Spatial data manipulation (part 2)

Lecture 7

Joaquin Cavieres

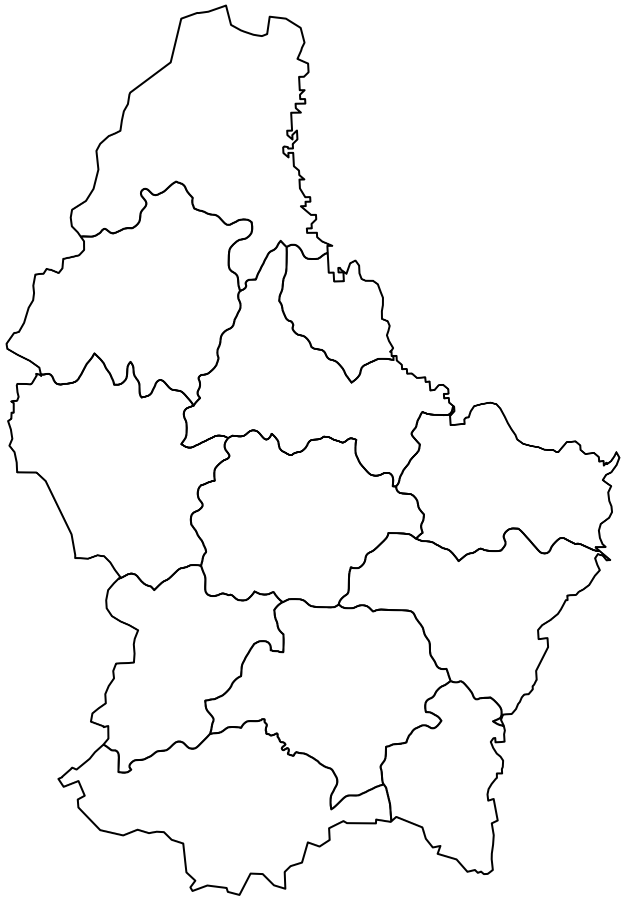
1. Vector data manipulation

First, we will read a "SpatialPolygons"

```
library(raster)
f <- system.file("external/lux.shp", package="raster")
p <- shapefile(f)
p
```

```
## 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
```

```
par(mai=c(0,0,0,0))
plot(p)
```



Basic operations are similar to the operations that you make with a `data.frame`.

1.1. Geomtery and attributes

We can extract the attributes (the variables in a `data.frame` for example) doing:

```
d <- data.frame(p)
head(d)
```

```
##   ID_1       NAME_1 ID_2     NAME_2 AREA
## 1    1     Diekirch    1   Clervaux  312
## 2    1     Diekirch    2   Diekirch  218
## 3    1     Diekirch    3    Redange  259
## 4    1     Diekirch    4    Vianden   76
## 5    1     Diekirch    5      Wiltz  263
## 6    2 Grevenmacher    6 Echternach  188
```

If we want to extract the geometry of the `p` object:

```
g <- geom(p)
head(g)
```

```
##      object part cump hole        x        y
## [1,]      1    1    1    0 6.026519 50.17767
## [2,]      1    1    1    0 6.031361 50.16563
## [3,]      1    1    1    0 6.035646 50.16410
## [4,]      1    1    1    0 6.042747 50.16157
## [5,]      1    1    1    0 6.043894 50.16116
## [6,]      1    1    1    0 6.048243 50.16008
```

1.2. Variables

We can access to the variables writing:

```
p$NAME_2
```

```
##  [1] "Clervaux"         "Diekirch"         "Redange"          "Vianden"
##  [5] "Wiltz"            "Echternach"       "Remich"           "Grevenmacher"
##  [9] "Capellen"         "Esch-sur-Alzette" "Luxembourg"       "Mersch"
```

With the approach below you get a new `SpatialPolygonsDataFrame` with only one variable.

```
p[, 'NAME_2']
```

```
## 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   : 1
## names       :   NAME_2
## min values  : Capellen
## max values  :    Wiltz
```

1.3. Merge

Also we can join a `data.frame` with a spatial object using the `merge()` function as:

```
dat_fr <- data.frame(district=p$NAME_1,
                     canton=p$NAME_2,
                     value=round(runif(length(p), 100, 1000)))
dat_fr <- dat_fr[order(dat_fr$canton), ]
data_tot <- merge(p, dat_fr, by.x=c('NAME_1', 'NAME_2'),
                      by.y=c('district', 'canton'))
data_tot
```

```
## 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   : 6
## names       :     NAME_1,   NAME_2, ID_1, ID_2, AREA, value
## min values  :   Diekirch, Capellen,    1,    1,   76,   235
## max values  : Luxembourg,    Wiltz,    3,   12,  312,   934
```

1.4. Records (measures)

Assuming that every row is a record, then we can do:

```
i <- which(p$NAME_1 == 'Grevenmacher')
g <- p[i,]
g
```

```
## class       : SpatialPolygonsDataFrame
## features    : 3
## extent      : 6.169137, 6.528252, 49.46498, 49.85403  (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  :    2, Grevenmacher,    6, Echternach,  129
## max values  :    2, Grevenmacher,   12,     Remich,  210
```

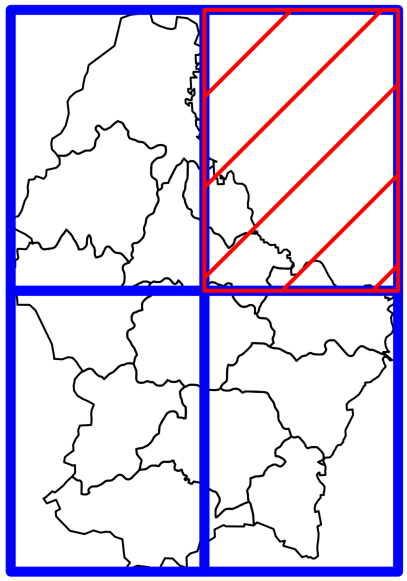
1.5. Append and aggregate

Here we will consider an object `u` that contains 4 polygons, and `u2` which is one of these 4 polygons. So,

```
u <- raster(p, nrow=2, ncol=2, vals=1:4)
names(u) <- 'Zone'
# coerce RasterLayer to SpatialPolygonsDataFrame
u <- as(u, 'SpatialPolygonsDataFrame')
u
```

```
## class       : SpatialPolygonsDataFrame
## features    : 4
## extent      : 5.74414, 6.528252, 49.44781, 50.18162  (xmin, xmax, ymin, ymax)
## crs         : +proj=longlat +datum=WGS84 +no_defs
## variables   : 1
## names       : Zone
## min values  :    1
## max values  :    4
```

```
u2 <- u[2,]
plot(p)
plot(u, add=TRUE, border='blue', lwd=5)
plot(u2, add=TRUE, border='red', lwd=2, density=3, col='red')
```



To append spatial objects of the same (vector) type you can use `bind` function:

```
b <- bind(p, u)
head(b)
```

```
##   ID_1       NAME_1 ID_2     NAME_2 AREA Zone
## 1    1     Diekirch    1   Clervaux  312   NA
## 2    1     Diekirch    2   Diekirch  218   NA
## 3    1     Diekirch    3    Redange  259   NA
## 4    1     Diekirch    4    Vianden   76   NA
## 5    1     Diekirch    5      Wiltz  263   NA
## 6    2 Grevenmacher    6 Echternach  188   NA
```

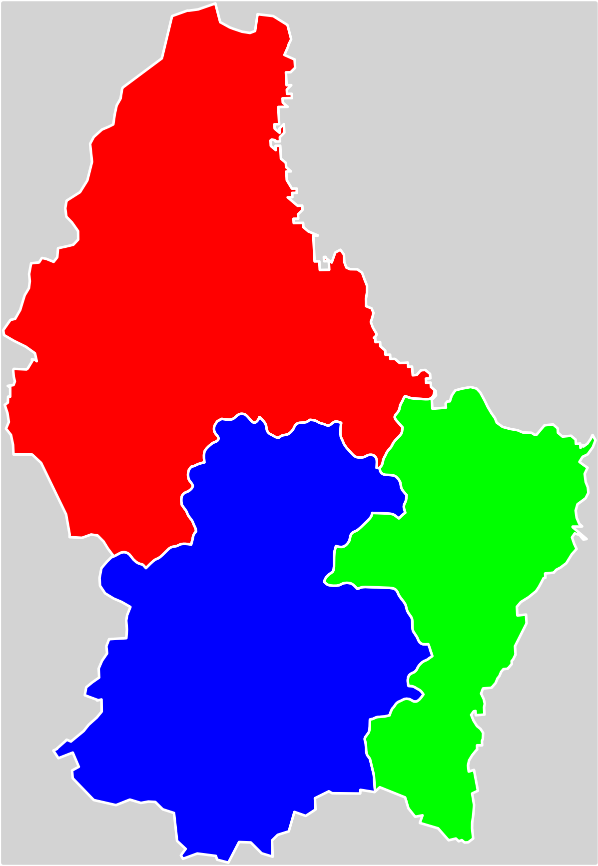
```
tail(b)
```

```
##    ID_1     NAME_1 ID_2     NAME_2 AREA Zone
## 11    3 Luxembourg   10 Luxembourg  237   NA
## 12    3 Luxembourg   11     Mersch  233   NA
```

```
## 13    NA         <NA>    NA         <NA>    NA     1
## 14    NA         <NA>    NA         <NA>    NA     2
## 15    NA         <NA>    NA         <NA>    NA     3
## 16    NA         <NA>    NA         <NA>    NA     4
```

Now we will see an example of the `aggregate` function.

```
pa <- aggregate(p, by='NAME_1')
ua <- aggregate(u)
plot(ua, col='light gray', border='light gray', lwd=5)
plot(pa, add=TRUE, col=rainbow(3), lwd=3, border='white')
```



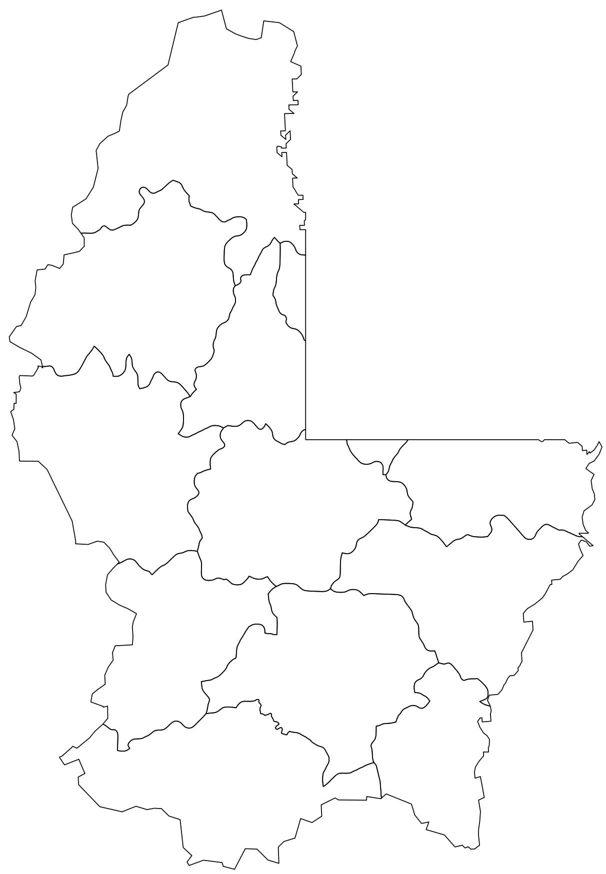
1.6. Overlay

a) Erase a part of a SpatialPolygons object

```
e <- erase(p, u2)
```

which is equivalent to write:

```
e <- p - u2
plot(e)
```



b) Intersect

If we want to intersect SpatialPolygons

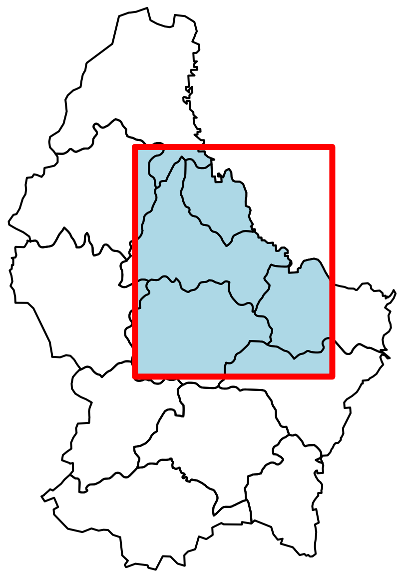
```
inter <- intersect(p, u2)
plot(inter)
```

which is equivalent to write:

```
inter <- p * u2
```

Also it is possible to intersect with an extent (rectangle), for example:

```
e <- extent(6, 6.4, 49.7, 50)
pe <- crop(p, e)
plot(p)
plot(pe, col='light blue', add=TRUE)
plot(e, add=TRUE, lwd=3, col='red')
```



c) Union

we can join two SpatialPolygon objects doing the following

```
t <- union(p, u)
```

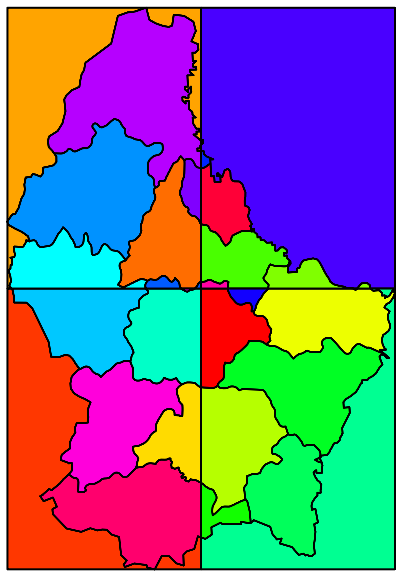
that is the same that do this:

```
t <- p + u
t
```

```
## class       : SpatialPolygonsDataFrame
## features    : 28
## extent      : 5.74414, 6.528252, 49.44781, 50.18162  (xmin, xmax, ymin, ymax)
## crs         : +proj=longlat +datum=WGS84 +no_defs
## variables   : 6
## names       : ID_1,      NAME_1, ID_2,    NAME_2, AREA, Zone
```

```
## min values  :     1,   Diekirch,    1, Capellen,   76,    1
## max values  :     3, Luxembourg,   12,    Wiltz,  312,    4
```

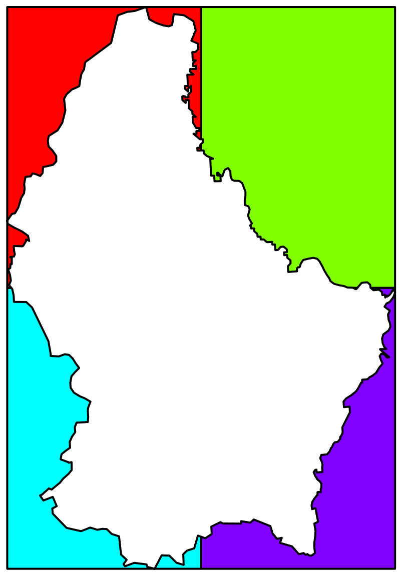
```
plot(t, col=sample(rainbow(length(t))))
```



d) Difference

We can find the symmetrical difference of two SpatialPolygons objects

```
difference <- symdif(u,p)
plot(difference, col=rainbow(length(difference)))
```



```
difference
```

```
## class       : SpatialPolygonsDataFrame
## features    : 4
## extent      : 5.74414, 6.528252, 49.44781, 50.18162  (xmin, xmax, ymin, ymax)
## crs         : +proj=longlat +datum=WGS84 +no_defs
## variables   : 1
## names       : Zone
## min values  :    1
## max values  :    4
```

2. Raster data manipulation

2.1. Creating raster objects

Using the package "raster" and the function `raster()` we can build a `RasterLayer` easily. The default arguments in the function consider a 1 by 1 degree cells to get a raster data structure with a longitude/latitude coordinate reference system.

You can change these settings by providing additional arguments such as `xmn, nrow, ncol, and/or crs`, in the function.

Here an example to create a `RasterLayer` object.

```
library(raster)
x <- raster() # default parameters
x
```

```
## class      : RasterLayer
## dimensions : 180, 360, 64800  (nrow, ncol, ncell)
## resolution : 1, 1  (x, y)
## extent     : -180, 180, -90, 90  (xmin, xmax, ymin, ymax)
## crs        : +proj=longlat +datum=WGS84 +no_defs
```

Set the coordinate reference system (CRS) (i.e., define the projection).

```
projection(x) <- "+proj=utm +zone=48 +datum=WGS84"
x
```

```
## class      : RasterLayer
## dimensions : 180, 360, 64800  (nrow, ncol, ncell)
## resolution : 1, 1  (x, y)
## extent     : -180, 180, -90, 90  (xmin, xmax, ymin, ymax)
## crs        : +proj=utm +zone=48 +datum=WGS84 +units=m +no_defs
```

The object `x` only contains the geometry of the raster,that is, the number of rows and columns, and where the raster is located in geographic space. If we want to add values associated with every cell, we can write:

```
r <- raster(ncol=10, nrow=10)
ncell(r)
```

```
## [1] 100
```

```r
hasValues(r)
```

```
## [1] FALSE
```

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

hasValues(r)
```

```
## [1] TRUE
```

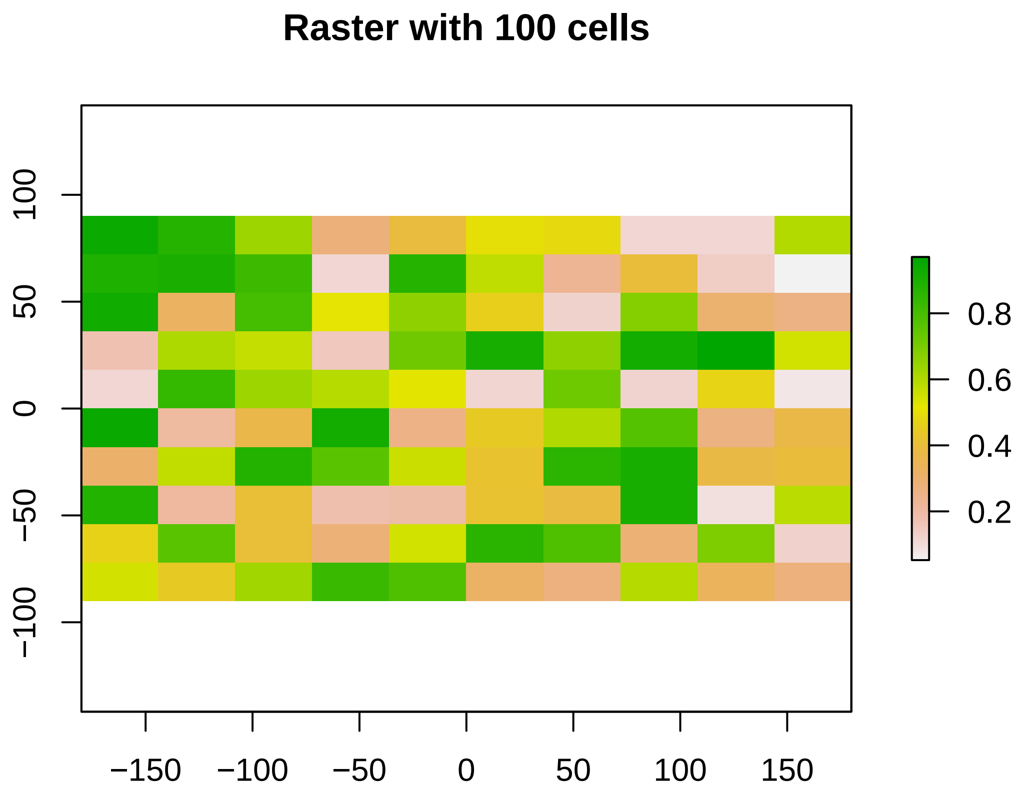
```r
inMemory(r)
```

```
## [1] TRUE
```

```r
values(r)[1:10]
```

```
##  [1] 0.9416227 0.8725538 0.6335872 0.2770864 0.3905521 0.5017617 0.4854508
##  [8] 0.1101254 0.1090514 0.5971786
```

```r
plot(r, main='Raster with 100 cells')
```



In cases where you change the number of columns or rows, you will lose the values associated with the `RasterLayer`. The same happens, in most cases, if you change the resolution directly since this can affect the number of columns or rows. If we change the extent, it does not affect the values because it only adjusts the resolution and neither the number of rows or columns.

```
hasValues(r)
```

```
## [1] TRUE
```

```
dim(r)
```

```
## [1] 10 10  1
```

```
xmax(r)
```

```
## [1] 180
```

Now we will change the maximum `x` coordinate of the extent (the bounding box) of the `RasterLayer`.

```
xmax(r) <- 0
hasValues(r)
```

```
## [1] TRUE
```

```
dim(r)
```

```
## [1] 10 10  1
```

But, if we change the number of columns, then the values disappear:

```
ncol(r) <- 6
hasValues(r)
```

```
## [1] TRUE
```

```
dim(r)
```

```
## [1] 10  6  1
```

```
xmax(r)
```

```
## [1] 0
```

Using the function `raster()` allows us to create a `RasterLayer` from another object, for example, a `RasterLayer`, `RasterStack` or a `RasterBrick`. However, it is common to create a `RasterLayer` object from a file.

The main characteristics of the "raster" package is that the raster datasets are stored in the disk and not in the memory of the computer (RAM). It means, we can work with large files only considering, from these large files, the structure of the data (number of rows, columns, spatial extent and the filename).

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

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

```
r <- raster(filename)
filename(r)
```

```
## [1] "C:\\Users\\joaquin\\AppData\\Local\\Programs\\R\\R-4.2.2\\library\\raster\\external\\te
```

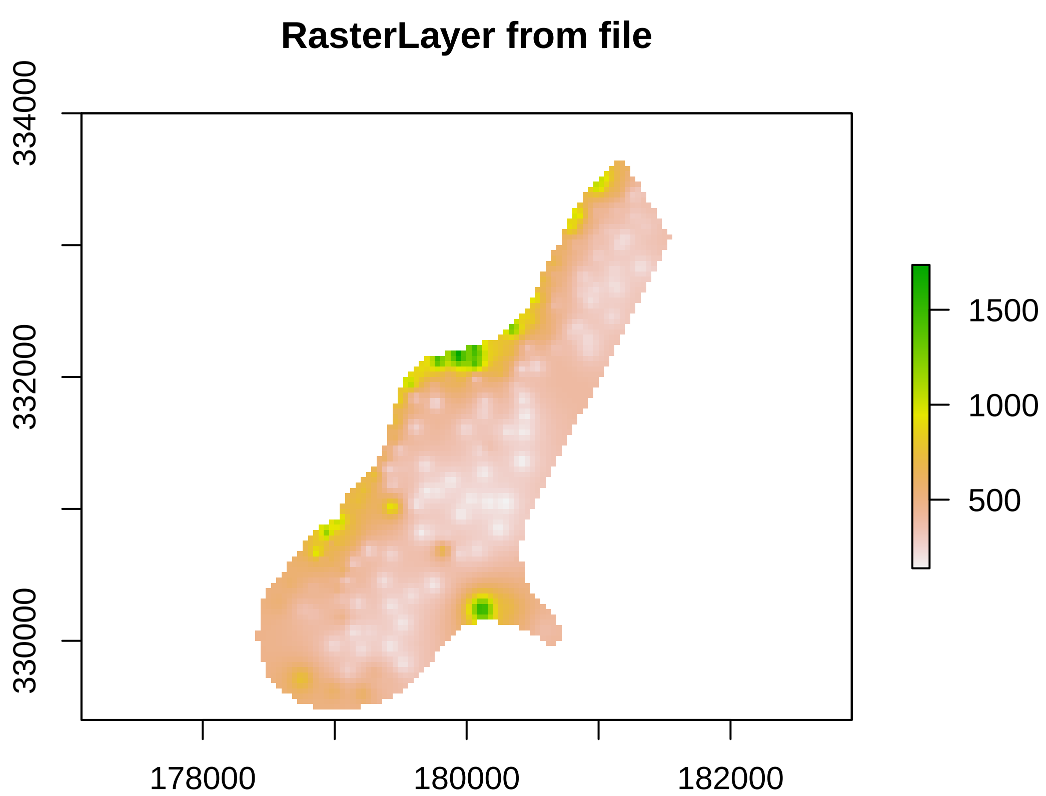
```
hasValues(r)
```

```
## [1] TRUE
```

```
inMemory(r)
```

```
## [1] FALSE
```

```
plot(r, main='RasterLayer from file')
```



We also can build a multi-layer objects from `RasterLayer` objects or from files. For example,

```r
r1 <- r2 <- r3 <- raster(nrow=10, ncol=10)
# Assign random cell values
values(r1) <- runif(ncell(r1))
values(r2) <- runif(ncell(r2))
values(r3) <- runif(ncell(r3))
```

and combine those three `RasterLayer` into a `RasterStack`

```r
s <- stack(r1, r2, r3)
s
```

```
## class      : RasterStack
## dimensions : 10, 10, 100, 3  (nrow, ncol, ncell, nlayers)
## resolution : 36, 18  (x, y)
## extent     : -180, 180, -90, 90  (xmin, xmax, ymin, ymax)
## crs        : +proj=longlat +datum=WGS84 +no_defs
## names      :      layer.1,      layer.2,      layer.3
## min values : 0.0019289961, 0.0009504876, 0.0022828276
## max values :    0.9994818,    0.9776395,    0.9707697
```

```r
nlayers(s)
```

```
## [1] 3
```

or combine those three `RasterLayer` in a `RasterBrick`

```r
b1 <- brick(r1, r2, r3)
```

If you want to create a `RasterBrick` from an external file:

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

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

```r
b <- brick(filename)
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
```

```
nlayers(b)
```

```
## [1] 3
```

and if we want to extract a `RasterLayer` from the previous `RasterBrick` created:

```
r <- raster(b, layer=2)
```

2.2. Raster algebra

Several functions can be applied to a raster object. For example; `+, -, *, /,` logical operators such as `>, >=, <, ==, ! and functions like abs, round, ceiling, floor, trunc, sqrt, log, log10, exp,`
You can chek this doing the following:

```
r <- raster(ncol = 15,  nrow = 15)
values(r) <- 1:ncell(r)
```

and make some operations:

```
s <- r + 10
s <- sqrt(s)
s <- s * r + 5
r[] <- runif(ncell(r))
r <- round(r)
r <- r == 1
```

If you are interested using multiple raster objects (in functions where this is relevant, such as range), these must have the same resolution and origin. Generally these objects have the same extent, but if they don't, the returned object covers the spatial intersection of the objects used.

```
r <- raster(ncol=4, nrow=4) # a new raster file
r[] <- 1

# Obtain values of r
v <- getValues(r)
length(v)
```

```
## [1] 16
```

```r
# Join two raster data
s <- stack(r, r+1)
q <- stack(r, r+2, r+4, r+6)
x <- r + s + q
x
```

```
## class      : RasterBrick
## dimensions : 4, 4, 16, 4  (nrow, ncol, ncell, nlayers)
## resolution : 90, 45  (x, y)
## extent     : -180, 180, -90, 90  (xmin, xmax, ymin, ymax)
## crs        : +proj=longlat +datum=WGS84 +no_defs
## source     : memory
## names      : layer.1, layer.2, layer.3, layer.4
## min values :       3,       6,       7,      10
## max values :       3,       6,       7,      10
```

Applying some functions to a raster data:

```r
a <- mean(r,s, 10) # mean between r and s of 10 values
b <- sum(r,s)
st <- stack(r, s, a, b)
sst <- sum(st)
sst
```

```
## class      : RasterLayer
## dimensions : 4, 4, 16  (nrow, ncol, ncell)
## resolution : 90, 45  (x, y)
## extent     : -180, 180, -90, 90  (xmin, xmax, ymin, ymax)
## crs        : +proj=longlat +datum=WGS84 +no_defs
## source     : memory
## names      : layer
## values     : 11.5, 11.5  (min, max)
```

The function `cellStats()` allow us go get a single number summarizing the cell values of each layer (instead of a `RasterLayer` object)

```r
cellStats(st, 'sum')
```

```
## layer.1.1 layer.1.2 layer.2.1 layer.2.2   layer.3
##        16        16        32        56        64
```

```r
cellStats(sst, 'sum')
```

```
## [1] 184
```

2.3. Summarizing functions

When we are using raster object as first argument, the summary statistics functions such as `min()`, `max()` and `mean()` return a `RasterLayer`. To get a frequency table of a raster file or get the count number of cells, you can use the `freq()` function. You also can apply the function `zonal()` to a raster file to summarize the zones of the object (for example, areas with the same integer number).

```r
r <- raster(ncol= 30, nrow= 20)
r[] <- runif(ncell(r))
cellStats(r, mean)
```

```
## [1] 0.520139
```

Applying the function `zonal()`

```r
s <- r
s[] <- round(runif(ncell(r)) * 5)
zonal(r, s, 'mean')
```

```
##      zone      mean
## [1,]    0 0.5314018
## [2,]    1 0.4685671
## [3,]    2 0.5272841
## [4,]    3 0.5252983
## [5,]    4 0.5304549
## [6,]    5 0.5691401
```

Counting cells

```r
freq(s)
```

```
##      value count
## [1,]     0    62
## [2,]     1   126
## [3,]     2   136
## [4,]     3    96
## [5,]     4   116
## [6,]     5    64
```

```r
freq(s, value=3)
```

```
## [1] 96
```

2.4. Helper functions

```r
r <- raster(ncol=30, nrow=20)
ncol(r)                # num of cols
```

```
## [1] 30
```

```r
nrow(r)                # num of rwos
```

```
## [1] 20
```

```r
ncell(r)               # num of cells
```

```
## [1] 600
```

```r
rowFromCell(r, 100)    # num of rows until the cell 100
```

```
## [1] 4
```

```r
colFromCell(r, 100)    # num of cols until the cell 100
```

```
## [1] 10
```

```r
cellFromRowCol(r, 5, 5) # Cell number from row ans cols
```

```
## [1] 125
```

```r
xyFromCell(r, 100)     # Get coordinates in that cell
```

```
##         x    y
## [1,] -66 58.5
```

2.5. Accessing cell values

You can access to the values in a cell using the function `getValues()` for all the values or in a single row, or using the function `getvaluesBlock()` to get the values of a block (rectangle).

```r
r <- raster(system.file("external/test.grd", package="raster"))
v <- getValues(r, 50)
v[35:39]
```

```
## [1] 743.8288 706.2302 646.0078 686.7291 758.0649
```

```r
getValuesBlock(r, 50, 1, 35, 5)
```

```
## [1] 743.8288 706.2302 646.0078 686.7291 758.0649
```

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.