Tutorial - Introducción al R

Curso de Probabilidad y Estadística 2017

Semestre par

Para empezar...

Este *mini-tutorial* está pensado para ir aprendiendo **R** a medida que "metemos mano", sin mucho preludio. Algunas secciones se pueden saltear en una primera lectura ya que las presentamos en este tutorial con el fin de que puedan ser utilizadas cada vez que las precisemos a lo largo del curso.

Lo primero que tendrá que hacer es instalar **R** en su computadora. No se preocupe. **R** es un es un entorno y lenguaje de programación libre, especialmente adaptado al cálculo estadístico y que permite hacer gráficos fácilmente. **R** es gratis y compila y funciona en una gran variedad de plataformas de UNIX, Windows y MacOS. Podrá instalarlo fácilmente desde la siguiente página: "Comprehensive R Archive Network" (CRAN): http://www.cran.r-project.org.

Se recomienda utilizar **RStudio**: https://www.rstudio.com/ en cualquiera de los sistemas operativos, aunque se puede usar **R** desde una terminal. Verá que le pedirá que primero instale **R**. Busque en la página las instrucciones específicas para su sistema operativo, siga los enlaces y todo saldrá bien...

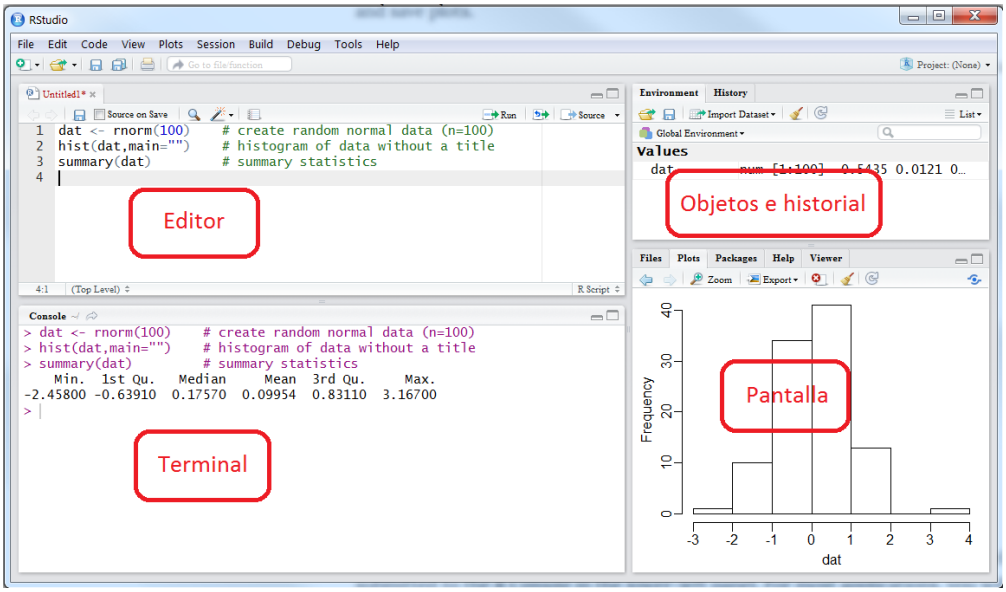
Si ya conoce otros lenguajes de programación tales como *Matlab*, verá que *R* es muy parecido. Si aún no conoce ningún lenguaje de programación (lo de *"aún"* es porque tarde o temprano tendrá que aprender a usar algún lenguaje de programación, aún cuando no estudie Ingeniería en Computación), ¡no se preocupe!. Este lenguaje es un buen lenguaje para aprender a "programar" por dos motivos: primero porque es bastante intuitivo (especialmente si usa *RStudio*) y segundo porque, como es un lenguaje de programación muy utilizado y conocido, está lleno de foros de ayuda de **R** (y de gente muy solidaria). Si *googlea*: "cómo hacer TAL COSA en **R**", seguramente se encontrará con que alguien ya hizo esa pregunta y algún generoso ya se la respondió con los comados a ser ejecutados.

Nota: Estas notas fueron escritas usando RMarkdown (otra de las ventajas de RStudio), que permite escribir informes en los que se presentan simulaciones, resultados o gráficos obtenidos con R (se hace todo en el mismo cuerpo del informe). Además, puede agregar código LaTeX para escribir fórmulas. Los archivos .Rmd (los originales) estarán a su disposición en el EVA para que les sirvan de ejemplo (si quiere usarlos) para cuando tengan que realizar informes para esta u otra materia.

Para utilizar un *comando* cualquiera en **R**, se debe tipear el nombre del comando (función) seguido por sus argumentos entre paréntesis. Esto se debe a que **R** utiliza funciones para obtener resultados. Es recomendable escribir la secuencia de funciones en un archivo de texto, que llamaremos *script* y luego correrlo en la *consola*, ya que el script es fácil de guardar para volver a reproducir dichas secuencias en otra instancia.

Ahora explicaremos qué significa todo esto.

Cuando abra **RStudio** (una vez instalado), se encontrará con que la ventana está partida en 4 pantallas, que describimos a continuación (en sentido antihorario):



Editor

En el extremo superior izquierdo se encuentra la pantalla donde abrirá/escribirá el *script*. Piense en esta pantalla como una especie de block de notas: no ocurre nada cuando escribe en ella; solo se ejecutará una orden una vez que la "pase a la consola" (a esto es a lo que la llamaremos *"ejecutar la orden"*). Se crea un nuevo script con: *File* > *New File* > *Rscript*. Este archivo podrá ser guardado como un *archivo.R* y se podrá abrir en cualquier otro momento para volver a ejecutar la secuencia de funciones. Puede ejecutar una línea (orden) del script en la consola con **Ctrl+r**, seleccionar la línea y luego ejecuar **Run** ó copiar y pegar en la consola. Lo más eficaz será usar **Ctrl+r**, pero respecto a gustos... Seguramente, querrá anotar aclaraciones de los comados para cuando vuelva a usar el script. Todo lo que escriba precedido por el símbolo # significa que lo que le sigue es solo un comentario.

Terminal

La *consola de comandos o terminal*, que es donde "ocurre todo". En ella aparecen en primer lugar una serie de mensajes con información de la versión de **R** que está usando. Debajo de estos mensajes aparece el símbolo > y el cursor parpadeando. Esto indica que **R** está esperando una orden para ser ejecutada. Recuerde que todo lo que escriba en la consola se borrará una vez que cierre la ventana (por eso es tan importante escribir todo en el script y no en la consola).

Pantalla

La pantalla de Ficheros (*Files*), Gráficos (*Plots*), Paquetes (*Packages*), Ayuda (*Help*) y Visor (*Viewer*).

Cada vez que creamos una *gráfica*, aparecerá en esta pantalla. Una vez aquí, podremos exportala como querramos (.pdf, .jpg, etc.).

Para obtener *ayuda* sobre una función específica, ejecutamos en la consola el comando:

```
help("Nombre")
```

La ayuda sobre esta función aparecerá en esta tercer pantalla. El nombre de la función se pasa como argumento de la función help. **R** es sensible a mayúsculas y minúsculas, por lo tanto *A* y *a* son símbolos diferentes y se refieren a variables distintas. Si no recuerda el nombre exacto de una función, se puede buscar en los archivos de ayuda de **R** tipeando:

```
help.search()
```

o *googleando*: "cómo hacer *tal cosa* en **R**".

Historial

Finalmente, en el extremo superior derecho se encontrará con el **Historial** (*History*) y el **entorno de trabajo** (*Enviroment o Workspace*). Desde aquí se podrán "cargar" datos a **R** desde cualquier carpeta. Si no usa **RStudio**, cargar archivos a **R** lleva un poquito más de trabajo.

Ejercicio 1. Mirar la ayuda de la función mean():

```
help("mean")
```

Manipulaciones simples: números y vectores

R opera en datos con estructuras predeterminadas. La estructura más simple es el vector numérico. Para asignar un valor a una variable se puede usar una flecha "<-" o el igual "=".

```
 a<-5 #Le asigno a la variable a el número 5
```

Para crear un vector se pueden *concatenar* varios números, utilizando la función c().

```
x<- c(3.5, 4, 0.6, 8, 12) #De esta manera le asignamos a la variable x
#el vector (3.5, 4, 0.6, 8, 12)
```

La flecha también funciona en la otra dirección "->":

```
c(3.5, 4, 0.6, 8, 12) -> z
```

Para ver el contenido de una variable, podemos tipearla en la consola. Por ejemplo, tipeando x y z verificamos que son el mismo vector.

```
x
```

```
## [1]  3.5  4.0  0.6  8.0 12.0
```

```
z
```

```
## [1]  3.5  4.0  0.6  8.0 12.0
```

También se pueden crear vectores concatenando vectores ya existentes. **R** contiene todos los operadores aritméticos estándar (ver tabla). Estas operaciones se pueden usar también con vectores. Además de las expresiones artiméticas comunes también se pueden usar: log(), sin(), cos(), exp(), tan(). Por ejemplo:

```
u<- 4*x + z
u
```

```
## [1] 17.5 20.0  3.0 40.0 60.0
```

```
v <- log(x)
v
```

```
## [1]  1.2527630  1.3862944 -0.5108256  2.0794415  2.4849066
```

```
w = c(x, z)
w
```

```
##  [1]  3.5  4.0  0.6  8.0 12.0  3.5  4.0  0.6  8.0 12.0
```

Ejercicio 2

- Crear un vector y que contenga al vector x, al número 3 y al vector z multiplicado por 2.
- ¿Cuál es el resultado de $x+y$? (dos vectores de diferente largo). ¿Y de $y+x$?

Ejercicio 3 Utilizar la función *help()* para explorar qué hacen las funciones range(), mean(), length(), sum() aplicadas a un vector.

Ejercicio 4 Explorar qué hace la función sort(). ¿Es posible ordenar los elementos de manera decreciente?

Los vectores se pueden crear "a mano", concatenando como antes, o usando otras de **R**. Para crear vectores que contienen secuencias de números se puede usar **a:b** que crea una secuencia que empieza en a y va sumando 1 hasta llegar a b, o la función **seq(a,b,c)**, que hace lo mismo, pero suma c en vez de uno (c puede ser cualquier real positivo). La función ":" tiene alta prioridad ante otras operaciones.

Ejercicio 5

- Comparar los resultados de la expresión *1:n-1* y *1:(n-1)* para algún número natural n.
- Generar una secuencia de números del 20 al 1.
- Generar otra secuencia del 20 al 2, sólo de números pares.

Para elegir subconjuntos de un vector alcanza con poner el nombre del vector y el índice o índices de los elementos que se quieren elegir entre paréntesis rectos [ ]. Los índices deben de variar entre 1 y *n=length(vector)* (largo del vector). Por ejemplo, para obtener la primera coordenada de y ejecutamos y[1].

Ejercicio 6 Usar la función ":" para elegir las coordenadas del 2 al 5 del vector y (es decir: y_2, y_3, y_4, y_5).

Operadores

Aritméticos	**Comparación**	**Lógicos**
+ adición	< menor	! x NO lógico
- substracción	> mayor	x & y Y lógico
multiplicación	<= menor o igual	x | y O inclusivo
multiplicación	<= menor o igual	xor(x,y) O exclusivo
/ división	>= mayor o igual	
^ exponente	== igual	
%% módulo (resto)	!= distinto	
%/% división entera		

Los vectores lógicos pueden ser utilizados con operadores aritméticos. En ese caso el FALSE se convierte en 0 y el TRUE se convierte en 1. Por ejemplo,

```
y[y > 2] # muestra los valores de y que son mayores que dos
```

```
## [1]  3.5  4.0  8.0 12.0  3.0  7.0  8.0 16.0 24.0
```

```
y>2 # devuelve un vector de TRUE y FALSE según si la
```

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

```
# correspondiente coordenada de y es mayor que 2 o no.
```

Ejercicio 7 Crear un vector m que contenga los elementos de y mayores que 2 y menores que 5. Explorar la diferencia entre las multiplicaciones:

m * m
m % * % m

Matrices

Son la extensión natural de los vectores. Una matriz se define del siguiente modo:

```
( x <- matrix(-1, 4, 5) )
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]   -1   -1   -1   -1   -1
## [2,]   -1   -1   -1   -1   -1
## [3,]   -1   -1   -1   -1   -1
## [4,]   -1   -1   -1   -1   -1
```

```
( x <- matrix(1:20, 4, 5) )
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    5    9   13   17
## [2,]    2    6   10   14   18
## [3,]    3    7   11   15   19
## [4,]    4    8   12   16   20
```

Esto genera una matriz de 4 filas y 5 columnas con los números del 1 al 20. En R la matriz se llena por columnas, es decir, la primer columna de la matriz x son los números del 1 al 4, la segunda son los números del 5 al 8 y así sucesivamente. Lo más importante a la hora de definir una matriz es especificar el vector de dimensiones.

En el ejemplo anterior, x[3, 2] nos daría el segundo elemento de la tercera fila de x, x[, 1] sería la primer columna y x[3,] la tercer fila.

La función matrix() va armando la matriz colgando la información por columnas, en caso de querer armar la matriz por filas se puede modificar el parámetro opcional *byrow*.

> **Ejercicio 8** *Crear un vector de largo 20 con los números que usted quiera. A partir del vector creado, crear una matriz m1 de dimensiones 5 x 4 completándola por filas. A partir del mismo vector, crear otra matriz m2 de dimensiones 2 por 10, completándola por columnas.*

Operaciones básicas con matrices

Dada una matriz A, la función *t(A)* calcula la traspuesta de la matriz A. Dada una matriz A, *nrow(A)* y *ncol(A)* nos dicen el número de filas y columnas de la matriz A y *dim(A) <- c(nrow(A), ncol(A))*. La función *solve()* aplicada a una matriz calcula su inversa. Dadas dos matrices A y B, el producto matricial entre ambas se hace mediante el operador

```
A %*% B
```

Si A y B tienen la misma dimensión, A*B devuelve el producto componente a componente, no el producto matricial.

> **Ejercicio 9** *Crear una matriz cuadrada m3 de dimensión 5. Ver qué operaciones de las vistas anteriormente se pueden aplicar a m1, m2 y m3. Observar en los casos que no se pueden hacer las operaciones, en particular los mensajes que da el programa.*

Es importante recordar que la función *c()*, se puede utilizar para concatenar cualquier tipo de objetos, no sólo variables numéricas, enteras, etc. De la misma familia son las funciones *cbind* y *rbind* que permiten combinar una serie de objetos bien por columnas o por filas. Habitualmente se usan para construir una matrices a partir de vectores.

> **Ejercicio 10**

- *Crear A una matriz 2x2 que contenga los elementos {1, 2, 3, 4}. Crear una matriz B que contenga los elementos de A multiplicados por 3.*
- *Crear una matriz llamada "mat" con los vectores vec1: [3,5,1,4,7], vec2: [4,6,7,9,11], y las columnas 3 a 5 de m3 usando la función cbind().*
- *Calcular el determinante de "mat".*
- *Crear una matriz "mat2" a partir de la matriz "mat" restando 3 al elemento (3,2) y agregando 2 al elemento (5,4).*

Creando funciones

Si en entre las funciones predeterminadas no esta la función que se necesite o si una serie de líneas se va a ejecutar varias veces cambiando algunos parámetros, se puede crear una función que tome esos parámetros como variables. La funcion que crea una función es

function(argumentos) {
código a ejecutar con los argumentos
return(algun_objeto_con_el_resultado *)*
}

Las funciones que solo tengan una única línea a ejecutar no necesitan los {} ni el return(), por ejemplo, una función de dos variables que calcula su producto.

```
g <- function(u, v) u*v
g(-1, 12)
```

```
## [1] -12
```

```
g(pi, sqrt(7))
```

```
## [1] 8.311873
```

Una función puede tener variables con valores por defecto.

```
probando <- function(empleados, el_valor_predefinido=12){
  aux <- empleados[ empleados > el_valor_predefinido ]
  return( min(aux) )
  }

probando( c(23, 45, 1, 4, 17) )
```

```
## [1] 17
```

```
probando( c(23, 45, 1, 4, 17), el_valor_predefinido=20 )
```

```
## [1] 23
```

Incluso pueden no tener argumentos como puede pasar con las funciones predefinidas *history()* o *sessionInfo()*. Para eso las líneas a ejecutar no deben depender de variables.

El resultado a retornar puede ser cualquier objeto incluido una función, en este caso sería una función que crea funciones y no es más que una función dentro de otra y que la primera le fija parámetros a la segunda. Pensemos en x^n y en que interesa según los datos fijar el n que no se conoce previamente, en ese caso una primer función debe tomar el valor n y fijarlo en la segunda, el ejemplo a continuación muestra como es el código correspondiente.

```
fija.el.n <- function(n){
    function(x) x^n
  }
```

```r
g <- fija.el.n(5)
g
```

```
## function(x) x^n
## <environment: 0x0653efe0>
```

```r
str(g)
```

```
## function (x)
##  - attr(*, "srcref")=Class 'srcref'  atomic [1:8] 2 5 2 19 5 19 2 2
##   .. ..- attr(*, "srcfile")=Classes 'srcfilecopy', 'srcfile' <environment: 0x064d9c68>
```

```r
g(1)
```

```
## [1] 1
```

```r
g(2)
```

```
## [1] 32
```

```r
g(4)
```

```
## [1] 1024
```

```r
# Equivale a pasar los dos valores, de n y x a fija.el.n() por separado.
fija.el.n(5)(4)
```

```
## [1] 1024
```

Data frames

Los data.frames (campos de datos) son el objeto más habitual para almacenar datos. La forma de pensar en un data.frame es considerar que cada fila representa a un individuo de una muestra y el correspondiente valor para cada columna representa la medición de alguna variable para ese individuo (fila-individuo, columna-variable). Si, por ejemplo, tenemos 100 alumnos y las notas numéricas de ellos en cada examen junto con la calificación final (tipo caracter), esto podrá representarse con un data.frame de 100 filas con una columna por cada examen y una columna más para la calificación final.

R trae algunos juegos de datos ya pre-cargados, uno de ellos se llama mtcars. Para acceder a él basta llamarlo por su nombre:

```r
mtcars
```

Y vermos todo el contenido de mtcars.

Usar las funciones View(), head() y help() para ver qué contiene esta base de datos.

Tanto en matrices como en data.frames podemos localizar sus objetos ya sea por la posición que ocupan, o usando el nombre de su fila y su columna. Por ejemplo:

```r
mtcars[3, 2]
```

```
## [1] 4
```

```r
mtcars['Datsun 710', 'cyl']
```

```
## [1] 4
```

De la misma manera, para seleccionar columnas de un data.frame, podemos hacer lo mismo que antes, pero usando el comando $.

```
mtcars[ , 4]
```

```
##  [1] 110 110  93 110 175 105 245  62  95 123 123 180 180 180 205 215 230
## [18]  66  52  65  97 150 150 245 175  66  91 113 264 175 335 109
```

```
mtcars$hp
```

```
##  [1] 110 110  93 110 175 105 245  62  95 123 123 180 180 180 205 215 230
## [18]  66  52  65  97 150 150 245 175  66  91 113 264 175 335 109
```

Se puede usar la función *all.equal()* parar verificar que ambas expresiones son equivalentes:

```
all.equal(mtcars$hp, mtcars[, 4])
```

```
## [1] TRUE
```

Ejercicio 11

- *Usar la función dim para saber cuántos autos tiene esta base de datos y cuántas características para cada auto.*
- *Crear un vector que contenga el peso del auto. Transformar dicho vector para cambiar el peso de libras a kg. Agregarlo al data.frame usando $.*

Para matrices y dataframes, colnames() y rownames() son las etiquetas de las columnas y las filas respectivamente. Se puede acceder también con dimnames() que devuelve una lista con ambos vectores. Puede ser útil ver un dataframe como una matriz con columnas que pueden tener diferentes modos y atributos. Sus filas o columnas pueden ser extraídas usando los índices habituales en matrices.

Condicionales y bucles.

R tiene disponible una construcción condicional de la forma:

> if (*condicion a verificar*) {*expresiones a evaluar si se cumple la condicion*} else {*expresiones a evaluar si se cumple la condicion*}

donde *condicion a verificar* evalúa una condición y debe devolver un valor: TRUE o FALSE. Si el resultado es TRUE, entonces devolverá el resultado de *expresiones a evaluar si se cumple la condicion* y si es FALSE el de *expresiones a evaluar si se cumple la condicion*. No es obligatorio incluir el else pudiendo usarse de la forma

> if (*condicion a verificar*) {*expresiones a evaluar si se cumple la condicion*}

El siguiente ejemplo utiliza *if* para decir si la variable *entero* es par o impar.

```
entero <- 2 # Tenemos una variable entera, en este caso vale 2.
if (entero %% 2 == 0) print('el numero es par') else  print('el numero es impar')
```

```
## [1] "el numero es par"
```

Para construir un loop se puede usar el comando "for" que tiene la siguiente forma:

> for (*índice* in *etiquetas*) { *expresiones a evaluar* }

donde *índice* es la variable que varía en el loop, típicamente usamos una letra. *expr 1* es un vector (generalmente es una secuencia, como por ejemplo 1:10), y *expresiones a evaluar* es generalmente un grupo de expresiones, donde alguna de ellas depende de la variable sobre la que estamos iterando.

Supongamos que tenemos el vector numérico y:

```
y <- c(3, 2, 0, 6, 1, 0)
```

y que para cada elemnto de y queremos ver si es igual a 0 o no. Si lo es, remplazaremos el elemento con el número 10.

```
for (i in 1:length(y)){
  if (y[i] == 0) y[i] <- 10
  }
```

Este ejercicio muestra una combinacion de un for y un if que se puede hacer con la única función:

> ifelse(*condición*, *expresión si se cumple la condición*, *expresión si no se cumple*)

La condición se puede hacer con vectores, listas, data.frame y devuelve un objeto igual cambiando los lugares donde se cumplió la condición por los resultados de las expresiones correspondientes.

El ejemplo anterior queda

```
y <- c(3, 2, 0, 6, 1, 0)

ifelse(y==0, 10, y)
```

```
## [1]  3  2 10  6  1 10
```

```
# Esto es, si en una entrada de hay un 0, se cambia a 10, de lo contrario,
# se deja igual.

print( y <- cbind(y, 'cambio' = ifelse(y==0, 10, y)) )
```

```
##      y cambio
## [1,] 3      3
## [2,] 2      2
## [3,] 0     10
## [4,] 6      6
## [5,] 1      1
## [6,] 0     10
```

```
# Es útil para agregar vectores con el cambio.

y <- c(3, 2, 0, 6, 1, 0)

y <- ifelse(y==0, 10, y) # O simplemente sobreescribir.
y
```

```
## [1]  3  2 10  6  1 10
```

> **Ejercicio 12** *Dado el vector: v <- c(4, 5, 9, 2, 1, 4), remplazar los elementos impares con el número 1.*

Otro bucle de interés es poder ejecutar expresiones sin una cantidad de veces predefinida como lo tiene el for. Este cometido lo cumple el siguiente:

> while (*condición*) {*expresiones*}

Tal como se lee, mientras se cumpla *condición* se va a ejecutar *expresiones* tantas veces como sea necesario. Es importante tener en *expresiones* el control de los argumentos de la condición para que en algún momento el while se deje de ejecutar. Para el curso, tiene más sentido usarlo con la función de la siguiente sección (generar eventos aleatorios hasta observar algo, por ejemplo).

Como comentario, solo queda decir que todo lo que se hace con un *for* se puede hacer con un *while*, la diferencia esta en el tamaño de memoria usado y el tiempo de ejecución. Mientras un for suele ser más rápido, es necesario crear el vector *etiquetas* el cual ocupa cierta cantidad de memoria RAM, con lo cual, el vector etiquetas tiene una limitante (debe poder ser contenido en la RAM). Eso no sucede con *while* porque

simplemente no se recorre un conjunto de etiquetas, el espacio usado es solo el espacio necesario para la variable necesaria en la condición, que en general, es de tamaño despreciable, con lo cual el *while* puede sustituir a un *for* en condiciones donde el *for* no es posible. Con esta observación es natural pensar que el *while* sea un mejor bucle, pero el precio a pagar el poco uso de memoria es el tiempo de ejecución, recorrer un conjunto de etiquetas es temporalmente menos costoso que ir controlando condiciones (la condición del *while*). Esto para el curso no es relevante, lo importante es tener el resultado, solo queda el comentario a modo de información.

También se dispone de un equivalente muy cercano al *while* que es el *repeat* junto a *break*, el cual tampoco es relevante para el curso.

Simular variables

Muchas veces queremos obtener valores aleatorios, para realizar ciertas simulaciones. Si bien hay muchas funciones para hacerlo, una de las más simples es la función *sample* y se puede pensar que es la realización de un sorteo aleatorio de un conjunto de etiquetas, elementos o individuos, pudiendo ser con o sin reemplazo. Esto permite poder generar sin necesidad de tener que realizarlo en persona, tiradas de monedas, tiradas de dados, extracciones de subgrupos aleatorios de personas de una población determinada, con las ventajas de ser más rápido y fácil, pudiendo realizase cuantas veces se desee sin un costo temporal significativo.

Ejemplos:

1. Tirar una moneda equilibrada una vez:

```
Moneda <- c("Cara", "Numero")
sample(Moneda, 1)
```

```
## [1] "Numero"
```

2. Tirar una moneda equilibrada 10 veces:

```
sample(Moneda, 10, replace = TRUE)
```

```
##  [1] "Numero" "Cara"   "Numero" "Cara"   "Numero" "Numero" "Cara"
##  [8] "Cara"   "Cara"   "Cara"
```

3. Tirar un dado equilibrado:

```
dado <- 1:6
sample(dado, 1)
```

```
## [1] 3
```

4. Tirar un dado no equilibrado:

Supongamos que tenemos un dado fallado ya que las probabilidades de salir el 1, 2,. . . ,6 son respectivamente: *P("sale el 1") = 1/12*, *P("sale el 2") = 1/12*, *P("sale el 3") = 1/12*, *P("sale el 4") = 1/12*, *P("sale el 5") = 1/3* y *P("sale el 6") = 1/3*. Luego, simulamos una tirada de este dado así:

```
sample(dado, 1, prob= c(1/12, 1/12, 1/12, 1/12, 1/3, 1/3))
```

```
## [1] 6
```

La siguiente tabla resume cómo simular n variables aleatorias X_1, X_2, ... , X_n i.i.d para las distribuciones más conocidas.

Nota: Esta última parte tendrá sentido cuando estemos más avanzados en el curso. Lo dejamos aquí como una referencia para cuando se necesite.

Distribución	Simulación

Distribución	Simulación
$X \sim N(\mu, \sigma)$	rnorm(n, mean= mu, sd= sigma)
$X \sim Exp(\lambda)$	rexp(n, rate= lambda)
$X \sim Poisson(\lambda)$	rpois(n, lambda)
$X \sim Bin(N, p)$	rbinom(n, size= N, prob= p)
$X \sim Geo(p)$	rgeo(n, prob= p)
$X \sim BinNeg(k, p)$	rnbinom(n, size= k, prob= p)
$X \sim Unif(a, b)$	runif(n, min= a, max= b)

Junto con sample() es importante la función set.seed(). Esta función permite generar siempre los mismos valores aleatorios luego de fijar una semilla. Si pensamos en experimentos llevados a cabo, por ejemplo, un experimento de dados, dicho experimento se hace una vez y se guardan los resultados aleatorios de ese experimento para usar luego. Si simulamos el experimento con sample(), cada vez que se ejecute va a dar un resultado distinto. En algún caso, puede ser útil tener los valores del experimento, para reproducir lo que se desee, en ese caso es que se usa set.seed() ya que una vez fijada la semilla, todo lo que se sortee va a aparecer siempre en el mismo orden, o sea, se fijan los resultados del experimento aleatorio.

Hay que tener cuidado con no elegir siempre la misma semilla, eso estaría dejando para usar siempre la misma secuencia aleatorio y todos los experimentos serían el mismo. Una forma de evitar esto es sortear un número de algún conjunto de valores, ver cual es y usar ese valor como semilla.

```
sample(letters[1:20], 7, replace=T)
```

```
## [1] "o" "j" "g" "s" "s" "g" "e"
```

```
set.seed(3541)
sample(letters[1:20], 7, replace=T)
```

```
## [1] "p" "l" "q" "d" "o" "p" "r"
```

```
sample(letters[1:20], 7, replace=T)
```

```
## [1] "d" "e" "s" "j" "f" "k" "m"
```

```
set.seed(3541)
sample(letters[1:20], 7, replace=T)
```

```
## [1] "p" "l" "q" "d" "o" "p" "r"
```

```
sample(letters[1:20], 7, replace=T)
```

```
## [1] "d" "e" "s" "j" "f" "k" "m"
```

Como se puede ver, cada vez que se fije la misma semilla, la secuencia aleatoria se repite, esto permite la reproducción de experimentos y de resultados. No hay que confundir con que se pierde aleatoriedad (salvo que siempre se elija la misma semilla, en ese caso si es cuestionable), sino que simplemente equivale a hacer un experimento y guardar los datos para procesarlos luego, es una forma de reproducir el mismo experimento cada vez que se desee. En un texto que se explicite el set.seed(), es facil poder reproducir lo que se lee, si el texto no hizo set.seed(), los resultados se pareceran pero difícilmente sean exactamente los mismos.

En caso de querer simular valores hasta que se cumpla alguna algo, lo usual es combinarlo con un *while*. Por ejemplo, jugar a tirar tres dados, el juego termina cuando se saca por segunda vez una suma de 3 entre todos los dados.

```r
set.seed(1039257)

# Tirar tres dados es como sacar de un conjunto de etiquetas numeradas del 1 al 6, tres
# consecutivas y con reposición al conjunto la etiqueta extraida en cada oportunidad.
# Esto simula es experimento individual.
sample(1:6, 3, replace=TRUE)
```

```
## [1] 1 2 6
```

```r
# Lo que va a decir si el while termina es algún indicador reportando si la
# suma fue 3 por segunda vez. Por ejemplo, antes del while se puede crear una
# variable que sea cant_tres, inicializada en 0 porque antes de empezar a jugar
# no hay jugadas que hayan sumado 3 y que por cada vez que una jugada sume 3,
# la variabale cant_tres aumente en 1. Esto se controla con un if.

sum( sample(1:6, 3, replace=TRUE) ) == 3
```

```
## [1] FALSE
```

```r
# Esto suma el resultado de cada dado y verifica si la suma es 3 o no.

# Además hay que tener un registro de la cantidad de veces que se jugo, similarmente se
# puede crear una variable jugadas que comienza valiendo 0 porque antes de empezar
# a jugar no hay jugadas hechas. Luego, cada vez que se ejecute la expresión del
# while, jugada debe aumentar en 1 sin importar si se suma 3 o no.
# Todo quedaría así.

jugadas <- 0
cant_tres <- 0

while ( cant_tres < 2 ) {
  jugadas <- jugadas + 1 # Aumenta la cantidad de jugadas.
  valores <- sample(1:6, 3, replace=TRUE) # Se juega
  if ( sum(valores) == 3 ) cant_tres <- cant_tres + 1 # Verifica si gana.
}

print(
  paste('Se necesito jugar', jugadas, 'veces para observar una suma de 3, por segunda vez.')
  )
```

```
## [1] "Se necesito jugar 622 veces para observar una suma de 3, por segunda vez."
```

```r
# Eso simula una sola vez el experimento, puede resultar más interesante simularlo
# más veces para ver como se comporta las jugadas necesarias para ver una suma de 3
# por segunda vez. Y como esa cantidad de repeticiones es un valor que si se puede elegir
# previamente, podemos llevar a cabo eso por medio de un for, en donde cada iteracion
# del for va a ser ejecutar el while.

n <- 12
repeticiones <- rep(NA, n)
for (i in 1:n) {
  jugadas <- 0
  cant_tres <- 0

  while ( cant_tres < 2 ) {
    jugadas <- jugadas + 1 # Aumenta la cantidad de jugadas.
```

```
    valores <- sample(1:6, 3, replace=TRUE) # Se juega
    if ( sum(valores) == 3 ) cant_tres <- cant_tres + 1 # Verifica si gana.
  }

  repeticiones[i] <- jugadas
  print(
    paste('Se necesito jugar', jugadas, 'veces para observar una suma de 3, por segunda vez.')
    )
}
```

```
## [1] "Se necesito jugar 335 veces para observar una suma de 3, por segunda vez."
## [1] "Se necesito jugar 464 veces para observar una suma de 3, por segunda vez."
## [1] "Se necesito jugar 186 veces para observar una suma de 3, por segunda vez."
## [1] "Se necesito jugar 129 veces para observar una suma de 3, por segunda vez."
## [1] "Se necesito jugar 260 veces para observar una suma de 3, por segunda vez."
## [1] "Se necesito jugar 598 veces para observar una suma de 3, por segunda vez."
## [1] "Se necesito jugar 1101 veces para observar una suma de 3, por segunda vez."
## [1] "Se necesito jugar 107 veces para observar una suma de 3, por segunda vez."
## [1] "Se necesito jugar 355 veces para observar una suma de 3, por segunda vez."
## [1] "Se necesito jugar 255 veces para observar una suma de 3, por segunda vez."
## [1] "Se necesito jugar 121 veces para observar una suma de 3, por segunda vez."
## [1] "Se necesito jugar 326 veces para observar una suma de 3, por segunda vez."
```

```
summary(repeticiones)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   107.0   171.8   293.0   353.1   382.2  1101.0
```

Procedimientos gráficos

El lenguaje R dispone de varias funciones preparadas para la representación gráfica de datos y estas serán muy importantes a lo largo del curso:

Estas funciones se dividen en dos grandes grupos:

1. Gráficos de alto nivel: crean un nuevo gráfico en la ventana de gráficos.
2. Gráficos de bajo nivel: permiten añadir líneas, puntos, etiquetas, etc. a un gráfico ya existente.

Gráficos de alto nivel

Algunas de las funciones que reproducen gráficos son: *plot()*, *barplot*, *hist()* y *curve*, que decribiremos brevemente a continuación:

1. **plot():** Esta función tiene muchas variantes y dependiendo del tipo de datos que se le pasen como argumento actuará de modos distintos (más adelante hablaremos de estos argumentos). Lo más común es plot(x,y) para representar un diagrama de puntos de y respecto de x. Por lo general, x e y serán vectores $x = (x_1, \ldots , x_n)$ e $y = (y_1, \ldots , y_n)$ tales que x_i e y_i pueden ser números, letras, palabras, etc. ("objetos de **R**"). plot(x, y) presentará gráficamente los pares (x_i, y_i).
2. **barplot():** La función *barplot* recibe un vector x y grafica en barras sus valores, en el orden que los recibe. Resulta adecuada para graficar información por categorías.
3. **hist():** Un *histograma* es una representación gráfica de un vector de coordenadas numéricas $x = (x_1, \ldots , x_n)$ $ en forma de barras, donde la superficie de cada barra es proporcional a la *frecuencia* con que los valores x_i caen en intervalos (bases de las barras) definidos por defecto por **R** o que uno puede contruir "a mano". *hist(x)* devuelve el histograma del vector $ x $.

4. **curve():** curve(f, from= x_1, to= x_2) dibuja la gráfica $\{(x, f(x)): x \in [x_1, x_2]\}$ si f fue declarada antes como una función real o ya viene incluída en **R**. Por ejemplo, curve(exp, from=0, to=1) devuelve el gráfico de la función $f(x)=e^x$ en el intervalo [0, 1].

Se pueden completar estas funciones agregándole como *argumento* los siguientes parámetros:

- **type** Indica el tipo de gráfico a realizar, cabe destacar:
  - type="p" (points) sirve para representar puntos (opción por defecto),
  - type="l" (lines) para unir los puntos con líneas,
  - type="b" (both) marca los puntos y los une con líneas,
  - type="s" (steps) une los puntos tipo escalera,
  - type="h" (histogram) marca líneas verticales (tipo histograma),
- **main** Para agregarle un título al gráfico (main="título entre comillas")
- **sub** Para agregarle un subtítulo al gráfico (sub="subtítulo entre comillas")
- **xlab** Para ponerle un nombre al eje de las **x** (xlab="título para el eje de las x")
- **ylab** Para ponerle un nombre al eje de las **y** (ylab="titulo para el eje de las y".)
- **xlim=, ylim=** Especifica los límites inferiores y superiores de los ejes. (xlim= c(0, 2) graficará los valores de x entre 0 y 2.)
- **pch** Indica la forma en que se dibujarán los puntos. Podemos escribir pch="el chirimbolo que querramos" (pch="*" si se quiere que los puntos aparezcan como "*") o pch=un número del 1 al 25 ya que la función pch tiene asociada una figura distinta para cada uno de estos números (usar help("pch") para ver que figura está asociada a cada número).
- **lty** Indica la forma enque se dibujarán las líneas (Line TYpe). (lty=2 ; usar help("par") para ver qué valores acepta).
- **lwd** Indica el ancho de las líneas (Line WiDth). (lty=2 ; usar help("par") para ver qué valores acepta). Tiene por defecto valor 1.
- **col** Es el color usado para el grafico (ya sea para puntos, líneas...). (col="Nombre del color en Inglés" o un número ; usar colours() para ver qué colores hay).
- **font** Es la fuente a usar en el texto, cada número tiene asociado una fuente. (font=3 ; para más información buscar en help("plot").
- **add=TRUE** Fuerza a la función a actuar como si fuese de bajo nivel (intenta super- ponerse a un gráfico ya existente). Hay que tener cuidado, no sirve para todas las funciones.

Gráficos de bajo nivel

Son de gran utilidad para sobreescribir un gráfico de alto nivel, compararlo con otro superponiendo ambos, etc. Destacan los siguientes (help(par) para una descripción completa de estos y otros parámetros):

- **lines** Permite superponer nuevas funciones en una gráfica ya existente.
- **points** Permite añadir puntos.
- **legend** Para añadir una nueva leyenda.
- **text** Añade texto.

Muchos de estos parámetros se pueden ingresar como un único valor, o como un vector de valores. En este caso el gráfico irá alternando dicho parámetro en cada punto/línea de la gráfica.

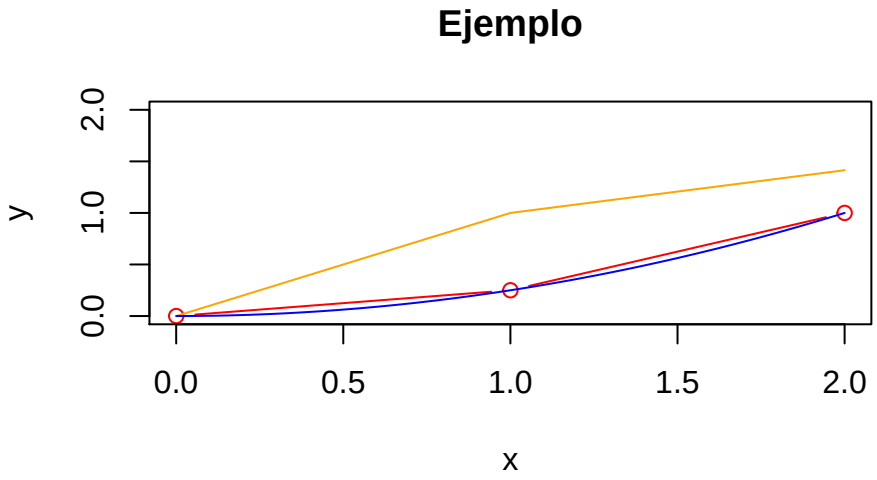
Una vez que en la ventana de gráficos tenemos algo representado, existen dos funciones que nos permiten trabajar interactivamente sobre dicha ventana:

- La función *locator()* sitúa el cursor en la ventana de gráficos y cada vez que pulsemos el botón izquierdo del mouse nos devolverá las coordenadas del punto en el que hayamos marcado.
- La función *identify()* nos permite marcar uno de los puntos del gráfico (con el ratón, igual que antes), y nos devuelve la componente de los vectores representados que dio lugar a ese punto. Muy útil en estadística para identificar outliers.

Ejemplos

El siguiente ejemplo primero grafica los *puntos* (x_i, y_i) con $y_i = f(x_i)$ siendo f la función $f(x)= 1/4\ x^2$ para $x_i = 0, 1, 2$, unidos por líneas. Luego, sobreescribe sobre la misma, los puntos $(x_i, x_i^{1/2}\}\$)$ unidos por líneas, usando la función *lines*. Finalmente, sobreescribe la gráfica de f en el intervalo [0, 2].

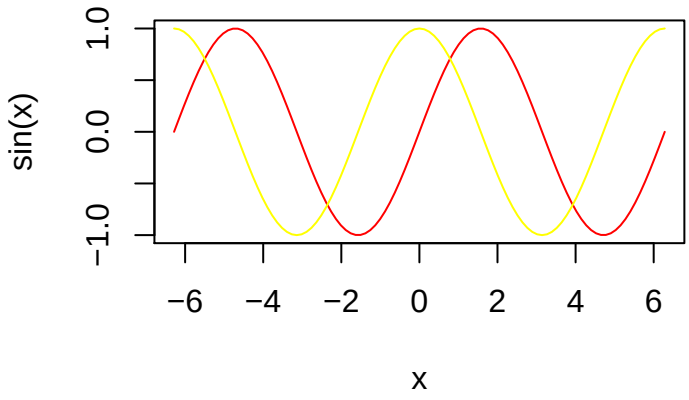
```
x <- 0:2
y <- (1/4)*x^2 # Evalúa la función (1/4)x^2  en cada entrada del vector x.
z  <- sqrt(x) # Idem con la función raíz.
plot(x, y, type = 'b', main = 'Ejemplo',
    col = 'RED', ylim = c(0,2))
lines(x,z, col = 'ORANGE')
curve((1/4)*x^2, from=0, to=2, col= 'BLUE', add=TRUE)
```



Nota: Observar que la función de alto nivel *curve* se transformó en una de bajo nivel debido al argumento *add=T*.

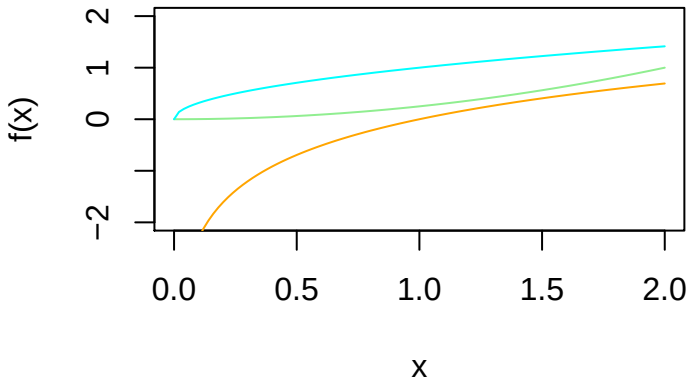
También se puede usar *curve* con funciones predefinidas.

```
curve(sin, -2*pi, 2*pi, col='red')
curve(cos, add=T, col='yellow')
```



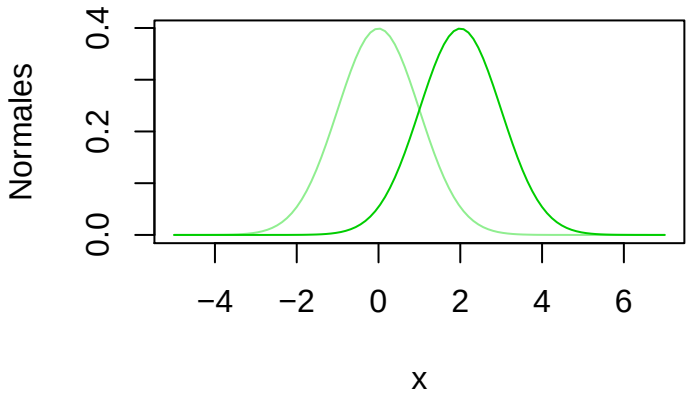
Los ejemplos anteriores con esta modalidad.

```
## Se crea una función para graficar.
f <- function(x) (1/4)*x^2
g <- function(u) sqrt(u)

curve(f, 0, 2, ylim=c(-2, 2), col='lightgreen')
curve(g, add=T, col='cyan')
curve(log, add=T, col='orange')
```



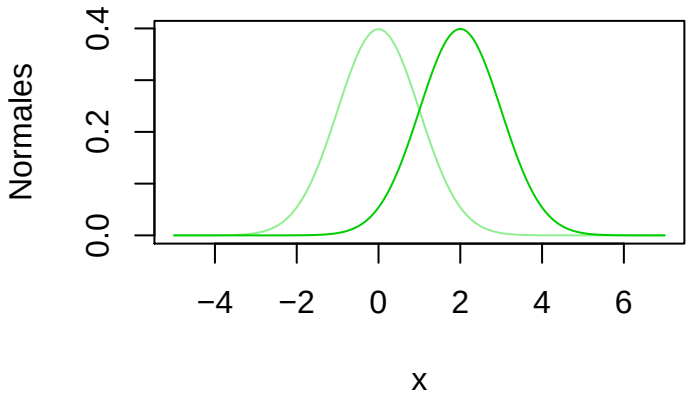
Usando funciones que seran necesarias durante el curso.

```
curve(dnorm, -5, 7, col='lightgreen', ylab='Normales')
f <- function(x) dnorm(x, mean=2)
curve(f, add=T, col='green3')
```



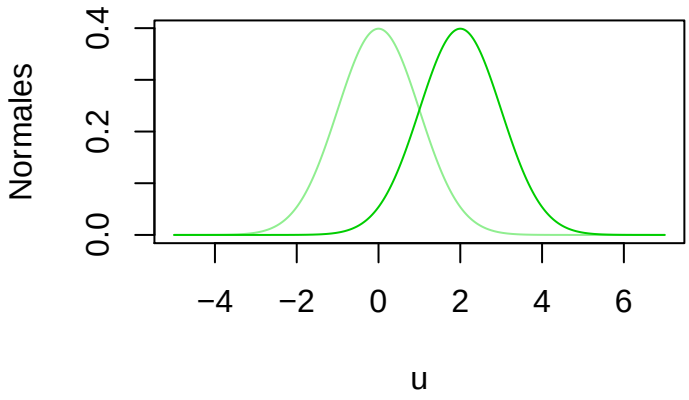
Otra manera combinando *points*.

```
curve(dnorm, -5, 7, col='lightgreen', ylab='Normales')
u <- seq(-5, 7, by=0.01)
points(u, dnorm(u, mean=2), col='green3', type='l')
```



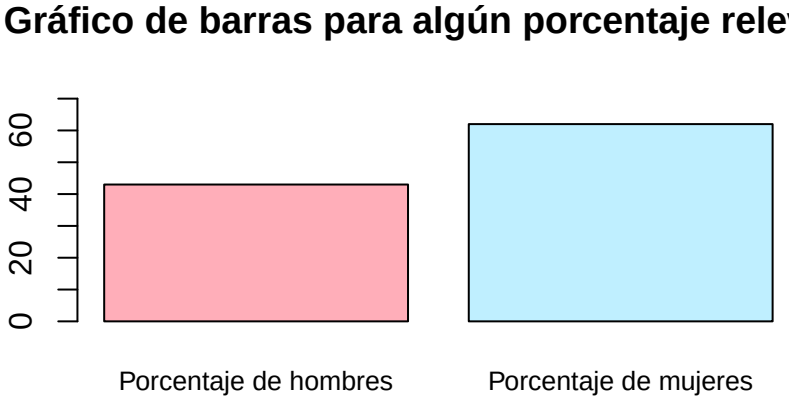
Con *plot* y *points*.

```
u <- seq(-5, 7, by=0.01)
plot(u, dnorm(u), type='l', col='lightgreen', ylab='Normales')
points(u, dnorm(u, mean=2), col='green3', type='l')
```



A continuación veremos un ejemplo de lo que devuelve la función *barplot()*.

```
datos_porc = c(43, 62) #por ejemplo, podemos suponer que
#son porcentajes tomados en categorías hombre/mujer
barplot(datos_porc, main = "Gráfico de barras para algún porcentaje relevado",
        names.arg = c('Porcentaje de hombres', 'Porcentaje de mujeres'),
        cex.names=0.8, col = c('lightpink1', 'lightblue1'), ylim=c(0,70))
```

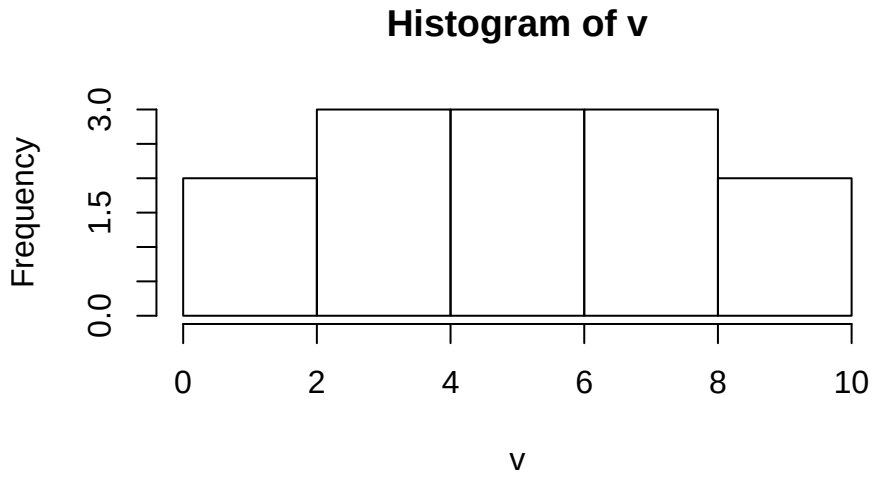


Finalmente, veamos ejemplos de cómo usar la función *hist()*.

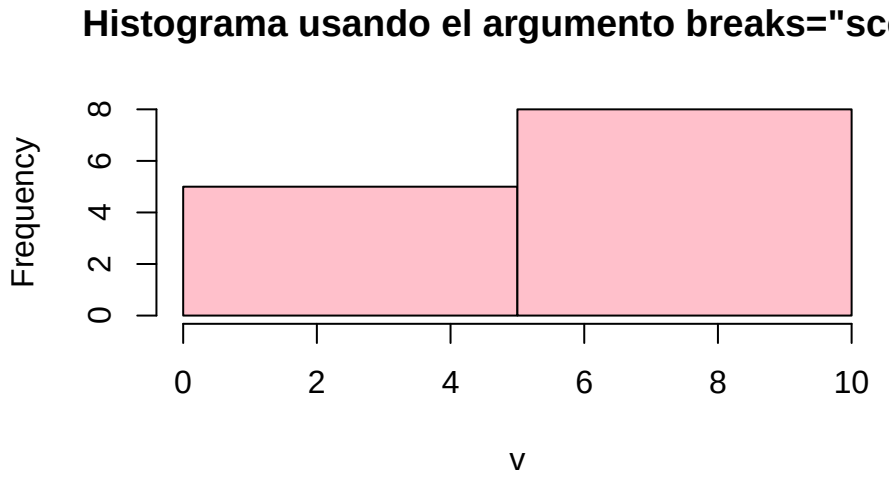
```
v <- c(1, 1.6, 3, 7,  3, 3.1, 5.4, 5.5, 6, 8, 9, 8, 8.3)
```

Este es el histograma con la partición de los intervalos (bases de las barras) que **R** define por defecto y luego observamos qué ocurre si los modificamos (ver qué valores podemos asignarle al argumento *breaks*):

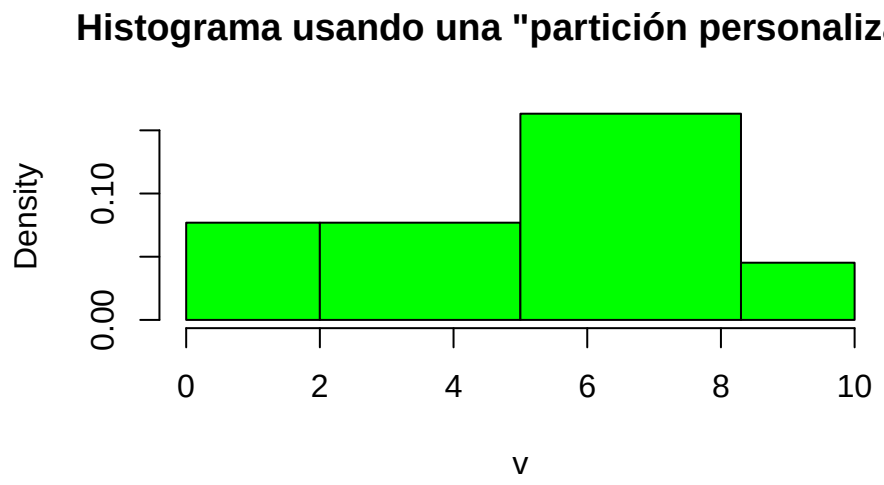
```r
hist(v)
```



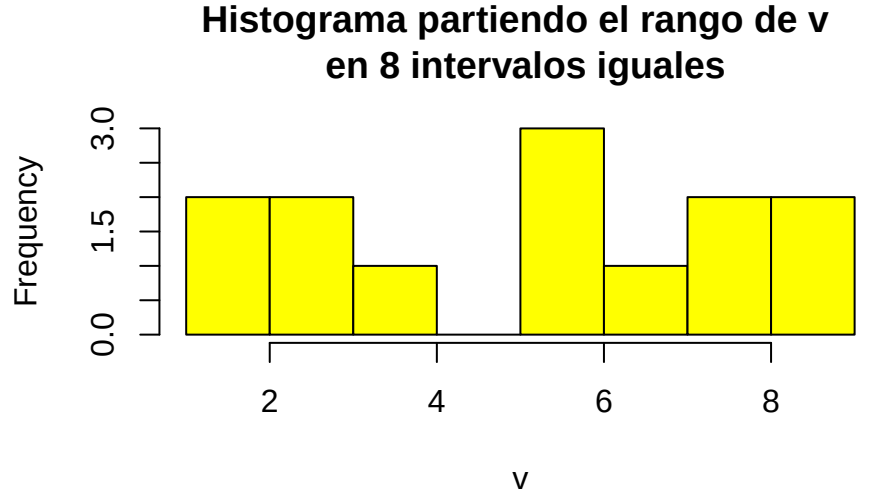

```r
hist(v, breaks= "scott", main= 'Histograma usando el argumento breaks="scott"', col= "pink")
```




```r
hist(v, breaks= c(0, 2, 5, 8.3, 10),
    main= 'Histograma usando una "partición personalizada"', col= "green")
```




```
hist(v, breaks=8,
    main= 'Histograma partiendo el rango de v \n en 8 intervalos iguales', col= "yellow")
```



Presentación de múltiples gráficas.

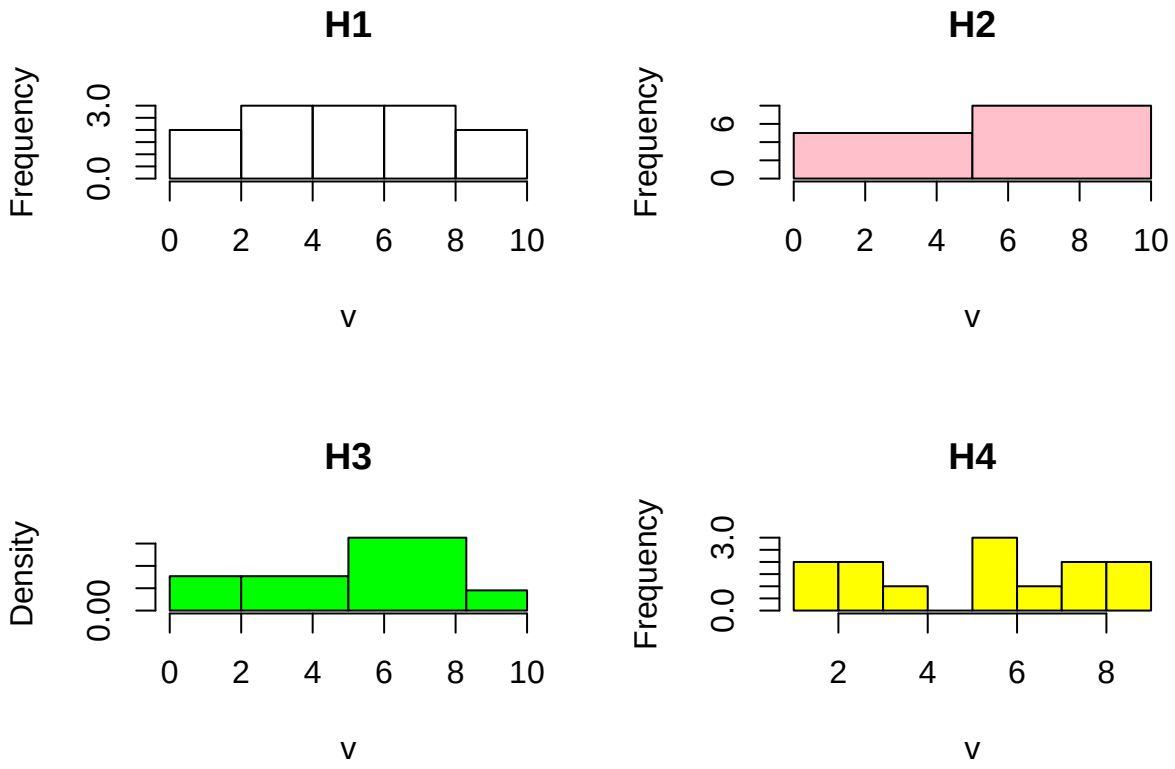
En el ejemplo anterior, estaría bueno poder presentar todos los histogramas en una misma imagen.

R permite presentar varias gráficas en una misma ventana. Por ejemplo, podemos usar la función *split.screen()*, a la que tenemos que pasarle un vector que diga en cuántas filas y columnas (en ese orden) queremos partir la pantalla. Luego, nos posicionamos en cada sub-ventana con la función *screen(número de la ventana)* e indicamos qué graficar.

```
v=c(1, 1.6, 3, 7,  3, 3.1, 5.4, 5.5, 6, 8, 9, 8, 8.3)
split.screen(c(2,2))
```

```
## [1] 1 2 3 4
```

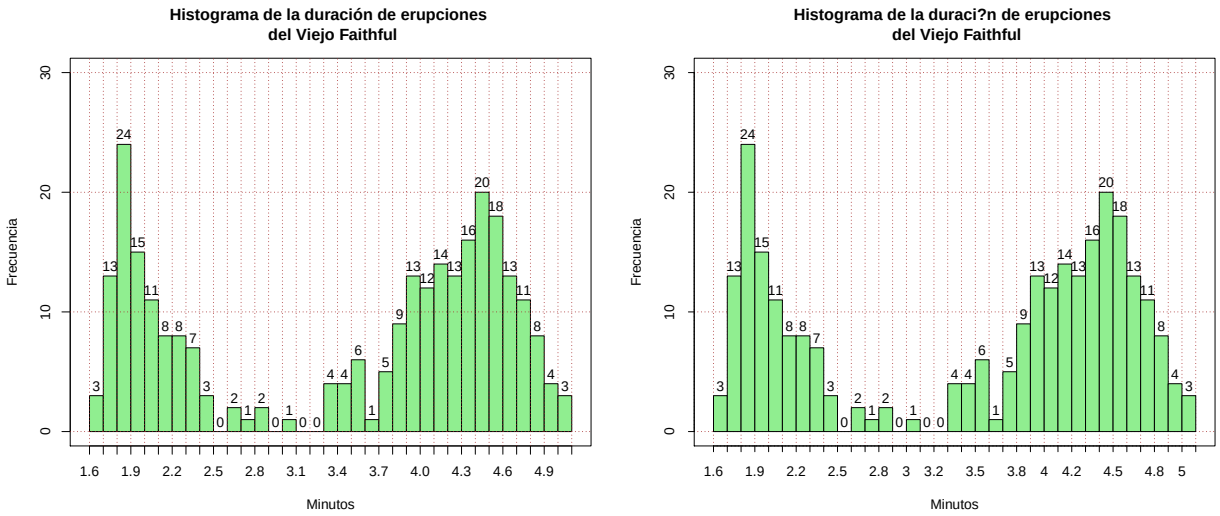
```r
screen(1)
hist(v, main= "H1")

screen(2)
hist(v, breaks= "scott", main= 'H2', col= "pink")

screen(3)
hist(v, breaks= c(0,2,5,8.3,10), main= 'H3', col= "green")

screen(4)
hist(v, breaks=8, main= 'H4', col= "yellow")
```



En general las marcas en los ejes no son muy buenas. Se pueden manipular para que se vean mejor, de paso usamos otra función para particionar la ventana de gráficos.

```r
par( mfrow=c(1, 2) )

# 1) Se acomodan los ejes y se agrega una grilla para ubicar mejor los valores.
# Se guarda el histograma sin graficar para tener las marcas.
# Queda como ejercicio ver que es hi (usar str() o attributes())
hi <- hist( faithful$eruptions, plot=F, n=30 )

# Grafico de lo mas interesante.
plot(hi,
     main='Histograma de la duración de erupciones \n del Viejo Faithful',
     xlab='Minutos', ylab='Frecuencia',
```

```
     axes=F, labels=T,
     ylim=c(0, ceiling(max(hi$counts)/10)*10),
     col='lightgreen' )
box()

# Unos ejes más adecuados.
axis(side=1, at=hi$breaks)
axis(side=2, at=pretty( c(0, ceiling(max(hi$counts)/10)*10), n=4))

# Finalmente la grilla.
abline(v=hi$breaks,
       h=pretty( c(0, ceiling(max(hi$counts)/10)*10), n=4),
       lty='dotted', col='brown')

# 2) Ahora que la grilla quede detrás del histograma.
# De nuevo se hace un histogrma sin graficar, con el objetivo de tener las marcas para los ejes.
hi <- hist(faithful$eruptions, plot=F, n=30)
marcas.x <- (hi$breaks - hi$breaks[1])/(hi$breaks[ length(hi$breaks) ] - hi$breaks[1])
marcas.y <- pretty( c(0, ceiling(max(hi$counts)/10)*10), n=4)
marcas.y <- (marcas.y - marcas.y[1])/(marcas.y[ length(marcas.y) ] - marcas.y[1])

# Ahora se hace el mismo gráfico usando la información anterior.
# Es importante que el gráfico sea exactamente el mismo.
hist(faithful$eruptions, breaks=hi$breaks,
     main='Histograma de la duraci?n de erupciones \n del Viejo Faithful',
     xlab='Minutos', ylab='Frecuencia',
     ylim=c(0, ceiling(max(hi$counts)/10)*10),
     labels=T, axes=F,
     col='lightgreen',
     panel.first={
         axis(1, at=marcas.x, labels=hi$breaks, tck=1, col.ticks="brown", lty='dotted')
         axis(1, at=marcas.x, labels=F, tck=-0.015) ## Esto hace la marquita negra en el n?mero.
         axis(2, at=marcas.y, labels=pretty( c(0, ceiling(max(hi$counts)/10)*10), n=4),
              tck=1, col.ticks="brown", lty='dotted')
         axis(2, at=marcas.y, labels=F, tck=-0.015) ## Esto hace la marquita negra en el n?mero.
         box()
     })
```

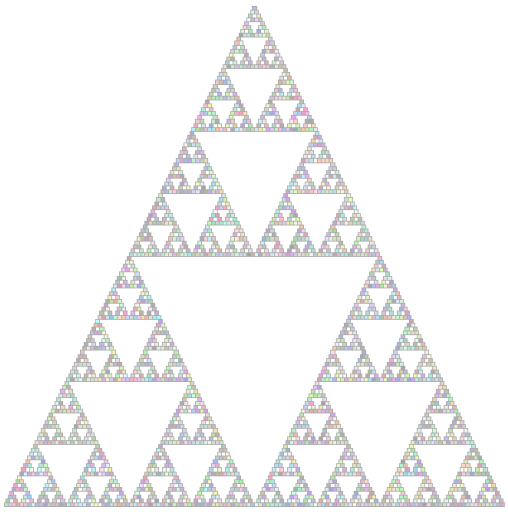


Ejercicio 13

- *Explorar la ayuda de plot, barplot, hist y curve.*
- *Crear gráficos de barras a partir de los datos edad-muerte-australia2. Dado que estos datos contienen la información para hombres y mujeres, explorar distintas maneras de representar estos datos.*

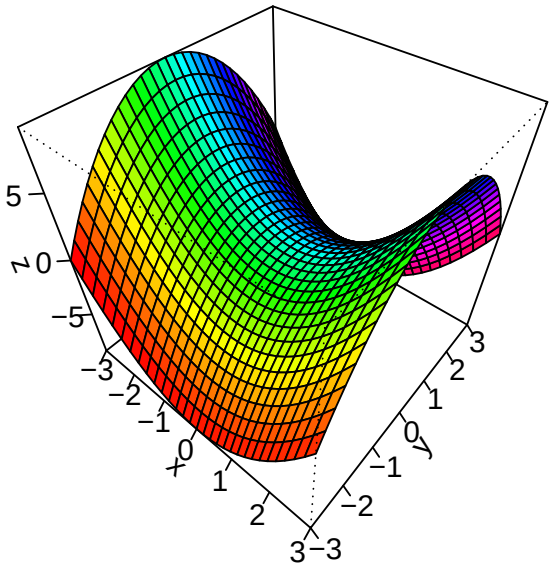
Alguna figura.

```
# https://www.r-bloggers.com/building-affine-transformation-fractals-with-r/

library(grid)
grid.newpage()

pmax <- 7 # Depth of the fractal
vp1 <- viewport(x=0.5, y=0.5,w=1, h=1)
vp2 <- viewport(w=0.5, h=0.5, just=c("centre", "bottom"))
vp3 <- viewport(w=0.5, h=0.5, just=c("left", "top"))
vp4 <- viewport(w=0.5, h=0.5, just=c("right", "top"))
pushViewport(vp1)
m <- as.matrix(expand.grid(rep(list(2:4), pmax)))

for (j in 1:nrow(m)){
  for(k in 1:ncol(m)) {pushViewport(get(paste("vp",m[j,k],sep="")))}
    grid.rect(gp=gpar(col="dark grey", lty="solid",
    fill=rgb(sample(0:255, 1),sample(0:255, 1),sample(0:255, 1), alpha= 95, max=255)))
    upViewport(pmax)
}
```



También es posible crear graficos de superficies

```
f <- function(x, y) x^2 - y^2
x <- y <- seq(-3, 3, 0.2)
z <- outer(x, y, f)
persp(x, y, z, col=rainbow(length(z)), ticktype='detailed', phi=45, theta=40)
```



```r
f <- function(x, y) dnorm(x) * dnorm(y)
x <- y <- seq(-3, 3, 0.2)
z <- outer(x, y, f)
persp(x, y, z, col=rainbow(length(z)), ticktype='detailed',
 phi=30, theta=40, main='Normal estandar bivariada')
```