>#Generamos x y z como vectores longitud 50 generados de una normal
>x<-rnorm(50);z<-rnorm(50)
>#Llamamos a la función y obtenemos la salida
>calculo(x,z)
$media
[1] -0.06090414
$sd
[1] 1.559908
```


En la llamada a la función es importante saber que no es necesario que los parámetros de entrada tomen el mismo nombre que tenían cuando se creó la función. Una función puede tener tantos parámetros de entrada o salida como se quiera, y pueden ser de distintos tipos y clases.

Cuando un usuario de crea funciones es altamente probable que se cometan errores, tanto

de programación como en los posibles cálculos/órdenes que se llevan acabo en su interior, y más especialmente cuanto más larga sea la función. En tales casos, y para depurar el programa, se hace muy útil el uso de la función `print` para escribir salidas en pantalla.





5. Librerías

Como ya comentamos previamente, una de las ventajas más importantes de  es que cada usuario puede construir sus propias funciones, almacenarlas en librerías o paquetes (*packages*) y hacerlas públicas de modo que estén disponibles para el resto de usuarios. Para conocer qué paquetes están ya instalados en nuestro entorno R, ejecutamos `library()`. Para que un determinado paquete funcione, no sólo ha de estar instalado, también ha de cargarse en la consola de R. Para saber cuáles de los paquetes instalados están cargados en nuestro entorno, ejecutamos la orden `search()`.

En muchas ocasiones, nos encontramos con que el paquete que nos interesa no se encuentra instalado en R. Si se dispone de una conexión a Internet, en el menú de  se va a *Paquetes e Instalar paquete(s)...*, donde el usuario ha de escoger una ventana lo más cercana posible en el espacio. Una vez hecho esto, nos aparece en la pantalla una lista (que va creciendo con el paso del tiempo) de paquetes disponibles. Tras instalarlo, se carga en pantalla mediante la orden `library(nombre_paquete)` y todas sus funciones pasarán a estar disponibles.

Para obtener información acerca de un paquete cargado en nuestra consola, podemos hacer un `help(nombre_paquete)`, o bien descargar la ayuda en pdf del paquete disponible en <http://cran.es.r-project.org/>.

6. Importación de datos

Aunque  y muchos de los paquetes de  proveen a los usuarios con bases de datos, en muchas ocasiones desharemos trabajar con datos almacenados en archivos externos, ya sean archivos de texto, Excel, SPSS, ... Los comandos y funciones de  para la importación de datos son sencillos pero también estrictos y bastante inflexibles. Esto se debe fundamentalmente a que los diseñadores del lenguaje  creen que no es complicado para el usuario modificar sus propios archivos de modo que cumplan los requerimientos de R.

6.1. La función *read.table()*

La función *read.table()* es la que se usa normalmente para leer datos de archivos externos tipo texto (con extensión .txt). Su formato es muy sencillo y el número de parámetros de la función muy grande, tal y como se puede apreciar en la ayuda de la misma (tecleando `?read.table`).

Un ejemplo de aplicación de dicha orden lo podemos ver descargándonos en formato texto cualquier base de datos de interés que encontremos por la red. Por ejemplo, en la página web del Instituto Gallego de Estadística (IGE), <https://www.ige.eu/>, podemos encontrar multitud de bases de datos midiendo índices de interés en la población gallega. Podemos descargar, a modo de ejemplo, la base de datos con las cifras de población de las cuatro provincias gallegas. La orden correcta para importar dichos datos a sería:

```
>DatosPob<-read.table(file='Ruta_DatosPoblacion.txt',sep='t',header=TRUE)
```

En *file* le indicamos a la ruta a seguir (en el propio ordenador) para llegar al archivo deseado; *header* es un parámetro lógico para indicar a si la primera fila del archivo corresponde (TRUE) o no (FALSE) a una cabecera indicando los nombres de las variables; con *sep* le indicamos a cual es la separación entre variables en un archivo (espacio, separador, coma, ...). Opcionalmente, se podrían usar cualquiera de los otros parámetros que aparecen en la ayuda, como *dec*, con el cual indicamos a el separador decimal utilizado en el archivo de texto de partida.

6.2. La función *read.csv()*

La orden *read.csv()* es el equivalente a *read.table()* para importar a datos procedentes de archivos Excel con extensión .csv (en algunos casos será necesario usar *read.csv2()*). En la ayuda de podemos encontrar su modo de uso:

```
>read.csv(file, header = TRUE, sep = ',', quote='\'', dec='.',  
> + fill = TRUE, comment.char='', ...)
```

Un ejemplo de aplicación de dicha orden lo podemos ver también a partir de la base de datos de población en las capitales de provincia, que ya usamos antes:

```
>DatosPob2<-read.csv(file='Ruta_DatosPoblacion.csv',sep=';',header=TRUE)
```

En casos como éste, es necesario que hayamos guardado el archivo Excel con la extensión .csv, indicando que la separación se ha realizado con punto y coma. Aunque en este caso no sea necesario, ya que los datos de población son números enteros, ha de tenerse mucho cuidado cuando se trabaje con archivos Excel, puesto que la separación decimal suele ser la coma, con lo cual sería necesario añadir *dec='.'* a los parámetros.

6.3. Acceso a las bases de datos de

Alrededor de un centenar de bases de datos son suministradas por , bien a través del paquete `datasets` (que se instala por defecto con , bien a través de otros paquetes. Podemos ver la lista de bases de datos a las que tenemos acceso desde usando el comando `data()`. El acceso a las bases de datos que se encuentran en los paquetes base de se realiza a través de su nombre. Obviamente, antes de usar dichas bases, estaremos interesados en conocer sus características, para lo cual haremos un `help` sobre el nombre de dichas bases:

```
>C02  
>help(C02)
```

Puede ocurrir que en determinadas ocasiones estemos interesados en disponer de las bases de datos presentes en paquetes de que no vienen instalados por defecto con el programa. Si por ejemplo deseamos conocer las bases de datos en el paquete `rpart`:

```
>data(package='rpart')
```

Si, por otro lado, deseamos obtener acceso a la base de datos `imports85` dentro del paquete `randomForest`, escribiremos:

```
>data(imports85,package='randomForest')
```