

---

# INTRODUCCIÓN A LA PROGRAMACIÓN ESTADÍSTICA CON R PARA PROFESORES

---

José Miguel Contreras García  
Elena Molina Portillo  
Pedro Arteaga Cezón

©

José Miguel Contreras García

Elena Molina Portillo

Pedro Arteaga Cezón

Todos los derechos reservados. Ninguna parte del libro puede ser reproducida, almacenada en forma que sea accesible o transmitida sin el permiso previo por escrito de los autores.

ISBN: 978-84-693-4859-8

Depósito legal:

Financiación:

Esta obra forma parte de los proyectos: SEJ2007-60110/EDUC (MEC-FEDER) y EDU2010-14947 (subprograma EDUC); con la colaboración de la beca FPI BES-2008-003573 y la beca FPU AP2007-03222.

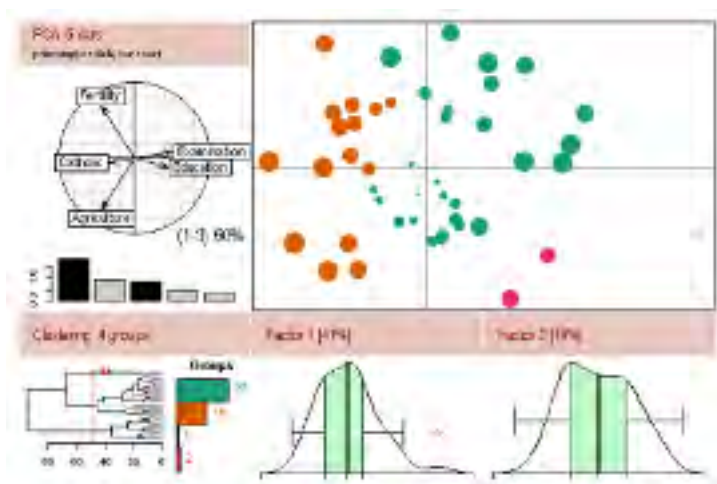
# Índice general

I	Introducción al lenguaje R	1
II	Primeras nociones	5
III	Manipulación de Datos	17
IV	Funciones	40
V	Dispositivos gráficos	53
VI	Estadística Descriptiva	78
VII	Distribuciones de probabilidad	90
VIII	Regresión	104
IX	Inferencia	113
X	Análisis de la varianza	123
XI	Combinatoria y probabilidad	132
XII	Entornos gráficos para trabajar con R	153

# Parte I

## Introducción al lenguaje R

# Introducción al lenguaje R



*R* es un lenguaje y entorno de programación, creado en 1993 por Ross Ihaka y Robert Gentleman del Departamento de Estadística de la Universidad de Auckland, cuya característica principal es que forma un entorno de análisis estadístico para la manipulación de datos, su cálculo y la creación de gráficos. En su aspecto *R* puede considerarse como otra implementación del lenguaje de programación *S*, con la particularidad de que es un software GNU, General Public License (conjunto de programas desarrollados por la Free Software Foundation), es decir, de uso libre.

La página principal del proyecto “*R – project*” es [http : //www.r – project.org](http://www.r-project.org), en ella podremos conseguir gratuitamente el programa en su última versión, o cualquiera de las anteriores (para el caso de utilizar paquetes no implementados para las últimas versiones), además de manuales, librerías o package y demás elementos que forman la gran familia que es *R*.

Hay que tener en cuenta *R* es un proyecto vivo y sus capacidades no coinciden totalmente con las de *S*. A menudo el lenguaje *S* es el vínculo escogido por investigadores que utilizan la metodología estadística, y *R* les proporciona una ruta de código abierto para la participación en esa actividad, los usuarios pueden contribuir al proyecto implementando cualquiera de ellas, creando modificaciones de datos y funciones, librerías (packages),... Ningún otro programa en la actualidad reúne las condiciones de madurez, cantidad de recursos y manejabilidad que posee *R*, además de ser el que en los últimos años ha tenido una mayor implantación en la comunidad científica.

Entre otras características dispone de:

- Almacenamiento y manipulación de datos.
- Operadores para cálculo sobre variables indexadas (Arrays), en particular matrices.
- Herramientas para análisis de datos.
- Posibilidades gráficas para análisis de datos.

El término entorno lo caracteriza como un sistema completamente diseñado y coherente de análisis de datos. Como tal es muy dinámico y las diferentes versiones no siempre son totalmente compatibles con las anteriores. En la introducción a *R* no se hace mención explícitamente a la palabra estadística, sin embargo mayoritariamente se utiliza *R* como un sistema estadístico, aunque la descripción más precisa sería la de un entorno en el que se han implementado muchas técnicas estadísticas. Algunas están incluidas en el entorno base de *R* y otras se acompañan en forma de bibliotecas (packages).

Una diferencia fundamental de la filosofía de *R*, y también de la de *S*, con el resto del software estadístico es el uso del “objetos” (variables, variables indexadas, cadenas de caracteres, funciones, etc.) como entidad básica. Cualquier expresión evaluada por *R* se realiza en una serie de pasos, con unos resultados intermedios que se van almacenando en objetos, para ser observados o analizados posteriormente, de tal manera que se puede hacer un análisis sin necesidad de mostrar su resultado inmediatamente produciendo unas salidas mínimas.

Cada objeto pertenece a una clase, de forma que las funciones pueden tener comportamientos diferentes en función de la clase a la que pertenece su objeto argumento. Por ejemplo no se comporta igual una función cuando su argumento es un vector que cuando es un fichero de datos u otra función.

*R* está disponible en varios formatos: en código fuente está escrito principalmente en *C* (y algunas rutinas en *Fortran*), esencialmente para máquinas *Unix* y *Linux*, o como archivos binarios precompilados para *Windows*, *Linux*(Debian, Mandrake, RedHat, SuSe), *Macintosh* y *Alpha Unix*.

Junto con *R* se incluyen ocho bibliotecas o paquetes (llamadas bibliotecas estándar) pero otros muchos están disponibles a través de Internet en (<http://www.r-project.org>). Actualmente se encuentran disponibles 2337 librerías (packages) desarrollados en *R*, que cubren multitud de campos desde aplicaciones Bayesianas, financieras, graficación de mapas, wavelets, análisis de datos espaciales, etc. Esto es lo que define *R* como un entorno vivo, que se actualiza con frecuencia y que está abierto a la mejora continua.

Podemos ver o modificar la lista de bibliotecas disponibles mediante la función “*libPaths*” y conocer el camino a la biblioteca predeterminada del sistema La variable “*Library*”. Estas bibliotecas se pueden clasificar en tres grupos: las que forman parte del sistema base y estarán en cualquier instalación, los paquetes recomendados (aunque no forman parte del sistema base

se aconseja su instalación) y otros paquetes desarrollados por investigadores de todo el mundo para tareas o métodos de lo más diverso. Destacando áreas nuevas como ciencias de la salud, epidemiología, bioinformática, geoestadística, métodos gráficos, etc.

Una característica del lenguaje *R* es que permite al usuario combinar en un solo programa diferentes funciones estadísticas para realizar análisis más complejos. además los usuarios de *R* tienen a su disponibilidad un gran número de programas escritos para *S* y disponibles en la red la mayoría de los cuales pueden ser utilizados directamente con *R*.

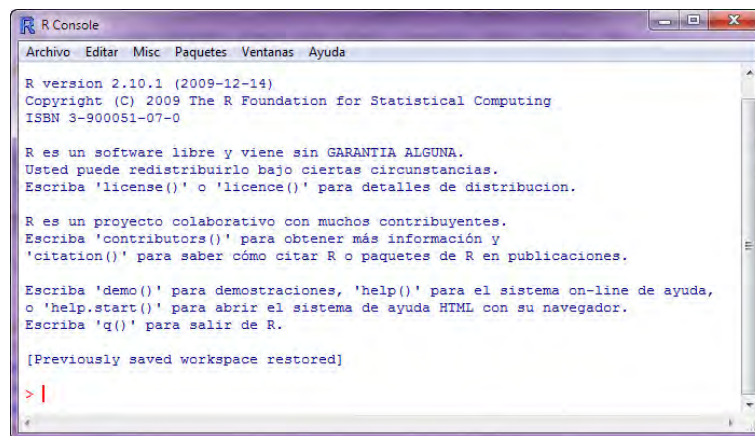
# Parte II

## Primeras nociones



# Primeras nociones

Una vez instalado el programa, la forma más fácil de usarlo es de forma interactiva mediante la línea de comandos. Cuando R se inicia, aparece la ventana del programa “*Gui*” (graphical user interface) con un mensaje de apertura.



Debajo del mensaje de apertura de la consola de R se encuentra el “*prompt*” que es el símbolo “>” (mayor que). Las expresiones en *R* se escriben directamente a continuación del “*prompt*” en la consola de *R*. Si se escribe e intenta ejecutar un comando que se ha escrito en forma incompleta, el programa presenta un “+” que es el prompt de continuación.

Una característica de *R* es que nos permite guardar la sesión de trabajo, lo que nos será muy útil en el caso de que necesitemos utilizar bibliotecas o datos que ya hemos implementado. Al cerrar el programa o al teclear “*q( )*” nos preguntará si desea salvar los datos de esta sesión de trabajo. Puede responder yes (Si), no (No) o cancel (cancelar) pulsando respectivamente las letras *y*, *n* o *c*, en cada uno de cuyos casos, respectivamente, salvará los datos antes de terminar, terminará sin salvar, o volverla a la sesión de *R*. Los datos que se salvan estarán disponibles en la siguiente sesión de *R*.

Es importante saber que el nombre de un objeto debe comenzar con una letra (A-Z ó a-z) y además puede incluir letras, dígitos (0-9) o puntos (.). *R* discrimina entre letras mayúsculas y minúsculas para el nombre de un objeto, de tal manera que *x* y *X* se refiere a objetos diferentes (inclusive bajo Windows).

## Conceptos iniciales.

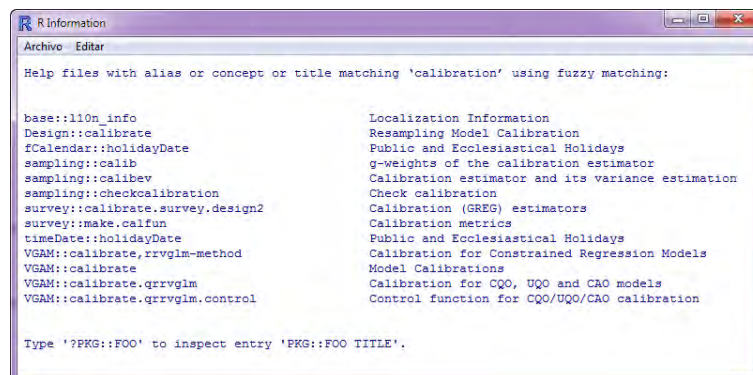
### Función `help` (ayuda)

*R* es un lenguaje funcional, esto es, realiza las tareas a través de funciones. La primera función que necesitamos conocer es la que nos proporciona la ayuda del programa, esta es “*help*”. Si escribimos *help* aparece la definición de la función *help*, la cual nos será útil conocer si queremos realizar alguna modificación en ella. Pero si lo que queremos es obtener el resultado de la función, debemos escribir entre paréntesis el conjunto de argumentos que quiere suministrar a la misma, o vacío si lo que se quiere es que aparezca la ayuda sin más. Una forma alternativa a ésta función es poner el carácter ? delante del elemento que queremos consultar.

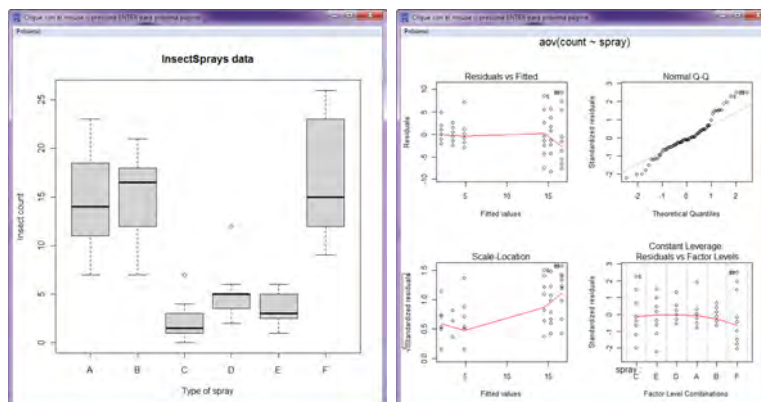
```
> help(mean)
```

Cuando queremos consultar funciones especificadas por caracteres especiales, el argumento deberá ir entre comillas para transformarlo en una cadena de caracteres. Por ejemplo: *help*(“[ ”). Otra forma de realizar la misma acción es con la función “*help.search*(“*median*”)”

```
> help.search("calibration")
```



Una opción interesante de la ayuda es que podemos encontrar ejemplos de uso. La forma de ejecutar los ejemplos es escribiendo “*example*( )” o “*demo*( )” si se quiere una demostración concreta. Por ejemplo con la sentencia “*example*(*InsectSprays*)” obtenemos una muestra de imágenes de diferentes gráficas que podemos realizar con los datos *InsectSprays*.



Otra opción muy útil de *R* es la ayuda a través de internet, si tecleamos `“help.start( )”` aparecerá en un navegador una ayuda en formato HTML con una información más actual. También podemos utilizar el buscador `“www.rseek.org”`, que al más estilo google nos proporciona todo el material publicado en la red sobre el tema buscado.

## Asignaciones

Al igual que ocurre con otros lenguajes de programación *R* asigna nombres a las operaciones. Esto lo conseguiremos mediante el símbolo `“< -”`, `“- >”` ó `“=”`.

Como para poder visualizar un dato renombrado hay de escribir el nombre después de haberlo asignado, es útil utilizar los `“;”` después de la asignación para ahorrarnos espacio, pero no es bueno abusar de ello por que sería más difícil interpretar códigos más extensos.

Hay que tener en cuenta que *R* utiliza determinados términos para referirse a algunas funciones por lo que lo mejor es evitar esos nombres a la hora de las asignaciones, por ejemplo `“c”` se utiliza para crear vectores o `“t”` que calcula la transpuesta de un conjunto de datos (vectores, matrices, dataframe,...), pero si nos confundimos no es dramático, ya que podemos arreglarlo.

Habitualmente a la hora de nombrar objetos en programación no existen reglas definidas pero se suele seguir una cierta pauta, por ejemplo se suele nombrar marcos de datos (dataframes) con la inicial en mayúscula, pero las variables siempre en minúscula. Si varias funciones hacen cosas parecidas a objetos distintos, se separan con `“;”` (o más fácil usando clases).

```
> rnorm(6)->x
> x
[1] 0.1364489 -0.5514473 0.2333607 1.1652428 0.4284878 0.5159063
> X<-rnorm(5)
> X
[1] -0.482520557 -0.951955735 1.537756423 0.839669367 0.772915316
> y<-1:15; y
[1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
> c
function (... , recursive = FALSE) .Primitive("c")
> c<-1:3
> c
[1] 1 2 3
```

En el caso que necesitemos poner notaciones o comentarios en *R*, estos se realizan poniendo delante del comentario el símbolo `“#”`. Recuerde la importancia de estos para explicar lo que estamos haciendo o simplemente recordar de que trata el programa o función que estemos implementando.

```
> #Hola
> #Espero que todo esto te sea útil
```

## Paquetes

Como ya hemos hablado antes, *R* consta de un sistema base de librerías y de un conjunto de paquetes adicionales que extienden su funcionalidad.

En esta sección vamos a ver como se anexan (“instalan”) los paquetes adicionales. Tenemos que tener en cuenta que la instalación depende del sistema operativo con el que funcionemos:

Según el sistema operativo que utilicemos tenemos varias formas distintas de instalar un paquete:

El método más sencillo consiste en hacerlo directamente desde *R*, mediante la función `install.packages( )`. Desde una ventana del sistema o desde Inicio-Ejecutar... Rcmdr INSTALL paquete. Y por último desde la interface de XEmacs.

Otra forma es descargar el archivo en formato zip (en el caso de windows), tar.gz (linux) o tgz (en el caso de Mac) desde la web de *R* e instalarlo en el directorio `R/.../library`. También podemos instalar un libro en cualquier otro directorio del ordenador e indicarle el camino del directorio mediante la función `library( "a" , lib.loc = "C : /..." )`, siendo “a” el nombre que le daremos a la biblioteca y “C : /...” la ruta hacia ella. (no es recomendable) aunque la forma más común es desde el menú Paquetes – > Instalar paquete(s) desde archivos Zip locales..., seleccionaremos la librería descargada y directamente se anexará en el programa.

Si nuestro sistema operativo es Linux, tenemos dos formas distintas de instalar un paquete:

La forma más cómoda es mediante los comandos `install.packages( )` ó `update.packages( )`. También nos permiten instalarlos si no eres root en este caso debemos especificar el `lib.loc`. R CMD INSTALL paquete-x.y.z.tar.gz, permite instalar paquetes, aunque uno no sea un root (especificar el directorio).

Después de haber instalado la librería en el programa, para poder utilizarla debemos de cargar el paquete. Esto lo vamos a hacer de la siguiente manera:

La forma más común es mediante desde el menú Paquetes – > cargar paquete..., aparecerá una lista con todos las librerías instaladas y nos pedirá que instalemos una de ellas, simplemente seleccionando una la tendremos operativa.

Con la función `search( )` podemos ver los libros que hay actualmente en memoria y con la función `library( )` una lista con los “*R* packages available” (librerías disponibles). Si dentro de los paréntesis ponemos el nombre de una de ellas, se cargará para su uso (aparecerá en segunda posición en la lista de libros en memoria).

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

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

## Funciones ls y rm

La función *ls* saca en pantalla los objetos almacenados en la memoria por el usuario, aunque sólo muestra los nombres de los mismos. Si se quiere listar solo aquellos objetos que contengan un carácter en particular “*ls(pat =)*” (usamos la opción “*pattern*”, que se puede abreviar como “*pat*”) y para restringir la lista a aquellos objetos que comienzan con este carácter utilizamos el símbolo exponente “*ls(pat = ^)*”. Tenga en cuenta que es recomendable conocer los elementos que R tiene en memoria, aparte de los que hemos memorizado nosotros. Esto se consigue con la orden “*ls(9)*”. La función “*ls.str( )*” muestra algunos detalles de los objetos en memoria. Al igual que antes la opción *pattern* se puede usar de la misma manera con *ls.str( )*.

Otra opción útil en esta función es “*max.level*” que especifica el nivel de detalle de visualización de objetos compuestos. Hay que tener en cuenta que “*ls.str( )*” muestra todos los detalles de los objetos en memoria, incluyendo las columnas de los marcos de datos (data frames), matrices y listas, lo cual puede generar una gran cantidad de información. Podemos evitar mostrar todos estos detalles con la opción “*max.level = -1*”.

```
> nombre<-"pepe"; x1<-5; x2<-100; b<-0.5
> ls()
[1] "b"      "nombre" "x1"      "x2"
> ls(pat="b")
[1] "b"      "nombre"
> ls(pat="^b")
[1] "b"
> ls.str()
b : num 0.5
nombre : chr "pepe"
x1 : num 5
x2 : num 100
> B <- data.frame(x1,x2,b)
> B
  x1 x2  b
1  5 100 0.5
> ls.str(pat="B")
B : 'data.frame':      1 obs. of  3 variables:
 $ x1: num 5
 $ x2: num 100
 $ b : num 0.5
```

Para borrar objetos almacenados en la memoria, utilizamos la función “*rm()*”, por ejemplo *rm(x)* elimina el objeto *x*; *rm(x, y)* elimina ambos objetos *x* e *y*, y “*rm(list = ls( ))*” elimina todos los objetos que estén en la memoria. Tenga en cuenta que las opciones mencionadas para la función *ls( )* pueden aplicarse para borrar selectivamente algunos objetos.

```

> rm(nombre)
> ls()
[1] "b" "B" "x1" "x2"
> #Hemos borrado "nombre"
> rm(list=ls(pat="x"))
> ls()
[1] "b" "B"
> #Hemos borrado las asignaciones que empiezan por x
> rm(list=ls())
> ls()
character(0)
> #Hemos borrado todo

```

## Función library

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

Por ejemplo `library()` devuelve una lista de los libros que hay en la biblioteca predefinida de R. `library(help="base")` devuelve una pequeña ayuda sobre el libro `base`. Si consultamos la lista de búsqueda observaremos los libros que hay actualmente en memoria.

```

> library()
Packages in library 'C:/PROGRA~1/R/R-210~1.1/library':

agricolae      Statistical Procedures for Agricultural Research
akima          Interpolation of irregularly spaced data
animation      Demonstrate Animations in Statistics
base           The R Base Package
bitops         Functions for Bitwise operations
boot           Bootstrap R (S-Plus) Functions (Canty)
caTools        Tools: moving window statistics, GIF, Base64, ROC AUC, etc.
class          Functions for Classification
cluster        Cluster Analysis Extended Rousseeuw et al.
...
...

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

> # Cargamos el paquete prob
> library(prob)
> # Ahora aparece segundo en la lista
> search()
[1] ".GlobalEnv"      "package:prob"        "package:stats"      "package:graphics"
[5] "package:grDevices" "package:utils"        "package:datasets"

```

```
[8] "package:methods"  "Autoloads"        "package:base"

> # Vemos la información del paquete prob
> library(help=prob)

      Information on package 'prob'

Description:

Package:          prob
Version:          0.9-2
Date:             2009-1-18
Title:            Elementary Probability on Finite Sample Spaces
Author:           G. Jay Kerns <gkerns@ysu.edu>
Maintainer:       G. Jay Kerns <gkerns@ysu.edu>
Depends:          base
Suggests:         combinat, fAsianOptions, hypergeo, VGAM
LazyLoad:         no
...
...
```

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

## Operaciones con números reales y complejos:

R es un lenguaje interpretado, es decir podemos escribir órdenes y R las ejecutará inmediatamente.

De hecho puede utilizar R como una calculadora, ya que es capaz de manejar todas las operaciones elementales: suma, resta, multiplicación, división, exponenciación, división entera y módulo, que se realizan mediante los símbolos: + , - , \* , / , ^ , % / % y %%.

Es importante saber que R no evalúa una expresión hasta que tiene la certeza de que se ha terminado su escritura. Si la expresión comienza con un paréntesis, deberemos cerrar la expresión con otro paréntesis sino se considerará que no terminó la acción, igualmente con corchetes, llaves etc. En general, para cualquier función, su escritura sin los paréntesis indica al lenguaje que debe devolver la definición, en tanto que al incluir paréntesis, le indica que intente ejecutarla.

```
> # R como calculadora
> 2+3
[1] 5
> 7*3
[1] 21
> 15/3
[1] 5
```

```
> 15/4
[1] 3.75
> 3^4
[1] 81
># Cociente de la división
> 15/%4
[1] 3
># Recto de la división
> 15%%4
[1] 3
```

También podemos hacer los mismos cálculos con números complejos (menos las operaciones división entera y módulo), en este caso utilizaremos la letra “i” para referirnos a la parte imaginaria.

```
> (3+3i)+(-3+2i)
[1] 0+5i
> (2+2i)*(3+i)
Error: objeto 'i' no encontrado
> # La unidad imaginaria i tiene que ir acompañado de un coeficiente
> (2+2i)*(3+1i)
[1] 4+8i
> 3+3i-3+2i
[1] 0+5i
> (2+2i)/(3+1i)
[1] 0.8+0.4i
> (2+2i)%/(3+1i)
Error: operación compleja no implementada
```

R proporciona funciones para hacer todo tipo de operaciones básicas: sumas, senos, cosenos, raíces, ...

- *sum( )* Suma de los elementos
- *sqrt( )* Raíz cuadrada.
- *abs( )* Valor absoluto.
- *sin( )*, *cos( )*, ... Funciones trigonométricas.
- *log( )*, *exp( )* Logaritmo y exponencial.
- *round( )*, *signif( )* Redondeo de valores numéricos.

```
> seq(1, 9, by = 2)
[1] 1 3 5 7 9
> rep(1:3,3)
[1] 1 2 3 1 2 3 1 2 3
> sqrt(81)
[1] 9
> abs(-9)
[1] 9
> sin(-2*pi)
[1] 2.449213e-16
> log(100)
[1] 4.60517
```



```
> exp(3)
[1] 20.08554
> round(2.345632)
[1] 2
> round(2.3456432, 3)
[1] 2.346
> signif(2.345432, 3)
[1] 2.35
```

## Operadores Comparativos y Lógicos:

Los operadores comparativos y lógicos son muy importantes a la hora de programar funciones ya que gracias a ellos podemos podremos hacer distinciones de datos, agrupaciones, etc. Veamos una pequeña introducción de ellos:

Los operadores comparativos o de relación interaccionan sobre cualquier tipo de elemento devolviendo uno o varios valores lógicos. Siempre necesitan dos elementos ( $a < b$ ) para poder actuar.

Los operadores lógicos pueden actuar sobre uno o dos objetos de tipo lógico, y pueden devolver uno (o varios) valores lógicos.

Relación		Lógicos	
<	menor que	!x	Negación
>	mayor que	x & y	conjunción
<=	menor o igual que	x && y	igual (*)
>=	mayor o igual que	x y	disyunción
==	igual	x  y	igual (*)
!=	diferente	xor(x,y)	O exclusivo (**)

(\*) Si se escriben dos símbolos repetidos, estos tienen el mismo significado que si apareciese uno, pero la diferencia consiste en que se evalúa primero la parte de la izquierda y, si ya se sabe el resultado (suponiendo que se pudiera calcular la expresión de la derecha) no se sigue evaluando, por lo que pueden ser más rápidos y eliminar errores.

(\*\*) Da como valor verdadero si uno y sólo un argumento es válido.

Para comparar “totalmente” dos objetos es necesario usar la función “*identical()*”.

```
> # Asignamos a x el valor 10
> x<-10; x
[1] 10
> # Le preguntamos si x es menor que 5
> x<5
[1] FALSE
> # Le preguntamos si x es mayor o igual que 5
> x>=5
[1] TRUE
> # Le preguntamos si x vale 5
> x==5
[1] FALSE
> # Le preguntamos si x es distinto de 5
> x!=5
[1] TRUE
```

```

> # Creamos dos vectores
> y<-1:3; z<-3:1
> # Le preguntamos si son iguales
> identical(y,z)
[1] FALSE
> # Vemos los elementos que coinciden
> y==z
[1] FALSE TRUE FALSE
> # Renombramos x e y
> x<-1:5
> y<-c(2,4,3,6,5)
> x==y
[1] FALSE FALSE TRUE FALSE TRUE
> x!=y
[1] TRUE TRUE FALSE TRUE FALSE
> x[x==y]
[1] 3 5
> x[x!=y]
[1] 1 2 4

```

Los vectores lógicos pueden utilizarse en expresiones aritméticas, en cuyo caso se transforman primero en vectores numéricos, de tal modo que F se transforma en 0 y T en 1.

En lógica una conjunción, &, es un “enunciado con dos o más elementos simultáneos”. Una lámpara eléctrica se enciende si hay corriente eléctrica, el interruptor esta conectado y el fusible esta bien, en cualquier otro caso la lámpara no se encenderá. En cambio la disyunción es un “enunciado con dos o más elementos optativos”. Por ejemplo “Puedes leer este manual o imprimirlo”, es una disyunción con dos elementos, mientras que “puedes leer este manual, imprimirlo o editarlo” es una disyunción con tres elementos.

Para dos entradas A y B, la tabla de verdad de la función conjunción y disyunción son las siguientes:

A	B	A & B	A	B	A   B
V	V	V	V	V	V
V	F	F	V	F	V
F	V	F	F	V	V
F	F	F	F	F	F

```

> A<-c(TRUE,FALSE);A
[1] TRUE FALSE
> !A
[1] FALSE TRUE
> B<-c(TRUE,TRUE);B
[1] TRUE TRUE
> xor(A,B)
[1] FALSE TRUE
> # A y B
> A & B
[1] TRUE FALSE
> A && B
[1] TRUE
> # A o B
> A | B
[1] TRUE TRUE
> A || B
[1] TRUE
> # Podemos comparar los elementos x e y creados anteriormente con:

```

```
> x<y
[1] TRUE TRUE FALSE TRUE FALSE
> x<=y
[1] TRUE TRUE TRUE TRUE TRUE
> x>=y
[1] FALSE FALSE TRUE FALSE TRUE
> x>y
[1] FALSE FALSE FALSE FALSE FALSE
> x[x>y]
integer(0)
> # No lo forma ningún elemento
> x[x<y]
[1] 1 2 4
> # Son los elementos de x que cumplen la condición
> y[x>y]
numeric(0)
> x[x<=y]
[1] 1 2 3 4 5
```

# Parte III

## Manipulación de Datos

# Manipulación de Datos

## Operaciones con Vectores.

El primer tipo de objeto que manejaremos es el vector (colección ordenada de elementos del mismo tipo). Podemos escribir vectores de varias maneras, utilizando la opción ":" (el vector comienza en el primer número suministrado y finaliza en el segundo o en un número anterior sin sobrepasarlo, tanto en orden ascendente como descendente) ó mediante la función de concatenación "c()".

```
> 1:10
[1] 1 2 3 4 5 6 7 8 9 10
> 10:1
[1] 10 9 8 7 6 5 4 3 2 1
> 1.5:7.5
[1] 1.5 2.5 3.5 4.5 5.5 6.5 7.5
> c(1,2,4,3)
[1] 1 2 4 3
> c(1:5)
[1] 1 2 3 4 5
> c(1:5,2,4,-3)
[1] 1 2 3 4 5 2 4 -3
```

Tenemos formas adicionales de crear vectores. Una de las más comunes es utilizando la función "seq(a,b,c)", que genera secuencias de números reales, donde el primer elemento indicaría el principio de la secuencia, el segundo el final y el tercero el incremento que se debe usar para generar la secuencia. Aunque también podemos poner la función de estas formas "seq(length = d, from = a, to = b)" o "seq(by = c, from = a, to = b)" siendo "d" la longitud del vector.

```
> seq(1,10,2)
[1] 1 3 5 7 9
> seq(from = 1, to = 20, by =2)
[1] 1 3 5 7 9 11 13 15 17 19
> seq(1,10,2)
[1] 1 3 5 7 9
> seq(from = 1, to = 10, by =2)
[1] 1 3 5 7 9
> # Si lo que queremos es 6 elementos
> seq(from = 1, to = 10, length =6)
[1] 1.0 2.8 4.6 6.4 8.2 10.0
> Con el argumento along creamos un vector con los índices del vector al que llamemos
> x<-c(1,2,1,3,4)
> seq(along=x)
[1] 1 2 3 4 5
> y<-25
> seq(along=y)
```

```
[1] 1
```

La función “*rep(a,b)*” que crea un vector con “b” elementos idénticos al valor “a”.

```
> # repetimos el elemento 3 diez veces
> rep(3,10)
[1] 3 3 3 3 3 3 3 3 3 3
> # repetimos el vector "a" 2 veces
> a=1:5; rep(a,2)
[1] 1 2 3 4 5 1 2 3 4 5
> # repetimos cada elemento del vector 3 veces de 5 en 5
> rep(1:3,rep(5,3))
[1] 1 1 1 1 1 2 2 2 2 2 3 3 3 3 3
> y<-rep(2,5)
> # repetimos cada elemento de "a" tantas veces como indica "y"
> rep(a,y)
[1] 1 1 2 2 3 3 4 4 5 5
> rep(a,a)
[1] 1 2 2 3 3 3 4 4 4 4 5 5 5 5 5
> x<-2
> # repetimos el elemento "x" 5 veces
> rep(x,length=5)
[1] 2 2 2 2 2
```

La función “*sequence()*” que crea una secuencias de enteros. Cada secuencia terminará en el número especificado como argumento.

```
> sequence(c(3,2))
[1] 1 2 3 1 2
> # Hemos creado una secuencia del vector 1:3 y del 1:2
> sequence(c(5,3))
[1] 1 2 3 4 5 1 2 3
> # Hemos creado una secuencia del vector 1:5 y del 1:3
> sequence(c(c(2,3),3))
[1] 1 2 1 2 3 1 2 3
> # Hemos creado una secuencia del vector 1:2, 1:3 y del 1:3
```

La función “*gl()*” (generador de niveles) genera series regulares de factores. Tiene la forma “*gl(a,b)*” donde “a” es el número de niveles (o clases) y “b” es el número de réplicas en cada nivel. Se pueden usar dos opciones: “length” para especificar el número de datos producidos, y “labels” para especificar los nombres de los factores.

```
> # Repetimos los dos niveles 6 veces con la etiqueta Hombre y Mujer
> gl(2, 6, labels = c("Hombre", "Mujer"))
[1] Hombre Hombre Hombre Hombre Hombre Hombre Mujer  Mujer  Mujer  Mujer
[11] Mujer  Mujer
Levels: Hombre Mujer
> # 20 elementos de dos niveles primero uno y luego el otro
> gl(2, 1, 20)
[1] 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2
Levels: 1 2
> # 20 elementos de dos niveles los dos a la vez
> gl(2, 2, 20)
[1] 1 1 2 2 1 1 2 2 1 1 2 2 1 1 2 2 1 1 2 2
Levels: 1 2
```

Finalmente, “`expand.grid()`” crea una tabla de datos con todas las combinaciones posibles de los elementos de los vectores o factores que proporcionemos como argumentos.

```
> expand.grid(edad=c(36,25), peso=c(75,60), sexo=c("Hombre","Mujer"))
  edad peso  sexo
1   36   75 Hombre
2   25   75 Hombre
3   36   60 Hombre
4   25   60 Hombre
5   36   75  Mujer
6   25   75  Mujer
7   36   60  Mujer
8   25   60  Mujer
> # Crea todas las combinaciones posibles
```

R puede escribir vectores con caracteres o números, pero siempre entiende los elementos como si fuesen del mismo tipo.

Un vector siempre está formado por elementos del mismo tipo, no pueden mezclarse números y cadenas de caracteres (se transformaría en cadenas de caracteres). Del mismo modo, si mezcla números reales y complejos, se entenderían como complejos.

```
> c("Hola", "Adios")
[1] "Hola" "Adios"
> c(1, 1+2i)
[1] 1+0i 1+2i
> c(1-1i, 2)
[1] 1-1i 2+0i
> dias.semana=c("Lunes","Martes","Miércoles","Jueves","Viernes","Sábado","Domingo")
> dias.semana
[1] "Lunes"      "Martes"      "Miércoles"    "Jueves"      "Viernes"      "Sábado"
[7] "Domingo"
```

Una opción de R es que podemos asignar nombres a los elementos de un vector mediante la función “`names`” (también se podrá utilizar para nombrar cualquier objeto).

```
> x<-1:7
> x
[1] 1 2 3 4 5 6 7
> names(x)<-c("Lunes","Martes","Miercoles","Jueves","Viernes","Sabado","Domingo")
> x
      Lunes      Martes Miercoles      Jueves      Viernes      Sabado      Domingo
        1         2         3         4         5         6         7
```

También podemos conocer o cambiar el modo o tipo de los elementos que forman el vector mediante la función “`mode`” (tomaría los valores: logical, numeric, complex, character, null, list, function, graphics, expression, name, frame, raw y unknown).

```
> y<-seq(from=3, to=11, by=2)
> y
[1] 3 5 7 9 11
> mode(y)
[1] "numeric"
> # Cambiamos y a complejo
> mode(y)<-"complex"
> y
[1] 3+0i 5+0i 7+0i 9+0i 11+0i
```

R como ocurría con números reales nos permite operar con vectores (dentro de las propiedades de estos). Podemos multiplicar vectores por escalares, vectores por vectores, potencias, sumas ... Teniendo en cuenta el uso del paréntesis y el número de elementos del vector, ya que nos llevaría a diferentes resultados e incluso a errores si no son múltiplos los tamaños entre sí.

```
> (1:5)*(2:6)
[1]  2  6 12 20 30
> (1:5)^2
[1]  1  4  9 16 25
> 1:5^2
[1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
> (1:5)(2:6)
Error: tentativa de aplicar una no-función
> (1:5)+(2:6)
[1]  3  5  7  9 11
> (1:5)+(2:4)
[1]  3  5  7  6  8
Mensajes de aviso perdidos
In (1:5) + (2:4) :
  longitud de objeto mayor no es múltiplo de la longitud de uno menor
```

Una de las opciones que hay que tener más cuidado es la potencial, ya que además de las formas en la que se puede escribir, también podemos elevar un vector a otro (en este caso debemos tener en cuenta el tamaño de cada uno).

```
> (1:5)^3
[1]  1  8 27 64 125
> (1:5)^(1:3)
[1]  1  4 27  4 25
Mensajes de aviso perdidos
In (1:5)^(1:3) :
  longitud de objeto mayor no es múltiplo de la longitud de uno menor
> (1:5)^(1:5)
[1]  1  4 27 256 3125
> (1:5)^(1:10)
[1]  1  4 27 256 3125 1 128 6561 262144 9765625
```

La longitud de un vector puede obtenerse con la función “length” que sirve además para definir la longitud del vector, haciéndolo más largo o más corto. En caso de hacerlo más largo, se completará con NA (valores faltantes).

```
> j<-4
> j
[1] 4
> length(j)
[1] 1
> # Asigno a j una longitud igual a 5
> length(j)<-5
> length(j)
[1] 5
> j
[1] 4 NA NA NA NA
> jj<-c("Pepe","Paco")
> jj
```



```
[1] "Pepe" "Paco"
> length(jj)
[1] 2
> # Asigno a jj una longitud igual a 5
> length(jj)<-5
> jj
[1] "Pepe" "Paco" NA      NA      NA
```

Podemos referirnos a un elemento que ocupa la posición “i” de un vector “x”, mediante “x[i]” o referirnos a un subconjunto mediante un subconjunto de subíndices. Si los subíndices son números naturales, hacen referencia a las posiciones dentro del vector. Si los subíndices son valores lógicos, se corresponden con los mismos elementos del vector, pero sólo hacen referencia a los que tienen un subíndice con valor lógico TRUE.

```
> aa<-(1:6)^3
> aa
[1] 1 8 27 64 125 216
> aa[5]
[1] 125
> aa[c(1,3,5)]
[1] 1 27 125
> # Asignamos el sexo a tres elementos ordenados
> sexo<-c("Mujer","Mujer","Hombre")
> sexo
[1] "Mujer" "Mujer" "Hombre"
> # Asignamos el nombre a tres elementos ordenados
> nombre<-c("Pepa","Pepita","Pepe")
> nombre
[1] "Pepa" "Pepita" "Pepe"
> # Preguntamos cual de los elementos son mujeres
> sexo=="Mujer"
Error: objeto 'Mujer' no encontrado
> # si es un carácter tiene que ir entre ""
> sexo=="Mujer"
[1] TRUE TRUE FALSE
> # Quiero solo los nombres de las mujeres
> nombre[sexo=="Mujer"]
[1] "Pepa" "Pepita"
```

Un factor es un vector de cadenas de caracteres que sirve para representar datos categóricos, aunque no solo incluye estos valores sino que también los diferentes niveles posibles de esta variable.

El atributo “levels” indica los valores numéricos posibles (es decir los caracteres diferentes que aparecen en el vector). Se utiliza, por ejemplo, para dividir una población en grupos.

La función “*factor*” se utiliza para codificar un vector como un factor. Creando tantos niveles como le indiquemos.

```
> # Tres niveles
> factor(1:3)
[1] 1 2 3
Levels: 1 2 3
> # Cinco niveles
> factor(1:3, levels=1:5)
[1] 1 2 3
```

```

Levels: 1 2 3 4 5
> # Tres niveles con etiquetas
> factor(1:3, labels=c("A","B","C"))
[1] A B C
Levels: A B C
> # Un vector de dos elementos con cuatro niveles
> aa<-factor(c(2,3),levels=2:5)
> aa
[1] 2 3
Levels: 2 3 4 5
> levels(aa)
[1] "2" "3" "4" "5"
> factor(c("Mujer","Mujer","Hombre"))
[1] Mujer Mujer Hombre
Levels: Hombre Mujer
> levels(factor(c("Mujer","Mujer","Hombre")))
[1] "Hombre" "Mujer"

```

Con la opción “exclude”, por ejemplo “*factor(1 : 5, exclude = 4)*” excluimos los valores de los niveles que necesitamos y con “ordered” especificamos si los niveles del factor están ordenados.

```

> # Cinco niveles excluyendo el cuarto
> factor(1:5, exclude=4)
[1] 1 2 3 <NA> 5
Levels: 1 2 3 5

```

Un error muy común es utilizar variables aparentemente numéricas en análisis estadísticos, por ejemplo n° telefónicos o códigos postales. Por ello antes de utilizar un vector con caracteres cualitativos o cuantitativos dentro de un análisis, hace falta convertirlo en un factor.

Existen funciones que nos permiten convertir diferentes clases de objetos a modos diferentes. Una situación frecuente es la conversión de factores a valores numéricos. En este caso, R realiza la conversión usando las expresiones “*as.numeric*” y “*as.character*”. Para realizar la conversión manteniendo los valores literales del factor, primero se debe convertir a carácter y después a numérico. Este procedimiento puede ser bastante útil si en un archivo una variable numérica también tiene valores no-numéricos.

```

> c.p<-c(18002,18194,18199)
> c.p
[1] 18002 18194 18199
> mode(c.p)
[1] "numeric"
> codigo.postal<-factor(c.p)
> codigo.postal
[1] 18002 18194 18199
Levels: 18002 18194 18199
> f<-factor(c(1,3))
> f
[1] 1 3
Levels: 1 3
> as.numeric(f)
[1] 1 2
> # No ha guardado los valores
> # Para guardarlos primero lo convertimos en carácter y luego en numérico
> as.numeric(as.character(f))
[1] 1 3

```

A continuación tenemos una tabla con un conjunto de funciones muy útiles a la hora de trabajar con vectores:

<code>mean(x)</code>	Media aritmética de los valores del vector x
<code>median(x)</code>	Mediana de los valores del vector x
<code>sum(x)</code>	Suma de los valores del vector x
<code>max(x)</code>	Máximo valor de los valores del vector x
<code>min(x)</code>	Mínimo valor de los valores del vector x
<code>range(x)</code>	Rango de los valores del vector x
<code>var(x)</code>	Cuasivarianza de los valores del vector x
<code>cov(x)</code>	covarianza de los valores del vector x
<code>cor(x,y)</code>	Coefficiente correlación entre los vectores x y y
<code>sort(x)</code>	Una versión ordenada de x
<code>rank(x)</code>	Vector de las filas de los valores en x
<code>order(x)</code>	Indica la posición que tendrían los valores ordenados de x
<code>quantile(x)</code>	Mínimo, primer cuartil, mediana, tercer cuartil y máximo de x
<code>cumsum(x)</code>	Vector formado por la frecuencia acumulada del vector x
<code>cumprod(x)</code>	Vector formado por el producto acumulado del vector x
<code>cummax(x)</code>	Vector de x tal que cada valor menor se sustituye por el anterior
<code>cummin(x)</code>	Vector de x tal que cada valor mayor se sustituye por el anterior
<code>pmax(x,y,z)</code>	Vector de longitud igual al más largo de x, y o z, formado por los valores máximos de cada posición
<code>pmin(x,y,z)</code>	Vector de longitud igual al más largo de x, y o z, formado por los valores mínimos de cada posición

Un ejemplo en el que podemos ver el comportamiento de cada una de las funciones:

```
> x<-c(1,2,3,4,5)
> y<-c(1,3,3,1,5)
> z<-c(1,3,3)
> mean(x)
[1] 3
> median(x)
[1] 3
> sum(x)
[1] 15
> max(x)
[1] 5
> min(x)
[1] 1
> range(x)
[1] 1 5
> # La cuasivarianza muestral
> var(x)
[1] 2.5
> # Veamos que es así
> n=length(x)
> sum((x-mean(x))^2)/(n-1)
[1] 2.5
> # La varianza sería
> ((n-1)/n)*var(x)
[1] 2
> sum((x-mean(x))^2)/(n)
[1] 2
```

```
> # La cuasicovarianza
> cov(x,y)
[1] 1.5
> sum((x-mean(x))*(y-mean(y)))/(n-1)
[1] 1.5
> # La covarianza sería
> ((n-1)/n)*cov(x,y)
[1] 1.2
> sum((x-mean(x))*(y-mean(y)))/(n)
[1] 1.2
> cor(x,y)
[1] 0.5669467
> sort(x)
[1] 1 2 3 4 5
> sort(y)
[1] 1 1 3 3 5
> rank(x)
[1] 1 2 3 4 5
> rank(y)
[1] 1.5 3.5 3.5 1.5 5.0
> order(y)
[1] 1 4 2 3 5
> quantile(x)
 0%  25%  50%  75% 100%
  1    2    3    4    5
> cumsum(y)
[1]  1  4  7  8 13
> cumprod(y)
[1]  1  3  9  9 45
> cummax(x)
[1] 1 2 3 4 5
> cummin(x)
[1] 1 1 1 1 1
> pmax(x,y,z)
[1] 1 3 3 4 5
> pmin(x,y,z)
[1] 1 2 3 1 3
```

## Operaciones con Matrices

Las matrices o variables indexadas (Arrays) son generalizaciones multidimensionales de vectores. De hecho, son vectores indexados por dos o más índices y que se imprimen de modo especial. Para crearlas utilizamos la función “matrix”.

Los parámetros principales de esta función son: data (vector que contiene los valores que formarán la matriz), nrow (número de filas), ncol (número de columnas).

```
> m1<-matrix(1,2,3)
> m1
      [,1] [,2] [,3]
[1,]    1    1    1
[2,]    1    1    1
> # Hemos formado una matriz de unos de dos filas y tres columnas
```

```

> # Tenemos que tener en cuenta el tamaño de la matriz
> m2<-matrix(1:4,3)
Mensajes de aviso perdidos
In matrix(1:4, 3) :
  la longitud de los datos [4] no es un submúltiplo o múltiplo del número de filas [3] en la matriz
> m2<-matrix(1:4,2)
> m2
      [,1] [,2]
[1,]     1     3
[2,]     2     4
> m2<-matrix(1:8,4)
> m2
      [,1] [,2]
[1,]     1     5
[2,]     2     6
[3,]     3     7
[4,]     4     8
> # Hemos formado una matriz con el vector 1:8 de cuatro filas
> m3<-matrix(1:8,ncol=4)
> m3
      [,1] [,2] [,3] [,4]
[1,]     1     3     5     7
[2,]     2     4     6     8
> # Hemos formado una matriz con el vector 1:8 de cuatro columnas

```

Si quiere dar nombres a las columnas (o a las filas) puede hacerlo asignando valores al parámetro “dimnames”, lista con los nombres de las filas y las columnas. Las componentes de la lista deben tener longitud 0 o ser un vector de cadenas de caracteres con la misma longitud que la dimensión de la matriz.

```

> # Renombramos las columnas
> matrix(1:9,3,3,dim=list(c(),c("A1","A2","A3"))))
      A1 A2 A3
[1,]   1  4  7
[2,]   2  5  8
[3,]   3  6  9
> # Renombramos las filas
> matrix(1:9,3,3,dim=list(c("a1","a2","a3"),c()))
      [,1] [,2] [,3]
a1      1     4     7
a2      2     5     8
a3      3     6     9
> # Renombramos filas y columnas
> matrix(1:9,3,3,dim=list(c("a1","a2","a3"),c("A1","A2","A3"))))
      A1 A2 A3
a1      1     4     7
a2      2     5     8
a3      3     6     9

```

Una operación muy común es hacer referencia a una submatriz o a un elemento de la matriz, se realiza indicando los índices de los elementos a los que se hace referencia. Podemos hacer referencia a una fila (vector) mediante *matriz*[*i*, ], con *i* el índice de la fila que queremos mostrar, o a una columna mediante *matriz*[ ,*j*] con *j* el índice de la columna que queremos mostrar. Si lo que queremos es un elemento concreto indicamos los dos índices *matriz*[*i*,*j*], por ejemplo con *matriz*[2,1], que da el valor 2º de la 1ª variable que coincide con el 1º valor de la 2ª variable. Un argumento útil en estas operaciones es la variable lógica “byrow” que indica si la matriz debe construirse por filas o por columnas (el valor predeterminado es F).

```

> # Introducimos los datos, peso, altura, edad.
> datos<-c(70,108,82,1.80,2.06,1.98,27,19,32)
> mm<-matrix(datos,ncol=3,dimnames=list(c("Peso","Altura","Edad"),c()),byrow=T)
> # Con byrow=T le hemos dicho que lea primero por filas
> mm
      [,1] [,2] [,3]
Peso   70.0 108.00 82.00
Altura  1.8   2.06  1.98
Edad   27.0  19.00 32.00
> mm<-matrix(datos,ncol=3,dimnames=list(c("Peso","Altura","Edad"),c()),byrow=F)
> # Con byrow=F le hemos dicho que lea primero por columnas, con lo que no se muestra correctamente
> mm
      [,1] [,2] [,3]
Peso     70 1.80  27
Altura  108 2.06  19
Edad    82 1.98  32

```

Podemos realizar operaciones con matrices de la misma forma que lo hacíamos con los vectores, es decir componente a componente: suma, resta, multiplicación por escalares, multiplicación elemento a elemento, división elemento a elemento, exponenciación, división entera y módulo, que se realizan mediante los símbolos: +, -, \*, /, ^, % / % y %%.

```

> M1<-matrix(1:6,2,3)
> M1
      [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
> M2<-matrix((1:6)^2,2,3)
> M2
      [,1] [,2] [,3]
[1,]    1    9   25
[2,]    4   16   36
> M1+M2
      [,1] [,2] [,3]
[1,]    2   12   30
[2,]    6   20   42
> # He sumado componente a componente
> M1*M2
      [,1] [,2] [,3]
[1,]    1   27  125
[2,]    8   64  216
> # He multiplicado componente a componente
> M2/M1
      [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
> # He dividido componente a componente
> M2%%M1
      [,1] [,2] [,3]
[1,]    0    0    0
[2,]    0    0    0
> M2*(3:6)
      [,1] [,2] [,3]
[1,]    3   45   75

```

```
[2,] 16 96 144
Mensajes de aviso perdidos
In M2 * (3:6) :
  longitud de objeto mayor no es múltiplo de la longitud de uno menor
> M2*(1:2)
      [,1] [,2] [,3]
[1,]    1    9   25
[2,]    8   32   72
```

También es posible realizar el producto matricial mediante el operador “%\*%”.

```
> A1<-matrix(1:6,2,3)
> A1
      [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
> B1<-matrix(1:12,3,4)
> B1
      [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
> A1%*%B1
      [,1] [,2] [,3] [,4]
[1,]   22   49   76  103
[2,]   28   64  100  136
> B1%*%A1
Error en B1 %*% A1 : argumentos no compatibles
> # Hay que tener en cuenta las dimensiones de las matrices
```

La función “*crossprod*” devuelve el producto matricial cruzado de dos matrices (la traspuesta de la primera matriz, multiplicada por la segunda). Si no especifica segunda matriz, R la toma igual a la primera, con lo que obtiene  $X^t X$ .

```
> A1<-matrix(1:6,2,3)
> A2<-matrix(c(1,1,2,3),2)
> A1
      [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
> crossprod(A1)
      [,1] [,2] [,3]
[1,]    5   11   17
[2,]   11   25   39
[3,]   17   39   61
> crossprod(A1,A2)
      [,1] [,2]
[1,]    3    8
[2,]    7   18
[3,]   11   28
```

La función “*outer*” realiza el producto exterior de dos matrices (o vectores) sobre una función dada. También puede usarse en forma de operador mediante % o %, en cuyo caso la función utilizada es el producto.

```

> x<-1:5
> y<-1.3
> outer(x,y)
      [,1]
[1,]  1.3
[2,]  2.6
[3,]  3.9
[4,]  5.2
[5,]  6.5
> x%o%y
      [,1]
[1,]  1.3
[2,]  2.6
[3,]  3.9
[4,]  5.2
[5,]  6.5

```

Mediante la función “*t()*” calculamos la matriz traspuesta.

```

> A1
      [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
> t(A1)
      [,1] [,2]
[1,]    1    2
[2,]    3    4
[3,]    5    6

```

La función “*diag()*” se puede usar para extraer o modificar la diagonal de una matriz o para construir una matriz diagonal.

```

> m1<-matrix(1,nr=2,nc=2)
> m1
      [,1] [,2]
[1,]    1    1
[2,]    1    1
> m2<-matrix(2,nr=2,nc=2)
> m2
      [,1] [,2]
[1,]    2    2
[2,]    2    2
> # Unimos las dos matrices por filas
> rbind(m1,m2)
      [,1] [,2]
[1,]    1    1
[2,]    1    1
[3,]    2    2
[4,]    2    2
> # Unimos las dos matrices por columnas
> cbind(m1,m2)
      [,1] [,2] [,3] [,4]
[1,]    1    1    2    2
[2,]    1    1    2    2

```



```

> diag(m1)
[1] 1 1
> diag(m2)
[1] 2 2
> diag(m1)=15
> m1
      [,1] [,2]
[1,]   15   1
[2,]    1  15
> # Creamos una matriz diagonal de orden 4
> diag(4)
      [,1] [,2] [,3] [,4]
[1,]    1    0    0    0
[2,]    0    1    0    0
[3,]    0    0    1    0
> v<-c(10,15,20)
> diag(v)
      [,1] [,2] [,3]
[1,]   10    0    0
[2,]    0   15    0
[3,]    0    0   20
> diag(3.6,nr=2,nc=5)
      [,1] [,2] [,3] [,4] [,5]
[1,]   3.6  0.0    0    0    0
[2,]   0.0  3.6    0    0    0
> # También podemos operar
> diag(rbind(m1,m2)%*%cbind(m1,m2))
[1] 226 226    8    8

```

La función “solve” invierte una matriz (si se le da un argumento) y resuelve sistemas lineales de ecuaciones (cuando se le dan dos). El primer argumento es una matriz que debe ser cuadrada no singular, y el segundo argumento (opcional) es un vector o matriz de coeficientes. También podemos utilizarla para resolver sistemas de ecuaciones (por ejemplo  $x + 2y = 1$ ,  $x + 3y = 2$ ), pasando el sistema a forma matricial.

```

> A2
      [,1] [,2]
[1,]    1    2
[2,]    1    3
> b1=c(1,2)
> b1
[1] 1 2
> solve(A2)
      [,1] [,2]
[1,]    3   -2
[2,]   -1    1
> solve(A2,b1)
[1] -1  1

```

La función “eigen” calcula los autovalores y autovectores de una matriz cuadrada, el resultado es una lista de dos componentes llamados valores y vectors. Con la función “det” calculamos el determinante de una matriz cuadrada. Además existen funciones como *svd()* que calcula la descomposición  $u*v$ .

```

> A2

```

```

      [,1] [,2]
[1,]    1    2
[2,]    1    3
> eigen(A2)
$values
[1] 3.7320508 0.2679492

$vectors
      [,1] [,2]
[1,] -0.5906905 -0.9390708
[2,] -0.8068982  0.3437238
> det(A2)
[1] 1

> svd(A2)
$d
[1] 3.8643285 0.2587772

$u
      [,1] [,2]
[1,] -0.5760484 -0.8174156
[2,] -0.8174156  0.5760484

$v
      [,1] [,2]
[1,] -0.3605967 -0.9327218
[2,] -0.9327218  0.3605967

```

Una opción interesante es construir matrices uniendo otras matrices, esto lo conseguimos con las funciones “cbind()” y “rbind()” que las une horizontalmente (modo columna) o verticalmente (modo fila) respectivamente.

```

> x1<-matrix(1:5,nr=3,nc=5)
> x1
      [,1] [,2] [,3] [,4] [,5]
[1,]    1    4    2    5    3
[2,]    2    5    3    1    4
[3,]    3    1    4    2    5
>
> i<-matrix(c(1:5,3:1),nr=3,nc=2)
> i
      [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    3
> # Uno x1 y i por columnas
> x1i<-cbind(x1,i)
> x1i
      [,1] [,2] [,3] [,4] [,5] [,6] [,7]
[1,]    1    4    2    5    3    1    4
[2,]    2    5    3    1    4    2    5
[3,]    3    1    4    2    5    3    3
> # Uno x1 y x2 por filas
> x11<-rbind(x1,x1)
> x11

```

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

Aunque en principio no están relacionadas las matrices con las tablas de frecuencias a partir de factores, podemos utilizar las primeras para expresar las segundas. Hemos visto que un factor define una tabla de entrada simple. Del mismo modo, dos factores definen una tabla de doble entrada, y así sucesivamente. La función `“table( )”` calcula tablas de frecuencias a partir de factores de igual longitud. Si existen  $k$  argumentos categóricos, el resultado será una variable  $k$ -indexada, que contiene la tabla de frecuencias.

```
> a=c(1,3,2)
> b=1:3
> b=1:3
> table(a,b)
  b
a  1 2 3
  1 1 0 0
  2 0 0 1
  3 0 1 0
```

## arrays (Variable multiindexada).

La generalización de los vectores y matrices son las variables multiindexadas, denominadas arrays, y de las cuales son casos particulares los vectores y matrices. Una variable indexada (array) es una colección de datos, por ejemplo numéricos, indexados por varios índices. R permite crear y manipular variables indexadas, por ejemplo para crear una variable multiindexada se utiliza la función `“array(data, dim, dimnames)”` donde `“dim”` es un vector de dimensiones. Además podemos hacer referencia a cualquier subconjunto de la misma, de modo similar a las matrices.

```
> # array se comporta de la misma manera que matrix
> x1
  [,1] [,2] [,3] [,4] [,5]
[1,]  1   4   2   5   3
[2,]  2   5   3   1   4
[3,]  3   1   4   2   5
> x2
  [,1] [,2] [,3] [,4] [,5]
[1,]  1   4   2   5   3
[2,]  2   5   3   1   4
[3,]  3   1   4   2   5
> i<-array(c(1:5,3:1),dim=c(3,2))
> i
  [,1] [,2]
[1,]  1   4
[2,]  2   5
[3,]  3   3
> z<-array(1:5,c(2,3,3))
```

```

> z
, , 1
      [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    1
, , 2
      [,1] [,2] [,3]
[1,]    2    4    1
[2,]    3    5    2
, , 3
      [,1] [,2] [,3]
[1,]    3    5    2
[2,]    4    1    3
> z[1,,]
      [,1] [,2] [,3]
[1,]    1    2    3
[2,]    3    4    5
[3,]    5    1    2
> z[1,1,]
[1] 1 2 3
> z[1,1,1]

```

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

La función “aperm” permite trasponer una variable multiindexada indicando la propia variable como primer argumento y el vector de permutaciones de índices como segundo.

```

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

```

## Listas.

Una lista se construye con la función list que devuelve un objeto de tipo lista con tantos componentes como argumentos se le suministren y es utilizado para devolver el resultado de una función.

```

> dias.semana=c("Lunes","Martes","Miercoles","Jueves","Viernes","Sabado","Domingo")
> dias.semana
[1] "Lunes"      "Martes"      "Miercoles"   "Jueves"      "Viernes"     "Sabado"
[7] "Domingo"
> list(A=dias.semana,B=1:7)
$A
[1] "Lunes"      "Martes"      "Miercoles"   "Jueves"      "Viernes"     "Sabado"

```

```
[7] "Domingo"
$B
[1] 1 2 3 4 5 6 7
```

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

```
> list(A=dias.semana,B=1:7)$A
[1] "Lunes"      "Martes"      "Miércoles"   "Jueves"      "Viernes"     "Sábado"      "Domingo"
> list(A=dias.semana,B=1:7)$B
[1] 1 2 3 4 5 6 7
> list(A=dias.semana,B=1:7)[[2]]
[1] 1 2 3 4 5 6 7
> list(A=dias.semana,B=1:7)[[1]]
[1] "Lunes"      "Martes"      "Miércoles"   "Jueves"      "Viernes"     "Sábado"      "Domingo"
```

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

```
> lista1<-list(A=dias.semana,B=1:7)
> lista2<-list(lista1,C="Hola", D=matrix(1:8,2))
> lista1
$A
[1] "Lunes"      "Martes"      "Miercoles"   "Jueves"      "Viernes"     "Sabado"      "Domingo"

$B
[1] 1 2 3 4 5 6 7

> lista2
[[1]]
[[1]]$A
[1] "Lunes"      "Martes"      "Miercoles"   "Jueves"      "Viernes"     "Sabado"      "Domingo"

[[1]]$B
[1] 1 2 3 4 5 6 7

$C
[1] "Hola"

$D
      [,1] [,2] [,3] [,4]
[1,]    1    3    5    7
[2,]    2    4    6    8

> lista2[[1]]
$A
[1] "Lunes"      "Martes"      "Miercoles"   "Jueves"      "Viernes"     "Sabado"      "Domingo"

$B
```

```
[1] 1 2 3 4 5 6 7

> lista2[[3]]
      [,1] [,2] [,3] [,4]
[1,]    1    3    5    7
[2,]    2    4    6    8
> lista2[[3]][2,4]
[1] 8
```

## Hoja de datos (data.frame).

Una hoja de datos (Data frame) es una lista que pertenece a la clase “data.frame”.

Hay restricciones en las listas que pueden pertenecer a esta clase, en particular: las componentes deben ser vectores (numéricos, cadenas de caracteres, o lógicos), factores, matrices numéricas, listas u otras hojas de datos. Las matrices, listas, y hojas de datos contribuyen a la nueva hoja de datos con tantas variables como columnas, elementos o variables posean, respectivamente.

Los vectores numéricos y los factores se incluyen sin modificar, los vectores no numéricos se fuerzan a factores cuyos niveles son los únicos valores que aparecen en el vector. Los vectores que constituyen la hoja de datos deben tener todos la misma longitud, y las matrices deben tener el mismo tamaño.

La diferencia fundamental con la matriz de datos es que este objeto no tiene por qué estar compuesto de elementos del mismo tipo. Los objetos pueden ser vectores, factores, matrices, listas e incluso hojas de datos.

```
> datos1<-data.frame(Peso=c(90,120,56), Altura=c(1.90,1.87,1.70))
> datos1
  Peso Altura
1   90   1.90
2  120   1.87
3   56   1.70
> datos2<-data.frame(datos1, Sexo=c("Hombre","Hombre","Mujer"))
> datos2
  Peso Altura  Sexo
1   90   1.90 Hombre
2  120   1.87 Hombre
3   56   1.70  Mujer
```

Este vector se puede transformar en un factor. Es importante identificar el uso de corchetes individuales o dobles ya que implica diferencia en los resultados.

```
> is.vector(datos1)
[1] FALSE
> is.matrix(datos1)
[1] FALSE
> is.data.frame(datos1)
[1] TRUE
> is.factor(datos1[2])
[1] FALSE
> is.factor(datos2[[3]])
[1] TRUE
> datos2[3]
```

```

Sexo
1 Hombre
2 Hombre
3 Mujer

```

Para introducir un vector de nombres como tales, sin transformarlo en factores, debe utilizar la función *I()*.

```

> nombres<-c("Pepe","Paco","Pepita")
> nombres
[1] "Pepe"  "Paco"  "Pepita"
> datos3<-data.frame(datos2, Nombres=I(nombres))
> datos3
  Peso Altura  Sexo Nombres
1   90   1.90 Hombre   Pepe
2  120   1.87 Hombre   Paco
3   56   1.70 Mujer  Pepita

```

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

Para realizar modificaciones en un data frame es muy útil la función *transform*.

```

> datos3
  Peso Altura  Sexo Nombres
1   90   1.90 Hombre   Pepe
2  120   1.87 Hombre   Paco
3   56   1.70 Mujer  Pepita
> subset(datos3,select=c(Sexo,Nombres))
  Sexo Nombres
1 Hombre   Pepe
2 Hombre   Paco
3  Mujer  Pepita
> subset(datos3,subset=c(Sexo=="Mujer"))
  Peso Altura  Sexo Nombres
3   56   1.70 Mujer  Pepita
> transform(datos3,logPeso=log(Peso))
  Peso Altura  Sexo Nombres logPeso
1   90   1.90 Hombre   Pepe 4.499810
2  120   1.87 Hombre   Paco 4.787492
3   56   1.70 Mujer  Pepita 4.025352
> transform(datos3,IMC=Peso/(Altura)^2)
  Peso Altura  Sexo Nombres   IMC
1   90   1.90 Hombre   Pepe 24.93075
2  120   1.87 Hombre   Paco 34.31611
3   56   1.70 Mujer  Pepita 19.37716

```

Hay una serie de funciones que permiten comprobar si un objeto es de un tipo determinado, todas comienzan por *is.*, o cambiar el objeto a un tipo concreto, funciones que comienzan por *as.*

```

> x<-1:10
> is.vector(x)
[1] TRUE
> is.data.frame(x)

```

```
[1] FALSE
> x<-as.data.frame(x)
> is.data.frame(x)
[1] TRUE
```

La función de R “*data.frame*” concatena todas las variables en un solo conjunto de datos, también podemos guardar en un archivo dicho conjunto de datos utilizando la función “*write.table*”.

El formato del archivo guardado es .CSV (comma separated variables, variables separadas por comas), el cual es un formato de texto muy fácil de leer con cualquier editor de texto o con Excel. El archivo se puede ser leído en R a través de la función “*read.csv*”.

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

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

La orden “*attach(abc)*” conecta la hoja de datos colocándola en la segunda posición de la trayectoria de búsqueda y, supuesto que no existen variables denominadas *u*, *v* o *w* en la primera posición; *u*, *v* y *w* aparecerán como variables por sí mismas. Sin embargo, si realiza una asignación a una de estas variables, como por ejemplo “*u ← v + w*” no se sustituye la componente *u* de la hoja de datos, sino que se crea una nueva variable, *u*, en el directorio de trabajo, en la primera posición de la trayectoria de búsqueda, que enmascarará a la variable *u* de la hoja de datos. Para realizar un cambio en la propia hoja de datos, basta con utilizar la notación \$, como en “*lentejas\$u ← v + w*”. Este nuevo valor de la componente *u* no sería visible de modo directo hasta que desconecte y vuelva a conectar la hoja de datos.

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

Hay que tener en cuenta que la trayectoria de búsqueda puede almacenar un número finito y pequeño de elementos, por tanto no debemos conectar una misma hoja de datos más de una vez. Del mismo modo, es conveniente desconectar una hoja de datos cuando termine de utilizar sus variables.

## Lectura y escritura de datos.

Una opción tan interesante como necesaria es la de lectura y escritura de ficheros de datos. R permite importar datos desde cualquier tipo de fichero de datos básico, tal como bases de datos, archivos excel, de SPSS, de Minitab, de STATA, de documentos de texto, archivos .dat, etc. La orden de lectura de datos es “*read.table*”, con ella y con sus argumentos podemos leer los ficheros e indicar el modo en el que ellos están contruidos. La forma correcta y más simple de utilizar la orden sería de la siguiente forma “*read.table(“ruta del archivo/datos1.txt”, header=TRUE, sep=, na.strings=“NA”, dec=“.”)*”, con ella estamos diciendo que lea los datos “*datos1.txt*.” que se encuentra en tal ruta (separada por /), que dichos datos tienen cabecera, es decir que existe una primera fila con los nombres de las variables (si no tienen los datos cabecera el argumento sería *header=FALSE*). Con *na.strings=“NA”* le decimos que los valores faltantes los tome como NA (nulos). El separador decimal de los datos que tenemos es el punto en vez de la coma. Hay otros argumentos que se le

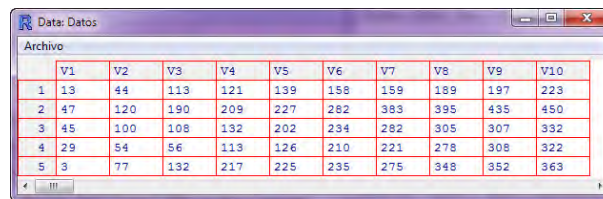


pueden añadir a la función tales como indicar el tipo de valores de los datos (lógicos, enteros, etc.) el número de columnas, etc.

```
> # Leemos el archivo s.txt y lo llamo Datos
> Datos <- read.table("C:/Users/jmcontreras/Desktop/MALETA/s.txt",
  header=FALSE, sep=" ", na.strings="NA", dec=".")
```

Con la función “View” visualizamos los datos que hemos cargado en memoria anteriormente.

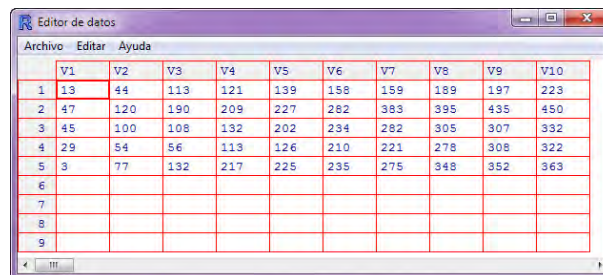
```
> View(Datos)
```



	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10
1	13	44	113	121	139	158	159	189	197	223
2	47	120	190	209	227	282	383	395	435	450
3	45	100	108	132	202	234	282	305	307	332
4	29	54	56	113	126	210	221	278	308	322
5	3	77	132	217	225	235	275	348	352	363

Con “fix” podemos modificar los datos cambiando elementos o añadiéndolos.

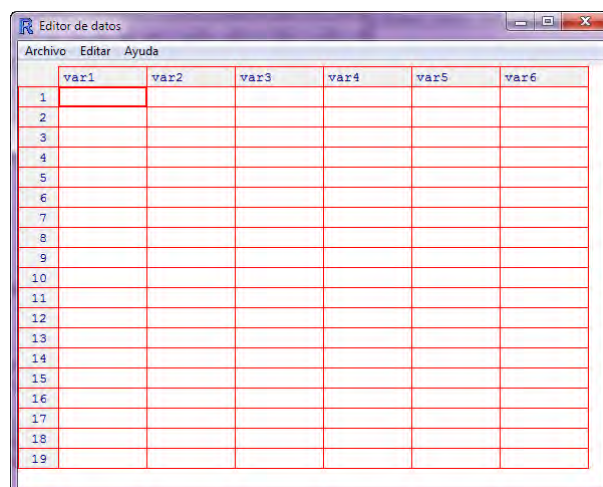
```
> fix(Datos)
```



	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10
1	13	44	113	121	139	158	159	189	197	223
2	47	120	190	209	227	282	383	395	435	450
3	45	100	108	132	202	234	282	305	307	332
4	29	54	56	113	126	210	221	278	308	322
5	3	77	132	217	225	235	275	348	352	363
6										
7										
8										
9										

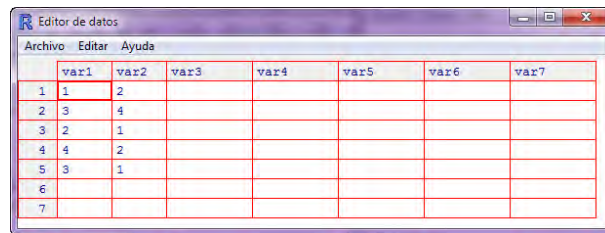
Para escribir datos utilizamos la función “write” con ella podemos crear ficheros a partir de los datos que vamos calculando con R o con datos que nosotros vamos introduciendo. Por ejemplo:

```
> # Con la siguiente orden creamos una tabla en blanco a la que nombramos como Datos1
> Datos1 <- edit(as.data.frame(NULL))
```



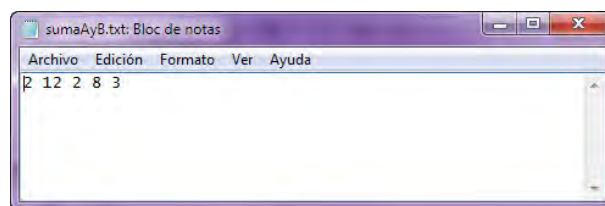
	var1	var2	var3	var4	var5	var6
1						
2						
3						
4						
5						
6						
7						
8						
9						
10						
11						
12						
13						
14						
15						
16						
17						
18						
19						

Vamos a crear dos variables de datos:



	var1	var2	var3	var4	var5	var6	var7
1	1	2					
2	3	4					
3	2	1					
4	4	2					
5	3	1					
6							
7							

```
> # Vemos si realmente tenemos los datos
> Datos1
  var1 var2
1    1    2
2    3    4
3    2    1
4    4    2
5    3    1
> # Vemos las columnas y las renombramos
> Datos1$var1->A
[1] 1 3 2 4 3
> Datos1$var2->B
[1] 2 4 1 2 1
> A
[1] 1 3 2 4 3
> B
[1] 2 4 1 2 1
> # Creamos un fichero vector con la suma de los dos elementos
> A+B
[1] 3 7 3 6 4
> # Lo guardamos en un fichero .txt que se llame sumaAyB
> write(A*B,"sumaAyB.txt")
```



	12	2	8	3
1	12	2	8	3

La función `write` tiene como argumentos los siguientes `write(x, file = "data", ncolumns = , append = FALSE, sep = " ")`, donde `x` el nombre de la variable que queremos exportar, `file` es el nombre y tipo de fichero que crearemos, `ncolumns` el número de columnas y `sep` es el separador (en nuestro caso espacios en blanco). El argumento "append" es el más útil a la hora de simulaciones ya que con el podemos conservar los datos del fichero que tuviésemos anteriormente si `append=TRUE`, solo que los nuevos se añadirán a los anteriores en filas posteriores. Si `append = FALSE` "machacaremos" los datos anteriores por los nuevos.

# Parte IV

## Funciones

# Funciones

Como hemos visto anteriormente, las funciones permiten realizar las diferentes acciones. Existen funciones definidas (pueden ser modificadas), pero lo más importante es que R permite crear objetos del modo `function`, es decir nos permite construir nuevas funciones de R que realicen tareas que no estaban definidas en el momento de instalar el programa, que además pueden utilizar a su vez en expresiones posteriores ganando considerablemente en potencia, comodidad y elegancia. Estas nuevas funciones se incorporan al lenguaje y se utilizan posteriormente como las previamente existentes. Para leer la definición, basta con escribir el nombre de la función sin paréntesis.

*Nota: Para obtener ayuda sobre qué es lo que hace una función utilizaremos la función `help`. Si necesita ejecutar un programa del sistema operativo puede utilizar la función `system`, que permite incluso salir al sistema operativo con la orden `system("cmd")`.*

Para obtener información del sistema utilizamos `shell`, por ejemplo en `shell("dir")`.

Las funciones tradicionales en cualquier lenguaje de uso matemático y estadístico, están definidas. Entre ellas se encuentran, entre otras, `abs`, `sqrt`, `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `exp`, `log`, `log10`, `min`, `max`, `sum`, `prod`, `length`, `mean`, `range`, `median`, `var`, `cov`, `summary`, `sort`, `rev`, `order`.

```
> sin(1:5)
[1] 0.8414710 0.9092974 0.1411200 -0.7568025 -0.9589243
> sin(pi)
[1] 1.224606e-16
> sin(pi/2)
[1] 1
> sum(1:5)
[1] 15
> prod(1:5)
[1] 120
```

## Definición de una función

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

$$\text{nombre} <- \text{function}(\text{arg1}, \text{arg2}, \dots) \text{expresion}$$

La expresión es una fórmula o grupo de fórmulas que utilizan los argumentos para calcular su valor. El valor de dicha expresión es el valor que proporciona R en su salida y éste puede ser un simple número, un vector, una gráfica, una lista o un mensaje.

Para definir una función debe realizar una asignación de la forma anterior, donde expresión es una expresión de R que utiliza los argumentos “arg i” para calcular un valor que es devuelto por la función.

El uso de la función es normalmente de la forma NombreDeFuncion(expr 1,expr 2,...) y puede realizarse en cualquier lugar en que el uso de una función sea correcto.

## Ejemplo de creación de una función

```
> función.suma<-function(A=10,B=5)
+ A+B
> función.suma()
[1] 15
> función.suma
function(A=10,B=5)
A+B
> función.suma(10,3)
[1] 13
> función.suma(12,30)
[1] 42
```

La función que acabamos de definir no se refiere a la suma de dos números, sino a la suma de dos objetos, que podrán ser vectores, matrices, variables indexadas, etc. Aunque si le asignamos dos valores concretos los sumará, como hemos visto en el ejemplo anterior.

Si desea poner de manifiesto el valor devuelto, puede utilizar la función “return” que termina la función y devuelve su argumento. Cuando se encuentra esta función se detiene la ejecución y se devuelven los valores indicados.

```
> primera.función<-function(A=1,B=2)
+ {
+ #devuelve A+B
+ return(A+B)
+ }
> primera.función()
[1] 3
> primera.función(2,4)
[1] 6
```

Es posible acceder a los argumentos de una función mediante la función “formals”.

```
> #Almacenamos los valores actuales en argumentos
> formals(primer.función)
$A
[1] 1

$B
[1] 2

> formals(primer.función)->argumentos
> argumentos
$A
[1] 1
```

```
$B
[1] 2

> # Modificamos los valores
> formals(primer.función)=alist(X=,Y=-1)
> primer.función
function (X, Y = -1)
{
  return(A + B)
}
> # Recuperamos los valores almacenados
> formals(primer.función)=argumentos
> primer.función
function (A = 1, B = 2)
{
  return(A + B)
}
```

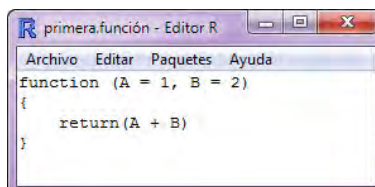
También podemos acceder al cuerpo de la misma mediante la función “body”.

```
> # Con body almacenamos los valores actuales en cuerpo
> body(primer.función)->cuerpo
> cuerpo
{
  return(A + B)
}
> # Modificamos los valores
> body(primer.función)<-expression(A*B)
> primer.función
function (A = 1, B = 2)
A * B
> # Recuperamos los valores almacenados
> body(primer.función)=cuerpo
> primer.función
function (A = 1, B = 2)
{
  return(A + B)
}
```

La orden más interesante para modificar funciones es “edit”, con ella podemos editar la función que necesitamos y adaptarla a nuestras necesidades.

```
> edit(primer.función)
function (A = 1, B = 2)
{
  return(A + B)
}
```

Aparecerá una ventana editable donde podremos cambiar la orden a nuestro antojo.



## Funciones elementales

Como pretendemos que este sea una introducción al uso de R sobretodo para su uso estadístico, empezaremos con la función más común en este area: la media.

### Función media

Es una función de un solo argumento (si no se especifica lo tomaremos por NA (valor nulo) y el resultado sería NA). En caso de que se le suministre un argumento, no se comprueba si es válido o no, sino que suponiendo que es un vector, se eliminan del mismo los elementos correspondientes a NA. Para ello se utiliza la negación ! y la condición de ser un valor no disponible.

```
> media<-function(x=NA)
+ {
+   x<-x[!is.na(x)]
+   sum(x)/length(x)
+ }
> media(c(2,4,1,3,6,7))
[1] 3.833333
> media(c(2,4,1,3,6,NA))
[1] 3.2
```

Aunque R ya tiene implementada como vimos anteriormente la orden “mean” que calcula la media de los valores que le indiquemos.

```
> mean(c(2,4,1,3,6,7))
[1] 3.833333
```

### Función varianza

Tenemos la opción de calcular la varianza de dos maneras:

La definición de la varianza poblacional como  $Varianza = \frac{\sum (x - \bar{x})^2}{n}$ :

```
> Varianza<-function(x=NA)
+ {
+   n<-length(x)
+   v<-sum((x-(sum(x)/n))^2)/n
+   return(v)
+ }

> Varianza(1:4)
[1] 1.25
> Varianza(c(2,4,1,3,6,7))
[1] 4.472222
```

Hay que tener en cuenta que R tiene implementada la orden “var” que calcula la varianza muestral o cuasivarianza, por lo que divide por n-1 en vez de por n.

```
> var(c(2,4,1,3,6,7))
[1] 5.366667
```

Si lo que queremos es calcular la cuasivarianza muestral tenemos que modificar la anterior función.

```

> Cuasivarianza<-function(x=NA)
+ {
+   n<-length(x)
+   cv<-sum((x-(sum(x)/n))^2)/(n-1)
+   return(cv)
+ }

> Cuasivarianza(c(2,4,1,3,6,7))
[1] 5.366667
> var(c(2,4,1,3,6,7))
[1] 5.366667
> # Vemos que ambas coinciden

```

Otra forma de calcular la varianza es mediante la formula de la relación entre los momentos.

```

> Koning<-function(x=NA)
+ {
+   n<-length(x)
+   v<-mean(x^2)-(mean(x))^2/(n)
+   return(v)
+ }
> # o
> Koning<-function(x=NA)
+ {
+   n<-length(x)
+   v<-sum(x^2)/n-(sum(x)/n)^2
+   return(v)
+ }

> Koning(c(2,4,1,3,6,7))
[1] 4.472222
> Koning(1:5)
[1] 2

```

Pero si lo que queremos es trabajar con “var” pero queremos que la varianza sea la poblacional lo mejor es multiplicar por  $\frac{n-1}{n}$ .

```

> var(1:5)
[1] 2.5
> # calculamos la longitud del vector
> n<-length(1:5)
> n
[1] 5
> var(1:5)*(n-1)/n
[1] 2
> # Comprobamos que coinciden
> Varianza(1:5)
[1] 2

```

## Función Desviación Típica

La definición de la Desviación Típica poblacional es  $DT = \sqrt{\frac{\sum (x-\bar{x})^2}{n}}$ :  
 Por tanto la definición es:



```
> DT<-function(x=NA)
+ {
+   n<-length(x)
+   v<-sqrt(sum((x-(sum(x)/n))^2)/n)
+   return(v)
+ }
> DT(1:3)
[1] 0.8164966
> DT(c(1,3,4,2,6,4))
[1] 1.598611
```

Si como con la varianza lo que queremos es que haga referencia solo a la muestra tenemos que dividir por n-1 en vez de por n.

```
> DTm<-function(x=NA)
+ {
+   n<-length(x)
+   v<-sqrt(sum((x-(sum(x)/n))^2)/(n-1))
+   return(v)
+ }
> DTm(1:3)
[1] 1
> DTm(c(1,3,4,2,6,4))
[1] 1.75119
```

## Función Covarianza

Definimos la covarianza poblacional de dos conjuntos de datos como  $Covarianza = \frac{\sum (x-\bar{x})(y-\bar{y})}{n}$ :

```
> Covarianza<-function(x=NA,y=NA)
+ {
+   n<-length(x) # o la de y
+   cv<-sum((x-(sum(x)/n))^2*(y-(sum(y)/n))^2)/n
+   return(cv)
+ }
> Covarianza(1:4,2:5)
[1] 2.5625
> x<-c(1,3,5,2,3,4)
> y<-c(3,2,1,6,3,2)
> Covarianza(x,y)
[1] 4.046296
```

Si lo que queremos es la covarianza muestral lo que tenemos que hacer es dividir por n-1 en vez de n.

```
> CovarianzaM<-function(x=NA,y=NA)
+ {
+   n<-length(x) # o la de y
+   cvm<-sum((x-(sum(x)/n))^2*(y-(sum(y)/n))^2)/(n-1)
+   return(cvm)
+ }
> CovarianzaM(x,y)
[1] 4.855556
```

## Elementos útiles a la hora de programar

### Operadores de relación

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

Estructuras condicionales.

Son aquellas que, según el resultado de una comparación, realizan una u otra acción. La primera estructura es if (condición) acción1 [else acción2]

```
> # Creamos una función que calcule el logaritmo de un número
> logaritmo<-function(x)
+ {
+   if(is.numeric(x)&& min(x)!=0)
+     log(x)
+   else{stop("x no es numérico o es cero")}
+ }
> logaritmo(3)
[1] 1.098612
> logaritmo(10)
[1] 2.302585
> logaritmo(e)
Error en logaritmo(e) : objeto 'e' no encontrado
> logaritmo("e")
Error en logaritmo("e") : x no es numérico o es cero
> logaritmo(exp(1))
[1] 1
```

La segunda estructura es ifelse(condición, acción en caso cierto, acción en caso falso).

```
> Inverso<-function(x)
+ ifelse(x==0,NA,1/x)
> Inverso(-2:3)
[1] -0.5000000 -1.0000000      NA  1.0000000  0.5000000  0.3333333
> Inverso(-10:10)
[1] -0.1000000 -0.1111111 -0.1250000 -0.1428571 -0.1666667 -0.2000000 -0.2500000 -0.3333333
[9] -0.5000000 -1.0000000      NA  1.0000000  0.5000000  0.3333333  0.2500000  0.2000000
[17]  0.1666667  0.1428571  0.1250000  0.1111111  0.1000000
```

La tercera estructura es switch(expresión,[valor-1]=acción-1,...,[valor-n]=acción-n).

```
> n<-3
> switch(n,"Uno", "Dos","Tres")
[1] "Tres"
> n<-1
> switch(n,"Uno", "Dos","Tres")
[1] "Uno"
> n<-5
> Decidir<-function(x)
+ switch(x,n=cat("Has dicho n minúscula\n"),
```

```

+ N=cat("Has dicho N mayúscula\n")
+ ,cat("Dime n o N"))
> Decidir
function(x)
switch(x,n=cat("Has dicho n minúscula"),
N=cat("Has dicho N mayúscula")
,cat("Dime n o N"))
> Decidir("N")
Has dicho N mayúscula
> Decidir("n")
Has dicho n minúscula
> Decidir("P")
Dime n o N
> n<-"Dos"
> switch(n,"Uno"=1, "Dos"=2, "Tres"=3)
[1] 2
>
> n<-"Dos"
> switch(n,"Uno"=1, "Dos"=2, "Tres"=3, "Otro")
[1] 2
> n<-"Cinco"
> switch(n,"Uno"=1, "Dos"=2, "Tres"=3, "Otro")
[1] "Otro"

```

## Estructuras de repetición definida e indefinida

Hay tres tipos de estructuras:

for (variable in valores) acción, while (condición) acción y repeat acción.

La estructura “for” asigna a variable cada uno de los valores y realiza la acción para cada uno. La estructura “while” evalúa la condición y mientras esta es cierta se evalúa la acción. La estructura “repeat” evalúa la acción indefinidamente. En los tres casos, el valor del ciclo completo es el de la última evaluación de la acción.

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

La estructura “next” indica que debe terminarse la iteración actual y pasar a la siguiente. La estructura “break” indica que debe terminarse el ciclo actual.

```

> for(i in -5:5)
+ {cat(i,"\t", i^3,"\n")
+ }
-5      -125
-4      -64
-3      -27
-2      -8
-1      -1
0        0
1         1
2         8
3        27
4        64
5       125
> for(i in -5:5)

```

```
+ {if(i==0) next; cat(i,"t", i^2, "\n")}
+ }
-5      25
-4      16
-3      9
-2      4
-1      1
1       1
2       4
3       9
4       16
5       25
> for(i in -5:5)
+ {if(i==0) break; cat(i,"t", i^2, "\n")}
+ }
-5      25
-4      16
-3      9
-2      4
-1      1
> i<-4
> while(i<=10){print(i^2);i=i+1}
[1] 16
[1] 25
[1] 36
[1] 49
[1] 64
[1] 81
[1] 100
> i<-5
> repeat{print(i^2);i=i+1; if(i==10)break}
[1] 25
[1] 36
[1] 49
[1] 64
[1] 81
```

## Invisible

La función “invisible” indica que un objeto no debe mostrarse. La sintaxis es “invisible(x)” siendo x cualquier objeto que es devuelto a su vez por la función. Esta función se usa muy a menudo para no presentar directamente informaciones devueltas por funciones que, sin embargo, pueden ser utilizadas. Para hacerlo visible basta con utilizar paréntesis.

```
> x<-2
> (x<-2)
[1] 2
> {x<-2}
> invisible(x)
> x
[1] 2
> (x)
[1] 2
> {x}
```

```
[1] 2
> {x<-1;y<-3;z<-12}
> ({x<-1;y<-3;z<-12})
[1] 12
> x<-y<-z
> x
[1] 12
> y
[1] 12
```

## Ejemplos de funciones

### Factorial de un número

```
> factorial<-function(n)
+ {
+ f<-1
+ if(n>1)
+ for(i in 1:n)
+ f<-f*i
+ return(f)
+ }
> factorial(3)
[1] 6
> factorial(25)
[1] 1.551121e+25
> factorial(0)
[1] 1
```

Dos procedimientos alternativos a la función anterior serían los siguientes. La función factorial.3 consume recursos debido a la recursividad y la función factorial.2 los consume porque genera todos los factores antes de multiplicarlos, consumiendo memoria.

```
> factorial.3<-function(n)
+ {
+ f<-1
+ while(n>0)
+ {
+ f<-n*f
+ n<-n-1
+ }
+ return(f)
+ }
> factorial.3(3)
[1] 6
> factorial.3(25)
[1] 1.551121e+25
> factorial.3(0)
[1] 1
> # o
> factorial.2<-function(n)
+ {
+ if(n>1)
```

```
+ f<-n*factorial.2(n-1)
+ else f<-1
+ return(f)
+ }
> factorial.2(3)
[1] 6
> factorial.2(25)
[1] 1.551121e+25
> factorial.2(0)
[1] 1
```

## Progresión aritmética

Podemos construir tres funciones para implementarla: la primera corresponde a la forma explícita, la segunda a la forma recursiva y la tercera a la forma recursiva vectorial.

```
> arit.1<-function(n=1,a1=1,d=1) a1+d*(n-1)
> arit.2<-function(n=1,a1=1,d=1)
+ {
+   if(n>1)
+   {return(arit.2(n-1,a1,d)+d)}
+   else
+   {return(a1)}
+ }
> arit.3<-function(n=1,a1=1,d=1)
+ {
+   A=1:n
+   A[1]=a1
+   for(i in 2:n)A[i]=A[i-1]+d
+   return(A[n])
+ }
> arit.1(10)
[1] 10
> arit.2(10)
[1] 10
> arit.3(10)
[1] 10
> # Si queremos comprobar el tiempo que emplea cada una...
```

## Progresión geométrica

```
> P.geometrica<-function(n=1,a1=1,r=1)
+ {
+   a1+r^(n-1)
+ }
> P.geometrica
function(n=1,a1=1,r=1)
{
  a1+r^(n-1)
}
> P.geometrica()
[1] 2
> P.geometrica(2,3,2)
[1] 5
```

```
> # otra forma sería
> P.geometrica2<-function(n=1,a1=1,r=1)
+ {
+   if(n>1)
+   {
+     P.geometrica(n-1,a1,r)+r
+   }
+   else
+   {
+     a1
+   }
+ }
> P.geometrica2(2,3,2)
[1] 6
> # Una última forma sería
> P.geometrica3<-function(n=1,a1=1,r=1)
+ {
+   if(n==1) return(a1)
+   A=1:n
+   A[1]=a1
+   for(i in 2:n) A[i]=A[i-1]*r
+   return(A[n])
+ }
> P.geometrica3(2,3,2)
[1] 6
```

## Factorial de un número

```
> n<-10
> # Se crean un par vectores vacíos
> pares<-c()
> impares<-c()
> for(i in 1:n){ #
+   if(i%%2==0) pares<-c(pares,i) # Si es par
+   else impares<-c(impares,i)} # Si es impar
> pares
[1] 2 4 6 8 10
> impares
[1] 1 3 5 7 9
```

# Parte V

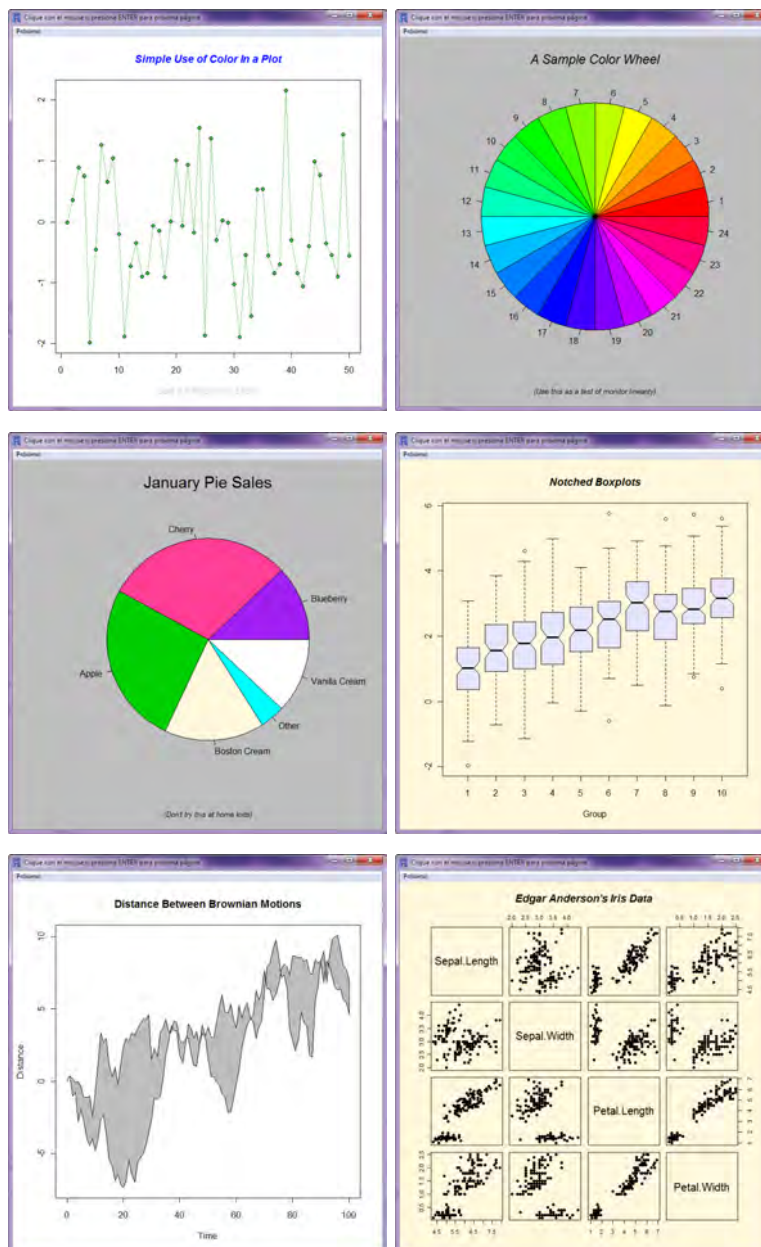
## Dispositivos gráficos



# Dispositivos gráficos

Los gráficos disponibles en R son todos de gran calidad y poseen una versatilidad muy grande. Para hacernos una idea (a modo de ejemplo), podemos ejecutar la función demo:

```
> demo("graphics")
```



Para poder crear gráficos es necesario inicializar un dispositivo gráfico (graphic device), y si hay varios, elegir uno de ellos.

```
> win.graph()
> x11()
> windows()
```



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

## Función x11.

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

## Función pictex.

La función pictex genera gráficos que pueden utilizarse en TEX y en LATEX. Para utilizarlo en LATEX, debe incluir la biblioteca pictex.

## Funciones bmp, jpeg y png...

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

Pero también podemos guardar nuestra gráfica en una gran variedad de formatos desde el menú “Archivo → Guardar como”, eligiendo el tipo de formato.

Las funciones que guardan el formato serían:

Función	Tipo de formato de salida
pdf(“grafica.pdf”)	pdf
win.metafile(“grafica.wmf”)	windows metafile
png(“grafica.png”)	png
jpeg(“grafica.jpg”)	jpeg
bmp(“grafica.bmp”)	bmp
postscript(“grafica.ps”)	postscript

Function Output to pdf(“mygraph.pdf”) pdf file win.metafile(“mygraph.wmf”) windows metafile png(“mygraph.png”) png file jpeg(“mygraph.jpg”) jpeg file bmp(“mygraph.bmp”) bmp file postscript(“mygraph.ps”) postscript file  
See input/output for details.

## funciones gráficas básicas

Algunos de los gráficos más usuales pueden generarse a partir de básicas tales como:

- *abline*( ) Función que añade una o más líneas rectas.
- *plot*( ) Función genérica para representar en el plano xy puntos, líneas, etc.
- *barplot*( ) Diagramas de barras.
- *pie*( ) Diagramas de sectores.
- *hist*( ) Histogramas.
- *boxplot*( ) Diagramas de box-and-whisker.
- *stripplot*( ) Similares a *boxplot*( ) con puntos.
- *sunflowerplot*( ) Representación en el plano xy de diagramas de girasol.
- *qqnor*( ) Diagramas de cuantil a cuantil frente a la distribución normal.
- *qqplot*( ) Diagramas de cuantil a cuantil de dos muestras.
- *qqline*( ) Representa la línea que pasa por el primer y el tercer cuantil.

Para crear gráficos más completos podemos ayudarnos de las siguientes funciones:

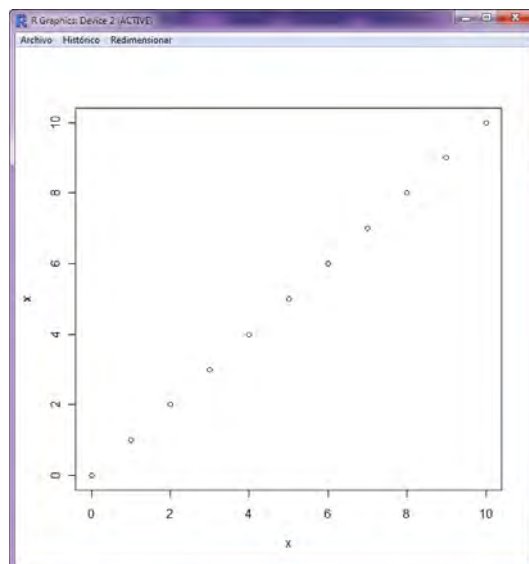
- *lines*( ) Añade líneas a un gráfico.
- *points*( ) Añade puntos a un gráfico.
- *segments*( ) Añade segmentos a un gráfico.
- *arrows*( ) Añade flechas a un gráfico.
- *polygons*( ) Añade polígonos a un gráfico.
- *rect*( ) Añade rectángulos a un gráfico.
- *abline*( ) Añade una recta de pendiente e intersección dada.
- *curve*( ) Representa una función dada.

Veamos a ver algunos de las funciones descritas anteriormente:

### plot

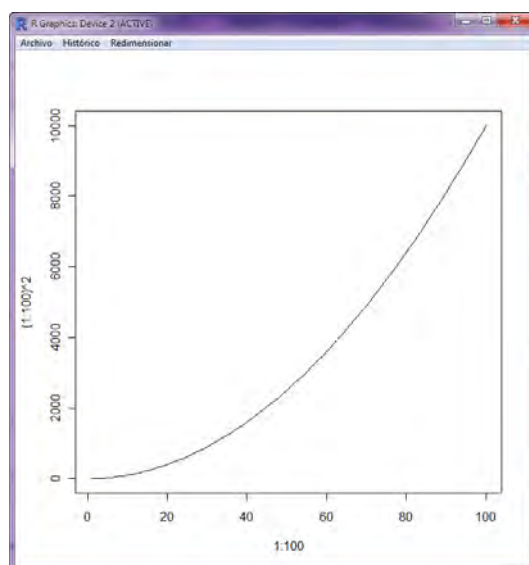
*plot* es una función genérica que crea un gráfico en el dispositivo gráfico actual. Además existen funciones específicas en las que funciona de modo especial, como por ejemplo para *data.frame*, *lm*, etc.

```
> x<-seq(-10,10)
> x
[1] -10 -9 -8 -7 -6 -5 -4 -3 -2 -1 0 1 2 3 4 5 6 7 8
[20] 9 10
> plot(x,x,xlim=c(0,10),ylim=c(0,10))
```

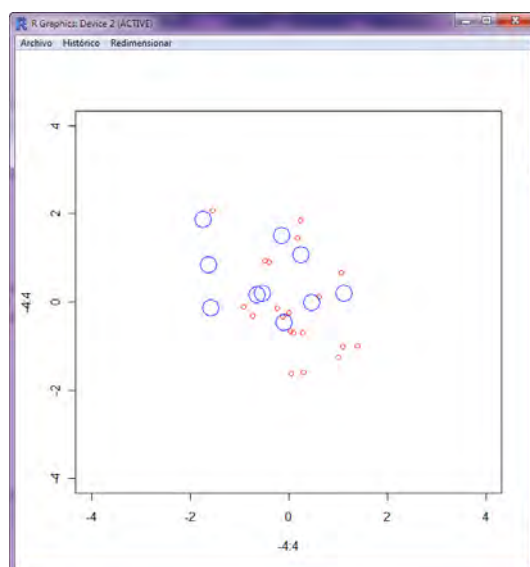


Lo complicamos un poco más.

```
> plot(1:100, (1:100)^2, type="l")
```

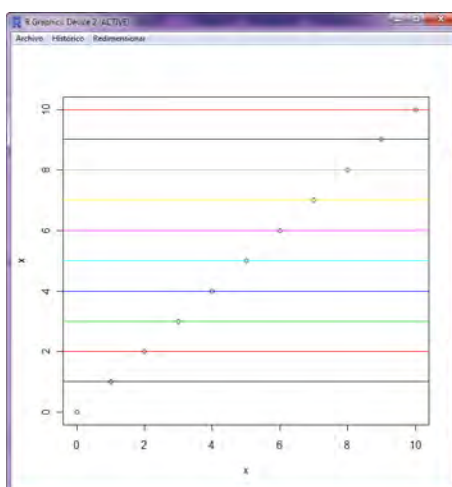


```
# Generamos un gráfico de dos variables comprendidas entre -4 y 4
> plot(-4:4, -4:4, type = "n")
# Representamos 20 números aleatorios de una distribución normal en rojo
> points(rnorm(20), rnorm(20), col = "red")
# Representamos 10 números aleatorios de una distribución normal en azul y más grandes
> points(rnorm(10), rnorm(10), col = "blue", cex = 3)
```



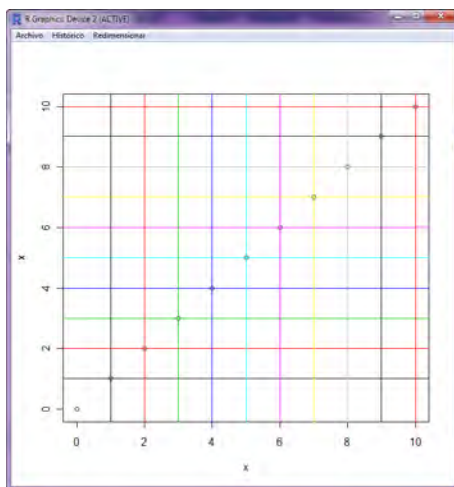
Utilizamos la función `abline` para crear un conjunto de líneas que separen en filas la gráfica.

```
> for(i in 1:10)  
+   abline(h=i,col=i)
```

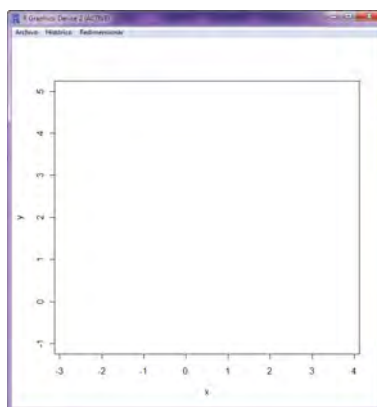


Y también para crear un conjunto de líneas que separen las líneas por columnas.

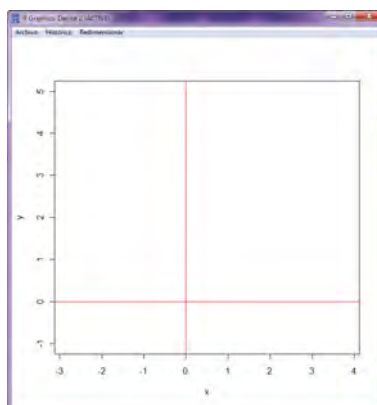
```
> for(i in 1:10)  
+   abline(v=i,col=i)
```



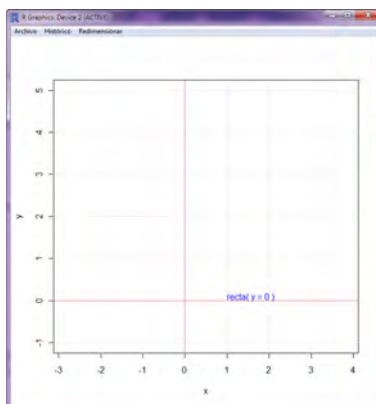
```
> # Creamos una gráfica de dos variables x e y con x entre -2 y 3 e y entre -1 y 5
> plot(c(-2,3), c(-1,5), type = "n", xlab="x", ylab="y", asp = 1)
```



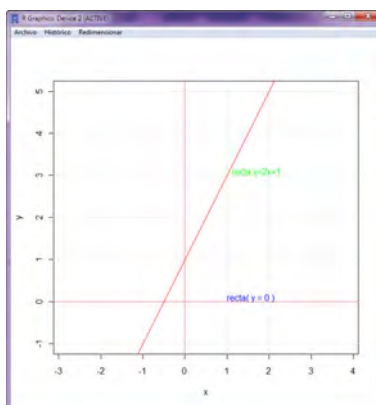
```
> # Creamos un eje xy de color rojo
> abline(h=0, v=0, col = "red")
```



```
> # Asignamos una leyenda (recta y = 0), de color azul en el par (1,0)
> text(1,0, "recta( y = 0 )", col = "blue", adj = c(0, -.1))
> # Creamos una red para distinguir los pares
> abline(h = -1:5, v = -2:3, col = "lightgray", lty=3)
```



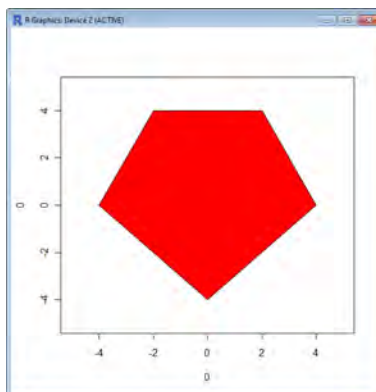
```
> # Creamos una recta de pendiente 2 y termino independiente 1
> abline(a=1, b=2, col = 2)
> # Añadimos una leyenda para nombrar la recta y=2x+1
> text(1,3, "recta y=2x+1", col= "green", adj=c(-.1,-.1))
```



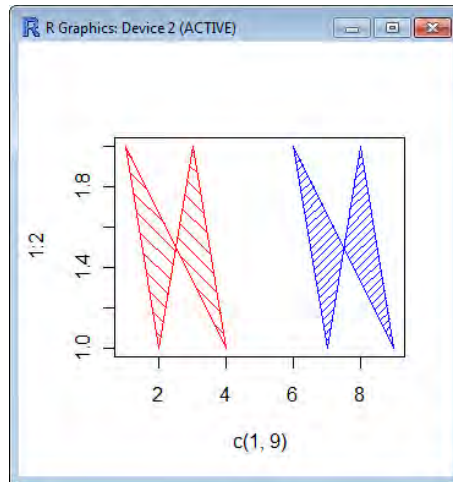
## polygon

La función “polygon” dibuja un polígono cuyos vértices se dan a partir de dos vectores x e y.

```
> plot(0,0,type="n",xlim=c(-5,5),ylim=c(-5,5))
> x <-c(0,4,2,-2,-4)
> y <-c(-4,0,4,4,0)
> polygon(x,y,col="red",border="black")
```

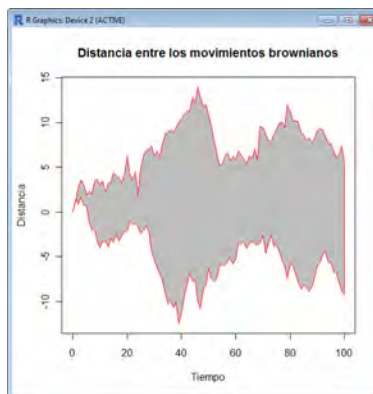


```
> polygon(1:9, c(2,1,2,1,NA,2,1,2,1),
+ density=c(10, 20), angle=c(-45, 45), col=c("red","blue"))
```



También podemos utilizar “polygon” para crear gráficos más complejos como:

```
> n <- 100
> xx <- c(0:n, n:0)
> yy <- c(c(0,cumsum(stats::rnorm(n))), rev(c(0,cumsum(stats::rnorm(n)))))
> plot (xx, yy, type="n", xlab="Tiempo", ylab="Distancia")
> polygon(xx, yy, col="gray", border = "red")
> title("Distancia entre los movimientos brownianos")
```

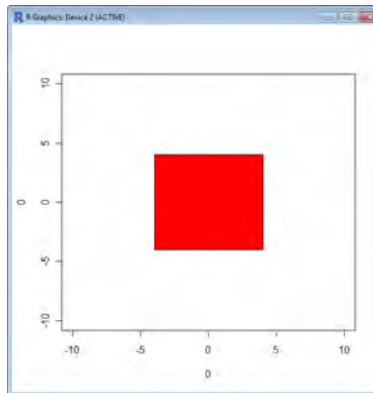


## rect

Con la función “rect” creamos rectángulos del color que deseemos.

```
> # Creamos un marco para la gráfica con x entre -10 y 10 e
> # y en entre -10 y 10
> plot(0,0,type="n",xlim=c(-10,10),ylim=c(-10,10))
> # Creamos un rectángulo con x de -4 a 4 e y de -4 a 4 de color rojo
> rect(-4,-4,4,4,col="red",border="black")
```

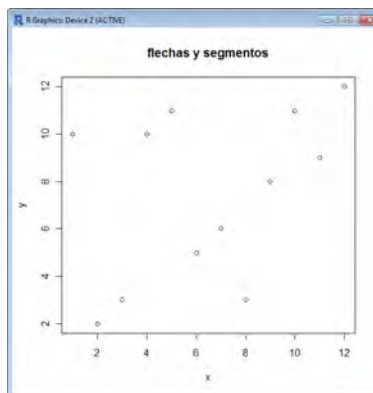




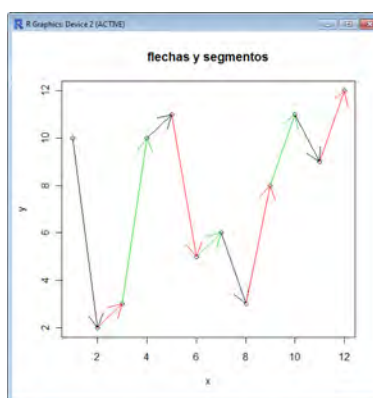
## arrows y segments

Con la función “arrows” creamos flechas entre puntos de un par de variables. Con “segments” creamos figuras o segmentos a partir de los valores.

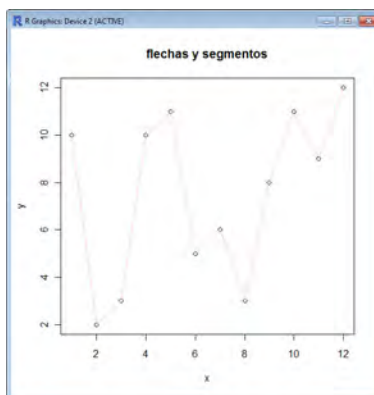
```
> x <- 1:12
> y <- c(10,2,3,10,11,5,6,3,8,11,9,12)
> plot(x,y, main="flechas y segmentos")
```



```
> # Flechas
> arrows(x[s], y[s], x[s+1], y[s+1], col= 1:3)
```

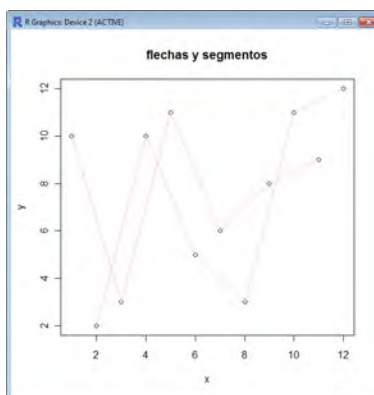


```
> # Segmentos
> # Volvemos a crear el diagrama de puntos
> plot(x,y, main="flechas y segmentos")
> segments(x[s], y[s], x[s+1], y[s+1], col= 'pink')
```



También podemos jugar con los valores de las x e y.

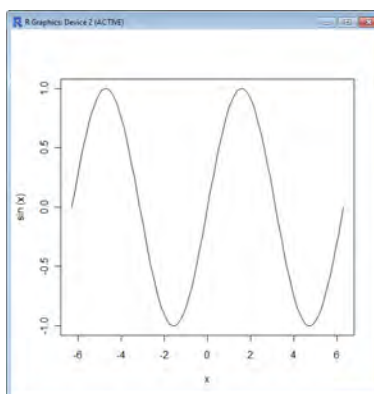
```
> # Volvemos a crear el diagrama de puntos
> plot(x,y, main="flechas y segmentos")
> # Unimos los pares
> segments(x[s], y[s], x[s+2], y[s+2], col= 'pink')
```



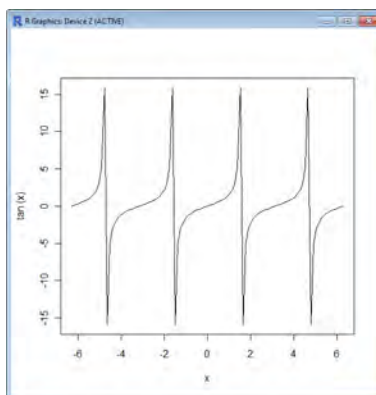
## curve

Con la función “curve” creamos funciones no lineales, tanto polinomiales como trigonométricas.

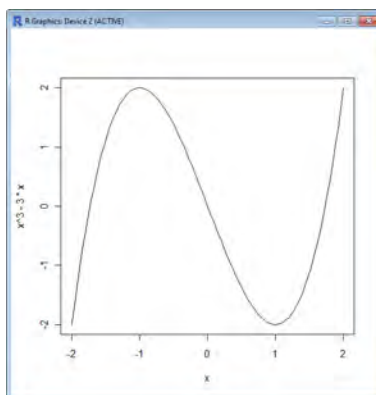
```
> curve(sin, -2*pi, 2*pi)
```



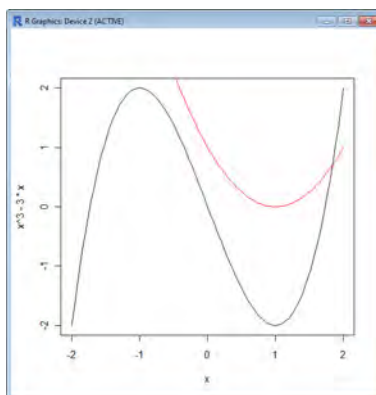
```
> curve(tan)
> # Si borramos la gráfica anterior sale diferente
```



```
> # Polinomiales
> curve(x^3-3*x, -2, 2)
```

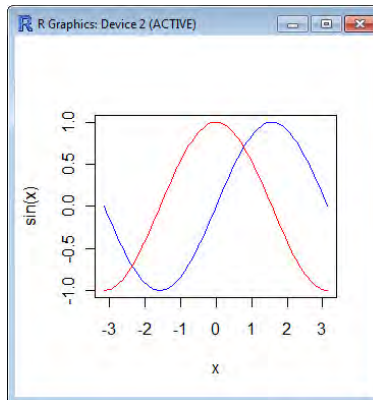


```
> # Le adjuntamos una parábola
> curve(x^2-2*x+1, add = TRUE, col = "red")
```



Un argumento interesante es add, con el podemos adjuntar varias gráficas a la vez.

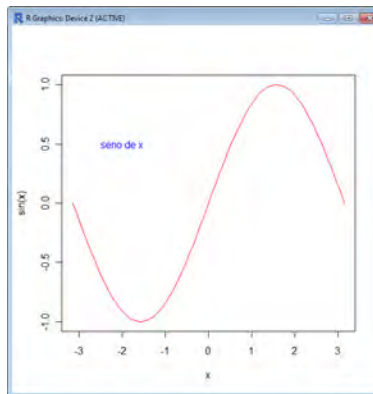
```
> curve(sin(x), -pi, pi, col="blue")
> curve(cos(x), -pi, pi, add=TRUE, col="red")
```



## Función text.

Una función muy útil es “text” ya que nos permite añadir texto a un gráfico existente.

```
> curve(sin(x), -pi, pi, col="red")
> text(-2, 0.5, "seno de x", col="blue")
```



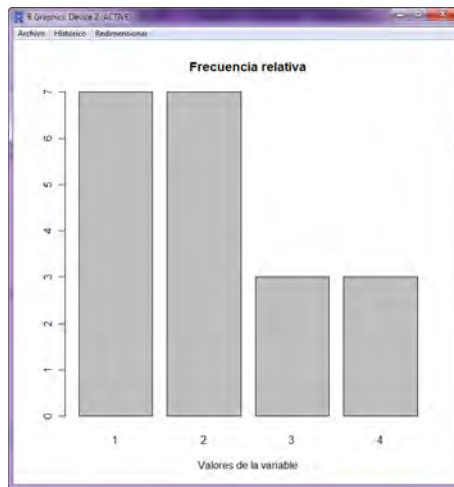
## Gráficos estadísticos

### barplot

Los diagramas de barras los creamos con la función *barplot*

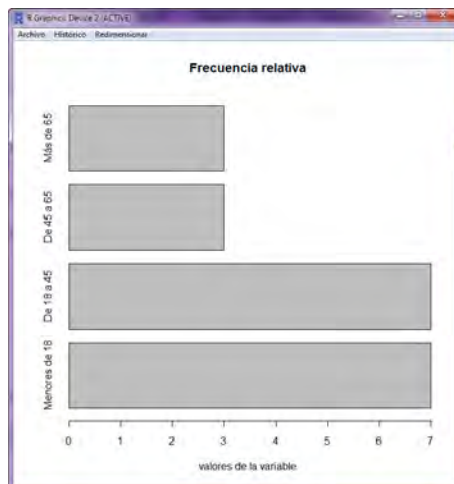
Empezamos con un ejemplo fácil, supongamos que hemos recogido unos datos que hemos codificado con escalas del 1 al 4. Queremos hacer un diagrama de barras con los resultados para cada una de los valores de las variables.

```
> x<-c(1,2,3,1,2,1,2,4,1,3,2,4,1,2,3,1,2,4,2,1)
> # Calculamos la frecuencia acumulada de cada una de las variables
> table(x)
x
1 2 3 4
7 7 3 3
> freq.x<-table(x)
> # Creamos el diagrama de barras con las leyendas
> barplot(freq.x, main="Frecuencia relativa", xlab="Valores de la variable")
```



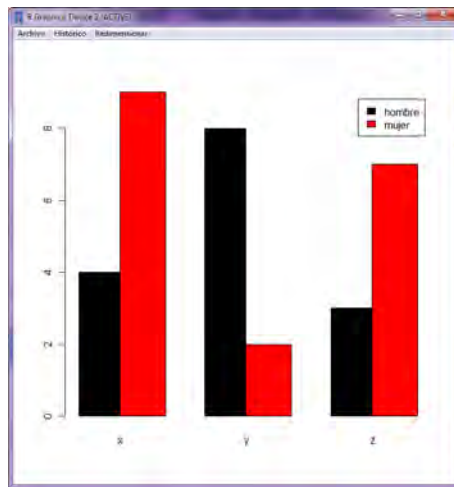
Vamos a ponerle nombre a las variables y cambiar las barras de vertical a horizontal.

```
> barplot(frec.x, main="Frecuencia relativa", xlab="valores de la variable",
+ horiz=TRUE, names.arg=c("Menores de 18", "De 18 a 45", "De 45 a 65",
+ "Más de 65"))
```



Vamos ahora a complicar un poco los gráficos haciendo gráficos de barras acumulados:

```
># Representamos un diagrama de barras con las variables x,y y z;
># agrupadas una al lado de la otra (para juntarlas cambiamos TRUE por FALSE);
># de grosor 45,50 y35;
># con los colores 1 y 2 (negro y rojo) y
># con la leyenda hombre y mujer
> barplot(height = cbind(x = c(4, 9), y = c(8, 2), z = c(3, 7) ), beside = TRUE,
+ width = c(45, 50, 35), col = c(1, 2), legend.text = c("hombre", "mujer"))
```



## pie

Los diagramas de sectores para datos cualitativos los creamos con la función *pie*. Por ejemplo un diagrama básico sería:

```
> # Diagrama de sectores simple
> muestra <- c(15, 12, 4, 16, 8)
> paises<- c("USA", "Inglaterra", "Australia", "Alemania", "Francia")
> pie(muestra, labels = paises, main="Diagrama de sectores de países")
```



Podemos mejorar la visualización de estos diagramas mediante distintas variantes:

```
> # Diagrama de sectores con porcentajes
> muestra <- c(15, 12, 4, 16, 8)
> paises <- c("USA", "Inglaterra", "Australia", "Alemania", "Francia")
> pct <- round(muestra/sum(muestra)*100)
> # Juntamos porcentajes y etiquetas
> paises <- paste(paises, pct)
> # Juntamos % y etiquetas
> paises <- paste(paises,"%",sep="")
> pie(muestra,labels = paises, col=rainbow(length(paises)),
+     main="Diagrama de sectores de países")
```



Veamos un ejemplo concreto. Calculamos un diagrama de sectores de la variable “Provincias de los alumnos de 5”.

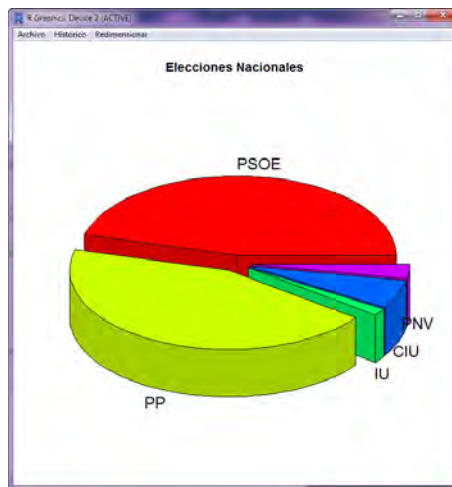
```
> # Cierro el visor de gráficas
> graphics.off()
> # Abro el visor de gráficas
> x11()
> Provincias.estudiantes<-c("GR", "CO", "MA", "GR","GR", "MA", "GR","GR", "CO")
> Provincias.estudiantes
[1] "GR" "CO" "MA" "GR" "GR" "MA" "GR" "GR" "CO"
> table(Provincias.estudiantes)
Provincias.estudiantes
CO GR MA
 2  5  2
> pie(table(Provincias.estudiantes), col=c("red","blue","green"))
> title("Provincias de nacimiento")
```



También podemos crear gráficos en 3-dimensiones, para ello tenemos que servirnos de uno de los mejores paquetes que podemos encontrar en CRAN a la hora de crear gráficos, el paquete “plotrix” creado por Jim Lemon y cols.

```
> # Diagrama de sectores 3D
> # Necesitamos el paquete plotrix
```

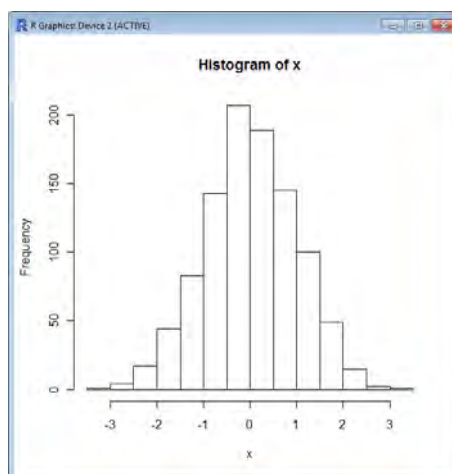
```
> # Si no lo tenemos lo podemos cargar con la orden
> # install.packages("plotrix")
> library(plotrix)
> escaños <- c(125, 120, 4, 16, 8)
> partidos <- c("PSOE", "PP", "IU", "CIU", "PNV")
> pie3D(escaños, labels=partidos, explode=0.1, main="Elecciones Nacionales")
```



## hist

Los histogramas para variables continuas lo representamos con la función “hist”

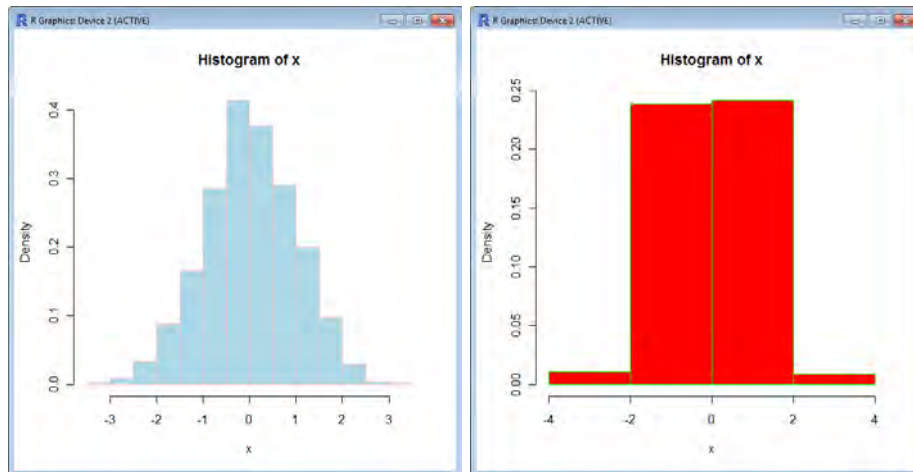
```
> # Representamos un histograma de la variable x, con x 1000 datos normales
> x<-rnorm(1000)
> hist(x)
```



Como vemos crea un histograma con los datos y como estos datos son normales el histograma se asemeja a la curva normal. Pero R asigna el número de intervalos y la amplitud de ellos. Si queremos modificarlos utilizamos los argumentos “break” que indican los puntos de corte junto con otros argumentos para darles efectos visuales.

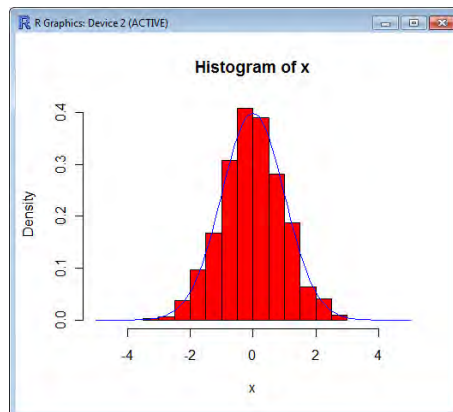


```
># creamos 13 intervalos (12 puntos de corte), de color azul y lineas rosas
> hist(x, freq = FALSE, breaks = 12, col="lightblue", border="pink")
># creamos 4 intervalos (3 puntos de corte), de color azul y lineas rosas
> hist(x, freq = FALSE, breaks = 3, col="red", border="green")
```



Como vimos anteriormente podemos dibujar el histograma junto con gráfica de la normal para ver si los datos se distribuyen o no normalmente.

```
> x<-rnorm(1000)
> hist(x,col="red",freq=F,xlim=c(-5,5))
> curve(dnorm(x),-5,5,add=T,col="blue")
```

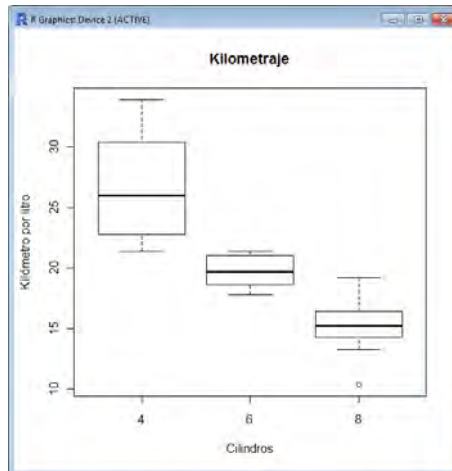


## boxplots

Con la función boxplots se pueden crear diagramas de cajas para una o varias funciones. Para nuestros ejemplos vamos a utilizar unos de los datos que R proporciona, la base de datos “mtcars”.

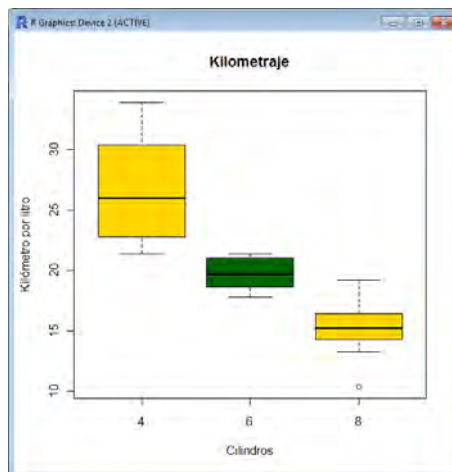
	mpg	cyl	displacement	hp	drat	wt	qsec	vs	am	gear	carb
Mazda RX4	21.0	6	160.0	110	3.90	2.620	16.46	0	1	4	4
Mazda RX4 Wag	21.0	6	160.0	110	3.90	2.875	17.02	0	1	4	4
Datsun 710	22.8	4	108.0	93	3.85	2.320	18.61	1	1	4	1
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
Volvo 142E	21.4	4	121.0	109	4.11	2.780	18.60	1	1	4	2

```
> boxplot(mpg~cyl,data=mtcars, main="Kilometraje",
+   xlab="Cilindros", ylab="Kilómetros por litro")
```



Podemos diferenciar con distintos colores la gráfica con los argumentos de boxplot.

```
> boxplot(mpg~cyl,data=mtcars, main="Kilometraje",
+   xlab="Cilindros", ylab="Kilómetro por litro", col=c("gold","darkgreen"))
```

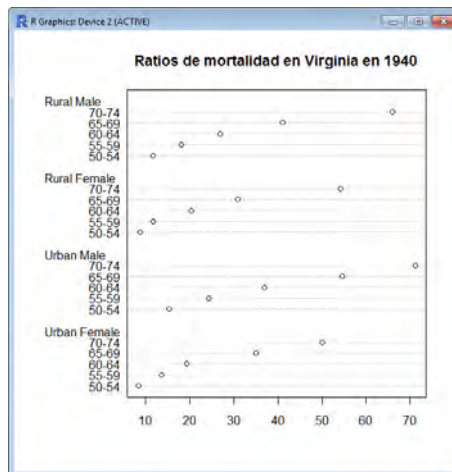


## dotchart

Con la función “dotchart” creamos diagramas de puntos por escalas (Cleveland dot plot). Utilizamos para ello los datos de R “VADeaths”.

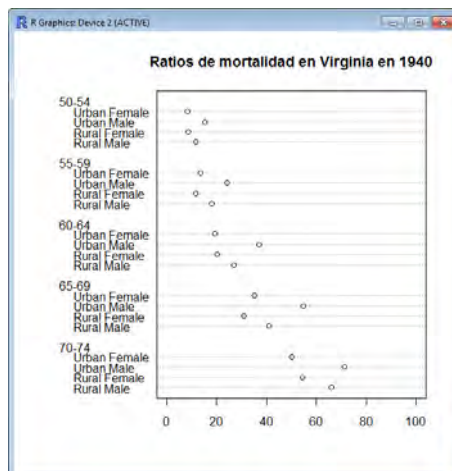
	Rural Male	Rural Female	Urban Male	Urban Female
50-54	11.7	8.7	15.4	8.4
55-59	18.1	11.7	24.3	13.6
60-64	26.9	20.3	37.0	19.3
65-69	41.0	30.9	54.6	35.1
70-74	66.0	54.3	71.1	50.0

```
> dotchart(VADeaths, main = "Ratios de mortalidad en Virginia en 1940")
```



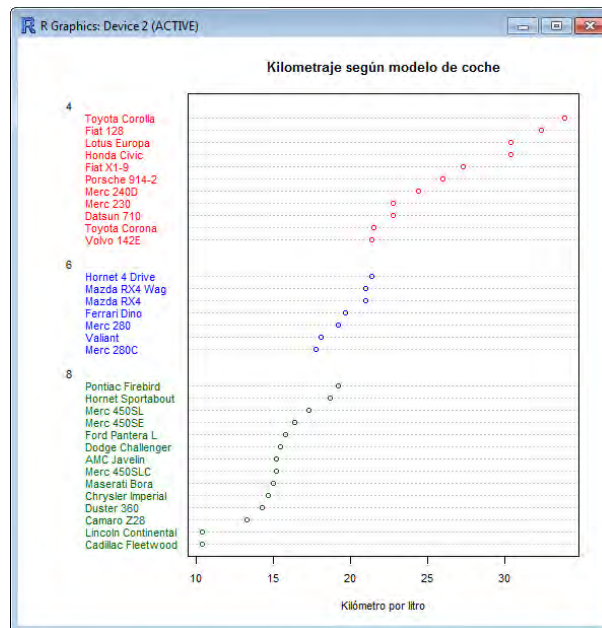
Si trasponemos los datos y acotamos la gráfica tenemos:

```
> dotchart(t(VADeaths), xlim = c(0,100), main = "Ratios de mortalidad en Virginia en 1940")
```



Una variante de este gráfico nos la proporciona los distintos argumentos de dotchart. Para ello utilizamos de nuevo los datos mtcars.

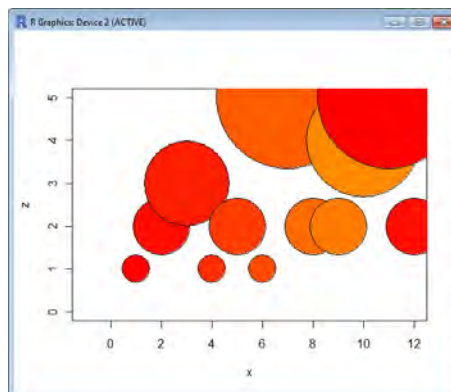
```
> # Ordenamos los datos por la variable mpg
> x <- mtcars[order(mtcars$mpg),]
> # Vemos los factores de la variable cyl
> x$cyl <- factor(x$cyl)
> x$cyl
[1] 8 8 8 8 8 8 8 8 8 8 8 6 6 8 6 6 6 6 6 4 4 4 4 4 4 4 4 4 4 4
Levels: 4 6 8
> # Asignamos colores a los factores
> x$color[x$cyl==4] <- "red"
> x$color[x$cyl==6] <- "blue"
> x$color[x$cyl==8] <- "darkgreen"
> #dibujamos los puntos
> dotchart(x$mpg, labels=row.names(x), cex=.7, groups= x$cyl,
+   main="Kilometraje según modelo de coche",
+   xlab="Kilómetro por litro", gcolor="black", color=x$color)
```



## Función symbols

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

```
> x=1:12
> z<-c(1,2,3,1,2,1,5,2,2,4,5,2)
> symbols(x,z,circles=z,xlim=c(-1,12),ylim=c(0, 5),bg=1:10)
```



## par

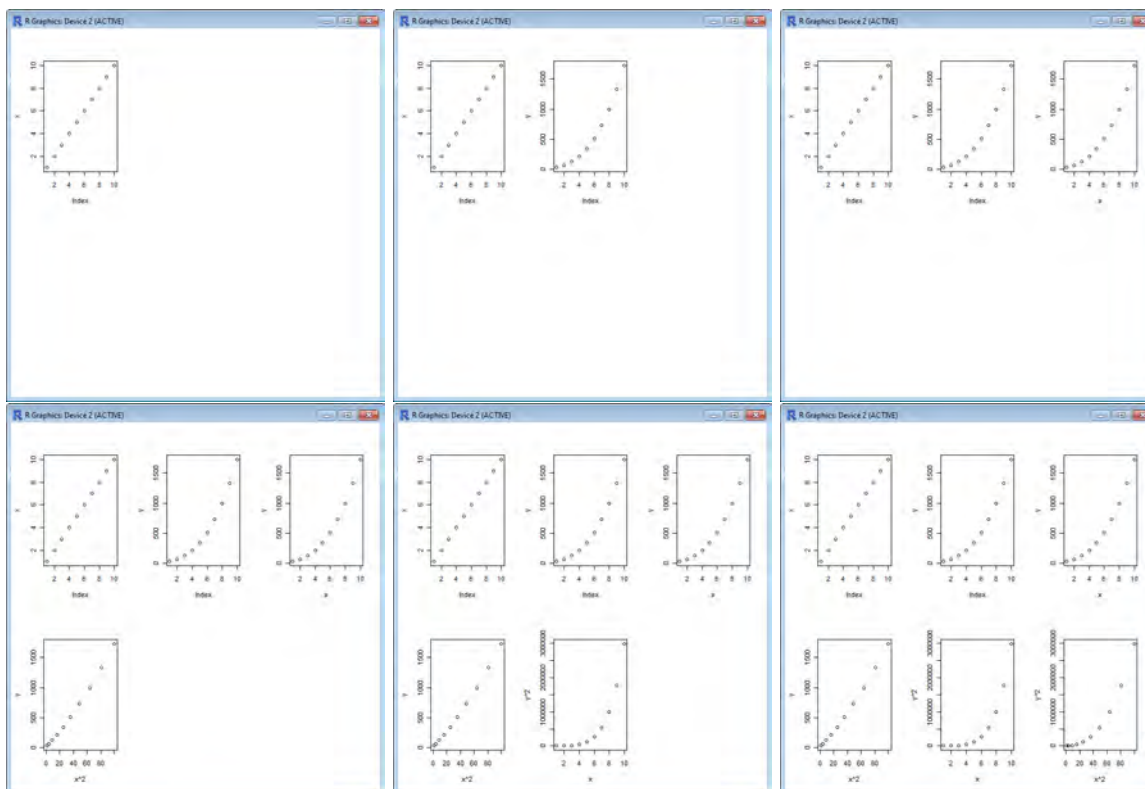
Entre los argumentos de la función “par” hay algunos que permiten presentar gráficos múltiples. Así, el argumento “mfcol=c(m,n)”, divide el dispositivo gráfico en  $m \times n$  partes iguales, que se rellenan por columnas, análogamente “mfrow=c(m,n)” rellena por filas.

```
> x=1:10
> y=(3:12)^3
> par(mfrow=c(2,3))
> plot(x)
```

```

> plot(y)
> plot(x,y)
> plot(x^2,y)
> plot(x,y^2)
> plot(x^2,y^2)

```



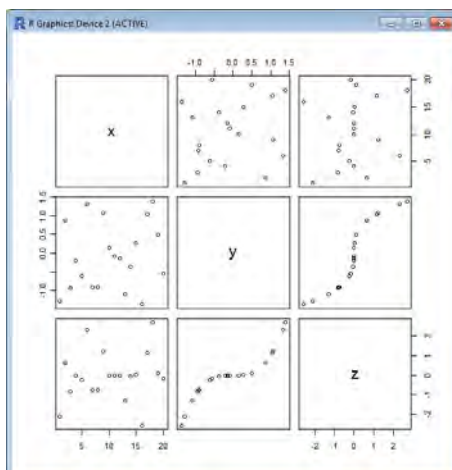
## pairs

Esta función permite crear una matriz de diagramas de puntos entre más de dos variables.

```

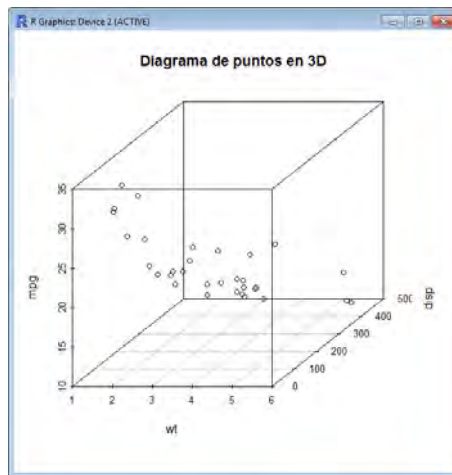
> x<-1:20
> y<-rnorm(20)
> z<-y^3
> pairs(~x+y+z)

```



También podemos hacer diagramas de puntos en 3D con la función “scatterplot3d” del paquete “scatterplot3d”.

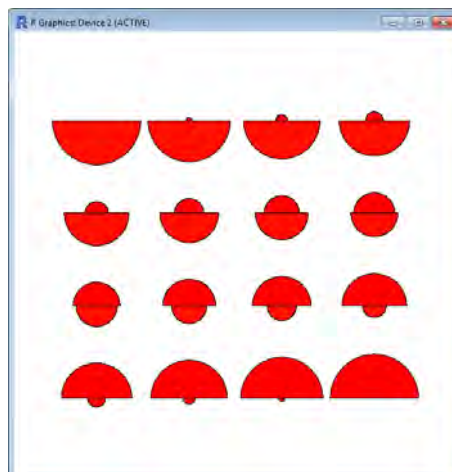
```
> library(scatterplot3d)
> # Tomamos los datos mtcars
> attach(mtcars)
> scatterplot3d(wt,disp,mpg, main="Diagrama de puntos en 3D ")
```



## stars

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

```
> stars(cbind(1:16,2*(16:1)),draw.segments=TRUE)
```



## Fractal con R

Como vamos a mostrar R proporciona el entorno gráfico suficiente par desarrollar simulaciones de valores para construir fractales. Tomemos un ejemplo para crear uno:

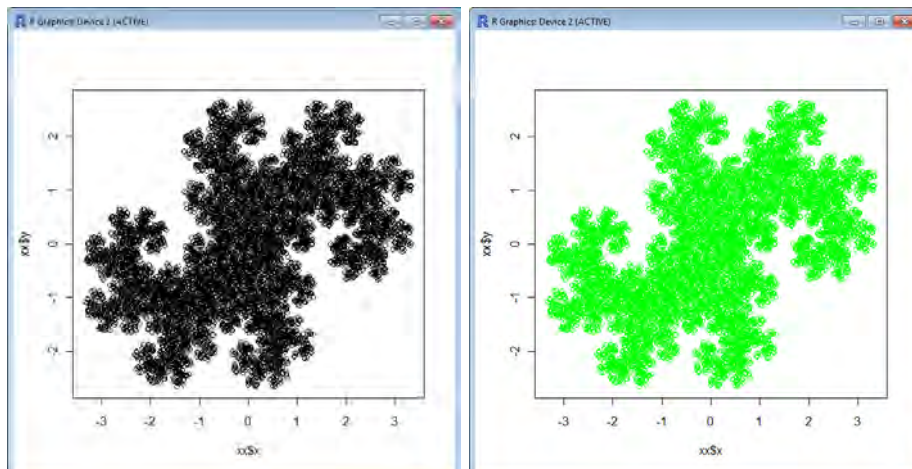
```
> # Creamos una función para dibujar nuestro ejemplo
> f1=function(numero = 100)
{
```

```

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

> f1=function(numero = 100)
+ {
+ x = vector(mode = "numeric", length = numero)
+ y = vector(mode = "numeric", length = numero)
+ x[1] = 1
+ y[1] = 1
+ for(i in 2:numero)
+ {
+ if(sample(2,1) == 2)
+ {m = 1}
+ else
+ {m = -1}
+ x[i] = 0.5 * x[i - 1] + 0.5 * y[i - 1] + m
+ y[i] = -0.5 * x[i - 1] + 0.5 * y[i - 1] + m
+ }
+ return(list(x = x[2:numero], y = y[2:numero]))
+ }
> xx<-f1(10000)
> plot(xx)
> plot(xx, col="green")

```

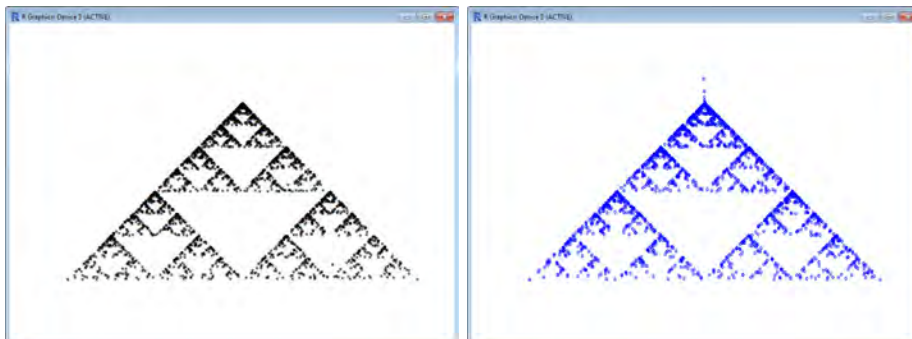


Otro fractal es el de Sierpinski creado por Caballero.

```

> f2<- function(puntos=3000, tipo="*", color="black"){
+   x <- 0
+   y <- 0
+   x[1] <- abs(rnorm(1))
+   if(x[1] < 0.5)
+     y[1] <- abs(rnorm(1)*sqrt(3)*x[1])
+   else
+     y[1] <- abs(rnorm(1)*(-sqrt(3)*x[1]+sqrt(3)))
+   plot.new()
+   # Representa tres puntos -> Define el triángulo.
+   points(c(0,1,0.5,0),c(0,0,sqrt(3)/2,0), pch=tipo, col=color)
+   # Punto al azar.
+   points(x[1],y[1], pch=tipo, col=color)
+   # Representa los restantes puntos dependiendo del tamaño especificado.
+   for(i in 2:puntos)
+   {
+     aux <- abs(3*rnorm(1))
+     if (aux <= 1)
+     {
+       x[i] <- x[i-1]/2
+       y[i] <- y[i-1]/2
+     }
+     else
+     {
+       if (aux <= 2)
+       {
+         x[i] <- (1+x[i-1])/2
+         y[i] <- y[i-1]/2
+       }
+       else
+       {
+         x[i] <- (0.5+x[i-1])/2
+         y[i] <- ((sqrt(3)/2)+y[i-1])/2
+       }
+     }
+   }
+   points(x,y,pch=tipo, col=color)
+ }
> f2()
> f2(puntos=2000, tipo="o", color="blue")

```





**Parte VI**

**Estadística Descriptiva**

# Estadística Descriptiva

La estadística descriptiva es una parte importante de la Estadística que se dedica a analizar y representar los datos. Su finalidad es obtener información, analizarla, elaborarla y simplificarla lo necesario para que pueda ser interpretada cómoda y rápidamente y, por tanto, pueda utilizarse eficazmente para el fin que se desee.

Veremos como calcular tablas de frecuencias (absolutas, relativas y acumuladas), medidas de tendencia central, de posición, dispersión, de asimetría, etc.

Empecemos con una muestra de resultados del lanzamiento de un dado 25 veces, los resultados obtenidos son: {1,2,5,3,6,4,2,1,2,4,1,5,3,2,4,1,6,2,3,1,6,2,4,2,1}.

Utilizamos la función “table” para calcular la frecuencia absoluta de cada valor de la variable. Para el calculo de la frecuencia relativa dividimos table entre la longitud del conjunto de datos (“length”). Para calcular el total de una variable utilizamos “addmargins” y para calcular la frecuencia acumulada utilizamos “cumsum”.

```
> dados<-c(1,2,5,3,6,4,2,1,2,4,1,5,3,2,4,1,6,2,3,1,6,2,4,2,1)
> table(x)
x
1 2 3 4
7 7 3 3
> dados<-c(1,2,5,3,6,4,2,1,2,4,1,5,3,2,4,1,6,2,3,1,6,2,4,2,1)
> table(dados)
dados
1 2 3 4 5 6
6 7 3 4 2 3
> # Calculamos la frecuencia absoluta de cada valor de la variable
> table(dados)/length(dados)
dados
  1    2    3    4    5    6
0.24 0.28 0.12 0.16 0.08 0.12
> # 0 con:
> table(dados)/sum(table(dados))
dados
  1    2    3    4    5    6
0.24 0.28 0.12 0.16 0.08 0.12
> # Calculamos la frecuencia relativa de cada valor de la variable
> addmargins(table(dados))
dados
  1    2    3    4    5    6 Sum
6    7    3    4    2    3 25
> # Calculamos la frecuencia absoluta y el total de la variable
> addmargins(table(dados)/length(dados))
dados
  1    2    3    4    5    6 Sum
0.24 0.28 0.12 0.16 0.08 0.12 1.00
```

```
> # Calculamos la frecuencia relativa y el total de la variable
> cumsum(table(dados))
 1  2  3  4  5  6
6 13 16 20 22 25
> # Calculamos la frecuencia absoluta acumulada
> cumsum(table(dados)/length(dados))
 1  2  3  4  5  6
0.24 0.52 0.64 0.80 0.88 1.00
```

Vamos a introducir otros datos para calcular medidas de dispersión con ellos. Nos serviremos de la función “scan” que permite introducir uno a uno los valores que necesitemos (como en una calculadora). R termina de introducir los valores en cuando pulsemos dos veces seguidas la tecla “Enter”.

```
> x<-scan()
1: 12
2: 13
3: 12
4: 13
5: 12
6: 15
7: 14
8: 12
9: 13
10: 15
11:
Read 10 items
> x
[1] 12 13 12 13 12 15 14 12 13 15
```

Con la función “summary” nos muestra el resumen estadístico de las variables que les indiquemos. Para cada variable nos muestra el valor mínimo de esta variable, el primer y tercer cuartil, la mediana, la media y el máximo:

```
> x
[1] 12 13 12 13 12 15 14 12 13 15
> summary(x)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 12.00   12.00   13.00   13.10   13.75   15.00
```

Otra función interesante es “fivenum” que calcula un resumen de los 5 elementos de Tukey (el mínimo, los cuartiles y el máximo) para los datos de entrada que suelen dar una buena idea del comportamiento de la distribución.

```
> x=c(1:5,5:1,4,2,3,2)
> summary(x)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 1.000   2.000   3.000   2.929   4.000   5.000
> fivenum(x)
[1] 1 2 3 4 5
> # Devuelve el mínimo, 1º cuartil, mediana, 3º cuartil y máximo
```

otra función que calcula un resumen estadístico es “stat.desc” del paquete “pastecs”, computa un conjunto de medidas de tendencia central y de dispersión.

```
> edad <- c(22,22,23,24,26,27,28,29,29,29,
+ 31,33,34,35,35,35,36,38,39,42,44,44,45,45,
+ 45,47,48,52,59,66,67,69,69)
> stat.desc(edad)
```

nbr.val	nbr.null	nbr.na	min	max	range	sum
33.0000000	0.0000000	0.0000000	22.0000000	69.0000000	47.0000000	1317.0000000
median	mean	SE.mean	CI.mean.0.95	var	std.dev	coef.var
36.0000000	39.9090909	2.4118409	4.9127592	191.9602273	13.8549712	0.3471633

```
>
```

## Medidas de tendencia central

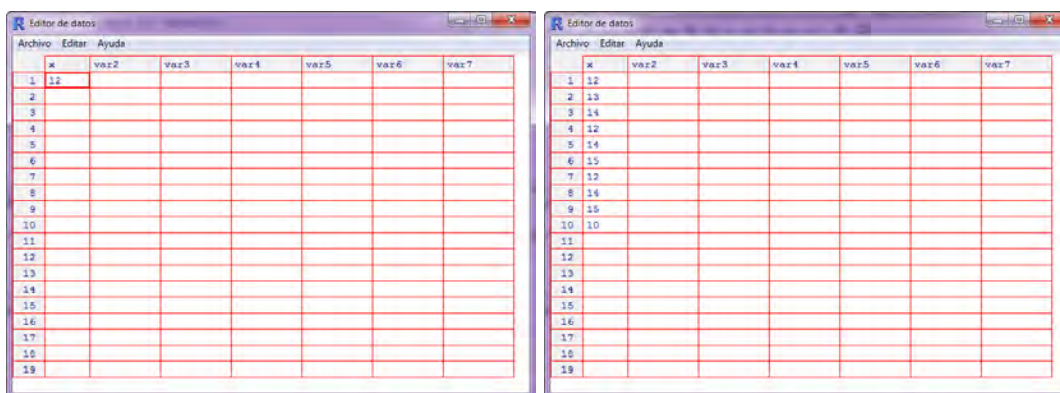
### Media aritmética

Dado un conjunto de  $n$  observaciones  $x_1, x_2, \dots, x_n$  se define la media muestral de las observaciones como:

$$\bar{x} = \frac{\sum(x)}{n}$$

Como vimos anteriormente podemos calcular la media de un conjunto de datos creando nuestra propia función o de una forma más sencilla a partir de la función “mean”.

```
> # En este ejemplo vamos a introducir los datos con la función data.entry
> # Creamos un vector vacío
> x<-c()
> # Introducimos los datos
> data.entry(x)
Error en de.ncols(sdata) : wrong argument to 'dataentry'
> # He de asignarle al vector por lo menos un valor, le pongo el primero
> x<-c(12)
> data.entry(x)
```



```
> x
[1] 12 13 14 12 14 15 12 14 15 10
> mean(x)
[1] 13.1
```

### Media geométrica

La media geométrica es otra forma de describir el valor central de un conjunto de datos, se define como:

$$\log \bar{x} = \frac{1}{n} \sum_{i=1}^n \log x_i$$

```
> x<-c(1:5,4:1,3)
> x
[1] 1 2 3 4 5 4 3 2 1 3
> # Calculamos la media geométrica calculando la media del logaritmo
> mean(log(x))
[1] 0.9064158
```

La media geométrica calculada con la ecuación anterior está expresada en una escala logarítmica, para volver a la escala original, utilizamos la función exponencial antilogaritmo.

```
> exp(mean(log(x)))
[1] 2.475434
```

## Moda

En estadística, la moda es el valor con una mayor frecuencia en una distribución de datos.

$$Moda = \max\{f_i : i \in \{1, \dots, n\}\}$$

Hablaremos de una distribución bimodal de los datos cuando encontremos dos modas, es decir, dos datos que tengan la misma frecuencia absoluta máxima. Si todas las variables tienen la misma frecuencia diremos que no hay moda.

El intervalo modal es el de mayor frecuencia absoluta. Cuando tratamos con datos agrupados antes de definir la moda, se ha de definir el intervalo modal. La moda, cuando los datos están agrupados, es un punto que divide al intervalo modal en dos partes de la forma  $p$  y  $c-p$ , siendo  $c$  la amplitud del intervalo, que verifiquen que:

$$\frac{p}{c-p} = \frac{n_i - n_{i-1}}{n_i - n_{i+1}}$$

Siendo la frecuencia absoluta del intervalo modal las frecuencias absolutas de los intervalos anterior y posterior, respectivamente, al intervalo modal.

R no proporciona una función que calcule la moda, pero nos la arreglaremos para calcularlos

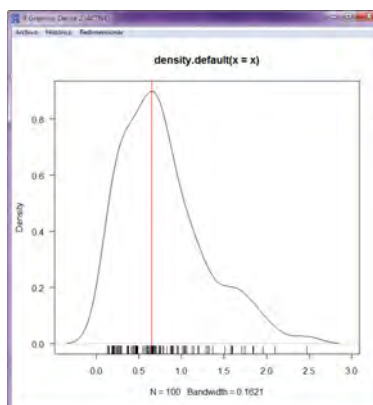
```
> edad <- c(22,22,23,24,26,27,28,29,29,29,
+ 31,33,34,35,35,35,36,38,39,42,44,44,45,45,
+ 45,47,48,52,59,66,67,69,69)
> table(edad)
edad
22 23 24 26 27 28 29 31 33 34 35 36 38 39 42 44 45 47 48 52 59 66 67 69
 2  1  1  1  1  1  3  1  1  1  3  1  1  1  1  2  3  1  1  1  1  1  1  2
> max(table(edad))
[1] 3
> # Por lo tanto la moda es 29,35 y 45
```

Hay paquetes que la incorporan, como puede ser “prettyR” que con la función “Mode” calcula la moda de una variable (pero solo en el caso de que sea unimodal)

```
> library(prettyR)
> Mode(edad)
[1] ">1 mode"
> x<-c(1, 2, 3, 4, 5, 4, 3, 2, 1, 3)
> x
[1] 1 2 3 4 5 4 3 2 1 3
> Mode(x)
[1] "3"
```

Podemos distinguir también si los datos son continuos o discretos

```
> # x discreta
> modad <- function(x) as.numeric(names(which.max(table(x))))
> # x es continua
> modac <- function(x){
+     dd <- density(x)
+     dd$x[which.max(dd$y)]
+ }
> # Ejemplo
> # x es discreta
> x <- rpois(100, 10)
> modad(x)
[1] 10
> table(x)
x
 1  4  5  6  7  8  9 10 11 12 13 14 15 18
 1  4  8  4 10  7 13 15 10  9  9  4  5  1
> # x es continua
> x <- rgamma(100, 3, 4)
> modac(x)
[1] 0.6471526
> dd <- density(x)
> plot(dd, type = 'l', las = 1)
> rug(x)
> abline(v = dd$x[which.max(dd$y)], col = 2)
```



## Mediana

La mediana es el valor de la variable que deja el mismo número de datos antes y después que él, una vez ordenados estos. De acuerdo con esta definición el conjunto de datos menores o iguales que la mediana representarán el 50 % de los datos, y los que sean mayores que la mediana representarán el otro 50 % del total de datos de la muestra. La mediana coincide con el percentil 50, con el segundo cuartil y con el quinto decil.

```
> edad <- c(22,22,23,24,26,27,28,29,29,29,
+ 31,33,34,35,35,35,36,38,39,42,44,44,45,45,
+ 45,47,48,52,59,66,67,69,69)
> edad
[1] 22 22 23 24 26 27 28 29 29 29 31 33 34 35 35 35
[16] 36 38 39 42 44 44 45 45 45 47 48 52 59 66 67 69 69
> median(edad)
[1] 36
> # Vemos que también coincide con summary
> summary(edad)
  Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 22.00  29.00   36.00   39.91   45.00   69.00
```

## Cuantiles

Los cuantiles son valores de la distribución que la dividen en partes iguales, es decir, en intervalos, que comprenden el mismo número de valores. Los más usados son los cuartiles, los deciles y los percentiles.

Los percentiles son 99 valores que dividen en cien partes iguales el conjunto de datos ordenados. Ejemplo, el percentil de orden 15 deja por debajo al 15 % de las observaciones, y por encima queda el 85 %

Los cuartiles son los tres valores que dividen al conjunto de datos ordenados en cuatro partes iguales, son un caso particular de los percentiles:

- El primer cuartil  $Q_1$  es el menor valor que es mayor que una cuarta parte de los datos
- El segundo cuartil  $Q_2$  (la mediana), es el menor valor que es mayor que la mitad de los datos
- El tercer cuartil  $Q_3$  es el menor valor que es mayor que tres cuartas partes de los datos

Los deciles son los nueve valores que dividen al conjunto de datos ordenados en diez partes iguales, son también un caso particular de los percentiles.

```
> # Calculamo el 1° y 3° cuartil
> # el cuartil uno deja el 25% a la izquierda
> quantile(edad, probs=.25)
25%
26.5
> # el cuartil tres deja el 75% a la izquierda
> quantile(edad, probs=.75)
75%
35
> # el decil tres deja el 30% a la izquierda
> quantile(edad, probs=.3)
30%
27.4
> # el decil ocho deja el 80% a la izquierda
> quantile(edad, probs=.8)
80%
35
> # el percentil ochenta y uno deja el 81% a la izquierda
> quantile(edad, probs=.81)
81%
35
```

## Medidas de dispersión

### Rango

Se denomina rango estadístico o recorrido estadístico al intervalo de menor tamaño que contiene a los datos; es calculable mediante la resta del valor mínimo al valor máximo; por ello, comparte unidades con los datos. Permitiendo obtener así una idea de la dispersión de los datos.

$$Rango = \text{máx}(x) - \text{mín}(x)$$

```
> edad <- c(22,22,23,24,26,27,28,29,29,29, 31,33,34,35,35,35,36,38,39)
> range(edad)
[1] 22 39
> # Veamos que coinciden con los resultados de summary
> summary(edad)
  Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 22.00  26.50   29.00   30.26   35.00   39.00
> # Lo creamos nosotros
> min(edad)
[1] 22
> max(edad)
[1] 39
> rango<-c(min(edad), max(edad))
> rango
[1] 22 39
```

### Rango Medio

El rango medio de un conjunto de valores numéricos es la media de los dos valores extremos (menor y mayor valor), o la mitad del camino entre el dato mínimo y el dato Máximo.

$$Rango\ Medio = \frac{\text{máx}(x) + \text{mín}(x)}{2}$$

```
> edad <- c(22,22,23,24,26,27,28,29,29,29, 31,33,34,35,35,35,36,38,39)
> range(edad)
[1] 22 39
> range(edad)->rg
> mean(rg)
[1] 30.5
```

### Rango intercuartílico

Al igual que ocurre con otras medidas ésta no está explícitamente implementada. Pero podemos programarla a partir de otras que si lo están.

```
> quantile(edad,.75)-quantile(edad,.25)
75%
8.5
```



## Varianza

Para conocer con detalle un conjunto de datos, no basta con conocer las medidas de tendencia central, sino que necesitamos conocer también la desviación que representan los datos en su distribución respecto de la media aritmética de dicha distribución, con objeto de tener una visión de los mismos más acorde con la realidad a la hora de describirlos e interpretarlos para la toma de decisiones.

La varianza de una variable aleatoria es una medida de dispersión definida como la esperanza del cuadrado de la desviación de dicha variable respecto a su media.

$$Varianza = \frac{\sum (x - \bar{x})^2}{n}$$

R no calcula como vimos anteriormente la varianza poblacional sino que calcula la muestral, por tanto el divisor es  $n - 1$  en vez de  $n$ .

```
> edad <- c(22,22,23,24,26,27,28,29,29,29, 31,33,34,35,35,35,36,38,39)
> var(edad)
[1] 29.20468
> sum((edad-mean(edad))^2)/(length(edad))
[1] 27.66759
> sum((edad-mean(edad))^2)/(length(edad)-1)
[1] 29.20468
> #Si lo que queremos es calcular la poblacional multiplicamos por n-1/n
> n<-length(edad)
> var(edad)*(n-1)/n
[1] 27.66759
```

## Desviación Típica

La desviación estándar o desviación típica es una medida de dispersión para variables de gran utilidad en la estadística descriptiva.

Se define como la raíz cuadrada de la varianza. Junto con este valor, la desviación típica es una medida (cuadrática) que informa de la media de distancias que tienen los datos respecto de su media aritmética, expresada en las mismas unidades que la variable.

$$DT = \sqrt{\frac{\sum (x - \bar{x})^2}{n}}$$

R no calcula como vimos anteriormente la varianza poblacional sino que calcula la muestral, por tanto el divisor es  $n - 1$  en vez de  $n$ . Para calcularla utilizaremos la función “sd”.

```
> # Muestral
> sd(edad)
[1] 5.404135
> sqrt(var(edad))
[1] 5.404135
> # Poblacional
> sd(edad)*(n-1)/n
[1] 5.119707
> sqrt(var(edad)*(n-1)/n)
[1] 5.119707
```

## Coeficiente de variación de Pearson

Cuando se quiere comparar el grado de dispersión de dos distribuciones que no vienen dadas en las mismas unidades o que las medias no son iguales se utiliza el coeficiente de variación de Pearson que se define como el cociente entre la desviación típica y el valor absoluto de la media aritmética.

$$CV = \frac{DT}{\bar{x}}$$

```
> edad <- c(22,22,23,24,26,27,28,29,29,29, 31,33,34,35,35,35,36,38,39)
> sqrt(var(edad))
[1] 5.404135
> mean(edad)
[1] 30.26316
> abs(mean(edad))
[1] 30.26316
> CV<-(sqrt(var(edad)))/(abs(mean(edad)))
> CV
[1] 0.1785714
```

Como es un poco fastidioso hacer esto cada vez que queremos calcular el coeficiente de variación vamos a crear una función para calcularlo.

```
> CV<-function(x)
+ {
+   cv<-(sqrt(var(x)))/(abs(mean(x)))
+   return(cv)
+ }
> CV(1:4)
[1] 0.5163978
> # Comprobamos si da lo mismo que antes para edad
> CV(edad)
[1] 0.1785714
```

## Medidas de forma

las medidas de forma comparan la forma que tiene la representación gráfica, bien sea el histograma o el diagrama de barras de la distribución, con la distribución normal.

### Medida de la asimetría

Diremos que una distribución es simétrica cuando su mediana, su moda y su media aritmética coinciden. Por lo tanto, diremos que una distribución es asimétrica a la derecha si las frecuencias (absolutas o relativas) descienden más lentamente por la derecha que por la izquierda y si las frecuencias descienden más lentamente por la izquierda que por la derecha diremos que la distribución es asimétrica a la izquierda.

Existen varias medidas de la asimetría de una distribución de frecuencias. Una de ellas es el Coeficiente de Asimetría de Pearson:

$$CAP = \frac{\bar{x} - M_o}{DT}$$

El coeficiente de asimetría sólo viene definido como función en algunos paquetes, pero vamos a crear una función que lo calcule a partir de la definición:

$$CAP = \frac{\sum_{i=1}^n (x_i - \bar{x})^3}{\frac{n}{DT_{n-1}^3}}$$

```
> asim<-function(x)
+ {
+ as<-sum((x-mean(x))^3/length(x))/(sd(x))^3
+ return(as)
+ }
> asim(edad)
[1] -0.0588123
```

Como hemos dicho anteriormente existen paquetes como “fBasics” que tienen implementadas funciones de asimetría como “skewness” que calcula el coeficiente de asimetría de Pearson.

```
> library(fBasics)
> skewness(edad)
[1] -0.0588123
attr(,"method")
[1] "moment"
> x <- runif(30)
> skewness(x)
[1] -0.02627598
attr(,"method")
[1] "moment"
```

## Medida de la curtosis

Las medidas de curtosis miden la mayor o menor cantidad de datos que se agrupan en torno a la moda. Se definen 3 tipos de distribuciones según su grado de curtosis:

- Distribución mesocúrtica: presenta un grado de concentración medio alrededor de los valores centrales de la variable (el mismo que presenta una distribución normal).
- Distribución leptocúrtica: presenta un elevado grado de concentración alrededor de los valores centrales de la variable.
- Distribución platicúrtica: presenta un reducido grado de concentración alrededor de los valores centrales de la variable.

Se define como:

$$\frac{M_4}{\sigma^4}$$

Utilizaremos el paquete “fBasics” que tiene implementada la función “kurtosis” que calcula el coeficiente de curtosis de un conjunto de datos.

```
> kurtosis(edad)
[1] -1.372259
attr(,"method")
[1] "excess"
> x <- runif(30)
> kurtosis(x)
[1] -1.434423
attr(,"method")
[1] "excess"
```

```
> kurtosis(x, method="moment")
[1] 1.565577
attr(,"method")
[1] "moment"
# Coincide con
> y = x - mean(x)
> mean(y^4)/var(x)^2
[1] 1.565577
```

## Parte VII

# Distribuciones de probabilidad

# Distribuciones de probabilidad

Cuando una variable aleatoria toma diversos valores, la probabilidad asociada a cada uno de tales valores puede ser organizada como una distribución de probabilidad, la cual es la distribución de las probabilidades asociadas a cada uno de los valores de la variable aleatoria.

Las distribuciones de probabilidad pueden representarse a través de una tabla, una gráfica o una fórmula, en cuyo caso tal regla de correspondencia se le denomina función de probabilidad.

Las distribuciones de probabilidad para variables pueden ser discretas, cuando la variable aleatoria tomaba un valor en concreto o continuas cuando los valores no son concretos y pueden ser cualquier valor de un intervalo.

R tiene definidas algunas de las más importantes funciones de distribución, en la siguiente tabla indicamos la función y la distribución que representan (discreta o continuas).

Función	Distribución
beta	beta
binom	binomial
cauchy	Cauchy
exp	exponencial
chisq	chi-cuadrado
fisher	F
gamma	gamma
geom	geométrica
hyper	hipergeométrica
lnorm	lognormal
logis	logística
nbinom	binomial negativa
norm	normal
pois	Poisson
t	t-Student
unif	uniforme
weibull	Weibull
wilcox	Wilcoxon

A cada nombre de función dado por R se le agrega un prefijo ‘d’ para obtener la **función de densidad**, ‘p’ para la **función de distribución acumulada**, ‘q’ para la **función cuantil o percentil** y ‘r’ para **generar variables pseudo-aleatorias** (random). La sintaxis es la siguiente:

```
> dxxx(x, ...)  
> pxxx(q, ...)  
> qxxx(p, ...)  
> rxxx(n, ...)
```

Donde xxx indica el nombre de cualquiera de las distribuciones, x y q son vectores que toman valores en el soporte de la distribución, p es un vector de probabilidades y n es un valor entero.

## Distribuciones de probabilidad para variables discretas

### Distribución de Bernoulli

La distribución de Bernoulli (o distribución dicotómica) es una distribución de probabilidad discreta, que toma valor 1 para la probabilidad de éxito ( $p$ ) y valor 0 para la probabilidad de fracaso ( $q = 1 - p$ ).

Si  $X$  es una variable aleatoria que mide "número de éxitos", y se realiza un único experimento con dos posibles resultados (éxito o fracaso), se dice que la variable aleatoria  $X$  se distribuye como una Bernoulli de parámetro  $p$ .

$$X \sim Be(p)$$

$$f(x) = p^x(1-p)^{1-x}; \text{ con } x = \{0, 1\}$$

Utilizaremos el paquete "Rlab" y cuatro funciones basados en "bern":

- `dbern(x, prob, log = FALSE)`; Devuelve resultados de la función de densidad.
- `pbern(q, prob, lower.tail = TRUE, log.p = FALSE)`; Devuelve resultados de la función de distribución acumulada.
- `qbern(p, prob, lower.tail = TRUE, log.p = FALSE)`; Devuelve resultados de los cuantiles de la binomial.
- `rbern(n, prob)`; Devuelve un vector de valores binomiales aleatorios.

Con:

- `x, q`: Vector de cuantiles.
- `p`: Vector de probabilidades.
- `n`: Número de observaciones
- `prob`: Probabilidad de éxito en cada ensayo.
- `log.p`: Parámetro booleano, si es TRUE, las probabilidades  $p$  se ofrecen como  $\log(p)$ .
- `lower.tail`: Parámetro booleano, si es TRUE (por defecto), las probabilidades son  $P[X \leq x]$ , de lo contrario,  $P[X > x]$ .

```
> # Calculamos la P(X=1) de una Bernoulli(0.7)
> dbern(1, 0.7)
[1] 0.7
> # Calculamos la P(X=0) de una Bernoulli(0.7)
> dbern(0, 0.7)
[1] 0.3
> # Calculamos la func. de distribución acumulada de una Bernoulli(0.7)
> pbern(0, 0.7)
[1] 0.3
> pbern(1, 0.7)
[1] 1
> # para la función cuantil o percentil
> qbern(0, 0.7)
[1] 0
> qbern(1, 0.7)
[1] 1
> qbern(0.3, 0.7)
[1] 0
```

```

> qbern(0.31, 0.7)
[1] 1
> qbern(0.5, 0.7)
[1] 1
> # para generar variables aleatorias
> rbern(0, 0.7)
numeric(0)
> rbern(1, 0.7)
[1] 1
> rbern(6, 0.7)
[1] 1 1 0 1 0 1

```

## Distribución Binomial

La distribución binomial es una distribución de probabilidad discreta que mide el número de éxitos en una secuencia de  $n$  ensayos independientes de Bernoulli con una probabilidad fija  $p$  de ocurrencia del éxito entre los ensayos.

Un experimento de Bernoulli se caracteriza por ser dicotómico, esto es, sólo son posibles dos resultados. A uno de estos se denomina éxito y tiene una probabilidad de ocurrencia  $p$  y al otro, fracaso, con una probabilidad  $q = 1 - p$ . En la distribución binomial el anterior experimento se repite  $n$  veces, de forma independiente, y se trata de calcular la probabilidad de un determinado número de éxitos. Para  $n = 1$ , la binomial se convierte, de hecho, en una distribución de Bernoulli.

Para representar que una variable aleatoria  $X$  sigue una distribución binomial de parámetros  $n$  y  $p$ , se escribe:

$$X \sim B(n, p)$$

$$f(x) = \binom{n}{x} p^x (1-p)^{1-x}; \text{ con } x = \{0, 1\}$$

Para obtener valores que se basen en la distribución binomial, R dispone de cuatro funciones basados en “binom”:

- `dbinom(x, size, prob, log = F)`; Devuelve resultados de la función de densidad.
- `pbinom(q, size, prob, lower.tail = T, log.p = F)`; Devuelve resultados de la función de distribución acumulada.
- `qbinom(p, size, prob, lower.tail = T, log.p = F)`; Devuelve resultados de los cuantiles de la binomial.
- `rbinom(n, size, prob)`; Devuelve un vector de valores binomiales aleatorios.

Con:

- `x, q`: Vector de cuantiles.
- `p`: Vector de probabilidades.
- `n`: Número de observaciones
- `size`: Números de ensayos(debe ser cero o más).
- `prob`: Probabilidad de éxito en cada ensayo.
- `log, log.p`: Parámetro booleano, si es TRUE, las probabilidades  $p$  se ofrecen como  $\log(p)$ .
- `lower.tail`: Parámetro booleano, si es TRUE (por defecto), las probabilidades son  $P[X \leq x]$ , de lo contrario,  $P[X > x]$ .



```

> # Calculamos la P(X=3) de una Binomial(10,0.5)
> dbinom(3, 10, 0.5)
[1] 0.1171875
> # Calculamos la P(X=6) de una Binomial(10,0.5)
> dbinom(6, 10, 0.5)
[1] 0.2050781
> # Calculamos la P(X=2,1,0) de una Binomial(10,0.5)
> dbinom(c(2, 1, 0), 10, 0.5)
[1] 0.0439453125 0.0097656250 0.0009765625
> # Calculamos la P(X<=2) de una Binomial(10,0.5)
> sum(dbinom(c(2, 1, 0), 10, 0.5))
[1] 0.0546875
> sum(dbinom(c(0, 1, 2), 10, 0.5))
[1] 0.0546875
> # Que coincide con la función pbinom
> pbinom(2,10,0.5)
[1] 0.0546875
> # Calculamos la P(X>2) de una Binomial(10,0.5)
> pbinom(2,10,0.5, lower.tail=F)
[1] 0.9453125
> pbinom(10,10,0.5)
[1] 1
> # Los valores de X que tiene una probabilidad de ocurrir en torno al 90\%
> qbinom(c(0.90), 10, 0.5)
[1] 7
> # Los valores de X que tiene una probabilidad de ocurrir en torno al 95\%
> qbinom(c(0.95), 10, 0.5)
[1] 8
> # 2 números aleatorios de una binomial
> rbinom(2,10,0.5)
[1] 6 5
> # 9 números aleatorios de una binomial
> rbinom(9,10,0.5)
[1] 8 7 5 5 7 5 5 8 5

```

## Distribución Geométrica

La distribución Geométrica es una serie de ensayos de Bernoulli independientes, con probabilidad constante  $p$  de éxito, donde la variable aleatoria  $X$  denota el número de ensayos hasta el primer éxito.

Entonces  $X$  tiene una distribución geométrica con parámetro  $p$ .

$$X \sim G(p)$$

$$f(x) = p(1-p)^{x-1}; \text{ con } x = \{0, 1\}$$

Para obtener valores que se basen en la distribución Geométrica, R dispone de cuatro funciones basados en “geom”:

- `dgeom(x, prob, log = F)`; Devuelve resultados de la función de densidad.
- `pgeom(q, prob, lower.tail = T, log.p = F)`; Devuelve resultados de la función de distribución acumulada.
- `qgeom(p, prob, lower.tail = T, log.p = F)`; Devuelve resultados de los cuantiles de la Geométrica.
- `rgeom(n, prob)`; Devuelve un vector de valores binomiales aleatorios.

Con:

- x, q: Vector de cuantiles que representa el número de fallos antes del primer éxito.
- p: Vector de probabilidades.
- n: Números de valores aleatorios a devolver.
- prob: Probabilidad de éxito en cada ensayo.
- log, log.p: Parámetro booleano, si es TRUE, las probabilidades p se ofrecen como log(p).
- lower.tail: Parámetro booleano, si es TRUE (por defecto), las probabilidades son  $P[X \leq x]$ , de lo contrario,  $P[X > x]$ .

```
> # Calculamos la P(X=4) de una G(0.5); representa el número de fallos antes del primer éxito
> dgeom(4, 0.5)
[1] 0.03125
> # Calculamos la P(X>4) de una G(0.5)
> pgeom(4, 0.5, lower.tail = F)
[1] 0.03125
> # Que en este caso coincide con dgeom(4, 0.5)
> # Lo comprobamos; P(X > 4) = 1 - P(X<=3) = 1 - [P(X=0)+P(X=1)+P(X=2)+P(X=3)+P(X=4)]
> 1 - (dgeom(0, 0.5) + dgeom(1, 0.5) + dgeom(2, 0.5) + dgeom(3, 0.5) + dgeom(4, 0.5))
[1] 0.03125
```

## Distribución Binomial Negativa

Surge de un contexto semejante al que conduce a la distribución geométrica. La distribución que asigna a la variable Y, el número de ensayo en el que ocurre el r-ésimo éxito.

Entonces X tiene una distribución Binomial Negativa.

$$X \sim BN(x, r, p)$$

Su función de probabilidad es:

$$P(y) = \binom{x-1}{r-1} p^r q^{x-r}; \text{ para } x = \{r, r+1, r+2, \dots\}$$

Para obtener valores que se basen en la distribución Binomial Negativa, R dispone de cuatro funciones basados en “nbinom”:

- dnbinom(x, size, prob, mu, log = F); Devuelve resultados de la función de densidad.
- pnbinom(q, size, prob, mu, lower.tail = T, log.p = F); Devuelve resultados de la función de distribución acumulada.
- qnbinom(p, size, prob, mu, lower.tail = T, log.p = F); Devuelve resultados de los cuantiles de la Binomial Negativa.
- rnbinom(n, size, prob, mu); Devuelve un vector de valores binomiales aleatorios.

Con:

- x, q: Vector de cuantiles (Valores enteros positivos). Corresponde a número de pruebas falladas.
- q: Vector de cuantiles.
- p: Vector de probabilidades.
- n: Números de valores aleatorios a devolver.

- prob: Probabilidad de éxito en cada ensayo.
- size: Número total de ensayos. Debe ser estrictamente positivo.
- mu: Parametrización alternativa por la media.
- log, log.p: Parámetro booleano, si es TRUE, las probabilidades p se ofrecen como  $\log(p)$ .
- lower.tail: Parámetro booleano, si es TRUE (por defecto), las probabilidades son  $P[X \leq x]$ , de lo contrario,  $P[X > x]$ .

```
> # Suponga que el 60% de los elementos no están defectuosos.
> # Para encontrar la probabilidad de localizar el quinto elemento sin defecto en el séptimo ensayo.
> # Calculamos  $P(X = 7)$ , con  $r=5$ 
> dnbinom(7-5, 5, 0.6)
[1] 0.186624
> # si queremos calcular en el séptimo ensayo o antes
> pnbinom(7-5, 5, 0.6, lower.tail = T)
[1] 0.419904
```

## Distribución De Poisson

La distribución de Poisson es una distribución de probabilidad discreta. Expresa la probabilidad de un número  $k$  de eventos ocurriendo en un tiempo fijo si estos eventos ocurren con una frecuencia media conocida y son independientes del tiempo transcurrido desde el último evento.

$$X \sim P(\lambda)$$

La función de densidad de la distribución de Poisson es:

$$f(k, \lambda) = \frac{e^{-\lambda} \lambda^k}{k!}$$

Para obtener valores que se basen en la distribución de Poisson, R dispone de cuatro funciones basados en “pois”:

- dpois(x, lambda, log = F); Devuelve resultados de la función de densidad.
- ppois(q, lambda, lower.tail = T, log.p = F); Devuelve resultados de la función de distribución acumulada.
- qpois(p, lambda, lower.tail = T, log.p = F); Devuelve resultados de los cuantiles de la Poisson.
- rpois(n, lambda); Devuelve un vector de valores binomiales aleatorios.

Con:

- x: Vector de cuantiles (Valores enteros positivos).
- q: Vector de cuantiles.
- p: Vector de probabilidades.
- n: Números de valores aleatorios a devolver.
- prob: Probabilidad de éxito en cada ensayo.
- lambda: Vector de medias (valor no negativo).
- log, log.p: Parámetro booleano, si es TRUE, las probabilidades p se ofrecen como  $\log(p)$ .
- lower.tail: Parámetro booleano, si es TRUE (por defecto), las probabilidades son  $P[X \leq x]$ , de lo contrario,  $P[X > x]$ .

```

> # Calcular la P(X = 1) de una Poisson(3)
> dpois(1, 3)
[1] 0.1493612
> dpois(c(1), 3)
[1] 0.1493612
> # Calcular la P(X <= 3) de una Poisson(3)
> ppois(3,3)
[1] 0.6472319
> # Calcular la P(X > 3) de una Poisson(3)
> ppois(3, 3, lower.tail=F)
[1] 0.3527681
> # Calcular el valor de la variable aleatoria X, dada la probabilidad 0.985
> qpois(0.985, 3)
[1] 7

```

## Distribuciones de probabilidad para variables continuas

### Distribución normal

La distribución normal, distribución de Gauss o distribución gaussiana, a una de las distribuciones de probabilidad de variable continua que con más frecuencia aparece en fenómenos reales.

La gráfica de su función de densidad tiene una forma acampanada y es simétrica respecto de un determinado parámetro. Esta curva se conoce como campana de Gauss. La importancia de esta distribución radica en que permite modelar numerosos fenómenos naturales, sociales y psicológicos. Mientras que los mecanismos que subyacen a gran parte de este tipo de fenómenos son desconocidos, por la enorme cantidad de variables incontrolables que en ellos intervienen, el uso del modelo normal puede justificarse asumiendo que cada observación se obtiene como la suma de unas pocas causas independientes.

Se dice que una variable aleatoria continua  $X$  sigue una distribución normal de parámetros  $\mu$  y  $\sigma$  y se denota  $X \sim N(\mu, \sigma)$  si su función de densidad está dada por:

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}, \quad x \in \mathbb{R}$$

Para obtener valores que se basen en la distribución normal, R dispone de cuatro funciones basados en “norm”:

- `dnorm(x, mean = 0, sd = 1, log = F)`; Devuelve resultados de la función de densidad.
- `pnorm(q, mean = 0, sd = 1, lower.tail = T, log.p = F)`; Devuelve resultados de la función de distribución acumulada.
- `qnorm(p, mean = 0, sd = 1, lower.tail = T, log.p = F)`; Devuelve resultados de los cuantiles de la Normal.
- `rnorm(n, mean = 0, sd = 1)`; Devuelve un vector de valores normales aleatorios.

Con:

- `x, q`: Vector de cuantiles.
- `p`: Vector de probabilidades.
- `n`: Números de observaciones.
- `mean`: Vector de medias. Por defecto, su valor es 0.
- `sd`: Vector de desviación estándar. Por defecto, su valor es 1.

- `log`, `log.p`: Parámetro booleano, si es `TRUE`, las probabilidades `p` son devueltas como  $\log(p)$ .
- `lower.tail`: Parámetro booleano, si es `TRUE` (por defecto), las probabilidades son  $P[X \leq x]$ , de lo contrario,  $P[X > x]$ .

```
> # Calcular la P(Z>1) de una N(0,1)
> pnorm(1, mean = 0, sd = 1, lower.tail = F)
[1] 0.1586553
> # Calcular la P(-2<Z<2) de una N(0,1)
> pnorm(c(2), mean = 0, sd = 1) - pnorm(c(-2), mean = 0, sd = 1)
[1] 0.9544997
> # Calcular la P(0<Z<1.96) de una N(0,1)
> pnorm(1.96, mean = 0, sd = 1) - pnorm(0, mean = 0, sd = 1)
[1] 0.4750021
> # Calcular la P(Z<=z)=0,5793 de una N(0,1)
> qnorm(0.5793, mean = 0, sd = 1)
[1] 0.2001030
> # Calcular la P(Z>150) de una Normal de media 125 y la desviación estándar 50.
> pnorm(150, mean = 125, sd = 50, lower.tail = F)
[1] 0.3085375
```

## Distribución t-Student

La distribución t-Student es una distribución de probabilidad que surge del problema de estimar la media de una población normalmente distribuida cuando el tamaño de la muestra es pequeño.

Aparece de manera natural al realizar la prueba t de Student para la determinación de las diferencias entre dos medias muestrales y para la construcción del intervalo de confianza para la diferencia entre las medias de dos poblaciones cuando se desconoce la desviación típica de una población y ésta debe ser estimada a partir de los datos de una muestra.

La distribución t de Student es la distribución de probabilidad del cociente:

$$\frac{Z}{\sqrt{V/\nu}}$$

donde:

1.  $Z$  tiene una distribución normal de media nula y varianza 1
2.  $V$  tiene una distribución chi-cuadrado con  $\nu$  grados de libertad
3.  $Z$  y  $V$  son independientes

La función de densidad de  $t$  es

$$f(t) = \frac{\Gamma((\nu+1)/2)}{\sqrt{\nu\pi} \Gamma(\nu/2)} (1 + t^2/\nu)^{-(\nu+1)/2}$$

Para obtener valores que se basen en la distribución t-Student, R dispone de cuatro funciones basados en “t”:

- `dt(x, df, ncp, log = F)`; Devuelve resultados de la función de densidad.
- `pt(q, df, ncp, lower.tail = T, log.p = F)`; Devuelve resultados de la función de distribución acumulada.
- `qt(p, df, ncp, lower.tail = T, log.p = F)`; Devuelve resultados de los cuantiles de la t-Student.
- `rt(n, df, ncp)`; Devuelve un vector de valores t-Student aleatorios.

Con:

- x, q: Vector de cuantiles.
- p: Vector de probabilidades.
- n: Números de observaciones.
- df: Grados de libertad.
- ncp: Parámetro que determina la centralidad de la gráfica t-Student. Si se omite, el estudio se realiza con la gráfica centralizada en 0.
- log, log.p: Parámetro booleano, si es TRUE, las probabilidades p son devueltas como log (p).
- lower.tail: Parámetro booleano, si es TRUE (por defecto), las probabilidades son  $P[X \leq x]$ , de lo contrario,  $P[X > x]$ .

```
> # Calcular P(T >= 1.3) con 7 grados de libertad.
> pt(1.3, 7, lower.tail = F)
[1] 0.1173839
> # Calcular P(T < 2.30) con 20 grados de libertad.
> pt(2.30, 20, lower.tail = T)
[1] 0.9838263
> # P(T >= t) = 0.05 con 25 grados de libertad.
> qt(0.05, 25, lower.tail = F)
[1] 1.708141
> # Calcular 5 números aleatorios con 25 grados de libertad.
> rt(5, 25)
[1] 0.6513232 0.7946921 0.5891572 0.8046254 2.4434150
```

## Distribución Chi-cuadrado

En estadística, la distribución  $\chi^2$  (de Pearson) es una distribución de probabilidad continua con un parámetro k que representa los grados de libertad de la variable aleatoria:

$$X = Z_1^2 + \cdots + Z_k^2$$

donde  $Z_i$  son variables de distribución normal, de media cero y varianza uno. El que la variable aleatoria X tenga esta distribución se representa habitualmente así:

$$X \sim \chi_k^2$$

Su función de densidad es:

$$f(x; k) = \begin{cases} \frac{1}{2^{k/2}\Gamma(k/2)} x^{(k/2)-1} e^{-x/2} & \text{para } x \geq 0, \\ 0 & \text{para } x < 0 \end{cases}$$

donde  $\Gamma$  es la función gamma:  $\Gamma(z) = \int_0^\infty t^{z-1} e^{-t} dt$ . Si n es un entero positivo, entonces  $\Gamma(n) = (n-1)!$ . Para obtener valores que se basen en la distribución chi-Cuadrada, R dispone de cuatro funciones basados en “chisq”:

- dchisq(x, df, ncp=0, log = F); Devuelve resultados de la función de densidad.
- pchisq(q, df, ncp=0, lower.tail = T, log.p = F); Devuelve resultados de la función de distribución acumulada.
- qchisq(p, df, ncp=0, lower.tail = T, log.p = F); Devuelve resultados de los cuantiles de la chi-Cuadrada.
- rchisq(n, df, ncp=0); Devuelve un vector de valores chi-Cuadrados aleatorios.

Con:

- x, q: Vector de cuantiles.
- p: Vector de probabilidades.
- n: Números de observaciones.
- df: Grados de libertad.
- ncp: Parámetro que determina la centralidad de la gráfica chi-Cuadrados. Si se omite, el estudio se realiza con la gráfica centralizada en 0.
- log, log.p: Parámetro booleano, si es TRUE, las probabilidades p son devueltas como log (p).
- lower.tail: Parámetro booleano, si es TRUE (por defecto), las probabilidades son  $P[X \leq x]$ , de lo contrario,  $P[X > x]$ .

```
> # Calcular  $X^2(0.52, 7)$  con 7 grados de libertad.
> qchisq(0.52, 7, lower.tail = F)
[1] 6.17039
> #  $P(X^2 < x) = 0.80$  con 25 grados de libertad
> qchisq(0.8, 25, lower.tail = T)
[1] 30.6752
> #  $P(X^2 \geq 18.49)$  con 24 grados de libertad.
> pchisq(18.49, 24, lower.tail = F)
[1] 0.7786126
> # Calcular 5 números aleatorios de una dist. Chi-cuadrado con 24 grados de libertad.
> rchisq(5, 24)
[1] 18.84684 20.79490 12.11334 19.63524 32.99550
```

## Distribución F de Snedecor

La distribución F de Snedecor es una distribución de probabilidad continua. Una variable aleatoria de distribución F se construye como el siguiente cociente:

$$F = \frac{U_1/d_1}{U_2/d_2}$$

donde

- $U_1$  y  $U_2$  siguen una distribución ji-cuadrada con  $d_1$  y  $d_2$  grados de libertad respectivamente, y
- $U_1$  y  $U_2$  son estadísticamente independientes.

La función de densidad de una  $F(d_1, d_2)$  viene dada por

$$g(x) = \frac{1}{B(d_1/2, d_2/2)} \left( \frac{d_1 x}{d_1 x + d_2} \right)^{d_1/2} \left( 1 - \frac{d_1 x}{d_1 x + d_2} \right)^{d_2/2} x^{-1}$$

Para obtener valores que se basen en la distribución F de Snedecor, R dispone de cuatro funciones basados en “f”:

- `df(x, df1, df2, ncp, log = F)`; Devuelve resultados de la función de densidad.
- `pf(q, df1, df2, ncp, lower.tail = T, log.p = F)`; Devuelve resultados de la función de distribución acumulada.
- `qf(p, df1, df2, ncp, lower.tail = T, log.p = F)`; Devuelve resultados de los cuantiles de la distribución F.
- `rf(n, df1, df2, ncp)`; Devuelve un vector de valores de la distribución F aleatorios.

Con:

- x, q: Vector de cuantiles.
- p: Vector de probabilidades.
- n: Números de observaciones.
- df1, df2: Grados de libertad, df1 corresponde al numerador y df2 al denominador.
- ncp: Parámetro que determina la centralidad de la gráfica de la distribución F. Si se omite, el estudio se realiza con la gráfica no centralizada.
- log, log.p: Parámetro booleano, si es TRUE, las probabilidades p son devueltas como log (p).
- lower.tail: Parámetro booleano, si es TRUE (por defecto), las probabilidades son  $P[X \leq x]$ , de lo contrario,  $P[X > x]$ .

```
> # Calcular F(0.15, 3, 2). con 3 y 2 grados de libertad.
> qf(0.15, 3, 2, lower.tail=F)
[1] 5.825814
> # P(F < f) = 0.025 con df1 = 20 y df2 = Infinito.
> qf(0.025, 20, Inf, lower.tail=T)
[1] 0.4795389
> # P(F >= 198.50) con df1 = 10 y df2 = 2.
> pf(198.50, 10, 2, lower.tail=F)
[1] 0.005022592
> # Calcular 5 números aleatorios de una dist. F de Snedecor con 24 y 10 grados de libertad.
> rf(5,24,10)
[1] 5.1400002 0.5186619 0.5719656 0.9080902 0.8035864
```

## Distribución beta

En estadística la distribución beta es una distribución de probabilidad continua con dos parámetros a y b cuya función de densidad para valores  $0 < x < 1$  es

$$f(x) = \frac{\Gamma(a+b)}{\Gamma(a)\Gamma(b)} x^{a-1} (1-x)^{b-1}$$

Donde  $\Gamma$  es la función gamma:  $\Gamma(z) = \int_0^\infty t^{z-1} e^{-t} dt$ . Si n es un entero positivo, entonces  $\Gamma(n) = (n-1)!$ . Para obtener valores que se basen en la distribución Beta, R dispone de cuatro funciones basados en “beta”:

- dbeta(x, shape1, shape2, ncp = 0, log = F); Devuelve resultados de la función de densidad.
- pbeta(q, shape1, shape2, ncp = 0, lower.tail = T, log.p = F); Devuelve resultados de la función de distribución acumulada.
- qbeta(p, shape1, shape2, ncp = 0, lower.tail = T, log.p = F); Devuelve resultados de los cuantiles de la distribución Beta.
- rbeta(n, shape1, shape2, ncp = 0); Devuelve un vector de valores de la distribución Beta aleatorios.

Con:

- x, q: Vector de cuantiles.
- p: Vector de probabilidades.
- n: Números de observaciones.
- shape1, shape2: Parámetros de la Distribución Beta. Shape1 =  $\alpha$  y Shape2 =  $\beta$ . Ambos deben ser positivos.
- ncp: Parámetro lógico que determina si la distribución es central o no.



- `log, log.p`: Parámetro booleano, si es TRUE, las probabilidades `p` son devueltas como  $\log(p)$ .
- `lower.tail`: Parámetro booleano, si es TRUE (por defecto), las probabilidades son  $P[X \leq x]$ , de lo contrario,  $P[X > x]$ .

```
> # Mediante una distribución beta con ALPHA = 5 y BETA = 4.
> # Calcule la probabilidad al menos del 75%
> pbeta(0.75, 5, 4, lower.tail = F)
[1] 0.1138153
> # Menos del 50%
> pbeta(0.5, 5, 4, lower.tail = T)
[1] 0.3632813
> # P(X < x)=0.25
> qbeta(0.25, 5, 4, lower.tail = T)
[1] 0.4445137
> # P(X > x) = 0.5
> qbeta(0.5, 5, 4, lower.tail = F)
[1] 0.5598448
> # Calcular 5 números aleatorios de una dist. Beta con ALPHA = 5 y BETA = 4
> rbeta(5,5,4)
[1] 0.7791496 0.6439238 0.6077232 0.6004583 0.3024351
```

## Distribución gamma

La distribución gamma es una distribución de probabilidad continua con dos parámetros  $k$  y  $\lambda$  cuya función de densidad para valores  $x > 0$  es

$$f(x) = \lambda^k e^{-\lambda x} \frac{x^{k-1}}{\Gamma(k)}$$

Aquí  $e$  es el número  $e$  y  $\Gamma$  es la función gamma.

Para obtener valores que se basen en la distribución Beta, R dispone de cuatro funciones basados en “beta”:

- `dgamma(x, shape, rate, scale = 1/rate, log = F)`; Devuelve resultados de la función de densidad.
- `pgamma(q, shape, rate, scale = 1/rate, lower.tail = T, log.p = F)`; Devuelve resultados de la función acumulada.
- `qgamma(p, shape, rate, scale = 1/rate, lower.tail = T, log.p = F)`; Devuelve resultados de los cuantiles de la distribución Gamma.
- `rgamma(n, shape, rate, scale = 1/rate)`; Devuelve un vector de valores de la distribución Gamma aleatorios.

Con:

- `x, q`: Vector de cuantiles.
- `p`: Vector de probabilidades.
- `n`: Números de observaciones.
- `rate`: Alternativa para especificar el valor de escala (Scale). Por defecto, su valor es igual a 1.
- `shape, scale`: Parámetros de la Distribución Gamma. Shape =  $a$  y Scale =  $s = 1/\text{rate}$ . Debe ser estrictamente positivo el parámetro Scale.
- `log, log.p`: Parámetro booleano, si es TRUE, las probabilidades `p` son devueltas como  $\log(p)$ .
- `lower.tail`: Parámetro booleano, si es TRUE (por defecto), las probabilidades son  $P[X \leq x]$ , de lo contrario,  $P[X > x]$ .

```
> # Mediante una distribución gamma con ALPHA = 3 y BETA = 0.5.  
> # Calcule la probabilidad de que sea mejor de 10  
> pgamma(10, 3, rate = 0.5, lower.tail = F)  
[1] 0.1246520  
> # Entre 4 y 8  
> pgamma(8, 3, rate = 0.5, lower.tail = T) - pgamma(4, 3, rate = 0.5, lower.tail = T)  
[1] 0.4385731  
> #  $P(X < x) = 0.7$   
> qgamma(0.7, 3, rate = 0.5, lower.tail = T)  
[1] 7.231135  
> #  $P(X > x) = 0.5$   
> qgamma(0.5, 3, rate = 0.5, lower.tail = F)  
[1] 5.348121  
> # Calcular 5 números aleatorios de una dist. gamma con ALPHA = 3 y BETA = 0.5  
> rgamma(5, 3, rate = 0.5)  
[1] 2.904413 6.250474 12.627805 11.155582 4.750176
```

# Parte VIII

## Regresión

# Regresión

La regresión es una técnica estadística que analiza la relación de dos o mas variables que principalmente se utilizada para inferir datos a partir de otros y hallar una respuesta de lo que puede suceder. Esta nos permite conocer el cambio en una de las variables llamadas respuesta y que corresponde a otra conocida como variable explicativa.

Se pueden encontrar varios tipos de regresión, por ejemplo:

- Regresión lineal simple
- Regresión múltiple ( varias variables)
- Regresión logística

Algunas ecuaciones regresión son:

- Regresión Lineal :  $y = A + Bx$
- Regresión Logarítmica :  $y = A + B\ln(x)$
- Regresión Exponencial :  $y = Ac(bx)$
- Regresión Cuadrática :  $y = A + Bx + Cx^2$

## Regresión lineal

La función en R para obtener modelos de regresión lineal simple,  $Y=aX+b$ , es “lm”, aunque también se puede utilizar esta función para el análisis de la varianza y análisis de covarianza.

```
> a.docencia <- c(3,1,1,2,5,6,12,7,3,10,6,11,4,4,16,4,5,3,5,2)
> edad <- c(35,27,26,30,33,42,51,35,45,37,43,36,36,56,29,35,37,29,34,29)
> # Y=años de docencia y X=edad
> lm(a.docencia~edad)->r1
> r1
```

Call:

```
lm(formula = a.docencia ~ edad)
```

Coefficients:

(Intercept)	edad
1.3081	0.1156

Hay que tener en cuenta que el orden en el que se escriben las variables es de gran importancia, en este caso la variable dependiente es los años de docencia, por lo que pretendemos una ecuación de regresión ajustada del tipo:

$$y = b_0 + b_1x$$

Con  $y$  (edad) y  $x$  (años de docencia).

Siendo  $Intercept = b_0$ , la ordenada en el origen de la recta de regresión y  $x = b_1$ , valor numérico del estimador.

Por lo tanto, la recta es:  $y(x) = (1,3081) - (0,1156)x$

```
> # X=edad y Y=años de docencia
> lm(edad~a.docencia)->r2
> r2
```

```
Call:
lm(formula = edad ~ a.docencia)
```

```
Coefficients:
(Intercept)  a.docencia
    33.7407      0.4562
```

Ahora la variable dependiente es edad, por lo que pretendemos una ecuación de regresión ajustada del tipo:

$$x = a_0 + a_1y$$

En este caso  $x(y) = (33,7407) - (0,4562)y$

Es importante tener controlados los elementos de la regresión, para ver cual es cada uno utilizamos el argumento “coefficients” o “coef”.

```
> coef(r1)
(Intercept)      edad
    1.308086    0.115639
> r1$coefficients
(Intercept)      edad
    1.308086    0.115639
> r1$coefficients[1]
(Intercept)
    1.308086
> r1$coefficients[2]
      edad
0.115639
> r2$coefficients
(Intercept)  a.docencia
    33.740741    0.456229
> r2$coefficients[1]
(Intercept)
    33.74074
> r2$coefficients[2]
a.docencia
    0.456229
```

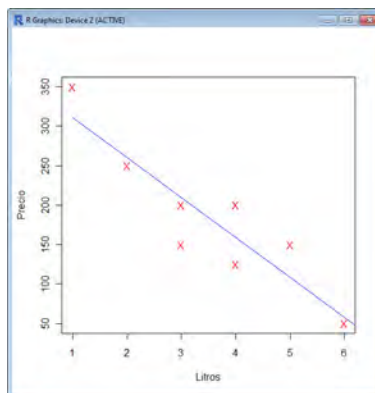
Vamos a dibujar la recta de regresión para un ejemplo cualquiera:

```
> # Defino los datos
> x <- c(3, 5, 2, 3, 1, 4, 6, 4)
> y <- c(150, 150, 250, 200, 350, 200, 50, 125)
> # Defino la recta de regresión
> lm(y~x)->ryx
```

```

> # Definimos el eje X
> litros <- seq(0:length(x))
> # Defino la recta
> precio <- (ryx$coefficients[1]) + (ryx$coefficients[2])*aceite# Ecuación ajustada
> #Dibujo los puntos, señalados con una X
> plot(x, y, pch="X", col=2, xlab="Litros", ylab="Precio")
> #Dibujo la recta
> lines(precio, col=4)

```

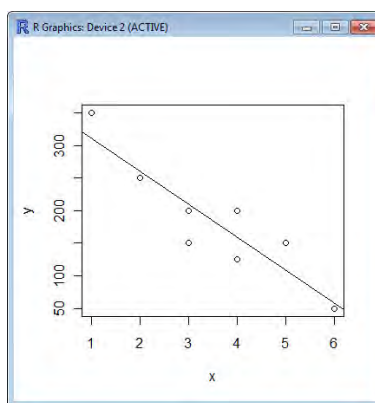


Aunque una forma más fácil sería con la función abline.

```

> # Creamos el diagrama de puntos
> plot(x, y)
> # Dibujamos la recta de regresión
> abline(lm(y ~ x))

```



```

> predict(ryx)
      1      2      3      4      5      6      7
209.72222 108.33333 260.41667 209.72222 311.11111 159.02778 57.63889
      8
159.02778
> # Son los valores de y cuando aplicamos la recta de regresión a los valores de x

```

Si lo que queremos es ajustar el modelo para poder usarlo posteriormente para predecir datos utilizaremos la función “predict”. Esta función obtiene todas las posibles predicciones para la variable x según la posición en la que se encuentren sus datos.

Si queremos saber que valor dará el modelo cuando  $x = 1$  que se encuentra en la posición 5 del vector, por lo que el valor predicho por el modelo es: 311.11111.

Si lo que queremos es calcular los residuos, es decir la diferencia entre los valores reales de la variable  $y$  y los predichos por el modelo utilizamos la función “residuals”.

```
> residuals(ryx)
      1      2      3      4      5      6      7
-59.72222 41.66667 -10.41667 -9.72222 38.88889 40.97222 -7.63889
      8
-34.02778
```

Pero para tener un resumen de las características más importantes de un modelo de regresión utilizaremos la función “summary”.

```
> summary(ryx)

Call:
lm(formula = y ~ x)

Residuals:
    Min       1Q   Median       3Q      Max
-59.72 -16.32  -8.68   39.41  41.67

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  361.806     36.482   9.917 6.07e-05 ***
x           -50.694     9.581  -5.291 0.00185 **
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 40.65 on 6 degrees of freedom
Multiple R-squared:  0.8235,    Adjusted R-squared:  0.7941
F-statistic:    28 on 1 and 6 DF,  p-value: 0.001845
```

Donde “Residuals” es un resumen de los residuos obtenidos por el modelo. “Coefficients” es una tabla con Estimate (valores de las estimaciones), sus errores estándar, el valor del estadístico y p-valor para las pruebas de hipótesis de los estimadores. Residual standard error es el valor del error estándar del residuo. Multiple R-squared de valor del coeficiente de determinación. Y Adjusted R-squared es el valor ajustado del coeficiente de determinación. El coeficiente de determinación es 0.8235 y el p-valor es 0.001845 por lo que existen diferencias significativas.

## Regresión lineal sin término independiente

Si lo que pretendemos es calcular una ecuación de regresión lineal sin termino independiente,  $Y=aX$ , tenemos que utilizar una variante de la anterior. Con  $lm(y \sim 0 + x)$  calculamos la pendiente (a) de la recta de regresión  $Y = aX$ . Aunque también es posible calcularlo con  $lm(y \sim x - 1)$ .

```
> x <- c(3, 5, 2, 3, 1, 4, 6, 4)
> y <- c(150, 150, 250, 200, 350, 200, 50, 125)
> # Recta de regresión sin termino independiente
> lm(y~0+x)
```

```
Call:
lm(formula = y ~ 0 + x)

Coefficients:
      x
36.64
> # Otra forma de calcularlo es con la orden lm(y~x-1)
> lm(y~x-1)
```

```
Call:
lm(formula = y ~ x - 1)

Coefficients:
      x
36.64
```

## Regresión Polinomial

Para calcular la función de regresión polinomial  $Y = a_0 + a_1X + a_2X^2 + \dots + a_pX^p$  utilizamos la función  $lm(y \sim x + I(x^2) + I(x^3) + \dots + I(x^p))$ .

```
> # Cuadrática
> lm(y~x+x^2)

Call:
lm(formula = y ~ x + x^2)

Coefficients:
(Intercept)          x
      361.81      -50.69
> # Cúbica
> lm(y~x+I(x^2)+I(x^3))

Call:
lm(formula = y ~ x + I(x^2) + I(x^3))

Coefficients:
(Intercept)          x      I(x^2)      I(x^3)
      596.820      -314.035       80.044       -7.127
> # De grado cuatro
> lm(y~x+I(x^2)+I(x^3)+I(x^4))

Call:
lm(formula = y ~ x + I(x^2) + I(x^3) + I(x^4))

Coefficients:
(Intercept)          x      I(x^2)      I(x^3)      I(x^4)
      404.112      14.090      -94.435       29.331       -2.604
```



## Regresión Polinomial sin término independiente

Como ocurría anteriormente calculamos la función de regresión polinomial  $Y = a_1X + a_2X^2 + \dots + a_pX^p$  con  $\text{lm}(y \sim 0 + x + I(x^2) + I(x^3) + \dots + I(x^p))$ .

```
> lm(y~0+x+x^2)

Call:
lm(formula = y ~ 0 + x + x^2)

Coefficients:
      x
36.64

> lm(y~0+x+I(x^2)+I(x^3))

Call:
lm(formula = y ~ 0 + x + I(x^2) + I(x^3))

Coefficients:
      x  I(x^2)  I(x^3)
326.50 -113.84   10.31

> lm(y~0+x+I(x^2)+I(x^3)+I(x^4))

Call:
lm(formula = y ~ 0 + x + I(x^2) + I(x^3) + I(x^4))

Coefficients:
      x  I(x^2)  I(x^3)  I(x^4)
659.318 -420.796   94.894  -7.145

> # También se calcula con lm(formula = y ~ x + x^2 - 1)
> lm(y~x+x^2-1)

Call:
lm(formula = y ~ x + x^2 - 1)

Coefficients:
      x
36.64
```

## Regresión Potencial

Para calcular la función de regresión potencial  $Y = aX^b$  utilizamos la función  $\text{lm}(\log(y) \sim \log(x))$ .

```
> lm(log(y)~log(x))

Call:
lm(formula = log(y) ~ log(x))

Coefficients:
(Intercept)      log(x)
   6.0369      -0.8338
```

Con “Intercept” el coeficiente “a” y “log(x)” el coeficiente “b”.

## Regresión exponencial

Para calcular la función de regresión potencial  $Y = e^{a+bX}$  utilizamos la función  $lm(\log(y) \sim x)$ .

```
> lm(log(y)~x)

Call:
lm(formula = log(y) ~ x)

Coefficients:
(Intercept)          x
      6.2048      -0.3179
```

Con “Intercept” el coeficiente “a” y “x” el coeficiente “b”.

## Regresión logarítmica

Para calcular la función de regresión logarítmica  $Y = a + b \log(x)$  utilizamos la función  $lm(y \sim \log(x))$ .

```
> lm(y~log(x))

Call:
lm(formula = y ~ log(x))

Coefficients:
(Intercept)    log(x)
      350.1      -146.3
```

Con “Intercept” el coeficiente “a” y “log(x)” el coeficiente “b”.

## Regresión hiperbólica

Para calcular la función de regresión hiperbólica  $Y = a + \frac{b}{x}$  utilizamos la función  $lm(y \sim I(1/x))$ .

```
> lm(y~I(1/x))

Call:
lm(formula = y ~ I(1/x))

Coefficients:
(Intercept)    I(1/x)
      72.45      295.18
```

Con “Intercept” el coeficiente “a” y “I(1/x)” el coeficiente “b”.

## Regresión doble inversa

Para calcular la función de regresión hiperbólica  $Y = \frac{1}{a + \frac{b}{x}}$  utilizamos la función  $lm(I(1/y) \sim I(1/x))$ .

```
> lm(I(y)~I(1/x))
```

Call:

```
lm(formula = I(y) ~ I(1/x))
```

Coefficients:

(Intercept)	I(1/x)
72.45	295.18

Con “Intercept” el coeficiente “a” y “I(1/x)” el coeficiente “b”.

## Coeficiente de correlación

Para obtener el modelo de regresión no es suficiente con establecer la regresión, ya que es necesario evaluar que lo bueno que es el modelo de regresión obtenido. El coeficiente de correlación mide el grado de relación existente entre las variables. El valor de este coeficiente varía entre -1 y 1, pero en la práctica se trabaja con el valor absoluto de R.

El coeficiente de correlación entre dos variables aleatorias X e Y es el cociente de su covarianza por el producto de sus desviaciones típicas.

$$r = \frac{\sigma_{xy}}{\sigma_x \sigma_y}$$

El valor del coeficiente de correlación se clasifica según el valor de r:

- Si  $r = 1$ , existe una correlación positiva perfecta. El índice indica una dependencia total entre las dos variables denominada relación directa: cuando una de ellas aumenta, la otra también lo hace en proporción constante.
- Si  $0 < r < 1$ , existe una correlación positiva.
- Si  $r = 0$ , no existe relación lineal. Pero esto no necesariamente implica que las variables son independientes: pueden existir todavía relaciones no lineales entre las dos variables.
- Si  $-1 < r < 0$ , existe una correlación negativa.
- Si  $r = -1$ , existe una correlación negativa perfecta. El índice indica una dependencia total entre las dos variables llamada relación inversa: cuando una de ellas aumenta, la otra disminuye en proporción constante.

Para calcular el coeficiente de correlación “r” podemos utilizar la función “cor” o calcularlo a partir de  $r^2$  con `summary(lm(y~x))[[8]]` calculando la raíz cuadrada de el.

```
> cor(x,y)
[1] -0.9074802
> # Esta función proporciona el signo de la dependencia
> summary(lm(y~x))[[8]]
[1] 0.8235203
> # r cuadrado
> summary(lm(y~x))[[8]]->r2
> sqrt(r2)
[1] 0.9074802
> # r en valor absoluto
```

# Parte IX

## Inferencia

# Inferencia

La inferencia estadística es una parte de la Estadística que comprende los métodos y procedimientos para deducir propiedades (hacer inferencias) de una población, a partir de una pequeña parte de la misma (muestra).

La bondad de estas deducciones se mide en términos probabilísticos, es decir, toda inferencia se acompaña de su probabilidad de acierto.

## Muestreo aleatorio

### Muestreo aleatorio simple

Selecciona muestras mediante métodos que permiten que cada posible muestra tenga igual probabilidad de ser seleccionada y que cada elemento de la población total tenga una oportunidad igual de ser incluido en la muestra.

Un muestreo aleatorio para un tamaño especificado lo realizamos con “sample”. Esta función se puede aplicar tanto a muestras con reemplazamiento como sin reemplazamiento indicando en ella tal reemplazamiento con el argumento “replace = TRUE” o sin el con “replace = FALSE” (por definición el argumento se toma como FALSE).

```
> x <- 1:12
> # Muestra aleatoria sin reemplazamiento del mismo tamaño
> sample(x)
[1] 10 2 5 11 4 8 6 7 12 9 1 3
> # Muestra aleatoria con reemplazamiento del mismo tamaño
> sample(x,replace=TRUE)
[1] 8 10 6 5 3 6 6 5 10 5 5 10
> # Muestras de tamaño 3 con y sin reemplazamiento
> sample(x,3,replace=TRUE)
[1] 12 10 10
> sample(x,3)
[1] 7 11 5
> # Muestra aleatoria con reemplazamiento de 100 elementos
> # de una Bernoulli
> sample(c(0,1), 100, replace = TRUE)
[1] 0 1 1 0 1 0 1 1 0 1 1 1 1 0 1 1 0 0 1 0 0 1 0 1 0 0 0 0 1 1 0 0 0 0 1 0 1
[38] 0 1 1 0 1 1 0 0 1 1 1 1 0 1 0 0 0 0 0 0 0 1 1 1 1 1 0 1 0 1 0 0 0 0 1 1 0
[75] 1 0 0 0 0 1 0 1 0 1 0 1 0 1 1 0 1 0 0 1 0 0 1 1 1 1
```

## Muestreo aleatorio sistemático

En el muestreo aleatorio sistemático los elementos de la población están ordenados de alguna forma (alfabéticamente, fecha, o algún otro método). Un primer elemento se selecciona de forma aleatoria y entonces cada  $n$  miembros de la población se toma para la muestra.

Suponga que la población de interés consiste de 2000 expedientes en un archivo. Para seleccionar una muestra de 100 con el método aleatorio simple primero se tendría que numerar todos los expedientes. En este método se selecciona el primer expediente de acuerdo al método aleatorio simple, luego como se quiere una muestra de 100, se divide  $2000 / 100 = 20$ , y se selecciona un expediente cada 20.

Para calcular un muestreo sistemático vamos a utilizar un paquete denominado “pps” y su orden específica “ppss” para calcular una muestra por muestreo sistemático.

```
> library(pps)
> x
[1] 1 2 3 4 5 6 7 8 9 10 11 12
> ppss(x,6)
[1] 4 6 8 10 12 2
> # Elige el elemento 4 y como 12/6=2 elige de dos en dos
> ppss(x,4)
[1] 6 9 12 3
```

## Muestreo estratificado

En el muestreo estratificado dividimos la población en grupos relativamente homogéneos, llamados estratos. Después seleccionamos aleatoriamente de cada estrato un número específico de elementos correspondientes a la fracción de ese estrato en la población como un todo o extraemos un número igual de elementos de cada estrato y damos peso a los resultados de acuerdo con la porción del estrato con respecto a la población total.

Con cualquiera de estos planteamientos, el muestreo estratificado garantiza que cada elemento de la población tenga posibilidad de ser seleccionado. Este método resulta apropiado cuando la población ya está dividida en grupos de diferentes tamaños y deseamos tomar en cuenta este hecho (por ejemplo: categorías profesionales de la población).

Para calcular un muestreo sistemático volvemos a utilizar el paquete “pps” y su orden específica “ppsstrat” para calcular una muestra por muestreo estratificado.

```
> library(pps)
> tamaños<-c(10, 20, 30, 40, 50, 100, 90, 80, 70, 60)
> # Indicamos el estrato al que pertenece cada tamaño
> estratos <- c(1,1,1,2,2,3,3,3,3,3)
> # n es un vector que contienen el tamaño de la muestra en cada estrato
> n <- c(2,1,3)
> ppssstrat(tamaños,estratos,n)
[1] 2 3 5 6 7 9
```

## Estimación de parámetros; Test de hipótesis

En muchas ocasiones el objetivo de la Estadística es hacer inferencia con respecto a parámetros poblacionales desconocidos, basadas en la información obtenida mediante datos muestrales. Estas inferencias se expresan de dos maneras, como estimaciones de los parámetros o con tests de hipótesis.

Con frecuencia, los problemas a los que nos enfrentamos no se refieren sólo a la estimación de un parámetro poblacional, sino a la toma de decisiones basadas en los datos. Por ejemplo cuando un investigador pretende demostrar que el fármaco  $z$  es más efectivo para el tratamiento de cierta enfermedad que el fármaco  $w$ .

### Contraste de normalidad

Un caso específico de ajuste a una distribución teórica es la correspondiente a la distribución normal. Este contraste se realiza para comprobar si se verifica la hipótesis de normalidad necesaria para que el resultado de algunos análisis sea fiable, como por ejemplo para el ANOVA.

Para comprobar la hipótesis nula de que la muestra ha sido extraída de una población con distribución de probabilidad normal se puede realizar un estudio gráfico y/o analítico.

### Prueba de Kolmogorov-Smirnov

Cuando la prueba Kolmogorov-Smirnov se aplica para contrastar la hipótesis de normalidad de la población, el estadístico de prueba es la máxima diferencia. La distribución del estadístico de Kolmogorov-Smirnov es independiente de la distribución poblacional especificada en la hipótesis nula y los valores críticos de este estadístico están tabulados. Para aplicar el test utilizamos la función “ks.test”.

```
> x<-exp(rnorm(25))
> ks.test(x, "pnorm", mean = mean(x), sd = sd(x))
```

One-sample Kolmogorov-Smirnov test

```
data: x
D = 0.383, p-value = 0.0008265
alternative hypothesis: two-sided
```

Por lo tanto como el p-valor es menor que el nivel de significación  $\alpha$  rechazo la hipótesis alternativa de que la distribución es normal.

Si queremos comprobar si dos muestras son normales utilizamos la misma función:

```
> y<-rnorm(25)
> ks.test(x, y, "pnorm", mean = mean(x), sd = sd(x))
```

Two-sample Kolmogorov-Smirnov test

```
data: x and y
D = 0.68, p-value = 8.494e-06
alternative hypothesis: two-sided
```

Por lo tanto como el p-valor es menor que el nivel de significación  $\alpha$  rechazo la hipótesis alternativa de que las dos muestras pertenecen a la misma distribución, en este caso la normal.

Aparte de comprobar si la distribución es la normal también podemos utilizar otras distribuciones (exponencial, uniforme, gamma, poisson,..)

```
> ks.test(x, y, "pgamma", mean = mean(x), sd = sd(x))
```

Two-sample Kolmogorov-Smirnov test

```
data: x and y
D = 0.68, p-value = 8.494e-06
alternative hypothesis: two-sided
```

```
> ks.test(x, y, "poisson", mean = mean(x), sd = sd(x))
```

Two-sample Kolmogorov-Smirnov test

```
data: x and y
D = 0.68, p-value = 8.494e-06
alternative hypothesis: two-sided
```

## Prueba de Shapiro-Wilk

Cuando la muestra es como máximo de tamaño 50 se puede contrastar la normalidad con la prueba de shapiro Shapiro-Wilk. Para efectuarla se calcula la media y la varianza muestral y se ordenan las observaciones de menor a mayor. A continuación se calculan las diferencias entre: el primero y el último; el segundo y el penúltimo; el tercero y el antepenúltimo, etc. y se corrigen con unos coeficientes tabulados por Shapiro y Wilk.

Se rechazará la hipótesis nula de normalidad si el estadístico es menor que el valor crítico proporcionado por los datos y el nivel de significación dado.

Para aplicar el test utilizamos la función “shapiro.test”.

```
> shapiro.test(x)
```

Shapiro-Wilk normality test

```
data: x
W = 0.3722, p-value = 2.538e-09
```

Por lo tanto como el p-valor es menor que el nivel de significación  $\alpha$  rechazo la hipótesis alternativa de que la distribución es normal.

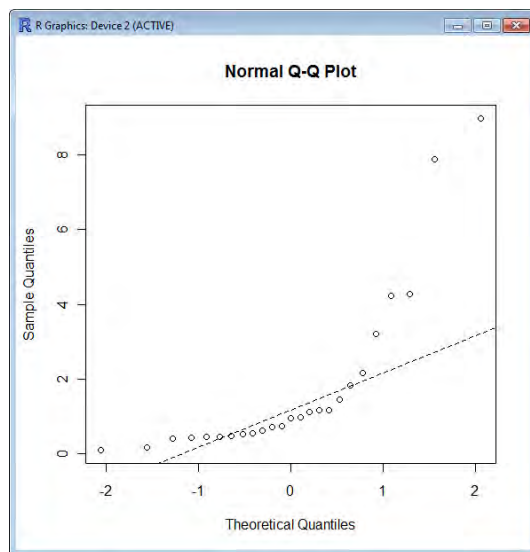
## Opciones gráficas de normalidad

Los gráficos Q-Q (“Q” viene de cuantil) son un método gráfico para el diagnóstico de diferencias entre la distribución de probabilidad de una población de la que se ha extraído una muestra aleatoria y una distribución usada para la comparación.

Un ejemplo del tipo de diferencias que pueden comprobarse es la no-normalidad de la distribución de una variable en una población. Para una muestra de tamaño  $n$ , se dibujan  $n$  puntos con los  $(n+1)$ -cuantiles de la distribución de comparación, por ejemplo, la distribución normal, en el eje horizontal el estadístico de  $k$ -ésimo orden, (para  $k = 1, \dots, n$ ) de la muestra en el eje vertical. Si la distribución de la variable es la misma que la distribución de comparación se obtendrá, aproximadamente, una línea recta, especialmente cerca de su centro. En el caso de que se den desviaciones sustanciales de la linealidad, los estadísticos rechazan la hipótesis nula de similitud.

```
> x<-exp(rnorm(25))
> qqnorm(x)
> qqline(x,lty=2)
```

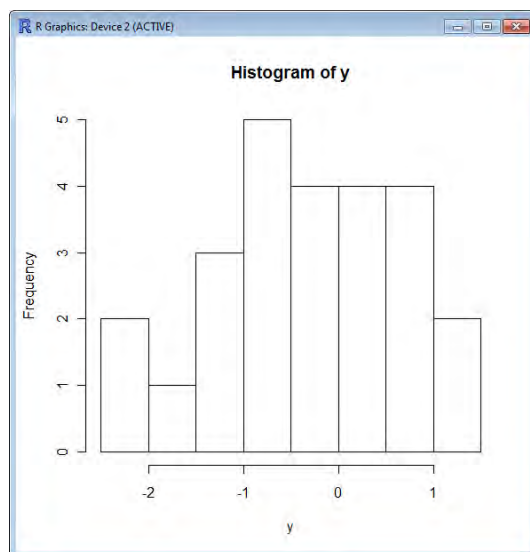




Como vemos hay diferencias entre los valores y la recta por lo que no son normales.

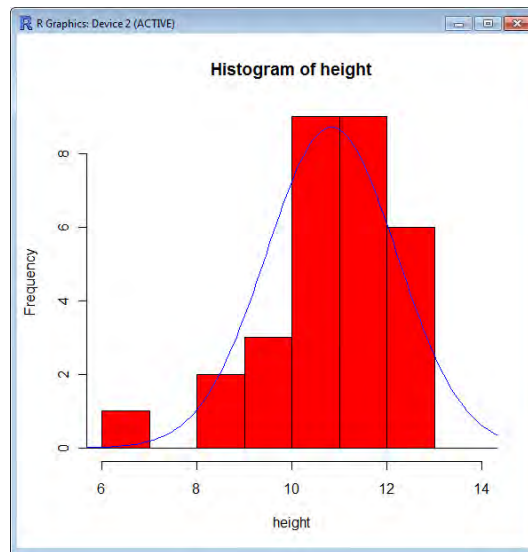
También podemos hacer la comparación de los datos con la distribución normal a partir del histograma. Si el histograma de los datos se asemeja a la distribución normal diremos que los datos son normales.

```
> y<-rnorm(25)
> hist(y)
```

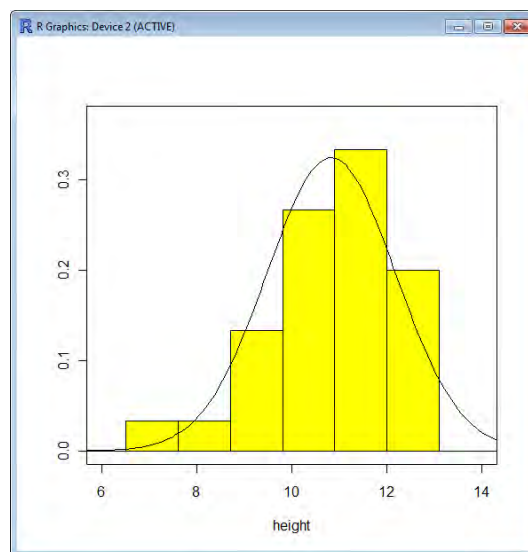


Podemos comprobar si el histograma coincide con la distribución normal dibujando ambas gráficas. Utilizamos el paquete "agricolae" y la función "normal.freq".

```
> library(agricolae)
> # Cogemos los datos que nos proporciona el paquete
> data(growth)
> attach(growth)
> #startgraph
> h1<-hist(height,col="red",xlim=c(6,14))
> normal.freq(h1,col="blue")
```



```
> h2<-graph.freq(height,col="yellow",xlim=c(6,14),frequency=2)
> normal.freq(h2,frequency=2)
```



## Contraste de medias

Un contraste de hipótesis (también denominado test de hipótesis o prueba de significación) es una metodología de inferencia estadística para juzgar si una propiedad que se supone cumple una población estadística es compatible con lo observado en una muestra de dicha población.

Mediante esta teoría, se aborda el problema estadístico considerando una hipótesis determinada (hipótesis nula)  $H_0$  e hipótesis alternativa,  $H_1$ , y se intenta dirimir cuál de las dos es la hipótesis verdadera, tras aplicar el problema estadístico a un cierto número de experimentos.

Posteriormente se puede evaluar la probabilidad de haber obtenido los datos observados si esa hipótesis es correcta. El valor de esta probabilidad coincide con el p-valor que nos proporciona cada test estadístico, de modo que cuanto menor sea éste más improbable resulta que la hipótesis inicial se verifique.

## t de Student para dos muestras independientes

Uno de los análisis estadísticos más comunes en la práctica es el utilizado para comparar dos grupos independientes de observaciones con respecto a una variable numérica que requiere la normalidad de las observaciones para cada uno de los grupos.

Bajo las hipótesis de normalidad e igual varianza la comparación de ambos grupos puede realizarse en términos de un único parámetro como el valor medio, de modo que en el ejemplo planteado la hipótesis de partida será, por lo tanto:

- $H_0$ : La media es igual en ambos grupos
- $H_1$ : La media es distinta en ambos grupos

Para realizar el test de Student de igualdad de medias utilizamos la función “t.test”.

```
> t.test(x=1:10,y=c(7:20))

Welch Two Sample t-test

data:  1:10 and c(7:20)
t = -5.4349, df = 21.982, p-value = 1.855e-05
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -11.052802  -4.947198
sample estimates:
mean of x mean of y
    5.5      13.5
```

Es decir como el p-valor es menor que  $\alpha = 0,05$  rechazamos la hipótesis nula de igualdad de medias.

Veamos que pasa cuando se aplica a un solo conjunto de datos.

```
> alt<-c(1.77,1.80,1.65,1.69,1.86,1.75,1.58,1.79,1.76,1.81,1.91,1.78,1.80,1.69,1.81)
> t.test(alt)

One Sample t-test

data:  alt
t = 82.3296, df = 14, p-value < 2.2e-16
alternative hypothesis: true mean is not equal to 0
95 percent confidence interval:
 1.717396 1.809270
sample estimates:
mean of x
 1.763333
> # Queremos estimar si la altura media es 1.78
> t.test(alt,mu=1.78)

One Sample t-test

data:  alt
t = -0.7782, df = 14, p-value = 0.4494
alternative hypothesis: true mean is not equal to 1.78
95 percent confidence interval:
 1.717396 1.809270
sample estimates:
mean of x
 1.763333
```

## F de Snedecor para dos muestras independientes

El caso en el que se dispone de dos grupos de observaciones independientes con diferentes varianzas, la distribución de los datos en cada grupo no puede compararse únicamente en términos de su valor medio. El contraste estadístico planteado en el apartado anterior requiere de alguna modificación que tenga en cuenta la variabilidad de los datos en cada población. Obviamente, el primer problema a resolver es el de encontrar un método estadístico que nos permita decidir si la varianza en ambos grupos es o no la misma. El F test o test de la razón de varianzas viene a resolver este problema. Bajo la suposición de que las dos poblaciones siguen una distribución normal y tienen igual varianza se espera que la razón de varianzas siga una distribución F de Snedecor con parámetros  $(n-1)$  y  $(m-1)$ .

Para realizar el test de igualdad de varianzas utilizamos la función “var.test”.

```
> # F para igualdad de varianzas
> var.test(x=1:10,y=c(7:20))

F test to compare two variances

data: 1:10 and c(7:20)
F = 0.5238, num df = 9, denom df = 13, p-value = 0.3343
alternative hypothesis: true ratio of variances is not equal to 1
95 percent confidence interval:
 0.1581535 2.0065024
sample estimates:
ratio of variances
 0.5238095
```

Como el p-valor es 0.3343 mayor que el nivel de significación  $\alpha$  aceptamos la hipótesis nula de igualdad de varianzas en la población.

## Test $\chi^2$ - cuadrado de Pearson

La prueba  $\chi^2$  de Pearson es considerada como una prueba no paramétrica que mide la discrepancia entre una distribución observada y otra teórica (bondad de ajuste), indicando en qué medida las diferencias existentes entre ambas, de haberlas, se deben al azar en el contraste de hipótesis. También se utiliza para probar la independencia de dos variables entre sí, mediante la presentación de los datos en tablas de contingencia. Es decir, esta prueba de significación estadística nos permite encontrar relación o asociación entre dos variables de carácter cualitativo que se presentan únicamente según dos modalidades (dicotómicas).

Cuanto mayor sea el valor de  $\chi^2$ , menos verosímil es que la hipótesis sea correcta. De la misma forma, cuanto más se aproxima a cero el valor de chi-cuadrado, más ajustadas están ambas distribuciones.

```
> x <- matrix(c(12, 5, 7, 7), ncol = 2)
> x
      [,1] [,2]
[1,]   12   7
[2,]    5   7
> chisq.test(x)

Pearson's Chi-squared test with Yates' continuity correction

data: x
X-squared = 0.6411, df = 1, p-value = 0.4233

> chisq.test(x)$p.value
[1] 0.4233054
```

```
> # Otro ejemplo
> x <- c(A = 20, B = 15, C = 25)
> x
  A  B  C
20 15 25
> chisq.test(x)
```

Chi-squared test for given probabilities

```
data:  x
X-squared = 2.5, df = 2, p-value = 0.2865
```

```
># Con as.table sale igual
> chisq.test(as.table(x))
```

Chi-squared test for given probabilities

```
data:  as.table(x)
X-squared = 2.5, df = 2, p-value = 0.2865
```

Por tanto en los dos ejemplo, al ser el p-valor mayor que el nivel de significación afirmamos que las distribución experimental se asemeja a la teórica.

En el caso comparación de dos variables cualitativas la hipótesis nula va a ser la de no relación o independencia.

```
> aa=rbind(c(17,35),c(8,53),c(22,491))
> aa
      [,1] [,2]
[1,]   17   35
[2,]    8   53
[3,]   22  491
> chisq.test(aa)
```

Pearson's Chi-squared test

```
data:  aa
X-squared = 57.9122, df = 2, p-value = 2.658e-13
```

Mensajes de aviso perdidos

```
In chisq.test(aa) : Chi-squared approximation may be incorrect
```

Como el p-valor es muy pequeño rechazamos la hipótesis nula de independencia por lo que las dos variables de la tabla de contingencia estarían relacionadas.

# Parte X

## Análisis de la varianza

# Análisis de la varianza

El análisis de la varianza o análisis de varianza (ANOVA) es un conjunto de modelos estadísticos y sus procedimientos asociados, en el cual la varianza esta particionada en ciertos componentes debidos a diferentes variables explicativas. El análisis de varianza sirve para comparar si los valores de un conjunto de datos numéricos son significativamente distintos a los valores de otro o más conjuntos de datos. El procedimiento para comparar estos valores está basado en la varianza global observada en los grupos de datos numéricos a comparar. Típicamente, el análisis de varianza se utiliza para asociar una probabilidad a la conclusión de que la media de un grupo de puntuaciones es distinta de la media de otro grupo de puntuaciones.

El ANOVA parte de algunos supuestos que han de cumplirse:

- La variable dependiente debe medirse al menos a nivel de intervalo.
- Independencia de las observaciones.
- La distribución de los residuales debe ser normal.
- Homocedasticidad: homogeneidad de las varianzas.

El método consiste en la descomposición de la suma de cuadrados en componentes relativos a los factores contemplados en el modelo.

$$SSTotal = SSError + SSFactores$$

El análisis de varianza se realiza mediante de pruebas de significación usando la distribución F de Snedecor.

## Modelo de efectos fijos

El modelo de efectos fijos de análisis de la varianza se aplica a situaciones en las que el experimentador ha sometido al grupo o material analizado a varios factores, cada uno de los cuales le afecta sólo a la media, permaneciendo la "variable respuesta" con una distribución normal.

## Modelo de efectos aleatorios

Los modelos de efectos aleatorios se usan para describir situaciones en que ocurren diferencias incompatibles en el material o grupo experimental. El ejemplo más simple es el de estimar la media desconocida de una población compuesta de individuos diferentes y en el que esas diferencias se mezclan con los errores del instrumento de medición.

## Diseño de un factor equilibrado

La hipótesis que se pone a prueba en el ANOVA de un factor es que las medias poblacionales son iguales. Si las medias poblacionales son iguales, eso significa que los grupos no difieren y que, en consecuencia, la variable independiente o factor es independiente de la variable dependiente.

```

> # Defino los valores de los tres niveles
> Nivel.A <- c(5.1, 4.9, 4.7, 4.4, 5, 5.4, 4.4, 5, 4.4, 4.9, 5.4, 4.8, 4.8, 4.3, 5.8)
> Nivel.B <- c(7, 4.4, 4.9, 5.5, 4.5, 5.7, 4.3, 4.9, 4.4, 5.2, 5, 5.9, 4, 4.1, 5.4)
> Nivel.C <- c(4.3, 5.8, 7.1, 4.3, 4.5, 7.4, 4.9, 7.3, 4.7, 7.2, 4.5, 4.4, 4.8, 5.7, 5.8)
> # Creo un vector con lo valores
> valores <- c(Nivel.A, Nivel.B, Nivel.C)
> # Defino los grados de libertad, con 3 niveles y 15 datos
> Niveles <- gl(3, 15, labels = c("Nivel.A", "Nivel.B", "Nivel.C"))
> # Represento los valores con los niveles
> split(valores, Niveles)
$Nivel.A
[1] 5.1 4.9 4.7 4.4 5.0 5.4 4.4 5.0 4.4 4.9 5.4 4.8 4.8 4.3 5.8

$Nivel.B
[1] 7.0 4.4 4.9 5.5 4.5 5.7 4.3 4.9 4.4 5.2 5.0 5.9 4.0 4.1 5.4

$Nivel.C
[1] 4.3 5.8 7.1 4.3 4.5 7.4 4.9 7.3 4.7 7.2 4.5 4.4 4.8 5.7 5.8

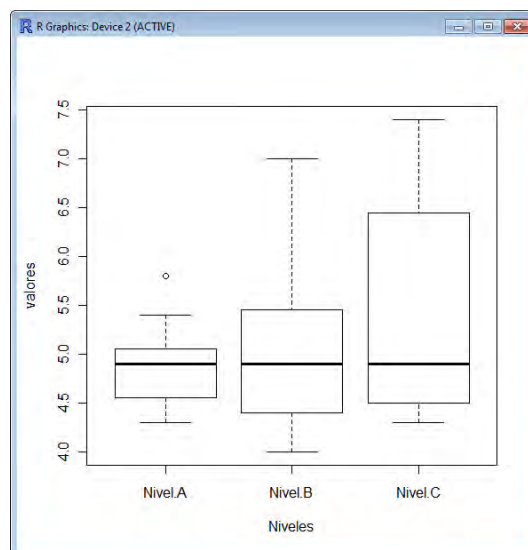
> # Veo un resumen de cada uno de los niveles del factor
> tapply(valores, Niveles, summary)
$Nivel.A
  Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 4.300   4.550   4.900   4.887   5.050   5.800

$Nivel.B
  Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 4.000   4.400   4.900   5.013   5.450   7.000

$Nivel.C
  Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 4.300   4.500   4.900   5.513   6.450   7.400

> # Representamos los niveles para ver como se comportan.

```



Asumimos que la variable valores sigue una distribución normal con varianza común para las tres poblaciones, la tabla de análisis de la varianza la podemos crear a partir de la función “aov” que muestra un análisis



de la varianza para cada estrato.

```
> anova <- aov(valores ~ Niveles)
> anova
Call:
  aov(formula = valores ~ Niveles)

Terms:
              Niveles Residuals
Sum of Squares   3.29378   31.67200
Deg. of Freedom         2         42

Residual standard error: 0.8683866
Estimated effects may be unbalanced
> summary(anova)
              Df Sum Sq Mean Sq F value Pr(>F)
Niveles        2  3.294   1.6469   2.1839 0.1252
Residuals     42 31.672   0.7541
```

Otra posibilidad es definir el modelo lineal y posteriormente obtener la tabla del ANOVA.

```
> regresión <- lm(valores ~ Niveles)
> anova(regresión)
Analysis of Variance Table

Response: valores
              Df Sum Sq Mean Sq F value Pr(>F)
Niveles        2  3.294   1.6469   2.1839 0.1252
Residuals     42 31.672   0.7541
```

Como el p-valor es mayor que el nivel de significación se concluye que no hay diferencias significativas entre los tres niveles. Las estimaciones de los parámetros se obtienen con la función “model.tables”.

```
> model.tables(anova)
Tables of effects

      Niveles
Niveles
Nivel.A Nivel.B Nivel.C
-0.2511 -0.1244  0.3756
> model.tables(anova, type = "mean")
Tables of means
Grand mean

5.137778

      Niveles
Niveles
Nivel.A Nivel.B Nivel.C
  4.887   5.013   5.513
> # Obtenemos más información con summary.
> summary(anova)
```

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
Niveles	2	3.294	1.6469	2.1839	0.1252
Residuals	42	31.672	0.7541		

Veamos un ejemplo con datos faltantes. Queremos ver si hay diferencias entre las valoraciones de cuatro jugadores de baloncesto, teniendo en cuenta que no han jugado todos los partidos.

Las valoraciones son:

Jugador										
jugador.A	20	10	15	-10	1	12	-3	NA	NA	NA
jugador.B	10	25	13	15	30	8	7	0	15	10
jugador.C	5	10	15	20	5	15	25	2	13	5
jugador.D	1	5	-5	5	2	0	4	6	NA	NA

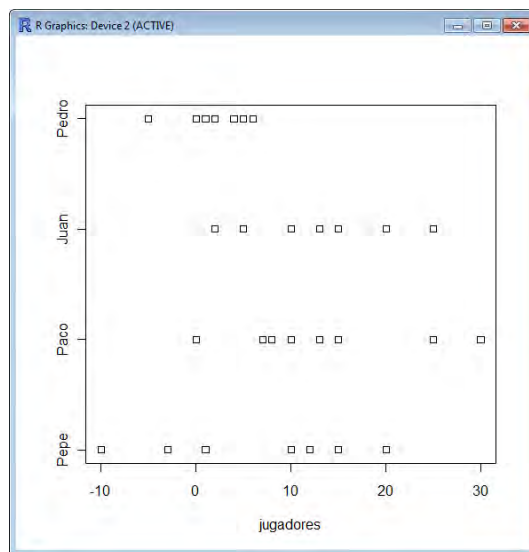
```
> # Definimos los valores
> jugador.A <- c(20, 10, 15, -10, 1, 12, -3, NA, NA, NA)
> jugador.B <- c(10, 25, 13, 15, 30, 8, 7, 0, 15, 10)
> jugador.C <- c(5, 10, 15, 20, 5, 15, 25, 2, 13, 5)
> jugador.D <- c(1, 5, -5, 5, 2, 0, 4, 6, NA, NA)
> jugadores <- c(jugador.A, jugador.B, jugador.C, jugador.D)
> valoración<- gl(4, 10, labels = c("Pepe", "Paco", "Juan", "Pedro"))
> # Vemos como se comportan los distintos niveles
> mean(jugadores, na.rm = TRUE)
[1] 8.885714
> tapply(jugadores, valoración, summary)
$Pepe
  Min. 1st Qu.  Median    Mean 3rd Qu.    Max.   NA's
-10.000 -1.000  10.000   6.429  13.500  20.000   3.000

$Paco
  Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   0.0    8.5    11.5    13.3   15.0   30.0

$Juan
  Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   2.0    5.0    11.5    11.5   15.0   25.0

$Pedro
  Min. 1st Qu.  Median    Mean 3rd Qu.    Max.   NA's
 -5.00    0.75    3.00    2.25    5.00    6.00    2.00

> stripchart(jugadores ~ valoración)
```



```
> reg <- lm(jugadores ~ valoración)
> anova(reg)
Analysis of Variance Table

Response: jugadores
          Df Sum Sq Mean Sq F value    Pr(>F)
valoración  3  657.73  219.243   3.4503 0.02837 *
Residuals 31 1969.81   63.542
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Como el p-valor es menor que el nivel de significación rechazamos la hipótesis de que los cuatro jugadores se comportan de la misma manera.

Como el resultado de la tabla de ANOVA resulta significativa, surge un problema hay que determinar qué medias son distintas entre sí. Utilizamos el test de Tukey para comparar las medias dos a dos. Para utilizar la función “TukeyHSD” tenemos que calcular la tabla de ANOVA utilizando la función “aov”.

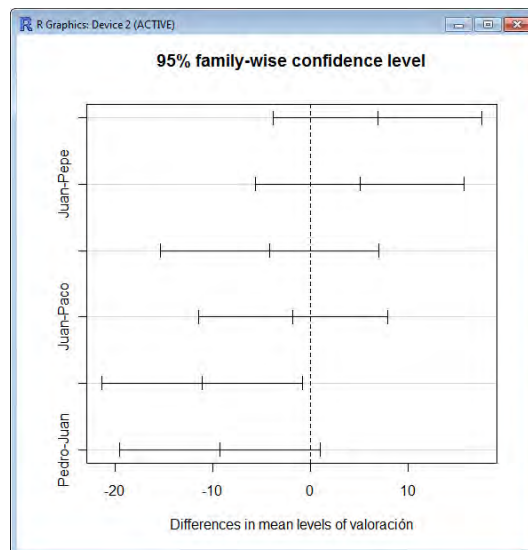
```
> aov(jugadores ~ valoración)->anv; summary(anv)
          Df Sum Sq Mean Sq F value    Pr(>F)
valoración  3  657.73  219.243   3.4503 0.02837 *
Residuals 31 1969.81   63.542
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
5 observations deleted due to missingness
> TukeyHSD(anv)
Tukey multiple comparisons of means
 95% family-wise confidence level
```

```
Fit: aov(formula = jugadores ~ valoración)
```

```
$valoración
      diff      lwr      upr    p adj
Paco-Pepe  6.871429 -3.790324 17.5331811 0.3165254
Juan-Pepe  5.071429 -5.590324 15.7331811 0.5753988
Pedro-Pepe -4.178571 -15.375651  7.0185084 0.7431954
```

```
Juan-Paco    -1.800000 -11.475383  7.8753828  0.9573021
Pedro-Paco   -11.050000 -21.312293 -0.7877068  0.0309962
Pedro-Juan   -9.250000 -19.512293  1.0122932  0.0891511
```

```
> plot(TukeyHSD(anv))
```



## Diseño de dos factores

Queremos comparar si la acidez depende de la edad del vino y del viñedo.

```
> tapply(acidez, edad, summary)
```

```
$ '1'
  Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   1.8    1.9    2.0    2.0    2.1    2.2
```

```
$ '2'
  Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 2.20    2.50    2.60    2.56    2.70    2.80
```

```
$ '3'
  Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 1.60    1.80    1.90    1.84    1.90    2.00
```

```
$ '4'
  Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 2.00    2.10    2.10    2.16    2.20    2.40
```

```
> tapply(acidez, viñedo, summary)
```

```
$ '1'
  Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 1.800    2.025    2.100    2.050    2.125    2.200
```

```
$ '2'
  Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
```

1.900 1.975 2.100 2.175 2.300 2.600

\$'3'

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
1.600	1.750	2.000	2.075	2.325	2.700

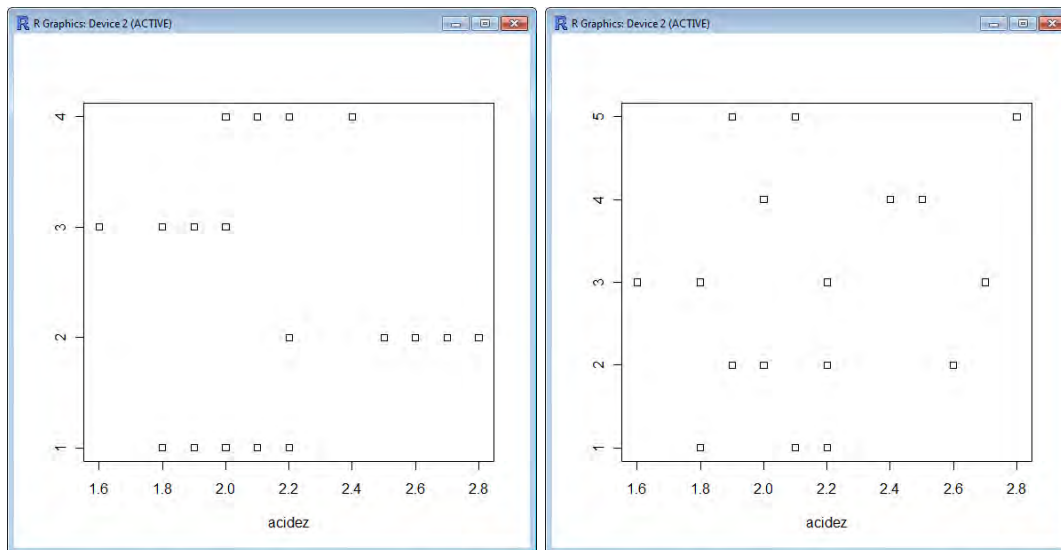
\$'4'

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
2.000	2.000	2.200	2.225	2.425	2.500

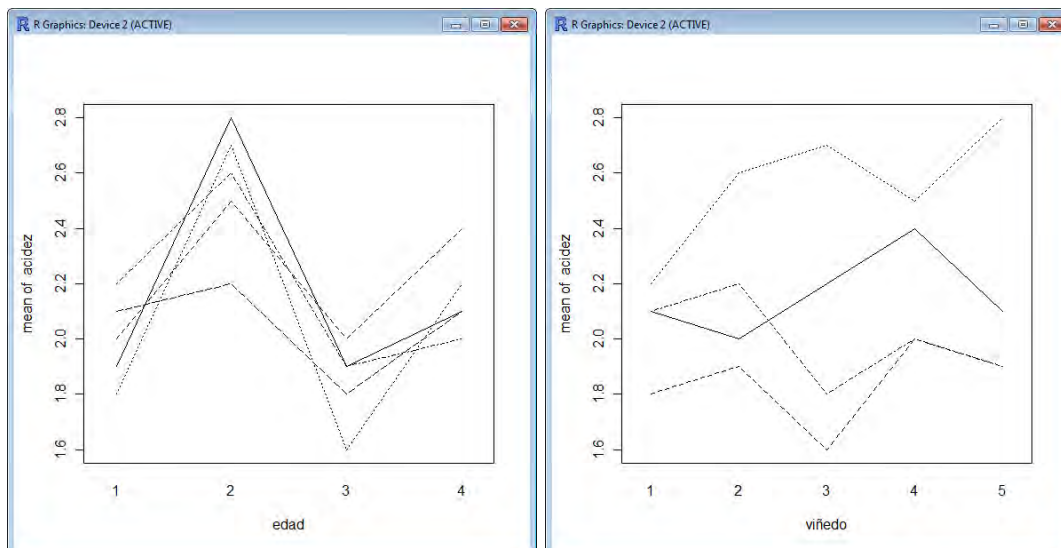
\$'5'

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
1.900	1.900	2.000	2.175	2.275	2.800

```
> stripchart(acidez ~ edad)
> stripchart(acidez ~ viñedo)
```



```
> interaction.plot(edad, viñedo, acidez, legend = F)
> interaction.plot(viñedo, edad, acidez, legend = F)
```



A la vista de las gráficas, no hay datos atípicos, asimetrías o heterocedasticidad. Tampoco parece haber interacciones.

Calculamos la tabla de ANOVA

```
> reg2 <- lm(acidez ~ edad + viñedo)
> anova(reg2)
Analysis of Variance Table

Response: acidez
      Df Sum Sq Mean Sq F value    Pr(>F)
edad    3  1.432  0.47733  14.0392 0.0003137 ***
viñedo   4  0.088  0.02200   0.6471 0.6395716
Residuals 12  0.408  0.03400
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Como vemos hay diferencias significativas entre la edad pero no entre los viñedos.

## Parte XI

# Combinatoria y probabilidad

# Combinatoria y probabilidad

La probabilidad mide la frecuencia con la que se obtiene un resultado (o conjunto de resultados) al llevar a cabo un experimento aleatorio, del que se conocen todos los resultados posibles, bajo condiciones suficientemente estables. La teoría de la probabilidad se usa extensamente en áreas como la estadística, la física, la matemática, la ciencia y la filosofía para sacar conclusiones sobre la probabilidad de sucesos potenciales y la mecánica subyacente de sistemas complejos.

Para este capítulo vamos a utilizar el paquete “prob” diseñada por G. Jay Kerns de la Youngstown State University. Para enseñar cómo funciona la probabilidad lo primero que vamos a construir son diferentes espacios muestrales y servirnos de la combinatoria para ver las diferentes variantes de estos.

## Espacio muestral

El primer paso es crear un espacio muestral, es decir el conjunto de posibles resultados simples de un experimento. La forma más simple de un espacio muestral está representado por un conjunto de datos, cada uno correspondiente a un resultado del experimento. Sin embargo, esta estructura no es lo suficientemente rica para describir algunas de las aplicaciones más interesantes de probabilidad nos encontraremos. Hay que tener en cuenta que el programa genera la mayor parte de los espacios muestrales mediante la orden “expand.grid” del paquete básico, aunque también lo podemos construir con la función “combn” del paquete “combinat”.

Por ejemplo para calcular el espacios muestral del experimento de lanzar una moneda, cuyos resultados pueden ser cara o cruz (heads and tails), podemos configurar el espacio muestral con la función “tosscoin”.

Hemos modificado la orden, que denominamos “EMmonedas” para adaptarla al Español para que sea más cómoda su interpretación.

```
> # Calcula el espacio muestral del lanzamiento de n monedas
> EMmonedas<- function (n)
+ {
+   temp <- list()
+   for (i in 1:n) {
+     temp[[i]] <- c("Cara", "Cruz")
+   }
+   resultados <- expand.grid(temp)
+   names(resultados) <- c(paste(rep("Lanzamiento ", n), 1:n, sep = ""))
+   return(resultados)
+ }
```

Veamos algunos ejemplos:

```
> # Calcular el espacio muestral de la variable aleatoria tirar dos monedas.
> EMmonedas(2)
Lanzamiento 1 Lanzamiento 2
```



```

1      Cara      Cara
2      Cruz      Cara
3      Cara      Cruz
4      Cruz      Cruz
> # Calcular el espacio muestral de la variable aleatoria tirar tres monedas.
> EMmonedas(3)
  Lanzamiento 1 Lanzamiento 2 Lanzamiento 3
1      Cara      Cara      Cara
2      Cruz      Cara      Cara
3      Cara      Cruz      Cara
4      Cruz      Cruz      Cara
5      Cara      Cara      Cruz
6      Cruz      Cara      Cruz
7      Cara      Cruz      Cruz
8      Cruz      Cruz      Cruz

```

Consideramos ahora el lanzamiento de un dado no sesgado, es decir, con la misma probabilidad de ocurrencia de cualquiera de las caras. Para ello utilizamos la función “rolldie( )” del paquete prob o la modificación “EMdados( )”

```

> # Calcula el espacio muestral del lanzamiento de n dados
> Emdados <- function (n, nlados = 6)
+ {
+   temp = list()
+   for (i in 1:n) {
+     temp[[i]] <- 1:nlados
+   }
+   resultados <- expand.grid(temp, KEEP.OUT.ATTRS = FALSE)
+   names(resultados) <- c(paste(rep("X", n), 1:n, sep = ""))
+   return(resultados)
+ }

```

Si queremos jugar con el número de lados del dado, por ejemplo un dado de 4 lados, utilizamos una variante de la anterior a la que denominaremos “EMdadosLado”.

```

> # Calcula el espacio muestral del lanzamiento de n dados con l dados
> Emdadoslado <- function (n, l)
+ {
+   temp = list()
+   for (i in 1:n) {
+     temp[[i]] <- 1:l
+   }
+   resultados <- expand.grid(temp)
+   names(resultados) <- c(paste(rep("X", n), 1:n, sep = ""))
+   return(resultados)
+ }

```

Veamos algunos ejemplos:

```

> # Calcular el espacio muestral de la variable aleatoria tirar dos dados normales
> Emdados(2)
  X1 X2
1   1  1
2   2  1

```

```
3  3  1
4  4  1
5  5  1
6  6  1
7  1  2
8  2  2
9  3  2
10 4  2
11 5  2
12 6  2
13 1  3
14 2  3
15 3  3
16 4  3
17 5  3
18 6  3
19 1  4
20 2  4
21 3  4
22 4  4
23 5  4
24 6  4
25 1  5
26 2  5
27 3  5
28 4  5
29 5  5
30 6  5
31 1  6
32 2  6
33 3  6
34 4  6
35 5  6
36 6  6
> # Calcular el espacio muestral de la variable aleatoria tirar dos dados de 4 lados.
> Emdadoslado(2,4)
  X1 X2
1   1  1
2   2  1
3   3  1
4   4  1
5   1  2
6   2  2
7   3  2
8   4  2
9   1  3
10  2  3
11  3  3
12  4  3
13  1  4
14  2  4
15  3  4
16  4  4
```

Si lo que queremos es simular un juego con cartas podemos utilizar la función “cards” que nos muestra

el espacio muestral de una baraja clásica o la variante “BarajaESP” nos proporciona el mismo resultado pero para la baraja española.

```
> # muestra el espacio muestral de una baraja española
> BarajaESP<-function ()
+ {
+   x <- c(1:7, "Sota", "Caballo", "Rey")
+   y <- c("Oros", "Copas", "Espadas", "Bastos")
+   resultados <- expand.grid(Carta = x, Palo= y)
+   return(resultados)
+ }
```

## Toma de muestras de Urnas

Unos de los experimentos estadísticos fundamentales en la enseñanza de la probabilidad consisten en distinguir los objetos de dibujo de una urna. La función “urnsamples(x, size, replace, ordered)” crea un espacio muestral asociado con el experimento de toma de muestras de objetos distinguibles de una urna. El argumento x representa la urna, es decir el conjunto de datos de los que se ha de tomar la muestra, el argumento size representa el tamaño de la muestra. Replace representa un valor lógico que indica si el muestreo debe hacerse con el reemplazo o no y ordered es un valor lógico que indica si el orden entre las muestras es importante.

Por ejemplo, para calcular el espacio muestral de la variable aleatoria sacar dos bolas sin reemplazamiento de una urna de 5 bolas numeradas del uno al cinco. En este ejemplo el muestreo es sin reemplazamiento, de modo que no puede aparecer el mismo número dos veces en cualquier fila.

```
> # Necesitamos los paquetes combinat y prob
> library(combinat)
> library(prob)
> urnsamples(1:3, size = 2, replace = FALSE, ordered = TRUE)
  X1 X2
1  1  2
2  2  1
3  1  3
4  3  1
5  2  3
6  3  2
```

Si el muestreo es con reemplazamiento:

```
> urnsamples(1:3, size = 2, replace = TRUE, ordered = TRUE)
  X1 X2
1  1  1
2  2  1
3  3  1
4  1  2
5  2  2
6  3  2
7  1  3
8  2  3
9  3  3
```

¿Importa el orden? Si importaría ya que el espacio muestral sería diferente ya que ahora se tendría en cuenta que pueden aparecer el par (i,j) y el (j,i).

```
> urnsamples(1:3, size = 2, replace=TRUE, ordered=FALSE)
```

```
  X1 X2
1  1  1
2  1  2
3  1  3
4  2  2
5  2  3
6  3  3
```

Otro ejemplo va a ser construir el espacio muestral resultante de elegir dos bolas sin reposición de la urna a partir de una urna que contiene tres bolas rojas, dos amarillas y una azul.

```
> library(prob)
```

```
> x = c("Roja", "Roja", "Roja", "Amarilla", "Amarilla", "Azul")
```

```
> urnsamples(x, size = 2, replace=FALSE, ordered=FALSE)
```

```
  X1      X2
1   Roja   Roja
2   Roja   Roja
3   Roja Amarilla
4   Roja Amarilla
5   Roja    Azul
6   Roja   Roja
7   Roja Amarilla
8   Roja Amarilla
9   Roja    Azul
10  Roja Amarilla
11  Roja Amarilla
12  Roja    Azul
13 Amarilla Amarilla
14 Amarilla    Azul
15 Amarilla    Azul
```

```
> # Calculamos ahora el espacio muestral de elegir una muestra con reposición
```

```
> urnsamples(x, size = 2, replace=TRUE, ordered=FALSE)
```

```
  X1      X2
1   Roja   Roja
2   Roja   Roja
3   Roja   Roja
4   Roja Amarilla
5   Roja Amarilla
6   Roja    Azul
7   Roja   Roja
8   Roja   Roja
9   Roja Amarilla
10  Roja Amarilla
11  Roja    Azul
12  Roja   Roja
13  Roja Amarilla
14  Roja Amarilla
15  Roja    Azul
16 Amarilla Amarilla
17 Amarilla Amarilla
18 Amarilla    Azul
19 Amarilla Amarilla
```

20 Amarilla      Azul  
21      Azul      Azul

En muchos casos, no necesitamos generar los espacios muestrales de interés, sino que es suficiente simplemente con contar el número de resultados. La función “nsamp” calculará el número de filas de un espacio muestral hechas por urnsamples. Los argumentos son: nsamp(n, k, replace = FALSE, ordered = FALSE), con n el número de elementos que forman la población, k el número de elementos que forman la muestra, replace es un valor lógico que indica si hay o no reemplazamiento y ordered es otro valor lógico que indica si hay o no orden o repetición.

```
> # N° de elementos de la muestra del espacio muestral de elegir de una población de 3
> # elementos muestras de 2, con reemplazamiento y sin repetición.
> nsamp(n=3, k=2, replace = TRUE, ordered = TRUE)
[1] 9
> # N° de elementos de la muestra del espacio muestral de elegir de una población de 3
> # elementos muestras de 2, con reemplazamiento y con repetición.
> nsamp(n=3, k=2, replace = TRUE, ordered = FALSE)
[1] 6
> # N° de elementos de la muestra del espacio muestral de elegir de una población de 3
> # elementos muestras de 2, sin reemplazamiento y con repetición.
> nsamp(n=3, k=2, replace = FALSE, ordered = TRUE)
[1] 6
> # N° de elementos de la muestra del espacio muestral de elegir de una población de 3
> # elementos muestras de 2, sin reemplazamiento y sin repetición.
> nsamp(n=3, k=2, replace = FALSE, ordered = FALSE)
[1] 3
```

Los valores que puede tomar nsamp(n, k, replace, ordered) son:

	ordered = TRUE	ordered = FALSE
replace = TRUE	$n^k$	$\frac{(n-1+k)!}{(n-1)!k!}$
replace = FALSE	$\frac{n!}{n-k!}$	$\frac{n!}{(n-k)!k!}$

Vamos a utilizar esta orden a la hora de calcular combinaciones y variaciones con o sin repetición.

a) Combinaciones:

Podemos determinar el número de subgrupos de elementos que se pueden formar con los "n" elementos de una muestra. Cada subgrupo se diferencia del resto en los elementos que lo componen, sin que influya el orden. Por ejemplo, calcular las posibles combinaciones de 2 elementos que se pueden formar con los números 1, 2 y 3. Se pueden establecer 3 parejas diferentes: (1,2), (1,3) y (2,3). En el cálculo de combinaciones las parejas (1,2) y (2,1) se consideran idénticas, por lo que sólo se cuentan una vez.

Para calcular el número de combinaciones se aplica la siguiente fórmula:

$$C_{n,k} = \frac{n!}{(n-k)!k!} = 220$$

La expresión  $C_{n,k}$  representa las combinaciones de n elementos, formando subgrupos de k elementos. Por ejemplo  $C_{12,4}$  son las combinaciones de 12 elementos agrupándolos en subgrupos de 4 elementos:

$$\frac{12!}{(12-4)!4!}$$

Con la orden nsamp tendríamos que indicar que no habría reemplazamiento ni repetición mediante los comandos replace = FALSE y ordered = FALSE.

```
> # Combinaciones de 12 elementos agrupándolos en subgrupos de 3 elementos.
> nsamp(n=12, k=3, replace = FALSE, ordered = FALSE)
[1] 220
```

Es decir, podríamos formar 220 subgrupos diferentes de 3 elementos, a partir de los 12 elementos.

b) Combinaciones con repetición:

Vamos a analizar ahora que ocurriría con el cálculo de las combinaciones en el supuesto de que al formar los subgrupos los elementos pudieran repetirse. Por ejemplo: tenemos bolas de 6 colores diferentes y queremos formar subgrupos en los que pudiera darse el caso de que 2, 3, 4 o todas las bolas del subgrupo tuvieran el mismo color. Para calcular el número de combinaciones con repetición se aplica la siguiente fórmula:

$$CR_{n,k} = \frac{(n-1+k)!}{(n-1)!k!}$$

Por ejemplo las combinaciones de 12 elementos con repetición, agrupándolos en subgrupos de 3, en los que 2 o los 3 elementos podrían estar repetidos serían:

$$CR_{12,3} = \frac{(12-1+3)!}{(12-1)!3!} = 364$$

Es decir, podríamos formar 364 subgrupos diferentes de 3 elementos. El código en R, mediante los comandos `replace = TRUE` y `ordered = FALSE` sería:

```
> # Combinaciones con repetición de 12 elementos agrupándolos en subgrupos de 3 elementos.
> nsamp(n=12, k=3, replace = TRUE, ordered = FALSE)
[1] 364
```

c) Variaciones:

Para calcular el número de subgrupos de elementos que se pueden establecer con los “k” elementos de una muestra en el que cada subgrupo se diferencia del resto en los elementos que lo componen en el orden de dichos elementos (es lo que le diferencia de las combinaciones). Por ejemplo, para calcular las posibles variaciones de 2 elementos que se pueden establecer con los números 1, 2 y 3, tendríamos 6 posibles parejas: (1,2), (1,3), (2,1), (2,3), (3,1) y (3,3). En este caso los subgrupos (1,2) y (2,1) se consideran distintos.

Para calcular el número de variaciones se aplica la siguiente fórmula:

$$V_{n,k} = \frac{n!}{n-k!}$$

La expresión  $V_{n,k}$  representa las variaciones de n elementos, formando subgrupos de k elementos. En este caso un subgrupo se diferenciará del resto, bien por los elementos que lo forman, o bien por el orden de dichos elementos. Por ejemplo:  $V_{12,3}$  son las variaciones de 12 elementos agrupándolos en subgrupos de 3 elementos:

$$V_{12,3} = \frac{12!}{12-3!} = 1320$$

Es decir, podríamos formar 1320 subgrupos diferentes de 3 elementos, a partir de los 12 elementos. El código en R, mediante los comandos `replace = FALSE` y `ordered = TRUE` sería:

```
> # Variaciones sin repetición de 12 elementos agrupándolos en subgrupos de 3 elementos.
> nsamp(n=12, k=3, replace = FALSE, ordered = TRUE)
[1] 1320
```

d) Variaciones con repetición:

Para calcular el número de variaciones con repetición se aplica la siguiente fórmula:

$$VR_{n,k} = n^k$$

Por ejemplo:  $VR_{12,3}$  son las variaciones de 12 elementos con repetición, agrupándolos en subgrupos de 3 elementos:

$$VR_{12,3} = 12^3 = 1728$$

Es decir, podríamos formar 1728 subgrupos diferentes de 3 elementos. El código en R, mediante los comandos `replace = TRUE` y `ordered = TRUE` sería:

```
# Variaciones con repetición de 12 elementos agrupándolos en subgrupos de 3 elementos.
> nsamp(n=12, k=3, replace = TRUE, ordered = TRUE)
[1] 1728
```

Un resumen de las ordenes sería:

	Combinaciones
Sin repetición	<code>nsamp(n, k, replace = FALSE, ordered = FALSE)</code>
Con Repetición	<code>nsamp(n, k, replace = TRUE, ordered = FALSE)</code>

	Variaciones
Sin repetición	<code>nsamp(n, k, replace= FALSE, ordered = TRUE)</code>
Con Repetición	<code>nsamp(n, k, replace = TRUE, ordered = TRUE)</code>

## Espacio muestral

Un espacio probabilístico está definido como una terna  $(\Omega, B, P)$  compuesta por un espacio medible  $(\Omega, B)$  y la función de probabilidad  $P$  definida para todo suceso perteneciente a  $B$ , de manera que satisfaga los conocidos como axiomas de probabilidad:

1. Certeza de la presentación de  $\Omega$  en el experimento estocástico.

$$P(\Omega) = 1$$

2. La probabilidad de un suceso  $E$  es un número comprendido entre 0 y 1.

$$0 \leq P(E) \leq 1$$

3. La probabilidad de la presentación de uno de los sucesos indicados, es igual a la suma de las probabilidades de cada uno de esos sucesos, siempre que se trate de sucesos mutuamente excluyentes.

$$P(\cup_{i=1}^n E_i) = P(E_1) + \dots + P(E_n)$$

Para todo conjunto  $E_1 + \dots + E_n$  de sucesos disjuntos, es decir  $E_i \cap E_j = \emptyset$  para todo  $i \neq j$

El package “prob” nos permite tratar un espacio probabilístico como un objeto de resultados  $S$  y un vector de probabilidades que llamaremos “probs” con valores correspondientes a las probabilidades de cada resultado  $S$ . Cuando  $S$  es un conjunto de datos, es posible añadir una columna llamada “probs” a  $S$  por lo que el espacio de probabilidad será simplemente un conjunto de datos al podemos llamar el espacio. En el caso de que  $S$  sea una lista, podemos combinar los resultados y probs en una lista más grande, por lo que el espacio tendrá dos componentes: resultados y probs. El único requisito es que las entradas de probs no puedan ser negativas y su suma (probs) sea uno.

Para lograr esto en R, podemos usar la función “probspace”. La sintaxis general es `probspace(x, probs)`, donde `x` es un espacio muestral de resultados y `probs` es un vector (de la misma longitud que el número de resultados en `x`). Vamos a utilizar algunos ejemplos para demostrar algunas de las opciones más comunes de la función y de conceptos básicos.

Ejemplos:

## El modelo de equiprobabilidad

El modelo de equiprobabilidad afirma que todos los resultados del espacio muestral tienen la misma probabilidad, por lo tanto, si un espacio muestra los resultados de  $n$  elementos, entonces `probs` sería un vector de longitud  $n$  con entradas idénticas  $1/n$ . La forma más rápida de generar `probs` es con la función “`rep`”. Vamos a comenzar con el experimento de lanzar un dado normal por lo que  $n = 6$ . Vamos a construir el espacio muestral, generando el vector `probs`, y ponerlos juntos con `probspace`.

```
# Calculamos los valores de la variable aleatoria de lanzar un dado con la función rolldie que simula el
> resultados = rolldie(1); resultados
X1
1  1
2  2
3  3
4  4
5  5
6  6
> # Llamamos p al vector de probabilidades simples
> p = rep(1/6, times = 6); p
[1] 0.1666667 0.1666667 0.1666667 0.1666667 0.1666667 0.1666667
> probspace(resultados, probs = p)
X1      probs
1  1 0.1666667
2  2 0.1666667
3  3 0.1666667
4  4 0.1666667
5  5 0.1666667
6  6 0.1666667
> # Otra forma es mediante el vector 1:6
> probspace(1:6, probs = p)
X1      probs
1  1 0.1666667
2  2 0.1666667
3  3 0.1666667
4  4 0.1666667
5  5 0.1666667
6  6 0.1666667
```

Además, puesto que el modelo de la misma probabilidad juega un papel fundamental en el estudio de la probabilidad, la función `probspace` asignará las probabilidades si no se especifica el vector `probs`. De este modo, se obtiene la misma respuesta con:

```
> probspace(1:6)
x      probs
1 1 0.1666667
2 2 0.1666667
```



```

3 3 0.1666667
4 4 0.1666667
5 5 0.1666667
6 6 0.1666667
> probspace(resultados)
  X1      probs
1  1 0.1666667
2  2 0.1666667
3  3 0.1666667
4  4 0.1666667
5  5 0.1666667
6  6 0.1666667

```

Modificamos nuestra función para adaptarla al español:

```

espacioProb <- function (x, probs, ...){
  y <- data.frame(x)
  if (missing(probs)) {
    y$probs <- rep(1, dim(y)[1])/dim(y)[1]
  }
  else {
    if (any(probs < 0)) {
      stop("'probs' contiene valores negativos")
    }
    y$probs <- probs/sum(probs)
  }
  return(y)
}

> espacioProb(1:6)
  x      probs
1 1 0.1666667
2 2 0.1666667
3 3 0.1666667
4 4 0.1666667
5 5 0.1666667
6 6 0.1666667
> # creamos nuestro propio vector, por ejemplo un dado de quiniela
> espacioProb(c(1,"X",2))
  x      probs
1 1 0.3333333
2 X 0.3333333
3 2 0.3333333

```

¿Qué ocurre cuando los sucesos simples no son equiprobables?, pues que podemos asignar a la función el vector de probabilidades, por ejemplo:

```

# Si fuese un dado con 3 unos, dos equis y un 2
> espacioProb(c(1,"X",2), probs=c(0.5,0.3333,0.1667))
  x      probs
1 1 0.5000
2 X 0.3333
3 2 0.1667

```

En algunas situaciones se desea repetir un cierto experimento varias veces en las mismas condiciones y de forma independiente. Hemos visto muchos ejemplos de este hecho: lanzar una moneda varias veces, lanzar un dado o dados, etc.

## Ejemplo: Una moneda no equilibrada

Es bastante fácil de configurar el espacio de probabilidad para un lanzamiento, sin embargo, la situación se complica más cuando hay lanzamientos múltiples involucrados. Evidentemente, el suceso (Cara, Cara, Cara) no debería tener la misma probabilidad que (Cara, Cruz, Cruz).

La función “iidspace” fue diseñada específicamente para esta situación. Tiene tres argumentos: *x* que es un vector de resultados, *ntrials* que es un entero que identifica cuántas veces se repite el experimento, y *probs* para especificar las probabilidades de los resultados. Por ejemplo podemos representar el lanzamiento de nuestra moneda no equilibrada tres veces de la siguiente manera:

```
# Calculamos el espacio muestral de tirar una moneda no equilibrada tres veces
> iidspace(c("C", "X"), ntrials = 3, probs = c(0.7, 0.3))
  X1 X2 X3 probs
1  C  C  C 0.343
2  X  C  C 0.147
3  C  X  C 0.147
4  X  X  C 0.063
5  C  C  X 0.147
6  X  C  X 0.063
7  C  X  X 0.063
8  X  X  X 0.027
# Vemos que es verdad que las probabilidades suman 1
> sum(ss[,4])
[1] 1
```

## Subconjuntos de un espacio muestral

Un espacio muestral contiene todos los resultados posibles de un experimento. Un evento representa un subconjunto de el espacio muestral, es decir, un subgrupo determinado de resultados. Hay muchos métodos para hallar subconjuntos de datos, mencionaremos algunos de ellos de pasada.

Dada una muestra *S*, podemos extraer filas mediante el operador [ ]:

```
# Calculamos el espacio muestral de tirar una moneda equilibrada tres veces
# Utilizando para ello la función tosscoin del paquete prob
> S = tosscoin(3, makespace = TRUE)
> S
  toss1 toss2 toss3 probs
1     H     H     H 0.125
2     T     H     H 0.125
3     H     T     H 0.125
4     T     T     H 0.125
5     H     H     T 0.125
6     T     H     T 0.125
7     H     T     T 0.125
8     T     T     T 0.125
> # Tomamos las tres primeras filas de S
> S[1:3, ]
  toss1 toss2 toss3 probs
1     H     H     H 0.125
2     T     H     H 0.125
3     H     T     H 0.125
```

```
> # Tomamos las filas segunda y cuarta
> S[c(2, 4), ]
      toss1 toss2 toss3 probs
2       T      H      H 0.125
4       T      T      H 0.125
```

También podemos extraer las filas que satisfacen una expresión lógica con la función “subset”, por ejemplo:

```
> # Utilizamos la función cards del paquete prob para crear
> # el espacio muestral de sacar una carta de una baraja
> E = cards()
> # Tomamos el subconjunto de las cartas que son corazones
> subset(E, suit == "Heart")
      rank suit
27      2 Heart
28      3 Heart
29      4 Heart
30      5 Heart
31      6 Heart
32      7 Heart
33      8 Heart
34      9 Heart
35     10 Heart
36      J Heart
37      Q Heart
38      K Heart
39      A Heart
> # Tomamos el subconjunto de las cartas que su valor está entre 2 y 5
> subset(E, rank %in% 2:5)
      rank suit
1        2 Club
2        3 Club
3        4 Club
4        5 Club
14       2 Diamond
15       3 Diamond
16       4 Diamond
17       5 Diamond
27        2 Heart
28        3 Heart
29        4 Heart
30        5 Heart
40        2 Spade
41        3 Spade
42        4 Spade
43        5 Spade
```

También podemos tomar el subconjunto de elementos que cumplen alguna expresión matemática. Por ejemplo vamos a tomar del espacio muestral del experimento lanzamiento de tres dados aquellos subconjuntos que cumplan alguna condición matemática.

```
> # Tomamos el espacio muestral del experimento: lanzamiento de tres dados
> F<-rolldie(3)
> # Calculamos el subconjunto que cumpla que su suma sea mayor o igual que 18
```

```

> # Solo nos debe mostrar la terna (6,6,6)
> subset(F, X1 + X2 + X3 >= 18)
  X1 X2 X3
216  6  6  6
> # Calculamos el subconjunto que cumpla que su suma sea mayor que 16
> subset(F, X1 + X2 + X3 > 16)
  X1 X2 X3
180  6  6  5
210  6  5  6
215  5  6  6
216  6  6  6
> # Calculamos el subconjunto que cumpla que su suma sea igual a 3
> # Solo nos debe mostrar la terna (1,1,1)
> subset(F, X1 + X2 + X3 ==3)
  X1 X2 X3
1   1  1  1
> # Calculamos el subconjunto que cumpla que su suma sea igual a 4
> subset(F, X1 + X2 + X3 ==4)
  X1 X2 X3
2   2  1  1
7   1  2  1
37  1  1  2

```

Además de las formas descritas anteriormente podemos utilizar otras funciones para calcular subconjuntos, tales como `%in%`, `isin`, `all`.

```

> # Creo dos vectores
> x=1:8
> y=2:5
> # Compruebo si los elementos de y están en x
> # pero uno a uno
> y%in%x
[1] TRUE TRUE TRUE TRUE
> # Pero quizás nos gustaría saber si todo el vector y está en x.
> # Utilizamos la función isin
> isin(x, y)
[1] TRUE
> # También podemos hacerlo con la función all que en este caso daría los mismos resultados
> all(y %in% x)
[1] TRUE
> # Veamos que pasa cuando en un vector se repiten varios elementos
> z = c(3, 3, 7)
> all(z %in% x)
[1] TRUE
> # all nos dice que todos los elementos de z están en x
> isin(x, z)
[1] FALSE
> # isin nos dice que hay alguno que no está ya que comprueba elemento a elemento y all no
> # También podemos tener en cuenta el orden de los elementos
> isin(x, c(3, 4, 5), ordered = TRUE)
[1] TRUE
> isin(x, c(3, 5, 4), ordered = TRUE)
[1] FALSE

```

A partir de estas funciones podemos calcular subconjuntos o muestras de elementos de un conjunto de datos anexándolas con la función `sample`. Por ejemplo:

```
> # Calcular el espacio muestral del lanzamiento de 3 dados
> S = rolldie(3)
> # Calculamos el subconjunto de elementos que cumple que por lo menos
> # dos de ellos son el (2,6) en ese orden
> subset(S, isin(S, c(2, 6), ordered = TRUE))
  X1 X2 X3
32  2  6  1
68  2  6  2
104 2  6  3
140 2  6  4
176 2  6  5
182 2  1  6
187 1  2  6
188 2  2  6
189 3  2  6
190 4  2  6
191 5  2  6
192 6  2  6
194 2  3  6
200 2  4  6
206 2  5  6
212 2  6  6
> # Calculamos el subconjunto de elementos que cumple que por lo menos
> # dos de ellos son el (2,6) sin orden
> subset(S, isin(S, c(2, 6), ordered = FALSE))
  X1 X2 X3
12  6  2  1
32  2  6  1
42  6  1  2
48  6  2  2
54  6  3  2
60  6  4  2
66  6  5  2
67  1  6  2
68  2  6  2
69  3  6  2
70  4  6  2
71  5  6  2
72  6  6  2
84  6  2  3
104 2  6  3
120 6  2  4
140 2  6  4
156 6  2  5
176 2  6  5
182 2  1  6
187 1  2  6
188 2  2  6
189 3  2  6
190 4  2  6
191 5  2  6
192 6  2  6
194 2  3  6
200 2  4  6
```

206 2 5 6  
212 2 6 6

Hay otras funciones útiles a la hora de calcular subconjuntos tales como “countrep” que calcula en número de veces que uno o varios elementos aparecen en un determinado vector ‘y’ “isrep” que comprueba si se dan o no repeticiones.

```
> x <- c(3,3,2,2,3,3,4,4)
> # Elementos que se repiten 4 veces
> countrep(x, nrep = 4)
[1] 1
> # Elementos que se repiten 2 veces
> countrep(x, nrep = 2)
[1] 2
> # Elementos que se repiten 3 veces
> countrep(x, nrep = 3)
[1] 0
> Repetición de elementos cuyo valor sea el 4
> countrep(x, vals = 4)
[1] 1
> # veamos si se dan o no repeticiones
> isrep(x, nrep = 4)
[1] TRUE
> isrep(x, vals = 4)
[1] TRUE
```

A menudo es útil manipular de manera algebraica los subconjuntos. Para ello, contamos con tres operaciones de conjuntos: unión, intersección y diferencia. A continuación se muestra una tabla que resume la información pertinente acerca de estas operaciones.

Unión	$A \cup B$	union(A,B)
Intersección	$A \cap B$	intersect(A,B)
Diferencia	$A \setminus B$	setdiff(A,B)

```
> library(prob)
> # Vamos a calcular los sucesos a partir de la order cards
> S = cards()
> A = subset(S, suit == "Heart")
> # Corazones
> B = subset(S, rank %in% 2:3)
> # Valores del 2 al 3
> # Unión
> union(A, B)
  rank suit
1     2  Club
2     3  Club
14    2 Diamond
15    3 Diamond
27    2  Heart
28    3  Heart
29    4  Heart
30    5  Heart
31    6  Heart
32    7  Heart
```

```

33    8    Heart
34    9    Heart
35   10    Heart
36    J    Heart
37    Q    Heart
38    K    Heart
39    A    Heart
40    2    Spade
41    3    Spade
> # Intersección
> intersect(A, B)
      rank suit
27     2  Heart
28     3  Heart
> # Diferencia A menos B
> setdiff(A, B)
      rank suit
29     4  Heart
30     5  Heart
31     6  Heart
32     7  Heart
33     8  Heart
34     9  Heart
35    10  Heart
36     J  Heart
37     Q  Heart
38     K  Heart
39     A  Heart
> # Diferencia B menos A
> setdiff(B, A)
      rank suit
1       2  Club
2       3  Club
14      2 Diamond
15      3 Diamond
40      2  Spade
41      3  Spade

```

La función `setdiff` no es simétrica. Podemos calcular el complemento de un conjunto A, elementos de S que no están en A, mediante “`setdiff(S, A)`”.

```

> # Complementario de A
> NoA<-setdiff(S, A)
> NoA
      rank suit
1       2  Club
2       3  Club
3       4  Club
4       5  Club
5       6  Club
6       7  Club
7       8  Club
8       9  Club
9      10  Club

```

```

10    J    Club
11    Q    Club
12    K    Club
13    A    Club
14    2 Diamond
15    3 Diamond
16    4 Diamond
17    5 Diamond
18    6 Diamond
19    7 Diamond
20    8 Diamond
21    9 Diamond
22   10 Diamond
23    J Diamond
24    Q Diamond
25    K Diamond
26    A Diamond
40    2    Spade
41    3    Spade
42    4    Spade
43    5    Spade
44    6    Spade
45    7    Spade
46    8    Spade
47    9    Spade
48   10    Spade
49    J    Spade
50    Q    Spade
51    K    Spade
52    A    Spade
> # Complementario de B
> NoB<-setdiff(S, B)
> NoB
      rank    suit
3         4    Club
4         5    Club
5         6    Club
6         7    Club
7         8    Club
8         9    Club
9        10    Club
10        J    Club
11        Q    Club
12        K    Club
13        A    Club
16         4 Diamond
17         5 Diamond
18         6 Diamond
19         7 Diamond
20         8 Diamond
21         9 Diamond
22        10 Diamond
23         J Diamond
24         Q Diamond

```



```

25    K Diamond
26    A Diamond
29    4   Heart
30    5   Heart
31    6   Heart
32    7   Heart
33    8   Heart
34    9   Heart
35   10   Heart
36    J   Heart
37    Q   Heart
38    K   Heart
39    A   Heart
42    4   Spade
43    5   Spade
44    6   Spade
45    7   Spade
46    8   Spade
47    9   Spade
48   10   Spade
49    J   Spade
50    Q   Spade
51    K   Spade
52    A   Spade
> # Complementario del complementario de B
> NoNoB<-setdiff(S, setdiff(S, B))
> NoNoB
      rank    suit
1         2    Club
2         3    Club
14        2 Diamond
15        3 Diamond
27        2   Heart
28        3   Heart
40        2   Spade
41        3   Spade
> B
      rank    suit
1         2    Club
2         3    Club
14        2 Diamond
15        3 Diamond
27        2   Heart
28        3   Heart
40        2   Spade
41        3   Spade

```

## Cálculo de probabilidades

Para calcular probabilidades asociadas con subconjuntos utilizamos la función “prob”. Consideremos el experimento de sacar una carta de una baraja de naipes. Denotamos el espacio de probabilidad asociada con el experimento como  $S$  y los subconjuntos  $A$  (corazones) y  $B$  (cartas con valor entre 2 y 3).

```
> S = cards(makespace = TRUE)
> A = subset(S, suit == "Heart")
> B = subset(S, rank %in% 2:3)
> # Probabilidad de A
> prob(A)
[1] 0.25
> # Probabilidad de B
> prob(B)
[1] 0.1538462
> # Probabilidad del espacio muestral
> prob(S)
[1] 1
> # Probabilidad del complementario de B
> prob(setdiff(S, B))
[1] 0.8461538
```

## Probabilidad condicional

Probabilidad condicionada es la probabilidad de que ocurra un evento A, sabiendo que también sucede otro evento B.

$$P(A | B) = \frac{P(A \cap B)}{P(B)}$$

Para calcular la probabilidad condicional es necesario usar el argumento “given” en la función prob.

```
> S = cards(makespace = TRUE)
> A = subset(S, suit == "Heart")
> B = subset(S, rank %in% 2:3)
> # Probabilidad de A dado B
> prob(A, given = B)
[1] 0.25
> # Probabilidad de A dado que los valores están entre 2 y 3
> prob(S, suit == "Heart", given = rank %in% 2:3)
[1] 0.25
> # Coinciden
> # Probabilidad de B dado A
> prob(B, given = A)
[1] 0.1538462
> # Vemos si coincide con la definición de probabilidad condicionada
> prob(intersect(A, B))/prob(B)
[1] 0.25
```

## Triángulo de Pascal

Un elemento muy útil a la hora de la combinatoria y la probabilidad es el triángulo de Pascal definido como un conjunto infinito de números enteros ordenados en forma de triángulo que expresan coeficientes binomiales. El interés del Triángulo de Pascal radica en que permite calcular de forma sencilla números combinatorios aplicables al binomio de Newton.

Vamos a calcular el Triángulo de Pascal con la función:

```
TPascal <- function(filas){
  aux <- matrix(nrow=filas+1,ncol=filas)
```

```

for (j in 1:filas){
  aux[1,j] <- "-"
}
for (i in 1:filas+1){
  for (j in 0:filas)
    aux[i,j] <- "."
}
for(i in 1:filas+1){
  j <- 1
  aux[i,j] <- 1
  while(i-1 > j){
    aux[i,j+1] <- factorial(i-2)/(factorial(j)*(factorial((i-2)-j)))
    j <- j+1
  }
}
print(data.frame(aux,check.names=F), row.names = FALSE)
}

```

> Triángulo de Pascal de orden 15

> TPascal(15)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
1	1	.	.	.	.	.	.	.	.	.	.	.	.	.	.
1	2	1	.	.	.	.	.	.	.	.	.	.	.	.	.
1	3	3	1	.	.	.	.	.	.	.	.	.	.	.	.
1	4	6	4	1	.	.	.	.	.	.	.	.	.	.	.
1	5	10	10	5	1	.	.	.	.	.	.	.	.	.	.
1	6	15	20	15	6	1	.	.	.	.	.	.	.	.	.
1	7	21	35	35	21	7	1	.	.	.	.	.	.	.	.
1	8	28	56	70	56	28	8	1	.	.	.	.	.	.	.
1	9	36	84	126	126	84	36	9	1	.	.	.	.	.	.
1	10	45	120	210	252	210	120	45	10	1	.	.	.	.	.
1	11	55	165	330	462	462	330	165	55	11	1	.	.	.	.
1	12	66	220	495	792	924	792	495	220	66	12	1	.	.	.
1	13	78	286	715	1287	1716	1716	1287	715	286	78	13	1	.	.
1	14	91	364	1001	2002	3003	3432	3003	2002	1001	364	91	14	1	.

## Parte XII

### Entornos gráficos para trabajar con R

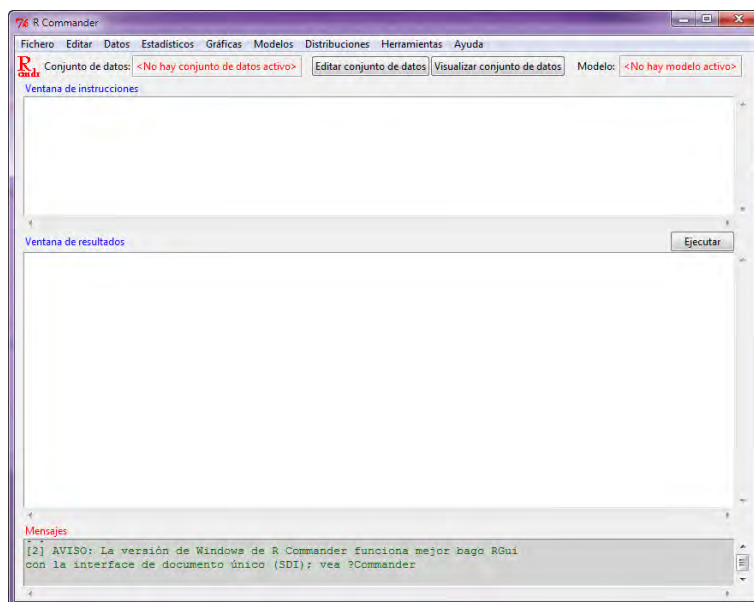
# Entornos gráficos para trabajar con R

R, a primera vista, parece un entorno solo para personas con grandes conocimientos en Estadística o en programación, lo que no es cierto ya que a diferencia de otros lenguajes este es muy fácil de aprender. Para los no iniciados en R (aunque siempre es necesario tener conocimientos como los explicados anteriormente) cuenta con entornos más amables que presentan un aspecto más habitual para los usuarios.

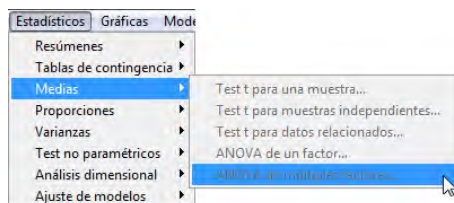
En este capítulo haremos una presentación de algunos de ellos, como el modulo *R – commander* que se ha convertido en el reclamo principal de R para no utilizar apenas programación. Pero como veremos en el mercado existen varios con sus virtudes y defectos.

## R-commander

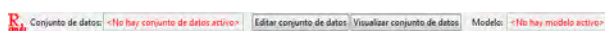
El R-commander (paquete Rcmdr) es una interfaz gráfica que se compone de una ventana que contiene varios menús, botones y campos de información. Por defecto R Commander consiste en una barra de menús, una barra de herramientas, una ventana de instrucciones, una ventana de salida y una ventana de mensajes.



Las instrucciones para leer, escribir, transformar y analizar datos se ejecutan usando la barra de menú de la parte superior de la ventana de R Commander. La mayor parte de los items de este menú le guiarán mediante ventanas de diálogo, preguntando más allá de la especificación.



Bajo la barra de menú se encuentra la barra de herramientas con un campo de información que muestra el nombre del conjunto de datos activos, botones para editar y mostrar el conjunto de datos activos y un campo de información mostrando el modelo estadístico activo. Bajo la ventana de instrucciones hay un botón Ejecutar para realizar las órdenes indicadas en la ventana de instrucciones. Los campos de información para los datos y el modelo activo son botones que pueden ser usados para seleccionar éstos entre, respectivamente, conjuntos de datos o modelos disponibles en memoria.



La mayor parte de las órdenes requiere un conjunto de datos activos. Cuando se ejecuta R Commander no hay conjunto de datos activos, como está indicado en el campo de información del conjunto de datos activos. Un conjunto de datos llega a ser un conjunto de datos activos cuando éste es leído en la memoria desde un paquete R o importado desde un archivo de texto, conjunto de datos SPSS, conjunto de datos Minitab, conjunto de datos STATA, Excel, Access o dBase. Además el conjunto de datos activos puede ser seleccionado desde conjuntos de datos R residentes en memoria. Los datos pueden ser elegidos de entre todos los conjuntos para cada sesión.

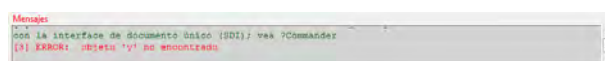
Por defecto, las órdenes son registradas en la ventana de instrucciones (la ventana de texto vacía inmediatamente después de la barra de herramientas); las órdenes y las salidas aparecen en la ventana de resultados (la ventana de texto vacía después de la ventana de instrucciones) y el conjunto de datos activos es adjuntado a la ruta de búsqueda. Para alterar éstos y otros parámetros por defecto, puede consultar la información pertinente en configuración.

Si el registro de instrucciones está activo, las órdenes de R generadas desde los menús y los cuadros de diálogos, son introducidas en la ventana de instrucciones de R Comander. Se pueden editar estas órdenes de manera normal y se pueden escribir otras nuevas en la ventana de instrucciones. Las órdenes individuales pueden ser continuadas en más de una línea. El contenido de la ventana de instrucciones puede ser almacenado durante o al final de la sesión y un conjunto de instrucciones guardado puede ser cargado en la ventana de instrucciones. El contenido de la ventana de resultados puede ser editado o guardado en un archivo de texto.

Para volver a ejecutar una orden o un conjunto de ellas, se seleccionan las líneas que se desean ejecutar usando el ratón y se presiona el botón Ejecutar, situado a la derecha de la barra de herramientas (o Control-R, para ejecutarlos). Si no hay texto seleccionado el botón Ejecutar (o Control-R) envía el contenido de la línea que contiene el cursor de inserción.



En la última ventana se generará información o un errores si la orden o las órdenes enviadas son incompletas.

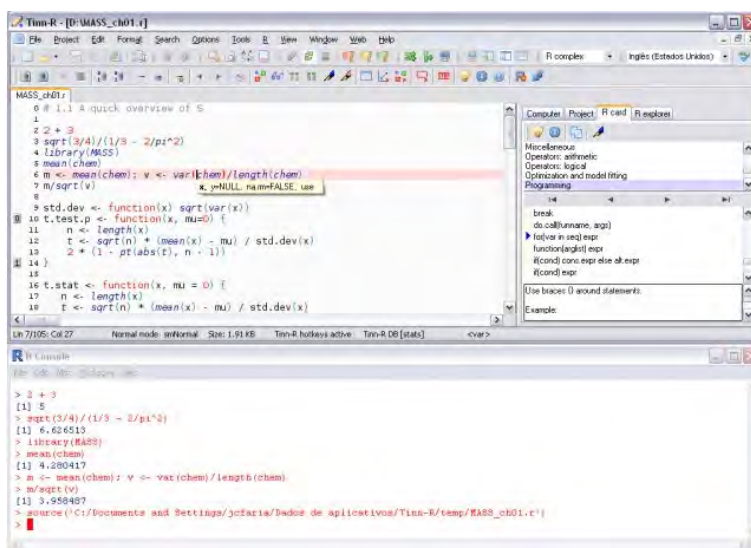


Para instalar R-Comander hay que instalar el paquete “Rcmdr”. En el caso de este paquete se van a instalar muchos otros paquetes (dependencias), que son necesarios para que puedan funcionar todas las opciones de los menús del entorno gráfico.

Cada vez que se quiera abrir el R Commander hay que teclear `library(Rcmdr)` en la consola de comandos, o entrar en el menú Paquetes -> Cargar paquete... y seleccionar Rcmdr en la ventana que aparece.

## Tinn-R

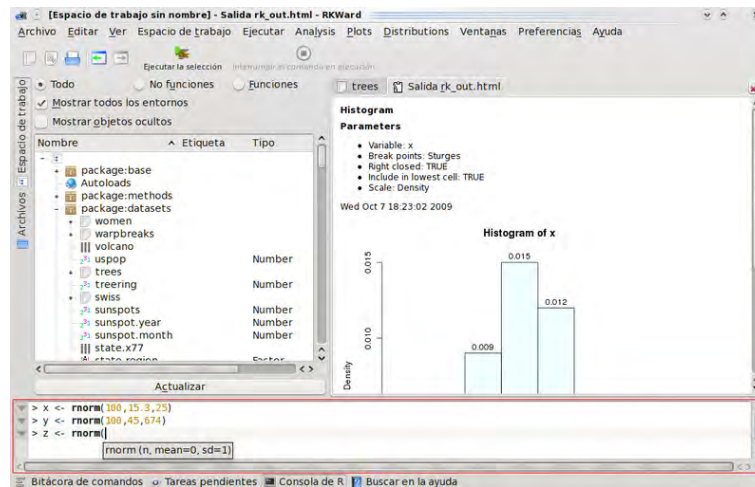
Tinn-R es un editor de archivos que mejora al bloc de notas de Windows ya que contiene mejoras para permitir el uso de sintaxis de R. También proporciona un menú y una barra de herramientas adicional que se comunica y llama al R-gui que nos permite ejecutar el código en el mismo programa.



## RKward

RKward ha sido desarrollado como una herramienta que combina el potencial de R con la facilidad de uso de otros paquetes estadísticos comerciales sin perder acceso a los beneficios del trabajo por línea de comandos o con guiones.

Gracias a su sistema de complementos RKward amplía constantemente el número de funciones a las cuales se puede acceder sin necesidad de escribir el código directamente. Estos componentes permiten que, a partir de una interfaz gráfica de usuario, se generen instrucciones en R para las operaciones estadísticas más usuales o complejas. De esta manera, incluso sin tener conocimientos profundos sobre el lenguaje es posible realizar análisis de datos avanzados o gráficas elaboradas. Los resultados de las computaciones son formateados y presentados como HTML, haciendo posible, con un sólo clic y arrastre, exportar tablas y gráficos hacia, por ejemplo, suites ofimáticas.



Incluye un visor del espacio de trabajo, donde se tiene acceso a los paquetes, funciones y variables cargados por R o importados de otras fuentes. Cuenta además con visor de archivos, y ventanas de edición de conjuntos de datos, visualización del contenido de las variables, ayuda, bitácora de comandos y la salida HTML.

Igualmente ofrece componentes que ayudan en la edición de código y ejecución directa de órdenes, como la ventana de guiones y la consola de R, donde se pueden introducir comandos o programas completos como se haría en la interfaz de texto original de R, con ayudas adicionales como coloreado de sintaxis documentación de funciones mientras se escribe, y con la característica de captura de gráficas o diálogos emergentes producidos ofreciendo opciones adicionales de manipulación, guardado y exportación de estos.

## Otros

Existen más entornos gráficos iguales de útiles que los anteriores, cada uno con sus ventajas e inconvenientes. Una lista de ellos es:

- JGR o Java GUI for R, una terminal de R multiplataforma basada en Java
- RExcel, que permite usar R y Rcmdr desde Microsoft Excel
- rggobi
- Sage
- Statistical Lab
- nexusBPM
- RWorkBench



# Referencias

1. Aqueronte; <http://unbarquero.blogspot.com/>
2. Arriaza A. J., Fernández F. y col. (2008), Estadística Básica con R y R-Commander, Servicio de Publicaciones de la Universidad de Cádiz
3. Carmona F., Ejercicios de Análisis de la Varianza con R
4. Crawley M.J.(2007), The R Book, Hardcover
5. González A. y González S.(2000), Introducción a R: Notas sobre R: Un entorno de programación para Análisis de Datos y Gráficos, R Development Core Team
6. Paradis E.(2002), R para Principiantes, Institut des Sciences de l'Évolution Universit Montpellier II
7. Quick-R; <http://www.statmethods.net/>
8. R bloggers; <http://www.r-bloggers.com/>