How to create and edit maps with R

Kévin Cazelles

March 31, 2015

Abstract

This document explains how to create and edit maps using the R language. It provides a basic understanding of the fundamental packages and functions needed to turn R into a Geographic Information System (GIS). Basics manipulations of spatial objects are too presented. However, it is beyond the range of this document to present exhaustively the potentialities R offers in term of spatial manipulation. Rather, I focus on explaining what are the essential classes of spatial objects and the methods that can be applied. I also strive to give as much information as I can to orient readers who wants to go further.

Ce document est distribué sous les termes de la licence CC-BY-NC-SA 4.0, licence de libre diffusion soumise à certaines conditions. En accord avec cette licence, vous pouvez librement utiliser, adapter, modifier et redistribuer le contenu de ce document dans n'importe quelle circonstance, sauf à des fins commerciales (NC), et à la condition non-négociable qu'un crédit suffisant soit attribué à l'auteur de ce présent document en citant leurs noms (BY). Toute version modifiée et redistribuée sera régie par les mêmes termes (SA). Ces droits et conditions seront valides tant que le présent travail sera placé sous les termes de cette licence.

Contents

List of Figures

Chapter 1

Introduction

Since the last decade, new software have emerged making the creation and edition of maps accessible to all. Scientists can now readily draw valuable spatial representations of their work without deep knowledge in mapping[1]. Basic mapping skills should be a part of educational background of most of scientists as maps are often a proper way to expose facts and thereby relevant representations that may ease the scientific debate. Among the tools available (see below) some Geographic Information System (GIS) are free, open-source such as Quantum GIS and GRASS GIS. Also, some high-level programming languages such as Python or R offer dedicated packages.

- ArcGIS
- CartoDB
- Google Map
- Goole Earth
- DIVA-GIS
- GRASS
- Leaflet
- Mapnik
- Mapinfo
- PostGIS
- Quantum GIS
- SAGA GIS
- TileMill

1.1 Why do I use the R programming language for map creation and edition?

Given the number of tools dedicated to spatial data representation and analysis, it is important to figure out the reasons why you should use R. According to me, the choice strongly relies upon (i) your ambition in term of mapping and (ii) your R skills. If you aim at creating a good-looking map without analysis and you are not familiar with R, I would discourage you to use R. But if you are familiar with R, you can quickly get a map as a R plot and so benefit from the plot system you already know. Also, when you need tricky spatial analysis (*e.g* krigging), even if you are not familiar with R, you may benefit from this language and you may not encounter any difficulties to realize any kind of analysis and gather the plot. Ultimately yo will analysis and create your plot with R only.

[1] If you can access to, I recommend the reading of the article: Zastrow, Mark. 2015. "Science on the Map." Nature 519: 119–120.

1.2 What can be found in this document?

This short document provides some basics to create and edit maps using R. One of the strongest advantage of using R as a Geographic Information System (GIS) is to use its facilities to manipulate efficiently spatial object and to undertake various spatial analysis. The spatial analysis are beyond the scope of this document. I will deal with the manipulation of spatial objects: I start by explaining how spatial classes work in R, then I show you how R can import and export spatial objects. The latter makes R a useful tool to create, manipulate and convert files that contain spatial data. Throughout the document I strive to provide advices (as good as possible) to edit simple but elegant maps.

To be able to use all the code below, four packages must be installed: "sp", "raster", "rgdal", "rgeos". Their installation will be explained along the document. To take maximum benefits from the present document you must be working on line. As the reader of this document, you must have basic R skills, meaning the user knows how to use object, how to code basic functions and how to edit plots. If it not the case, you may benefit from a clear introduction to R such as the one available on the CRAN website.

If the reader of the present document is eager to learn more about the potentialities R offers, a good starting point is to read the valuable package guide Roger Bivand - one of the most important contributor to spatial analysis in R - wrote. You can also pursue your learning by having a look of the book the same author wrote with two colleagues.

1.3 Colophon

The present document has been written using R Markdown on MacOSX 10.10.4. I conceived the code to make it working standalone: additional material is downloaded from R commands. Also, instead of a reference list, I provide hyper-links to better explain what we do when we use spatial object in R. The version of R that creates the document is:

As a convention, I use bold fonts for R objects and I add parentheses for objects that are functions (*e.g.*, object **obj** and function **foo()**). I use italics for class and function arguments (for instance, argument *arg*) and I employ quotemarks for package names (*e.g*, package "sp"). Also, *path* denotes the character string that contains the path of the ".Rmd" file that generates this document. If you find some error or have any comments about this document and/or the script, please feel free to send me an email (kevin(dot)cazelles(at)gmail(dot)com).

Many functions are used throughout this document and some are not detailed. Whenever you want to get more details, I recommend you to read the documentation associated. To get access to this documentation, you need to type "?" before the name of the function then press Enter in the R console. The example below shows two commands to access the help associated to the **getwd()** function.

```
?getwd
help("getwd")
```

Chapter 2

Points, Lines, Polygons and Grids

Basically, spatial objects are split into two categories: vectors and rasters. Roughly speaking, vectors are a set of coordinates, whereas rasters are regular grids. The first step to create a map is to manipulate points, lines and grids (polygons can be considered as a sets of closed lines). This is what I present in this section. At first glance, one can describe our planet as a sphere and locate any point on Earth by two values: the longitude (values taken in [-180,180]) and the latitude (values taken in [-90,90])[1]. For the sake of illustration, I define a object **vec** that contains the coordinates of 10 points randomly distributed on Earth .

```
nbp <- 10
vec <- list(x=runif(nbp,-160,160), y=runif(nbp,-80,80))
```

```
ras <- matrix(runif(nbp*nbp),nbp)
```

To keep things as simple as possible, that is all that should be defined! Drawing very basic maps is no more that plotting these objects:

```
# Split the plot region into four subregions, the box is of "L" form.
par(mfrow=c(2,2), bty="l")
## ----
plot(c(-180,+180),c(-90,90), col=0, main="Points", xlab="", ylab="Latitude (°)")
points(vec, pch=19, col=c(1,"grey45"))
text(vec$x, vec$y, 1:10, pos=3, col=c(1,"grey45"))
## ----
plot(c(-180,+180),c(-90,90), col=0, main="Lines", xlab="", ylab="")
lines(vec$x[1:4], vec$y[1:4])
lines(vec$x[5:6], vec$y[5:6], col=2)
lines(vec$x[7:10], vec$y[7:10], col=3)
## ----
plot(c(-180,+180),c(-90,90), col=0, main="Polygon", xlab="Longitude (°)",
    ylab="Latitude (°)")
polygon(vec$x[1:6], vec$y[1:6], col=2, lwd=4, border=4)
## ----
image(seq(-120,120,length.out=11), seq(-60,60,length.out=11), ras, main="Grid",
    xlab="Longitude (°)", ylab="")
```

[1] Obviously, there are many ways to locate points on Earth.

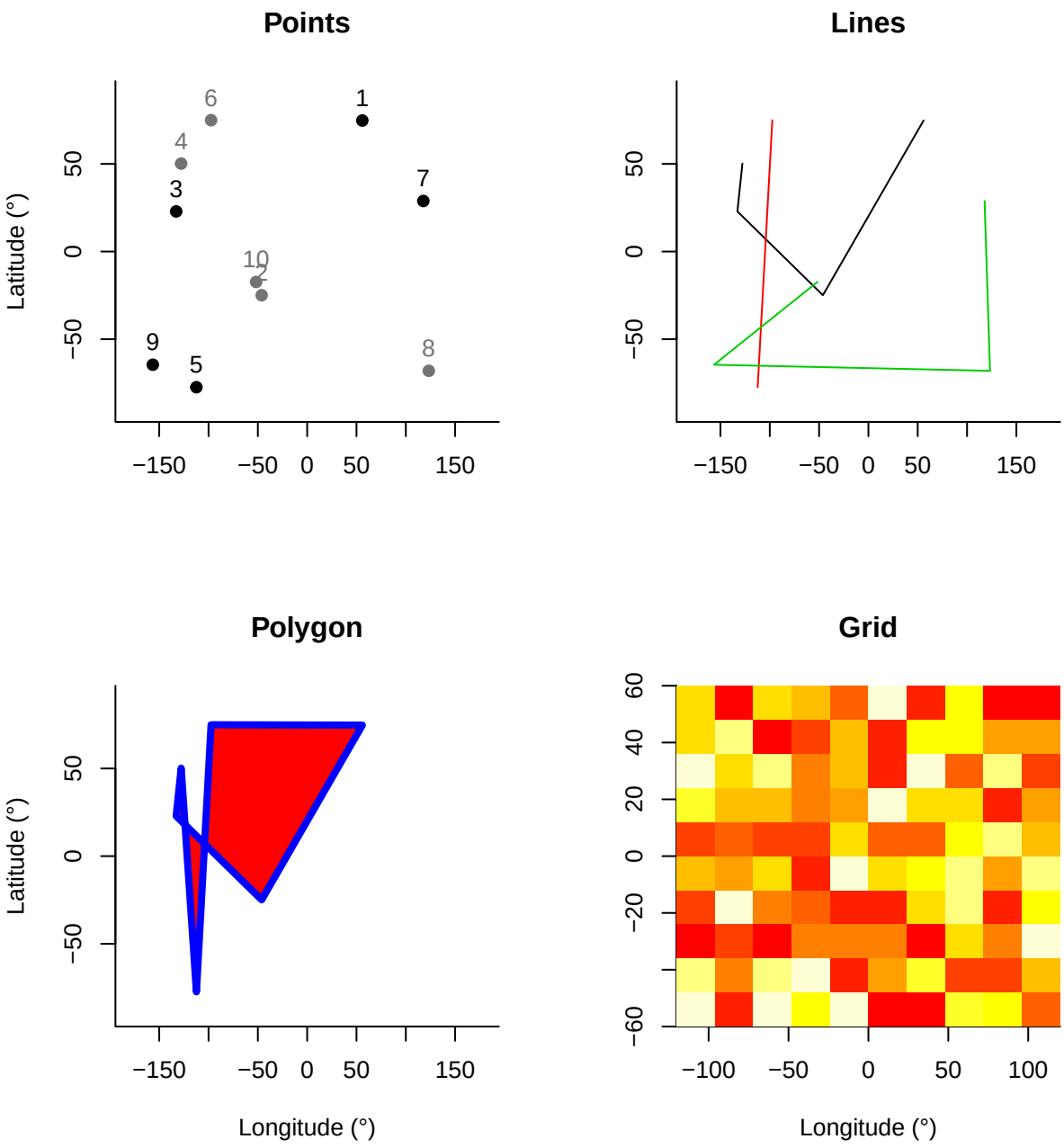

Figure 2.1: Points, lines, polygons and grids. These are the basic object manipulated when drawing maps.

Chapter 3

Classes for spatial objects in R

3.1 The "sp" package

Manipulating efficiently spatial data requires more than graphical skills. Indeed, spatial operations often involve to associate spatial objects with datasets or with a Coordinate Reference System (CRS). To provide a consistent framework to work with spatial objects in R, the "sp" package defines adequate classes and methods. These classes of spatial objects are used by other packages (such as "rgdal", "rgeos" and "raster" that are used below) which makes R a powerful GIS:

```
install.packages("sp")
```

The documentation of this package and any package found on the CRAN website is available as a pdf file. For the "sp" package, you can either look at the reference manual on line or you can enter the following command line in your R console:

```
library(help="sp")
```

Then I load the package[1]:

```
library("sp")
```

The "sp" package is the core package to turn R into a powerful GIS[2] and the other packages that will be used depend on it. The table below present the spatial classes that are used in this document. To create an object of any of those classes, the function to be called is named after the class of object it returns.

Table 3.1: Summary of the spatial classes defined by the "sp" package that will be detail in this document.

Classes / Functions	Contents
Points	list of points (set of coordinates)
SpatialPoints	list of points + CRS
SpatialPointsDataPoints	list of points + CRS + attribute table
Line	a line (set of coordinates)

[1] Loading a package prevents you from using the name space of the package, "sp::" for "sp" package, before each function.
[2] Contributed packages extent a lot R functionalities. The number of contributed package increases exponentially since its first release (June 2000); on the August 12, 2015, 7002 contributed packages were available.

Classes / Functions	Contents
Lines	list of lines
SpatialLines	list of lines + CRS
SpatialLinesDataFrame	list of lines + CRS + attribute table
Polygon	a polygon (set of coordinates)
Polygons	list of polygons
SpatialPolygons	list of polygons + CRS
SpatialPolygonsDataFrame	list of polygons + CRS + attribute table
GridTopology	Grid (smallest coordinates + cell size and number)
SpatialGrids	Grid + CRS
SpatialGridsDataFrame	Grid + CRS + attribute table

3.2 Spatial points

3.2.1 Class *SpatialPoints*

Now, I re-use the coordinates that **vec** countains and I call the **SpatialPoints()** function to get an object of class *SpatialPoints*.

```
ptsp <- SpatialPoints(vec)
print(ptsp)
```

```
## class       : SpatialPoints
## features    : 10
## extent      : -156.5034, 123.2854, -77.40712, 74.94329  (xmin, xmax, ymin, ymax)
## coord. ref. : NA
```

Hence, an object of class *SpatialPoints* is created. You may have noticed that no CRS is defined. At this stage, it is optional. It can be defined with the *proj4string* that is encountered each time a CRS can be associate a CRS to your object. Note that the way it actually works is detailed in section [The "rgdal" package][]. Below, I specify that longitude and latitude will be used together with the geodetic datum WGS84 (the most common).

```
ptsp <- SpatialPoints(vec, proj4string=CRS("+proj=longlat +datum=WGS84 +ellps=WGS84"))
print(ptsp)
```

```
## class       : SpatialPoints
## features    : 10
## extent      : -156.5034, 123.2854, -77.40712, 74.94329  (xmin, xmax, ymin, ymax)
## coord. ref. : +proj=longlat +datum=WGS84 +ellps=WGS84 +towgs84=0,0,0
```

```
plot(ptsp)
```

This plot differs from the one drawn in the previous section! This is because there is a specific **plot()** method for spatial object[3]. This method is very helpful, especially the *add* argument that allows us to add spatial objects on a base map. Now, I get a closer look at the newly-defined spatial object.

[3]To list all the different **plot()** methods, you can enter "methods(plot)" in the R console.

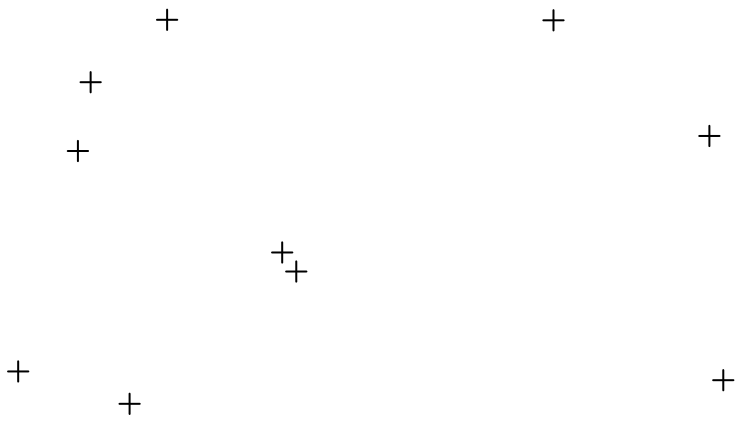

Figure 3.1: Spatial points and the default plot associated.

```
isS4(ptsp)
```

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

```
structure(ptsp)
```

```
## class       : SpatialPoints
## features    : 10
## extent      : -156.5034, 123.2854, -77.40712, 74.94329  (xmin, xmax, ymin, ymax)
## coord. ref. : +proj=longlat +datum=WGS84 +ellps=WGS84 +towgs84=0,0,0
```

It is a S4 object[4], let's look at its attributes:

```
attributes(ptsp)
```

```
## $bbox
##          min       max
## x -156.50336 123.28543
## y  -77.40712  74.94329
##
## $proj4string
## CRS arguments:
##  +proj=longlat +datum=WGS84 +ellps=WGS84 +towgs84=0,0,0
##
## $coords
##                x         y
##  [1,]   55.91372  74.74400
##  [2,]  -46.15640 -24.91694
##  [3,] -132.80656  22.88590
##  [4,] -127.75164  50.18241
##  [5,] -112.28936 -77.40712
##  [6,]  -97.40883  74.94329
##  [7,]  117.77705  28.85688
```

[4]Roughly speaking, objects in R are of two different kinds: S3 and S4. The first one is somehow more flexible while the latter is more formal. For more details, I recommend you to read what Hadley Wickham wrote about it.

```
##  [8,]  123.28543 -68.03787
##  [9,] -156.50336 -64.56185
## [10,]  -51.74587 -17.36584
##
## $class
## [1] "SpatialPoints"
## attr(,"package")
## [1] "sp"
```

One way to access to these attributes is given below:

```
# Let's call ptsp's extent
attributes(ptsp)$bbox
```

```
##          min       max
## x -156.50336 123.28543
## y  -77.40712  74.94329
```

Some attributes are slots that can be called using "@":

```
slotNames(ptsp)
```

```
## [1] "coords"      "bbox"        "proj4string"
```

```
# Let's call ptsp's proj4string
ptsp@proj4string
```

```
## CRS arguments:
##  +proj=longlat +datum=WGS84 +ellps=WGS84 +towgs84=0,0,0
```

This is a powerful way to access any piece of information that spatial objects contain.

3.2.2 Class *SpatialPointsDataFrame*

In spatial analysis, data may be associated with coordinates. For vectors, data are stored in an attribute table. For instance, consider you have geographic coordinates for different points corresponding to different soil sample. To store the soil organic matter and/or the percentage of clay associated to each sample, an attribute table must be created. In R, creating a data is very straightforward thanks to the **data.frame()** that creates *data.frame* object:

```
# Brackets are used to print the table.
(datapt <- data.frame(cbind(Var1=rnorm(nbp), Var2=1+runif(nbp,0,10))))
```

```
##           Var1     Var2
## 1  -2.55937263 8.994566
## 2   0.57306048 8.437298
## 3   1.00790501 3.322411
## 4   0.06321444 5.198494
## 5  -0.17106886 3.374135
## 6   0.09254566 2.728074
```

```
## 7  -0.29537317 4.862037
## 8   0.06234264 3.861219
## 9  -0.67856379 1.396514
## 10 -0.30999519 5.780343
```

Given one *SpatialPoints* object together with one *data.frame*, I create a *SpatialPointsDataframe* object. It contains all attributes you can find in a Shapefile of points.

```
ptspd <- SpatialPointsDataFrame(ptsp, data=datapt)
structure(ptspd)
```

```
## class       : SpatialPointsDataFrame
## features    : 10
## extent      : -156.5034, 123.2854, -77.40712, 74.94329  (xmin, xmax, ymin, ymax)
## coord. ref. : +proj=longlat +datum=WGS84 +ellps=WGS84 +towgs84=0,0,0
## variables   : 2
## names       :              Var1,             Var2
## min values  : -2.55937263002232, 1.39651442319155
## max values  :  1.00790500547928,  8.9945656307973
```

Our **datapt** is now the attribute table of **ptspd**. I can efficiently access to the coordinates and the attributes tables using "@":

```
ptspd@data
```

```
##           Var1     Var2
## 1  -2.55937263 8.994566
## 2   0.57306048 8.437298
## 3   1.00790501 3.322411
## 4   0.06321444 5.198494
## 5  -0.17106886 3.374135
## 6   0.09254566 2.728074
## 7  -0.29537317 4.862037
## 8   0.06234264 3.861219
## 9  -0.67856379 1.396514
## 10 -0.30999519 5.780343
```

```
ptspd@coords
```

```
##                x         y
##  [1,]   55.91372  74.74400
##  [2,]  -46.15640 -24.91694
##  [3,] -132.80656  22.88590
##  [4,] -127.75164  50.18241
##  [5,] -112.28936 -77.40712
##  [6,]  -97.40883  74.94329
##  [7,]  117.77705  28.85688
##  [8,]  123.28543 -68.03787
##  [9,] -156.50336 -64.56185
## [10,]  -51.74587 -17.36584
```

I plot **ptspd** and I specify that the size of points depends on the second variable of the attribute table:

```
plot(ptspd, pch=1, cex=ptspd@data$Var2)
```

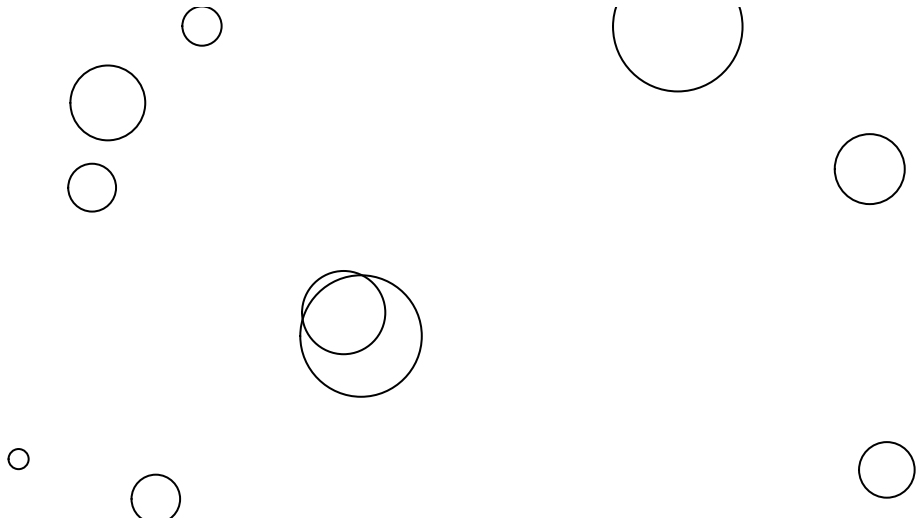

Figure 3.2: A simple but customized plot of the *SpatialPointsDataframe* object **ptspd**.

Among the advantage of these spatial classes, there is the easiness of the definition of proper subsets. For instance, you can create a new *SpatialPointsDataFrame* with points 1,2,4,5,7,8 of the previous as follows:

```
ptspd2 <-ptspd[c(1,2,4,5,7,8),]
# or ptspd2 <-ptspd[-c(3,6,9,10),]
```

Consistently, I get:

```
summary(ptspd2)
```

```
## Object of class SpatialPointsDataFrame
## Coordinates:
##          min      max
## x -127.75164 123.2854
## y  -77.40712  74.7440
## Is projected: FALSE
## proj4string :
## [+proj=longlat +datum=WGS84 +ellps=WGS84 +towgs84=0,0,0]
## Number of points: 6
## Data attributes:
##       Var1               Var2
##  Min.   :-2.55937   Min.   :3.374
##  1st Qu.:-0.26430   1st Qu.:4.111
##  Median :-0.05436   Median :5.030
##  Mean   :-0.38787   Mean   :5.788
##  3rd Qu.: 0.06300   3rd Qu.:7.628
##  Max.   : 0.57306   Max.   :8.995
```

```
ptspd2@data
```

```
##          Var1     Var2
## 1 -2.55937263 8.994566
## 2  0.57306048 8.437298
## 4  0.06321444 5.198494
## 5 -0.17106886 3.374135
## 7 -0.29537317 4.862037
## 8  0.06234264 3.861219
```

The main benefit of using such classes is to manipulate efficiently spatial objects. There are a lot of functions already implemented and it is straightforward to automate basic operations. For instance, you can get all the distance between your points as follows:

```
spDists(ptspd2)
```

```
##           [,1]      [,2]      [,3]      [,4]      [,5]      [,6]
## [1,]     0.000 12995.493  6141.253 19540.769  6114.860 16513.511
## [2,] 12995.493     0.000 11517.166  6739.598 18368.919  9658.024
## [3,]  6141.253 11517.166     0.000 14175.266  9141.671 15770.972
## [4,] 19540.769  6739.598 14175.266     0.000 14024.033  3438.004
## [5,]  6114.860 18368.919  9141.671 14024.033     0.000 10737.235
## [6,] 16513.511  9658.024 15770.972  3438.004 10737.235     0.000
```

3.3 Spatial Lines and Polygons

Creating either spatial points, either spatial lines or spatial polygons are essentially the same approach. In the last section, I show how to create spatial points, so, for lines and polygon, instead of points, fundamental units are lists of lines or polygons. Therefore, the first step is to define these new units thanks to two functions: **Line()** and **Polygon()**. As for points, I re-use objects defined in the first section. Our lines are split into two groups of lines. First, I create empty lists:

```
# Two lists for the lines.
li1 <- list()
li2 <- list()
# One for the polygons (it could have been more!).
pol1 <- list()
```

Second, I fill the empty list using **Line()** and **Polygon()** to create the fundamental units.

```
li1[[1]]<- Line(cbind(vec$x[1:4], vec$y[1:4]))
li1[[2]]<- Line(cbind(vec$x[5:6], vec$y[5:6]))
##
li2[[1]]<- Line(cbind(vec$x[7:10], vec$y[7:10]))
####
pol1[[1]] <- Polygon(cbind(c(vec$x[1:6],vec$x[1]), c(vec$y[1:6],vec$y[1])))
```

The next step transforms the previous lists into *Lines* and *Polygons* object that must be identifies with an unique ID for the next building steps.

```
lin1 <- Lines(li1,ID=1)
lin2 <- Lines(li2,ID=2)
##
poly1 <-  Polygons(pol1, ID=1)
```

Then, I define object of class *SpatialLines* and *SpatialPolygons*. They are respectively made of lists of lists of lines and polygons together with a CRS.

```
linsp <- SpatialLines(list(lin1,lin2),
    proj4string=CRS("+proj=longlat +datum=WGS84 +ellps=WGS84"))
#
polsp <- SpatialPolygons(list(poly1),
    proj4string=CRS("+proj=longlat +datum=WGS84 +ellps=WGS84"))
```

Finally, I add an attribute table to obtain objects of class *SpatialLinesDataFrame* and *SpatialPolygons-DataFrame*.

```
datalin<-data.frame(var1=runif(2), var2=rnorm(2), var3=rpois(2,10))
linspd <- SpatialLinesDataFrame(linsp, data=as.data.frame(datalin))
#
datapol<-data.frame(var1=runif(1), var2=rnorm(1), var3=rpois(1,10))
polspd <- SpatialPolygonsDataFrame(polsp, data=datapol)
```

I suggest you look at these objects using **attributes()** or **structures()** functions. To conclude the section, let's make a quick plot:

```
par(mfrow=c(1,2), ann=FALSE, bty="l")
plot(linspd, col=c(2,4))
plot(polspd, col=4)
```

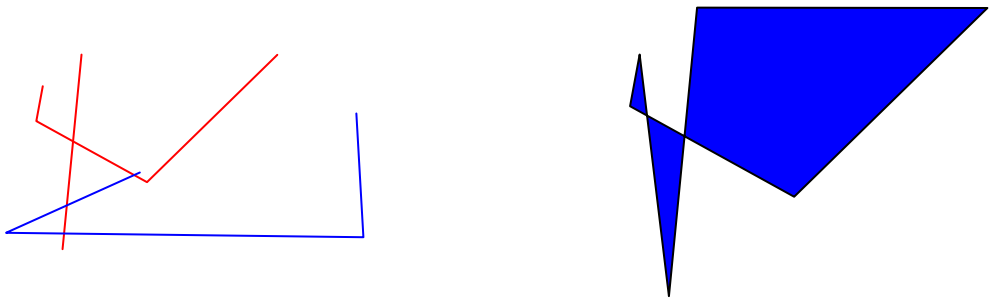

Figure 3.3: *SpatialLines* (on the left) and *SpatialPolygons* (on the right).

Chapter 4

Spatial grids

The last classes of spatial defined in the "sp" package to be presented are grids. Here, I do not present how to use a *SpatialPixel*. Rather, I shortly focus on *SpatialGrid* objects. Note that the next chapter is dedicated to raster objects which are more used (as far as I know). First, you need to define a grid topology using the **GridTopology()** function. You must specify the following arguments:

1. *cellcentre.offset*: coordinates of the centre of the bottom-left
2. *cellsize*: a vector with the cell size in each dimension.
3. *cells.dim*: a vector with the number of cells in each dimension.

```
grd <- GridTopology(cellcentre.offset=c(-179.5,-89.5), cellsize=c(1,1), cells.dim=c(360,180))
```

Then, I create a *SpatialGrid* by adding a CRS:

```
grdsp <- SpatialGrid(grd, proj4string=CRS("+proj=longlat +datum=WGS84 +ellps=WGS84"))
```

Finally, an attribute table is added to **grdsp** to obtain a *SpatialGridDataFrame* object called **grdspd**.

```
var1 <- runif(360*180, 0, 10)
var2 <- rnorm(360*180, 0, 10)
datagd <- data.frame(var1, var2)
#
grdspd <- SpatialGridDataFrame(grdsp, data=datagd)
par(mar=c(4,4,1,1))
```

I plot the new object and I add **ptspd** on it.

```
image(grdspd)
axis(1, seq(-180, 180, by=20))
axis(2, seq(-90, 90, by=30))
title(xlab="Longitude", ylab="latitude")
plot(ptspd, add=TRUE)
```

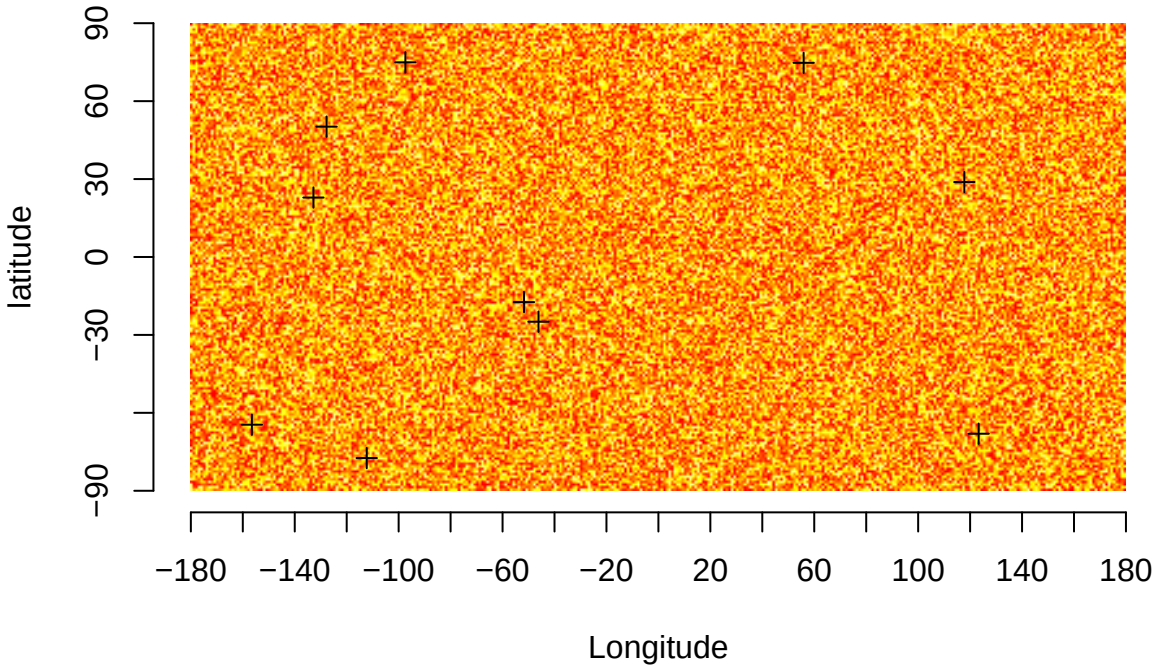

Figure 4.1: SpatialGrid plotted with the *image* function together with

Chapter 5

Rasters

5.1 Package "raster"

The "raster" package defines classes, methods and functions for raster files. To the best of my knowledge, I feel that people more often work with this package rather than with the classes for grids defined in the "sp" package (but "raster" still imports "sp"). I personally use "sp" package to manipulate vector files and "raster" for raster formats. First, let's install and the package.

```
install.packages("raster")
library(raster)
```

In the following sections below, I introduce the different classes defined in "raster"; that is *RasterLayer*, *RasterStack* and *RasterBrick*.

5.2 RasterLayer

5.2.1 Function raster

To create a first "raster" object, I call **raster** function. As mentioned in the documentation of the function, they are many way to build a *RasterLayer*, notably:

1- a path to a raster file 2- a *SpatialGrid*, defined as above 3- a matrix 4- an object of class *RasterLayer*, *RasterStack* or *RasterBrick*

The first example I choose is creating a *RasterLayer* from a matrix (others are similar, see the documentation). To do so, I specify the minimum and maximum centre coordinates in each dimension. The number of cells in each dimension is provided by the number of rows and columns of the matrix passed to the function.

```
Ra1 <- raster(matrix(runif(360*180,0,10),ncol=360,nrow=180),
    crs=CRS("+proj=longlat +datum=WGS84 +no_defs +ellps=WGS84 +towgs84=0,0,0"),
    xmn=-179.5, xmx=+179.5, ymn=-79.5, ymx=+79.5)
Ra1
```

```
## class       : RasterLayer
## dimensions  : 180, 360, 64800  (nrow, ncol, ncell)
## resolution  : 0.9972222, 0.8833333  (x, y)
```

```
## extent      : -179.5, 179.5, -79.5, 79.5  (xmin, xmax, ymin, ymax)
## coord. ref. : +proj=longlat +datum=WGS84 +no_defs +ellps=WGS84 +towgs84=0,0,0
## data source : in memory
## names       : layer
## values      : 2.185581e-05, 9.999946  (min, max)
```

I suggest you have a look on the attributes of the *RasterLayer* object defined using **structure** or **attributes** function. Notably, you can visualize values using function **values**:

```
val1 <- values(Ra1)
## I print the ten first values.
val1[1:10]
```

```
##  [1] 7.0841971 5.3563819 3.3810307 0.1200356 1.5798287 8.9251603 8.6659849
##  [8] 5.3162230 1.4668017 9.8247923
```

You may have noticed that there is no "proj4string" argument but a CRS argument. However it exists a common way to access the CRS for all the spatial objects we are studying: function **projection**.

```
projection(Ra1)
```

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

```
projection(ptspd)
```

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

Similarly, function **extent** provides a common way to get the extent of spatial objects:

```
extent(Ra1)
```

```
## class       : Extent
## xmin        : -179.5
## xmax        : 179.5
## ymin        : -79.5
## ymax        : 79.5
```

```
extent(ptspd)
```

```
## class       : Extent
## xmin        : -156.5034
## xmax        : 123.2854
## ymin        : -77.40712
## ymax        : 74.94329
```

Note that using *extent* object is also a convenient way to create a *RasterLayer*:

```
Ra2 <- raster(extent(Ra1), nrows=100, ncols=100, crs=projection(Ra1))
Ra2
```

```
## class       : RasterLayer
## dimensions  : 100, 100, 10000  (nrow, ncol, ncell)
## resolution  : 3.59, 1.59  (x, y)
## extent      : -179.5, 179.5, -79.5, 79.5  (xmin, xmax, ymin, ymax)
## coord. ref. : +proj=longlat +datum=WGS84 +no_defs +ellps=WGS84 +towgs84=0,0,0
```

I now plot *Ra1* using either **plot** or **image**. Note that their default rendering differs.

```
par(mfrow=c(2,1))
plot(Ra1)
image(Ra1)
```

The **raster** function is also a useful method to convert *SpatialGrid* to *RasterLayer*.

```
Ra3 <- raster(grdsp)
Ra3
```

```
## class       : RasterLayer
## dimensions  : 180, 360, 64800  (nrow, ncol, ncell)
## resolution  : 1, 1  (x, y)
## extent      : -180, 180, -90, 90  (xmin, xmax, ymin, ymax)
## coord. ref. : +proj=longlat +datum=WGS84 +ellps=WGS84 +towgs84=0,0,0
```

5.2.2 The *rasterize* function

Rasters can also be created based on vector object such as the one we have previously scrutinized. To do so, you may use the **rasterize** function and a "RasterLayer". The latter is used to define the grid you want to use as a support.

```
# Points as raster.
Ra3 <- rasterize(ptspd,Ra1)
# Lines as raster.
Ra4 <- rasterize(linspd,Ra1)
val4 <- values(Ra4)
# Polygons as raster.
Ra5 <- rasterize(polsp,Ra1)
```

You can plot the raterized polygons as follows:

```
image(Ra4)
```

Note that **rasterize** can also be used to create *rasterlayer* from matrix.

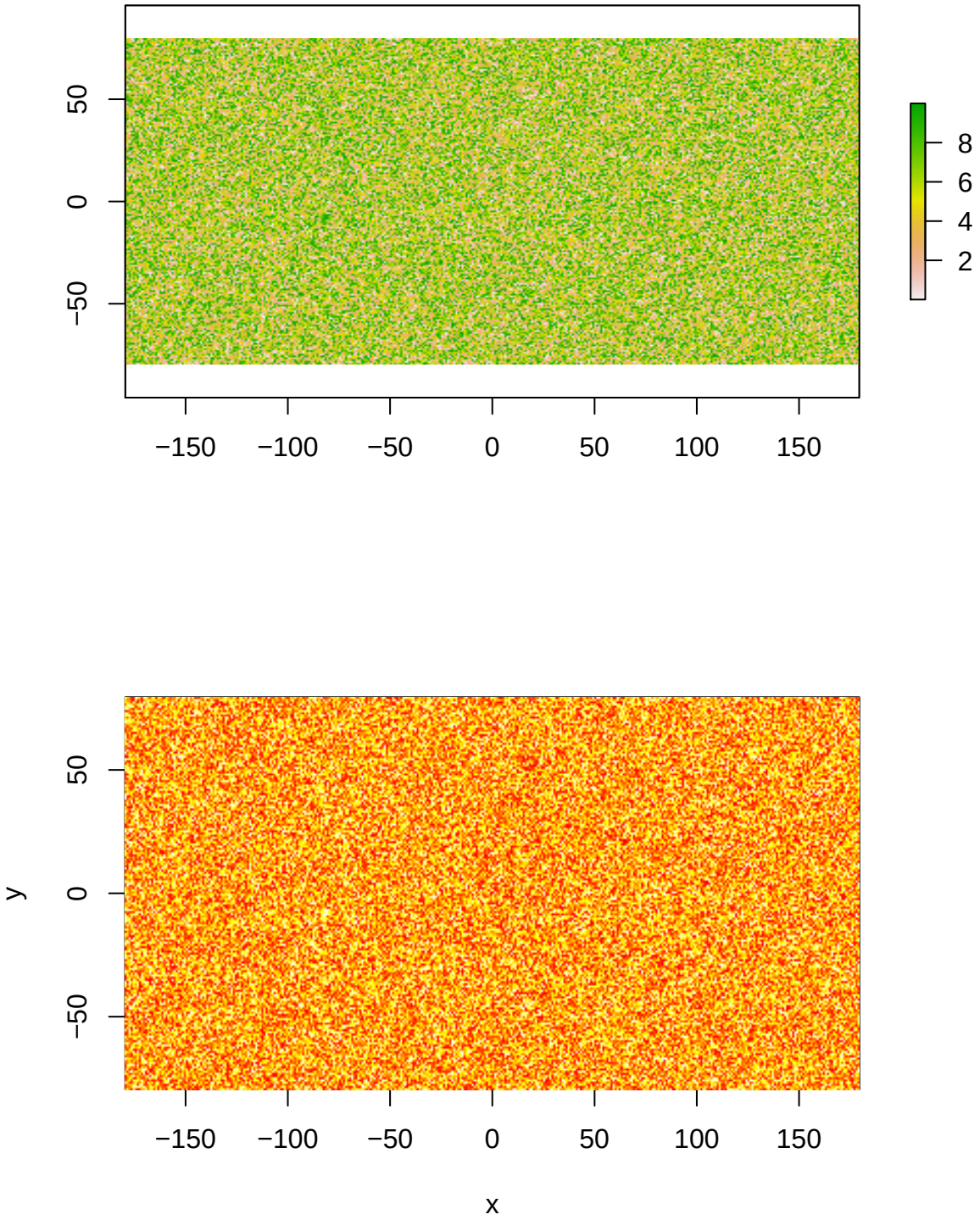

Figure 5.1: Plot of a *RasterLayer* object using *plot* and *image* functions

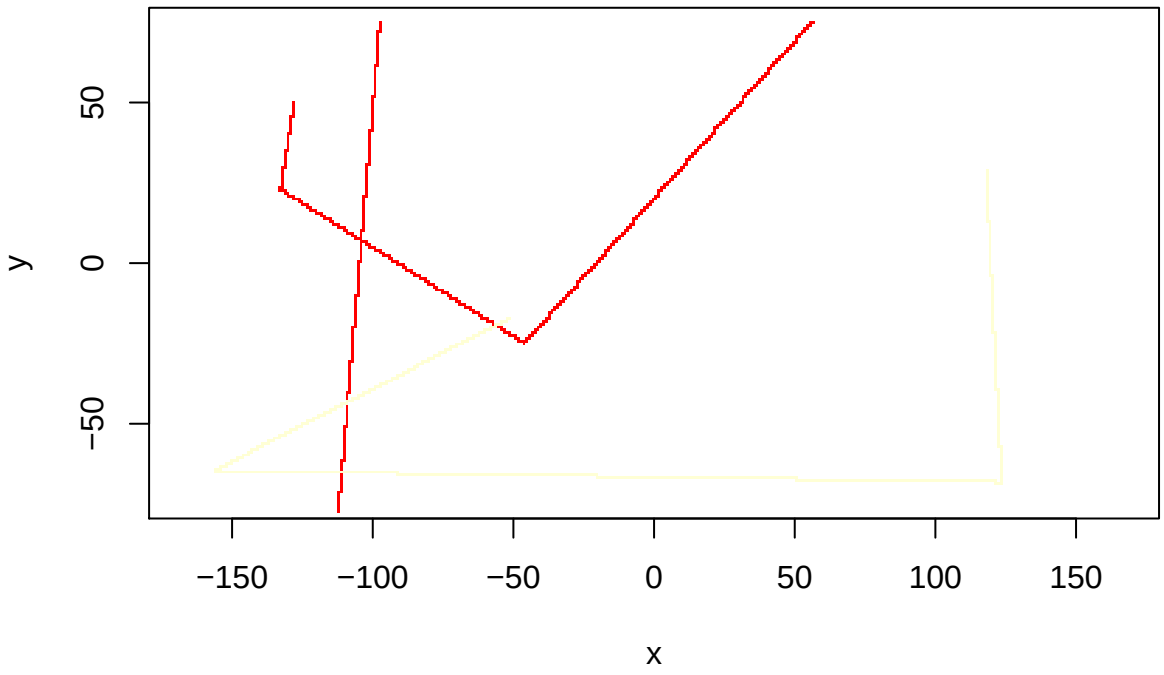

Figure 5.2: SpatialLinesDataFrame rasterized

5.3 RasterStacks and RasterBricks

These classes deal with multi-layer raster objects. They are basically similar but "the processing time should be shorter when using a RasterBrick"[1]. Also, RasterBricks "are less flexible as they can only point to a single file"[2]. To create an object of class *RasterStacks*, one uses the function **stack**. Similarly to create a *RasterBricks* object, one uses **brick**.

```
Ras1 <- stack(Ra1, Ra3, Ra4)
Rab1 <- brick(Ra1, Ra3, Ra4)
```

Let's have a look at the classes of the object:

```
class(Ras1)
```

```
## [1] "RasterStack"
## attr(,"package")
## [1] "raster"
```

```
class(Rab1)
```

```
## [1] "RasterBrick"
## attr(,"package")
## [1] "raster"
```

The numbers of layers is provided by function **nlayers**:

[1] Documentation of function "brick".
[2] Documentation of function "brick".

```
nlayers(Ras1)
```

```
## [1] 5
```

```
nlayers(Rab1)
```

```
## [1] 5
```

Note that you can also look at the class of the different layers:

```
class(Ras1[[1]])
```

```
## [1] "RasterLayer"
## attr(,"package")
## [1] "raster"
```

```
class(Rab1[[1]])
```

```
## [1] "RasterLayer"
## attr(,"package")
## [1] "raster"
```

Again, there is a plot method associated with these objects:

```
plot(Ras1)
```

```
plot(Rab1)
```

Function **unstack** permits the reverse operation, i.e. getting a list of *RasterLayers* from *RasterLayers* or *RasterBricks*:

```
liRa <- unstack(Ras1)
```

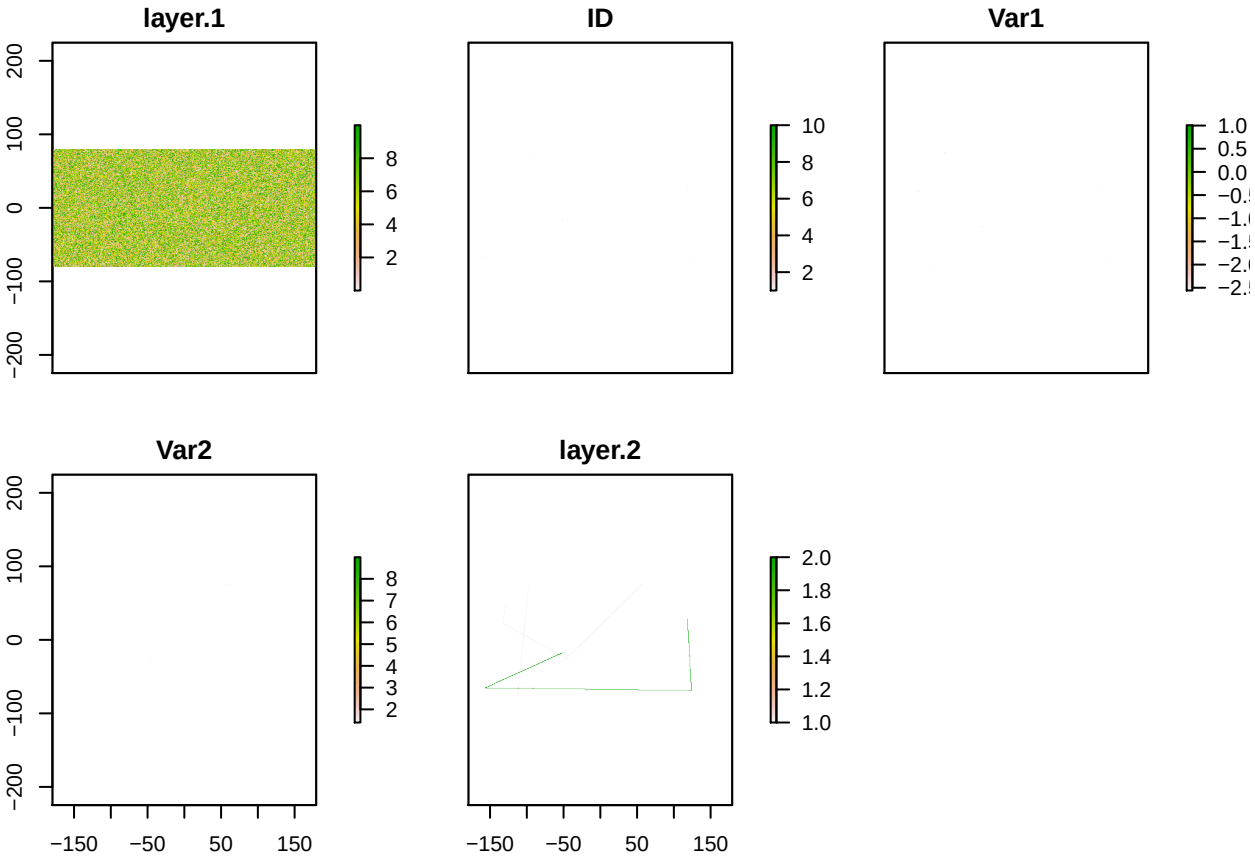

Figure 5.3: Raster

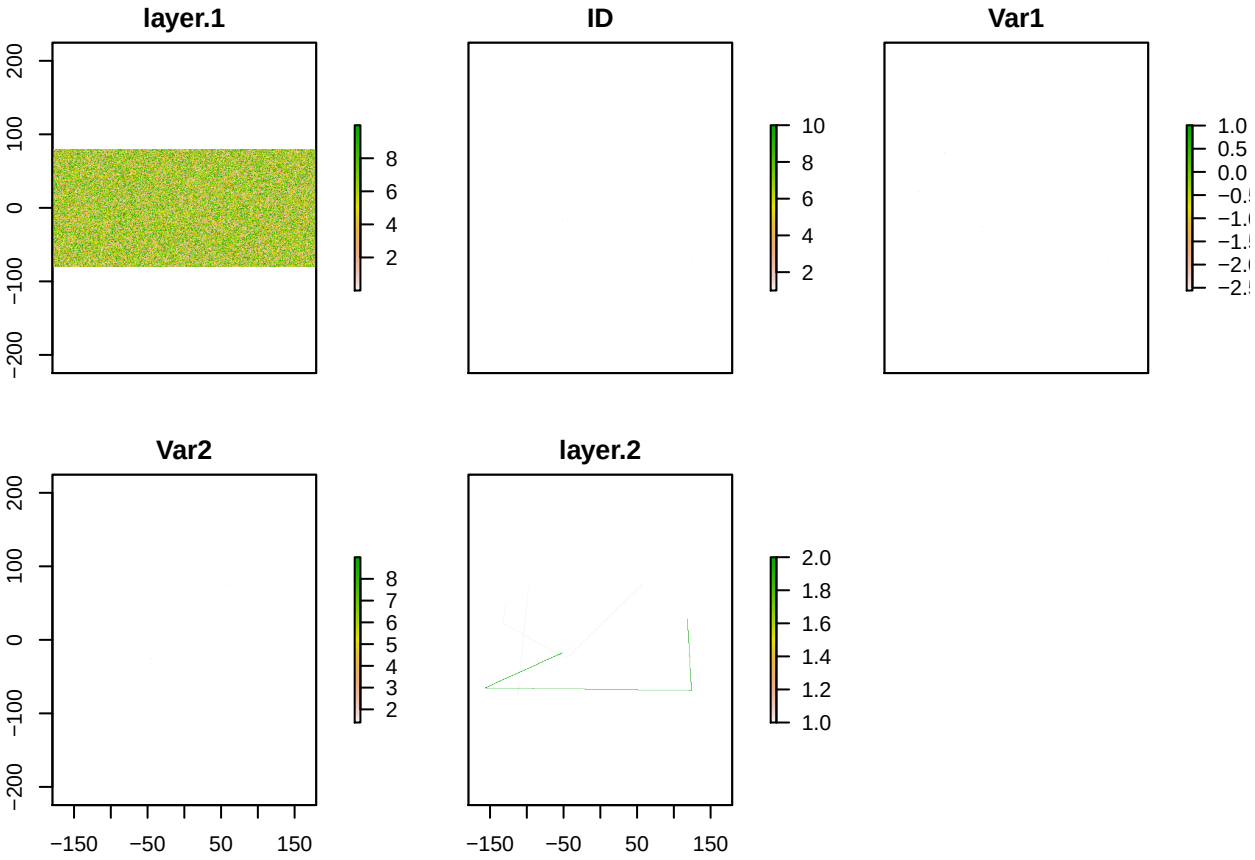

Figure 5.4: Raster

Chapter 6

Import and export spatial objects

6.1 Package "rgdal"

This package makes possible to handle many spatial formats. By doing so, it turns R into a powerful spatial converter. Below, I focus on few important functions of this package:

1. **writeOGR** / **writeGDAL**: to write spatial objects.
2. **readOGR**/**readGDAL**: to read spatial files.
3. **spTransform**: to change geographical projection.

This package takes advantage from the open-source GDAL library to read the different formats and the cartographic projection library PROJ.4 to change the CRS. Therefore, it requires to install them[1]. On August 11, 2015, version 1.11.2 of rgdal and version 4.9.1 of proj are installed on my computer. In GDAL library, "OGR" and "GDAL" refer, respectively, to the distinction between vector and grid formats. Now, I install and load the package:

```
install.packages("rgdal")
library("rgdal")
```

6.2 Export your spatial object.

Once the spatial objects are created, you can save it to share it with others and/or to use it with other GIS. In order to know what the writeable formats are, I use the commands below (not evaluated):

```
ogrDrivers()
gdalDrivers()
```

Note that for OGR, all the displayed format are readable. So far, I have only used "ESRI shapefile", "KML" and "Geotiff". Hereafter, all files exported are stored in a folder called "outputs" I create as follows:

```
dir.create("outputs")
```

```
## Warning in dir.create("outputs"): 'outputs' existe déjà
```

[1] You may take advantage from visiting the associated webpage

Now, I export vector objects I previously built as *ESRI Shapefile*. Argument "layer" specifies the name of your file and argument "overwrite" allow me to replace any layer of the same name[2].

```
writeOGR(ptspd, dsn="./outputs", layer="mypoints",
    driver="ESRI Shapefile", overwrite_layer=TRUE)
writeOGR(linspd, dsn="./outputs", layer="mylines",
    driver="ESRI Shapefile", overwrite_layer=TRUE)
writeOGR(polspd, dsn="./outputs", layer="mypolygons",
    driver="ESRI Shapefile", overwrite_layer=TRUE)
```

We can now export our grid using **writeGDAL**. The default format is Geotiff. You can try different format, there are few examples in the documentation of the function.

```
writeGDAL(grdspd, drivername="GTiff", "./outputs/mygrid.tiff")
```

The "raster" package also provides some functions to export raster objects. The two I often use are **writeRaster** and **KML**. For the first one, we can have a look at the list of supported file types (not evaluated here):

```
writeFormats()
```

I export the rasterized lines as a *Geotiff* file:

```
writeRaster(Ra4, filename="./outputs/rast4.tiff", format="GTiff", overwrite=TRUE)
```

I also export the rasterized polygon as a *.kmz* file you can read using GoogleEarth.

```
KML(Ra5,"./outputs/rast4.kml", overwrite=TRUE)
```

6.3 Import your spatial objects.

This is the reverse operation: turning spatial files into R spatial objects. There three main functions: **readOGR** and **readGDAL** of package "rgdal" and **raster** for package "raster". For the sake of illustration, I willfully import objects I previously exported. First, I use **readOGR** to import the shape file "mypolygons". As it is a "ESRI shapelfile", argument "dsn" must be a path of a folder and argument layer must be name before the file extension within the folder (similar for the four "ESRI shapelfile" is made of).

```
mypol <- readOGR(dsn="outputs/", layer="mypolygons")
```

```
## OGR data source with driver: ESRI Shapefile
## Source: "outputs/", layer: "mypolygons"
## with 1 features
## It has 3 fields
```

You can check whether "mypol" and "polspd" are similar:

[2]It was quite helpful during the edition of this document!

```
mypol
```

```
## class       : SpatialPolygonsDataFrame
## features    : 1
## extent      : -132.8066, 55.91372, -77.40712, 74.94329  (xmin, xmax, ymin, ymax)
## coord. ref. : +proj=longlat +datum=WGS84 +no_defs +ellps=WGS84 +towgs84=0,0,0
## variables   : 3
## names       :              var1,               var2, var3
## min values  : 0.391229668166488, -0.675002882424971,    8
## max values  : 0.391229668166488, -0.675002882424971,    8
```

```
polspd
```

```
## class       : SpatialPolygonsDataFrame
## features    : 1
## extent      : -132.8066, 55.91372, -77.40712, 74.94329  (xmin, xmax, ymin, ymax)
## coord. ref. : +proj=longlat +datum=WGS84 +ellps=WGS84 +towgs84=0,0,0
## variables   : 3
## names       :              var1,               var2, var3
## min values  : 0.391229668166488, -0.675002882424971,    8
## max values  : 0.391229668166488, -0.675002882424971,    8
```

I now import "rast3.tif" using both **readGDAL** and **raster**.

```
Ra5 <- raster("./outputs/mygrid.tiff")
Ra5
```

```
## class       : RasterLayer
## band        : 1  (of  2  bands)
## dimensions  : 180, 360, 64800  (nrow, ncol, ncell)
## resolution  : 1, 1  (x, y)
## extent      : -180, 180, -90, 90  (xmin, xmax, ymin, ymax)
## coord. ref. : +proj=longlat +datum=WGS84 +no_defs +ellps=WGS84 +towgs84=0,0,0
## data source : /Users/kcazelles/Documents/Codes/R_workspace/GeoR/MapsR-eng/outputs/mygrid.tiff
## names       : mygrid
```

```
Ra3
```

```
## class       : RasterBrick
## dimensions  : 180, 360, 64800, 3  (nrow, ncol, ncell, nlayers)
## resolution  : 0.9972222, 0.8833333  (x, y)
## extent      : -179.5, 179.5, -79.5, 79.5  (xmin, xmax, ymin, ymax)
## coord. ref. : +proj=longlat +datum=WGS84 +no_defs +ellps=WGS84 +towgs84=0,0,0
## data source : in memory
## names       :        ID,      Var1,      Var2
## min values  :  1.000000, -2.559373,  1.396514
## max values  : 10.000000,  1.007905,  8.994566
```

6.4 From one CRS to another

When working with different data source, the projection of spatial objects may differ. Therefore, it is essential to navigate efficiently from one CRS to another. The function to be used is **spTransform** which actually calls the *Cartographic Projection Library*, PROJ.4 to convert CRS.

```
(mypol <- spTransform(polspd, CRS=CRS("+proj=merc +ellps=GRS80")))
```

```
## class       : SpatialPolygonsDataFrame
## features    : 1
## extent      : -14783959, 6224287, -14016965, 12866580  (xmin, xmax, ymin, ymax)
## coord. ref. : +proj=merc +ellps=GRS80
## variables   : 3
## names       :              var1,               var2, var3
## min values  : 0.391229668166488, -0.675002882424971,    8
## max values  : 0.391229668166488, -0.675002882424971,    8
```

Many examples are given in the documentation of the function of the "rgdal" package. Note that function **spTransform** is already defined is package "sp" but transformations are actual when we "rgdal" package is installed.

It is beyond the scope of this document (and beyond the scope of my skills) to develop about projections available. However, if you are eager to learn more, I recommend the visit of remotesensing.org I found on the PROJ.4 website.

Chapter 7

Free geographic data

7.1 Resources available on line

When I started handling spatial data I was seeking for free data for hours. I have found many websites and even better: a very good index of free GIS datasets listed by Robin Wilson:

```
browseURL("http://freegisdata.rtwilson.com/")
```

For some free datasets, there are R packages dedicated to create requests on line and directly import desired data. For instance, for the "Global Biodiversity Information Facility" (GBIF), there is an associated R package: "rgbif". For similar package, I suggest you visit the website of R open science.

7.2 Function *getData*

In package "raster", *getData* function generates requests to access to different spatial datasets. Argument "name" specifies the types of variable required. I detail below the different possibilities offered.

7.2.1 name="GADM"

"GADM" stands for Global ADMinistrative Areas which are the data tghis option provides. To access to data at country level, the argument "country" must be specified in the form of a country code. The list of codes can be printed in the R console as follows :

```
getData("ISO3")
```

Once the country is selected, then "level" argument must be species to define the desired administrative subdivision. For instance, to obtain the first levels of administrative areas of Belgium, I do as in:

```
## Country level:
mapBEL0 <- getData(name="GADM", country="BEL", path="./outputs", level=0)
## First level of administrative divisions:
mapBEL1 <- getData(name="GADM", country="BEL", path="./outputs", level=1)
```

Let's look at *mapBEL0*:

```
class(mapBEL0)
```

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

It is a "SpatialPolygonsDataFrame" as seen above. So, remember you can plot it quickly:

```
plot(mapBEL1, lty=2, lwd=0.8)
plot(mapBEL0, lwd=1.2, add=TRUE)
```

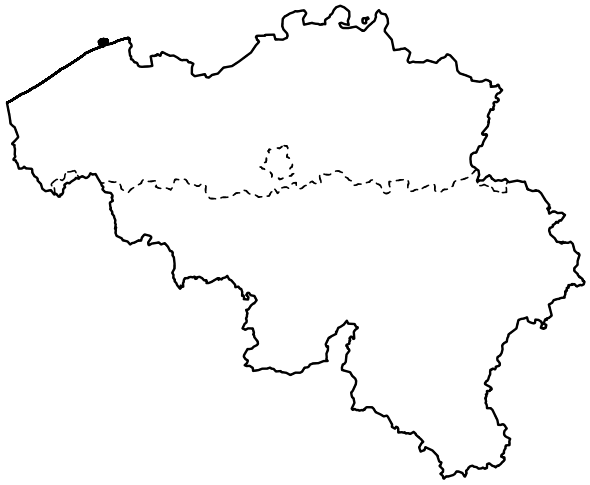

Figure 7.1: Quick plot of Belgium

7.2.2 name="alt"

This three letters stand for altitude (elevation), this option provides a way to download data of the Shuttle Radar Topography Mission (SRTM):

```
browseURL("http://srtm.csi.cgiar.org")
```

It provides data with a resolution of 90 meters (at the equator) mosaiced as tiles of 5 degrees x 5 degrees. Again, the argument "country" must be specified.

```
altBEL <- getData(name="alt", country="BEL", path="./outputs")
```

As usual, I check its class.

```
class(altBEL)
```

```
## [1] "RasterLayer"
## attr(,"package")
## [1] "raster"
```

This is a "RasterLayer" object we already know about and so, we plot it.

```
plot(altBEL, xlab="Longitude", ylab="Latitude")
```

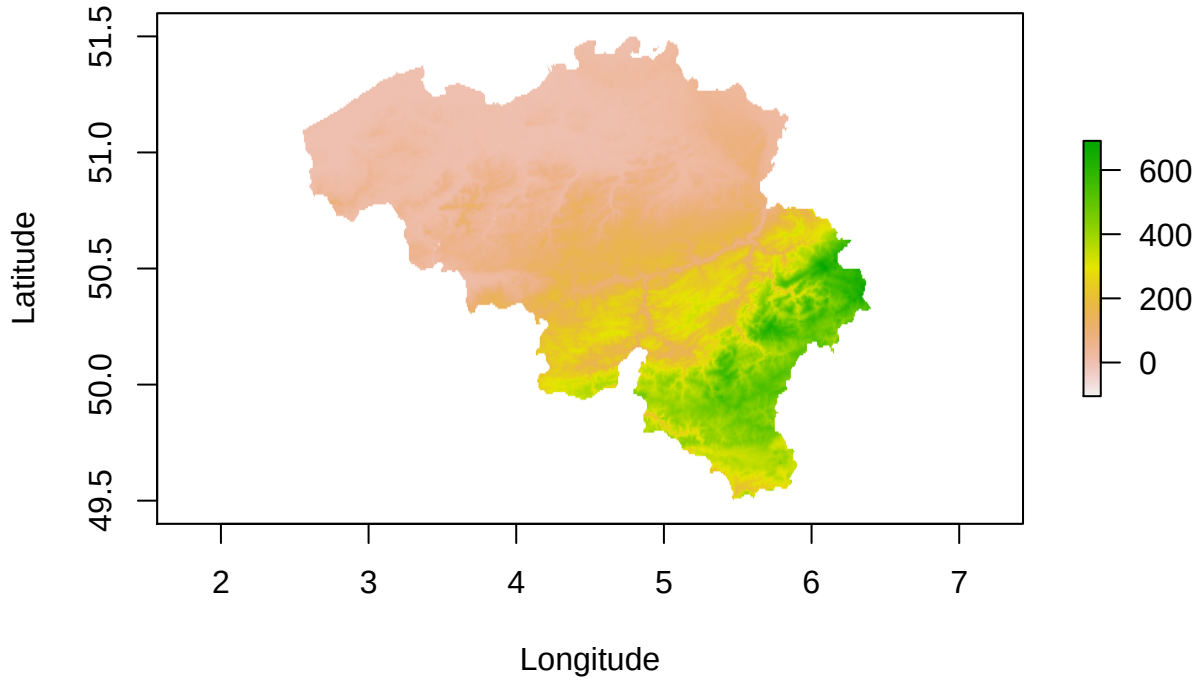

Figure 7.2: Elevation raster of Belgium

7.2.3 name="worldclim"

It retrieves data from WolrdClim. Once this option is selected, arguments *var* and *res* must be specified. The *var* names are 'tmin', 'tmax', 'prec' and 'bio'. Also, valid resolution, *res*, are 0.5, 2.5, 5, and 10 (minutes of a degree). Let's retrieve the temperature minimum with a resolution of 10 minutes. I store the file in the folder "outputs".

```
tminW <- getData(name="worldclim", var="tmin", res=10, path="./outputs")
```

Again, I look at the type of objet I got:

```
class(tminW)
```

```
## [1] "RasterStack"
## attr(,"package")
## [1] "raster"
```

and I plot it:

```
plot(tminW[[1:4]])
```

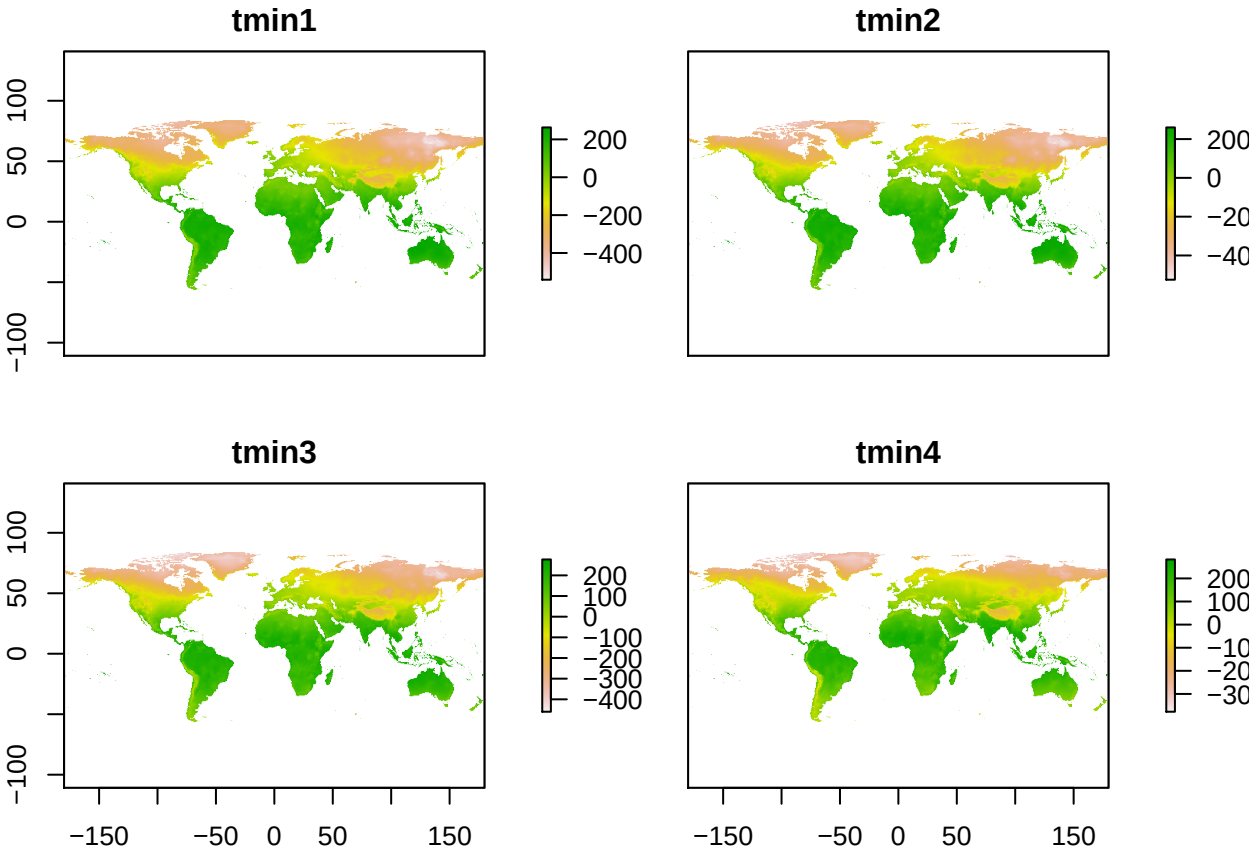

Figure 7.3: First four Worldwide minimum temperature of the rasterStack *tminW*, resolution=10 minutes of a degree

7.2.4 name="CMIP5"

This argument allow us to get climate projections. Once this argument is selcted, the user must still specify *var* and *res* but also *year*, *model* and *rcp*. Look a the documentation to see the valuea and below for an example.

```
pred1 <- getData(name="CMIP5", var="tmin", year=50, model="HD", rcp=45, res=10, path="./outputs")
class(pred1)
```

```
## [1] "RasterStack"
## attr(,"package")
## [1] "raster"
```

```
plot(pred1)
```

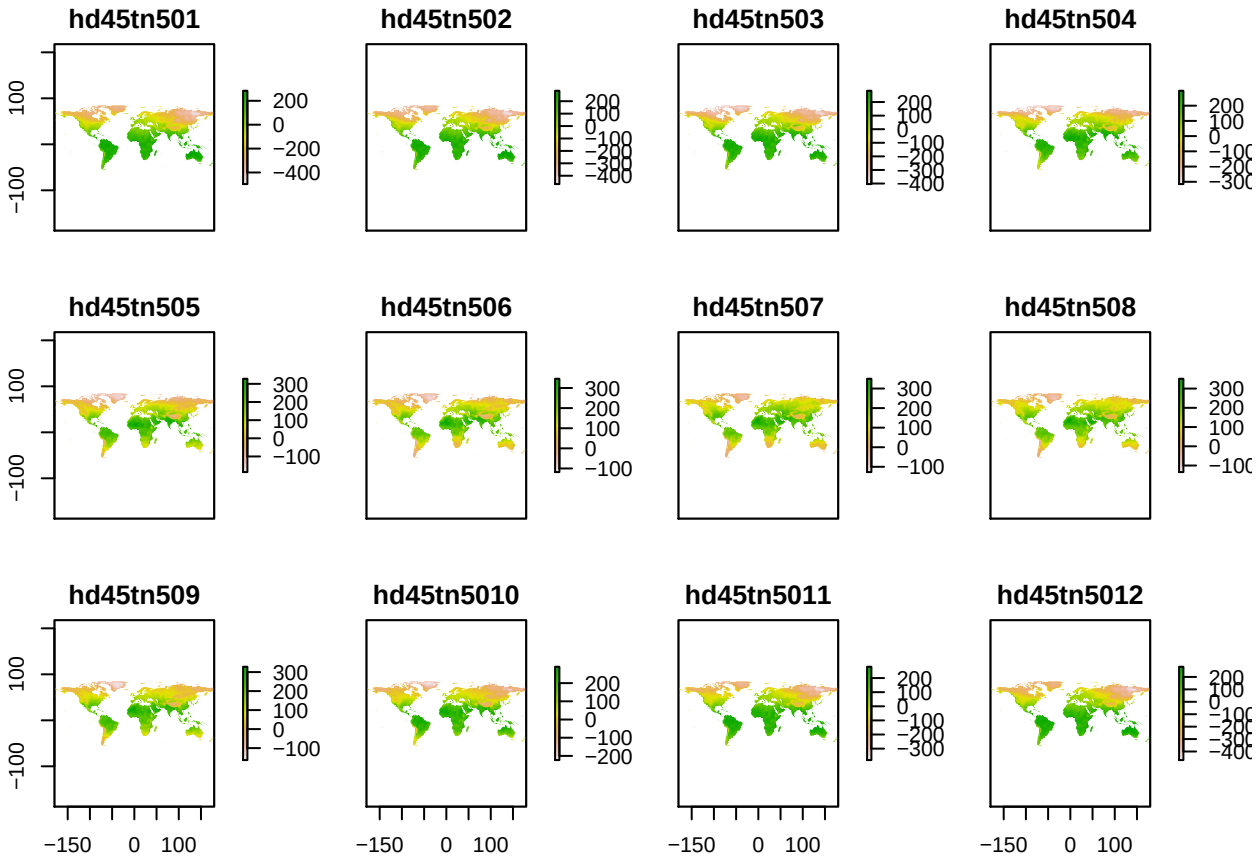

Figure 7.4:

7.2.5 Alternative methods

There are R different functions to automate data retrieval. You can go in deep with the "RCurl" package and/or the package "Downloader". There is also the function **download.file()** in the "utils" package which I use as in:

```
download.file(url="http://biogeo.ucdavis.edu/data/diva/wat/BEL_wat.zip", destfile="./outputs/watBEL.zip
unzip("./outputs/watBEL.zip", exdir="./outputs/BEL_wat")
```

Then I import these data and plot it:

```
watBEL <- readOGR(dsn="outputs/BEL_wat/", "BEL_water_lines_dcw")
```

```
## OGR data source with driver: ESRI Shapefile
## Source: "outputs/BEL_wat/", layer: "BEL_water_lines_dcw"
## with 220 features
## It has 5 fields
```

```
# Areas (such as lakes)
wataBEL <- readOGR(dsn="outputs/BEL_wat/", "BEL_water_areas_dcw")
```

```
## OGR data source with driver: ESRI Shapefile
## Source: "outputs/BEL_wat/", layer: "BEL_water_areas_dcw"
## with 18 features
## It has 5 fields
```

```
# Plot
plot(watBEL, col=4)
plot(wataBEL, col=4, border=4, add=TRUE)
```

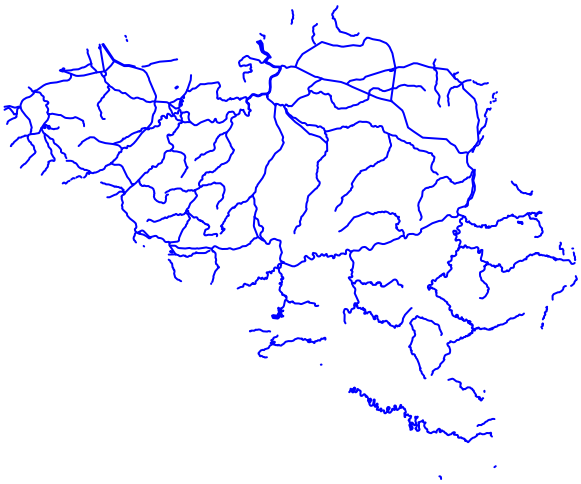

Figure 7.5: Waters in Belgium

Chapter 8

First customized map

In the last chapter, I have gathered several spatial objects that I can assemble to draw a map of Belgium. There is nothing complicated, however, to get a good-looking map it requires some graphical skills. First trick is to draw a base map. To do so, I draw a rectangle of the exact dimension of the plot region and color it as in:

```
BELbox <- mapBEL0@bbox
plot(BELbox[1,],BELbox[2,], type="n", ann=FALSE, axes=FALSE, asp=1.53)
rect(par()$usr[1],par()$usr[3],par()$usr[2],par()$usr[4], col="lightblue",
    border="transparent")
```


Figure 8.1: Base map

Then I add the elevation raster and I chose the color palette thank to **colorRampPalette()** function:

```
mypal <- colorRampPalette(c("white","brown","black"))
image(altBEL, add=TRUE, col=mypal(100))
```

We then add the water lines and areas:

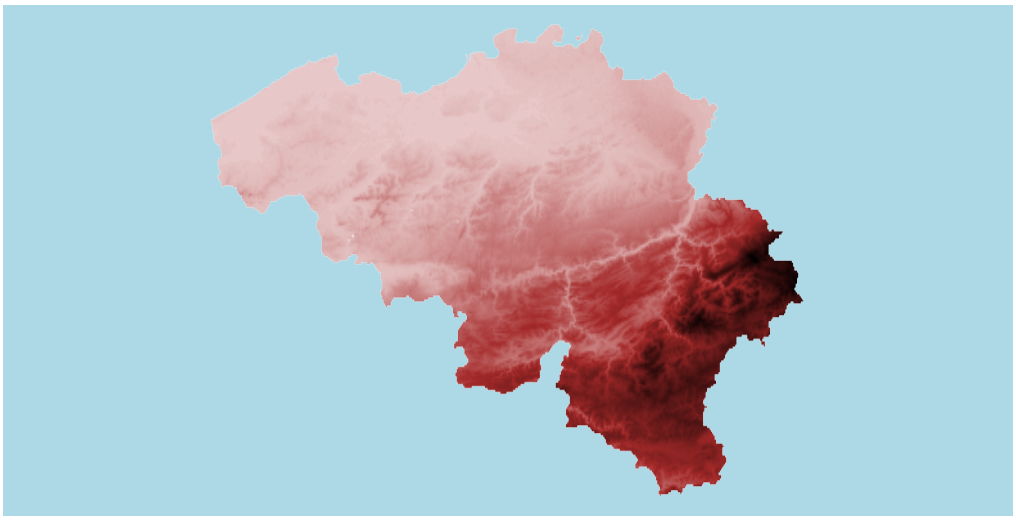

Figure 8.2: Base map + elevation raster

```
plot(watBEL, col="lightblue", add=TRUE)
plot(wataBEL, col="lightblue", border=4, add=TRUE)
```

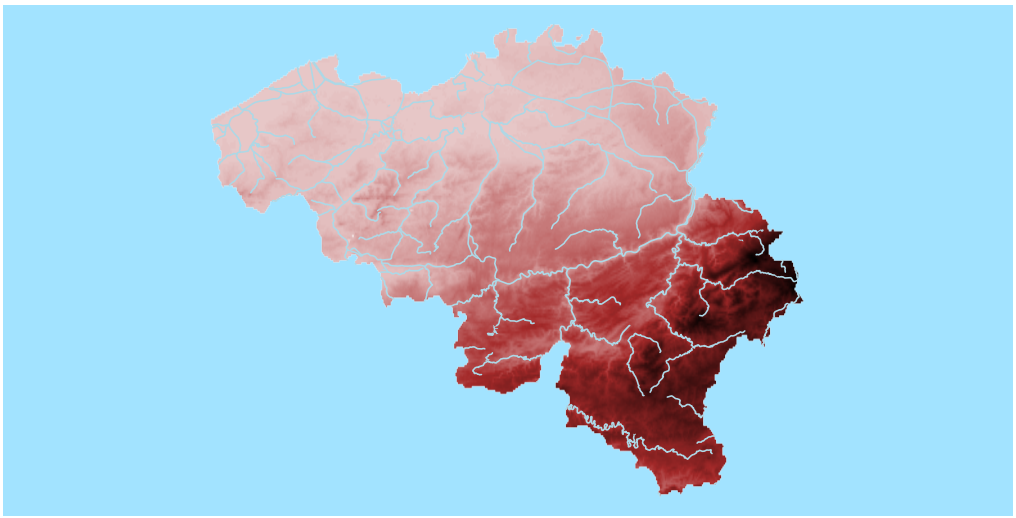

Figure 8.3: Base map + elevation raster + waters

Then we add the administrative boundaries:

```
plot(mapBEL0, lwd=1.2, add=TRUE)
plot(mapBEL1, lty=2, lwd=0.6, add=TRUE)
```

You may want to add :

```
mapDEU0 <- getData(name="GADM", country="DEU", path="./outputs", level=0)
mapFRA0 <- getData(name="GADM", country="FRA", path="./outputs", level=0)
mapLUX0 <- getData(name="GADM", country="LUX", path="./outputs", level=0)
mapNLD0 <- getData(name="GADM", country="NLD", path="./outputs", level=0)
```

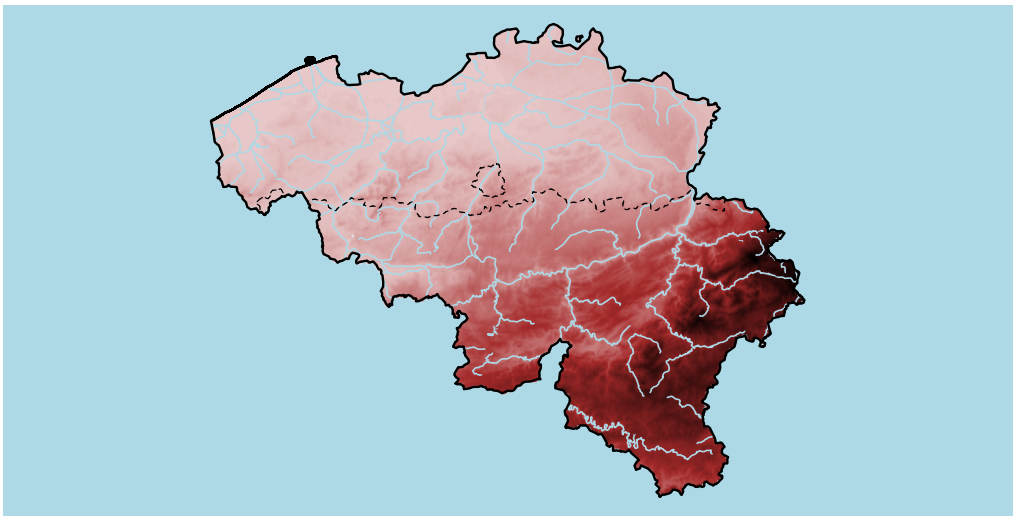

Figure 8.4: Base map+ elevation raster+ waters+ administrative boubaries

We add them on the map.

```
plot(mapFRA0, add=TRUE, col="#B0CC99", border="transparent")
plot(mapDEU0, add=TRUE, col="#C79F4B", border="transparent")
plot(mapLUX0, add=TRUE, col="#9E8479", border="transparent")
plot(mapNLD0, add=TRUE, col="#B0CC99", border="transparent")
```

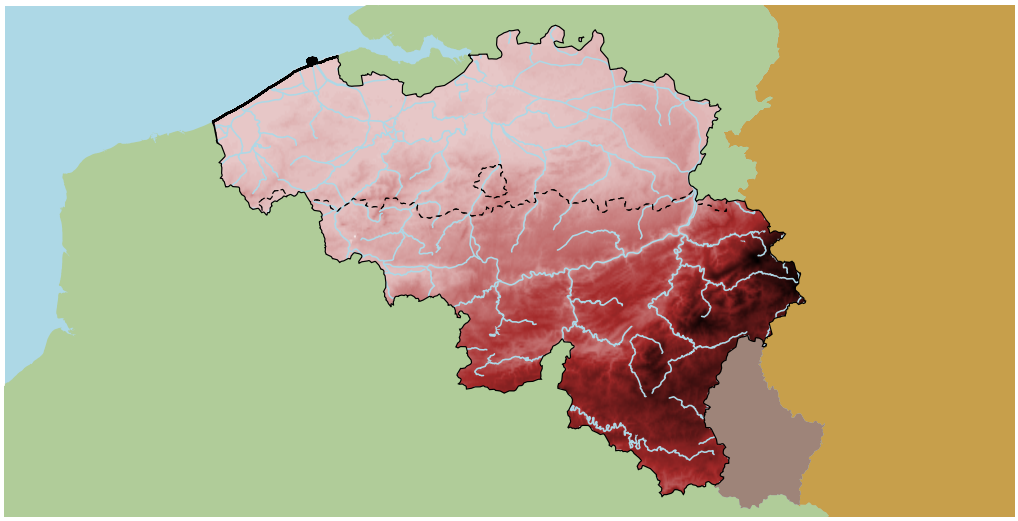

Figure 8.5: Base map+ elevation raster+ waters+ administrative boubaries

Chapter 9

Basic geometry manipulation

9.1 Packages "rgeos"

It provides an efficient interface with the Geometry Engine Open-Source which must be installed to install. Currently (august 2015) I have the 3.4.2 version installed. Remember to look at the documentation of the package.

```
install.packages("rgeos")
library(rgeos)
```

9.2 Unions

Let us start by plotting the interior administrative areas of Belgium (level 2).

```
mapBEL2 <- getData(name="GADM", country="BEL", path="./outputs", level=2)
plot(mapBEL2)
text(coordinates(mapBEL2), labels=seq(1,length(mapBEL1)), col=2)
```

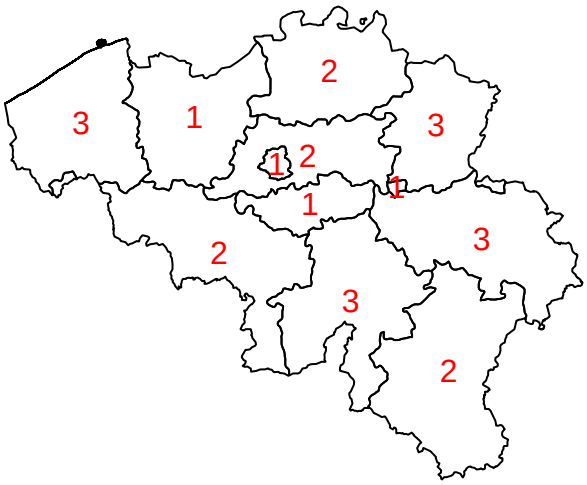

Figure 9.1:

```
slc <- c(8,11,12)
mapBELts <- mapBEL2[slc,]
class(mapBELts)
```

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

```
mapBELS <- gUnionCascaded(mapBELts)
mapBELN <- gUnionCascaded(mapBEL2[-slc,])
mapBELSN <- gUnionCascaded(mapBEL2[c(slc,2),])
```

```
par(mfrow=c(1,3), mar=c(1,1,1,1))
plot(mapBEL2)
##
plot(mapBEL2)
plot(mapBELS, add=T, col=2)
plot(mapBELN, add=T, col=4)
##
plot(mapBEL2)
plot(mapBELSN, add=T, col=3)
```

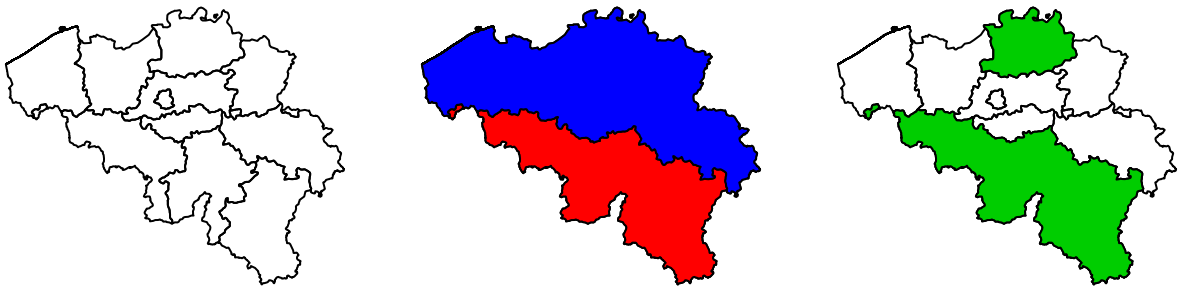

Figure 9.2:

```
par(mfrow=c(1,2))
plot(mapBEL1)
plot(gIntersection(mapBELSN, mapBELS), col=5, add=T)
plot(mapBEL1)
plot(gIntersection(mapBELSN, mapBELN), col=6, add=T)
```

9.3 Buffers

```
par(mfrow=c(1,1))
plot(mapBEL1)
plot(gBuffer(mapBELS, width=0.5), add=T, lwd=2, lty=2)
plot(gBuffer(mapBELS, width=0.1), add=T, lwd=3)
plot(mapBELS, add=T, col=2)
```

Figure 9.3:

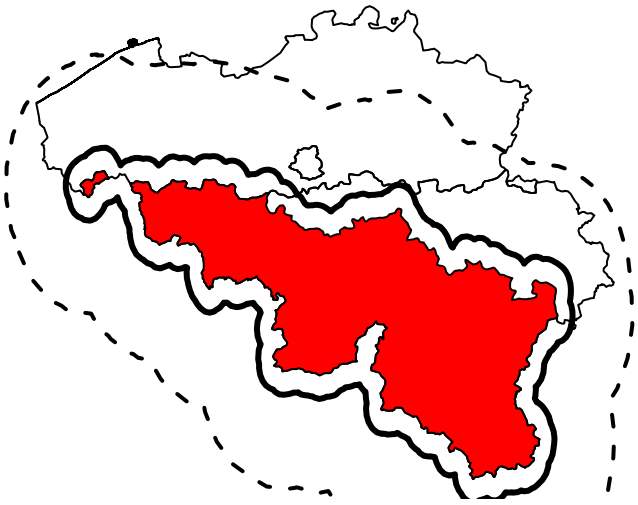

Figure 9.4:

9.4 Difference

```
Diff <- gDifference(gBuffer(mapBELS, width=0.5), gBuffer(mapBELS, width=0.1))
```

```
## Warning in gBuffer(mapBELS, width = 0.5): Spatial object is not projected;
## GEOS expects planar coordinates
```

```
## Warning in gBuffer(mapBELS, width = 0.1): Spatial object is not projected;
## GEOS expects planar coordinates
```

```
plot(mapBEL1)
plot(mapBELS, add=T, col=4)
plot(Diff, add=T, lwd=2, lty=3, col=2)
plot(mapBELS, add=T, col=2)
```

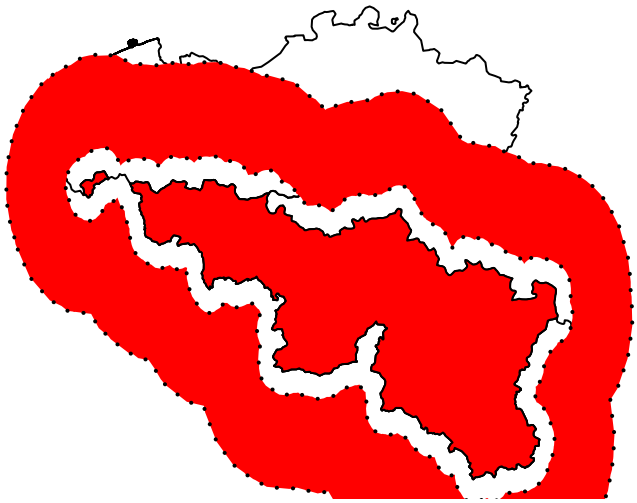

Figure 9.5:

9.5 Overlays

One very useful tool is the **over()** function which provides a consistent spatial overlay. For instance, here, I have defined points and one grid, in order to get the identities of cells my points belong to, I can use the **over** function.

```
over(ptspd,grdsp) # or ptspd%over%grdsp
```

```
##     1     2     3     4     5     6     7     8     9    10
##  5636 41174 24168 14093 60188  5483 22258 57184 55464 38649
```

Chapter 10

Basic Rasters Manipulation

10.1 Crop and mask

```
plot(rasterize(mapBEL0, altBEL, mask=TRUE))
```

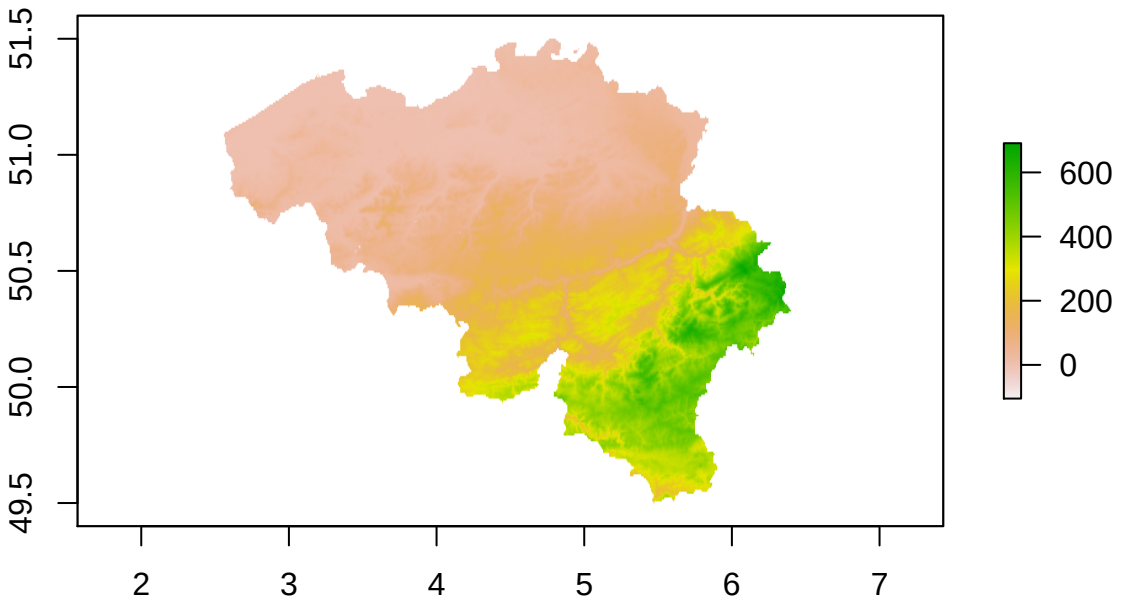

Figure 10.1:

10.2 Resample