Spatial data manipulation (part 1)

Geovisualization

Joaquin Cavieres

1. Introduction

1.2. A simple overview of spatial data

Spatial data (geospatial data) – data that can be mapped using geographic information systems (GIS) – has become increasingly widespread in the social sciences, environmental problems, ecological processes, etc.

Spatial data comprise contains information about geometries (points, lines, polygons, grids) related to a location on a map. This relation is usually established through coordinate data, which carry information on longitude-latitude or easting and northing.

The spatial coordinates are projected onto the earth's surface, and are used to represent a geometry. For example, a spatial location can be represented by a point using the pair of coordinates in the space. On the other hand, roads or rivers can be represented by a linestring, i.e., a connected sequence of such points. Areas (such as the layouts of buildings or the boundaries of neighborhoods), municipalities, counties, or countries, can be represented by polygonal shapes. Finally, grid cells (rasters) can be used to provide spatial summaries of variables such as population density or the proportion of ethnic minority residents in small artificial square areas.

The main characteristics of spatial data is that they are "georeferenced", that means, we have direct spatial identifiers for some value measured. However, there is a big problem when we are referencing this data since the earth is three-dimensional and the maps are only two-dimensional. For the above projection of points comes with the price of distorting geometries upon display.

The Mercator projector is one of the most used system to make projections (used for example, for navigation purposes). Nevertheless, there are different projection systems or Coordinate Reference Systems (CRS) which can be used for various purposes.

1.3. Geographical system information (GIS)

The geographic information system (GIS) is a system that creates, manages, analyzes, and maps all types of data. GIS connects data to a map, integrating location data (where things are) with all types of descriptive information (what things are like there). This provides a foundation for mapping and analysis that is used in science and almost every industry. GIS helps users understand patterns, relationships, and geographic context. The benefits include improved communication and

efficiency as well as better management and decision making (definition from the GIS page: https://www.esri.com/en-us/what-is-gis/overview) .

To work in a GIS environment, the real observations (objects or events that can be recorded in 2D or 3D space) need to be reduced to spatial entities. These spatial entities can be represented in a GIS as a vector data model or a raster data model.

To make spatial data analysis, the "sp" package play a central role in the software R, since by this we can define a set of *classes* to represent spatial data. In this case, a class defines a particular data type, for example, a `data.frame` is a class.

The main reason for defining classes is to create a standard representation of a particular data type to make it easier to write functions (also known as "methods") for them.

Observation: The package "sp" will be replaced in the future by the package "sf", however it is still more commonly used.

2. Vector data

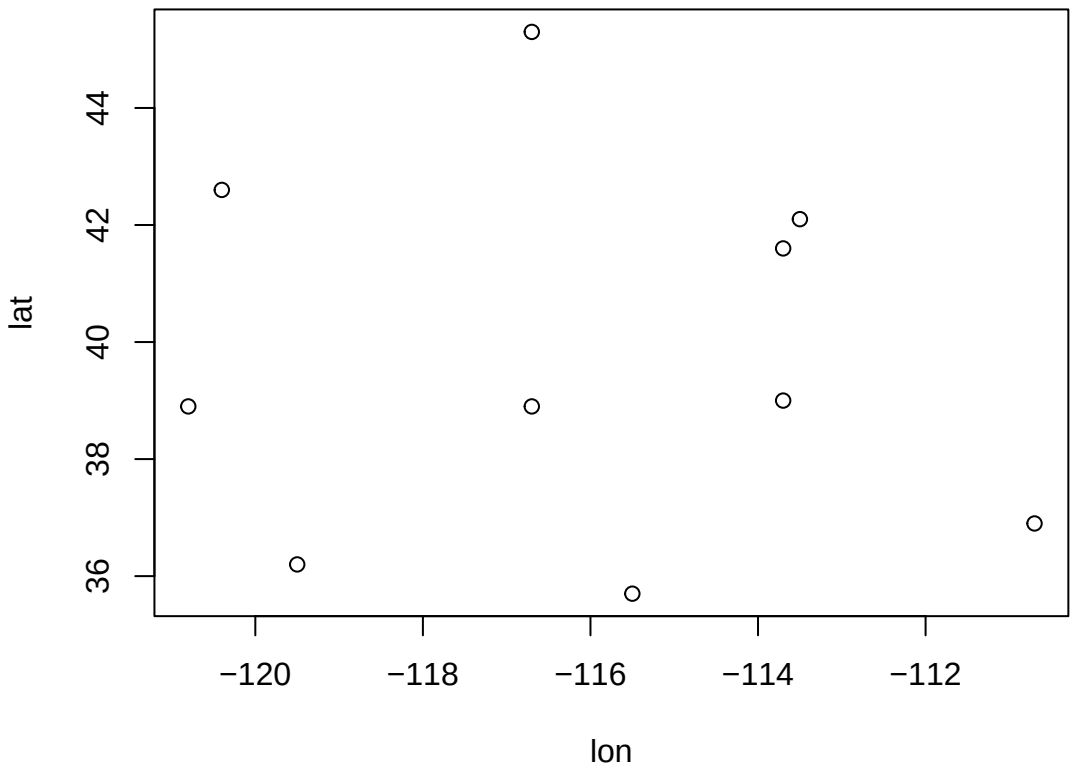
With the "sp" package we can use a set of classes for vector data, for example:

- `SpatialPoints`
- `SpatialLines`
- `SpatialPolygons`

These classes represent the Geometry of the object, but if we want to store attributes we can use the function `DataFrame`, for example writing `SpatialPolygonsDataFrame` or `SpatialPointsDataFrame`

2.1. SpatialPoints

```
lon <- c(-116.7, -120.4, -116.7, -113.5, -115.5, -120.8, -119.5, -113.7,
               -113.7, -110.7)
lat <- c(45.3, 42.6, 38.9, 42.1, 35.7, 38.9, 36.2, 39, 41.6, 36.9)
lonlat <- cbind(lon, lat)
plot(lonlat)
```



Now we will create a `SpatialPoints` object using the package "sp".

```
library(sp)
pts <- SpatialPoints(lonlat)
```

We can check what is the class of the object `pts`,

```
class (pts)
```

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

and what are it's components,

```
showDefault(pts)
```

```
## An object of class "SpatialPoints"
## Slot "coords":
##          lon  lat
##  [1,] -116.7 45.3
##  [2,] -120.4 42.6
##  [3,] -116.7 38.9
##  [4,] -113.5 42.1
##  [5,] -115.5 35.7
##  [6,] -120.8 38.9
##  [7,] -119.5 36.2
##  [8,] -113.7 39.0
##  [9,] -113.7 41.6
## [10,] -110.7 36.9
##
## Slot "bbox":
##        min    max
## lon -120.8 -110.7
## lat   35.7   45.3
##
## Slot "proj4string":
## Coordinate Reference System:
## Deprecated Proj.4 representation: NA
```

The object `pts` has the coordinates and also a `bbox`. This is a "bounding box", or the "spatial limits" that was computed from the coordinates. There is also a "`proj4string`". This stores the coordinate reference system ("crs"). We did not provide the crs so it is unknown (`NA`). That is not good, so let's recreate the object, and now provide a crs.

```
crdref <- CRS('+proj=longlat +datum=WGS84')
pts <- SpatialPoints(lonlat, proj4string=crdref)
```

By the function `SpatialPointsDataFrame` we can create an object using the previous `SpatialPoints` created before. To do this, we need a `data.frame` with the same number of rows as there are in the geometries (the `SpatialPoints` object).

```
# Generate a simulated data for "precipitations", with same quantity as points
precipvalue <- runif(nrow(lonlat), min=0, max=100)
df <- data.frame(ID=1:nrow(lonlat), precip=precipvalue)
```

and match with the `SpatialPoints`:

```
ptsdf <- SpatialPointsDataFrame(pts, data=df)
ptsdf
```

```
##       coordinates ID   precip
## 1  (-116.7, 45.3)  1 81.81375
## 2  (-120.4, 42.6)  2 11.91877
## 3  (-116.7, 38.9)  3 83.59056
## 4  (-113.5, 42.1)  4 35.47170
## 5  (-115.5, 35.7)  5 83.31657
## 6  (-120.8, 38.9)  6 21.33531
## 7  (-119.5, 36.2)  7 44.88974
## 8    (-113.7, 39)  8 82.91508
## 9  (-113.7, 41.6)  9 82.49980
## 10 (-110.7, 36.9) 10 70.51894
```

Now we can see what are the components of this new data (`ptsdf`):

```
showDefault(ptsdf)
```

```
## An object of class "SpatialPointsDataFrame"
## Slot "data":
##    ID   precip
## 1   1 81.81375
## 2   2 11.91877
## 3   3 83.59056
## 4   4 35.47170
## 5   5 83.31657
## 6   6 21.33531
## 7   7 44.88974
## 8   8 82.91508
## 9   9 82.49980
## 10 10 70.51894
##
## Slot "coords.nrs":
## numeric(0)
##
## Slot "coords":
##          lon  lat
##  [1,] -116.7 45.3
##  [2,] -120.4 42.6
##  [3,] -116.7 38.9
##  [4,] -113.5 42.1
##  [5,] -115.5 35.7
##  [6,] -120.8 38.9
##  [7,] -119.5 36.2
```

```
##  [8,] -113.7 39.0
##  [9,] -113.7 41.6
## [10,] -110.7 36.9
##
## Slot "bbox":
##        min    max
## lon -120.8 -110.7
## lat   35.7   45.3
##
## Slot "proj4string":
## Coordinate Reference System:
## Deprecated Proj.4 representation: +proj=longlat +datum=WGS84 +no_defs
## WKT2 2019 representation:
## GEOGCRS["unknown",
##     DATUM["World Geodetic System 1984",
##         ELLIPSOID["WGS 84",6378137,298.257223563,
##             LENGTHUNIT["metre",1]],
##         ID["EPSG",6326]],
##     PRIMEM["Greenwich",0,
##         ANGLEUNIT["degree",0.0174532925199433],
##         ID["EPSG",8901]],
##     CS[ellipsoidal,2],
##         AXIS["longitude",east,
##             ORDER[1],
##             ANGLEUNIT["degree",0.0174532925199433,
##                 ID["EPSG",9122]]],
##         AXIS["latitude",north,
##             ORDER[2],
##             ANGLEUNIT["degree",0.0174532925199433,
##                 ID["EPSG",9122]]]]
```

2.2. SpatialLines and SpatialPolygons

Build a `SpatialPoints` is relatively easy, however, build a `SpatialLines` and `SpatialPolygons` object is a little bit complicated, but still relatively simple using the `spLines()` and `spPolygons(9)` functions (from the "raster" package). For example, for SpatialLines objects:

```
library(raster)
lon <- c(-116.8, -114.2, -112.9, -111.9, -114.2, -115.4, -117.7)
lat <- c(41.3, 42.9, 42.4, 39.8, 37.6, 38.3, 37.6)
lonlat <- cbind(lon, lat)
lns <- spLines(lonlat, crs=crdref)
lns
```

```
## class       : SpatialLines
```

```
## features    : 1
## extent      : -117.7, -111.9, 37.6, 42.9  (xmin, xmax, ymin, ymax)
## crs         : +proj=longlat +datum=WGS84 +no_defs
```

or for SpatialPolygons objects:

```
pols <- spPolygons(lonlat, crs=crdref)
pols
```

```
## class       : SpatialPolygons
## features    : 1
## extent      : -117.7, -111.9, 37.6, 42.9  (xmin, xmax, ymin, ymax)
## crs         : +proj=longlat +datum=WGS84 +no_defs
```

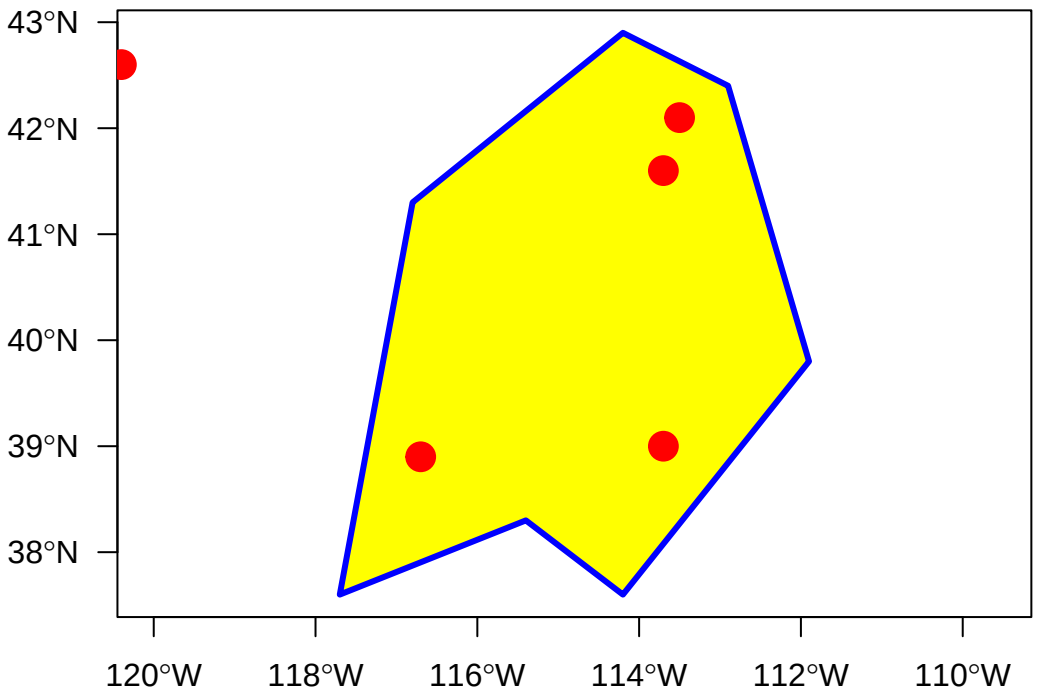
As you can see, the structure of the SpatialPolygons class is complex to understand, since it needs to be accommodate to the possibility of storage multiple polygons, and each each consisting of multiple sub-polygons.

```
str(pols)
```

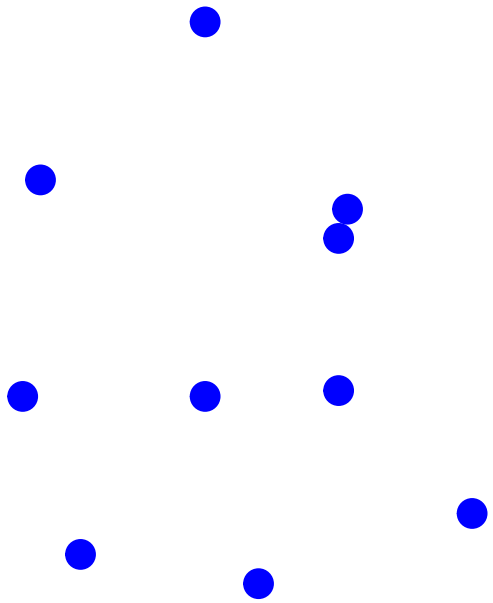
```
## Formal class 'SpatialPolygons' [package "sp"] with 4 slots
##   ..@ polygons   :List of 1
##   .. ..$ :Formal class 'Polygons' [package "sp"] with 5 slots
##   .. .. .. ..@ Polygons :List of 1
##   .. .. .. .. ..$ :Formal class 'Polygon' [package "sp"] with 5 slots
##   .. .. .. .. .. .. ..@ labpt  : num [1:2] -114.7 40.1
##   .. .. .. .. .. .. ..@ area   : num 19.7
##   .. .. .. .. .. .. ..@ hole   : logi FALSE
##   .. .. .. .. .. .. ..@ ringDir: int 1
##   .. .. .. .. .. .. ..@ coords : num [1:8, 1:2] -117 -114 -113 -112 -114 ...
##   .. .. .. ..@ plotOrder: int 1
##   .. .. .. ..@ labpt    : num [1:2] -114.7 40.1
##   .. .. .. ..@ ID       : chr "1"
##   .. .. .. ..@ area     : num 19.7
##   ..@ plotOrder  : int 1
##   ..@ bbox       : num [1:2, 1:2] -117.7 37.6 -111.9 42.9
##   .. ..- attr(*, "dimnames")=List of 2
##   .. .. ..$ : chr [1:2] "x" "y"
##   .. .. ..$ : chr [1:2] "min" "max"
##   ..@ proj4string:Formal class 'CRS' [package "sp"] with 1 slot
##   .. .. ..@ projargs: chr "+proj=longlat +datum=WGS84 +no_defs"
##   .. .. ..$ comment: chr "GEOGCRS[\"unknown\",\n    DATUM[\"World Geodetic System 1984\",\n
##   ..$ comment: chr "FALSE"
```

In this case you don't need to understand how the structure of the SpatialPolygons is organized. The main take home message is that they store geometries (coordinates), the name of the coordinate reference system, and attributes. Finally, we can make a plot using the `plot()` function:

```
plot(pols, axes=TRUE, las=1)
plot(pols, border='blue', col='yellow', lwd=3, add=TRUE)
points(pts, col='red', pch=20, cex=3)
```




```
plot(pts, col='blue', pch=20, cex=3)
```



3. Raster data

The package "sp" also support raster data by the `SpatialGridDataFrame` and `SpatialPixelsDataFrame` classes. However, we will use the package "raster" to work with this type of data.

The "raster" package allows us to build the `RasterLayer`, `RasterBrick`, and `RasterStack` classes, in they are the most important classes in this case. If we can operate in any of these three objects, they are commonly referred as "raster" objects.

The "raster" package has functions for creating, reading, manipulating, and writing raster data. The package provides, among other things, general raster data manipulation functions that can easily be used to develop more specific functions.

3.1. RasterLayer

To represent a single layer (variable) raster data, we should create a `RasterLayer` object. This object stores a number of fundamental parameters that describe the single layer raster data. The fundamental parameters are:

- Number of columns and rows
- The Spatial extent
- The Coordinate Reference System

we will create a `RasterLayer` object, but remember, this type of objects are created from real data.

```
library(raster)
ras <- raster(ncol=10, nrow=10, xmx=-80, xmn=-150, ymn=20, ymx=60)
ras
```

```
## class      : RasterLayer
## dimensions : 10, 10, 100  (nrow, ncol, ncell)
## resolution : 7, 4  (x, y)
## extent     : -150, -80, 20, 60  (xmin, xmax, ymin, ymax)
## crs        : +proj=longlat +datum=WGS84 +no_defs
```

The object `ras` does not has values associated with it, only has the location, resolution, etc. If we want to assign some values, the length must be equal to the number of cells of the `RasterLayer` object.

```
values(ras) <- runif(ncell(ras))
ras
```

```
## class      : RasterLayer
## dimensions : 10, 10, 100  (nrow, ncol, ncell)
## resolution : 7, 4  (x, y)
```

```
## extent     : -150, -80, 20, 60  (xmin, xmax, ymin, ymax)
## crs        : +proj=longlat +datum=WGS84 +no_defs
## source     : memory
## names      : layer
## values     : 0.00775849, 0.9771263  (min, max)
```

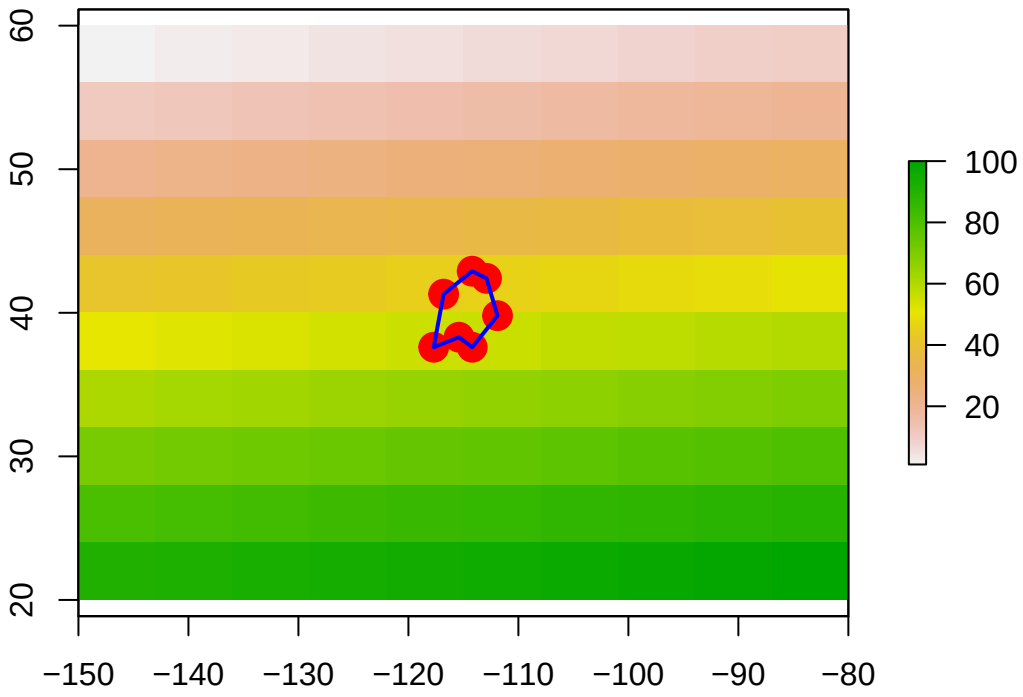
You can also assign cell numbers (in this case overwriting the previous values)

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

```
## class      : RasterLayer
## dimensions : 10, 10, 100  (nrow, ncol, ncell)
## resolution : 7, 4  (x, y)
## extent     : -150, -80, 20, 60  (xmin, xmax, ymin, ymax)
## crs        : +proj=longlat +datum=WGS84 +no_defs
## source     : memory
## names      : layer
## values     : 1, 100  (min, max)
```

Now we can make a simple plot using the `plot()` function:

```
plot(ras)
# add polygon and points
lon <- c(-116.8, -114.2, -112.9, -111.9, -114.2, -115.4, -117.7)
lat <- c(41.3, 42.9, 42.4, 39.8, 37.6, 38.3, 37.6)
lonlat <- cbind(lon, lat)
pols <- spPolygons(lonlat, crs='+proj=longlat +datum=WGS84')
points(lonlat, col='red', pch=20, cex=3)
plot(pols, border='blue', lwd=2, add=TRUE)
```



3.2. RasterStack and RasterBrick

Commonly we also work with multi-variable raster data sets, and for this the "raster" package has teo classes for multi-layer data: `RasterStack` and the `RasterBrick`. The difference between `RasterBrick` and `RasterStack` is that the former can only be linked to a single (multi-layer) file, while the last one can be formed from separate files and/or from a few layers ('bands') from a single file.

In simple words:

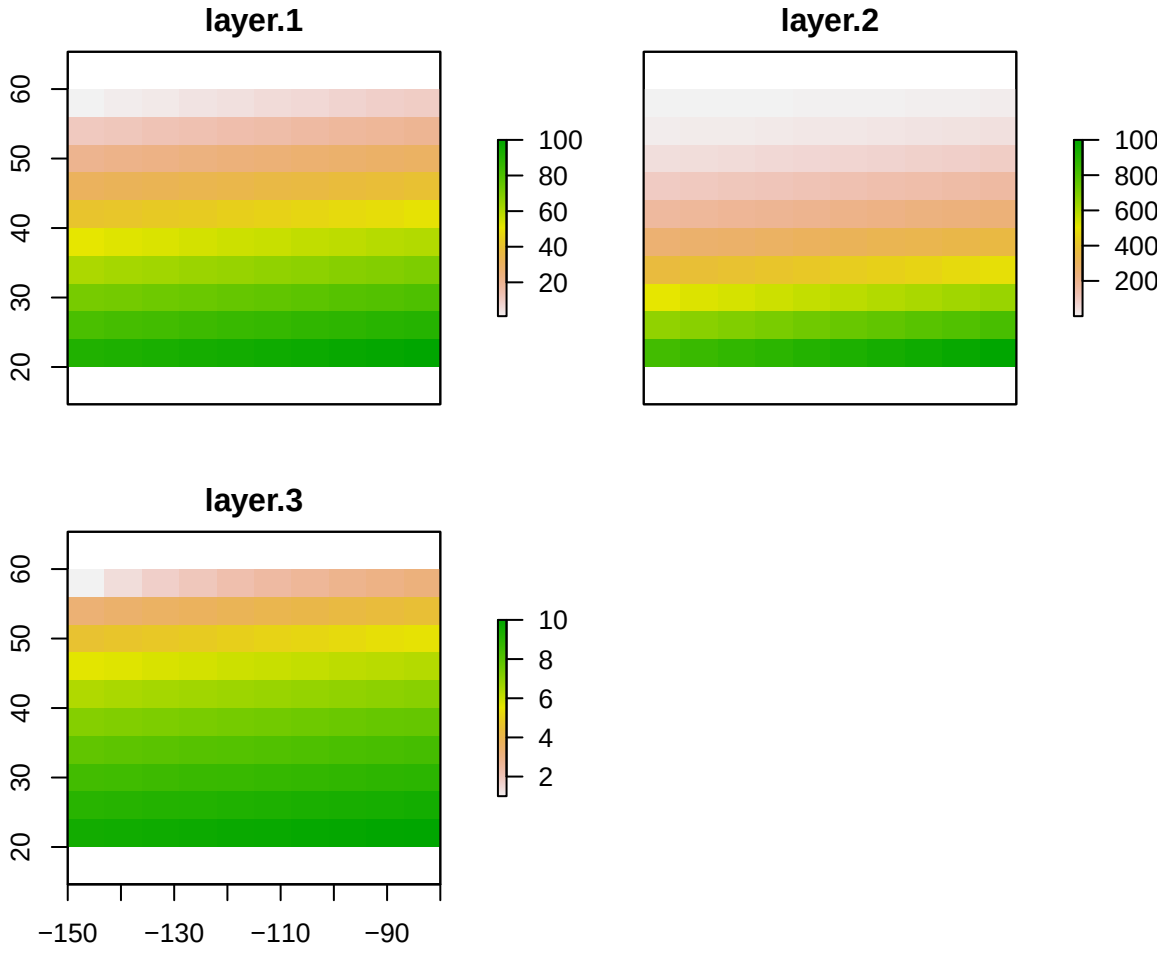
- `RasterStack` is a collection of `RasterLayer` objects with the same spatial extent and resolution.
- A `RasterBrick` is truly a multi-layered object, and processing a `RasterBrick` can be more efficient than processing a `RasterStack` representing the same data. However, it can only refer to a single file.

For example,

```
ras2 <- ras * ras
ras3  <- sqrt(ras)
s <- stack(ras, ras2, ras3)
s
```

```
## class      : RasterStack
## dimensions : 10, 10, 100, 3  (nrow, ncol, ncell, nlayers)
## resolution : 7, 4  (x, y)
## extent     : -150, -80, 20, 60  (xmin, xmax, ymin, ymax)
## crs        : +proj=longlat +datum=WGS84 +no_defs
## names      : layer.1, layer.2, layer.3
## min values :       1,       1,       1
## max values :     100,   10000,      10
```

```
plot(s)
```



and you can make a `RasterBrick` from a `RasterStack`

```
rast_brick <- brick(s)
rast_brick
```

```
## class      : RasterBrick
## dimensions : 10, 10, 100, 3  (nrow, ncol, ncell, nlayers)
## resolution : 7, 4  (x, y)
## extent     : -150, -80, 20, 60  (xmin, xmax, ymin, ymax)
## crs        : +proj=longlat +datum=WGS84 +no_defs
## source     : memory
## names      : layer.1, layer.2, layer.3
## min values :       1,       1,       1
## max values :     100,   10000,      10
```

4. Reading and writing spatial data

4.1. Vector files

The `shapefile` is the most commonly used file format for vector data. For this, we also will use the "raster" package.

a) Reading

using the `system.file` function we can get the full path name of the file's location. We need to do this as the location of this file depends on where the raster package is installed. You don't have the use the `system.file` function for your own files.

```
filename <- system.file("external/lux.shp", package="raster")
filename
```

```
## [1] "C:/Users/joaquin/AppData/Local/Programs/R/R-4.2.2/library/raster/external/lux.shp"
```

Now we have to use the `shapefile` function of the the "raster" package.

```
s <- shapefile(filename)
s
```

```
## class       : SpatialPolygonsDataFrame
## features    : 12
## extent      : 5.74414, 6.528252, 49.44781, 50.18162  (xmin, xmax, ymin, ymax)
## crs         : +proj=longlat +datum=WGS84 +no_defs
## variables   : 5
## names       : ID_1,     NAME_1, ID_2,   NAME_2, AREA
## min values  :    1,   Diekirch,    1, Capellen,   76
## max values  :    3, Luxembourg,   12,    Wiltz,  312
```

Here the `shapefile` returns a `SpatialDataFrame` object, and for this case, a `SpatialPolygonsDataFrame`.

Note: If you are working with the package *rgdal*, then you can use `readOGR` function.

b) Writing

To write a shapefile you also can uses the `shapefile` function, for this, you have to provide a vector type Spatial object as a first argument and a filename as second argument. If you are storing a shapefile with the same name of a file saved before, you can add the argument `overwrite=TRUE`.

```
output_file <- 'test.shp'
shapefile(s, output_file, overwrite=TRUE)
```

Note: If you are working with the package *rgdal*, then you can use `writeOGR` function.

4.2. Raster files

a) Reading

```
f <- system.file("external/rlogo.grd", package="raster")
f
```

```
## [1] "C:/Users/joaquin/AppData/Local/Programs/R/R-4.2.2/library/raster/external/rlogo.grd"
```

and do

```
ras1 <- raster(f)
ras1
```

```
## class      : RasterLayer
## band       : 1  (of  3  bands)
## dimensions : 77, 101, 7777  (nrow, ncol, ncell)
## resolution : 1, 1  (x, y)
## extent     : 0, 101, 0, 77  (xmin, xmax, ymin, ymax)
## crs        : +proj=merc +lon_0=0 +k=1 +x_0=0 +y_0=0 +datum=WGS84 +units=m +no_defs
## source     : rlogo.grd
## names      : red
## values     : 0, 255  (min, max)
```

The object `ras1` is a `RasterLayer` of the first layer in the file (out of three layers). We can request another layer.

```
ras2 <- raster(f, band=2) # here band is the layer
ras2
```

```
## class      : RasterLayer
## band       : 2  (of  3  bands)
## dimensions : 77, 101, 7777  (nrow, ncol, ncell)
## resolution : 1, 1  (x, y)
## extent     : 0, 101, 0, 77  (xmin, xmax, ymin, ymax)
## crs        : +proj=merc +lon_0=0 +k=1 +x_0=0 +y_0=0 +datum=WGS84 +units=m +no_defs
## source     : rlogo.grd
## names      : green
## values     : 0, 255  (min, max)
```

If you want to use all the layers in a single object then you have to use the `brick` function:

```
b <- brick(f)
b
```

```
## class      : RasterBrick
## dimensions : 77, 101, 7777, 3  (nrow, ncol, ncell, nlayers)
## resolution : 1, 1  (x, y)
## extent     : 0, 101, 0, 77  (xmin, xmax, ymin, ymax)
## crs        : +proj=merc +lon_0=0 +k=1 +x_0=0 +y_0=0 +datum=WGS84 +units=m +no_defs
## source     : rlogo.grd
## names      : red, green, blue
## min values :   0,     0,    0
## max values : 255,   255,  255
```

b) Writing

In this case you have to use the `writeRaster` function to write raster data. You must provide a raster object and a filename.

```
x <- writeRaster(b, 'output_raster.tif', overwrite=TRUE)
x
```

```
## class      : RasterBrick
## dimensions : 77, 101, 7777, 3  (nrow, ncol, ncell, nlayers)
## resolution : 1, 1  (x, y)
## extent     : 0, 101, 0, 77  (xmin, xmax, ymin, ymax)
## crs        : +proj=merc +lon_0=0 +k=1 +x_0=0 +y_0=0 +datum=WGS84 +units=m +no_defs
## source     : output_raster.tif
## names      : layer.1, layer.2, layer.3
## min values :       0,       0,       0
## max values :     255,     255,     255
```

References

- **Bivand, R. S., Pebesma, E. J., Gomez-Rubio, V., & Pebesma, E. J. (2008)**. Applied spatial data analysis with R (Vol. 747248717, pp. 237-268). New York: Springer.

- **Spatial Data Science with R and "terra"**. https://rspatial.org/index.html.

- **Stefan Jünger** and **Denis Cohen**. https://www.r-bloggers.com/2021/06/using-geospatial-data-in-r/.