Introducción a R

Sesión 4
Gráficos y Programación

Joaquín Ortega Sánchez

Centro de Investigación en Matemáticas, CIMAT
Guanajuato, Gto., Mexico

Verano de Probabilidad y Estadística
Junio-Julio 2008

Outline

Outline

Comandos de Bajo Nivel

Estos comandos sirven para ajustar los resultados de los comandos que estudiamos en la sección anterior y para añadir información. Algunas de las funciones más útiles son:

`points(x,y)` Añade puntos al gráfico activo.

`lines(x,y)` Añade curvas al gráfico activo.

`abline(a,b)` Añade una recta de pendiente `b` y corte `a`.

`abline(h=y)` `h=y` indica recta horizontal con altura `y`.

`abline(v=x)` `v=x` indica recta vertical que pasa por $x =$`x`.

`abline(lm.obj)` `lm.obj` es el resultado de un modelo lineal y se dibuja la recta correspondiente al modelo.

`polygon(x,y,...)` Dibuja un polígono definido por los vértices (ordenados) en `x`, `y`

Comandos de Bajo Nivel

`legend(x,y,legend,...)`
Añade una leyenda a la gráfica activa en la posición específicada. Los caracteres usados, estilos de línea, colores, etc. se identifican con las etiquetas incluidas en el vector `legend`. Es necesario incluir al menos un vector *v* de igual longitud que `legend` que contiene los valores de la característica que se quiere identificar, según se indica a continuación:

`legend( , fill=`*v*`)` Colores correspondientes a las regiones sombreadas,

`legend( , col=`*v*`)` Colores de los puntos o líneas,

`legend( , lty=`*v*`)` Estilos de líneas,

`legend( , lwd=`*v*`)` Anchos de líneas,

`legend( , pch=`*v*`)` Caracteres usados para la gráfica,

Comandos de Bajo Nivel

`text(x,y,labels,...)` Añade texto en el punto con coords. `x`, `y`.

`title(main,sub)` Añade un título. `main` se coloca en la parte superior de la gráfica con letras grandes y `sub` (opcional) añade un subtitulo debajo del eje *x* con tamaño de letra más pequeño.

`axis(side,...)` Añade un eje a la gráfica activa en el lado indicado por el primer argumento (1 a 4, contando desde abajo en el sentido de las agujas del reloj). Otros argumentos controlan la posición del eje en la gráfica, las marcas y las etiquetas. Es útil para añadir ejes cuando se usa la función `plot` con argumento `axes=FALSE`.

`rug(x)` Dibuja los datos `x` en el eje *x* como segmentos verticales cortos.

`box()` Dibuja un rectángulo alrededor de la gráfica activa.

Comandos de Bajo Nivel

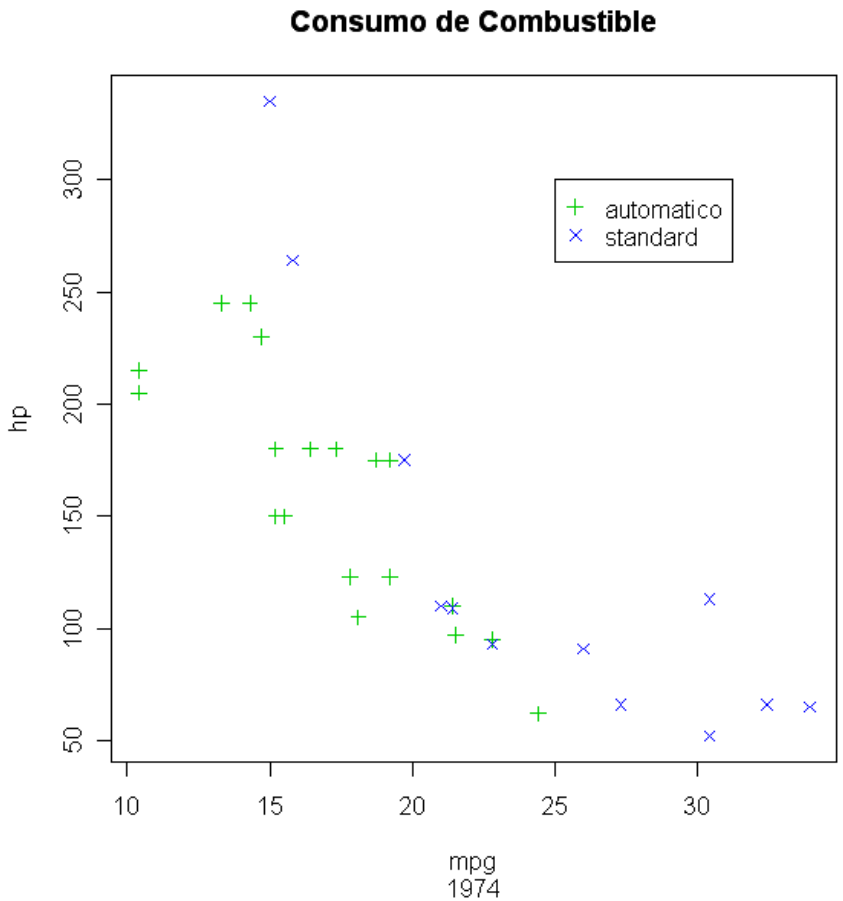
Veamos un ejemplo que usa algunos de estos comandos para hacer una gráfica usando datos del conjunto `mtcars`

```
> attach(mtcars)
> plot(disp, mpg, type='n',axes=F)
> points(disp[cyl==4],mpg[cyl==4],pch=16,col=2)
> points(disp[cyl==6],mpg[cyl==6],pch=17,col=3)
> points(disp[cyl==8],mpg[cyl==8],pch=18,col=4)
> axis(1)
> axis(4)
> title('Consumo de Combustible', 1974)
> arrows(470, 15, 470, 12, code = 2)
> text(475,17,'menor\nrendimiento',font=3,adj=1)
> arrows(100,34, 140,34, code=1)
> text(225, 34, 'mayor rendimiento', font=2)
> leg.txt <- c('4 cil.', '6 cil.', '8 cil.')
> legend(350,32, leg.txt,col=2:4, pch=16:18)
```

Ejercicio

Ejercicio 4.1

1. Haga una gráfica como la de la pantalla.



Outline

Parámetros Gráficos

Además de los comandos de bajo nivel, es posible modificar la presentación de los gráficos usando los parámetros gráficos. Estos pueden ser usados como opciones de funciones gráficas (pero esto no siempre funciona) o con la función `par` que cambia de manera pemanente los parámetros, es decir, las gráficas que se hagan a continuación se harán usando los nuevos parámetros. Hay 73 parámetros gráficos. La lista completa se puede ver usando la instrucción `?par`. A continuación presentamos algunos de los parámetros más útiles.

Parámetros Gráficos

`adj` Controla la justificación del texto con respecto al borde izquierdo. `0` corresponde a justificado a la izquierda, `0.5` a centrado, `1` a justificado a la derecha y valores mayores a 1 mueven el texto más a la izquierda. Si se dan dos valores, por ejemplo `c(0,1)`, el segundo controla la justificación vertical.

`bg` Especifica el color del fondo.

`bty` Controla el tipo de caja que se dibuja alrededor de la gráfica. Los posibles valores son `"o"`, `"l"`, `"7"`, `"c"`, `"u"`, `"]"` y las formas corresponden aproximadamente a la forma del símbolo. Si `bty="n"` no se dibuja ninguna caja.

`cex` Número que controla el tamaño de los textos y símbolos con respecto al valor por defecto. Los siguientes parámetros tienen el mismo efecto para los números de los ejes, `cex.axis`, las etiquetas de los ejes `cex.lab`, el título, `cex.main`, y el subtítulo, `cex.sub`.

Parámetros Gráficos

`col` Controla el color de los símbolos. Al igual que para `cex` existen `col.axis, col.lab, col.main, col.sub`.

`font` Un entero que controla el tipo de letra: `1`: normal, `2`: cursiva `3`: negritas, `4`: negritas cursivas.

`las` Un entero que controla la orientación de las etiquetas de los ejes: `0`: paralelo a los ejes, `1`: horizontal, `2`: perpendicular a los ejes, `3`: vertical.

`lty` Controla el tipo de línea según los códigos de la tabla 3.1.

`lwd` Número que controla el ancho de las líneas.

Parámetros Gráficos

`mfcol` Un vector de forma `c(m,n)` que divide la ventana gráfica en una matriz de gráficos con `m` filas y `n` columnas. Las gráficas se dibujan por columna.

`mfrow` Similar al anterior pero las gráficas se dibujan por filas.

`pch` Controla el tipo de símbolo. Puede ser un entero entre 1 y 25, o cualquier carácter colocado entre comillas.

`ps` Un entero que controla el tamaño en puntos de textos y símbolos.

`tck` Número que controla la longitud de las marcas en los ejes como fracción del mínimo entre el ancho y alto de la gráfica. Si `tck=1` se dibuja una rejilla.

Parámetros Gráficos

Si escribimos

```
> par()
```

obtenemos un listado de los valores vigentes de los parámetros. Como los cambios efectuados con la instrucción `par()` tienen carácter permanente, es útil guardar los valores anteriores de modo de poder restaurarlos.

```
> plot(cars)
> oldpar <- par(bg=0,bty='o',cex=1,col=
    'black',font=1,lty='solid',lwd=1,pch=1)
> par(bg=7, bty='7', cex=1.5, col='blue',
    font=2, lty='dashed', lwd=2, pch=3)
> plot(cars)
> plot(iris)
> par(oldpar)
> plot(cars)
```

Parámetros Gráficos

La instrucción `op <- par(no.readonly = TRUE)` guarda en `op` todos los valores por defecto de los parámetros que pueden ser modificados con la instrucción `par`. Luego, la instrucción `par(op)` restaura estos valores.

Outline

Ventanas Gráficas

Si deseamos conservar un gráfico es posible abrir una nueva ventana gráfica donde se harán los gráficos siguientes. Para ello usamos la instrucción `windows()`. También es posible enviar un gráfico a un archivo en lugar de presentarlo en pantalla. Para esto hay diversas instrucciones dependiendo del formato que se desée usar: `postscript()`, `pdf()`, `png()`, etc.

Ventanas Gráficas

El último dispositivo gráfico (ventana o archivo) abierto se transforma en el dispositivo activo para los gráficos subsiguientes. La función `dev.list()` muestra los dispositivos abiertos.

```
> windows(); pdf(); postscript()
> dev.list()
 windows  windows  pdf  postscript
       2        3    4           5
```

Los números que aparecen identifican a los dispositivos y deben usarse para cambiar el dispositivo activo.

Ventanas Gráficas

Para saber cuál está activo usamos la instrucción

```
> dev.cur()
> postscript
          5
```

Para cambiarlo usamos

```
> dev.set(3)
windows
      3
```

La función `dev.off()` se usa para cerrar los dispositivos gráficos. Si no lleva argumento, se cierra el dispositivo activo.

```
> dev.off(5)
windows
      3
> dev.off(4)
windows
      3
```

En cada caso la respuesta es el dispositivo que queda activo.

Ventanas Gráficas

Otra facilidad disponible en `R` para Windows que resulta de mucha utilidad es la posibilidad de guardar un registro de todos los gráficos hechos durante una sesión de trabajo. Al tener un dispositivo gráfico abierto, se activa un menú llamado `History` en el cual se puede seleccionar la opción `Recording` para grabar la sesión.

División de Ventanas Gráficas

Hay varias maneras de dividir una ventana gráfica para mostrar simultáneamente varios gráficos. Una posibilidad es modificar los parámetros gráficos usando la función `par()` con los argumentos `mfrow` o `mfcol`. Otra posibilidad es usar la instrucción `split.screen(c(m,n))` que divide la ventana en m filas y n columnas. Las partes pueden ser seleccionadas con `screen(r)` para $1 \leq r \leq m \cdot n$. `erase.screen()` borra el último gráfico.
Estas funciones son incompatibles con otras como `coplot` o `layout` y no deben usarse con dispositivos gráficos múltiples.

División de Ventanas Gráficas

Otra función que permite dividir la ventana gráfica es `layout`, que la divide en varias partes en las cuales las gráficas se dibujarán sucesivamente. El argumento es una matriz de enteros que indica el número de las divisiones. Por ejemplo, para dividir el dispositivo en cuatro partes podemos usar

```
> mat <- matrix(1:4,2,2)
> mat
     [,1] [,2]
[1,]    1    3
[2,]    2    4
> layout(mat)
```

Para ver la división que se creó podemos usar el comando `layout.show` cuyo argumento es el número de ventanas (4 en el ejemplo):

```
> layout.show(4)
```

División de Ventanas Gráficas

Los siguientes ejemplos muestran algunas de las posibilidades

```
> layout(matrix(1:6,3,2))
> layout.show(6)
> layout(matrix(1:6,2,3))
> layout.show(6)
> layout(matrix(1:6,3,2,byrow=TRUE))
> layout.show(6)
> (m <- matrix(c(1:3,3), 2, 2))
     [,1] [,2]
[1,]    1    3
[2,]    2    3
> layout(m)
> layout.show(3)
```

División de Ventanas Gráficas

Por defecto, `layout()` divide la ventana en partes iguales, pero esto puede modificarse con las opciones `widths` y `heights`. Por ejemplo,

```
> m <- matrix(1:4, 2, 2)
> layout(m, widths=c(1,3), heights=c(3,1))
> layout.show(4)
> m <- matrix(c(1,1,2,1), 2, 2)
> layout(m, widths=c(2,1), heights=c(1,2))
> layout.show(2)
```

Los números de la matriz pueden incluir ceros, lo que permite divisiones más complejas:

```
> m <- matrix(0:3, 2, 2)
> layout(m, widths=c(1,3), heights=c(1,3))
> layout.show(3)
```

División de Ventanas Gráficas

```
> x <- pmin(3, pmax(-3, rnorm(50)))
> y <- pmin(3, pmax(-3, rnorm(50)))
> xhist <- hist(x,breaks=seq(-3,3,0.5),
    plot=FALSE)
> yhist <- hist(y,breaks=seq(-3,3,0.5),
    plot=FALSE)
> top <- max(c(xhist$counts, yhist$counts))
> xrange <- c(-3,3)
> yrange <- c(-3,3)
```

División de Ventanas Gráficas

```
> nf <- layout(matrix(c(2,0,1,3),2,2,byrow=
    TRUE), c(3,1), c(1,3), TRUE)
> layout.show(nf)
> plot(x, y, xlim=xrange, ylim=yrange,
    xlab="", ylab="")
> barplot(xhist$counts, axes=FALSE,
    ylim=c(0, top), space=0)
> barplot(yhist$counts, axes=FALSE,
    xlim=c(0, top), space=0, horiz=TRUE)
```

Outline

locator

Esta función permite al usuario hacer click dentro de una gráfica y obtener como resultado las coordenadas del punto seleccionado. También es posible usarla para colocar símbolos en lugar donde se hace click o dibujar segmentos entre los puntos seleccionados. La sintaxis es `locator(n,type)`. Con esta instrucción `R` espera que el usuario seleccione n puntos en la gráfica activa. El argumento `type` permite dibujar en los puntos seleccionados y tiene la misma sintaxis que para los comandos gráficos de alto nivel. La opción por defecto es no dibujar nada. El resultado de `locator()` son las coordenadas de los puntos seleccionados como una lista con dos componentes `x` y `y`.

locator

```
> plot(cars)
> locator(3,type='n')
 $x
 [1]  7.292207  10.788086  13.930449
 $y
 [1]  70.27013  83.19953   80.27212
> locator(2,type='l')
 $x
 [1]  5.642467  17.779843
 $y
 [1]  67.09876  100.03215
> text(locator(1), 'Punto', adj=0)
```

identify

Esta función puede usarse para identificar datos en una gráfica. Se identifica el dato más cercano al click. La sintaxis es `identify(x, y, labels)`. El procedimiento es similar a `locator` pero en lugar de identificar las coordenadas del punto se identifica a través de `labels`. Si `labels` no está presente en la llamada a la función, se usan como etiquetas la fila en la cual están los datos en la matriz. Para terminar el proceso de identificación apretamos el botón derecho del ratón y seleccionamos `Stop`. Veamos un ejemplo con el mismo gráfico anterior.

```
> identify(cars$speed, cars$dist)
[1] 22 33 37
```

Outline

Gráficos `lattice`

El paquete `lattice` es básicamente una implementación en `R` de los gráficos Trellis desarrollados principalmente por W. S. Cleveland para S-PLUS. Este sistema permite la visualización de datos multivariados que es especialmente útil para explorar las relaciones o interacciones entre las variables. La idea fundamental es la de los gráficos múltiples condicionados, de modo que los gráficos bivariados se dividen en varios gráficos según el valor de una tercera variable. La función `coplot`, que ya estudiamos, tiene un enfoque similar, pero `lattice` es mucho más flexible y tiene mayores opciones.
La mayoría de las funciones en `lattice` usan como argumento una fórmula, así que vamos a revisar brevemente cómo se escriben las fórmulas en `R`.

Gráficos `lattice`

Ya hemos usado esta notación en algunas ocasiones. La situación más simple es cuando tenemos dos variables y queremos presentar una de ellas en función de la otra. Por ejemplo, si estamos trabajando con el cuadro de datos `iris`, podemos presentar `Petal.Width` como función de `Petal.Length`. Para esto escribimos

```
Petal.Width ~ Petal.Length
```

La variable respuesta se coloca en el lado izquierdo mientras que la variable predictora va del lado derecho. Cuando hay más de una variable predictora las cosas son algo más confusas. En particular, las operaciones matemáticas usuales no tienen el significado habitual. El uso principal de las fórmulas es la construcción de modelos estadísticos y muchas de las formulas posibles no tienen utilidad para los gráficos. El lector interesado puede obtener una descripción detallada de los distintos tipos de fórmula en el manual de `R`.

Gráficos `lattice`

Supongamos que las variables se llaman `Y, X1, X2`.

Fórmula	Significado
`Y ~ X1`	`Y` es función de `X1`
`Y ~ X1 + X2`	`Y` es función de `X1` y `X2`
`Y ~ X1 * X2`	`Y` es función de `X1, X2` y `X1*X2`
`Y ~ X1 * I((X2)^2)`	`Y` es función de `X1` y `X2^2`
`Y ~ X1 | X2`	`Y` es función de `X1` condicional a `X2`

Gráficos `lattice`

La siguiente tabla da una lista de las principales funciones en `lattice`.

Univariados

`assocplot(x)`	Gráficas de asociación
`barchart(y ~ x)`	Gráficos de barra
`bwplot(y ~ x)`	Diagramas de caja
`densityplot( ~ x)`	Gráficas de densidad
`dotplot(y ~ x)`	Gráficas de puntos
`histogram( ~ x)`	Histogramas
`qqmath( ~ x)`	Gráficas de cuantiles contra distintas dist.
`stripplot(y ~ x)`	Gráfica de dispersión unidimensional

Bivariados

`qq(y ~ x)`	Gráficas de cuantiles para comparación
`xyplot(y ~ x)`	Gráficos de dispersión

Gráficos `lattice`

Trivariados

`levelplot(z ~ x*y)`	Gráficas de nivel
`contourplot(z ~ x*y)`	Gráficos de contornos
`cloud(z ~ x*y)`	Gráficos de dispersión 3-D
`wireframe(z ~ x*y)`	Superficies 3-D

Varias variables

`splom( ~ x)`	Matriz de Gráficos de dispersión
`parallel( ~ x)`	Gráficos paralelos

Otros

`rfs`	Gráficas de residuales y valores ajustados
`tmd`	Gráficos de diferencia promedio de Tukey

Veamos algunos de estos gráficos en detalle. Para poder usar los comandos respectivos, es necesario cargar el paquete `lattice` con la instrucción `library(lattice)`.

barchart

Para ver como funciona esta instrucción vamos a usar el cuadro de datos `barley` que tiene los resultados de un experimento sobre rendimiento del cultivo de cebada. Para ver la estructura de este conjunto de datos escribimos

```
> str(barley)
'data.frame': 120 obs. of 4 variables:
$ yield : num 27.0 48.9 27.4 39.9 33.0 ...
$ variety: Factor w/ 10 levels "Svansota",
    "No. 462",..: 3 3 3 3 3 3 7 7 ...
$ year : Factor w/ 2 levels "1932","1931": 2
    2 2 2 2 2 2 2 2 2 ...
$ site : Factor w/ 6 levels "Grand
    Rapids",..: 3 6 4 5 1 2 3 6 4 5 ...
```

barchart

Para obtener mayor información sobre los valores de cada variable escribimos

```
> ?barley
```

La ayuda muestra el significado de cada variable. En particular hay

- 10 variedades de cebada: 'Svansota', 'No. 462', 'Manchuria', 'No. 475', 'Velvet', 'Peatland', 'Glabron', 'No. 457', 'Wisconsin No. 38', 'Trebi',
- dos años: 1931 y 1932 y
- 6 lugares: 'Grand Rapids', 'Duluth', 'University Farm', 'Morris', 'Crookston', 'Waseca'.

barchart

El experimento consistió en medir el rendimiento de 10 variedades de cebada en seis estaciones experimentales durante dos años. En resumen tenemos una variable numérica y tres factores.

Veamos como podemos obtener un diagrama de barras para los resultados de este experimento, presentando las 4 variables. Inicialmente escribimos

```
> barchart(yield~variety|site, data=barley,
  groups=year)
```

barchart

El resultado es una matriz de datos 2×3 en la cual cada gráfico corresponde a una estación experimental, que es la variable condicionante. En el eje *x* tenemos la variedad de cebada, en el eje *y* el rendimiento, que corresponden a las variables de la fórmula `yield ~ variety`, y para cada variedad tenemos dos barras lado a lado, una de color azul para el año 1931 y otra de color rojo para 1932, que corresponden a la opción `groups`.

barchart

Sin embargo, el resultado nos es satisfactorio. Por una parte no es posible identificar las distintas variedades, por otra, los gráficos lucen apilados y, finalmente, no es fácil comparar los resultados de las distintas estaciones. Para mejorar los resultados vamos a realizar algunas modificaciones.

```
> barchart(yield~variety|site, data=barley,
  groups=year, layout=c(1,6),
  ylab='Rendimiento')
```

barchart

El resultado es mejor. Las gráficas permiten una comparación fácil entre distintas estaciones experimentales y distintos años. Un inconveniente que aún vemos es que en algunos casos los nombres de las variedades se superponen. Para evitar esto hacemos una modificación adicional.

```
> barchart(yield variety|site, data=barley,
    groups=year,layout=c(1,6),ylab=
    'Rendimiento', scales=list(x=
    list(abbreviate=TRUE,minlength=5)))
```

barchart

Para ver una variación de esta gráfica escribimos las instrucciones

```
> barchart(yield  variety | site, data =
    barley, groups = year, layout = c(1,6),
    stack = TRUE,auto.key = list(points =
    FALSE, rectangles =TRUE, space =
    'right'), ylab = 'Rendimiento', scales =
    list(x = list(rot = 45)))
```

densityplot

Para producir una gráfica de la densidad estimada para la variable `Petal.Length` en todo el conjunto de datos escribimos

```
> densityplot(~ Petal.Length, data=iris)
```

El resultado es una gráfica de la densidad estimada que tiene en la parte inferior puntos que corresponden a los datos. En este caso el operador $\sim$ no tiene nada a la izquierda porque para este tipo de gráficos sólo hace falta un conjunto de datos: se trata de un gráfico univariado.

densityplot

Para añadir un condicionante, usamos el operador `|`, que puede leerse como 'condicional a' las variables que aparecen a su derecha. Por ejemplo:

```
> densityplot(~ Petal.Length | Species,
    data=iris)
```

Ahora tenemos una gráfica por especie. Podemos ver claramente las diferencias entre las distintas especies, en particular el hecho de que las distribuciones tienen modas distintas, lo cual produce la distribución multimodal para los datos en conjunto.

densityplot

Veamos otro ejemplo un poco más complicado. En cada panel vamos a superponer la gráfica de la densidad estimada y la de la densidad normal correspondiente. Para esto es necesario usar el argumento `panel` en la llamada a la instrucción, que define qué ponemos en cada gráfica. Los comandos son:

densityplot

```
  (n <- seq(5,45,5))
> (x <- rnorm(sum(n)))
> (y <- factor(rep(n,n), labels=
   paste('n=',n)))
> densityplot(~ x|y,
   panel = function(x,...)
     {
     panel.densityplot(x,
     col='DarkOliveGreen',...)
     panel.mathdensity(dmath=dnorm,
       args=list(mean=mean(x), sd=sd(x)),
       col='darkblue')
     })
```

densityplot

Las tres primeras instrucciones generan una muestra aleatoria de una distribución normal que esta dividida en grupos de tamaño $5, 10, 15, \ldots, 45$, clasificados según el factor `y`. Luego viene el llamado a la función `densityplot`, que produce una gráfica para cada grupo. Luego la opción `panel` tiene como argumento una función. En este ejemplo definimos una función que toma como argumento dos funciones predefinidas en `lattice`: `panel.densityplot`, que grafica la densidad empírica, y `panel.mathdensity` que grafica la densidad de una distribución normal con parámetros estimados a partir de cada muestra. La función `panel.densityplot` es la opción por defecto, si hubieramos escrito como instrucción `densityplot(~ x | y)` hubieramos obtenemos la misma gráfica pero sin las curvas azules.

histogram

Esta función permite dibujar histogramas condicionales y su uso es similar a la función anterior. Veamos dos ejemplos tomados de la ayuda de R. Para ello usamos el cuadro de datos `singer` que consiste de 235 observaciones de la altura de cantantes y su rango de voz en un `factor` clasificado en niveles `'Bass 2', 'Bass 1', 'Tenor 2', 'Tenor 1', 'Alto 2', 'Alto 1', 'Soprano 2', 'Soprano 1'`.

```
> histogram( ~ height | voice.part, data =
    singer, nint = 17, layout = c(2,4),
    aspect = 1, xlab = 'Altura (pulgadas)')
```

histogram

Obtenemos una matriz de histogramas con dos columnas y cuatro filas. La opción `nint` controla el número de clases mientras que `aspect` controla la relación entre los ejes de cada gráfico. Otra opción con los mismos datos es

```
> histogram( ~ height | voice.part, data =
    singer, xlab = 'Altura (pulgadas)',
    type = 'density', panel = function(x,
    ...) { panel.histogram(x, ...)
    panel.mathdensity(dmath = dnorm, col =
    'black', args = list(mean=mean(x),
    sd=sd(x)))} )
```

En esta gráfica la matriz es de 3×3, que es seleccionada por el programa automáticamente. En cada gráfica aparece no sólo el histograma sino también la densidad normal correspondiente a los parámetros estimados de cada muestra. Esto se logra a través de las instrucciones `panel.histogram` y `panel.mathdensity`.

Ejercicio

Ejercicio

1. Haga histogramas de la variable `hp` del archivo `mtcars`, condicionando primero por `cyl` y luego por cyl y `am`.
2. Repita el ejercicio anterior con la función `densityplot`

dotplot

En este tipo de gráfica se dibuja un punto por cada dato y son útiles cuando hay pocos datos que pueden ser clasificados según distintos factores. Veamos dos ejemplo. El primero usa el conjunto de datos `mtcars`, tomados de la revista *Motor Trend* y presenta el consumo de combustible y 10 aspectos más del diseño de 32 autos, modelos 1973-74.

```
> dotplot(mpg ~ disp, groups=as.factor(cyl),
  data=mtcars,auto.key=T)
> dotplot(mpg ~ disp|as.factor(cyl),
  groups=am, data=mtcars,auto.key=T)
```

Para el segundo ejemplo usamos el conjunto `barley`:

```
> dotplot(variety ~ yield | site, groups=
    year, data=barley, auto.key=
    list(columns=2))
```

bwplot

Esta es la versión *trellis* del boxplot. Veamos su uso con los datos `singer`

```
> bwplot(voice.part ~ height, data=singer,
    xlab='Altura (pulgadas)')
> bwplot(height ~ voice.part, data=singer,
    xlab='Altura (pulgadas)')
```

stripplot

Otra manera de ver los datos del conjunto `singer` es con el `stripplot`. Veamos el resultado de usar la instrucción directamente para graficar la altura clasificando los datos por el rango de voz:

```
> stripplot(voice.part ~ height, data =
    singer, xlab = 'Altura (pulgadas)')
```

Vemos que hay pocos puntos en la gráfica, teniendo en cuenta que el conjunto tiene datos de 235 cantantes. Lo que ocurre es que la altura está medida en enteros que corresponden a pulgadas, y por lo tanto hay muchos individuos con alturas repetidas. Al graficar, `R` superpone los valores. Para estos casos existe una instrucción útil que es `jitter`. Su efecto es añadir algo de ruido a la o las coordenadas del punto, de modo de moverlo un poco para que no coincidan.

stripplot

Veamos el efecto de esta instrucción, primero en el eje *x* y luego en ambos ejes.

```
> stripplot(voice.part ~ jitter(height),
    data = singer, xlab = 'Altura
    (pulgadas)')
> stripplot(voice.part ~ jitter(height),
    data = singer, jitter = TRUE,
    xlab = 'Altura (pulgadas)')
```

qqmath

Esta instrucción es la version *trellis* de `qqnorm` y `qqplot`, pero con mayor flexibilidad porque no sólo permite hacer gráficos condicionados, sino que es posible hacer gráficas de cuantiles respecto a distribuciones distintas a la normal, especificando la opción `distribution = q`*dist*, donde *dist* son las siglas que identifican la distribución en `R`, de acuerdo a la tabla 1.4.

qqmath

Como ejemplo vamos a generar diez muestras de tamaño 50 de la distribución exponencial y luego haremos las gráficas de cuantiles para cada una, comparando con la distribución exponencial.

```
> variables <- rexp(500, rate=0.5)
> grupos <- rep(1:10,rep(50,10))
> ejexp <- data.frame(cbind(variables,grupos))
> qqmath(~ejexp[,1] | as.factor(ejexp[,2]),
    distribution=qexp,xlab='Dist. Exp.',
    ylab='Muestra')
> qqmath(~ejexp[,1], distribution=qexp,
    xlab='Dist. Exp.',ylab='Muestra')
```

qqmath

Si queremos añadir rectas a los gráficos tenemos que usar la opción `panel` de la siguiente manera

```
> qqmath(~ejexp[,1] | as.factor(ejexp[,2]),
    distribution=qexp, xlab='Dist. Exp.',
    ylab='Muestra', prepanel=
    prepanel.qqmathline,
    panel=function(x,...){
        panel.qqmathline(x,...)
        panel.qqmath(x,...)})
> qqmath(~ejexp[,1], distribution=qexp,
    xlab='Dist. Exp.',ylab='Muestra',
    prepanel=prepanel.qqmathline,
    panel=function(x,...){
        panel.qqmathline(x,...)
        panel.qqmath(x,...)})
```

Ejercicio

Ejercicio

1. Cree una matriz `mt` que tenga en la primera columna 50 números generados de una distribución `t` con 3 grados de libertad, en la segunda columna 50 números generados de la normal típica y en la tercera 50 de χ^2_2.
2. Haga gráficas de cuantiles de cada una de estas muestras contra los cuantiles de la distribución t_2.
3. Repita comparando con los cuantiles de la distribución normal y la distribución χ^2_2.

xyplot

El gráfico más común es este, en el cual se representa la variable dependiente en el eje *y* y la variable independiente en el eje *x*. Usamos de nuevo el conjunto de datos `iris`.

```
> xyplot(Petal.Width ~ Petal.Length, data =
    iris, groups=Species, auto.key=T)
```

El resultado es un gráfica con leyenda en la parte superior y las distintas especies diferenciadas por color. Si en cambio modificamos la fórmula poniendo `Species` como condicionante

```
> xyplot(Petal.Width ~ Petal.Length |
    Species, data = iris, groups=Species,
    auto.key=T)
```

xyplot

Otra variación es la siguiente:

```
> xyplot(Petal.Width ~ Petal.Length, data =
    iris, groups=Species, panel =
    panel.superpose, type=c('p','smooth'),
    span=.75, auto.key=list(x=0.15, y=0.85))
```

En este caso la opción `panel.superpose` hace que las tres gráficas se superpongan y aparezcan juntas. La opción `type`, al igual pero los gráficos clásicos, indica el tipo de dibujo que se hará, pero ahora acepta valores vectoriales, de modo que se pueden presentar los gráficos de varias formas simultáneamente. En este caso los valores son `'p'`, que dibuja puntos y `'smooth'` que dibuja una curva regular cuyo grado de regularidad está dado por `span`. El valor de `auto.key` indica en qué lugar del gráfico se coloca la leyenda.

levelplot

Esta función permite graficar una función tridimensional a través de los niveles, que se representan el plano por regiones con distintos colores. Veamos un ejemplo tomado de la ayuda.

```
> x <- seq(pi/4, 5 * pi, length = 100)
> y <- seq(pi/4, 5 * pi, length = 100)
> r <- as.vector(sqrt(outer(x^2, y^2, '+')))
> grid <- expand.grid(x=x, y=y)
> grid$z <- cos(r^2) * exp(-r/(pi^3))
> levelplot(z~x*y, grid, cuts = 50, scales=
    list(log='e'), xlab='', ylab='', main =
    'Funcion Extraña', sub='con escalas
    logaritmicas', region = TRUE)
```

contourplot

Esta función es la versión *trellis* de `contour`.

```
> contourplot(volcano, at =
    seq(floor(min(volcano)/10)*10,
    ceiling(max(volcano)/10)*10, by=10),
    main='Curvas de Nivel, Volcan Maunga
    Whau, Auckland', sub='Intervalo entre
    curvas 10 m.', region=T, col.regions =
    terrain.colors(100))
```

cloud

Este comando permite dibujar una nube de puntos en tres dimensiones. Veamos algunos ejemplos,

```
> cloud(Sepal.Length ~ Petal.Length * Petal.Width,
    groups=Species,data=iris)
> cloud(Sepal.Length ~ Petal.Length * Petal.Width,
    groups = Species, data = iris, screen =
    list(x = -90, y = 70))
> cloud(Sepal.Length ~ Petal.Length * Petal.Width,
    groups = Species, data = iris, screen =
    list(x = -90, y = 70), distance = .4, zoom = .6)
> cloud(Sepal.Length ~ Petal.Length * Petal.Width |
    Species, data = iris, screen =
    list(x = -90, y = 70), distance = .4, zoom = .6)
```

wireframe

Sirve para dibujar superficies en tres dimensiones usando una rejilla. Veamos un par de ejemplos tomados de la ayuda de R.

```
> wireframe(volcano, shade = TRUE, aspect =
    c(61/87, 0.4), light.source = c(10,0,10))
> g <- expand.grid(x = 1:10, y = 5:15, gr = 1:2)
> g$z <- log((g$x^$gr + g$y^2) * g$gr)
> wireframe(z ~ x * y, data = g, groups = gr,
    scales = list(arrows = FALSE), drape = TRUE,
    colorkey = TRUE, screen = list(z=30, x=-60))
```

splom

El nombre de esta función es un acrónimo de *Scatterplot matrix*, es decir, matriz de gráficos de dispersión. Veamos el efecto de esta función usando de nuevo el conjunto de datos `iris`

```
> splom(~ iris[1:4], groups = Species,
    data = iris, xlab = ", panel =
    panel.superpose, auto.key =
    list(columns=3))
```

Al argumento ahora es una matriz con los valores de las cuatro variables numéricas del archivo `iris`. El resultado es una matriz de gráficas similar a la que se obtiene con la instrucción `pairs`. Algunas de las opciones que aparecen ya han sido usadas, la única nueva es `columns=3` para `auto.key`, que pone la leyenda en tres columnas.

splom

La figura que obtuvimos la hubieramos podido obtener usando la función `pairs` y los comandos que vimos anteriormente, pero esa función no puede realizar gráficos condicionales. Veamos un ejemplo con `splom`

```
> splom(~iris[1:3] | Species, data = iris,
    pscales=0, varnames = c('Sepal\nLength',
    'Sepal\nWidth', 'Petal\nLength'))
```

Las opciones nuevas que hemos usado en esta ocasión son `pscales=0` que elimina la marcas en los ejes, y redefinimos los nombres de las variables con `varnames` para aparecieran en dos líneas. Esto mejora la visibilidad de los gráficos.

splom

Una variación es la siguiente:

```
> splom(~iris[1:3] | Species, data = iris,
    layout = c(2,2), pscales = 0,
    varnames = c('Sepal\nLength',
    'Sepal\nWidth', 'Petal\nLength'),
    page=function(...)ltext(x =
    seq(.65,.8,len=4), y=seq(.9,.6,len=4),
    lab = c('Tres','Variedades','de','Iris'),
    cex=2))
```

parallel

Esta instrucción genera gráficos que son útiles para el análisis exploratorio de datos multivariados Las variables se colocan en un eje y los valores observados se colocan en el otro eje. Las variables se transforman a una escala similar normalizándolas. Para los datos `iris` tenemos

```
> parallel(~ iris[, 1:4] | Species, data =
    iris, layout = c(3,1))
```

Outline

Introducción

Un aspecto interesante de R es que no sólo es un paquete estadístico, con numerosas funciones y rutinas incorporadas, sino que también es un lenguaje de programación que nos permite crear nuevas funciones. Estas funciones tienen el mismo carácter que las funciones residentes de R. De hecho, muchas de las funciones residentes están programadas en R, y por lo tanto es posible modificarlas para crear nuevas funciones. A continuación veremos una introducción de las principales características de R como lenguaje de programación.

Funciones Estándar

Hay varias funciones para convertir númerso decimales a enteros:

- `round` Sintaxis: `round(x,n)`, donde `n` es el número de decimales. Números negativos redondean a potencias positivas de 10.
- `trunc` Redondea al entero más cercano en la dirección al cero.
- `floor` Redondea al entero más cercano por debajo.
- `ceiling` Redondea al entero más cercano por arriba.

```
> round(12.345)
[1] 12
> round(12.345,2)
[1] 12.35
> round(123456,-3)
[1] 123000
```

Funciones Estándar

```
> trunc(12.345)
[1] 12
> floor(12.345)
[1] 12
> ceiling(12.345)
[1] 13
> trunc(-12.345)
[1] -12
> floor(-12.345)
[1] -13
> ceiling(-12.345)
[1] -12
```

Funciones Estándar

Las funciones comunes están disponibles:

`abs, sign, log, log10, sqrt, exp, sin, cos, tan, asin, acos, atan, sinh, cosh, tanh, gamma, lgamma`

`sum, prod` dan la suma y el producto de las componentes de un vector. Las versiones acumuladas de estas operaciones son `cumsum, cumprod`.

`max, min` dan el mayor y menor valor de las componentes de un vector. Las versiones acumuladas de estas operaciones son `cummax, cummin`.

Las funciones `pmax(x1,x2,..., xn), pmin(x1,x2,..., xn)` tienen como argumentos vectores y calculan el máximo (o mínimo) componente a componente. El resultado es un vector de longitud igual a la del vector más largo y los vectores más pequeños se reciclan.

Funciones Estándar

`range(x)` da como resultado `(min(x), max(x))` para un vector `x`.

`sort(x)` ordena el vector `x` de manera creciente.

`rev(x)` coloca las componentes de un vector o lista en orden inverso.

`duplicated` produce un vector lógico con valor `T` cuando la componente de un vector es un valor repetido.

`unique` elimina los valores duplicados.

`union, intersect, setdiff, is.element` ejecutan las operaciones de conjunto $A \cup B$, $A \cap B$, $A \setminus B$ y $x \in A$. Sus argumentos pueden ser vectores de cualquier tipo pero, como conjuntos, no debe haber elementos repetidos.

Funciones Estándar

La función `%/%` denota la división entera. El resultado de `a%/%b` es `floor(a/b)`. Por ejemplo, la división entera de 5 entre 2 da como resultado 2:

```
> 5%/% 2
[1] 2
```

El resto de la división entera se obtiene con la operación `%%`, es decir, `a%%b = a-(a%\%b)*b`. En el ejemplo anterior, el resultado de esta operación entre 5 y 2 es el resto de la división, que es 1:

```
> 5%%2
[1] 1
```

Funciones Estándar

Estas operaciones pueden usarse, por ejemplo, para identificar los múltiplos de cierto entero `n`, ya que en este caso el resto de la división entera debe ser 0.
Por ejemplo, si `x` es un vector de enteros, los múltiplos de 3 son aquellas componentes que satisfacen la condición de que el resto es 0 al realizar la división entera por tres, es decir `x%%3 == 0`:

```
> x <- 1:10
> x[x%%3==0]
[1] 3 6 9
```

Funciones Estándar

Tres funciones que resultan útiles para manejar cuadros de datos y obtener subconjuntos de ellas son `subset`, `transform` y `split`.
`subset` permite obtener un subconjunto de valores que satisfacen una condición dada. Por ejemplo, para extraer un cuadro de datos que contenga los valores para la especie `setosa` del archivo `iris` podemos escribir

```
> iris.setosa <- subset(iris, Species ==
    'setosa')
```

y el archivo `iris.setosa` generado tiene los 50 valores de las variables que corresponden a esta especie.

Funciones Estándar

La función `transform` permite añadir variables al cuadro de datos que se obtiene haciendo transformaciones de las variables ya presentes. Por ejemplo, vamos a añadir al cuadro de datos que acabamos de formar una nueva columna llamada `PLC` que va a contener el resultado de restar la media a los valores de `Sepal.Length`:

```
> iris.setosa <- transform(iris.setosa,PLC =
    (Petal.Length-mean(Petal.Length)))
```

Funciones Estándar

La función `split` divide a un vector o un cuadro de datos según los valores de una variable. El resultado es una lista que tiene tantas componentes como valores tiene la variable que se usa para hacer la división. Por ejemplo, la instrucción

```
> split(iris$Sepal.Length,iris$Species)
```

produce una lista con tres componentes, una para cada especie, y en cada componente hay un vector con los valores de `Sepal.Length` para esa especie. La instrucción

```
> iris.lista <- split(iris[,1:4],
    iris$Species)
```

Produce una lista también con tres componentes, pero cada componente es ahora un cuadro de datos que tiene la información sobre las cuatro variables numéricas del conjunto que corresponden a esa especie.

Expresiones Lógicas

Los vectores de valores lógicos se genera usualmente a través de expresiones lógicas. Los operadores lógicos son

$<$	Menor que
$<=$	Menor o igual que
$>$	Mayor que
$>=$	Mayor o igual que
$==$	Igual a
$!=$	Diferente de

Expresiones Lógicas

Si `A` y `B` son expresiones lógicas con valores vectoriales, entonces

- `A & B` es su intersección,
- `A | B` es su unión (`A` o `B` o ambos),
- `xor(A,B)` es su unión excluyente (`A` o `B` pero no ambos),
- `!A` es la negación de `A`.

Estas operaciones se efectuan de manera separada para cada componente de los vectores.

Expresiones Lógicas

```
> x <- 1:6
> x > 2 & x <= 4
[1] FALSE FALSE TRUE TRUE FALSE FALSE
> x > 2 | x <= 4
[1] TRUE TRUE TRUE TRUE TRUE TRUE
> x <= 2 | x > 4
[1] TRUE TRUE FALSE FALSE TRUE TRUE
> xor(x <= 2 , x > 4)
[1] TRUE TRUE FALSE FALSE TRUE TRUE
> xor(x > 2 , x <= 4)
[1] TRUE TRUE FALSE FALSE TRUE TRUE
> ! x<2
[1] FALSE TRUE TRUE TRUE TRUE TRUE
```

Expresiones Lógicas

Es posible construir tablas de verdad usando la función `outer` para estas operaciones

```
> y <- c(NA, TRUE, FALSE)
> names(y) <- as.character(y)
> outer(y,y,'&')
          <NA>   TRUE  FALSE
 <NA>       NA     NA  FALSE
 TRUE       NA   TRUE  FALSE
 FALSE   FALSE  FALSE  FALSE
```

Expresiones Lógicas

```
> outer(y,y,'|')
       <NA>  TRUE  FALSE
<NA>     NA  TRUE     NA
TRUE   TRUE  TRUE   TRUE
FALSE    NA  TRUE  FALSE
> outer(y,y,'xor')
       <NA>   TRUE  FALSE
<NA>     NA     NA     NA
TRUE     NA  FALSE   TRUE
FALSE    NA   TRUE  FALSE
```

Expresiones Lógicas

La función `all.equal` permite comparar dos objetos en `R` y determinar si son 'casi iguales'. La tolerancia se puede fijar con el argumento `tolerance`, que toma un valor por defecto que depende de la precisión en la máquina (del orden de 1.5×10^{-8} en la mía). Veamos un par de ejemplos de su uso:

```
> 355/113
[1] 3.141593
> all.equal(pi,355/113)
[1] "Mean relative difference: 8.491368e-08"
> all.equal(pi,355/113,tolerance=0.01)
[1] TRUE
> all.equal(pi,355/113,tolerance=0.0001)
[1] TRUE
```

Ejercicio

Ejercicio

1. Genere un vector de 100 números aleatorios tomados a partir de la distribución Gaussiana típica. Escriba una instrucción que genere un vector lógico que indique los valores de la muestra tienen valor absoluto mayor que 1.96. Luego escriba una instrucción que cuente cuantos elementos satisfacen esta condición.
2. Genere un vector de 20 enteros tomados al azar de los dígitos $\{0, 1, 2, \ldots, 9\}$. Escriba una instrucción que indique cuando un elemento está en el conjunto {2,3,6,7,8}.
3. Para el vector que generó en la primera pregunta, cuente cuántos elementos son iguales a 0 con una tolerancia de una centesima.

Datos Faltantes

Para datos faltantes en un vector o cualquier otro objeto de `R` se usa el símbolo `NA`, que viene de las iniciales de *not available*, y es importante saber como reacciona `R` al encontrar un dato faltante.
Cualquier operación aritmética que incluya valores `NA`, da como resultado `NA`. Por ejemplo

Datos Faltantes

```
> aa <- c(1:3, ,9)
Aviso: an element is empty and has been omitted
    the part of the args list of 'c' being
    evaluated was:
    (, 9)
> aa
[1] 1 2 3 9
> aa <- c(1:3,NA,9)
> aa
[1] 1 2 3 NA 9
> sum(aa)
[1] NA
> max(aa)
[1] NA
> 2*aa
[1] 2 4 6 NA 18
```

Datos Faltantes

Lo mismo ocurre con las relaciones `<`, `<=`, `>`, `>=`, `==`, `!=`. En particular, la expresión `x == NA` tiene como resultado `NA`

```
> aa == NA
[1] NA NA NA NA NA
```

Sin embargo, el uso de los operadores lógicos de comparación usados con un vector que incluye valores `NA` da los siguientes resultados.

```
> aa>2
[1] FALSE FALSE TRUE NA TRUE
> aa[aa>2]
[1] 3 NA 9
```

Datos Faltantes

Para poder identificar las componentes de un vector (o de cualquier otro objeto) que son `NA` hay que usar la función `is.na()`, que da como resultado un vector lógico con valor `TRUE` cuando la componente es `NA`:

```
> is.na(aa)
[1] FALSE FALSE FALSE TRUE FALSE
```

Con esta función podemos extraer las componentes que son `NA`:

```
> aa[is.na(aa)]
[1] NA
```

o las que no lo son:

```
> aa[!is.na(aa)]
[1] 1 2 3 9
```

Datos Faltantes

También podemos usar esta función para asignar el valor `NA` a una componente de un vector:

```
> (bb <- 1:9)
[1] 1 2 3 4 5 6 7 8 9
> is.na(bb)[6] <- T
> bb
[1] 1 2 3 4 5 NA 7 8 9
```

Datos Faltantes

Hay un segundo tipo de 'valores faltantes' que son el resultado de hacer un cálculo numérico cuyo resultado no es un número, y que se designan por las letras `NaN` (*Not a Number*). Por ejemplo

```
> 0/0
[1] NaN
> 1/0 + log(0)
[1] NaN
> is.na(0/0)
[1] TRUE
```

Datos Faltantes

Vemos que la función `is.na` identifica a `NaN` como un dato faltante. Para distinguir tenemos la función `is.nan()`. También existen las funciones `is.finite` y `is.infinite`. Veamos el efecto de estas funciones con un vector que tenga componentes de distintos tipos.

```
> (cc <- c(1, 1/0, 0/0, NA))
[1] 1 Inf NaN NA
> is.na(cc)
[1] FALSE FALSE TRUE TRUE
> is.finite(cc)
[1] TRUE FALSE FALSE FALSE
> is.infinite(cc)
[1] FALSE TRUE FALSE FALSE
> is.nan(cc)
[1] FALSE FALSE TRUE FALSE
```

Manejo de Caracteres

La función `nchar` da, en forma de vector, el número de caracteres en cada elemento de un vector de caracteres:

```
  > (tt <- c('esta es una prueba', 'otra', 'y
      otra mas'))
[1] "esta es una prueba" "otra" "y otra mas"
> nchar(tt)
[1] 18 4 10
```

Manejo de Caracteres

La función `paste` usa un número arbitrario de argumentos y los une, elemento por elemento, produciendo un vector de caracteres. Por ejemplo

```
  > paste(c('Altura','Peso'), rep(c(1,2), c(2,2)))
[1] "Altura 1" "Peso 1" "Altura 2" "Peso 2"
```

Por defecto los elementos que se unen quedan separados por un espacio. Para evitar esto se puede usar el argumento `sep='algo'`, donde `algo` es lo que se coloca entre los elementos, que puede ser incluso un espacio vacío. Por ejemplo,

```
  > paste(c('X','Y'),1:4, sep=' ')
[1] "X1" "Y2" "X3" "Y4"
```

Manejo de Caracteres

El argumento `collapse`, permite que el resultado se concatene en una expresión larga, ya que permite determinar que caracter se coloca entre los componentes al hacer la concatenación. Por defecto toma el valor `NULL` y en consecuencia no se hace esta concatenación.

```
  > paste(c('X','Y'),1:4, sep="",collapse='+')
[1] "X1+Y2+X3+Y4"
```

Ejercicio

Ejercicio

1. Defina el vector $z < -1/(-2:2)$. Halle el máximo de los valores numéricos de la expresión `exp(-z)*z`.
2. Escriba las instrucciones para obtener un vector con las siguientes componentes

```
[1] "Paciente 1 tiene altura igual a"
[2] "Paciente 1 tiene peso igual a"
[3] "Paciente 2 tiene altura igual a"
[4] "Paciente 2 tiene peso igual a"
```

Outline

Scripts

A través de una ventana `script` es posible desarrollar un programa que luego puede ser ejecutado usando la función `source()`. Por ejemplo, podemos abrir una nueva ventana `script` y escribir las siguientes instrucciones

```
> a1 <- rexp(100) ♯ Simulamos 100 exponenciales
> a2 <- rexp(100) ♯ Simulamos 100 exponenciales
b <- rnorm(100) ♯ Simulamos 100 normales
x1 <- a1 + b
x2 <- a2 + b
r <- cor(x1,x2)
print(r)
```

Scripts

Ahora seleccionamos el menú `File` en `R` y guardamos el `script` en el directorio de trabajo con el nombre `prueba.R`. Para correrlo escribimos:

```
> source('prueba.R')
[1] 0.6422564
```

En este caso no solo se creó el objeto `r` sino también `a1, a2, b, x1` y `x2`.

Ejercicio

Ejercicio

1. Escriba en una nueva ventana *script* instrucciones que hagan los siguiente:
   - 1.1 Genere dos números al azar `a` y `b` con distribución uniforme en [0,2].
   - 1.2 Genere dos muestras `x` y `y` de tamaño 50 de la distribución normal centrada de desviación típica `a` y `b`, respectivamente.
   - 1.3 Calcule la correlación entre `x` y `y`.
   - 1.4 Haga una gráfica de `x` contra `y`.
2. Guarde el *script* con el nombre `ejer1.R` y luego ejecutelo desde la cónsola.