SIG. Práctica 2. Tema 1.
Cartografía Digital y Temática con R

José Samos Jiménez

Departamento de Lenguajes y Sistemas Informáticos
Universidad de Granada

2019 jsamos (LSI-UGR)

Curso 2019-2020

Índice

Los objetivos de este capítulo son:

- Repasar conceptos sobre *Información Geográfica* explicados en clase de Teoría.
- Introducir R mediante ejemplos. No es una introducción exhaustiva sino un primer contacto; se pondrá en práctica:
  - Estructuras de datos.
  - Control de flujo.
  - Funciones.
  - Plots básicos.
  - Paquetes.
  - Visualización de capas.
  - Crear un mapa temático.

A continuación, después de una introducción sobre la herramienta, se irán presentando algunas de sus funcionalidades mediante pasos que deberás ir realizando. Las actividades propuestas se basan principalmente en [BC19] y [Mat11]. Los datos se han elaborado a partir de datos obtenidos de [IDE17] e [INE19].

Descarga el archivo `Granada.zip` que contiene los archivos de las capas que se usarán como fuente de datos. Extrae todos archivos y sitúalos en una carpeta de trabajo.

1. Introducción

R[1] es un proyecto de software libre, se basa en el lenguaje *S*, desarrollado originalmente en *Bell Laboratories* en los años 70 y 80 del siglo pasado. *R* como tal fue desarrollado en la *University of Auckland, New Zealand*. Comenzó como proyecto en 1992 y la primera versión estable se lanzó en 2000.

Como principales características se puede remarcar la facilitad que presenta para trabajar con estructuras de datos y con gráficos, y el repositorio de paquetes disponibles para su uso.

En lugar de trabajar directamente sobre *R*, usaremos *RStudio*[2], entorno de desarrollo integrado (IDE) especialmente pensado para *R* que permite ejecución paso a paso, visualización del entorno, acceso a la ayuda, formateo del código, definición de proyectos y directorio de trabajo, entre otras funcionalidades.

A continuación, en la sección 2, se van a introducir distintos aspectos de *R* mediante ejemplos. No pretende ser un resumen de *R* sino un repaso de aspectos que se van a utilizar al tratar información geográfica. Se pueden encontrar verdaderas introducciones a *R*, por ejemplo, en:

- National Center for Ecological Analysis and Synthesis: *R: A self-learn tutorial*[3].
- W. J. Owen: *The R Guide*[4].
- G. Rodríguez: *Introducing R*[5].
- P. Torfs, C. Braue: *A (very) short introduction to R*[6].
- W. N. Venables, D. M. Smith and the R Core Team: *An Introduction to R*[7].

The R Guide de Owen es posiblemente la más conocida.

[1] `https://www.r-project.org/`
[2] `https://www.rstudio.com/`
[3] `https://www.nceas.ucsb.edu/files/scicomp/Dloads/RProgramming/BestFirstRTutorial.pdf`
[4] `http://www.mathcs.richmond.edu/~wowen/TheRGuide.pdf`
[5] `http://data.princeton.edu/R/introducingR.pdf`
[6] `https://cran.r-project.org/doc/contrib/Torfs+Brauer-Short-R-Intro.pdf`
[7] `https://cran.r-project.org/doc/manuals/R-intro.pdf`

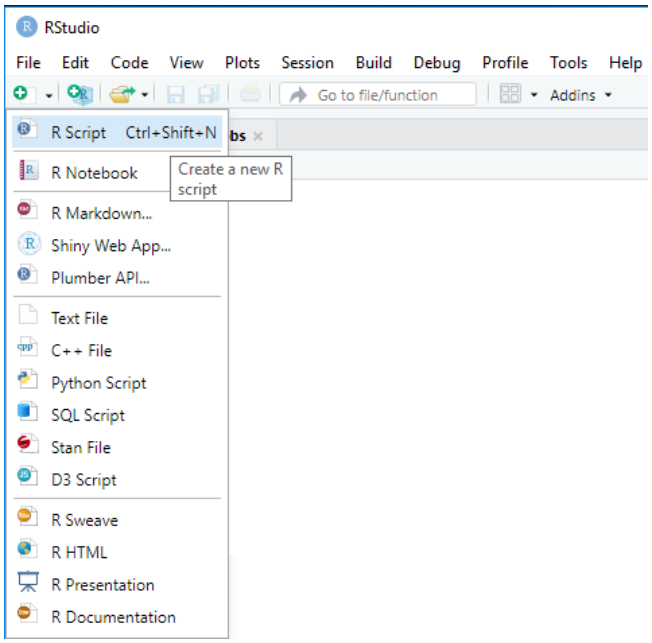

Figura 1: Crear un archivo de *R*.

2. Introducción a *R* mediante ejemplos

Trabajaremos con *R* desde *RStudio*. La manera más inmediata de realizar un desarrollo sencillo en *R* es creando un nuevo archivo de *R* (un script), como se muestra en la figura 1.

1. Inicia *RStudio* y crea un archivo de *R*.

La ventana de *RStudio* tiene cuatro zonas (figura 2). La parte superior izquierda corresponde al archivo *R* que acabamos de crear. En esta zona iremos escribiendo el código *R* que desarrollemos (en este caso, copiando-y-pegando el de las figuras siguientes).

Para ejecutar una línea de código, situamos el cursor en cualquier parte de la línea y pulsamos sobre el botón «Run» (figura 2). El cursor se sitúa en la línea siguiente para que podamos ejecutarla al volver a pulsar el mismo botón[8]. En la ventana inferior izquierda, «Console», se muestra lo que se envía a *R* y el resultado que este devuelve. Si se ha creado o modificado una variable, el resultado se puede ver en la ventana de la parte superior derecha, «Environment».

2.1. Funcionamiento general

2. Ejecuta el código de las figuras 3 a 21 (trata de entender cómo se producen los resultados que se obtienen).

En la figura 3 se muestran dos formas de asignación (`=` y `<-`). En la mayoría de la situaciones resultan equivalentes; puede haber diferencias dependiendo del ámbito de las variables. En las guías de estilo de *R* se recomienda el uso de `<-`. Al poner una asignación entre paréntesis adicionalmente se presenta el resultado obtenido.

En *R* se pueden usar funciones y constantes predefinidas (figura 4). Obtenemos ayuda sobre lo que hace una función ejecutando la línea con `?` y el nombre de la función. La ayuda aparecerá en la zona inferior derecha de la ventana, se activará automáticamente la pestaña «Help».

Pulsando sobre el botón «Show in new window» de la zona de ayuda de la ventana (figura 5), se abre la ayuda en una nueva ventana, para poder leerla más fácilmente.

En la figura 6, se define un vector numérico y se realizan operaciones con él. Observa las diferencias entre la clase de los resultados que se obtienen (número o vector).

[8]Si seleccionamos varias líneas, al pulsar este botón, se ejecutan todas. En esta actividad, nos interesará ejecutar línea a línea.

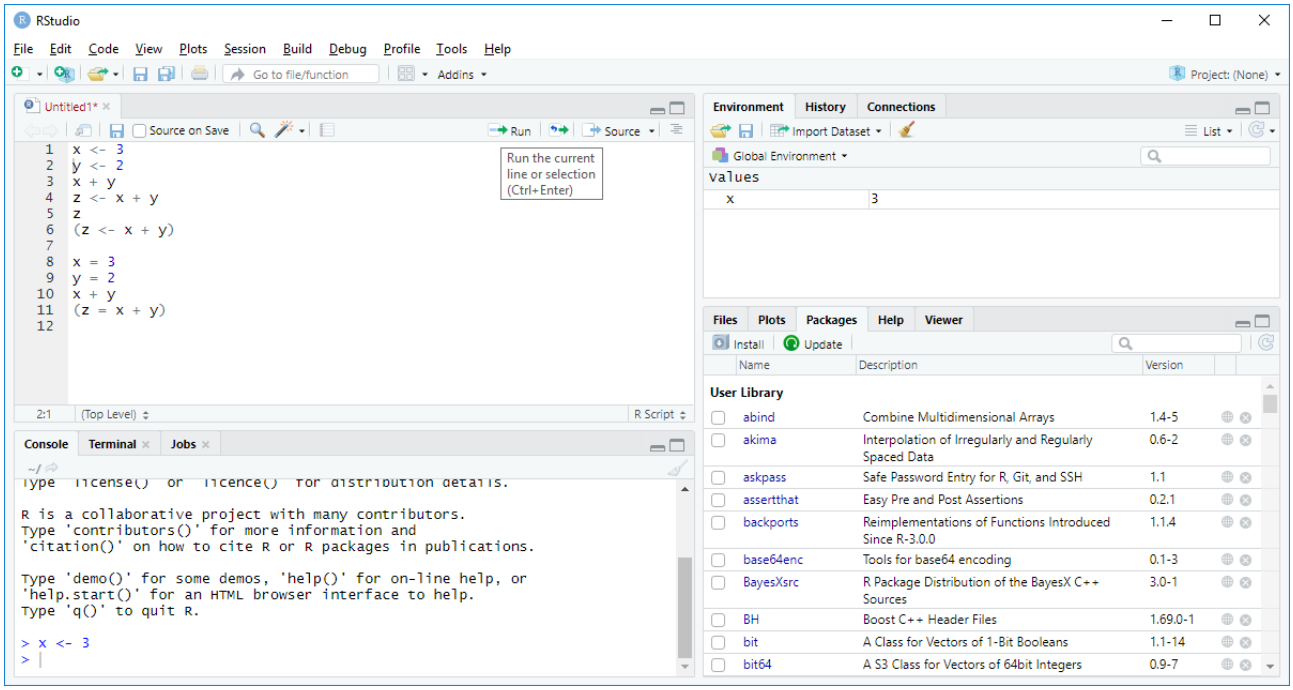

Figura 2: Ejecutar código *R*.

```
x <- 3
y <- 2
x + y
z <- x + y
z
(z <- x + y)

x = 3
y = 2
x + y
(z = x + y)
```

Figura 3: Asignación y operaciones aritméticas.

```
2 * pi * z
sqrt(z)

?sqrt
```

Figura 4: Constantes y funciones.

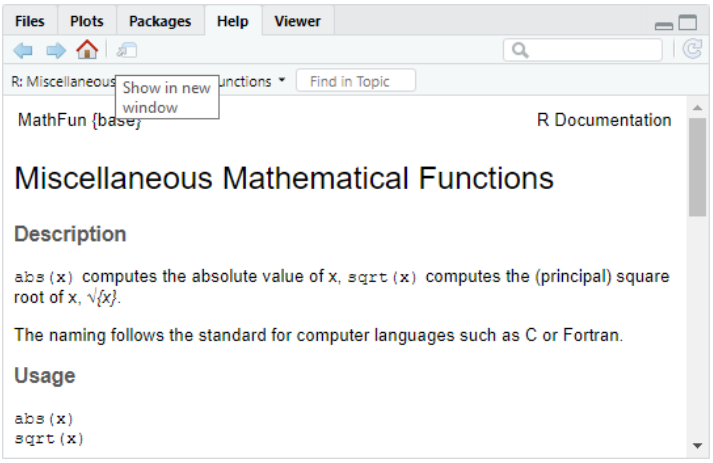

Figura 5: Ayuda.

```
r1 <- c(2, 3, 3, 5, 2, 1, 4)
2 * pi * r1

r2 <- 1:5
2 * pi * r2

sum(r1)
sqrt(r1)
round(sqrt(r1), 2)

r1^2
r1 + 1
r1 * 2
```

Figura 6: Definición y operaciones matemáticas con vectores.

```
r1[2:4]
r1[c(1,3,6)]

1:length(r1)

r1 > 3
r1[r1 > 3]

s <- c(FALSE, FALSE, FALSE, TRUE, FALSE, FALSE, TRUE)
r1[s]

s1 <- !s
r1[s1]

s2 <- r1 > 3

n <- c("María", "Carmen", "Guillermo", "Ana", "Sergio",
       "Lara", "Lucas")
n[s2]

n[2]
n[-2]
n[-(2:4)]
```

Figura 7: Selección de elementos en un vector.

```
(n2 <- c(n, n))

(n3 <- c(n, r1, s))
(n4 <- c(r1, s, n))
(n5 <- c(r1, s))
```

Figura 8: Unión de vectores.

Los elementos de un vector se pueden seleccionar mediante diversos métodos. En la figura 7, se muestran algunos de ellos: un vector de índices, un vector de índices de los elementos que no se incluyen en el resultado, un vector de booleanos. En este último caso, se muestra cómo definirlo explícitamente o bien como resultado de una operación de comparación.

Mediante la función `c()` se combinan elementos para formar un vector. Como se muestra en la figura 8, los elementos pueden ser también vectores. Como todos los elementos de un vector han de ser del mismo tipo, el tipo del resultado ha de ser el más general de los tipos de los vectores implicados, entendiendo por este, el tipo al que se puedan transformar el resto de tipos sin perder información.

2.2. Estructuras de datos

En la figura 9, podemos ver los valores y tamaño por defecto de un vector cuando se crea, según los parámetros que indiquemos o si lo creamos implícitamente; también se muestran distintas posibilidades que ofrece la función `rep()` para crear vectores.

Podemos crear un data frame a partir de vectores, a los que les asignamos nombre dentro de la estructura: son las columnas del data frame (figura 10). Un data frame puede tener tantas columnas como deseemos (no necesariamente dos como en los ejemplos de la figura 10). Podemos acceder a las columnas mediante su nombre o bien su posición en la estructura. Los vectores que forman las columnas pueden no tener el mismo número de elementos, siempre que el mayor sea múltipo del resto.

En la figura 11, se muestran distintas formas de definir una matriz (matrix), según sus dimensiones

```r
(v1 <- vector(mode="numeric", length = 7))

(v2 <- vector(length = 7))

(v3 <- vector(mode="numeric"))
v3[4] <- 1
v3

rep(0, times = 7)
rep(1:4, times = 2)
rep(1:4, each = 2)
rep(1:4, times = c(2,1,3,2))
rep(1:4, each = 2, len = 6)
rep(1:4, each = 2, times = 3)
```

Figura 9: Vector.

```r
n
r1
(d <- data.frame(persona = n, puntos = r1))

d$persona
d$puntos
colnames(d)
colnames(d) <- c("person", "score")
d$score
d[, "score"]
d[, 2]

d[2,]
d[2, 2]
d[2:5, ]
d[-c(1,6,7), ]
d[2:5, 2]

n2
(d2 <- data.frame(persona = n2, puntos = r1))
(d3 <- data.frame(persona = n2, puntos = c(1, 2)))

# Error: arguments imply differing number of rows: 14, 3
(d4 <- data.frame(persona = n2, puntos = c(1, 2, 3)))
```

Figura 10: Data frame.

```r
matrix(ncol = 2, nrow = 0)
matrix(ncol = 2, nrow = 1)
matrix(1:6)
matrix(1:6, ncol = 2)
matrix(1:6, ncol = 2, byrow = TRUE)
(m <- matrix(1:6, nrow = 3))

colnames(m) <- c("a", "b")
rownames(m) <- c("x", "y", "z")
m
m[, 2]
m[, "b"]
m[2, ]
m["y", ]
m["y", 2]
```

Figura 11: Matrix.

```r
(d2 <- data.frame(persona = n2, puntos = r1))
d2$persona

(d3 <- data.frame(persona = n2, puntos = r1,
                  stringsAsFactors = FALSE))
d3$persona

(f <- factor(n2))
(f2 <- factor(n2, levels = c("Lara", "Ana")))
```

Figura 12: Factor.

y el orden de creación (por filas o por columnas). Asimismo, podemos ver cómo se le pueden asignar nombres a las filas y columnas de la matriz, y acceder a los elementos por nombre o por posición.

Cuando incluimos un vector de strings en un data frame, lo trata como un factor: considera los valores distintos (levels) y representa los datos en función de esos valores (figura 12). Por defecto, asocia un número a cada nivel pero se puede indicar explícitamente que lo trate como string (mediante `stringsAsFactors`). Si definimos explícitamente los niveles a tener en cuenta en el factor, no se consideran los valores no incluidos entre los niveles.

Una lista (list) puede estar compuesta por elementos de distinto tipo y número de instancias (figura 13). A los elementos se les puede asignar nombres y podemos acceder a ellos por nombre o por posición.

```r
(l <- list(n2, m, d, 2))
l[[2]][1,]

(l2 <- list(nombres = n2, matriz = m, df = d))
l2[[2]]
l2$matriz
l2[["matriz"]]
```

Figura 13: List.

```r
for (nombre in n) {
  print(nombre)
}

for (i in 1:length(n)) {
  print(n[i])
}

i <- 1
while (i <= length(n)) {
  print(n[i])
  i <- i + 1
}

print(n)
```

Figura 14: Iteraciones.

```r
a <- 2
if (a == 3) {
  print(a)
} else {
  print(a*2)
}

ifelse(n == 'Ana', 1, 0)

ifelse(r1 < 2, 0, 1)
```

Figura 15: Ejecución condicional.

2.3. Control de flujo

En la figura 14, se muestran distintas formas de iterar sobre una estructura de ejemplo. Adicionalmente, se puede ver que se puede imprimir sin necesidad de iterar por ella.

En lo que se refiere a la ejecución condicional (figura 15), además de la construccón `if-else`, común en los lenguajes de programación, es destacable la posibilidad de tratar todos los elementos de un vector directamente mediante `ifelse`.

2.4. Funciones

En la figura 16, se muestra la definición y uso de una función con un parámetro por defecto. Las funciones devuelven el último valor calculado, aunque se puede indicar explícitamente el resultado

```r
f <- function(a, b = 1) {
  a + b
}

f(2)
f(2, 3)
```

Figura 16: Función con parámetros por defecto.

```r
g <- function(a) {
  return(a + 1)
}
g(2)

h <- function(k, a) {
  k(a)
}
h(g, 2)
h(f, 2)
```

Figura 17: Función a la que se pasa otra función como parámetro.

```r
a <- 2

p <- function(b) {
  b <- b + a
  a <- 1
  return(b)
}

p(a)
a
```

Figura 18: Ámbito de las variables.

mediante `return()`.

Se pueden definir funciones a las que se pasa como parámetro otra función (figura 17) y, dentro de la función, invocar a la función que se ha pasado.

Desde una función se puede acceder a una variable global pero esta no se puede modificar desde la función (figura 18), ni directamente, ni pásandola como parámetro.

2.5. Leer y escribir estructuras de datos

Cuando trabajamos con archivos, es práctico configurar la carpeta de trabajo mediante `setwd()`[9] porque, de esta forma, basta con indicar el nombre del archivo (figura 19). Adicionalmente, se puede observar en la figura 19 que, al escribir y leer un data frame, la estructura que se lee no coincide exactamente con la escrita: se le ha añadido una columna adicional con el número de línea. Si abrimos el archivo generado, se puede comprobar que la columna se ha generado al guardar los datos. Si queremos que no se almacene el número de fila lo tendremos que indicar al guardar el data frame, se

[9] Actualiza el parámetro que se pasa a esta función en la figura 19 con tu carpeta de trabajo.

```r
setwd("~/sig/practicas/pr02")

d
write.csv(d, "marcador.csv")
(res <- read.csv("marcador.csv", header = TRUE, sep = ","))
res$person
class(res)
```

Figura 19: Guardar un data frame en un archivo y volverlo a leer.

```
write.csv(d, "marcador.csv", row.names = FALSE)
(res <- read.csv("marcador.csv", header = TRUE, sep = ",",
                 stringsAsFactors = FALSE))
res$person
```

Figura 20: Guardar y leer un data frame, e interperetar strings como factores.

```
m
write.csv(m, "matriz.csv")
(mat <- read.csv("matriz.csv", header = TRUE, sep = ","))
class(mat)
rownames(mat) <- mat[, 1]
(mat <- as.matrix(mat[, -1]))
class(mat)
```

Figura 21: Guardar y leer una matriz.

hace mediante el parámetro `row.names` (figura 20). Se puede observar que la clase del resultado que obtenemos es un data frame.

En la figura 20, escribe y lee el data frame con el mismo número de columnas. Adicionalmente, se indica que interprete como estring los valores de los factores que lea.

En el caso de guardar y leer una matriz (figura 21), si le hemos dado nombre a las filas, nos interesa guardar sus nombres y recuperarlos al leer los datos. Se puede observar que leemos un data frame, pero lo podemos transformar en una matriz eliminando la columna correspondiente al nombre de las filas y haciendo una conversión de tipos.

2.6. Plots básicos

La función genérica para representar gráficamente objetos en *R* es `plot()` (figura 22). En este caso, estamos representando puntos. La forma de representación de los puntos se determina mediante el parámetro `pch`[10]. Adicionalmente, podemos utilizar otras funciones para obtener otros tipos de representación, como `hist()`, en la figura 22.

La figura 23 muestra cómo usar la misma función `plot()` para representar los puntos unidos mediante una línea. Adicionalmente, se pueden representar puntos en el mismo gráfico.

En la figura 24, se divide la superficie de representación en dos partes mediante la función `par()`, una fila y dos columnas para poder mostrar conjuntamente dos figuras. Posteriormente, cuando se ha hecho la representación, se vuelve a dejar una sola columna mediante la misma función. En las dos representaciones se muestan los mismos datos, la diferencia entre ellas se produce por los parámetros

[10] En `https://www.statmethods.net/advgraphs/parameters.html` se pueden consultar los posibles valores que puede tomar este parámetro y el resultado obtenido en cada caso. También hay información de otros parámetros para definir el estilo de las líneas o los colores.

```
x1 <- rnorm(100)
y1 <- rnorm(100)

plot(x1, y1, pch=16, col='blue')

hist(x1)
```

Figura 22: Diagrama de dispersión e histograma.

```
x2 <- seq(0, 2*pi, len=100)
y2 <- sin(x2)

plot(x2, y2, type='l', lwd=2, col='darkgreen', ylim=c(-1.2, 1.2))
y2r <- y2 + rnorm(100, 0, 0.1)
points(x2, y2r, pch='.', col='darkred')
```

Figura 23: Línea con puntos.

```
# x2 <- seq(0, 2*pi, len=100)
# y2 <- sin(x2)
y3 <- cos(x2)

par(mfrow=c(1,2))

plot(y2, y3, asp=1)
polygon(y2, y3, col='lightgreen')

plot(y2, y3, type='n')
polygon(y2, y3, col='lightgreen')

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

Figura 24: Polígonos y plots múltiples.

`asp` (para mantener la proporcionalidad de la figura) y `type` (para no representar los datos propiamente dichos, sino establecer el área de representación que requieren).

En la figura 25, se muestra el uso de colores y la posibilidad de definir niveles de transparencia (se define mediante el cuarto parámentro de la función `rgb()`, que es opcional).

3. Representación de capas

En el resto del documento, basándonos en las operaciones básicas presentadas en la sección 2, vamos a usar *R* como un SIG.

3. Ejecuta el código de las figuras 28 a 44 (trata de entender cómo se producen los resultados que se obtienen) y, si se producen como resultado imágenes, expórtalas e inclúyelas en un documento

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

plot(c(-1.5,1.5),c(-1.5,1.5),asp=1, type='n')
rect(-0.5,-0.5,0.5,0.5, border=NA, col=rgb(0,0.5,0.5))
rect(0,0,1,1, col=rgb(1,0.5,0.5))

plot(c(-1.5,1.5),c(-1.5,1.5),asp=1, type='n')
rect(-0.5,-0.5,0.5,0.5, border=NA, col=rgb(0,0.5,0.5,0.7))
rect(0,0,1,1, col=rgb(1,0.5,0.5,0.7))

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

Figura 25: Rectángulos, colores y transparencia.

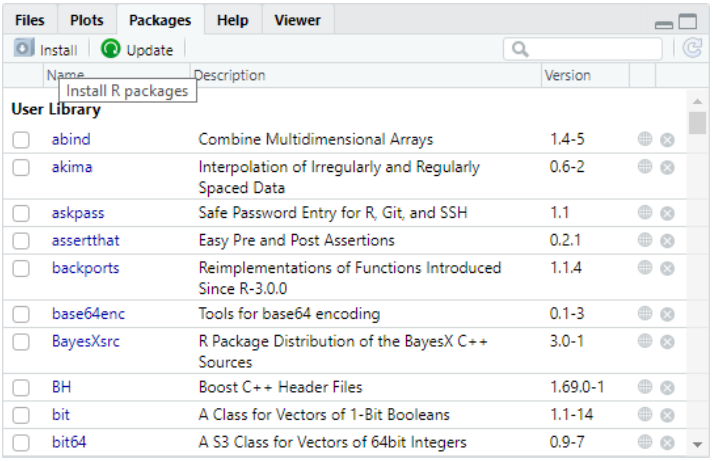

Figura 26: Instalar paquetes.

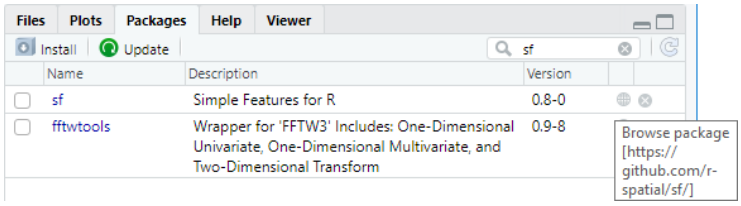

Figura 27: Acceso a la información de un paquete.

indicando el número de la figura (el del fragmento de código que la ha generado) y una breve explicación de lo que se representa. En la figura 37, tal y como se indica en los comentarios asociados a ella, **no olvides seleccionar un municipio distinto al indicado y al seleccionado por el resto de compañeros**[11].

3.1. Añadir capas

La funcionalidad básica de *R* se amplía mediante paquetes (packages) que podemos añadir a la instalación. Los paquetes instalados se pueden ver desde la pestaña «Packages» de la zona inferior derecha de la ventana. Si necesitamos usar un paquete que no está instalado aún, pulsando sobre el botón «Install» (figura 26) podemos indicar que se instale.

Un paquete puede contener funciones y datos. Podemos acceder a la información de un paquete pulsando sobre el primer icono que aparece a la derecha de su nombre y versión, en la pestaña «Packages» (figura 27). Para utilizar el contenido de un paquete fácilmente, tenemos que indicar que se cargue, esto se hace mediante la función `library()`.

En el código de la figura 28, comenzamos cargando el paquete `sf`. Hay muchos paquetes para trabajar con información geográfica. El paquete `sp` se usaba muy frecuentemente. Ahora, `sf` está ocupando su lugar. Muchos paquetes, inicialmente basados en `sp`, se han adaptado a `sf`. Todavía quedan paquetes basados en `sp` que es posible que nos interese usar: podemos usarlos convirtiendo los datos de la estructura `sf` a `sp`.

Mediante la función `st_read()` del paquete `sf` leemos una capa de la carpeta de trabajo. La función `class()` nos muestra que hemos leído un data frame. La función `summary()` nos muestra un resumen de las columnas del data frame. Las columnas son las mismas que muestra *QGIS* en la tabla de atributos pero, adicionalmente, tenemos la columna `geometry` que es la que contiene los datos geográficos propiamente dichos de la capa. Para representar solo la geometría de una capa, esta es la columna que debemos utilizar explícitamente.

La función `st_geometry()` del paquete `sf` nos devuelve el componente `geometry` de una capa[12].

[11] Para asegurarte de que es así, consulta al profesor.

[12] Es recomendable usar esta función en lugar de acceder directamente al componente `geometry` del data frame porque el resultado obtenido con ambos métodos de acceso no siempre es idéntico.

```
library(sf)

setwd("~/sig/practicas/pr02")

censo <- st_read("municipios censo GR.shp")
class(censo)
summary(censo)

hidro <- st_read("red hidrografica GR.shp")
summary(hidro)
```

Figura 28: Leer capas.

```
plot(st_geometry(censo))
plot(st_geometry(hidro), col = "blue", add = TRUE)
```

Figura 29: Representar capas.

Representamos la capa mediante la función `plot()`. Al resultado podemos añadir otras capas mediante la misma función, pero utilizando el parámetro `add` para indicarlo (figura 29).

El resultado se muestra en la zona inferior derecha de la ventana, en la pestaña «Plots». Podemos exportarlo a distintos formatos mediante la opción «Export» (figura 30).

3.2. Cambiar la representación de una capa

Para cambiar la representación de una capa debemos representarla de nuevo. La función `plot()` ofrece muchas posibilidades, algunas de ellas se muestran en la figura 31: hemos cambiado el color, el grosor de las líneas y añadido ejes a la representación.

3.3. Importar una imagen desde la Web

Vamos a importar una imagen desde *OpenStreetMap*[13]. Hay un paquete *R* llamado `OpenStreetMap` que ofrece esta funcionalidad.

La función de `OpenStreetMap` que obtiene un mapa de una zona determinada es `openmap`. Si miramos su definición, trabaja con la latitud y longitud. Deberemos transformar la capa que queramos representar a un sistema de referencia que permita obtener directamente estos datos: por ejemplo, el de código EPSG `4230`. La transformación se puede llevar a cabo mediante la función `st_transform` (figura 32).

A partir de esta representación de la capa, obtenemos los vértices del rectángulo más pequeño que la incluye. Con dos de esos vértices, mediante la función `openmap` (figura 32) obtenemos el mapa de la Web. Como vamos a representar el mapa obtenido conjuntamente con la capa de partida, vamos

[13] `https://www.openstreetmap.org`

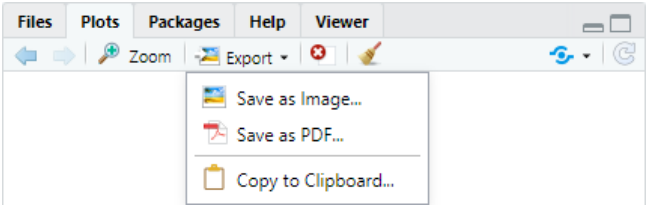

Figura 30: Presentación de las capas.

```
plot(st_geometry(censo), col = "lightgreen", lwd = 1.5, axes = TRUE)
```

Figura 31: Cambiar el color y otros elementos de una capa.

```
library(OpenStreetMap)
? openmap

censo_cg <- st_transform(censo, "+init=epsg:4230")
summary(censo_cg)

# rectángulo más pequeño que incluye la capa
sup_izq <- as.vector(cbind(st_bbox(censo_cg)['ymax'],
                           st_bbox(censo_cg)['xmin']))
inf_der <- as.vector(cbind(st_bbox(censo_cg)['ymin'],
                           st_bbox(censo_cg)['xmax']))

mapa_osm <- openmap(sup_izq, inf_der)
censo_osm <- st_transform(censo_cg, osm())

par_seg <- par()
par(mar = c(0, 0, 0, 0))
plot(mapa_osm)
plot(st_geometry(censo_osm), add = TRUE, lwd = 1.2)
par(par_seg$mar)
```

Figura 32: Importar una imagen desde *OpenStreetMap*.

a transformar la capa para que tenga el mismo sistema de referencia que *OpenStreetMap* (mediante `st_transform`).

Por último, eliminamos los márgenes del área de representación para que el mapa descargado la ocupe por completo (`par(mar = c(0, 0, 0, 0))`), y representamos las dos capas conjuntamente. Es recomendable, antes de eliminar los márgenes, guardar los parámetros y, al acabar, restaurar los márgenes existentes.

Nota importante: En la instalación del aula de prácticas hay un problema debido a que *Java* se ha instalado en la versión de 32 bits y *R* en la de 64, y el paquete `OpenStreetMap` usa *Java*. No es necesario intentar arreglarlo. Procuraré corregirlo para la siguiente versión que genere. Por este motivo, el resultado se muestra en la figura 33.

4. Explorar una capa

Si consultamos una capa mediante la función `class()` (figura 28), podemos comprobar que es un data frame. Podemos acceder a esta estructura como se mostró con ejemplos en la figura 10.

4.1. Acceder a elementos de una capa

En la figura 34, en primer lugar, se muestran los datos de un municipio (accediendo al data frame que almacena los datos); a continuación, se representa gráficamente, de la misma forma que representamos toda la capa; por último, se representan varios municipios seleccionados en el data frame.

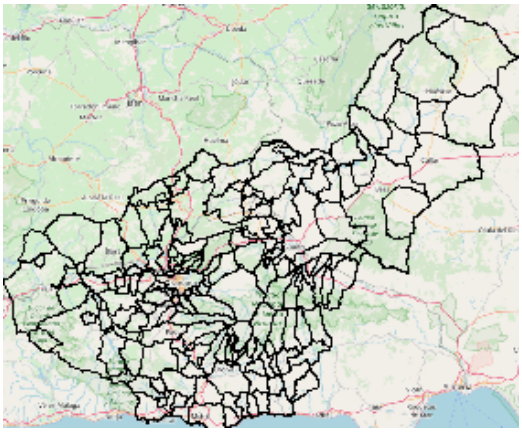

Figura 33: Mapa con imagen obtenida de la Web.

```
censo[censo$municipio == "Lanjarón", ]
plot(st_geometry(censo[censo$municipio == "Lanjarón", ]))

plot(st_geometry(censo[censo$municipio == "Lanjarón" |
                       censo$municipio == "Monachil"|
                       censo$municipio == "Busquístar",]))
```

Figura 34: Acceder a elementos de una capa.

```
censo

censo$Ce2018T
censo[, 'Ce2018T']

censo[, c('Ce2015T', 'Ce2018T')]
censo[censo$municipio == "Lanjarón", c('Ce2015T', 'Ce2018T')]
```

Figura 35: Acceso a la tabla de atributos.

```
censo$Di2018a15T = censo$Ce2018T - censo$Ce2015T
censo[, 'Di2018a15T']
```

Figura 36: Definir una nueva columna.

```
censo_aumenta <- censo[censo$Di2018a15T > 0, ]
plot(st_geometry(censo_aumenta))
st_write(censo_aumenta, "censo aumenta GR.shp", delete_layer = T)

seleccion <- "Lanjarón"
municipio <- censo[censo$municipio == seleccion, ]
st_write(municipio, "municipio.shp", delete_layer = T)

plot(st_geometry(censo), axes = TRUE, main = seleccion)
plot(st_geometry(municipio), col = "lightgreen", add = TRUE)
```

Figura 37: Consultar los datos de una capa y definir una nueva.

4.2. Tabla de atributos de una capa

La tabla de atributos de una capa es un data frame (la capa tiene más componentes). Podemos comprobarlo al mostrar todos sus datos (figura 35). Podemos acceder a los componentes de varias formas, aunque no son equivalentes: por ejemplo, si accedemos a una columna mediante `censo$Ce2018M` obtenemos solo los datos de la columna; si lo hacemos mediante `censo[, 'Ce2018T']` obtenemos adicionalmente la columna `geometry`. También podemos seleccionar varias columnas o combinar la selección de filas y columnas.

4.3. Añadir un campo a la tabla de atributos

Añadir un campo a la tabla de atributos equivale a definir una nueva columna en un data frame (figura 36). No hay que hacer una definición previa de la columna, es suficiente con definir el nombre de la nueva columna junto con la expresión para obtener sus valores.

4.4. Operaciones de consulta sobre una capa

Sobre cualquier capa podemos definir las condiciones que necesitemos, tratándola como un data frame, como se ha hecho en varios de los ejemplos anteriores (figuras 34 y 35).

En la figura 37, seleccionamos datos de una capa y, adicionalmente, los salvamos en un archivo (en un Shapefile). Para guardar el archivo se usa la función `st_write()` con el parámetro `delete_layer` que permite sobreescribir un archivo previamente guardado.

En lugar del municipio seleccionado en la figura 37, selecciona un municipio distinto. El municipio que selecciones será el foco de trabajo de futuras actividades.

5. Añadir elementos a la representación las de capas

Se puede representar en una capa el valor de cualquier columna de la tabla de atributos. En particular, vamos a representar mediante etiquetas los nombres de los municipios y, creando un mapa de coropletas, un valor numérico.

5.1. Mostrar etiquetas

Las etiquetas las mostramos en la figura 38 mediante la función `text()`. A esta función le tenemos que indicar, para cada fila, la posición donde escribir el texto y el campo que contiene el texto a

```
plot(st_geometry(censo), axes = TRUE, main = seleccion)
plot(st_geometry(municipio), col = "lightgreen", add = TRUE)

text(st_coordinates(st_centroid(st_geometry(censo))),
     labels = censo$municipio, pos = 3, cex = 0.5)
```

Figura 38: Mostrar etiquetas de todos los municipios.

```
library(tmap)

tm_shape(censo) +
  tm_fill("white") +
  tm_borders() +
  tm_text("municipio", size = 0.3) +
  tm_shape(municipio) +
  tm_fill("lightgreen") +
  tm_borders(lwd = 2) +
  tm_text("municipio", size = 0.3) +
  tm_layout(frame = FALSE, title = seleccion,
            title.size = 1, title.position = c(0.35, "top"))
```

Figura 39: Mostrar etiquetas de todos los municipios usando el paquete `tmap`.

escribir. Para determinar automáticamente las coordenadas, utilizamos la función `st_centroid()` que devuelve el centroide de la representación de cada municipio; a partir del centoide, mediante la función `st_coordinates()` se obtienen sus coordenadas. El texto que se representa es el contenido en el campo `municipio`. Mediante el resto de parámetros se indica la posición relativa del texto respecto a centroide y su tamaño.

En la figura 39, se muestra una forma alternativa de obtener un mapa con las etiquetas de los nombres; en este caso, mediante las funciones del paquete `tmap`. En la primera línea, mediante la función `tm_shape()`, se indica qué se quiere representar; en las líneas siguientes, se incluyen más detalles o más datos de la representación[14]. En este caso, no se puede ejecutar línea a línea sino que se ejecuta conjuntamente como un todo.

5.2. Representar información numérica

La función `plot()` representa directamente información numérica en forma de mapa de coropletas, basta con indicar explícitamente el campo de la tabla de atributos a utilizar en la representación (figura 40).

En la figura 41 se muestra cómo obtener un resultado similar al de la figura 40 pero con el paquete `tmap`.

[14]Esta es una manera de componer funciones que ofrece el paquete `magrittr` (`https://magrittr.tidyverse.org/`).

```
plot(censo[, "Superficie"], main = "Granada")
```

Figura 40: Mapa para representar la superficie de los municipios.

```
tm_shape(censo) +
  tm_fill("Superficie") +
  tm_borders() +
  tm_layout(frame = FALSE, title = "Granada",
            title.size = 1, title.position = c(0.35, "top"))
```

Figura 41: Mapa para representar la superficie de los municipios usando el paquete `tmap`.

```
png("selección.png")

tm_shape(censo) +
  tm_fill("white") +
  tm_borders() +
  tm_text("municipio", size = 0.3) +
  tm_shape(municipio) +
  tm_fill("lightgreen") +
  tm_borders(lwd = 2) +
  tm_text("municipio", size = 0.3) +
  tm_layout(frame = FALSE, title = seleccion,
            title.size = 1, title.position = c(0.35, "top"))
dev.off()
```

Figura 42: Generar un mapa y guardarlo como una imagen.

6. Generar la salida del proyecto

En los apartados siguientes se va a generar la salida del proyecto usando el paquete `tmap`. Para generar una imagen o un mapa también se puede usar el paquete `sf`.

En todos los casos funciona igual: se redirige la salida al tipo que se desee, se genera el mapa mediante las funciones necesarias y, cuando se haya acabado, se desactiva la salida para que se guarden los cambios.

6.1. Guardar como una imagen

En la figura 42, se muestra cómo generar un archivo png (hay funciones similares para otros formatos gráficos, por ejemplo `jpg()`). En la función `png()` se pueden usar parámetros para indicar diversas características de la imagen que se genera. El proceso de generación se acaba al ejecutar la función `dev.off()`, entonces se guarda el archivo en la carpeta de trabajo.

6.2. Imprimir un mapa

A un mapa le podemos añadir el título, la escala, la flecha del norte, entre otros elementos. Podemos guardarlo como un gráfico o como un archivo en formato PDF, como se hace en la figura 43. En las pruebas que he realizado, la resolución es mucho mejor en formato PDF que en cualquier formato gráfico, por lo que este sería el formato más adecuado para generar un mapa.

6.3. Generar un mapa para la Web

Podemos generar un mapa para la Web usando funciones del paquete `tmap`. Funciona igual que para la generación de un archivo gráfico o en formato PDF: en este caso se usa la función `tmap_mode()` para indicar el inicio y el final del proceso de generación. En este caso, la generación se ha de llevar a cabo mediante funciones del propio paquete.

```
pdf("selección.pdf")

tm_shape(censo) +
  tm_fill("Superficie") +
  tm_borders() +
  tm_text("municipio", size = 0.3) +
  tm_layout(frame = FALSE, title = "Granada",
            title.size = 1, title.position = c(0.35, "top")) +
  tm_compass(type = "8star", position = c("right", "bottom")) +
  tm_scale_bar(breaks = c(0, 25, 50), text.size = 0.5, position = c("right", "bottom"))
dev.off()
```

Figura 43: Generar un mapa con la escala y la flecha del norte.

```
tmap_mode('view')

tm_shape(censo) +
  tm_fill(alpha = 0) +
  tm_borders() +
  tm_text("municipio", size = 0.3) +
  tm_shape(municipio) +
  tm_fill("lightgreen") +
  tm_borders(lwd = 2) +
  tm_text("municipio", size = 0.3) +
  tm_layout(frame = FALSE, title = seleccion,
            title.size = 1, title.position = c(0.35, "top"))
tmap_mode('plot')
```

Figura 44: Generación de un mapa con dos capas para la Web.

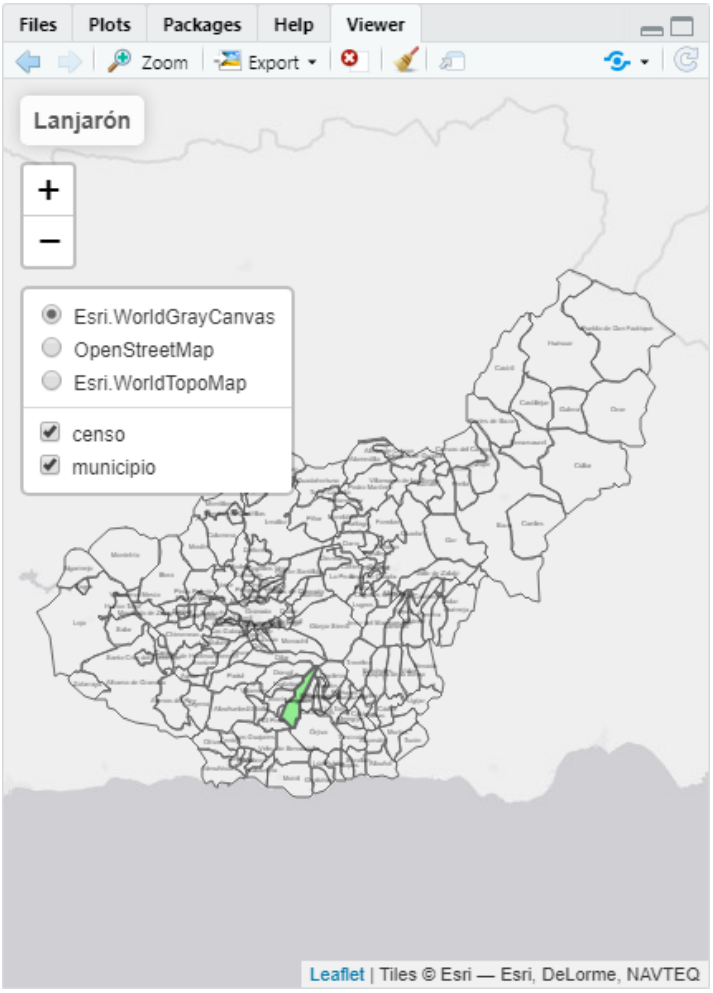

Figura 45: Selección de capas en la salida para la Web.

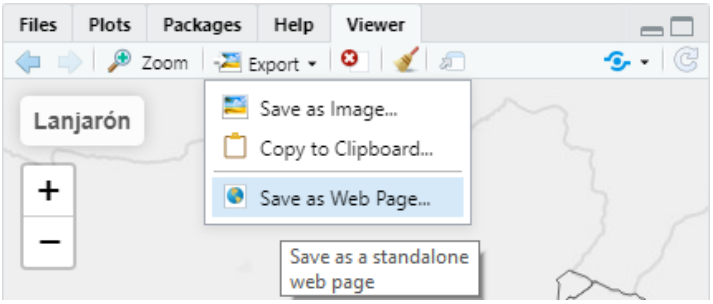

Figura 46: Guardar el resultado como una página Web en el disco local.

En la página Web resultado, podemos seleccionar dinámicamente tanto las capas que se muestran como las que se pueden usar de fondo (figura 45)

Adicionalmente, nos permite guardar el resultado como una página Web local (figura 46), que podemos abrir con nuestro navegador.

Bibliografía

[BC19] Chris Brundsdon and Lex Comber. *An Introduction to R for Spatial Analysis & Mapping (Second Edition)*. SAGE, 2019.

[IDE17] Andalucía IDE. *Infraestructura de Datos Espaciales de Andalucía*. Junta de Andalucía, http://www.ideandalucia.es, 2017.

[INE19] INE. *INEbase / Nomenclátor: Población del Padrón Continuo por Unidad Poblacional*. Instituto Nacional de Estadística, http://www.ine.es/nomen2/index.do, 2019.

[Mat11] Norman Matloff. *The Art of R Programming: A Tour of Statistical Software Design*. no starch press, 2011.