Introducción a R.

Sesión 2: Elementos de programación con R

Jorge de la Vega Góngora

Instituto Tecnológico Autónomo de México

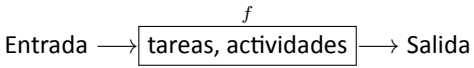
10 de septiembre de 2022

ITAM

Funciones en R

Uso de funciones en R

- En R, la definición de funciones es muy similar al concepto que se utiliza en matemáticas. En un sentido práctico, una función toma una entrada, a esa entrada le hace algo, y devuelve una salida:



- Una de las fortalezas de R es su enfoque funcional; prácticamente todos los objetos son modificados por funciones.
- Se pueden crear funciones propias y usualmente se interactúa con el software principalmente a través de funciones.
- Las funciones se pueden agrupar en varios tipos:
  - funciones para transformar datos: matemáticas y estadísticas
  - funciones para leer, importar, manipular y exportar datos
  - funciones para graficar datos

Funciones de usuario I

- Las entradas de las funciones, se especifican a través de argumentos.
- La estructura básica de una función en R es:

```
        nombre.de.mi.funcion <- function(argumentos){
#poner aquí todos los pasos a seguir
}
```

- Las funciones pueden devolver como salida números, vectores, matrices, dataframes, listas, mensajes o gráficas.

Ejercicio

Crear una función, que tome un vector `x` y devuelva la raíz cuadrada de la suma de cada uno de sus componentes al cuadrado. Llama a la función `norma`.

Algunas propiedades de las funciones. I

- Si se llama al nombre de una función sin los paréntesis, en muchos casos se puede ver el código de la función. Cuando no se puede ver, es porque la función está *compilada*.
- Una función puede ser terminada usando los comandos `return`, `stop` o bien mandando un mensaje con el comando `warning`. En este último caso, la función continúa su evaluación.
- Una función se puede definir dentro de otra función. Si se define la función `f2` dentro de la función `f1`, entonces:
  - Si llamo a `f2` dentro de `f1`, se usará en forma anidada, y
  - La función `f2` no será visible fuera de `f1`

```
gx <- function(x){
                y2 <- function(y){ rnorm(1, mean = y, sd = y^2)}
        x^2 + y2(x+3)
        }
gx(10)
```

```
[1] -72.1789
```

```
y2(5)
```

```
Error in y2(5): no se pudo encontrar la función "y2"
```

Argumentos de funciones I

- Un tipo de argumento especial de las funciones son los tres puntos "...". Usualmente se usa para pasar posibles argumentos de una función a otra sin ponerlos explícitos
- También se usan cuando hay un número variable de argumentos.
- Las funciones pueden tener sus argumentos *especificados* o *no especificados* (...). Por ejemplo:

```
mi.funcion <- function(x, y = 1, ...){
********
}
```

En este ejemplo, los argumentos son x: que el usuario tiene que especificar, y: que en caso de que el usuario no especifique tomará el valor de 1, y los tres puntos indican que puede haber más parámetros no especificados.

- Los argumentos *formales* son los que se usan en la definición de la función con un nombre, y los *reales* son los que se usan en una llamada a la función. Los argumentos reales son un subconjunto de los formales.

```
mi.funcion(4, color = T)
```

Argumentos de funciones II

- Para conocer los argumentos de una función f usamos args(f)

```
args(sum)
```

```
function (..., na.rm = FALSE)
NULL
```

```
args(sqrt)
```

```
function (x)
NULL
```

Reglas de argumentos I

Hay una serie de reglas que definen cómo se deben aparejar los argumentos *formales* y los *reales*.

1. Los argumentos reales que se dan de la forma `nombre = valor` donde el nombre es *exactamente* el nombre de un argumento formal, son apareados primero. Si el argumento formal aparece después de . . ., ésta es la única forma en que será apareado.

```
args(seq.int)
function (from, to, by, length.out, along.with, ...)
NULL
seq.int(from = 10, to = 20, by = 2)
[1] 10 12 14 16 18 20
seq.int(from = 10, to = 20, length.out = 5)
[1] 10.0 12.5 15.0 17.5 20.0
```

2. Si hay argumentos reales sin nombre, son apareados a los parámetros formales uno por uno en la sucesión de argumentos dados.

```
seq.int(10,20,2)
[1] 10 12 14 16 18 20
```

Reglas de argumentos II

3. Argumentos especificados de la forma `nombre = valor` para los que hay una correspondencia parcial con un argumento formal, son apareados.

```
seq.int(10,20,length.out=5)
[1] 10.0 12.5 15.0 17.5 20.0
```

4. Todos los restantes argumentos reales que no están apareados, formarán parte del argumento formal . . ., si hay uno, y si no se provee, entonces ocurre un error.

```
args(sqrt)
function (x)
NULL
sqrt(10,20) #noten que es diferente a:
Error in sqrt(10, 20): 2 arguments passed to 'sqrt' which requires 1
sqrt(c(10,20))
[1] 3.162278 4.472136
```

5. Tener argumentos formales no apareados NO es un error, como se vió en el caso se `seq.int`

Funciones especiales para manipulación de datos

Uso de las funciones apply, lapply, tapply y sapply I

- La función apply permite aplicar otras funciones sobre los renglones o las columnas de una matriz.

```
args(apply)             # vemos sus argumentos

function (X, MARGIN, FUN, ..., simplify = TRUE)
NULL

x <- runif(100)
dim(x) <- c(25,4)      # crea una matriz de datos artificiales
apply(x,2,mean)        # el MARGIN es 1 para renglones, 2 para columnas

[1] 0.5559515 0.4775516 0.4572061 0.4264314
```

- La función equivalente para listas es la función lapply, que aplica una función cada componente de una lista.

Uso de las funciones apply, lapply, tapply y sapply II

```
y <-list(NULL)  # Crea una lista con un elemento nulo. Para inicializar la lista
length(y) <- 10  # Incrementa la langitud de la lista
Z <- lapply(y, function(x){runif(10)}) # En cada componente de la lista, pon 10 uniformes.
Z[1:2]          # Los dos primeros elementos de la lista

[[1]]
 [1] 0.57810731 0.47007675 0.75351817 0.97703039 0.91615055 0.04390869 0.92724400 0.92133345 0.40642434 0.77517351

[[2]]
 [1] 0.8703252 0.5877177 0.2688901 0.2277290 0.9708786 0.7509139 0.6267808 0.8841218 0.5753789 0.9004451

u <- lapply(Z, max)  # Calcula el máximo en cada elemento de la lista
u[1:2]

[[1]]
[1] 0.9770304

[[2]]
[1] 0.9708786
```

- La función `tapply` aplica funciones sobre *grupos* de un dataframe o un array (identificados por una variable categórica); es equivalente a las tablas dinámicas en Excel. Como ejemplo, tomamos unos datos públicos correspondientes de la STPS, la serie estadística sobre la magnitud de la ocupación en el comercio, en comparación con la población ocupada total (trimestral).

Uso de las funciones apply, lapply, tapply y sapply III

```
datos <- read.csv(file = 'http://datosabiertos.stps.gob.mx/Datos/DIL/clave/Tasa_de_Ocupacion_en_el_Comercio.csv', nrow = 4224,
                  fileEncoding = "latin1") #necesario para que el archivo tenga la codificación de caractéres correcta
head(datos,3)

  Periodo Trimestre Entidad_Federativa    Sexo Población_ocupada_en_el_comercio Población_ocupada Tasa_neta_de_ocupación_en_el_comercio  X
1    2005         1           Nacional Hombres                          4155341          26597801                            15.62287424 NA
2    2005         1           Nacional Mujeres                          3986139          14843275                            26.85484841 NA
3    2005         1     Aguascalientes Hombres                            39813            246177                            16.17251002 NA
  X.1 X.2
1  NA  NA
2  NA  NA
3  NA  NA

datos$X <- NULL; datos$X.1 <- NULL; datos$X.2 <- NULL  # Quita las columnas que no tienen datos
# Vemos los tipos de datos
str(datos)

'data.frame': 4224 obs. of  7 variables:
 $ Periodo                              : int  2005 2005 2005 2005 2005 2005 2005 2005 2005 2005 ...
 $ Trimestre                            : int  1 1 1 1 1 1 1 1 1 1 ...
 $ Entidad_Federativa                   : chr  "Nacional" "Nacional" "Aguascalientes" "Aguascalientes" ...
 $ Sexo                                 : chr  "Hombres" "Mujeres" "Hombres" "Mujeres" ...
 $ Población_ocupada_en_el_comercio     : chr  "4155341" "3986139" "39813" "36001" ...
 $ Población_ocupada                    : chr  "26597801" "14843275" "246177" "145656" ...
 $ Tasa_neta_de_ocupación_en_el_comercio: chr  "15.62287424" "26.85484841" "16.17251002" "24.71645521" ...

class(datos$Población_ocupada_en_el_comercio) <- "numeric"

Warning in class(datos$Población_ocupada_en_el_comercio) <- "numeric": NAs introducidos por coerción
```

Uso de las funciones `apply`, `lapply`, `tapply` y `sapply` IV

```
# Queremos ver el total nacional por sexo:
tapply(datos$Población_ocupada_en_el_comercio, datos$Sexo, sum, na.rm=T)

  Hombres   Mujeres
582307911 610675397
```

- Ahora queremos ver lo mismo, pero además por año:

```
with(datos, tapply(Población_ocupada_en_el_comercio, list(Sexo, Periodo), sum))

            2005     2006     2007     2008     2009     2010     2011     2012     2013     2014     2015     2016     2017     2018
Hombres 33160198 33758446 34170082 34647482 35098740 36034552 36239884 37136498 37346310 37683084 37826984 37711120 37728228 39073824
Mujeres 32797844 34162968 35693658 36128838 37148866 36910360 38102690 39498168 39856596 39108978 39937694 40481042 39558678 41084596
            2019 2020
Hombres 40800546   NA
Mujeres 44607670   NA

# Incluyendo Entidad Federativa:
Arreglote <- with(datos, tapply(Población_ocupada_en_el_comercio, list(Entidad_Federativa, Sexo, Periodo), mean))
Arreglote["Tlaxcala",,]

            2005     2006     2007     2008     2009     2010     2011     2012     2013     2014     2015     2016     2017     2018  2019
Hombres 36674.75 38660.00 38324.00 37326.25 37953.75 37225.25 42016.25 41232.50 41590.25 44867.00 45670.75 45517.50 47890.25 48064.5 50443
Mujeres 37184.00 40056.75 42660.25 43704.50 44885.75 41025.50 43221.25 46509.75 50924.25 48168.25 48612.00 50800.75 54185.25 56631.0 60348
        2020
Hombres   NA
Mujeres   NA
```

Algunas observaciones sobre calidad de datos

- El valor `NA` significa Not Available, se usa para representar datos faltantes o no disponibles. La función `is.na` se utiliza para verificar si un vector tiene datos faltantes

```
x <- c(1,2,3,NA,4,10)
is.na(x)
```

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

- El símbolo `NaN` significa *Not a Number*, `Inf` denota el infinito ∞ y `-Inf` es menos infinito, $-\infty$. Algunas funciones pueden devolver este resultado.

```
Inf + 0
```

```
[1] Inf
```

```
Inf*(-Inf)
```

```
[1] -Inf
```

- `NULL` es el objeto vacío o nulo. Su longitud siempre es 0.

```
x <- NULL
length(x)
```

```
[1] 0
```

Últimos detalles sobre coerción I

- Los modos se pueden ordenar por la cantidad de información que contienen:

$$\texttt{"NULL"} < \texttt{"logical"} < \texttt{"numeric"} < \texttt{"complex"} < \texttt{"character"} < \texttt{"list"}$$

- En general, los modos de los objetos se pueden cambiar sin perder información. Con operaciones aritméticas o de comparación entre objetos de diferentes modos se realiza, los objetos se convierten al mismo modo de tal forma que no se pierda información. A esto se le llama *coerción*.
- A las funciones que tienen como argumentos caractéres, se les pueden dar datos de *cualquier* modo.
- Para cada modo, hay tres funciones:
  - para crear objetos de ese modo (`logical(10)`)
  - para probar si objetos son de ese modo (`is.logical`)
  - para forzar objetos a tal modo (`as.logical`)

Últimos detalles sobre coerción II

```
as.numeric("13")+ 12
[1] 25
as.character(13) + 12
Error in as.character(13) + 12: argumento no-numérico para operador binario
is.logical("TRUE")
[1] FALSE
is.logical(2)
[1] FALSE
is.logical(as.logical(2))
[1] TRUE
```

Otras funciones importantes I

- La función `attach` y `detach` permiten acceder a los elementos de un data frame como series independientes, pero no permite modificaciones a las series.

```
datos[1,"Sexo"]
```

```
[1] "Hombres"
```

```
attach(datos) # deja entrar a los datos
Sexo[1:10]
```

```
 [1] "Hombres" "Mujeres" "Hombres" "Mujeres" "Hombres" "Mujeres" "Hombres" "Mujeres" "Hombres" "Mujeres"
```

```
Sexo[1] <- "Mujeres"
Sexo[1:10] # cambia el valor de esta variable
```

```
 [1] "Mujeres" "Mujeres" "Hombres" "Mujeres" "Hombres" "Mujeres" "Hombres" "Mujeres" "Hombres" "Mujeres"
```

```
datos[1, "Sexo"]  # pero no el dataframe
```

```
[1] "Hombres"
```

```
detach(datos) # ya no es accesible, y sólo se mantiene una copia de las variables
Sexo[1:10]    # que se cambiaron
```

```
 [1] "Mujeres" "Mujeres" "Hombres" "Mujeres" "Hombres" "Mujeres" "Hombres" "Mujeres" "Hombres" "Mujeres"
```

Otras funciones importantes II

- `subset` toma un subconjunto de renglones de un dataframe.

```
A2010 <- subset(datos, datos$Periodo == "2010")
head(A2010)
```

```
     Periodo Trimestre Entidad_Federativa    Sexo Población_ocupada_en_el_comercio Población_ocupada Tasa_neta_de_
1321    2010         1           Nacional Hombres                           4478623          28421331
1322    2010         1           Nacional Mujeres                           4525262          17103008
1323    2010         1     Aguascalientes Hombres                             46881            274223
1324    2010         1     Aguascalientes Mujeres                             40376            173151
1325    2010         1    Baja California Hombres                            138286            820298
1326    2010         1    Baja California Mujeres                            115881            494120
```

- La función `merge` sirve para unir data frames en varias formas: una aplicación útil se utiliza para unir diferentes registros en bases de datos con un campo común.

Otras funciones importantes III

- La función `table` crea tablas de frecuencias (conteos) de variables categóricas

```
table(datos$Sexo)

Hombres Mujeres
   2112    2112
```

```
with(datos, table(Sexo, Periodo))
```

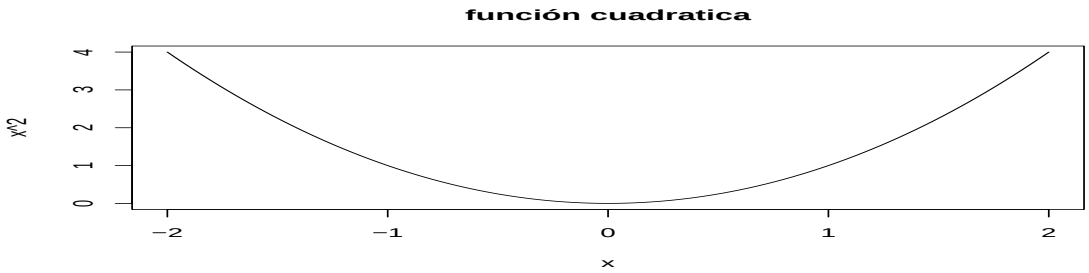
```
         Periodo
Sexo      2005 2006 2007 2008 2009 2010 2011 2012 2013 2014 2015 2016 2017 2018 2019 2020
  Hombres  132  132  132  132  132  132  132  132  132  132  132  132  132  132  132  132
  Mujeres  132  132  132  132  132  132  132  132  132  132  132  132  132  132  132  132
```

Curvas y funciones matemáticas

Usando R para hacer gráficas de funciones I

La función `curve` sirve para hacer gráficas de funciones matemáticas.

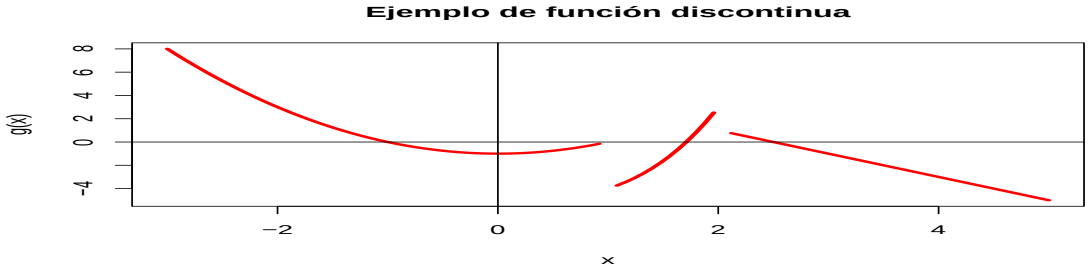
```
curve(x^2, from = -2, to = 2, main = "función cuadratica")
```



Podemos definir nuestras propias funciones, incluso con discontinuidades, como por ejemplo

Usando R para hacer gráficas de funciones II

```
g <- function(x){
                ifelse(
                        x < 1, x^2-1,
                        ifelse(
                                1 < x & x < 2, x^3-5,
                                ifelse(
                                        x>2.1, 5 - 2*x, NA)))
        }
curve(g(x), from = -3, to = 5, col = "red", lwd=3, main="Ejemplo de función discontinua")
abline(v = 0, h = 0)
```



Estructuras de Control

Control de ejecución: condicionales

Es en la creación de funciones de usuario, que se requiere un poco de conocimientos de programación. Estas se verán más adelante en ejemplos prácticos. Aquí se mencionan las principales estructuras.

- se puede usar: `if (cond)  else (cond)`
- Las condiciones pueden incluir operadores lógicos `&`, `|`.
- Otra posibilidad es la función `ifelse` sobre vectores.
- Para sustituir if's anidados, se usa la función `switch`.

Control de ejecución: ciclos

- Las funciones disponibles son `for, while, repeat`.
- `while (condicion.logica) instruccion`. Esta función termina cuando la condición es falsa.
- `for (variable.loop in valores) instruccion`
- `repeat (condicion) instruccion`.
- Para salir de los ciclos en cualquier punto, usamos `break`.
- Para saltar a la siguiente iteración, se usa `next`.

Gráficas

Introducción I

- Generar gráficas en R puede ser muy fácil o extremadamente complicado, ¡pero siempre es divertido!
- R produce el rango usual de gráficas estadísticas básicas, que incluye diagramas de dispersión, boxplots (gráficas de caja y brazo), histogramas, gráficas de pie, gráficas tridimensionales, animadas y muchas otras.
- El motor gráfico de R se concentra en algunos cuantos paquetes básicos (que están incluídos en la instalación original) y otros recomendados (que el usuario tiene que instalar, se marcan abajo con *):

  graphics contiene funciones de graficación para el sistema "base", incluye `plot, hist, boxplot` y muchas otras funciones. Una panorámica muy general de las gráficas puede verse con la instrucción `demo(graphics)`.

  * lattice contiene el código para generar gráficas Trellis (conocidas como gráficas condicionales), que son independientes del sistema base; incluye funciones como `xyplot, bwplot, levelplot`

  grid implementa un sistema de graficación independiente del sistema base. El paquete **lattice** está construido sobre **grid**. Rara vez las funciones de **grid** se llaman directamente.

Introducción II

grDevices contiene el código que implementa los diferentes dispositivos gráficos: X11, PDF, PostScript PNG, etc.

* ggplot2 Este es otro sistema basado en grid que interpreta y extiende las ideas de Leland Wilkinson de *The Grammar of Graphics*. Este es el paquete de moda, parte de `tidyverse`.

- La calidad de las gráficas de R permite que éstas estén listas para publicarse en artículos y libros.
- Se puede consultar la galería de gráficas de R para mayor detalle.

Decisiones iniciales

Cuando se hace una gráfica es importante tomar algunas decisiones iniciales, sobre todo porque cambios en la decisión pueden afectar la calidad de la imagen:

- ¿a qué dispositivo se enviará (impresora, pantalla)?
- ¿cuántos puntos habrá en la gráfica? (grandes cantidades, unos cuantos).
- ¿se requiere escalar la gráfica? Por ejemplo, si se usará en Word o PowerPoint.
- ¿Qué sistema de graficación se usará: base, grid/lattice, ggplot2? Usualmente no se pueden mezclar.
- Las gráficas del sistema Base se construyen con el principio del pintor: los objetos de la gráfica se agregan por capas.
- Las gráficas en el sistema ggplot2 también tienen capas o *layers*, pero son de conceptos.
- Las gráficas grid/lattice se crean de un sólo golpe: todos los parámetros se tienen que especificar de una sola vez en la función, y no se pueden agregar capas posteriores.

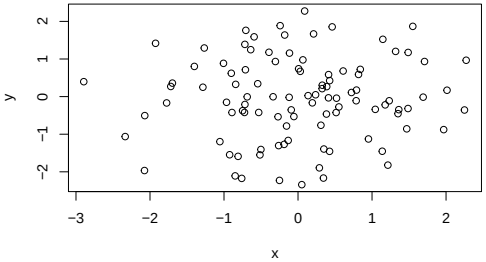
Gráficas del sistema base I

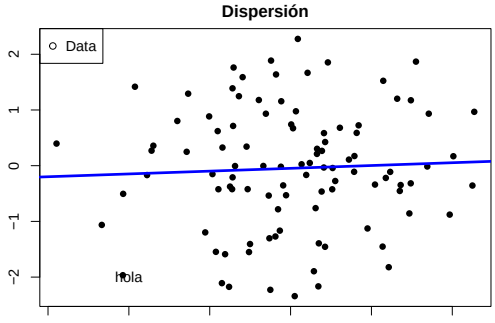
Es el sistema más fácil y el primero que se creó: las gráficas se pueden programar y armar en etapas, agregando diferentes componentes como se necesiten.

- Tiene muchos parámetros: están documentados en la función `par`
- Hay muchos acordeones para tener las funciones presentes.
- La función `par` se utiliza para especificar parámetros gráficos globales que afectan todas las gráficas en una sesión de R. Algunos ejemplos:
  - `pch`: **p**lotting **ch**aracter. Es el tipo de símbolo a usar en la gráfica.
  - `lty`: **l**ine **ty**pe (tipo de linea, continua, punteada, rayada, punto-raya, etc)
  - `lwd`: **l**ine **w**idth (ancho de línea, se especifica como un entero positivo)
  - `col`: **col**or, la función `colors()` da un vector de colores por nombre.
  - `las`: la orientación de las etiquetas en el eje x.
  - `bg`: el color del fondo de la gráfica.
  - `mar`: el tamaño del margen. Hay cuatro márgenes, en el órden: abajo, izquierda, arriba, derecha.
  - `oma`: el tamaño del margen exterior
  - `mfrow`: número de gráficas en un arreglo por renglón, columna (llenada por renglones)
  - `mfcol`: número de gráficas por renglón, columna (llenada por columna)

Algunos Ejemplos

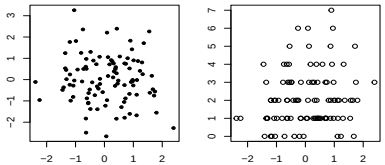
```
x <- rnorm(100); y <- rnorm(100)
plot(x,y) # función más básica
par("mar") # el vector de márgenes, empezando por bottom, left, top, right.

[1] 5.1 4.1 4.1 2.1

par(mar=c(2,2,2,2))
plot(x, y, pch = 16); title("Dispersión"); text(-2, -2, "hola"); legend("topleft", legend = "Data", pch = 1)
fit <- lm(y ~ x) #ajusta una linea recta a los datos
abline(fit, lwd = 3, col = "blue")
```





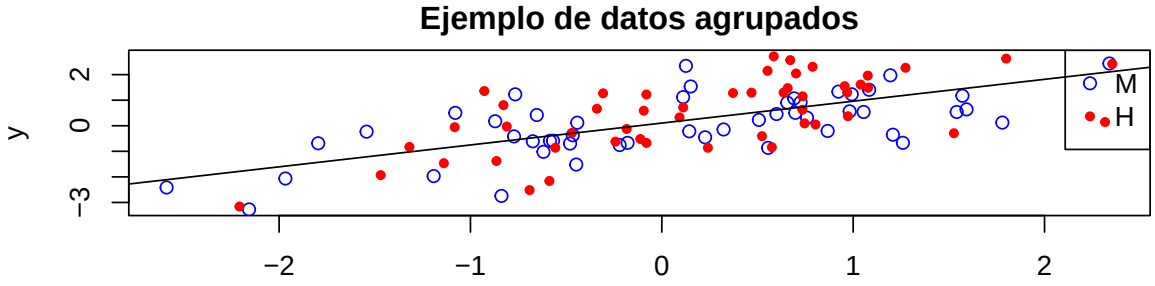
Ejemplos

```r
#genera dos gráficas en la misma hoja
x <- rnorm(100); y <- rnorm(100); z <- rpois(100,2) # genera datos artificiales
par(mar = c(2,2,2,2), oma = c(2,2,2,2))
par(mfcol = c(1,2)) # 1 renglón y dos columnas
plot(x, y, pch = 20)
plot(x, z, pch = 21)
```



Ejemplos

```r
par(mfrow = c(1,1), mar = c(2,4,2,0.1))
x <- rnorm(100); y <- x + rnorm(100)
g <- gl(2, 50, labels = c("H", "M")) #genera un factor con dos niveles
plot(x, y, type = "n", main = "Ejemplo de datos agrupados")
points(x[g == "M"], y[g == "M"], col = "blue")
points(x[g == "H"], y[g == "H"], col = "red", pch = 20)
legend("topright", legend = c("M", "H"), pch = c(1, 20), col = c("blue", "red"))
fit2 <- lm(y ~ x)
abline(fit2)
```



Funciones básicas del sistema base

`plot` hace una gráfica de dispersión, u otros tipos de gráficas dependiendo del tipo de objeto que se grafica.

`lines` agrega lineas a una gráfica ya existente, dado un vector de valores `x` y el correspondiente vector de valores `y`. La función sólo conecta las lineas.

`points` agrega puntos a una gráfica ya existente.

`text` agrega etiquetas de texto a una gráfica usando coordenadas específicas `x`, `y`

`title` agrega anotaciones a las etiquetas de los ejes `x` y `y`, título, subtítulo, margen exterior

`mtext` agrega texto arbitrario a los márgenes (interior y exterior) de la gráfica

`axis` agrega ticks/etiquetas a los ejes.

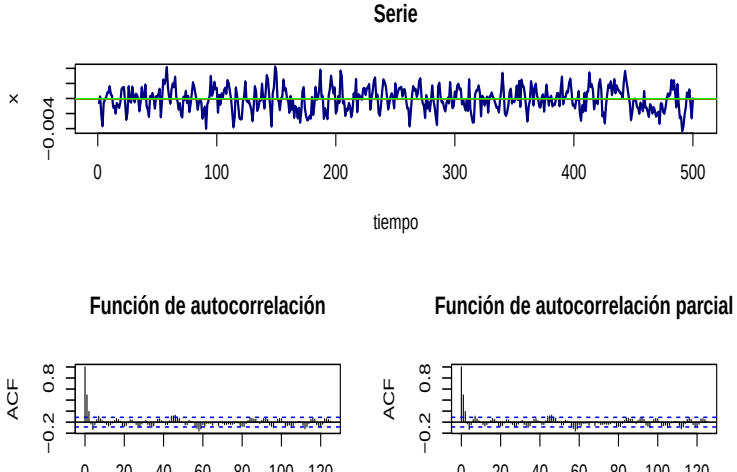
Ejemplos de una función que genera una gráfica

El siguiente código define una función que genera varias gráficas sobre una serie de tiempo

```
tsgraficas <- function(x, k = round(length(x)/4)){
                        oldpar <- par("oma")  # guarda los valores del parámetro oma
                        par("oma" = c(0.1,0.1,0.1,0.1))
                                              layout(matrix(c(1,1,2,3), nrow = 2, byrow = T)) #acomodo de las gráficas en el espacio
                                              plot.ts(x, lwd = 2, lty = 1, col = "navy",xlab = "tiempo", main="Serie")
                        abline(h = mean(x), col = "red")
                        abline(h = median(x), col = "green")
                        acf(x, main = "Función de autocorrelación", lag.max = k)
                        acf(x, main = "Función de autocorrelación parcial", lag.max = k)
                        par("oma" = oldpar)   # regresa los valores a su valor anterior
              }
```

Ejemplos de una función gráfica

```
x <- arima.sim(n=500,list(ar=c(0.8,-0.4),ma=c(-0.227,0.24)),sd=0.0013) #ejemplo de una serie de tiempo simulada
tsgraficas(x)
```



Algunas Funciones del paquete `lattice`

- El paquete `lattice` tiene la característica de hacer gráficas de datos por diferentes tipos de grupos.
- Hay un libro completo de como utilizar el paquete `latice`, el cual es un poco avanzado. Estará disponible en Dropbox para los interesados.
- Algunas de sus principales funciones son las siguientes:
  - `xyplot` Función principal para crear gráficas de dispersión
  - `bwplot` boxplots
  - `histogram`
  - `stripplot` como un boxplot pero con puntos reales
  - `dotplot` gráficas de puntos sobre 'cuerdas de violin'
  - `splom` matrices de gráficas de dispersión; como `pairs` en el sistema base
  - `levelplot, contourplot`: para gráficas tridimensionales de los datos.

Funciones lattice

Las funciones lattice tienen usualmente una fórmula como primer argumento, de la forma: `y ~ x | f * g`

- A la izquierda de la tilde está la variable `y` y a la derecha la `x`.
- Después de la | estan las *variables condicionantes* — son opcionales; el * indica una interacción
- El segundo argumento de la función es el dataframe o lista de la cual se toman las variables de la fórmula.

Ejemplos de lattice I

```r
library(lattice)    # carga la librería, hay que instalarla
data(environmental) # carga un conjunto de datos ejemplo
head(environmental) # muestra los primeros 6 renglones de datos
```

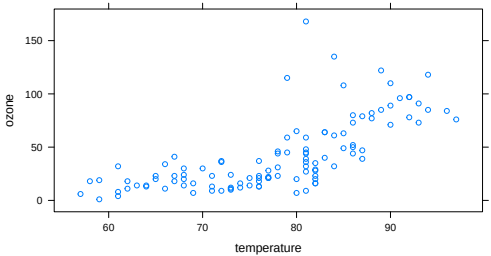
```
  ozone radiation temperature wind
1    41       190          67  7.4
2    36       118          72  8.0
3    12       149          74 12.6
4    18       313          62 11.5
5    23       299          65  8.6
6    19        99          59 13.8
```

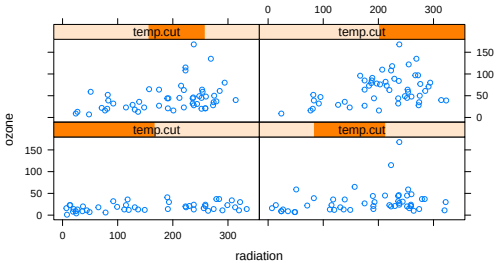
```r
temp.cut = equal.count(environmental$temperature,4) # vamos a discretizar la variable temperatura.
temp.cut
```

```
Data:
  [1] 67 72 74 62 65 59 61 69 66 68 58 64 66 57 68 62 59 73 61 61 67 81 79 76 82 90 87 82 77 72 65 73 76 84 85 81 83 83 88 92 92 89 73 81 80
 [46] 81 82 84 87 85 74 86 85 82 86 88 86 83 81 81 81 82 89 90 90 86 82 80 77 79 76 78 78 77 72 79 81 86 97 94 96 94 91 92 93 93 87 84 80 78
 [91] 75 73 81 76 77 71 71 78 67 76 68 82 64 71 81 69 63 70 75 76 68

Intervals:
   min  max count
1 56.5 76.5    46
2 67.5 81.5    51
3 75.5 86.5    51
4 80.5 97.5    51

Overlap between adjacent intervals:
[1] 27 30 31
```

Ejemplos de lattice

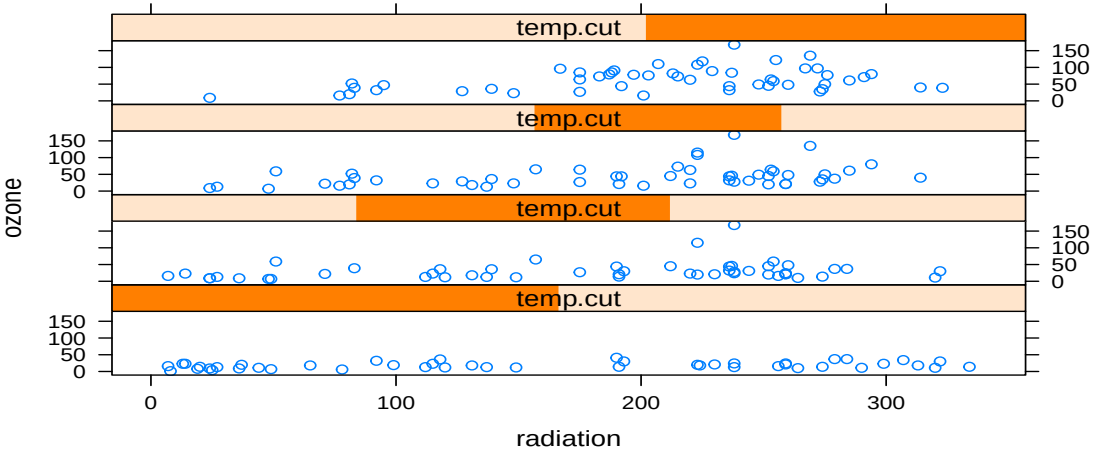
```
xyplot(ozone ~ temperature,data = environmental)
xyplot(ozone ~ radiation | temp.cut,data=environmental) # muestra la misma gráfica anterior para diferentes cortes de temperatura
```





Ejemplos de lattice

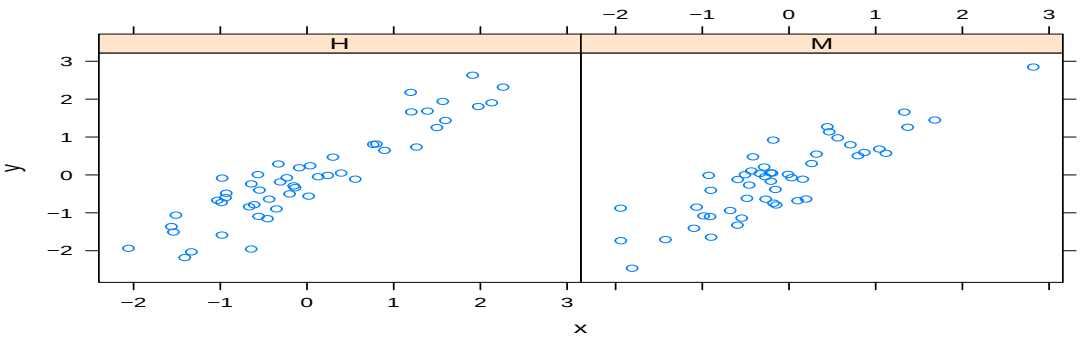
```
xyplot(ozone ~ radiation | temp.cut, data = environmental, layout = c(1,4)) # la graf anterior, pero con otra configuración
```



Funciones panel de lattice

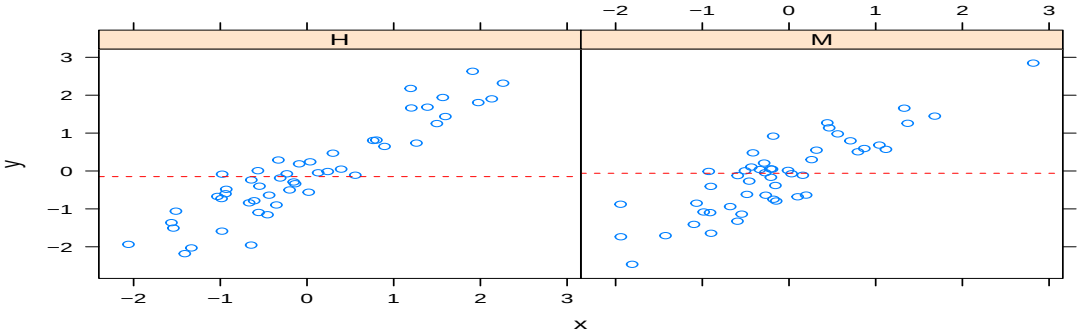
Las funciones lattice tienen una función `panel` que controla lo que pasa dentro de cada panel de la gráfica completa.

```
x <- rnorm(100)
y <- x + rnorm(100, sd = 0.5)
f <- gl(2, 50, labels = c("H", "M"))
xyplot(y ~ x | f)
```



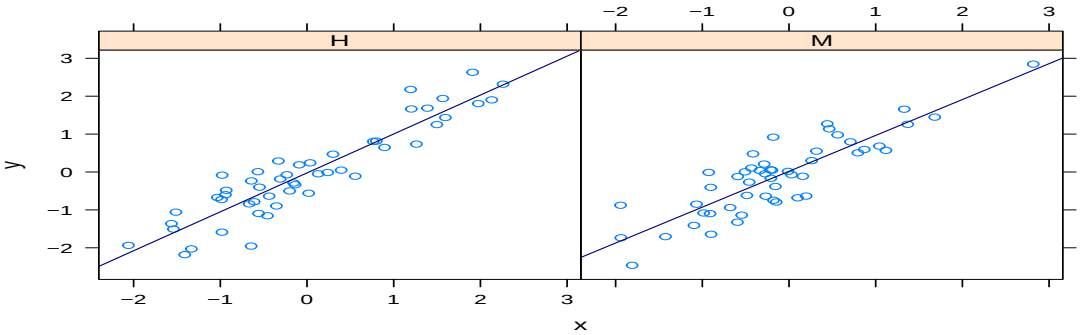
Funciones panel de lattice

```
xyplot(y~x|f,
       panel = function(x,y,...){     # agregamos una función panel que cacula la mediana de los datos en cada grupo
                panel.xyplot(x,y,...)
                panel.abline(h = median(y), lty = 2, col = "red")
              }
)
```



Funciones panel de lattice

```
xyplot(y~x|f,
       panel = function(x,y,...){     # este panel calcula una regresión
                  panel.xyplot(x,y,...)
                  panel.lmline(x,y,col="navy")
               }
)
```



Ejemplos de lattice, ajuste no paramétrico

```
xyplot(ozone ~ radiation | temp.cut,data=environmental,
                    panel = function(x,y,...){
                                panel.xyplot(x, y, ...)
                                panel.loess(x, y, lwd = 2)
})
```