Manipulating and analyzing spatial data in R

Pedro Abellán, Quees College, CUNY

This tutorial provides an introduction to the manage and analysis of spatial data in *R*. It requires basic knowledge of *R*.

As previous step, we will install and load several packages that will help us to work with spatial data

```
install.packages(c("sp","raster","maptools","raster","rgdal"),dep=TRUE)

library(sp)           # Provides classes (objects) and methods for spatial data
library(maptools)     # Tools for reading and manipulating geographic data, especially ESRI shapefiles
library(raster)       # Tools for reading, manipulating and analyzing raster spatial data
library(rgdal)        # Allows project and transform spatial data
```

Here we can download the data that we will use in this tutorial. Download and unzip this file into your working directory.

I. Vector data in R

In **R**, spatial data are handled in complex object classes, which are defined by the **sp** package, from points to lines to polygons (each possibly with attributes) to grids. All spatial objects have 2 slots or components in common: a bounding box (or extent in rasters) and a coordinate system or CRS. Furthermore, each spatial class have additional slots: lists of coordinates for points, coordinates of vertices for polygons, descriptions of dimensions and a matrix of values for rasters, etc. The *str()* function returns the structure of an object.

The simplest spatial data are the stapial points (*SpatialPoints*), which consist of coordinate pairs, and an associated coordinate system. For instance, we can generate a set of points:

```
x <- c(-1.4, -2.2, -1.3, -1.3)
y <- c(37.8, 38, 38.2, 37.9)
xy <- cbind(x,y)
xy

##         x    y
## [1,] -1.4 37.8
## [2,] -2.2 38.0
## [3,] -1.3 38.2
## [4,] -1.3 37.9

class(xy)

## [1] "matrix"
```

and this matrix of 4x2 can ve converted to a *SpatialPoints* object

```
xy.sp = SpatialPoints(xy)
xy.sp

## class       : SpatialPoints
## features    : 4
## extent      : -2.2, -1.3, 37.8, 38.2  (xmin, xmax, ymin, ymax)
## coord. ref. : NA

class(xy.sp)

## [1] "SpatialPoints"
## attr(,"package")
## [1] "sp"
```

Another way of obtaining a *SpatialPoints* object is using the *coordinates()* function, which sets the coordinates of an R dataframe object, and converts it into a *SpatialPointsDataFrame* object.

```
xy.d <- as.data.frame(xy)
coordinates(xy.d) <- c("x","y")
xy.d

## class       : SpatialPoints
## features    : 4
## extent      : -2.2, -1.3, 37.8, 38.2  (xmin, xmax, ymin, ymax)
## coord. ref. : NA
```

Because we have not specified any coordinate system, the CRS (*Coordinate Reference System*) appears as 'empty' (*NA*). It can be also checked by directly asking which is the CRS of our object with the *proj4string()* function:

```
proj4string(xy.sp)

## [1] NA
```

CRS uses *PROJ.4* strings to define projections and coordinate reference systems. *PROJ.4* is an open-source library that provides coordinate reprojection to convert between geographic coordinate systems. More information here; see also spatialreference.org.

We can use the *proj4string()* function to set the CRS of our spatial object:

```
proj4string(xy.sp) <- "+proj=longlat +datum=WGS84 +no_defs +ellps=WGS84 +towgs84=0,0,0"      # PROJ.4
argument for geographic (log/lat) coordinates and WGS84 datum
proj4string(xy.sp)

## [1] "+proj=longlat +datum=WGS84 +no_defs +ellps=WGS84 +towgs84=0,0,0"
```

Similarly, if we write now *xy.sp* we can see that it has its associated coordinate system.

```
xy.sp

## class       : SpatialPoints
## features    : 4
## extent      : -2.2, -1.3, 37.8, 38.2  (xmin, xmax, ymin, ymax)
## coord. ref. : +proj=longlat +datum=WGS84 +no_defs +ellps=WGS84 +towgs84=0,0,0
```

The *spTransform()* function provides transformation between datum(s) and conversion between projections (also known as projection and/or re-projection) for vector data, from one unambiguously specified coordinate reference system to another, using *PROJ.4* projection arguments.

```
ed50 <-  "+proj=utm +zone=30 +ellps=intl +units=m +no_defs"      # PROJ.4 argument often used for spatial
data in Spain. It uses a UTM projection, and the International ellipsoid.
xy.sp.ed50 <- spTransform(xy.sp, CRS(ed50))
xy.sp.ed50

## class       : SpatialPoints
## features    : 4
## extent      : 570241.6, 649470, 4184900, 4229442  (xmin, xmax, ymin, ymax)
## coord. ref. : +proj=utm +zone=30 +ellps=intl +units=m +no_defs
```

We can see how our spatial points have changed

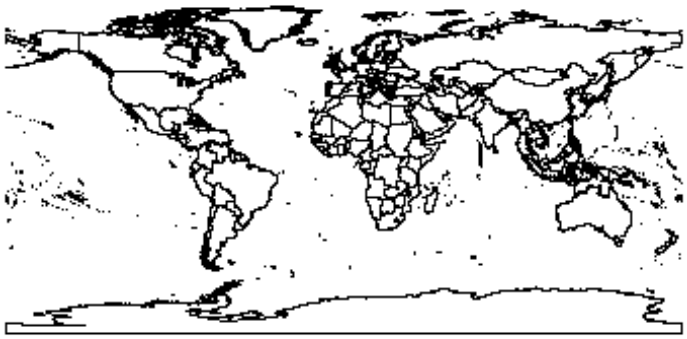
```
coordinates(xy.sp)

##         x    y
## [1,] -1.4 37.8
## [2,] -2.2 38.0
## [3,] -1.3 38.2
## [4,] -1.3 37.9

coordinates(xy.sp.ed50)
```

```
##             x       y
## [1,] 640867.3 4184900
## [2,] 570241.6 4206187
## [3,] 648861.0 4229442
## [4,] 649470.0 4196152
```

Now we will work with *ESRI shapefiles*, a popular geospatial vector data format for geographic information system. A *shapefile* consists actually of a collection of files with a common filename prefix, stored in the same directory. The three mandatory files have filename extensions *.shp*, *.shx*, and *.dbf*. It can include other optional files, such as *.prj*, which is a plain text file providing the coordinate system and projection information. A *shapefile* represents a list of spatial objects -a list of points, a list of lines, or a list of polygonal regions- and each object in the list may have additional variables attached to it.

The *readShapePoly()* function in the **maptools** package reads data from a polygon shapefile into a *SpatialPolygonsDataFrame* object. The *sp* package provides a number of useful functions to work with this class of objects. For instance, we can plot our *shapefile*. The *readShapePoints()* y *readShapeLines()* functions allow reading point and line shapefiles, respectively.

```
world <- readShapePoly("countries.shp") # Global country boundaries
plot(world)   # We can add axes with axes=TRUE, and set the color with col()
```



However, I strongly recommend using the **rgdal** package, which provides bindings to the open source GDAL (Frank Warmerdam's Geospatial Data Abstraction Library for raster data), proj4, and OGR (the complement to gdal for vector data). *readGDAL()* and *readOGR()* provide seamless access to raster and vector file formats. The only trick is that specification of directories v. file names is a bit tricky, as GDAL & OGR drivers for different file formats interpret dsn and layer differently (sometimes as directory and file, sometimes as file and layer within file).

```
world <- readOGR(".", layer= "countries")  # the first argument provides information on the directory of
the target file. "." indicates that the file is in the current working directory.

## OGR data source with driver: ESRI Shapefile
## Source: ".", layer: "countries"
## with 265 features and 18 fields
## Feature type: wkbPolygon with 2 dimensions
```

We can see the structure of our spatial object using the *summary()* function and see its attribute table with *my_shapefile@data*. The names() function will show us the fields included in its attribute table.

```
names(world)

##  [1] "OBJECTID"  "NAME"      "ISO3"      "ISO2"      "FIPS"
##  [6] "COUNTRY"   "ENGLISH"   "FRENCH"    "SPANISH"   "LOCAL"
## [11] "FAO"       "WAS_ISO"   "SOVEREIGN" "CONTINENT" "UNREG1"
## [16] "UNREG2"    "EU"        "SQKM"

head(world@data)
```

```
##   OBJECTID           NAME ISO3 ISO2 FIPS        COUNTRY        ENGLISH
## 0        1         Ã…land  ALA   AX   AX         Ã…land         Ã…land
## 1        2    Afghanistan  AFG   AF   AF    Afghanistan    Afghanistan
## 2        3        Albania  ALB   AL   AL        Albania        Albania
## 3        4        Algeria  DZA   DZ   AG        Algeria        Algeria
## 4        5 American Samoa  ASM   AS   AQ American Samoa American Samoa
## 5        6        Andorra  AND   AD   AN        Andorra        Andorra
##               FRENCH         SPANISH          LOCAL            FAO WAS_ISO
## 0               <NA>            <NA>         Ã…land           <NA>    <NA>
## 1        Afghanistan     AfganistÃ¡n    Afghanestan    Afghanistan    <NA>
## 2            Albanie         Albania      Shqiperia        Albania    <NA>
## 3          AlgÃ©rie         Argelia     Al Jaza'ir        Algeria    <NA>
## 4 Samoa AmÃ©ricaines Samoa Americana American Samoa American Samoa    <NA>
## 5            Andorre         Andorra        Andorra        Andorra    <NA>
##       SOVEREIGN CONTINENT          UNREG1  UNREG2 EU         SQKM
## 0       Finland    Europe Northern Europe  Europe  0    1243.7191
## 1   Afghanistan      Asia   Southern Asia    Asia  0  641383.4228
## 2       Albania    Europe Southern Europe  Europe  0   28486.1096
## 3       Algeria    Africa Northern Africa  Africa  0 2316558.6585
## 4 United States   Oceania       Polynesia Oceania  0     211.0151
## 5       Andorra    Europe Southern Europe  Europe  0     474.6375
```

Again, with the *proj4string()* function we can know the CRS of this object.

```
proj4string(world)
```

```
## [1] "+proj=longlat +datum=WGS84 +no_defs +ellps=WGS84 +towgs84=0,0,0"
```

In case the *shapefile* does not include the *.prj* file, if we know its CRS we could set it with this function.

Exercise 1

Get the list of countries. How many are there? Plot the world map with a different color for each country. Obtain a new spatial object including only the polygons of Spain. Name it 'spain'. Plot it with longitude and latitude axes. Project this object to ED50 (name it 'spain.ed50') and save it as a new *shapefile* using *writeOGR()*.

II. Raster data in R

A raster object in **R** (*RasterLayer*) represents single-layer (variable) raster data and stores a number of parameters that describe it. These include the number of columns and rows, the coordinates of its spatial extent ('bounding box'), and the coordinate reference system. In addition, a *RasterLayer* can store information about the file in which the raster cell values are stored (if there is such a file). A *RasterLayer* can also hold the raster cell values in memory.

The **raster** package has functions for creating, reading, manipulating, and writing raster data. A notable feature of the **raster** package is that it can work with raster datasets that are stored on disk and are too large to be loaded into memory (RAM). The package can work with large files because the objects it creates from these files only contain information about the structure of the data, such as the number of rows and columns, the spatial extent, and the filename, but it does not attempt to read all the cell values in memory.

We can create a raster object (*RasterLayer* object) with the *raster()* function.

```
r <- raster(nrow=18,ncol=36)    # raster with 18 rows and 36 columns
r
```

```
## class       : RasterLayer
## dimensions  : 18, 36, 648  (nrow, ncol, ncell)
```

```
## resolution  : 10, 10  (x, y)
## extent      : -180, 180, -90, 90  (xmin, xmax, ymin, ymax)
## coord. ref. : +proj=longlat +datum=WGS84 +ellps=WGS84 +towgs84=0,0,0
```

We can get the parameters of our raster with different functions (for instance *ncol()* o *ymax()*)

```
ncol(r)

## [1] 36

ymax(r)

## [1] 90
```

We can see that our raster has 648 cells (36x18). By default, it has a global spatial extension and a longitude/latitude coordinate reference system with WGS84 datum.

The objects 'r' created in the example above only consist of a 'skeleton', that is, we have defined the number of rows and columns, and where the raster is located in geographic space, but there are no cell-values associated with it. We can set it values with the *values()* function.

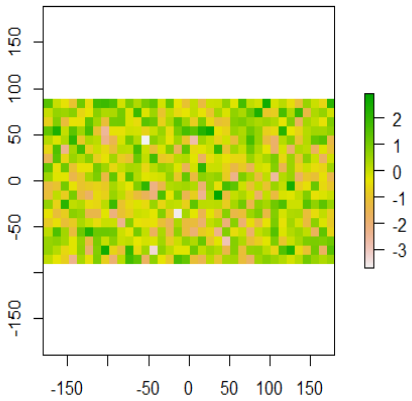
```
hasValues(r)

## [1] FALSE

values(r) <- 1:ncell(r)
```

or for example also:

```
values(r) <- rnorm(ncell(r)) # assign random values from a normal distribution
hasValues(r)

## [1] TRUE

r

## class       : RasterLayer
## dimensions  : 18, 36, 648  (nrow, ncol, ncell)
## resolution  : 10, 10  (x, y)
## extent      : -180, 180, -90, 90  (xmin, xmax, ymin, ymax)
## coord. ref. : +proj=longlat +datum=WGS84 +ellps=WGS84 +towgs84=0,0,0
## data source : in memory
## names       : layer
## values      : -3.68987, 2.9297  (min, max)
```

We can plot our raster with the *plot()* function.

```
plot(r)
```



When we modify the spatial extent of our raster we will modify also its resolution (unless we modify accordingly its number of rows and columns).

```
res(r)

## [1] 10 10

xmin(r) <- 0
xmax(r) <- 6
ymin(r) <- 10
ymax(r) <- 20
res(r)

## [1] 0.1666667 0.5555556
```

Similarly, a change in the resolution affects the number of rows and columns, and viceversa.

```
res(r) <- 1
r

## class       : RasterLayer
## dimensions  : 10, 6, 60  (nrow, ncol, ncell)
## resolution  : 1, 1  (x, y)
## extent      : 0, 6, 10, 20  (xmin, xmax, ymin, ymax)
## coord. ref. : +proj=longlat +datum=WGS84 +ellps=WGS84 +towgs84=0,0,0
```

As you can see now, when you change the resolution or the number of columns or rows, you will lose the values associated with the *RasterLayer* (if there were any). We can also set it values with the *setValues()* function:

```
r <- setValues(r, rnorm(ncell(r)))  # this function allows also creating a new raster object
```

Actually, cell values in the raster object are stored as a (one-dimensional) vector, which can be accessed with several methods. Use the *getValues()* function to get all cells values.

```
r.values <- getValues(r)
r.values

##  [1] -1.77955303 -0.25981310  1.19721101  0.90158540  0.01154251
##  [6]  1.19483113  0.88943247 -0.06005183 -0.12906653  0.72596657
## [11] -1.82395950 -0.89368454  1.74198416 -0.98142918  0.26609051
## [16]  0.17962103  1.05808699 -1.07985861 -2.72688604 -0.42535288
## [21] -1.19982857 -0.13929516  1.10140769 -1.28988236  1.79883896
## [26] -0.59895439 -0.52176514 -0.33823693 -0.69114746 -0.69685001
## [31]  0.50046036  0.18172752  0.30720007  1.70144510  0.80851924
## [36] -0.14386389 -0.91355649  2.42372408  0.85278325 -0.99289260
## [41]  0.83373821 -0.82878632  1.13059994 -0.77359221 -1.12738368
## [46] -1.98480045 -1.34262327 -1.72758149 -0.27153853 -0.35222119
## [51]  0.69906111 -1.88759595 -1.62665589  0.68919942 -1.13595667
## [56]  0.15475757 -0.15225237  0.93516121  0.40309696 -1.78297283

r.values[1:10]

##  [1] -1.77955303 -0.25981310  1.19721101  0.90158540  0.01154251
##  [6]  1.19483113  0.88943247 -0.06005183 -0.12906653  0.72596657
```

By writing *values(r)* we can also see the values of each cell. In addition, you can use standard **R** indexing to access values.

```
r[1:5]

## [1] -1.77955303 -0.25981310  1.19721101  0.90158540  0.01154251
```

Values can also be inspected using a (two-dimensional) matrix notation. As for R matrices, the first index represents the row number, the second the column number.

```
r.matrix <- as.matrix(r)
r.matrix
```

```
##             [,1]        [,2]       [,3]       [,4]        [,5]       [,6]
##  [1,] -1.7795530 -0.25981310  1.1972110  0.9015854  0.01154251  1.1948311
##  [2,]  0.8894325 -0.06005183 -0.1290665  0.7259666 -1.82395950 -0.8936845
##  [3,]  1.7419842 -0.98142918  0.2660905  0.1796210  1.05808699 -1.0798586
##  [4,] -2.7268860 -0.42535288 -1.1998286 -0.1392952  1.10140769 -1.2898824
##  [5,]  1.7988390 -0.59895439 -0.5217651 -0.3382369 -0.69114746 -0.6968500
##  [6,]  0.5004604  0.18172752  0.3072001  1.7014451  0.80851924 -0.1438639
##  [7,] -0.9135565  2.42372408  0.8527833 -0.9928926  0.83373821 -0.8287863
##  [8,]  1.1305999 -0.77359221 -1.1273837 -1.9848005 -1.34262327 -1.7275815
##  [9,] -0.2715385 -0.35222119  0.6990611 -1.8875960 -1.62665589  0.6891994
## [10,] -1.1359567  0.15475757 -0.1522524  0.9351612  0.40309696 -1.7829728
```

We can also use the *raster()* function to create a *RasterLayer* object from a raster file. The **R** package **Raster** can use raster files in several formats, including some 'natively' supported formats and other formats via the rgdal package. Supported formats for reading include *GeoTIFF*, *ESRI*, *ENVI*, and *ERDAS*. Most formats supported for reading can also be written to.

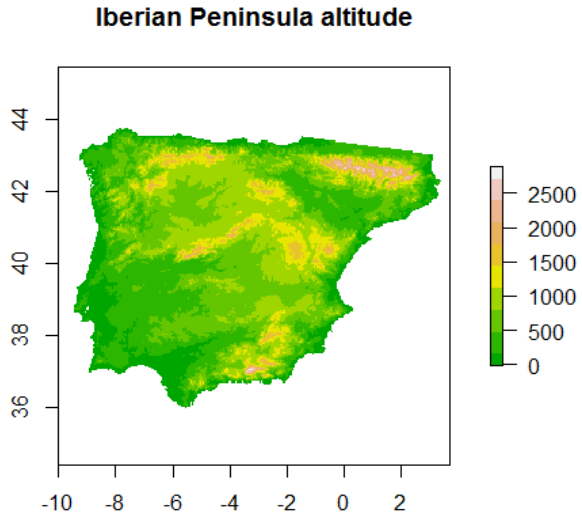
```
alt <- raster ("alt.tif") # raster file of altitud for the Iberian Peninsula at 2.5 arc-minutes resolution
alt
```

```
## class       : RasterLayer
## dimensions  : 195, 329, 64155  (nrow, ncol, ncell)
## resolution  : 0.04166667, 0.04166667  (x, y)
## extent      : -10, 3.708333, 35.875, 44  (xmin, xmax, ymin, ymax)
## coord. ref. : +proj=longlat +datum=WGS84 +no_defs +ellps=WGS84 +towgs84=0,0,0
## data source : C:\Users\huertoroenas\Dropbox\in_progress\R_classes\Spatial Data Analysis with the
R\clase\alt.tif
## names       : alt
## values      : -10, 2889  (min, max)
```

We can see that now the raster does not handle in memory the data, but just the path of the file in which the raster cell values are stored.

When plotting our raster, we can apply different colour schemes using the function *col=*, for example with *terrain.colors()*, *topo.colors()*, *heat.colors()*, *rgb()*, *rainbow()*, *divPalette()*, etc. With *click()* and *zoom()* we can examine the altitude patterns.

```
plot(alt, col=terrain.colors(10), main="Iberian Peninsula altitude")
```



Again, we can use the *proj4string()* to access the coordinate reference system of our raster.

```
proj4string(alt)

## [1] "+proj=longlat +datum=WGS84 +no_defs +ellps=WGS84 +towgs84=0,0,0"
```

As we did with the vector spatial data, we can change the coordinate reference system of a raster (i.e. project it). However, this is a little more complicated in this case.

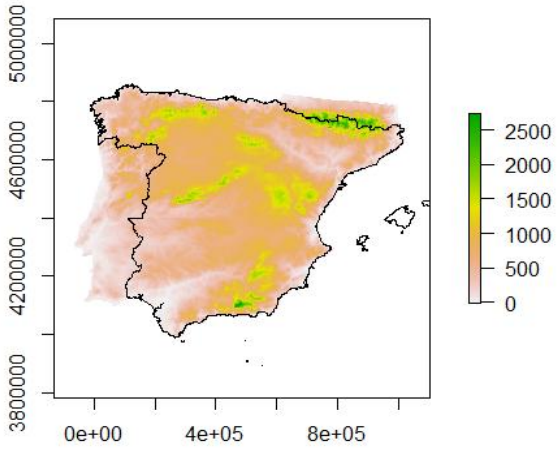
First, we have to project the spatial extent ('bounding box') of our raster using the function *projectExtent()*, which returns us a *RasterLayer* object with a projected extent, but without any values.

This *RasterLayer* can then be adjusted (e.g. by setting its resolution) and used as a template 'to' in the *projectRaster()* function. If we had already a raster object with the parameters to which we want to project our raster, we could use it directly as a template in the *projectRaster()* function.

```
ext.ed50 <- projectExtent(alt, ed50)
ext.ed50

## class       : RasterLayer
## dimensions  : 195, 329, 64155  (nrow, ncol, ncell)
## resolution  : 3763.258, 4747.143  (x, y)
## extent      : -132410.2, 1105702, 3970148, 4895841  (xmin, xmax, ymin, ymax)
## coord. ref. : +proj=utm +zone=30 +ellps=intl +units=m +no_defs

res(ext.ed50)<- xres(ext.ed50)    # we make the cells square, this is, with the same height and width.
alt.ed50 <- projectRaster(alt, ext.ed50)   # we project our raster 'alt' using the parameters of the
projected extent (as a template).
```

If everything was okay, our raster should match our *shapefile* of Spain that we previously projected to ed50 ("spain.ed50"):

```
plot(alt.ed50)
plot(spain.ed50, add=TRUE)  # because it is a spatial object, we can add it over our raster with the
option add=TRUE.
```



We can save our raster with the *writeRaster(x, filename, format, ...)* function. Some formats: "raster" (formato nativo), "ascii" (ESRI Ascii), "IDRISI" (Idrisi), "GTiff" (GeoTiff), etc.

```
writeRaster(alt.ed50, "alt_ed50", format="ascii")

## class       : RasterLayer
## dimensions  : 246, 329, 80934  (nrow, ncol, ncell)
## resolution  : 3763.258, 3763.258  (x, y)
## extent      : -132410.2, 1105702, 3970079, 4895841  (xmin, xmax, ymin, ymax)
## coord. ref. : NA
## data source : C:\Users\huertoroenas\Dropbox\in_progress\R_classes\Spatial Data Analysis with the
```

```
R\clase\alt_ed50.asc
## names       : alt_ed50
```

With the *raster()* function we can also create a *RasterLayer* object from a previous object (for example another *RasterLayer*). In this case, the new object presents the same parameters (extent, resolution...) than the object that we use as template but not the cell-values associated with it.

```
new.r <- raster(alt)
new.r

## class       : RasterLayer
## dimensions  : 195, 329, 64155  (nrow, ncol, ncell)
## resolution  : 0.04166667, 0.04166667  (x, y)
## extent      : -10, 3.708333, 35.875, 44  (xmin, xmax, ymin, ymax)
## coord. ref. : +proj=longlat +datum=WGS84 +no_defs +ellps=WGS84 +towgs84=0,0,0
```

Other useful functions to manipute rasters:

- *aggregate(raster, fact=2, fun=mean, ...)* # Aggregate a Raster* object to create a new RasterLayer with a lower resolution (larger cells). The value for the resulting cells is computed with a user-specified function (e.g. mean).
- *extent(x, ...)* # Returns an *Extent* object; x can be a matrix, or a vector of four numbers (xmin, xmax, ymin, ymax)

- *crop(raster, extent, ...)* # Returns a geographic subset of an object as specified by an *Extent* object
- *reclasify(raster, rcl, ...)* # Reclassify values of a *Raster* object. Reclassification is done with matrix *rcl*, in the row order of the reclassify table.

R allows also working with multiple raster layers simultaneously. A *RasterStack* is a collection of *RasterLayer* objects with the same spatial extent and resolution. With the *stack()* function we can create a *RasterStack* object providing a vector of *RasterLayer* objects*.

```
tma <- raster("bio1_tma.tif")    # raster file of annual mean temperature for the Iberian Peninsula.
tmax <- raster("bio5_tmax.tif")   # raster file of maximum temperature of the warmest month for the
Iberian Peninsula.
prec <- raster("bio12_aprec.tif") # raster file of annual precipitation for the Iberian Peninsula.
z <- stack(c(tma,tmax,prec))
```

Now we can, for example, aggregate the cells of all the rasters at once.

```
z2 <- aggregate(z, fact=3)
prec2 <- z2[[3]]   # By this way we can recover any of the rasters in our stack.
prec2
plot(prec2)
```

Exercise 2

Obtain a raster of altitude for only the southeastern quarter (approximately) of the Iberian Peninsula (name it "alt2"), then obtain a new raster with a resolution of 0,25 degrees (name it "alt3"). Finally, obtain a new raster (from the previous one) in which the cell values are grouped in 500 meters intervals with new values among 1 and 4 (1 for 0-500 m, 2 for 500-1000 m, etc.)

III. Introduction to the spatial analysis in R

Many generic functions that allow for simple and elegant raster algebra have been implemented for raster objects in the *R* package *Raster*. In these functions you can mix raster objects with numbers, as long as the first argument is a raster object.

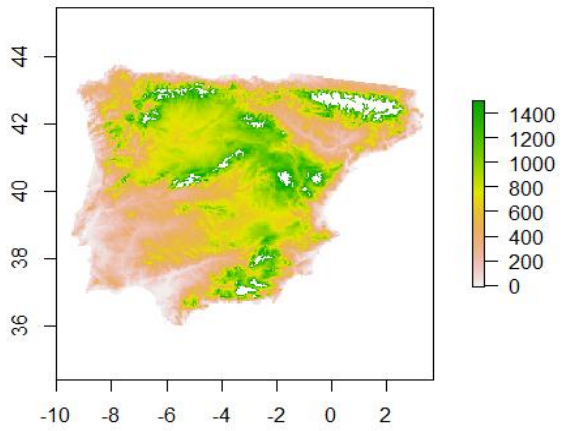
```
r <- raster(ncol=10,nrow=10)    # We create a empty raster
values(r) <- 1:ncell(r) # We set its values
s <- r + 10
s <- sqrt(s)
s <- s * r + 5
s <- round(s)     # round the values

r[] <- runif(ncell(r))
r <- round(r)
```

Furthermore, the *getValues()* function can be useful to other analysis working directly with the cells values. Imagine, for example, that we want to know how two spatial variables are correlated in our study area (Iberia, for example):

```
values.tma <- getValues(tma)     # remind that "tma" is a raster file of annual mean temperature for the
Iberian Peninsula.
values.tmax <- getValues(tmax)   # remind that "tmax" is a raster file of maximum temperature of the
warmest month for the Iberian Peninsula.
cor(values.tma,values.tmax, use="complete.obs")   # with the cor() function we get the correlation
coeficient between both vectors. With the option use="complete.obs" we leave out the NAs.

## [1] 0.7245132
```

We can also get the information and manipulate the vector with the cell values of a raster. Then, with the *setValues()* function we can set the new values to a new raster.

```
values.alt <- getValues(alt)
values.alt[which(values.alt>1500)] <- NA
alt.new <- setValues(alt, values.alt)
plot(alt.new)
```



And we can obtain different statistics for the vector with the cell values:

```
mean(values.tma, na.rm=T) # mean value of annual mean temperature in Iberia. With na.rm=T  we leave out
the NAs.

## [1] 133.292

max(values.tma, na.rm=T)  # maximun value of annual mean temperature.

## [1] 184.375
```

```
min(values.tma, na.rm=T)  # minimum value of annual mean temperature.

## [1] 23.8125
```

We can get the same result with the *cellStats()* function of the package **raster**.

```
cellStats(tma, stat="mean", na.rm=TRUE)

## [1] 133.292

cellStats(tma, stat="max", na.rm=TRUE)

## [1] 184.375

cellStats(tma, stat="min", na.rm=TRUE)

## [1] 23.8125
```

In ecological studies, we often work with sampling localities, which usually have associated spatial information (for example coordinates from a GPS device)

We can associate values for environmental variables from spatial layers (raster) to our spatial points.

Imagine that the table "points.txt" represents 1000 sampling localities.

```
p <- read.table("points.txt", header=TRUE)
head(p)

##   ID          X        Y
## 1  1 -4.8737452 42.02986
## 2  2 -5.5130233 42.91814
## 3  3 -5.6860470 38.23044
## 4  4  0.1961967 41.20043
## 5  5 -7.5899741 43.02702
## 6  6 -8.3687907 38.71001
```

We can start converting our table into a *SpatialPoints* object and setting a CRS:

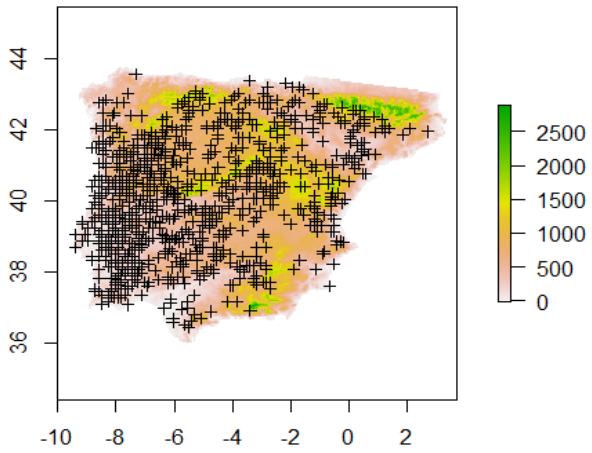
```
coordinates(p) <- c("X","Y")
proj4string(p) <- "+proj=longlat +datum=WGS84 +no_defs +ellps=WGS84 +towgs84=0,0,0"
```

With the *pointDistance(p1, p2, longlat, ...)* function we can calculate the distance between two points (or set of points).

```
pointDistance(p[1,], p[2,], longlat=T)  # calculates the distance between the fist and second points in
our table

##        Y
## 111950.1

pointDistance(p[1:7,], longlat=T)  # calculates the distance between the fist ten points in
our table

##           [,1]     [,2]     [,3]     [,4]     [,5]     [,6]     [,7]
##  [1,]      0.0       NA       NA       NA       NA       NA       NA
##  [2,] 111950.1      0.0       NA       NA       NA       NA       NA
##  [3,] 428556.7 522037.6      0.0       NA       NA       NA       NA
##  [4,] 431858.3 508997.2 602289.9      0.0       NA       NA       NA
##  [5,] 248929.1 169597.4 557611.5 674053.9      0.0       NA       NA
##  [6,] 473659.8 526537.0 239823.3 781247.5 485011.9      0.0       NA
##  [7,] 492772.7 564224.1 156504.4 742866.6 551828.7 106388.6      0.0
```

If we plot our altitude raster with our spatial points we will get something like this:

```
plot(alt)
plot(p, cex=0.8, add=TRUE)
```



We can obtain, for a rater or group of rasters (stack), the values of those cells where our sampling localities or spatial points fall with the *extract()* function

```
p.alt <- extract(alt,p)  # the first element is a raster layer and the second one a set of spatial points
p.alt[1:10]

##  [1]  757 1374  535  229  441  140  193  308  230  759

p.clima <- extract(z,p)     # remain that z was a RasterStack object with several climatic layers
head(p.clima)

##      bio1_tma bio5_tmax bio12_aprec
## [1,]  118.625  285.0625    456.9375
## [2,]   81.125  234.5000    848.6875
## [3,]  155.500  345.7500    627.3125
## [4,]  149.125  301.3125    441.3750
## [5,]  125.375  241.8125   1044.3750
## [6,]  167.000  298.4375    650.1875
```

By this way we can obtain a table with our localities and the different variables for, for example, subsiguient statistical analysis.

```
p.loc <- as.data.frame(p) # convert our points into a R dataframe
p.variables <- cbind(p.loc,p.alt,p.clima)  # combine everything in a single table
head(p.variables)

##            x        y ID p.alt bio1_tma bio5_tmax bio12_aprec
## 1 -4.8737452 42.02986  1   757  118.625  285.0625    456.9375
## 2 -5.5130233 42.91814  2  1374   81.125  234.5000    848.6875
## 3 -5.6860470 38.23044  3   535  155.500  345.7500    627.3125
## 4  0.1961967 41.20043  4   229  149.125  301.3125    441.3750
## 5 -7.5899741 43.02702  5   441  125.375  241.8125   1044.3750
## 6 -8.3687907 38.71001  6   140  167.000  298.4375    650.1875
```

We can also use the *extract()* function to extract, for each point, the cell values of a raster within a specific radius or *buffer*.

```
tma.buffer <- extract(tma, p[1:10,], buffer = 50000) # obtain a list object in which each element includes the cell values within the specified distance

mean.tma.buffer <- extract(tma, p[1:10,], buffer = 50000, fun=mean) # we can also calculate, for example, the mean value for each point.

mean.tma.buffer
```

```
##  [1] 117.11694  93.63105 156.38616 144.57292 119.87083 166.56552 165.79241
##  [8] 162.78448 154.46250 123.06452
```

We can also use the *extract()* function to extract cell values within polygons of a *shapefile*.

```
tma.spain <- extract(tma, spain)  # extract the cell values within the polygon of Spain that we obtained
previously.
extract(tma, spain, fun=mean)   # canculate directly the mean value

## [1] NA
```

The R package **raster** provides other ways to access to the cells of a raster and their values.

```
cells <- cellFromRowCol(tma, 25, 35:39) # obtain the cells of specific column(s) and row(s) of a raster.
coord = xyFromCell(tma, cells) # otain the coordinates of specific cells of a raster.
head(coord)

##              x        y
## [1,] -4.416658 40.25001
## [2,] -4.249991 40.25001
## [3,] -4.083324 40.25001
## [4,] -3.916657 40.25001
## [5,] -3.749991 40.25001
```

Similarly, we can extract the cells within a specific spatial extent.

```
ext <- extent(c(-4,-2,37,38))   # create a Extent object
cells2 <- cellsFromExtent(tma, ext)
```

In addition, we can use standard **R** indexing to access values, or to replace values (assign new values to cells) in a Raster object.

```
tma[cells]

## [1] 137.6250 144.8125 141.0625 140.8125 140.5000

which(tma[]>180)

##  [1] 2750 3082 3165 3166 3167 3249 3250 3332 3333 3386 3387 3388 3417 3468
## [15] 3469 3470 3497 3552 3553 3554 3637 3663 3827 3899 3981

mean(tma[cells2], rm.na=T)

## [1] 137.5764
```

Exercise 3

The geographic coordinates of Madrid are 3.41 W, 40.24 N. Plot them over the altitude raster. Obtain the altitude of Madrid. The *cellFromXY()* function gets the X and Y coordinates of a point and returns the cell index for that point. Then, you can use this index to get the altitude value from the raster "alt"

Finally, the **R** package **raster** includes the fucntion *getData()*, which allows obtaining useful spatial data, such as altitude and clima for everywhere.

```
elevation <- getData("alt", country = "ESP")   # altitude data for countries
tmin <- getData('worldclim', var='tmin', res=0.5, lon=5, lat=45)   # climate data from
Worldclim database
```

Final exercise

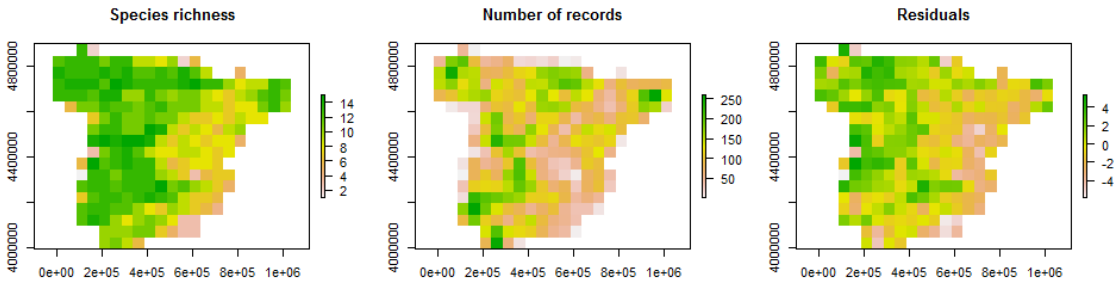
The file "amphibians.txt" contains the location (in geographic coordinates and WGS84 datum) corresponding to amphibian records in Spain (source: Spanish National Biodiversity Inventory).

Obtain a raster map displaying the number of amphibian records. This raster (whose extent will match the extent of the records), should be in UTM projection, GRS80 ellipsoid and ETRS89 for the zone 30 N (http://spatialreference.org/ref/epsg/25830/) and have a resolution of 50 km. The function *table()* can be useful to group the data of our table.

Obtain also a similar raster map displaying species richness, and assess the relation between species richness and sampling effort (number of records) with a linear regresion analysis (*lm()" function).

Obtain a new raster displaying the residuals of this relationship (this raster will allow identifying, for example, areas with high species richness despite low sampling effort).

The obtained raster maps should look something like these (but not necessarily identical):



Useful resources

Start with the R Spatial Task View

The book Spatial Data Analysis in R (Bivand et al., 2008) can be very useful too.

Some useful tutorials and vignettes:

- Introduction to the 'raster' package
- Handling shapefiles in the spatstat package
- Species distribution modeling with R

Some useful links:

- R wiki: tips for spatial data
- R Spatial Tips
- Spatial Data in R
- create-maps-with-maptools
- maps_with_r
- Using R as a GIS

Exercise solutions

Please note that in R there are many ways to accomplish a goal.

Exercise 1

```
dim(world@data)
plot(world, col=1:265)

which(world$COUNTRY=="Spain")  # identify the position of Spain in the attribute table

spain <- world[223,]  # option 1
spain <- world[world$COUNTRY=="Spain",]  # option 2

plot(spain, axes=T)
ed50 <-  "+proj=utm +zone=30 +ellps=intl +units=m +no_defs"
spain.ed50 <- spTransform(spain, CRS(ed50))
writeOGR(spain.ed50, ".", "spain.ed50", driver = "ESRI Shapefile")
```

Exercise 2

```
ext <- extent(-5,1,36,40)          # xmin, xmax, ymin, ymax
alt2 <- crop(alt, ext)
0.25/res(alt)
alt3 <- aggregate(alt2, fact=6)
m <- c(0, 500, 1,  500, 1000, 2,  1000, 1500, 3,  1500, 2000, 4)
rclmat <- matrix(m, ncol=3, byrow=TRUE)
rc <- reclassify(alt3, rclmat)
```

Exercise 3

```
plot(alt)
points(-3.41, 40.24)
cell.madrid <- cellFromXY(alt,c(-3.41, 40.24))
alt[cell.madrid]
```

FINAL EXERCISE

```
# The first step is loading and transforming the data to etrs89
amp <- read.table("amphibians.txt", header=T)  # load tablewith distribution records for amphibians
coordinates(amp) <- c("X","Y")    # convert points to a SpatialPoints object
proj4string(amp) <- "+proj=longlat +datum=WGS84"   # set the CRS(geographic coordinates and WGS84 datum)
etrs89 <- "+proj=utm +zone=30 +ellps=GRS80 +units=m +no_defs"  # create an object with CRS parameters
amp.etrs89 <- spTransform(amp, CRS(etrs89))  # project points to etrs89

# Next step: set the spatial extent of our raster
min_x <- min(amp.etrs89@coords[,1])  # identify the minimum x coordinate in our point dataset
(max(amp.etrs89@coords[,1])-min_x)/50000  # calculate the aproximate number of columns for our raster at a
50 km resolution and the range of date along the x axis in our dataset
max_x = min_x + (50000*21)  # calculate the maximum x coordinate for our raster taking into account the
obtained number of columns
# And now the same for the y axis
min_y <- min(amp.etrs89@coords[,2])
(max(amp.etrs89@coords[,2])-min_y)/50000
max_y = min_y + (50000*18)

# Next step: create the raster (named here "cells") with that dimensions
cells = raster(nrows = 18,ncols = 21,xmn = min_x, xmx=max_x, ymn= min_y, ymx = max_y,crs = etrs89)
cells = setValues(cells, NA) # set values (NAs) to the cells of our raster

# Next step: calculate number of records for each cell
values.records <- getValues(cells)  # get a vector with raster  values
cellID <- cellFromXY(cells, amp.etrs89) # for each point in the table (coordinate pair), we identify the
corresponding cell (index) in the raster.
amp2 <- data.frame(amp.etrs89, cellID) # convert our table into a R dataframe
amp.table <- table(amp2[,c("cellID","CODE")]) # with the table() function we obtain a pivot table with the
cell codes as rows and the species as columns, obtaining the number of records or points of each species
```

```
in each cell
n.records <- apply(amp.table, 1, sum)  # sum the values of each row, which provides the number of records
in each cell
values.records[as.numeric(names((n.records)))] <- n.records  ## set the number of records to the vector
with NAs of the "cells" raster using the index of each cell.
records <- setValues(cells,values.records) # create a new raster using the "cells" raster as template and
the richness values

# Similarly, we can calculate the species richness in each cell
values.sr <- getValues(cells) # get a vector with raster values
amp.table[which(amp.table[]>0)] <- 1  # assign 1 to cells with species records (we obtain a
presence/absence table)
sr.cells <- apply(amp.table, 1, sum)  # sum the values of each row, which provides the number of species
in each cell
values.sr[as.numeric(names((sr.cells)))] <- sr.cells  # set the species richness values to the vector with
NAs of the "cells" raster using the index of each cell.
sr <- setValues(cells,values.sr) # create a new raster using the "cells" raster as template and the
richness values


# Next step: linear regression between species richness and number of records
model <- lm(values.sr ~ values.records)
residuals <- model$residuals   # get the model residuals

# Finally, we obtain a raster with the residuals
values.res <- getValues(cells)   # get a vector with raster  values
values.res[as.numeric(names((residuals)))] <- residuals  # set the residual values to the vector with NAs
of the "cells" raster using the index of each cell.
res <- setValues(cells,values.res) # create a new raster using the "cells" raster as template and the
residual values

# An alternative and more direct way to obtain the rasters of species richness and number of records could
be use the function raterize(), which allows transfering values associated with 'object' type spatial data
(points, lines, polygons) to raster cells:

records2 <- rasterize(coordinates(amp.etrs89), cells, fun="count") # It counts the number of points in
each grid cell.

sr2 <- rasterize(amp.etrs89, cells, field= as.numeric(factor(amp$SPECIES)), fun=function(x,
...){length(unique(x))}) # we need to specify the field to be transferred, in this case the species name.
Because this field must contain numeric categories, we first have to convert the species names to
categories (factor) and then these categories to numbers (as.numeric). Furthermore, the function to
summarize the data in each grid cell will be the length of unique values (different species).
```