Instituto de Salud Carlos III

Introducción a 

5. Gráficos y Funciones

Instituto de Salud Carlos III

Indice

- Gráficos
- Funciones

Instituto de Salud Carlos III

Graficos

- Introducción
- plot ()
- Múltiples gráficos por ventana
- Identificación datos
- Datos multivariantes
- Boxplots
- Histogramas
- Jittering
- Adición rectas regresión
- Otros gráficos
- Guardando gráficos
- Otras cosas

Introducción

- R incluye *muchas y variadas* funciones para hacer gráficos.
- El sistema permite desde simples plots a figuras de calidad para incluir en artículos y libros.
- Sólo examinaremos la superficie.
- Más detalles:

  -Capítulo 4 de "*Modern applied statistics with S*"

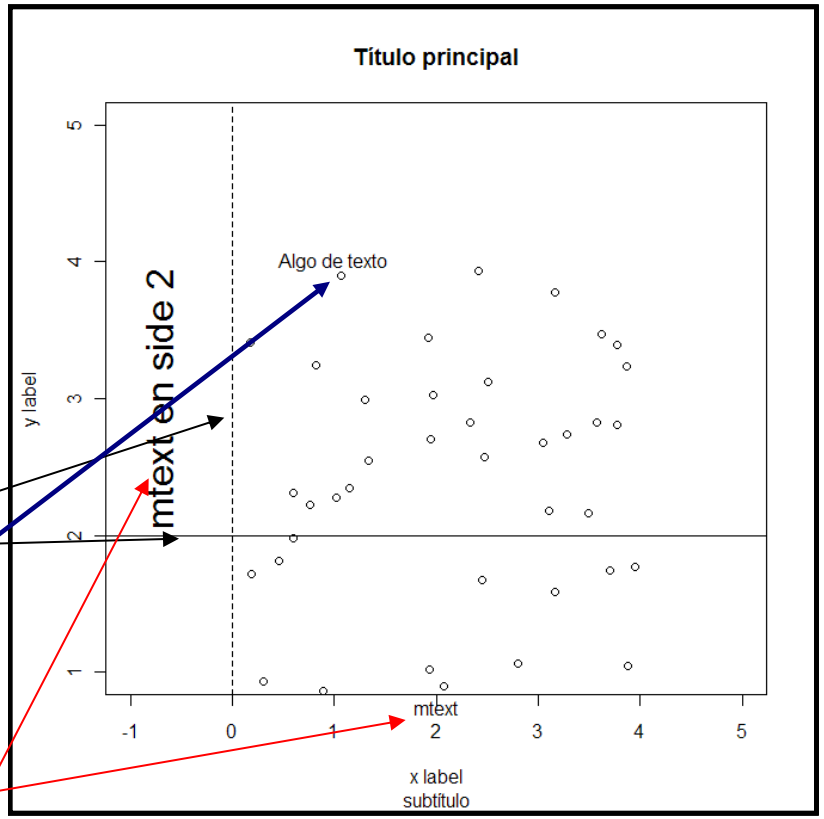
  -Capítulos 3 y 7 de "*An R and S-PLUS companion to applied regression*".

  -Capítulo 4 de "*R para principiantes*"; el capítulo 12 de "*An introduction to R*".
- También **demo(graphics)**.

plot()

- ***plot() función gráfica básica.***

```
x <- runif(50, 0, 4)
y <- runif(50, 0, 4)

plot(x, y, main = "Título principal",
sub = "subtítulo",
xlab = "x label",
ylab = "y label", xlim = c(-1, 5),
ylim = c(1, 5))

abline(h = 2, lty = 1)
abline(v = 0, lty = 2)

text(1, 4, "Algo de texto")

mtext("mtext", side = 1)
mtext("mtext en side 2", side = 2,
line = -3, cex = 2)
```



plot()

- ***Argumentos usuales de plot():***

  A) Especificar labels:

  -**main** – títuloprovides a title
  -**xlab** – label para el eje x
  -**ylab** – label para el eje y

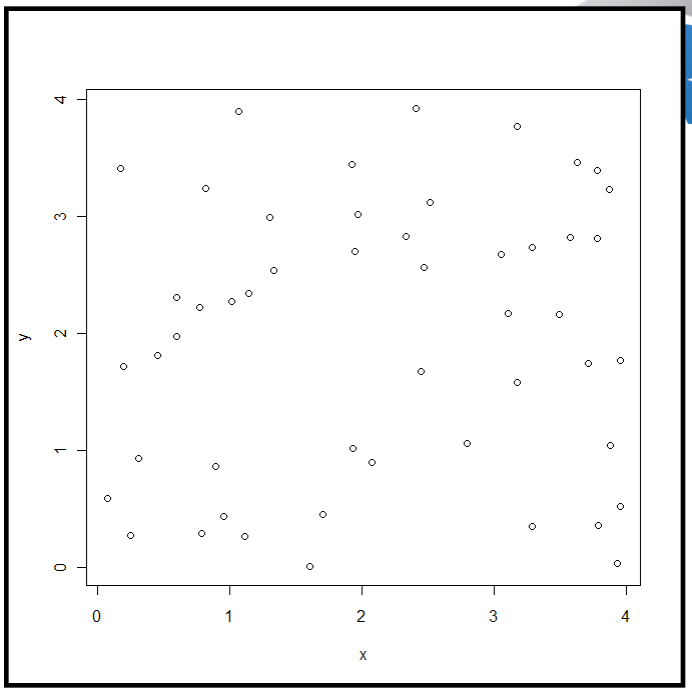
  B) Límites de los ejes

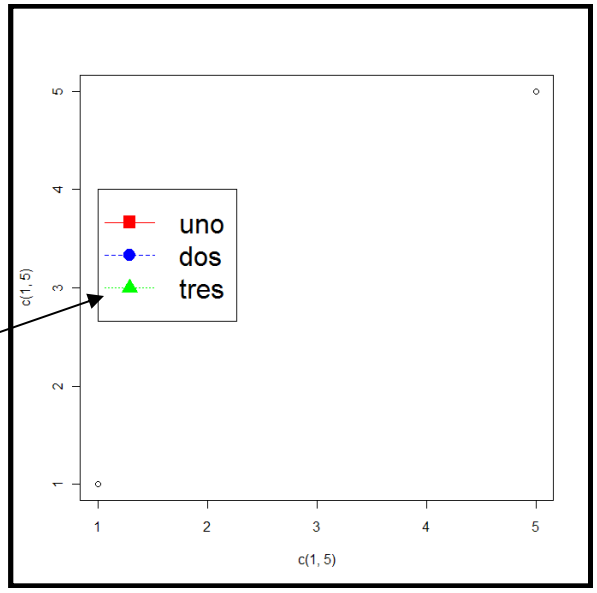
  -**ylim** – range del eje x
  -**xlim** – range del eje y

plot()

- ***Variaciones de plot(x)***

```
z <- cbind(x,y)

plot(z)
plot(y ~ x)

plot(log(y + 1) ~ x) # transformacion de y

plot(x, y, type = "p")
plot(x, y, type = "l")
plot(x, y, type = "b")

plot(c(1,5), c(1,5))

legend(1, 4, c("uno", "dos", "tres"), lty = 1:3,
col = c("red", "blue", "green"),
pch = 15:17, cex = 2)
```

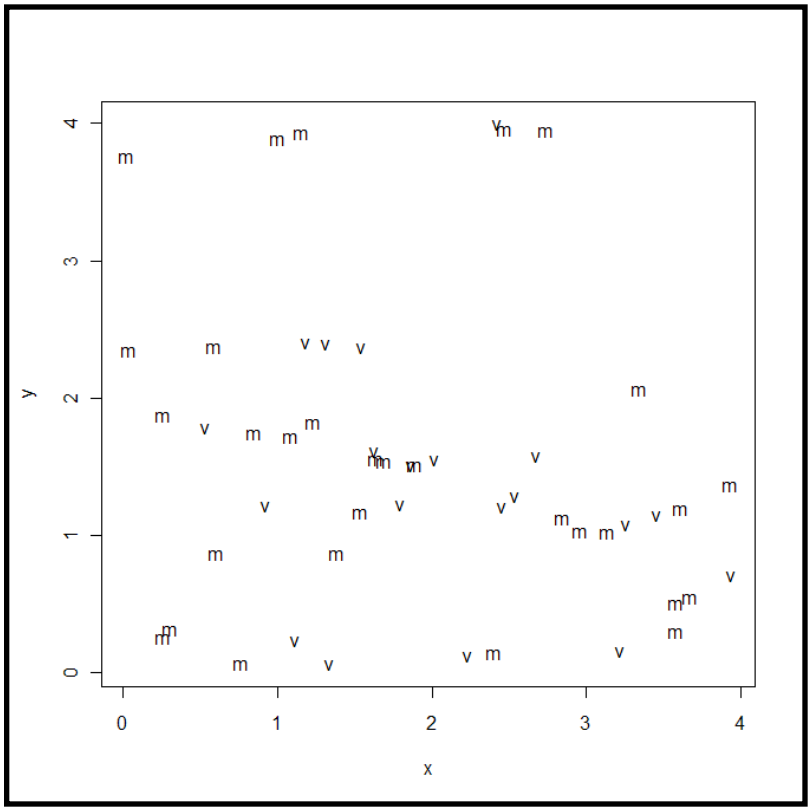




plot()

- Con **text** podemos representar caracteres de texto directamente:

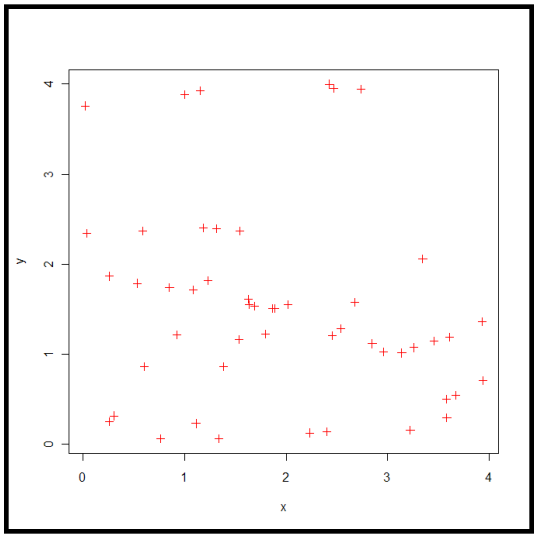
```
x <- runif(50, 0, 4)
y <- runif(50, 0, 4)
sexo <- c(rep("v", 20), rep("m", 30))

plot(x, y, type = "n")
text(x, y, labels = sexo)
```



plot()

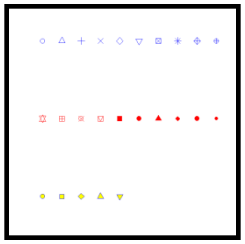
- ***Plot: pch, col***

```
plot(x, y, type = "n")
points(x, y, pch = 3, col = "red")

plot(x, y,pch=3,col="red")
```



- **Tipos de puntos:**

```
plot(c(1, 10), c(1, 3), type = "n", axes = FALSE, xlab = "", ylab="")
points(1:10, rep(3, 10), pch = 1:10, cex = 2, col = "blue")
points(1:10, rep(2, 10), pch = 11:20, cex = 2,col = "red")
points(1:10, rep(1, 10), pch = 21:30, cex = 2,col = "blue", bg = "yellow")
```

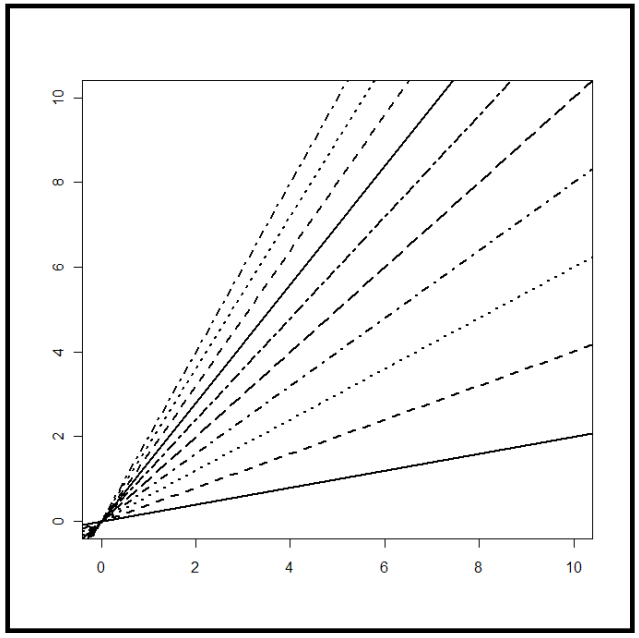


plot()

- ***Tipos de líneas:***

plot(x=c(0, 10),y=c(0, 10), type = "n", xlab ="", ylab ="")
Tambien➔plot(c(0, 10), c(0, 10), type = "n", ann=FALSE)

for(i in 1:10){
abline(0, i/5, lty = i, lwd = 2)
}



- **lty** permite especificaciones más complejas (longitud de los segmentos que son alternativamente dibujados y no dibujados).
- **lwd** permite aumentar/disminuir el grosor de la línea.
- **par** controla muchos parámetros gráficos. Por ejemplo, **cex** puede referirse a los "labels" (**cex.lab**), otro, **cex.axis**, a la anotación de los ejes, etc.
- Hay muchos más colores. Ver **palette**, **colors**.

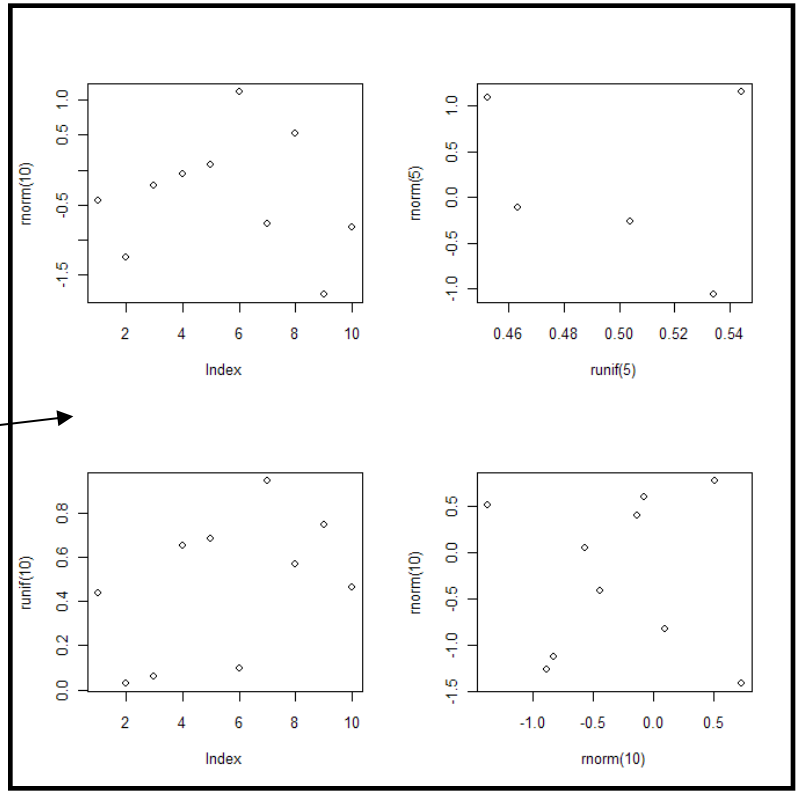
Múltiples gráficos por ventana

- Podemos mostrar muchos gráficos en el mismo dispositivo gráfico. La función más flexible y sofisticada es **split.screen**, bien explicada en *R para principiantes*, secc. 4.1.2 (p. 30).

- Mucho más sencillo es

  par(mfrow=c(filas, columnas))

```
par(mfrow = c(2, 2))
plot(rnorm(10))
plot(runif(5), rnorm(5))
plot(runif(10))
plot(rnorm(10), rnorm(10))
```

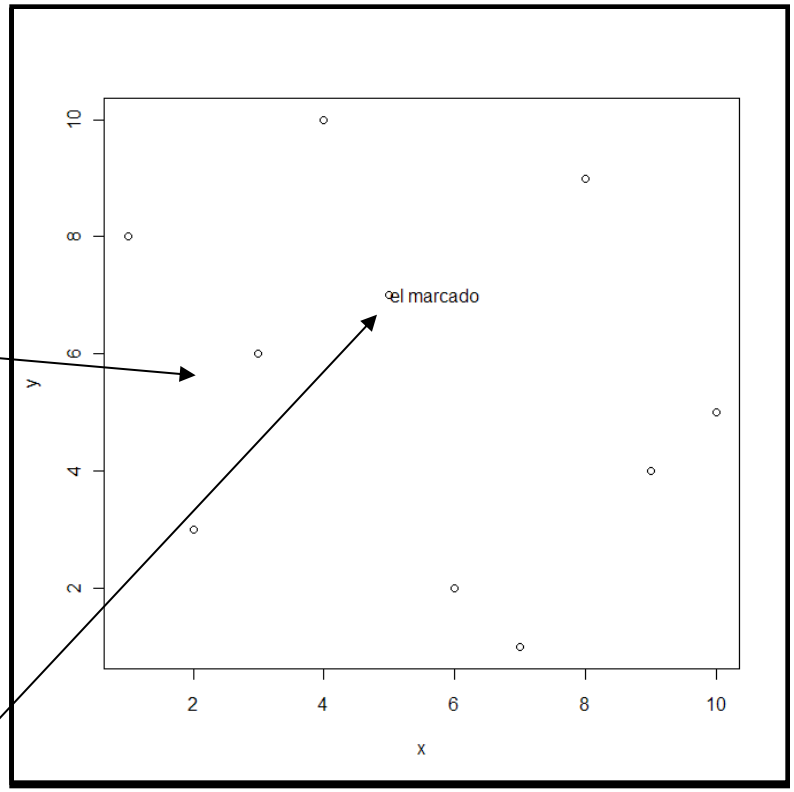


Identificación interactiva de datos

```
> x <- 1:10
> y <- sample(1:10)
>nombres <- paste("punto", x, ".",
 y, sep ="")
> plot(x, y)
> identify(x, y, labels = nombres)
```

-**locator** devuelve la posición de los puntos.

```
> plot(x, y)
# locator()
> text(locator(1), "el marcado", adj = 0)
```



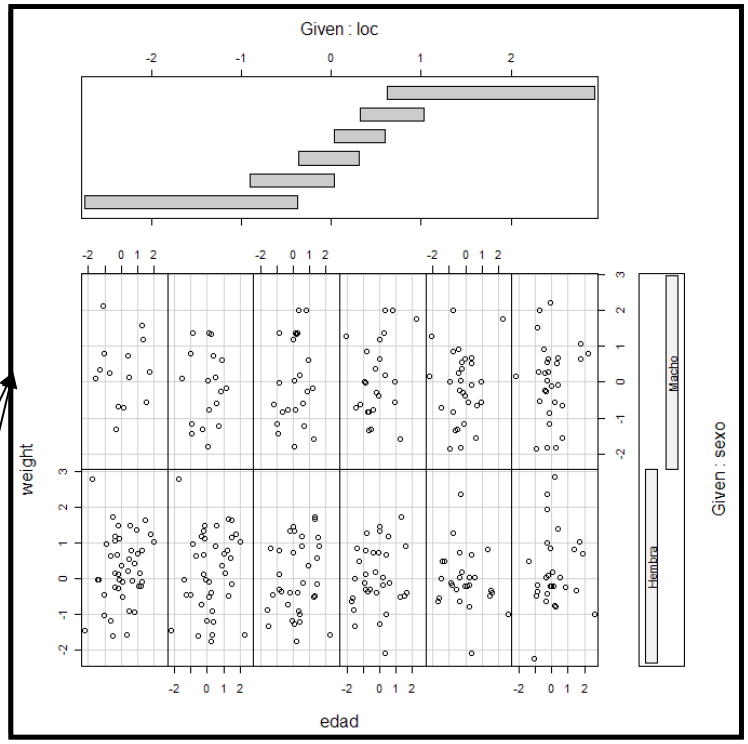
Datos multivariantes

- ***Una "pairwise scatterplot matrix":***

```
> X <- matrix(rnorm(1000), ncol = 5)
> colnames(X) <- c("a", "id", "edad", "loc", "weight")
> pairs(X)
```

- ***"Conditioning plots" (revelan, entre otros, interacciones):***

```
> Y <- as.data.frame(X)
> Y$sexo <- as.factor(c(rep("Macho", 80), rep("Hembra", 120)))
> coplot(weight ~ edad | sexo, data = Y)
> coplot(weight ~ edad | loc, data = Y)
> coplot(weight ~ edad | loc + sexo, data = Y)
```

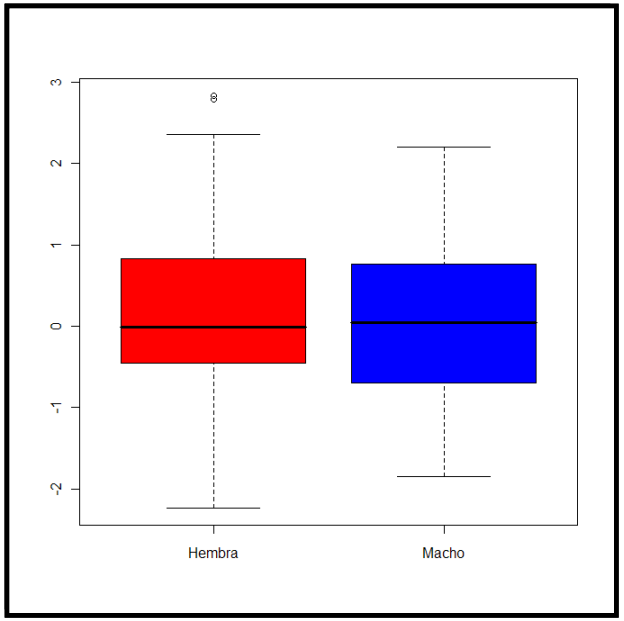


- La librería **lattice** permite lo mismo, y mucho más, que coplot. Ver secc. 4.6 de *R para principiantes*.

Boxplots (I)

- Muy útiles para ver rápidamente las características de una variable, o comparar entre variables.

```
Y <- as.data.frame(X)
Y$sexo <- as.factor(c(rep("Macho", 80),
rep("Hembra", 120)))

attach(Y)
boxplot(weight)
plot(sexo, weight)
detach()
boxplot(weight ~ sexo, data = Y,
col = c("red", "blue"))
```



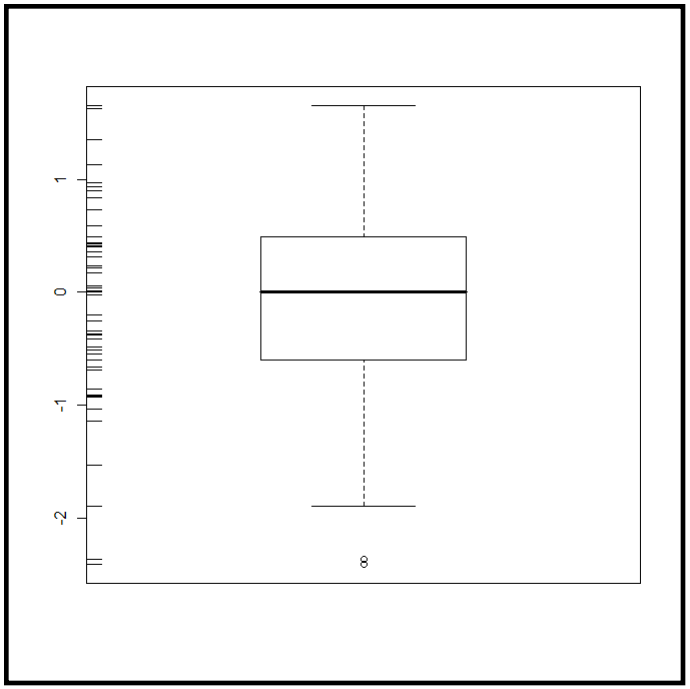
- **boxplot** tiene muchas opciones; se puede modificar el aspecto, mostrarlos horizontalmente, en una matriz de boxplots, etc. Vease la ayuda (?boxplot).

Boxplots (II)

- **rug()** para añadir cada observación a un lado del boxplot u otros gráficos.

- Dentro de rug(…, **side**=?) especifica donde quieres que sean dibujados

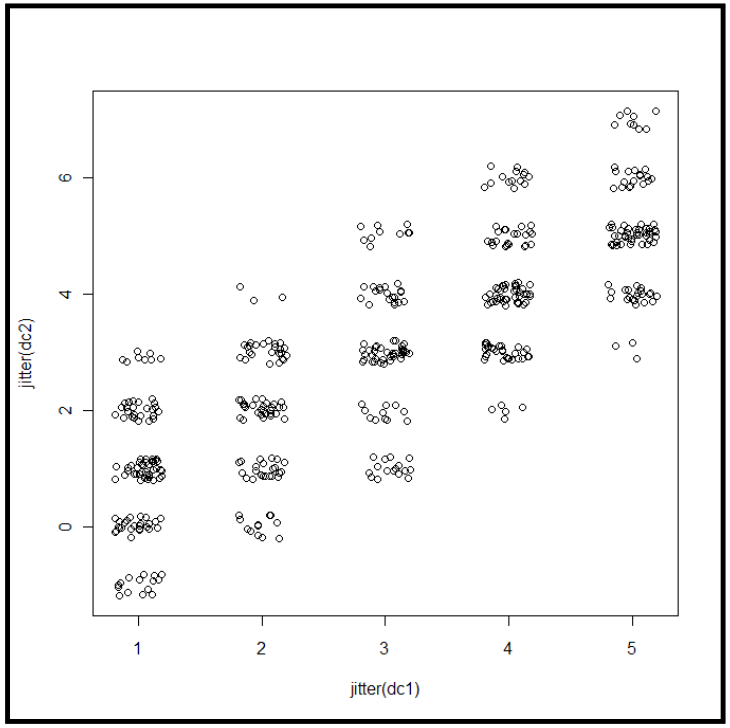
```
> x<-rnorm(50)
> boxplot(x)
> rug(x, side=2)
```



Jittering en scatterplots

- Los datos cuantitativos discretos pueden ser difíciles de ver bien:

```
dc1 <- sample(1:5, 500, replace = TRUE)

dc2 <- dc1 + sample(-2:2, 500,
replace = TRUE,
prob = c(1, 2, 3, 2, 1)/9)

plot(dc1, dc2)

plot(jitter(dc1), jitter(dc2))
```

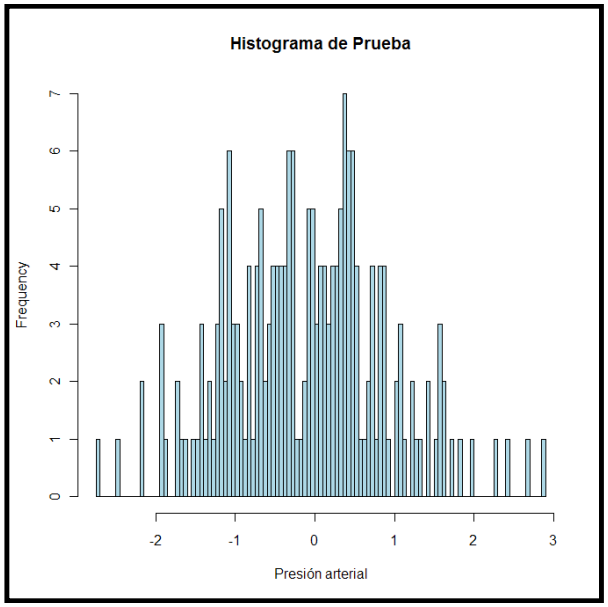


- También útil si sólo una de las variables está en pocas categorías.

Histogramas

- Función **hist()**
- Parámetro **breaks** especifica el número de categorías o los puntos de corte para cada categoría.
- El resto de las opciones como xlab, ylab, xlim y ylim funcionan como siempre.

```
hist(rnorm(200),col="lightblue",
xlab="Presión arterial",
main="Histograma de Prueba")

hist(rnorm(200),col="lightblue",
xlab="Presión arterial",
main="Histograma de Prueba",
breaks=100)
```

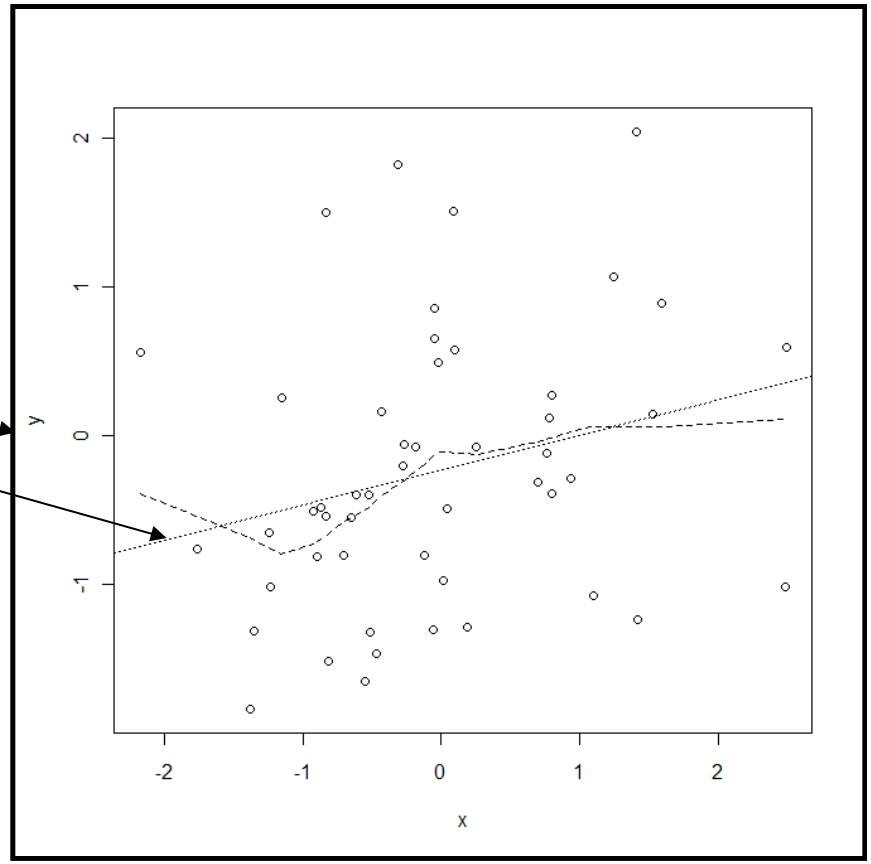


Adición rectas de regresión

- Podemos añadir muchos elementos a un gráfico, además de leyendas y líneas rectas:

```
> x <- rnorm(50)
> y <- rnorm(50)
> plot(x, y)
> lines(lowess(x, y), lty = 2)
> abline(lm(y ~ x), lty = 3)
```

LOWESS smoother (polynomial regression)



- Podemos añadir otros elementos con "panel functions" en otras funciones (como pairs, lattice, etc).

Otros

- Podemos modificar márgenes exteriores de figuras y entre figuras (vease ?par y búsquense oma, omi, mar, mai; ejemplos en *An introduction to R*, secc. 12.5.3 y 12.5.4.

- También **gráficos 3D**: persp, image, contour; **histogramas**: hist; **gráficos de barras**: barplot; **gráficos de comparación de cuantiles**, usados para **comparar la distribución** de dos variables, o la disribución de unos datos frente a un estándard (ej., distribución normal): qqplot, qqnorm y, en paquete "car", qq.plot.

- Notación matemática (plotmath) y expresiones de texto arbitrariamente complejas.

- Gráficos tridimensionales dinámicos con XGobi y GGobi. (Ver http://cran.r-project.org/src/contrib/PACKAGES.html#xgobi,http://www.ggobi.org, http://www.stat.auckland.ac.nz/~kwan022/pub/gguide.pdf).

Guardando Gráficos

- En Windows, podemos usar los menús y guardar con distintos formatos.

- Tambien podemos especificar donde queremos guardar el gráfico:

```
> pdf(file = "f1.pdf", width = 8, height = 10)
> plot(rnorm(10))
> dev.off()
```

- O bien, podemos copiar una figura a un fichero: **>**

```
> plot(runif(50))
> dev.copy2eps()
```

Otras cosas
Dibujar texto

- Dos funciones ya vistas:

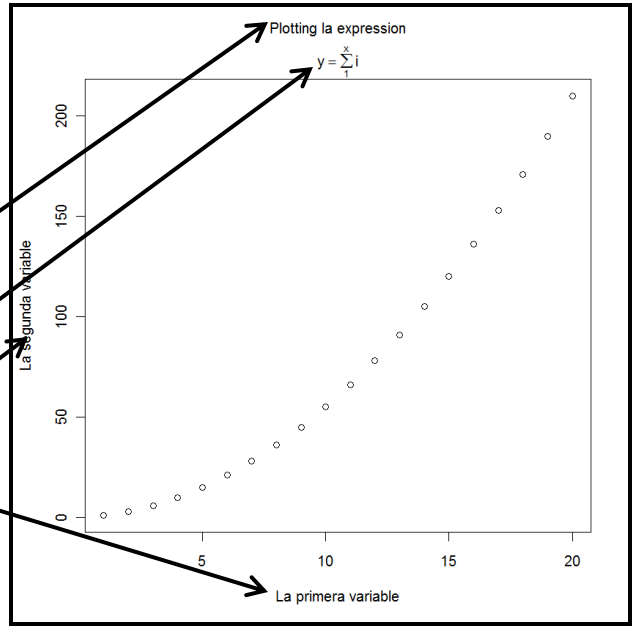
  -**text**() – escribe dentro de la región del plot y puede ser utilizada para poner lables a los datos.

  -**mtext**() – escribe en los márgenes del plot y puede ser utilizado para añadir múltiples leyendas.

- Estas dos funciones pueden imprimir expresiones matemáticas creadas por **expression**()

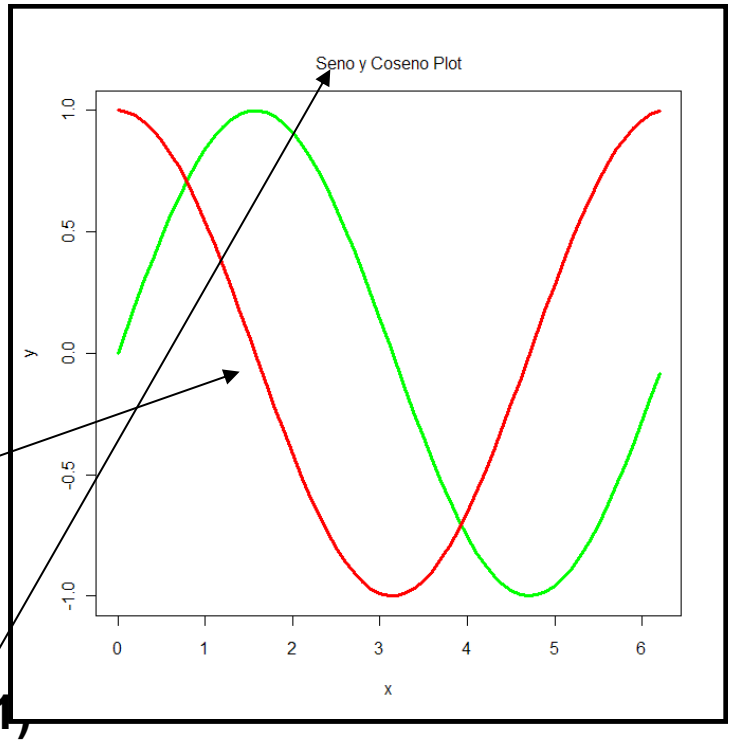
Otras cosas
ej. Márgenes con expresiones

```
f<-function(x) x*(x+1)/2
x<-1:20
y<-f(x)
plot(x,y,xlab="",ylab="")
mtext("Plotting la expression",side=3,line=2.5)
mtext(expression(y==sum(i,1,x,i)),side=3,line=0)
mtext("La primera variable",side=1,line=3)
mtext("La segunda variable",side=2,line=3)
```



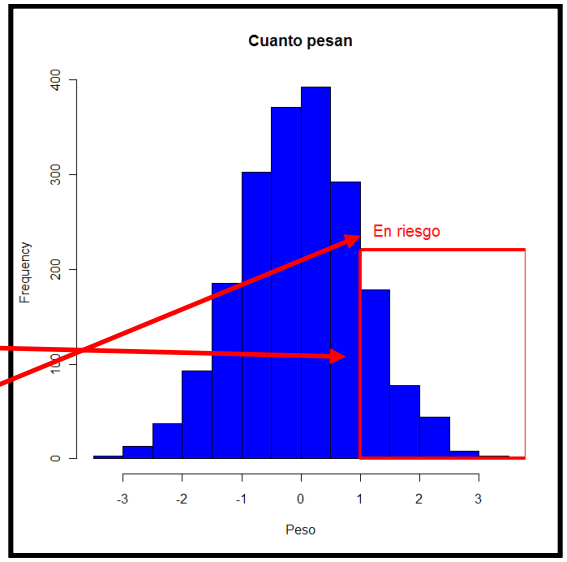
Otras cosas
Series temporales

```
x<-seq(0,2*pi,by=0.1)

y<-sin(x)

y1<-cos(x)

plot(x,y,col="green",type="l", lwd=3)

lines(x,y1,col="red",lwd=3)

mtext("Seno y Coseno Plot",side=3,line=1)
```



Otras cosas
Añadir labels dentro del plot

```
x<-rnorm(2000)

hist(x,xlab="Peso",
main="Cuanto pesan",col="blue")

rect(1,0,3.81,220,border="red",lwd=4)

text(1.8,240,"En riesgo",col="red",cex=1.25)
```



Otras cosas
Símbolos matemáticos

Big Operators	
sum(x[i], i = 1, n)	$\sum_{1}^{n} x_i$
prod(plain(P)(X == x), x)	$\prod_{x} P(X = x)$
integral(f(x) * dx, a, b)	$\int_{a}^{b} f(x)dx$
union(A[i], i == 1, n)	$\bigcup_{i=1}^{n} A_i$
intersect(A[i], i == 1, n)	$\bigcap_{i=1}^{n} A_i$
lim(f(x), x %->% 0)	$\lim_{x \to 0} f(x)$
min(g(x), x >= 0)	$\min_{x \geq 0} g(x)$
inf(S)	$\inf S$
sup(S)	$\sup S$

Instituto de Salud Carlos III

Funciones

Plan de Formación Interna

"Nuevas Tecnologías"

Instituto de Salud Carlos III

Índice

- Definición de funciones
- Argumentos
- Control de ejecución: condicionales, loops
- Cuando algo va mal: traceback, browser, debug
- Eficiencia: unix.time, Rprof
- Cosas variadas

Definición de funciones

- Ya hemos definido varias funciones. Aquí una más:

```
my.f2 <- function(x, y) {
        z <- rnorm(10)
        y2 <- z * y
        y3 <- z * y * x
        return(y3 + 25)
        }
```

- Lo que una función devuelve puede ser:
  - un simple número
  - vector
  - Una gráfica
  - Una lista
  - Un mensaje
  - … o todo.

Argumentos (I)

```
otra.f <- function (a, b, c = 4, d = FALSE) {
        x1 <- a * z ...}
```

- Los argumentos "a" y "b" tienen que darse en el orden debido o, si los nombramos, podemos darlos en cualquier orden:

```
otra.f(4, 5)
otra.f(b = 5, a = 4)
```

- Pero los argumentos con nombre siempre se tienen que dar después de los posicionales

```
otra.f(c = 25, 4, 5) # error
```

- Los argumentos **"c"** y **"d"** tienen "**default values**". Podemos especificarlos nosotros, o no especificarlos (i.e., usar los valores por defecto).

- **args**(nombre.funcion) nos muestra los argumentos (de cualquier función).

Argumentos (II)

- **"z"** es una "**free variable**": ¿cómo se especifica su valor?

- **Lexical scoping**. Ver documento *Frames, environments,and scope in R and S-PLUS* de J. Fox (en http://cran.r-project.org/doc/contrib/Fox-Companion/appendix.html) y sección 10.7 en *An introduction to R*. También **demo**(scoping).

- **". . . "** permite pasar argumentos a otra función:

```
> f3 <- function(x, y, label = "la x", ...) {
plot(x, y, xlab = label, ...)
}

> f3(1:5, 1:5)
> f3(1:5, 1:5, col = "red")
```

Control de ejecución if (I)

- **if** (*condicion.logica*) *instruccion* donde "*instruccion*" es cualquier expresión válida (incluida una entre { }).

- **if** (*condicion.logica*) *instruccion* **else** *instruccion.alternativa*.

```
> f4 <- function(x) {
        if(x > 5) print("x > 5")
        else {
        y <- runif(1)
        print(paste("y is ", y))
        }
        }
```

Control de ejecución if (II)

- **ifelse** es una versión vectorizada:

```
> odd.even <- function(x) {
        ifelse(x %% 2 == 1, "Odd", "Even")
        }

> mtf <- matrix(c(TRUE, FALSE, TRUE, TRUE),
nrow = 2)
>ifelse(mtf, 0, 1)
```

Control de ejecución: while, for

- **while** (*condicion.logica*) *instrucción*

- **for** (*variable.loop* in *valores*) *instrucción*

```
for(i in 1:10) cat("el valor de i es", i, "nn")
continue.loop <- TRUE
x <- 0
while(continue.loop) {
x <- x + 1
print(x)
if( x > 10) continue.loop <- FALSE
}
```

- **repeat**, **switch** también están disponibles.
- **break**, para salir de un loop.

Cuando algo va mal (I)

- Cuando se produce un **error**, **traceback**() nos informa de la secuencia de llamadas antes del crash de nuestra función (útil cuando se produce mensajes de error incomprensibles).

- Cuando se producen **errores** o la función da **resultados incorrectos o warnings indebidos** podemos seguir la ejecución de la función.

- **browser** interrumpe la ejecución a partir de ese punto y permite seguir la ejecución o examinar el entorno; con "n" paso a paso, si otra tecla sigue ejecución normal. "Q" para salir.

- **debug** es como poner un "broswer" al principio de la función, y se ejecuta la función paso a paso. Se sale con "Q".

```
> debug(my.buggy.function)

> undebug(my.buggy.function)
```

Cuando algo va mal (II)

- **Ejemplo:**

```
my.f2 <- function(x, y) {

                z <- rnorm(10)
                y2 <- z * y
                y3 <- z * y * x
                return(y3 + 25)

        }

my.f2(runif(3), 1:4)
debug(my.f2)
my.f2(runif(3), 1:4)
undebug(my.f2)
# insertar un browser() y correr de nuevo
```

Eficiencia

- Nuestro objetivo aquí no es producir las funciones más eficientes, sino funciones que hagan lo que deben.
- Pero a veces útil saber cuanto dura la ejecución de una función: **unix.time(my.f2(runif(10000), rnorm(1000)))**.
- **Rprof**: un profiler para ver cuantas veces es llamada cada función y cuanto tiempo se usa en esa función.
- Se puede ver el status con **gc**(), que sirve también para despejar las cosas después de operaciones con manejo de grandes objetos.

Cosas variadas (I)

source y BATCH: Para la ejecución no interactiva de código.

- Con source abrimos una sesión de R y hacemos **>** **source("mi.fichero.con.codigo.R")**.
- Con **BATCH**: **% R CMD BATCH mi.fichero.con.codigo.R**.
- **source** es en ocasiones más útil porque informa inmediatamente de errores en el código.
- **BATCH** no informa, pero no requiere tener abierta una sesión (se puede correr en el background).
- Puede que necesitemos explícitos print statements o hacer **source**(my.file.R, echo = TRUE).
- **sink** es el inverso de source (manda todo a un fichero).

Cosas variadas (II)

- Se pueden crear paquetes, con nuestras funciones, que se comporten igual que los demás paquetes. Ver *Writing R extensions*.
- R puede llamar código compilado en C/C++ y FORTRAN. Ver .C, .Call, .Fortran.Lexical scoping importante en programación más avanzada.
- No hemos mencionado el "computing on the language" (ej., do.call, eval, etc).
- R es un verdadero "object-oriented language". Dos implementaciones, las S3 classes y las S4 classes.