Curso básico de R

Francesc Carmona

fcarmona@ub.edu

15 de febrero de 2007

El objetivo principal de este curso es proporcionar los elementos básicos para empezar a trabajar con el lenguaje de programación R en el ámbito de la Estadística.

Índice

Este curso está basado muy especialmente en el documento
Introducción al uso y programación del sistema estadístico R
de Ramón Díaz-Uriarte de la Unidad de Bioinformática del CNIO.

1. Introducción

1.1. R y S

R, también conocido como "GNU S", es un entorno y un lenguaje para el cálculo estadístico y la generación de gráficos. R implementa un dialecto del premiado lenguaje S, desarrollado en los Laboratorios Bell por John Chambers et al.

Para los no iniciados diremos que R provee un acceso relativamente sencillo a una amplia variedad de técnicas estadísticas y gráficas.

Para los usuarios avanzados se ofrece un lenguaje de programación completo con el que añadir nuevas técnicas mediante la definición de funciones.

"S ha modificado para siempre la forma en la que las personas analizan, visualizan y manipulan los datos" (Association of Computer Machinery Software System Award 1998 a John Chambers).

Actualmente, S y R son los dos lenguajes más utilizados en investigación en estadística.

Los grandes atractivos de R/S son:

- La capacidad de combinar, sin fisuras, análisis "preempaquetados" (ej., una regresión logística) con análisis ad-hoc, específicos para una situación: capacidad de manipular y modificar datos y funciones.
- Los gráficos de alta calidad: visualización de datos y producción de gráficos para *papers*.
- La comunidad de R es muy dinámica, con gran crecimiento del número de paquetes, e integrada por estadísticos de gran renombre (ej., J. Chambers, L. Terney, B. Ripley, D. Bates, etc.).
- Hay extensiones específicas a nuevas áreas como bioinformática, geoestadística y modelos gráficos.
- Es un lenguaje orientado a objetos.
- Se parece a Matlab y a Octave, y su sintaxis recuerda a C/C++.

R es la implementación GNU de S.
Filosofía y objetivos del proyecto GNU: `http://www.gnu.org`

R se distribuye con licencia GNU GPL o General Public
`http://www.gnu.org/licenses/gpl.html`

La GPL no pone ninguna restricción al uso de R. Restringe su distribución (ha de ser GPL).

R se obtiene por 0 euros en `http://cran.es.r-project.org`

S-PLUS es un programa comercial.

1.2. Obtención y instalación de R

Depende del sistema operativo, pero todo se puede encontrar en `http://cran.es.r-project.org/bin`

Windows: se puede obtener un ejecutable desde

`http://cran.es.r-project.org/bin/windows/base` por ejemplo, `http://cran.es.r-project.org/bin/windows/base/R-2.4.1-win32.exe`

Al ejecutar el archivo se instalará el sistema base y los paquetes recomendados.

GNU/Linux: (dos opciones)

1. Obtener el `R-x.y.z.tar.gz` y compilar desde las fuentes. También bajar los paquetes adicionales y instalarlos. (Buena forma de comprobar que el sistema tiene development tools).
2. Obtener binarios (ej., `*.deb` para Debian, `*.rpm` para RedHat, SuSE, Mandrake).

1.3. Paquetes

R consta de un sistema base y de paquetes adicionales que extienden su funcionalidad.

`http://cran.es.r-project.org/src/contrib/PACKAGES.html`

Tipos de paquetes:

- Los que forman parte del **sistema base** (ej. ctest).
- Los que no son parte del sistema base, pero son **recommended** (ej., survival, nlme).

  En GNU/Linux y Windows ya forman parte de la distribución estándar.
- Otros paquetes; ej., UsingR, foreing, los paquetes de Bioconductor (como multtest, etc.).

  Éstos se han de seleccionar y instalar individualmente. Más adelante veremos cómo.

1.4. Documentación

Los manuales de R, incluidos en todas las instalaciones son:

- *An introduction to R.* (De lectura obligatoria)
- *Writing R extensions.*
- *R data import/export.*
- *The R language definition.*
- *R installation and administration.*

Documentación general:

- *R para principiantes*, de E. Paradis. En `http://cran.es.r-project.org/doc/contrib/Paradis-rdebuts_en.pdf` o `http://cran.es.r-project.org/doc/contrib/rdebuts_es.pdf`.
- *A guide for the unwilling S user*, de P. Burns. En `http://cran.es.r-project.org/doc/contrib/Burns-unwilling_S.pdf` o `http://www.burns-stat.com/pages/tutorials.html`. ¡Sólo 8 páginas!
- *R Graphics*, de Paul Murrell.
- *S Programming*, de W. Venables y B. Ripley.

  Ver también `http://www.stats.ox.ac.uk/pub/MASS3/Sprog`.

Estadística:

- simpleR *Using R for Introductory Statistics* de John Verzani en `http://www.math.csi.cuny.edu/Statistics/R/simpleR/index.html`
- *Introductory statistics with R* de P. Dalgaard.
- *An R and S-PLUS companion to applied regression* de J. Fox.
- *Modern applied statistics with S, 4th ed.* de W. Venables y B. Ripley.

  Ver también `http://www.stats.ox.ac.uk/pub/MASS4`.
- *Practical regression and ANOVA using R* de J. Faraway, en `http://cran.r-project.org/other-docs.html` o `http://www.stat.lsa.umich.edu/~faraway/book/`.
- *Estadística Aplicada con S-PLUS* de M. Dolores Ugarte y Ana F. Militino.
- Otros documentos en `http://cran.es.r-project.org/other-docs.html`

- *S-PLUS 6.0 for Unix. Guide to statistics.* Vol. I & II. En `http://www.insightful.com/support/documentation.asp?DID=3`
- *Mixed-effects models in S and S-PLUS* de J. Pinheiro y D. Bates.
- *Regression modelling strategies* de F. Harrell.
- *Modelos Lineales* de F. Carmona.
- "Siteçon documentación sobre análisis para datos categóricos ("site"para el libro de A. Agresti *Categorical data analysis*). `http://www.stat.ufl.edu/~aa/cda/cda.html`
- *Modeling survival data: extending the Cox model* de T. M. Therenau y P. M. Grambsch.
- Documentos varios en la página de J. Fox `http://cran.r-project.org/doc/contrib/Fox-Companion/appendix.html`

1.5. Ayudas sobre R

Hay diversas formas de ayuda:

- Ayuda incluida con el programa (lo veremos más adelante).
- FAQ: `http://cran.es.r-project.org/faqs.html`
- **Rtips** de Paul Johnson `http://pj.freefaculty.org/R/Rtips.html`
- R Help Center `http://www.stat.ucl.ac.be/ISdidactique/Rhelp/`
- Las **e-mail lists** son "consultables". Ver `http://cran.es.r-project.org/search.html` y `http://finzi.psych.upenn.edu/search.html`.

  Permiten hacer las búsquedas no sólo sobre las listas de e-mail sino también sobre la documentación (incluyendo paquetes).

2. Empezamos

2.1. Inicio de una sesión en R

Windows:

- Hacer doble-click en el icono. Se abrirá "Rgui".
- Desde una "ventana del sistema" ejecutar Rterm o desde Inicio-Ejecutar...
  `C:\Archivos de programa\R\R-2.4.1\bin\Rterm.exe`
  parecido a R en Unix o Linux.
- Iniciar R desde Tinn-R, XEmacs o un editor apropiado.

GNU/Linux:

- Escribir "R" en una shell.
- Iniciar R desde (X)Emacs (M-X R).

En cualquier caso, se puede adaptar el inicio de una sesión de R (ej., qué paquetes se cargan, mensajes, etc.). Ver sección 10.8 en *An introduction to R*.

2.2. Una primera sesión

```
> rnorm(5) # 5 números aleatorios de una normal (mean= 0, sd = 1)
> ##  "#" indica el principio de un comentario
> ## Los números se calculan y se muestran (print)
>
> x <- rnorm(5) # asignamos unos números a un objeto (un vector) x
> summary(x) ## muestra un resumen de x (un summary "inteligente")
>
> ## o también:
> w <- summary(x)
> w
> print(w) # escribir w y print(w) produce el mismo resultado
>
> ## summary(x) TAMBIÉN es un objeto.
```

(virtually) "everything is an object"

2.3. R como calculadora

```
> 2+2
> sqrt(10)
> 2*3*4*5
> # Intereses sobre 1000 euros
> 1000*(1+0.075)^5 - 1000
> # R conoce pi
> pi
> # Circunferencia de la Tierra en el Ecuador en km
> 2*pi*6378
> # Convertimos ángulos a radianes y luego calculamos el seno
> sin(c(0,30,45,60,90)*pi/180)
```

2.4. Ayuda dentro del programa

- `?rnorm`
- `help.start()`
- `?help.search`
- `help.search("normal")`
- `?apropos`
- `apropos("normal")`
- `?demo`
- `demo(graphics); demo(persp); demo(lm.glm)`

¡Cuidado!

```
> ?if # mal
> help("if")
```

2.5. Instalación de paquetes adicionales

Depende del sistema operativo:

Windows:

- Desde el menú `Packages->Install package(s)...`
  Primero nos pide seleccionar el "CRAN mirror".
- Desde R, con `install.packages()` como en GNU/Linux.
- Desde una "ventana del sistema" o desde Inicio-Ejecutar...
  `Rcmd INSTALL paquete`
- Desde la *interface* de XEmacs.

GNU/Linux:

- `R CMD INSTALL paquete-x.y.z.tar.gz`
  Permite instalar aunque uno no sea root (especificar el dir).
- Más cómodo, desde R,
  `install.packages()`, `update.packages()`, etc.
  También permiten instalar si no eres root (especificar lib.loc).

2.6. Usar R desde un editor

¿Por qué usar R desde un editor como Tinn-R, R-WinEdt o XEmacs?

- El uso de scripts y el mantenimiento del código ordenado y comentado es una "buena práctica estadística" (ver también loadhistory, savehistory).
- Colores de sintaxis, completa paréntesis, etc.
- Una interface común para otros paquetes estadísticos (ej., SAS, XLispStat, Arc, etc.), numéricos (ej., Octave) o procesador de texto (LaTeX).
- Una buena elección: Tinn-R.
- También **WinEdt** tiene una configuración adaptada: R-WinEdt.
- **(X)Emacs** es MUCHO más que un editor..., aunque no es muy conocido para los usuarios de Windows (mejor con las modificaciones de J. Fox).

2.7. Un ejemplo

Sea X una v.a. con distribución exponencial de parámetro α y $X_1, X_2, \ldots, X_n$ una muestra aleatoria simple. Se sabe que la distribución de $Z = n \cdot \text{mín}\{X_1, X_2, \ldots, X_n\}$ es exponencial de parámetro α.

```
> alpha <- 0.01; n <- 50; m <- 1000
> datos <- matrix(rexp(n * m, alpha), ncol=n)
> fz <- function(x) n*min(x)
> z <- apply(datos,1,fz)
> mean(z) # debe ser 1/alpha=100
>
> hist(z,freq=F)
> points(dexp(0:600,alpha),type="l")
>
> ks.test(z,"pexp",alpha)
```

3. Objetos en R

3.1. ¿Qué objetos?

Casi todo en R es un objeto, incluyendo funciones y estructuras de datos.

- Para saber los objetos que tenemos en el espacio de trabajo utilizaremos `ls()`.
- Escribir el nombre de un objeto muestra su contenido: `mean`.
- Para guardar el contenido del espacio de trabajo se pueden utilizar las funciones `save.image()` y `save(<objetos>,file="nombre.RData")`
- Para acceder a objetos de la carpeta de trabajo (o del camino que especifiquemos) se pueden adjuntar:

```
> attach("misdatos.RData")
> ls(pos=2) # segunda posición en la ‘‘search list’’
```

3.2. Tipos de objetos

objetos del lenguaje:

- llamadas
- expresiones
- nombres

expresiones: colecciones de expresiones correctas no evaluadas

funciones:

Constan de

- lista de argumentos
- código
- entorno

sin objeto: NULL

Objetos para los datos:

vector: colección ordenada de elementos del mismo tipo.

```
> x <- c(1, 2, 3); y <- c("a", "b", "c")
> z <- c(TRUE, TRUE, FALSE)
```

array: generalización multidimensional del vector. Elementos del mismo tipo.

data frame: como el array, pero con columnas de diferentes tipos. Es el objeto más habitual para los datos experimentales.

```
> dades <- data.frame(ID=c("gen0", "genB", "genZ"),
+ subj1 = c(10, 25, 33), subj2 = c(NA, 34, 15),
+ oncogen = c(TRUE, TRUE, FALSE),
+ loc = c(1,30, 125))
```

factor: tipo de vector para datos cualitativos.

```
> x <- factor(c(1, 2, 2, 1, 1, 2, 1, 2, 1))
```

list: vector generalizado.

Cada lista está formada por componentes que pueden ser otras listas. Cada componente puede ser de distinto tipo. Son contenedores generales de datos. Muy flexibles, pero sin estructura. Muchas funciones devuelven una lista o conjunto de resultados de distinta longitud y distinto tipo.

```
> una.lista <- c(un.vector = 1:10,
+                una.palabra = "hola",
+                una.matriz = matrix(rnorm(20), ncol = 5),
+                lista2 = c(a = 5,
+                           b = factor(c("a", "b"))))
```

3.3. Atributos de los objetos

Modo: Tipo básico en un vector o array: lógico, entero, real, carácter,... `mode`

Tipo: de los vectores o arrays: double,... `typeof`

Nombres: etiquetas de los elementos individuales de un vector o lista: `names`

Dimensiones: de los arrays (alguna puede ser cero): `dim`

Dimnames: nombres de las dimensiones de los arrays: `dimnames`

Clase: vector alfanumérico con la lista de las clases del objeto: `class`

Otros: atributos de una serie temporal.

Ejemplos:

```
> x <- 1:15; length(x)
> y <- matrix(5, nrow = 3, ncol = 4); dim(y)
> is.vector(x); is.vector(y); is.array(x)
> x1 <- 1:5; x2 <- c(1, 2, 3, 4, 5); x3 <- "patata"
> typeof(x1); typeof(x2); typeof(x3)
> mode(x); mode(y); z <- c(TRUE, FALSE); mode(z)
> attributes(y)
> w <- list(a = 1:3, b = 5); attributes(w)
> y <- as.data.frame(y); attributes(y)
> f1 <- function(x) {return(2 * x)}
> attributes(f1); is.function(f1)
```

3.4. Nombres para los objetos

- Los nombres válidos para un objeto son combinaciones de letras, números y el punto (“.”).

  Los nombres no pueden empezar con un número.

- R es “case-sensitive”: `x != X`.

- Hay nombres reservados (“function”, “if”, etc.).

- Otras consideraciones:
  - El uso del “.” es distinto del de C++.
  - Mejor evitar nombres que R usa (ej., “c”) (se puede arreglar).

    ```
    > c <- 4; x <- c(3, 8); c
    > rm(c); c
    ```

  - Las asignaciones se hacen con `"<-"` y se recomiendan los espacios. El signo `"="` se reserva para los argumentos de las funciones.

    ```
    > x<-1:5   # Estilo incorrecto
    > x <- 1:5 # Mucho mejor
    ```

3.5. Operaciones aritméticas

- Las operaciones con vectores mejoran el uso de bucles.
- Todo más claro:
  - Es la forma natural de operar sobre objetos completos.
  - Código más fácil de entender.
  - Más sencillo de modificar y mantener.
  - Más fácil de hacer "debugging".
  - Más rápido de escribir.
- Más eficiente (en tiempo y memoria).

Principales operaciones aritméticas:

- suma `+`, resta `-`, multiplicación `*`, división `/`
- potencia `^`, raíz cuadrada `sqrt`
- `%/%` división entera, `%%` módulo: resto de la división entera
- logaritmos `log`, `log10`, `log2`, `logb(x, base)`, exponencial `exp`
- trigonométricas `sin`, `cos`, `tan`, `asin`, `acos`, `atan`
- otras:

  `max`, `min`, `range`, `pmax`, `pmin`, `mean`, `median`, `var`, `sd`, `quantile`

  `sum`, `prod`, `diff` `cumsum`, `cumprod`, `cummax`, `cummin`

- Ejemplo:

```
> data(presidents)
> help(presidents)
> range(presidents, na.rm = TRUE)
> which.min(presidents)# 28
> which.max(presidents)# 2
```

3.6. Operadores lógicos

- `<`, `>`, `<=`, `>=`, `==`, `!=`
- `!`, `&`, `|`, `xor()` y los parecidos `&&`, `||`

```
> x <- 5; x < 5; x >= 5; x == 6; x != 5
> y <- c(TRUE, FALSE); !y; z <- c(TRUE, TRUE)
> xor(y, z)
> y & z; y | z
```

- Las formas `&&`, `||` se evalúan de izquierda a derecha, examinando sólo el primer elemento de cada vector (si decide). Se suelen usar dentro de instrucciones "if".
- `if (is.numeric(x) && min(x) > 0) {entonces...`

  `min(x)` no tiene sentido si `x` no es numérico.
- `0 + y; as.numeric(y); mode(y) <- "numeric"`

Ejemplo:

```
> peso <- c(19,14,15,17,20,23,30,19,25)
> peso < 20
> peso < 20 | peso > 25
> peso[peso<20]
> peso[peso<20 & peso!=15]
> trat <- c(rep("A",3),rep("B",3),rep("C",3))
> peso[trat=="A"]
> peso[trat=="A"|trat=="B"]
> split(peso,trat)
> split(peso,trat)$A
```

3.7. Discretizar datos

- La función `split(x,f)`

```
> split(peso,trat)
> split(peso,trat)$A
```

- La función `cut`

```
> vv <- rnorm(100)
> cut1 <- cut(vv, 5)
> table(cut1)
>
> cut2 <- cut(vv, quantile(vv, c(0, 1/4, 1/2, 3/4, 1)))
> summary(cut2)
> class(cut2)
```

3.8. Operaciones con conjuntos

```
> x <- 1:5; y <- c(1, 3, 7:10)
> union(x, y)
> intersect(x, y)
> setdiff(y, x)
> v <- c("bcA1", "bcA2", "blX1")
> w <- c("bcA2", "xA3")
> union(v, w)
> intersect(v, w)
> setdiff(w, v)
> setdiff(v, w)
```

4. Vectores

4.1. Generación de secuencias

```
> x <- c(1, 2, 3, 4, 5)
> x <- 1:10; y <- -5:3
> 1:4+1; 1:(4+1)
> x <- seq(from = 2, to = 18, by = 2)
> x <- seq(from = 2, to = 18, length = 30)
> y <- seq(along = x)
> z2 <- c(1:5, 7:10, seq(from=-7,to=5,by=2))
```

```
> rep(1, 5)
> x <- 1:3; rep(x, 2)
> y <- rep(5, 3); rep(x, y)
> rep(1:3, rep(5, 3))
> rep(x, x)
> rep(x, length = 8)
> gl(3, 5) # como rep(1:3, rep(5, 3))
> gl(4, 1, length = 20) # !Alerta! gl genera factores
> gl(3, 4, label = c("Sano", "Enfermo", "Muerto"))
> expand.grid(edad = c(10, 18, 25),
> sexo = c("Macho", "Hembra"), loc = 1:3)
```

Podemos combinar: `z5 <- c(1:5, rep(8, 3))`

4.2. Generación de secuencias aleatorias

```
> sample(5)
> sample(5, 3)
> x <- 1:10
> sample(x)
> sample(x, replace = TRUE)
> sample(x, length = 2* length(x), replace = TRUE)
> probs <- x/sum(x)
> sample(x, prob = probs)
```

Números aleatorios `rDistribución(n,parámetros)`

```
> rnorm(10) # rnorm(10, mean = 0,sd = 1)
> runif(8,min=2,max=10)
```

4.3. Selección de elementos de un vector

```
> x <- 1:5; x[1]; x[3]; x[c(1,3)]
> x[x > 3]
> x > 3
> y <- x > 3
> x[y]
> x[-c(1, 4)]; y <- c(1, 2, 5); x[y]
> names(x) <- c("a", "b", "c", "d", "patata")
> x[c("b", "patata")]
```

4.4. Valores perdidos

- `NA` es el código de "Not available".

```
> v <- c(1,6,9,NA)
> is.na(v); which(is.na(v))
> w <- v[!is.na(v)] # sin los valores perdidos
> v == NA # !No funciona! ¿Por qué?
```

- Sustituir `NA` por, p.ej., `0`:

```
> v[is.na(v)] <- 0
```

- El infinito y `NaN` ("not a number") son diferentes de `NA`.

```
> 5/0; -5/0; 0/0
> is.infinite(-5/0); is.nan(0/0); is.na(5/0)
```

- Con algunas funciones

```
> xna <- c(1, 2, 3, NA, 4); mean(xna)
> mean(xna, na.rm = TRUE)
```

- Para "modelling functions" (ej. lm) lo mejor es usar

  `na.omit`

  `na.exclude`

  Esta última es más conveniente para generar predicciones, residuos, etc.

- Eliminar todos los `NA`:

```
> XNA <- matrix(c(1,2,NA,3,NA,4), nrow = 3)
> XNA
> X.no.na <- na.omit(XNA)
```

4.5. Ordenación de vectores

```
> x1 <- c(5, 1, 8, 3)
> order(x1)
> sort(x1)
> rev(x1)
> rank(x1)
> x1[order(x1)]
> x2 <- c(1, 2, 2, 3, 3, 4); rank(x2)
> min(x1); which.min(x1); which(x1 == min(x1))
> y <- c(1, 1, 2, 2); order(y, x)
```

`order` y `sort` admiten `decreasing = TRUE`.

4.6. Vectores de caracteres

```
> codigos <- paste(c("A", "B"), 2:3, sep = "")
> codigos <- paste(c("A", "B"), 2:3, sep = ".")
> juntar <-
  paste(c("una", "frase", "simple"), collapse =" ")
> columna.a <- LETTERS[1:5]; columna.b <- 10:15
> juntar <- paste(columna.a, columna.b, sep ="")
> substr("abcdef",2,4)
> x <- paste(LETTERS[1:5], collapse="")
> substr(x, 3, 5) <- c("uv")
```

Otras funciones de manipulación de caracteres:

`nchar`, `grep`, `match`, `pmatch`, `tolower`,

`toupper`, `sub`, `gsub`, `regexpr`.

4.7. Factores

- Consideremos el código postal:

```
> codigo.postal <- c(28430, 28016, 28034);
> mode(codigo.postal)
```

  No deberíamos usar el código postal en, por ejemplo, un ANOVA como si fuera un vector numérico. Usar variables aparentemente numéricas en análisis estadísticos es un grave error.

```
> codigo.postal <- factor(codigo.postal) # mejor
```

- Antes de utilizar un vector con caracteres dentro de un análisis, hace falta convertirlo en un factor. En caso contrario, R protesta.

```
> y <- rnorm(10); x <- rep(letters[1:5], 2)
> aov(y ~ x) # !error!
> aov(y ~ factor(x)) # funciona
```

- Si queremos convertir un vector factor en numérico:

```
> x <- c(34, 89, 1000); y <- factor(x); y
> as.numeric(y) # mal
> # los valores han sido recodificados
> as.numeric(as.character(y)) # bien
```

- Podemos fijar el orden de las etiquetas:

```
> ftr1 <- factor(c("alto", "bajo", "medio"))
> ftr1
> ftr1 <- factor(c("alto", "bajo", "medio"),
+ levels = c("bajo", "medio", "alto"))
```

5. Arrays y matrices

5.1. Definiciones

- Un array es una colección de datos del mismo tipo con varias dimensiones.

  ```
  > a <- 1:24; dim(a) <- c(3,4,2)
  ```

  El vector `a` pasa a ser un array `3x4x2`.

- Una `matrix` es un array con dos dimensiones. Tienen una funcionalidad muy parecida, pero matrix es más cómoda.

  ```
  > a1 <- array(9, dim = c(5,4))
  > a2 <- matrix(1:20, nrow = 5)# como en FORTRAN
  > a3 <- matrix(1:20, nrow = 5, byrow = TRUE)
  > a4 <- 1:20; dim(a4) <- c(5, 4)
  ```

- Con las coordenadas se obtienen los elementos particulares, como en los vectores:

```
> a[1,1,1]; a[1,1,2]; a[3,4,2]
```

- También podemos considerar subconjuntos de un array

```
> a[2, , ] # es un array de dimensión c(4,2)
> a4[1, ]; a4[, 2]; a4[c(1, 3), c(2, 4)]
```

- También se pueden dar las coordenadas matricialmente. Observar el ejemplo:

```
> im <- matrix(c(1, 3, 2, 4), nrow = 2)
> im
> a4[im]
```

Ejemplo:

```
> x <- c(190,8,22,191,4,1.7,223,80,2,210,50,3)
> datos <- matrix(x,nrow=4,byrow=T); dim(datos)
> ciudades <- c("Barna","Tarraco","Lleida","Gi")
> dimnames(datos) <- list(ciudades,NULL)
> variables <- c("A","B","C")
> dimnames(datos) <- list(ciudades,variables)
> datos
> dimnames(datos)
> datos["Barna", ]
> datos[ ,"C"]
```

Otro ejemplo:

```
> a4 <- 1:20; dim(a4) <- c(5, 4)
> attributes(a4)
> colnames(a4) <- paste("v", 1:4, sep = "")
> rownames(a4) <- paste("id", 1:5, sep = ".")
> a4[, c("v1", "v3")]
> attributes(a4)
```

Para ordenar un array por una columna:

```
> matriz <- matrix(rnorm(20),ncol=4)
> o.matriz <- matriz[order(matriz[, 1]), ]
```

5.2. Operaciones con matrices

- `A %*% B` : producto de matrices
- `t(A)` : transpuesta de la matriz `A`
- `solve(A,b)` : solución del sistema de ecuaciones `Ax=b`.
- `solve(A)` : inversa de la matriz `A`
- `svd(A)` : descomposición en valores singulares
- `qr(A)` : descomposición QR
- `eigen(A)` : valores y vectores propios
- `diag(b)` : matriz diagonal (`b` es un vector)
- `diag(A)` : matriz diagonal (`A` es una matriz)
- `A %o% B == outer(A,B)` : producto exterior de dos vectores o matrices

- Las funciones `var`, `cov` y `cor` calculan la varianza de `x` y la covarianza o correlación de `x` y `y` si éstos son vectores. Cuando `x` y `y` son matrices, entonces calculan las covarianzas (o correlaciones) entre las columnas de `x` y las columnas de `y`.

```
> data(longley)
> (Cl <- cor(longley))
> ## Graphical Correlation Matrix:
> symnum(Cl) # highly correlated


> ## Spearman's rho
> cor(apply(longley, 2, rank))
> cor(longley, method = "spearman") # better
```

- La función `cov2cor` convierte "eficientemente" una matriz de covarianzas en la correspondiente matriz de correlaciones.

- La función `outer(X, Y, FUN="*", ...)` proporciona por defecto el producto exterior de los dos arrays. Sin embargo, podemos introducir otras funciones e incluso nuestras propias funciones.

```
> x <- 1:9; names(x) <- x
> # Multiplication & Power Tables
> x %o% x
> y <- 2:8; names(y) <- paste(y,":",sep="")
> outer(y, x, "^")
```

5.3. Combinación de arrays

Para combinar vectores, matrices o arrays utilizamos las instrucciones `rbind` y `cbind`.

```
> x1 <- 1:10; x2 <- 11:20
> a6 <- diag(6) # matriz identidad
> a7 <- cbind(x1, x2); a8 <- rbind(x1, x2)
> a24 <- cbind(a2, a4)
> cbind(a4, a6) # no funciona
> rbind(a4, a6) # no funciona
> a9 <- matrix(rnorm(30), nrow = 5)
> cbind(a4, a9)
> rbind(a4, a9) # no funciona
```

6. Data.frames

6.1. Definición

- Para datos de diferentes tipos:

```
> x3 <- letters[1:10]
> a9 <- cbind(x1, x2, x3)
```

- ¿De qué tipo es `a9`? ¿Es eso lo que queríamos?

  Mejor con un `data.frame`:

```
> a10 <- data.frame(x1, x2, x3)
> prcomp(a10[, c(1,2)])# comp. principales
> prcomp(a10[, c("x1", "x2")])
> prcomp(a10[, -3])
```

- También podemos añadir alguna columna a una matriz como `datos`:

```
> playa <- c("si","si","no","no")
> datos.df <- data.frame(datos,playa)
> datos.df$playa
```

- Usar `$` facilita el acceso y la creación de nuevas columnas:

```
> set.seed(1) # fija la semilla del random number generator
> d1 <- data.frame(g1 = runif(10), g2 = rnorm(10))
> d1$edad <- c(rep(20, 5), rep(40, 5))
> set.seed(1)
> d2 <- cbind(g1 = runif(10), g2 = rnorm(10))
> d2[, 3] <- c(rep(20, 5), rep(40, 5)) # error
> d2 <- cbind(d2, edad = c(rep(20, 5), rep(40, 5)))
```

- Además, en los data.frame los "character vectors" se convierten en factores.
- Podemos convertir matrices a `data.frame` con `as.data.frame()`.
- Los `data.frame` también tienen `rownames`, `colnames`.

  ```
  > attributes(a10) # cuando no están definidos
  ```

- También podemos usar `dimnames(a10)`.

6.2. La familia apply

```
> ax <- matrix(rnorm(20), ncol = 5)
> medias.por.fila <- apply(ax, 1, mean)
> por.si.na <- apply(ax, 1, mean, na.rm = TRUE)
> mi.f1 <- function(x) { return(2*x - 25)}
> mi.f1.por.fila <- apply(ax, 1, mi.f1)
> mas.simple <- apply(ax, 1, function(x){return(2*x -25)})
> medias.por.columna <- apply(ax, 2, mean)
> sample.rows <- apply(ax, 1, sample)
> dos.cosas <- function(y){return(c(mean(y), var(y)))}
> apply(ax, 1, dos.cosas)
> t(apply(ax, 1, dos.cosas))
```

- Utilizar `apply` es generalmente mucho más eficiente que un bucle. Además de más claro, más fácil, etc..

```
> parameters <- cbind(mean = -5:5, sd = 2:12)
> z.data <- matrix(rnorm(1000 * 11), nrow = 11)
> data <- (z.data * parameters[,2]) + parameters[,1]
> apply(data, 1, mean); apply(data, 1, sd)
```

- Las funciones `sapply(X,función)` y `lapply(X,función)` son como `apply(x,i,función)` pero no hay que especificar el índice `i=2`;

  `sapply` intenta simplificar el resultado a un vector o a una matriz (la "s" es de "simplify"), pero `lapply` siempre devuelve una lista. Ambas pueden aplicarse a vectores, listas, arrays.

```
> data(airquality)
> sapply(airquality, function(x)sum(is.na(x)))
```

- La función `tapply(x,y,función)` calcula la función especificada sobre el objeto `x` según las categorías de `y`.

```
> x <- c(19,14,15,17,20,23,19,19,21,18)
> trat <- c(rep("A",5),rep("B",5))
> x.media <- tapply(x,trat,mean)
> x.media
```

- `apply`, `sapply`, `lapply` y `tapply` son funciones muy útiles que contribuyen a hacer el código más legible, fácil de entender, y facilitan posteriores modificaciones y aplicaciones.

  Consejo: Cada vez que vayamos a usar un "loop" intentemos substituirlo por algún miembro de familia `apply`.

- Algunas funciones hacen un `apply`:

```
> x1 <- 1:10
> m1 <- matrix(1:20, ncol = 5)
> d1 <- as.data.frame(m1)
> mean(x1); mean(d1); sd(x1); sd(d1); median(m1); median(d1)
```

6.3. Tablas

- La tabulación cruzada de dos variables cualitativas se consigue con la función `table`.

```
> table(sexo,nivel)
```

- Para introducir una tabla de contingencia también se utiliza la instrucción `table`. Las variables se definen con sus modalidades con la instrucción `expand.grid(var1,var2)`.

```
> resultado <- cbind(expand.grid(
+                    calif=c("mejor","peor","igual"),
+                    tratam=c("A","B")))
> frec <- c(21,34,5,7,12,14)
> tabla <- table(calif,tratam)*frec
> tabla
```

6.4. Ejemplo

```
> d3 <- data.frame(g1=runif(10),g2=rnorm(10),
+ id1 = c(rep("a", 3), rep("b", 2),
+ rep("c", 2), rep("d", 3)))
> my.fun <- function(x) {
+ las.medias <- mean(x[, -3])
+ las.vars <- var(x[, -3])
+ max.total <- max(x[, -3])
+ tabla.clases <- table(x[, 3])
+ return(list(row.means = las.medias,
+ row.vars = las.vars, maximum = max.total,
+ factor.classes = tabla.clases))
+ }
> my.fun(d3)
```

7. Muchos objetos

7.1. Listas

```
> una.lista <- my.fun(d3); una.lista
> attributes(una.lista); names(una.lista)
> length(una.lista)
> una.lista[[4]]
> una.lista[4] # ¿por qué sale el nombre? class
> una.lista$factor.classes
> una.lista[[3]] <- list(NULL); una.lista
> una.lista[[3]] <- NULL
> una.lista # hemos eliminado el "slot" maximum
> unlist(una.lista)
> otra.lista <- list(cucu = 25, una.lista)
> unlist(otra.lista)
> unlist(otra.lista, drop = FALSE)
> una.lista <- c(una.lista, otro.elemento = "una frase")
```

7.2. Los objetos que tenemos

- Para saber los objetos que hemos definido hacemos

```
> ls()
> objects()
> objects(pattern="a*")
```

- R tiene una lista donde buscar los objetos accesibles: "the search list". Cargar un paquete extiende la "search list".

  Para obtener la lista de los directorios, llamados "databases":

```
> search()
> library(MASS)
> search()
```

- Para que un objeto o directorio sea fácilmente accesible lo podemos poner en la "search list" de R. En el caso de un data.frame, esto permite acceder directamente a las columnas por su nombre.

```
> str(datos.df) # es un data.frame
> A # error
> attach(datos.df)
> A # ahora sí
> plot(A,B) # en lugar de plot(datos.df$A,datos.df$B)
```

- La actualización no es dinámica

```
> datos.df$D <- 1:4 # una nueva columna
> datos.df # aquí está
> D        # pero aquí no
```

- Para desconectar

```
> detach(objeto)
```

- Para borrar objetos concretos

```
> rm(objetos)
```

- Para borrar todos los objetos del entorno de trabajo:

```
> rm(list = ls())
```

- Para cargar un archivo `nombre.RData` con objetos diversos (datos, funciones,...) se puede hacer un `attach` o un `load`. La primera instrucción accede a los objetos cuando se requieren, la segunda los carga todos.

```
> load("nombre.RData")
```

- ¡Alerta!

```
> datos.df
> A <- 1
> A  # usa la última
> search() # el search path
> detach(datos.df)
> attach(datos.df)
> D
> A # cuidado
```

Conclusión: En "entornos confusos", como un análisis que se prolonga dos semanas, es mejor evitar `attach` y acceder siempre a las variables usando su localización explícita y completa.

7.3. En resumen

- La manipulación de datos en R es muy flexible.
- Podemos seleccionar variables, casos, subsecciones de datos, etc, de acuerdo con criterios arbitrarios (que usan, además, condiciones que pueden implicar a un número arbitrario de variables y casos).
- Los data.frames y las matrices pueden separarse, juntarse, cambiarse de forma (`reshape`), etc.
- El indexado y selección de casos pueden usar números, factores, cadenas de caracteres, etc.
- Podemos preparar código que repita las mismas operaciones con datos semejantes (i.e., podemos automatizar el proceso con sencillez).
- Podemos verificar "al vuelo" que estas transformaciones hacen lo que queremos que hagan (mirando selectivamente los resultados, o "emulando" el proceso en unos datos artificiales más pequeños).
- Por tanto, una vez que los datos están en R, no hay muchas

razones para exportarlos y hacer la selección y manipulación con otros lenguajes (ej., Python, Perl) para luego volver a leerlos en R.

8. Datos

8.1. Lectura de datos

- Para leer un fichero simple, con los datos separados por espacios en blanco, tabuladores o saltos de línea, se utiliza la instrucción `read.table` en la forma:

```
> fichero.df <- read.table("c:/dir/mi.fichero",
+ header = TRUE, sep = "",
+ comment.char = "")
```

- Si el carácter decimal no es un punto sino, por ej., una coma, usar: `dec = ","`.
- Se pueden saltar líneas (`skip`) o leer un número fijo de líneas (`nrows`).
- Hay funciones especializadas para otros archivos (ej., `read.csv`) pero son casos específicos de `read.table`.

8.2. Guardar y leer datos

- Resulta muy importante poder guardar datos, funciones, etc., para ser usados en otras sesiones de R. Esos datos así guardados pueden compartirse con otros usuarios e incluso utilizarse en distintos sistemas operativos.

```
> x <- runif(20)
> y <- list(a = 1, b = TRUE, c = "patata")
> save(x, y, file = "xy.RData")
```

- Los leeremos con

```
> load("xy.RData")
```

- Podemos guardar todos los objetos con

```
> save.image() # guardado como ".RData"
> save.image(file = "nombre.RData")
```

- El fichero `.RData` se carga al iniciarse R.
- R y muchos otros paquetes incorporan archivos con datos:

  Se cargan con `load("nombre.RData")`.
- La instrucción `data` permite cargar archivos de las librerías disponibles.

```
> data() # muestra todos los archivos
> data(iris)
> data(iris, package = "base") # equivalente
> ?iris
```

8.3. Importar datos de Excel

- Lo mejor es exportar los datos desde Excel a un archivo de texto separado por tabuladores.
- Cuidado con las últimas columnas y *missing data* (Excel elimina los "trailing tabs"). Dos formas de minimizar problemas:
  - Usar `NA` para missing.
  - Poner una última columna con datos arbitrarios (ej., una columna llena de 2s).
- Cuidado también con líneas extra al final del fichero.
- Salvamos como texto (sólo salvamos una de las hojas).
- Importamos en R con `read.table`.

8.4. Exportar datos

- Lo más sencillo es exportar una matriz
  (es necesario transponer la matriz).

```
> write(t(x), file = "c:/dir/data.txt",
+       ncolumns = n,
+       append = FALSE)
```

- Pero para exportar un `data.frame` es mejor

```
> write.table(my.data.frame,
+       file = "mi.output.txt",
+       sep = "",row.names = FALSE,
+       col.names = TRUE)
```

- Para escribir un fichero CSV importable desde Excel

```
>  write.table(x, file = "foo.csv", sep = ",",
+              col.names = NA)
```

9. Gráficos

9.1. Introducción

- R incluye muchas y variadas funciones para hacer gráficos.
- El sistema permite desde gráficos muy simples a figuras de calidad para incluir en artículos y libros.
- Sólo examinaremos la superficie. Más detalles en el libro *R Graphics* de Paul Murrell.
- También podemos ver un buen conjunto de ejemplos con `demo(graphics)`.
- El comando `plot` es uno de los más utilizados para realizar gráficos.

9.2. El comando `plot`

- Si escribimos `plot`$(\mathbf{x}, \mathbf{y})$ donde $\mathbf{x}$ e $\mathbf{y}$ son vectores con n coordenadas, entonces R representa el gráfico de dispersión con los puntos de coordenadas (x_i, y_i).
- Este comando incluye por defecto una elección automática de ejes, escalas, etiquetas de los ejes, densidad de las líneas, etc., que pueden ser modificados añadiendo parámetros gráficos al comando y que pueden visualizarse con `help(par)`.

```
> x <- runif(50, 0, 4); y <- runif(50, 0, 4)
> plot(x, y, main = "Título principal",
+ sub = "subtítulo", xlab = "eje x", ylab = "eje y",
+ xlim = c(-5,5),ylim = c (-5,5))
```

- Variaciones de `plot`:

```
> z <- cbind(x,y)
> plot(z)
> plot(y ~ x)
> plot(log(y + 1) ~ x) # transformación de y
> plot(x, y, type = "p")
> plot(x, y, type = "l")
> plot(x, y, type = "b")
> plot(c(1,5), c(1,5))
> legend(1, 4, c("uno", "dos", "tres"), lty = 1:3,
+ col = c("red", "blue", "green"),
+ pch = 15:17, cex = 2)
```

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

```
> sexo <- c(rep("v", 20), rep("m", 30))
> plot(x, y, type = "n")
> text(x, y, labels = sexo)
```

- Puntos.

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

- Tipos de puntos.

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

- Tipos de líneas.

```
> plot(c(0, 10), c(0, 10), type = "n", xlab ="",
+ ylab ="")
> for(i in 1:10)
+ abline(0, i/5, lty = i, lwd = 2)
> for(i in 1:10)
+ abline(0, i/5, lty = i, lwd = 2)
```

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

9.3. Identificación interactiva de datos

- `identify`$(x, y,$ *etiquetas*$)$ identifica los puntos con el ratón y escribe la correspondiente etiqueta.

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

- `locator()` devuelve las coordenadas de los puntos.

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

9.4. Múltiples gráficos por ventana

- Empezamos con `par(mfrow=c(`*filas*,*columnas*`))` antes del comando `plot`.

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

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

9.5. Datos multivariantes

- Diagrama de dispersión múltiple.

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

- Gráficos condicionados (revelan interacciones).

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

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

9.6. Boxplots

- Los diagramas de caja son muy útiles para ver rápidamente las principales características de una variable cuantitativa, o comparar entre variables.

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

- La función `boxplot` tiene muchas opciones; se puede modificar el aspecto, mostrarlos horizontalmente, en una matriz de boxplots, etc. Véase la ayuda `?boxplot`.

9.7. Un poco de ruido

Los datos cuantitativos discretos pueden ser difíciles de ver bien. Podemos añadir un poco de ruido con el comando `jitter`.

```
> dc1 <- sample(1:5, 500, replace = TRUE)
> dc2 <- dc1 + sample(-2:2, 500, replace = TRUE,
+ prob = c(1, 2, 3, 2, 1)/9)
> plot(dc1, dc2)
> plot(jitter(dc1), jitter(dc2))
```

9.8. Dibujar rectas

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

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

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

9.9. Más gráficos

- Podemos modificar márgenes exteriores de figuras y entre figuras (véase `?par` y búsquense `oma`, `omi`, `mar`, `mai`; ejemplos en *An introduction to R*, secc. 12.5.3 y 12.5.4.
- También gráficos 3D: `persp`, `image`, `contour`; histogramas: `hist`; gráficos de barras: `barplot`; gráficos de comparación de cuantiles, usados para comparar la distribución de dos variables, o la disribución de unos datos frente a un estándar (ej., distribución normal): `qqplot`, `qqnorm` y, en paquete `car`, `qq.plot`.
- Notación matemática (`plotmath`) y expresiones de texto arbitrariamente complejas.
- Gráficos tridimensionales dinámicos con XGobi y GGobi. Ver: `http://cran.r-project.org/src/contrib/Descriptions/xgobi.html`, `http://www.ggobi.org`, `http://www.mcs.vuw.ac.nz/~ray/R-stuff/windows/gguide.pdf`.

9.10. Guardar los gráficos

- En Windows, podemos usar los menús y guardar con distintos formatos.
- También podemos especificar donde queremos guardar el gráfico.

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

- O bien, podemos copiar una figura a un fichero.

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

10. Funciones

10.1. Definición de funciones

- R es un lenguaje que permite crear nuevas funciones. Una función se define con una asignación de la forma

```
> nombre <- function(arg1,arg2,...){expresión}
```

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

  Ejemplo: Suma de una progresión aritmética

```
> suma <- function(a1,d,n){
+         an <- a1+(n-1)*d;
+         ((a1+an)*n)/2}
```

10.2. Argumentos

- Una función con cuatro argumentos

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

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

```
> una.f(4, 5)
> una.f(b = 5, a = 4)
```

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

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

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

- `args(nombre.funcion)` nos muestra los argumentos de cualquier función.
- "..." permite pasar argumentos a otra función:

```
> f3 <- function(x, y, label = "la x", ...){
+                 plot(x, y, xlab = label, ...)}
>
> f3(1:5, 1:5)
> f3(1:5, 1:5, col = "red")
```

- Para realizar funciones de dos variables se puede utilizar el comando `outer`. Por ejemplo:

```
> f <- function(x,y){cos(y)/(x^2-3)}
> z <- outer(x,y,f)
```

10.3. Scope

- En la función `una.f` "`z`" es una "free variable": ¿cómo se especifica su valor? Lexical scoping. Ver documento *Frames, environments and scope in R and S-PLUS* de J. Fox en `http://cran.r-project.org/doc/contrib/Fox-Companion/appendix.html` y sección 10.7 en *An introduction to R*. También ver `demo(scoping)`.
- Un ejemplo

```
> cubo <- function(n) {
+ sq <- function() n*n # aquí n no es un argumento
+ n*sq()
+ }
```

- En esto R (lexical scope) y S-PLUS (static scope) son distintos.

10.4. Control de ejecución

- Principales instrucciones

```
if(cond) expr
if(cond) cons.expr  else  alt.expr
for(var in seq) expr
while(cond) expr
repeat expr
break
next
```

- La expresión `expr` (también `alt.expr`) puede ser una expresión simple o una de las llamadas expresiones compuestas, normalmente del tipo `{expr1; expr2}`.
- Uno de los errores más habituales es el olvido de los corchetes `{...}` alrededor de las instrucciones, i.e. después de `if(...)` o `for(...)`.

- `if (cond.logica)` *instrucción* `else` *instrucción.alternativa*

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

- `ifelse` es una versión vectorizada (Thomas Unternährer, R-help, 2003-04-17)

```
> odd.even <- function(x) {
+ ifelse(x %% 2 == 1, "Odd", "Even")
+ }
> mtf <- matrix(c(TRUE, FALSE, TRUE, TRUE),
+ nrow = 2)
> ifelse(mtf, 0, 1)
```

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

```
> for(i in 1:10) cat("el valor de i es", i, "\n")
```

```
> continue.loop <- TRUE
> x <- 0
> while(continue.loop) {
+ x <- x + 1
+ print(x)
+ if( x > 10) continue.loop <- FALSE
+ }
```

- `break` para salir de un loop.

10.5. Cuando algo va mal

- Cuando se produce un error, `traceback()` nos informa de la secuencia de llamadas antes del "crash" de nuestra función. Es útil cuando se producen mensajes de error incomprensibles.
- Cuando se producen errores o la función da resultados incorrectos o "warnings" indebidos podemos seguir la ejecución de la función.
- `browser` interrumpe la ejecución a partir de ese punto y permite seguir la ejecución o examinar el entorno; con "n" paso a paso, si otra tecla sigue la ejecución normal. "Q" para salir.
- `debug` es como poner un `broswer` al principio de la función y se ejecuta la función paso a paso. Se sale con "Q".

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

Ejemplo:

```
> my.f2 <- function(x, y) {
+ z <- rnorm(10) + y2 <- z * y + y3 <- z * y * x + return(y3 + 25)
+ }
> my.f2(runif(3), 1:4)
> debug(my.f2)
> my.f2(runif(3), 1:4)
> undebug(my.f2)
> # insertar un browser() y correr de nuevo
```

10.6. Ejecución no interactiva

- Con `source` abrimos una sesión de R y hacemos

  ```
  > source("mi.fichero.con.codigo.R")
  ```

- Con BATCH:

  ```
  Rcmd BATCH mi.fichero.con.codigo.R
  ```

- `source` es en ocasiones más útil porque informa inmediatamente de errores en el código. BATCH no informa, pero no requiere tener abierta una sesión (se puede correr en el background).

- Ver la ayuda: `Rcmd BATCH --help`

- Puede que necesitemos explícitos `print` statements o hacer `source(my.file.R, echo = TRUE)`.

- `sink` es el inverso de `source` (lo manda todo a un fichero).

- Se pueden crear paquetes, con nuestras funciones, que se comporten igual que los demás paquetes. Ver *Writing R extensions*.

- R puede llamar código compilado en C/C++ y FORTRAN. Ver `.C`, `.Call`, `.Fortran`.
- "Lexical scoping" importante en programación más avanzada.
- No hemos mencionado el "computing on the language" (ej., `do.call`, `eval`, etc.).
- R es un verdadero "object-oriented language". Dos implementaciones, las S3 classes y las S4 classes.