Spatial Data Analysis with R

Release 0.1

Robert Hijmans

August 18, 2016

CONTENTS

CHAPTER

ONE

1. INTRODUCTION

This is an introduction to spatial data manipulation with *R*. In this context "spatial data" refers to data about geographical locations, that is, places on earth. So to be more precise, we should speak about "geospatial" data, but we use the shorthand "spatial".

This is the introductory part of a set of resources for learning about spatial analysis and modeling with *R*. Here we cover the basics of data manipulation. When you are done with this section, you can continue with the introduction to spatial data analysis.

You need to need to know some of the basics of the *R* language before you can work with *spatial data* in *R*. If you have not worked with *R* before, or not recently, have a look at this brief introduction to R.

You can download this manual as a pdf.

CHAPTER

TWO

2. SPATIAL DATA

2.1 2.1 Introduction

Spatial phenomena can generally be thought of as either discrete locations (objects with boundaries) or to a continuous phenomenon that can be observed everywhere, but does not have natural boundaries. Discrete locations, or "spatial objects" may refer to a river or road, country or town, or a research site. Examples of continuous phenomena, or "spatial fields" include elevation, temperature, and air quality.

Spatial objects are usually represented by *vector* data. Such data consists of a description of the "geometry" or "shape" of the locations, and normally also includes variables with additional information about the locations. For example, a vector data set may describe the borders of the countries of the world, and also store their names and the size of their population in 2015; or the roads in an area, and their type and names. These variables are often referred to as "attributes". Spatial fields are usually represented by *raster*. We discuss these two data types in turn.

2.2 2.2 Vector data

The main vector data types are **points**, **lines** and **polygons**. In all cases, the geometry of these data structures consists of sets of coordinate pairs (x, y). Points are the simplest case. Each point has one coordinate pair, and *n* associated variables. For example, a point might represent a place where a rat was trapped, and the attributes could include the date it was captured, the person who captured it, the species size and sex, and information about the habitat. It is also possible to combine several points into a multi-point structure, with a single attribute record. For example, all the coffee shops in a town could be considered as a single geometry.

The geometry of **lines** is a just a little bit more complex. First note that in this context, the term 'line' refers to a set of one or more polylines (connected series of line segments). For example, in spatial analysis, a river and all its tributaries could be considered as a single 'line' (but they could also also be several lines, perhaps one for each tributary river). Lines are represented as ordered sets of coordinates (nodes). The actual line segments can be computed (and drawn on a map) by connecting the points. Thus, the representation of a line is very similar to that of a multi-point structure. The main difference is that the ordering of the points is important, because we need to know which points should be connected. A **network** (e.g. a road or river network), or spatial graph, is a special type of lines geometry where there is additional information about things like flow, connectivity, direction, and distance.

A **polygon** refers to a set of closed polylines. The geometry is very similar to that of lines, but to close a polygon the last coordinate pair coincides with the first pair. A complication with polygons is that they can have holes (that is a polygon entirely encolsed by another polygon, that serves to remove parts of the encolosing polygon (for examplel to show an island inside a lake. Also, valid polygons do not self-intersect (but it is OK for a line to self-cross). Again, multiple polygons can be considered as a single geometry. For example the United States state of Hawaii consists of several islands. Each can be represented by a single polygon, but together then can be represent a single (multi-) polygon of the Hawaiian islands.

2.3 2.3 Raster data

Raster data is commonly used to represent continuous variables. A raster divides the world into a grid of equally sized rectangles (referred to as cells or, in the context of remote sensing, pixels) that all have a values (or a missing value) for the variables of interest. A raster cell value should normally represeent the average (or majority) value for the area it covers. However, in some cases the values are actually estiamtes for the center of the cell (in essensce becoming a regular set of points with an attribute).

In contrast to vector data, in raster data the geometry is not explicitly stored as coordinates. It is implicitly set by knowing the spatial extent and the number or rows and columns in which the area is divided. From the extent and number of rows and columns, the size of the raster cells (spatial resolution) can be computed. While raster cells can be thought of as a set of regular polygons, it would be very inefficient to represent the data that way as coordiantes for each cell would have to be stored explictly. It would also dramatically increase processing speed in most cases.

Continuous surface data are sometimes stored as triangulated irregular networks (TINs); these are not discussed here.

2.4 2.4 Simple representation of spatial data

The basic data types in *R* are numbers, characters, logical (TRUE or FALSE) and factor values. Values of a single type can be combined in vectors and matrices, and variables of multiple types can be combined into a data.frame. We can represent (only very) basic spatial data with these data types. Let's say we have the location (represented by longitude and latitude) of ten weather stations (named A to J) and their annual precipitation.

In the example below we make a very simple map. Note that a *map* is special type of plot (like a scatter plot, barplot, etc.). A map is a plot of geospatial data that also has labels and other graphical objects such as a scale bar or legend. The spatial data itself should not be referred to as a map.

```
name <- LETTERS[1:10]
longitude <- c(-116.7, -120.4, -116.7, -113.5, -115.5,
               -120.8, -119.5, -113.7, -113.7, -110.7)
latitude <- c(45.3, 42.6, 38.9, 42.1, 35.7, 38.9,
              36.2, 39, 41.6, 36.9)
stations <- cbind(longitude, latitude)
# Simulated rainfall data
set.seed(0)
precip <- (runif(length(latitude))*10)^3
```

A map of point locations is not that different from a basic x-y scatter plot. Here I make a plot (a map in this case) that shows the location of the weather stations, and the size of the dots is proportional to the amount of precipitation. The point size is set with argument `cex`.

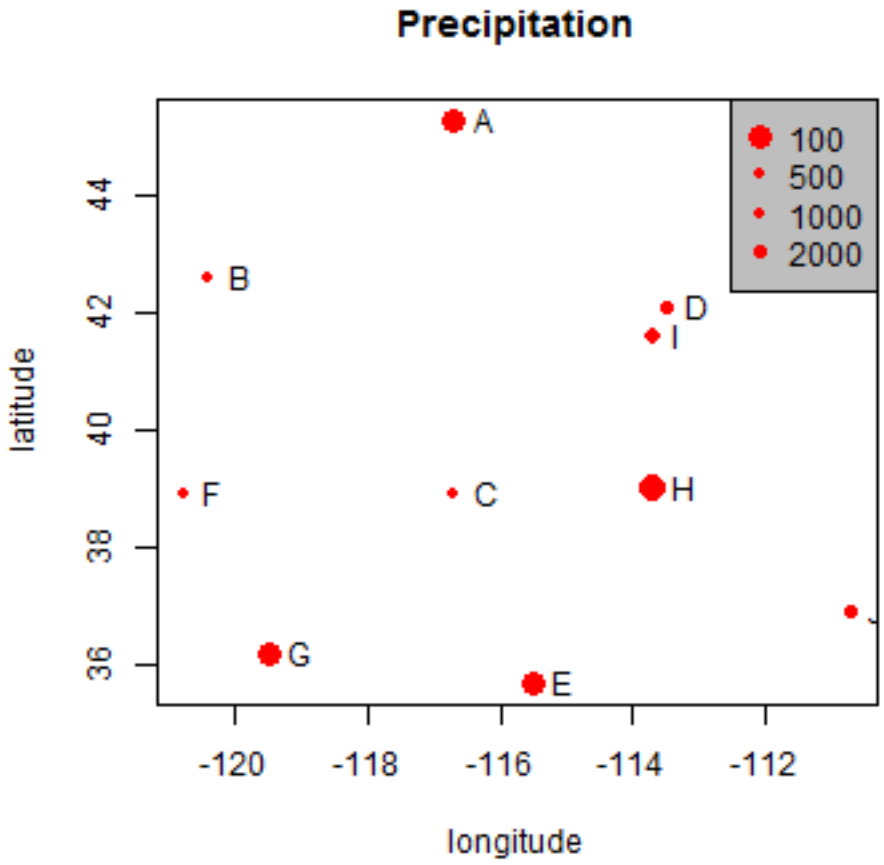
```
psize <- 1 + precip/500
plot(stations, cex=psize, pch=20, col='red', main='Precipitation')

# add names to plot
text(stations, name, pos=4)

# add a legend
breaks <- c(100, 500, 1000, 2000)
legend("topright", legend=breaks, pch=20, pt.cex=psize, col='red', bg='gray')
```

Note that the data are represented by "longitude, latitude", in that order, do not use "latitude, longitude" because on most maps latitude (North/South) is used for the vertical axis and longitude (East/West) for the horizontal axis. This is important to keep in mind, as it is a very common source of mistakes!

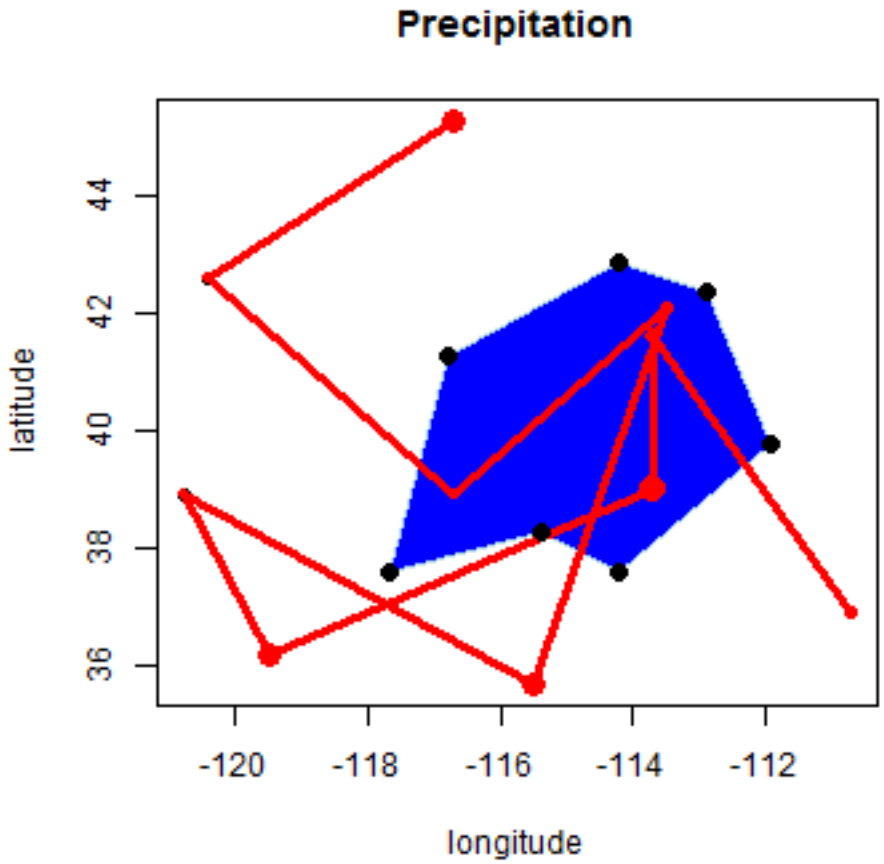
We can add multiple sets of points to the plot, and even draw lines and polygons:


Precipitation
A
B
C
D
E
F
G
H
I
100
500
1000
2000
latitude
longitude
36
38
40
42
44
-120
-118
-116
-114
-112

```
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)
x <- cbind(lon, lat)

plot(stations, main='Precipitation')

polygon(x, col='blue', border='light blue')
lines(stations, lwd=3, col='red')
points(x, cex=2, pch=20)
points(stations, cex=psize, pch=20, col='red', main='Precipitation')
```



The above illustrates how numeric vectors representing locations can be used to draw simple maps. It also shows how points can (and typically are) represented by pairs of numbers, and a line and a polygons by a number of these points. Polygons is that they are "closed", i.e. the first point coincides with the last point, but the `polygon` function took care of that for us.

There are cases where a simple approach like this may suffice and you may come across this in older *R* code or packages. Likewise, raster data could be represented by a matrix or higher-order array. Particularly when only dealing with point data such an approach may be practical. For example, a spatial data set representing points and attributes could be made by combining geometry and attributes in a single 'data.frame'.

```
wst <- data.frame(longitude, latitude, name, precip)
wst
##    longitude latitude name     precip
```

```
## 1     -116.7     45.3    A 721.003613
## 2     -120.4     42.6    B  18.716993
## 3     -116.7     38.9    C  51.530302
## 4     -113.5     42.1    D 187.988119
## 5     -115.5     35.7    E 749.127376
## 6     -120.8     38.9    F   8.203534
## 7     -119.5     36.2    G 725.093932
## 8     -113.7     39.0    H 843.038944
## 9     -113.7     41.6    I 288.539816
## 10    -110.7     36.9    J 248.993575
```

However, `wst` is a data.frame and *R* does not automatically understand the special meaning of the first two columns, or to what coordinate reference system it refers (longitude/latitude, or perhaps UTM zone 17S, or ....?).

Moreover, it is non-trivial to do some basic spatial operations. For example, the blue polygon drawn on the map above might represent a state, and a next question might be which of the 10 stations fall within that polygon. And how about any other operation on spatial data, including reading from and writing data to files? To facilitate such operation a number of *R* packages have been developed that define new spatial data types that can be used for this type of specialzed operations. The most important packages that define such spatial data structures are `sp` and `raster`. These datatypes are discussed in the next chapters.

CHAPTER

THREE

3. VECTOR DATA

3.1 3.1 Introduction

Package `sp` is the central package supporting spatial data analysis in *R*. `sp` defines a set of *classes* to represent spatial data. A class defines a particular data type. The `data.frame` is an example of a class. Any particular `data.frame` you create is an *object* (instantiation) of that 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. In fact, the `sp` package does not provide many functions to modify or analyse spatial data; but the classes it defines are used in more than 100 other *R* packages that provide specific functionality. See Hadley Wickham's Advanced R or John Chambers' Software for data analysis for a detailed discussion of the use of classes in *R*).

Package `sp` introduces a number of classes with names that start with `Spatial`. For vector data, the basic types are the `SpatialPoints`, `SpatialLines`, and `SpatialPolygons`. These classes only represent geometries. To also store attributes, classes are available with these names plus `DataFrame`, for example, `SpatialPolygonsDataFrame` and `SpatialPointsDataFrame`. When referring to any object with a name that starts with `Spatial`, it is common to write `Spatial*`. When referring to a `SpatialPolygons` or `SpatialPolygonsDataFrame` object it is common to write `SpatialPolygons*`. The `Spatial` classes (and their use) are described in detail by Bivand, Pebesma and Gómez-Rubio.

It is possible to create `Spatial*` objects from scratch with *R* code. That can be very useful to create small self contained example to illustrate something, for example to ask a question about how to do a particular operation without needing to give access to the real data you are using (which is always cumbersom). But in real life you will read these from a file or database, for example from a shapefile see Chapter 5.

To get started, let's make some Spatial objects from scratch anyway, using the same data as were used in the previous chapter.

3.2 3.2 SpatialPoints

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

Now create a `SpatialPoints` object

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

Let's check what kind of object `pts` is.

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

And what is inside of it

```
showDefault(pts)
## An object of class "SpatialPoints"
## Slot "coords":
##       longitude latitude
##  [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
## longitude -120.8 -110.7
## latitude    35.7   45.3
##
## Slot "proj4string":
## CRS arguments: NA
```

So we see that the object has the coordinates we supplied, but also a `bbox`. This is a 'bounding box', or the 'spatial extent' that was computed from the coordinates. There is also a "proj4string". This stores the coordinate reference system ("crs", discussed in more detail later). 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)
```

I load to `raster` package to improve how Spatial objects are printed.

```
library(raster)
pts
## class       : SpatialPoints
## features    : 10
## extent      : -120.8, -110.7, 35.7, 45.3  (xmin, xmax, ymin, ymax)
## coord. ref. : +proj=longlat +datum=WGS84 +ellps=WGS84 +towgs84=0,0,0
```

We can use the `SpatialPoints` object to create a `SpatialPointsDataFrame` object. First we need a `data.frame` with the same number of rows as there are geometries.

```
df <- data.frame(ID=1:nrow(lonlat), precip=(latitude-30)^3)
```

Combine the `SpatialPoints` with the `data.frame`.

```
ptsdf <- SpatialPointsDataFrame(pts, data=df)
ptsdf
## class       : SpatialPointsDataFrame
## features    : 10
## extent      : -120.8, -110.7, 35.7, 45.3  (xmin, xmax, ymin, ymax)
```

```
## coord. ref. : +proj=longlat +datum=WGS84 +ellps=WGS84 +towgs84=0,0,0
## variables   : 2
## names       : ID,   precip
## min values  :  1,  185.193
## max values  : 10, 3581.577
```

To see what is inside:

```
str(ptsdf)
## Formal class 'SpatialPointsDataFrame' [package "sp"] with 5 slots
##   ..@ data       :'data.frame':  10 obs. of  2 variables:
##   .. ..$ ID    : int [1:10] 1 2 3 4 5 6 7 8 9 10
##   .. ..$ precip: num [1:10] 3582 2000 705 1772 185 ...
##   ..@ coords.nrs : num(0)
##   ..@ coords     : num [1:10, 1:2] -117 -120 -117 -114 -116 ...
##   .. ..- attr(*, "dimnames")=List of 2
##   .. .. ..$ : NULL
##   .. .. ..$ : chr [1:2] "longitude" "latitude"
##   ..@ bbox       : num [1:2, 1:2] -120.8 35.7 -110.7 45.3
##   .. ..- attr(*, "dimnames")=List of 2
##   .. .. ..$ : chr [1:2] "longitude" "latitude"
##   .. .. ..$ : chr [1:2] "min" "max"
##   ..@ proj4string:Formal class 'CRS' [package "sp"] with 1 slot
##   .. .. ..@ projargs: chr "+proj=longlat +datum=WGS84 +ellps=WGS84 +towgs84=0,0,0"
```

Or

```
showDefault(ptsdf)
## An object of class "SpatialPointsDataFrame"
## Slot "data":
##    ID   precip
## 1   1 3581.577
## 2   2 2000.376
## 3   3  704.969
## 4   4 1771.561
## 5   5  185.193
## 6   6  704.969
## 7   7  238.328
## 8   8  729.000
## 9   9 1560.896
## 10 10  328.509
##
## Slot "coords.nrs":
## numeric(0)
##
## Slot "coords":
##       longitude latitude
##  [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
## longitude -120.8 -110.7
## latitude    35.7   45.3
##
## Slot "proj4string":
## CRS arguments:
##  +proj=longlat +datum=WGS84 +ellps=WGS84 +towgs84=0,0,0
```

3.3 3.3 SpatialLines and SpatialPolygons

Making a `SpatialPoints` object was easy. Making a `SpatialLines` and `SpatialPolygons` object is a bit harder, but stil relatively straightforward with the `spLines` and `spPolygons` functions (from the `raster` package).

```
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)
## coord. ref. : +proj=longlat +datum=WGS84 +ellps=WGS84 +towgs84=0,0,0
```

```
pols <- spPolygons(lonlat, crs=crdref)
pols
## class       : SpatialPolygons
## features    : 1
## extent      : -117.7, -111.9, 37.6, 42.9  (xmin, xmax, ymin, ymax)
## coord. ref. : +proj=longlat +datum=WGS84 +ellps=WGS84 +towgs84=0,0,0
```

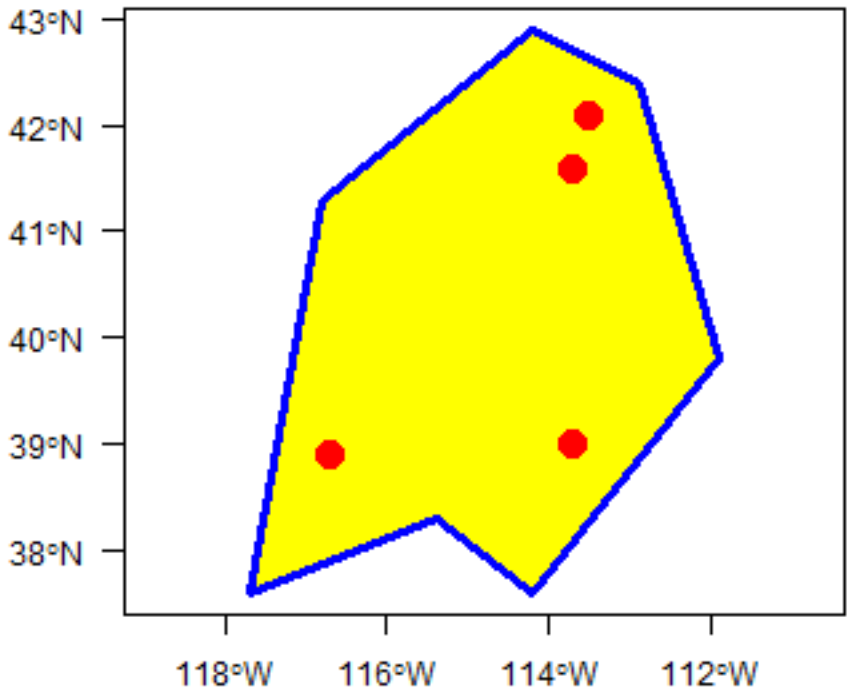
The structure of the SpatialPolygons class is somewhat complex as it needs to accommodate the possibility of multiple polygons, each consisting of multiple sub-polygons, some of which may be "holes".

```
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 +ellps=WGS84 +towgs84=0,0,0"
```

Fortunately, you do not need to understand how these structures are organized. The main take home message is that they store geometries (coordinates), the name of the coordinate reference system, and attributes.

We can make use generic *R* function plot to make a map.

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



We'll make more fancy maps later.

CHAPTER

FOUR

4. RASTER DATA

4.1 4.1 Introduction

The `sp` package supports raster (gridded) data with with the `SpatialGridDataFrame` and `SpatialPixelsDataFrame` classes. However, we will focus on classes from the `raster` package for raster data. The raster package is built around a number of classes of which the `RasterLayer`, `RasterBrick`, and `RasterStack` classes are the most important. When discussing methods that can operate on all three of these objects, they are referred to 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. For example, there are functions to read a chunk of raster values from a file or to convert cell numbers to coordinates and back. The package also implements raster algebra and many other functions for raster data manipulation.

4.2 4.2 RasterLayer

A `RasterLayer` object represents single-layer (variable) raster data. A `RasterLayer` object always stores a number of fundamental parameters that describe it. These include the number of columns and rows, the spatial extent, and the Coordinate Reference System. In addition, a `RasterLayer` can store information about the file in which the raster cell values are stored (if there is such a file). A `RasterLayer` can also hold the raster cell values in memory.

Here I create a `RasterLayer` from scratch. But note that in most cases where real data is analyzed, these objects are created from a file.

```
library(raster)
r <- raster(ncol=10, nrow=10, xmx=-80, xmn=-150, ymn=20, ymx=60)
r
## class       : RasterLayer
## dimensions  : 10, 10, 100  (nrow, ncol, ncell)
## resolution  : 7, 4  (x, y)
## extent      : -150, -80, 20, 60  (xmin, xmax, ymin, ymax)
## coord. ref. : +proj=longlat +datum=WGS84 +ellps=WGS84 +towgs84=0,0,0
```

Object `r` only has the skeleton of a raster data set. That is, it knows about its location, resolution, etc., but there are no values associated with it. Let's assign some values. In this case I assign a vector of random numbers with a length that is equal to the number of cells of the RasterLayer.

```
values(r) <- runif(ncell(r))
r
## class       : RasterLayer
## dimensions  : 10, 10, 100  (nrow, ncol, ncell)
```

```
## resolution  : 7, 4  (x, y)
## extent      : -150, -80, 20, 60  (xmin, xmax, ymin, ymax)
## coord. ref. : +proj=longlat +datum=WGS84 +ellps=WGS84 +towgs84=0,0,0
## data source : in memory
## names       : layer
## values      : 0.01339033, 0.9926841  (min, max)
```

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

```
values(r) <- 1:ncell(r)
r
## class       : RasterLayer
## dimensions  : 10, 10, 100  (nrow, ncol, ncell)
## resolution  : 7, 4  (x, y)
## extent      : -150, -80, 20, 60  (xmin, xmax, ymin, ymax)
## coord. ref. : +proj=longlat +datum=WGS84 +ellps=WGS84 +towgs84=0,0,0
## data source : in memory
## names       : layer
## values      : 1, 100  (min, max)
```

We can plot this object.

```
plot(r)

# 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=crdref)
## Error in spPolygons(lonlat, crs = crdref): object 'crdref' not found

plot(pols, border='blue', lwd=2, add=TRUE)
## Error in plot(pols, border = "blue", lwd = 2, add = TRUE): object 'pols' not found
points(lonlat, col='red', pch=20, cex=3)
```

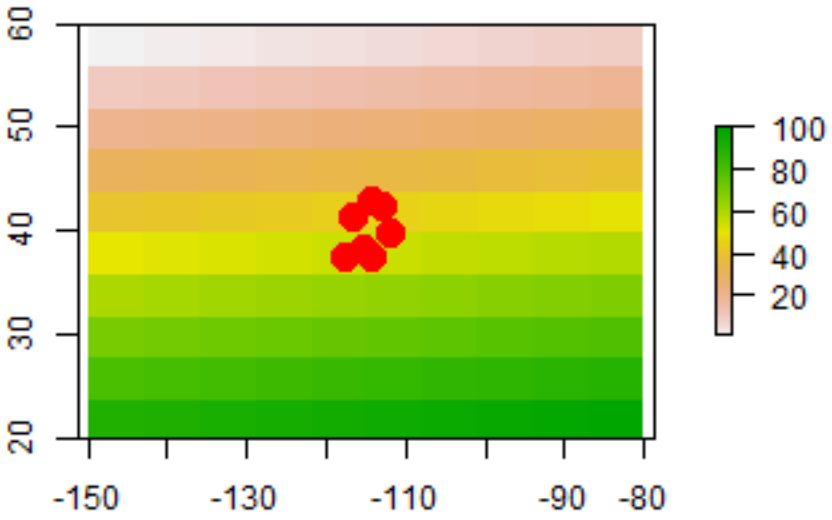
4.3 4.3 RasterStack and RasterBrick

It is quite common to analyze raster data using single-layer objects. However, in many cases multi-variable raster data sets are used. The `raster` package has two classes for multi-layer data the `RasterStack` and the `RasterBrick`. The principal difference between these two classes is that a `RasterBrick` can only be linked to a single (multi-layer) file. In contrast, a `RasterStack` can be formed from separate files and/or from a few layers ('bands') from a single file.

In fact, a `RasterStack` is a collection of `RasterLayer` objects with the same spatial extent and resolution. In essence it is a list of `RasterLayer` objects. A `RasterStack` can easily be formed form a collection of files in different locations and these can be mixed with `RasterLayer` objects that only exist in the RAM memory (not on disk).

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. A typical example of such a file would be a multi-band satellite image or the output of a global climate model (with e.g., a time series of temperature values for each day of the year for each raster cell). Methods that operate on `RasterStack` and `RasterBrick` objects typically return a `RasterBrick` object.

Thus, the main difference is that a `RasterStack` is loose collection of `RasterLayer` objects that can refer to different files (but must all have the same extent and resolution), whereas a `RasterBrick` can only point to a single file.

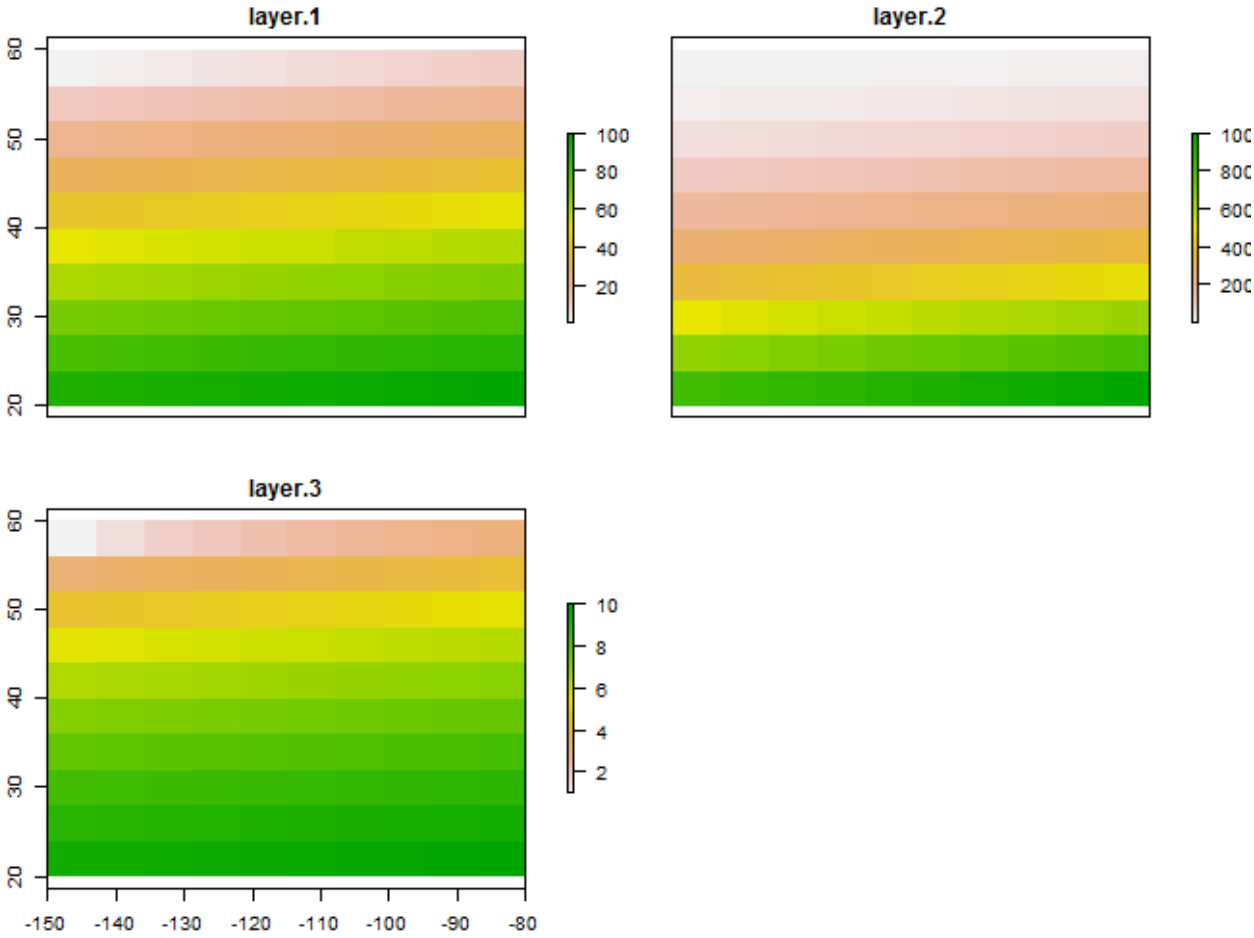


Here is an example how you can make a RasterStack from multiple layers.

```
r2 <- r * r
r3  <- sqrt(r)
s <- stack(r, r2, r3)
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)
## coord. ref. : +proj=longlat +datum=WGS84 +ellps=WGS84 +towgs84=0,0,0
## 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.

```
b <- brick(s)
b
## 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)
## coord. ref. : +proj=longlat +datum=WGS84 +ellps=WGS84 +towgs84=0,0,0
## data source : in memory
## names       : layer.1, layer.2, layer.3
## min values  :       1,       1,       1
## max values  :     100,   10000,      10
```


layer.1
60
50
40
30
20
100
80
60
40
20
layer.2
100
800
600
400
200
layer.3
60
50
40
30
20
10
8
6
4
2
-150
-140
-130
-120
-110
-100
-90
-80

CHAPTER

FIVE

5. READING AND WRITING SPATIAL DATA

5.1 5.1 Introduction

Reading and writing spatial is complicated by the fact that there are many different file formats. However, there are a few formats that are most common that we discuss here.

5.2 Vector files

The `shapefile` is the most commonly used file format for vector data. It is trivial to read and write such files. Here we use a shapefile that comes with the `raster` package.

5.2.1 Reading

We use the `system.file` function to 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 should not use the `system.file` function for your own files. It only serves for creating examples with data that ships with *R*.

```
library(raster)
filename <- system.file("external/lux.shp", package="raster")
filename
## [1] "C:/soft/R/R-3.3.1/library/raster/external/lux.shp"
```

Now we have the filename we need we use the `shapefile` function. This function comes with the `raster` package. For it to work you must also have the `rgdal` package.

```
s <- shapefile(filename)
s
## class       : SpatialPolygonsDataFrame
## features    : 12
## extent      : 5.74414, 6.528252, 49.44781, 50.18162  (xmin, xmax, ymin, ymax)
## coord. ref. : +proj=longlat +datum=WGS84 +no_defs +ellps=WGS84 +towgs84=0,0,0
## 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
```

The `shapefile` function returns `Spatial*DataFrame` objects. In this case a `SpatialPolygonsDataFrame`. It is important to recognise the difference between this type of *R* object (SpatialPolygonsDataFrame), and the file (shapefile) that was used to create it.

For other formats, you can use `readOGR` function in package `rgdal`.

5.2.2 Writing

You can also write shapefiles using the `shapefile` method. In stead of a filename, you need to provide a vector type `Spatial*` object as first argument and a new filename as a second argument. You can add argument `overwrite=TRUE` if you want to overwrite an existing file.

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

For other formats, you can use `writeOGR` function in package `rgdal`.

5.3 5.2 Raster files

The raster package can read and write several raster file formats.

5.3.1 Reading

Again we need to get a filename for an example file.

```
f <- system.file("external/rlogo.grd", package="raster")
f
## [1] "C:/soft/R/R-3.3.1/library/raster/external/rlogo.grd"
```

Now we can do

```
r1 <- raster(f)
r1
## 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)
## coord. ref. : +proj=merc +datum=WGS84 +ellps=WGS84 +towgs84=0,0,0
## data source : C:\soft\R\R-3.3.1\library\raster\external\rlogo.grd
## names       : red
## values      : 0, 255  (min, max)
```

Note that `r1` is a RasterLayer of the first "band" (layer) in the file (out of three bands (layers)). We can request another layer.

```
r2 <- raster(f, band=2)
r2
## 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)
## coord. ref. : +proj=merc +datum=WGS84 +ellps=WGS84 +towgs84=0,0,0
## data source : C:\soft\R\R-3.3.1\library\raster\external\rlogo.grd
## names       : green
## values      : 0, 255  (min, max)
```

More commonly, you would want all layers in a single object. For that you can 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)
## coord. ref. : +proj=merc +datum=WGS84 +ellps=WGS84 +towgs84=0,0,0
## data source : C:\soft\R\R-3.3.1\library\raster\external\rlogo.grd
## names       : red, green, blue
## min values  :   0,     0,    0
## max values  : 255,   255,  255
```

Or you can use `stack`, but that is less efficient in most cases.

```
s <- stack(f)
s
## class       : RasterStack
## dimensions  : 77, 101, 7777, 3  (nrow, ncol, ncell, nlayers)
## resolution  : 1, 1  (x, y)
## extent      : 0, 101, 0, 77  (xmin, xmax, ymin, ymax)
## coord. ref. : +proj=merc +datum=WGS84 +ellps=WGS84 +towgs84=0,0,0
## names       : red, green, blue
## min values  :   0,     0,    0
## max values  : 255,   255,  255
```

The same approach holds for other raster file formats, including GeoTiff, NetCDF, Imagine, and ESRI Grid formats.

5.3.2 Writing

Use `writeRaster` to write raster data. You must provide a Raster* object and a filename. The file format will be guessed from the filename extension (if that does not work you can provide an argument like `format=GTIFF`). Note the argument `overwrite=TRUE` and see `?writeRaster` for more arguments, such as `datatype=` to set the datatype (e.g., integer, float).

```
x <- writeRaster(s, 'output.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)
## coord. ref. : +proj=merc +lon_0=0 +k=1 +x_0=0 +y_0=0 +datum=WGS84 +units=m +no_defs +ellps=WGS84 +
## data source : C:\bitbucket\rweb\source\spatial\_R\output.tif
## names       : output.1, output.2, output.3
## min values  :        0,        0,        0
## max values  :      255,      255,      255
```

CHAPTER

SIX

6. COORDINATE REFERENCE SYSTEMS

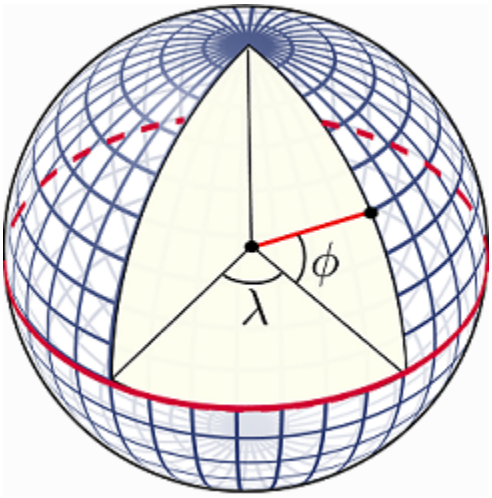
6.1 6.1 Introduction

A very important aspect of spatial data is the coordinate reference system (CRS) that is used. For example, a location of (140, 12) is not meaningful if you do know where the origin is and if the x-coordinate is 140 meters, kilometers, or perhaps degrees away from it (in the x direction).

6.2 6.2 Coordinate Reference Systems

6.2.1 Angular coordinates

The earth has an irregular spheroid-like shape. The natural coordinate reference system for geographic data is longitude/latitude. This is an *angular* system. The latitude (phi) of a point is the angle between the equatorial plane and the line that passes through a point and the center of the Earth. Longitude (lambda) is the angle from a reference meridian (lines of constant longitude) to a meridian that passes through the point.



Obviously we cannot actually measure these angles. But we can estimate them. To do so, you need a model of the shape of the earth. Such a model is called a 'datum'. The simplest datums are a spheroid (a sphere that is 'flattened' at the poles and bulges at the equator). More complex datums allow for more variation in the earth's shape. The most commonly used datum is called WGS84 (World Geodesic System 1984). This is very similar to NAD83 (The North American Datum of 1983). Other, local datums exist to more precisely record locations for a single country or region.

So the basic way to record a location is a coordinate pair in degrees and a reference datum. (Sometimes people say that their coordinates are "in WGS84". That is meaningless; but they typically mean to say that they are longitude/latitude relative to the WGS84 datum).

6.2.2 Projections

A major question in spatial analysis and cartography is how to transform this three dimensional angular system to a two dimensional planar (sometimes called "Cartesian") system. A planar system is easier to use for certain calculations and required to make maps (unless you have a 3-d printer). The different types of planar coordinate reference systems are referred to as 'projections'. Examples are 'Mercator', 'UTM', 'Robinson', 'Lambert', 'Sinusoidal' 'Robinson' and 'Albers'.

There is not one best projection. Some projections can be used for a map of the whole world; other projections are appropriate for small areas only. One of the most important characteristics of a map projection is whether it is "equal area" (the scale of the map is constant) or "conformal" (the shapes of the geographic features are as they are seen on a globe). No two dimensional map projection can be both conformal and equal-area (but they can be approximately both for smaller areas, e.g. UTM, or Lambert Equal Area for a larger area), and some are neither.

6.2.3 Notation

A planar CRS is defined by a projection, datum, and a set of parameters. The parameters determine things like where the center of the map is. The number of parameters depends on the projection. It is therefore not trivial to document a projection used, and several systems exist. In *R* we use the [PROJ.4[(ftp://ftp.remotesensing.org/proj/OF90-284.pdf) notation. PROJ.4 is the name of an open source software library that is commonly used for CRS transformation.

Here is a list of commonly used projections and their parameters in PROJ4 notation. You can find many more of these on spatialreference.org

Most commonly used CRSs have been assigned a "EPSG code" (EPSG stands for European Petroleum Survey Group). This is a unique ID that can be a simple way to identify a CRS. For example `EPSG:27561` is equivalent to `+proj=lcc +lat_1=49.5 +lat_0=49.5 +lon_0=0 +k_0=0.999877341 +x_0=6 +y_0=2 +a=6378249.2 +b=6356515 +towgs84=-168,-60,320,0,0,0,0 +pm=paris +units=m +no_defs`. However `EPSG:27561` is opaque and should not be used outside of databases. In R use the PROJ.4 notation, as that can be readily interpreted without relying on software.

Below is an illustration of how to find a particular projection you may need (in this example, a list of projections for France).

```
library(rgdal)
epsg <- make_EPSG()
i <- grep("France", epsg$note, ignore.case=TRUE)
# first three
epsg[i[1:3], ]
##       code                                        note
## 661   2192 # ED50 / France EuroLambert (deprecated)
## 3987 27561      # NTF (Paris) / Lambert Nord France
## 3988 27562    # NTF (Paris) / Lambert Centre France
##
## 661                                  +proj=lcc +lat_1=46.8 +lat_0=46.8 +lon_0=2.337229166666667 +
## 3987 +proj=lcc +lat_1=49.50000000000001 +lat_0=49.50000000000001 +lon_0=0 +k_0=0.999877341 +x_0=6
## 3988                             +proj=lcc +lat_1=46.8 +lat_0=46.8 +lon_0=0 +k_0=0.99987742 +x_0=6
```

Now let's look at an example with a spatial data set in *R*.

```
library(raster)
library(rgdal)
```

```
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)
## coord. ref. : +proj=longlat +datum=WGS84 +no_defs +ellps=WGS84 +towgs84=0,0,0
## 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
```

We can inspect the coordinate reference system like this.

```
crs(p)
## CRS arguments:
##  +proj=longlat +datum=WGS84 +no_defs +ellps=WGS84 +towgs84=0,0,0
```

6.3 6.3 Assigning a CRS

Sometimes we have data without a CRS. This can be because the file used was incomplete, or perhaps because we created the data ourselves with R code. In that case we can assign the CRS **if we know what it should be**. Here it first remove the CRS of `pp` and then I set it again.

```
pp <- p
crs(pp) <- NA
crs(pp)
## CRS arguments: NA
crs(pp) <- CRS("+proj=longlat +datum=WGS84")
crs(pp)
## CRS arguments:
##  +proj=longlat +datum=WGS84 +ellps=WGS84 +towgs84=0,0,0
```

Note that you should **not** use this approach to change the CRS of a data set from what it **is** to what you **want it to be**_. Assigning a CRS is like labeling something. You need to provide the label that corresponds to the item. Not to what you would like it to be. For example if you label a bicycle, you can write "bicycle". Perhaps you would prefer a car, and you can label your bicycle as "car" but that would not do you any good. It is still a bicycle. You can try to transform your bicycle into a car. That would not be easy. Transforming spatial data is easier.

6.4 6.4 Transforming vector data

We can transform these data to a new data set with another CRS using the `spTransform` function from the `rgdal` package.

Here we use the Robinson projection. First we need to find the correction notation.

```
newcrs <- CRS("+proj=robin +datum=WGS84")
```

Now use it

```
rob <- spTransform(p, newcrs)
rob
## class       : SpatialPolygonsDataFrame
## features    : 12
```

```
## extent      : 471320.7, 536010.5, 5269709, 5345677  (xmin, xmax, ymin, ymax)
## coord. ref. : +proj=robin +datum=WGS84 +ellps=WGS84 +towgs84=0,0,0
## 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
```

After the transformation, the units of the geometry are no longer in degrees, but in meters away from (longitude=0, latitude=0). The spatial extent of the data is also in these units.

We can backtransform to longitude/latitude:

```
p2 <- spTransform(rob, CRS("+proj=longlat +datum=WGS84"))
```

6.5 6.5 Transforming raster data

Vector data can be transformed from lon/lat coordinates to planar and back with loss of precision. This is not the case with raster data. A raster consists of rectangular cells of the same size (in terms of the units of the CRS; their actual size may vary). It is not possible to transform cell by cell. Rather estimates for the values of new cells must be made based on the values in the old cells. If the values are class data, the 'nearest neighbor' is commonly used. Otherwise some sort of interpolation (e.g. 'bilinear').

Because projection of rasters affects the cells values, in most cases you will want to avoid projecting raster data and rather project vector data. But when you do project raster data, you want to assure that you project to exactly the raster definition you need (so that it lines up with other raster data you are using).

```
r <- raster(xmn=-110, xmx=-90, ymn=40, ymx=60, ncols=40, nrows=40)
r <- setValues(r, 1:ncell(r))
r
## class       : RasterLayer
## dimensions  : 40, 40, 1600  (nrow, ncol, ncell)
## resolution  : 0.5, 0.5  (x, y)
## extent      : -110, -90, 40, 60  (xmin, xmax, ymin, ymax)
## coord. ref. : +proj=longlat +datum=WGS84 +ellps=WGS84 +towgs84=0,0,0
## data source : in memory
## names       : layer
## values      : 1, 1600  (min, max)
plot(r)
```

Here is a new PROJ4 projection description.

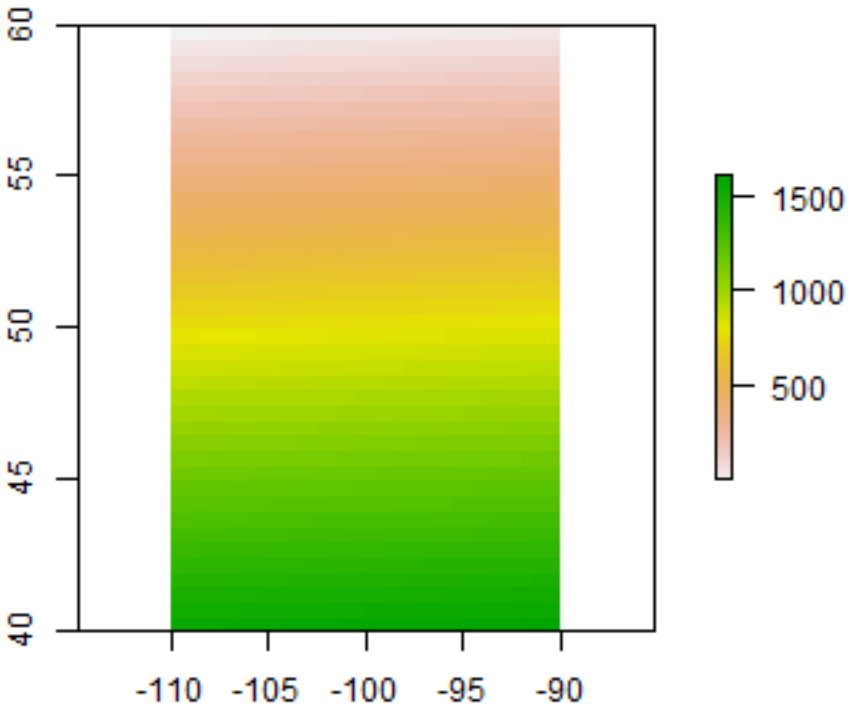
```
newproj <- "+proj=lcc +lat_1=48 +lat_2=33 +lon_0=-100 +ellps=WGS84"
```

Simplest approach

```
pr1 <- projectRaster(r, crs=newproj)
crs(pr1)
## CRS arguments:
##  +proj=lcc +lat_1=48 +lat_2=33 +lon_0=-100 +ellps=WGS84
```

Alternatively, you can also set the resolution.

```
pr2 <- projectRaster(r, crs=newproj, res=20000)
pr2
## class       : RasterLayer
## dimensions  : 124, 94, 11656  (nrow, ncol, ncell)
## resolution  : 20000, 20000  (x, y)
```


60
55
50
45
40
-110
-105
-100
-95
-90
1500
1000
500

```
## extent      : -944881.5, 935118.5, 4664378, 7144378  (xmin, xmax, ymin, ymax)
## coord. ref. : +proj=lcc +lat_1=48 +lat_2=33 +lon_0=-100 +ellps=WGS84
## data source : in memory
## names       : layer
## values      : -16.22972, 1616.249  (min, max)
```

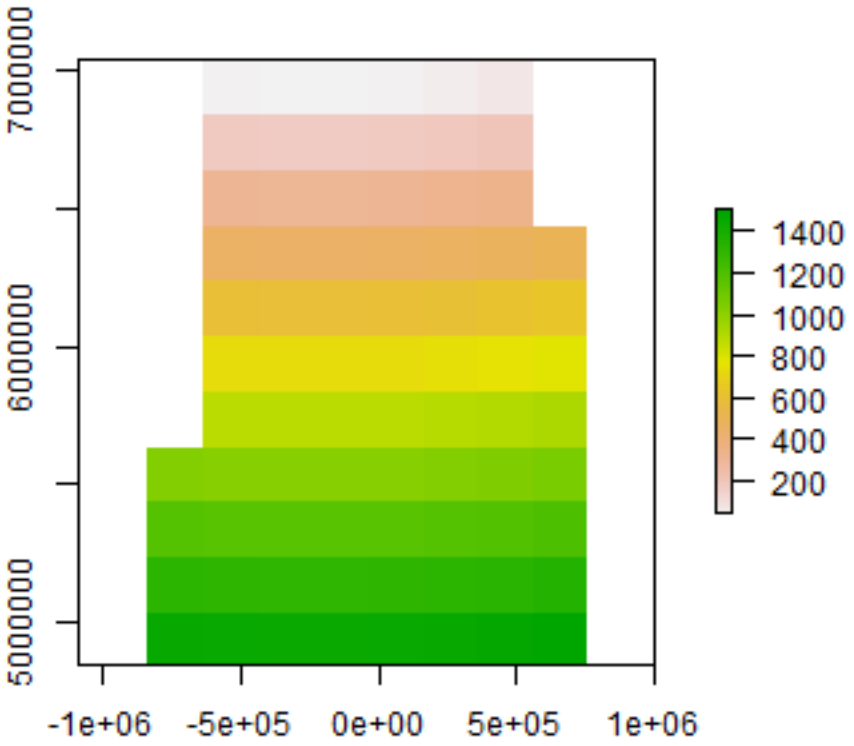
But to have more control, provide an existing Raster object. That is generally the best way to project raster. By providing an existing Raster object, such that your newly projected data perfectly aligns with it. In this example we do not have an existing Raster object, so we create one using `projectExtent`.

```
pr3 <- projectExtent(r, newproj)
# Set the cell size
res(pr3) <- 200000
```

Now project, and note the change in the coordinates.

```
pr3 <- projectRaster(r, pr3)
pr3
## class       : RasterLayer
## dimensions  : 11, 8, 88  (nrow, ncol, ncell)
## resolution  : 2e+05, 2e+05  (x, y)
## extent      : -844881.5, 755118.5, 4844378, 7044378  (xmin, xmax, ymin, ymax)
## coord. ref. : +proj=lcc +lat_1=48 +lat_2=33 +lon_0=-100 +ellps=WGS84
## data source : in memory
## names       : layer
## values      : 41.84528, 1503.516  (min, max)
plot(pr3)
```

For raster based analysis it is often important to use equal area projections, particularly when large areas are analyzed. This will assure that the grid cells are all of same size, and therefore comparable to each other.


7000000
6000000
5000000
-1e+06
-5e+05
0e+00
5e+05
1e+06
1400
1200
1000
800
600
400
200

CHAPTER

SEVEN

7. VECTOR DATA MANIPULATION

Example SpatialPolygons

```
f <- system.file("external/lux.shp", package="raster")
library(raster)
p <- shapefile(f)
p
## class       : SpatialPolygonsDataFrame
## features    : 12
## extent      : 5.74414, 6.528252, 49.44781, 50.18162  (xmin, xmax, ymin, ymax)
## coord. ref. : +proj=longlat +datum=WGS84 +no_defs +ellps=WGS84 +towgs84=0,0,0
## 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)
```

7.1 7.1 Basics

Basic operations are pretty much like working with a data.frame.

7.1.1 7.1.1 Geometry and attributes

To extracting the attributes (data.frame) from a Spatial object, use:

```
d <- data.frame(p)
head(d)
##   ID_1       NAME_1 ID_2     NAME_2 AREA
## 0    1     Diekirch    1   Clervaux  312
## 1    1     Diekirch    2   Diekirch  218
## 2    1     Diekirch    3    Redange  259
## 3    1     Diekirch    4    Vianden   76
## 4    1     Diekirch    5      Wiltz  263
## 5    2 Grevenmacher    6 Echternach  188
```

Extracting geometry (rarely needed).

```
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
```

7.1.2 7.1.2 Variables

Extracting a variable.

```
p$NAME_2
##  [1] "Clervaux"         "Diekirch"         "Redange"
##  [4] "Vianden"          "Wiltz"            "Echternach"
##  [7] "Remich"           "Grevenmacher"     "Capellen"
## [10] "Esch-sur-Alzette" "Luxembourg"       "Mersch"
```

Sub-setting by variable. Note how this is different from the above example. Above a vector of values is returned. 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)
## coord. ref. : +proj=longlat +datum=WGS84 +no_defs +ellps=WGS84 +towgs84=0,0,0
## variables   : 1
## names       :   NAME_2
## min values  : Capellen
## max values  :    Wiltz
```

Adding a new variable.

```
set.seed(0)
p$new <- sample(letters, length(p))
p
## class       : SpatialPolygonsDataFrame
## features    : 12
## extent      : 5.74414, 6.528252, 49.44781, 50.18162  (xmin, xmax, ymin, ymax)
## coord. ref. : +proj=longlat +datum=WGS84 +no_defs +ellps=WGS84 +towgs84=0,0,0
## variables   : 6
## names       : ID_1,     NAME_1, ID_2,   NAME_2, AREA, new
## min values  :    1,   Diekirch,    1, Capellen,   76,   a
## max values  :    3, Luxembourg,   12,    Wiltz,  312,   x
```

Assigning a new value to an existing variable.

```
p$new <- sample(LETTERS, length(p))
p
## class       : SpatialPolygonsDataFrame
## features    : 12
## extent      : 5.74414, 6.528252, 49.44781, 50.18162  (xmin, xmax, ymin, ymax)
## coord. ref. : +proj=longlat +datum=WGS84 +no_defs +ellps=WGS84 +towgs84=0,0,0
## variables   : 6
## names       : ID_1,     NAME_1, ID_2,   NAME_2, AREA, new
## min values  :    1,   Diekirch,    1, Capellen,   76,   D
## max values  :    3, Luxembourg,   12,    Wiltz,  312,   Y
```

To get rid of a variable.

```
p$new <- NULL
```

7.1.3 7.1.3 Merge

You can join a table (data.frame) with a Spatial* object with `merge`.

```
dfr <- data.frame(District=p$NAME_1, Canton=p$NAME_2, Value=round(runif(length(p), 100, 1000)))
dfr <- dfr[order(d$Canton), ]
## Error in order(d$Canton): argument 1 is not a vector
pm <- merge(p, dfr, by.x=c('NAME_1', 'NAME_2'), by.y=c('District', 'Canton'))
pm
## class       : SpatialPolygonsDataFrame
## features    : 12
## extent      : 5.74414, 6.528252, 49.44781, 50.18162  (xmin, xmax, ymin, ymax)
## coord. ref. : +proj=longlat +datum=WGS84 +no_defs +ellps=WGS84 +towgs84=0,0,0
## variables   : 6
## names       :     NAME_1,   NAME_2, ID_1, ID_2, AREA, Value
## min values  :   Diekirch, Capellen,    1,    1,   76,   112
## max values  : Luxembourg,    Wiltz,    3,   12,  312,   883
```

7.1.4 7.1.4 Records

Selecting rows (records).

```
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)
## coord. ref. : +proj=longlat +datum=WGS84 +no_defs +ellps=WGS84 +towgs84=0,0,0
## 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
```

It is also possible to interactively selecion and query records by clikcing on a plotted dataset. That is difficult to show here. See `?select` for interactively selecting spatial features and `?click` to identify attributes by clicking on a plot (map).

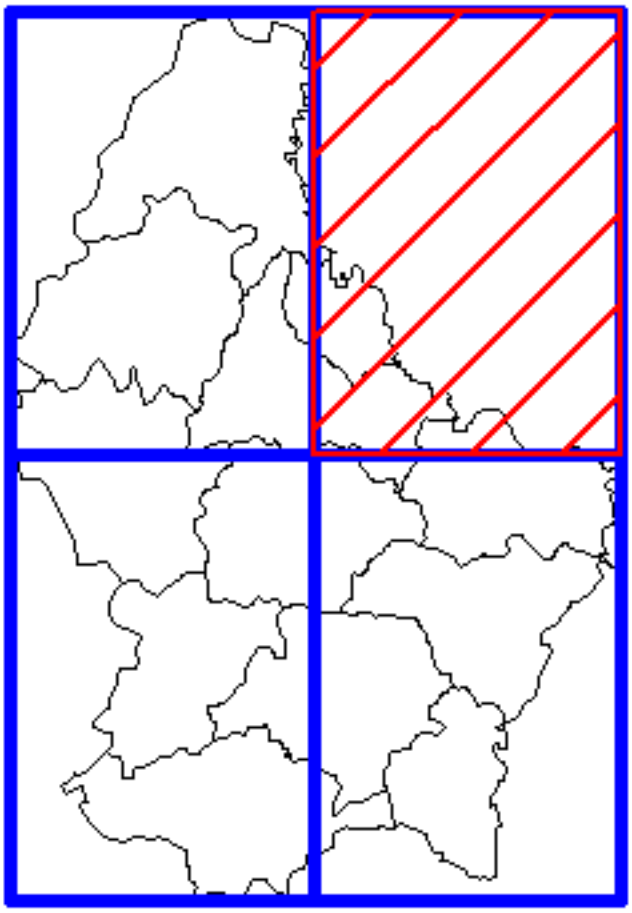
7.2 7.2 Append and aggregate

7.3 7.2.1 Append

More example data. Object `z`, consisting of four, and `z2` which is one of these four polygons.

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

```
## features    : 4
## extent      : 5.74414, 6.528252, 49.44781, 50.18162  (xmin, xmax, ymin, ymax)
## coord. ref. : +proj=longlat +datum=WGS84 +no_defs +ellps=WGS84 +towgs84=0,0,0
## variables   : 1
## names       : Zone
## min values  :    1
## max values  :    4
z2 <- z[2,]
plot(p)
plot(z, add=TRUE, border='blue', lwd=5)
plot(z2, add=TRUE, border='red', lwd=2, density=3, col='red')
```



To append Spatial* objects of the same (vector) type you can use `bind`

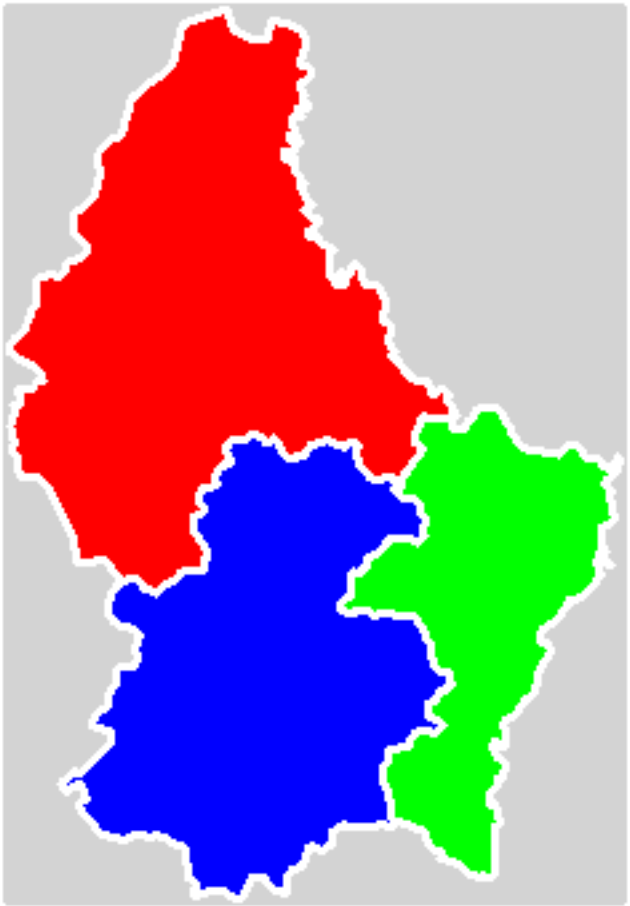
```
b <- bind(p, z)
head(b)
##   ID_1       NAME_1 ID_2     NAME_2 AREA Zone
## 1    1     Diekirch    1   Clervaux  312   NA
## 2    1     Diekirch    2   Diekirch  218   NA
## 3    1     Diekirch    3    Redange  259   NA
## 4    1     Diekirch    4    Vianden   76   NA
## 5    1     Diekirch    5      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
```

Note how `bind` allows you to append `Spatial*` objects with different attribute names.

7.4 7.2.2 Aggregate

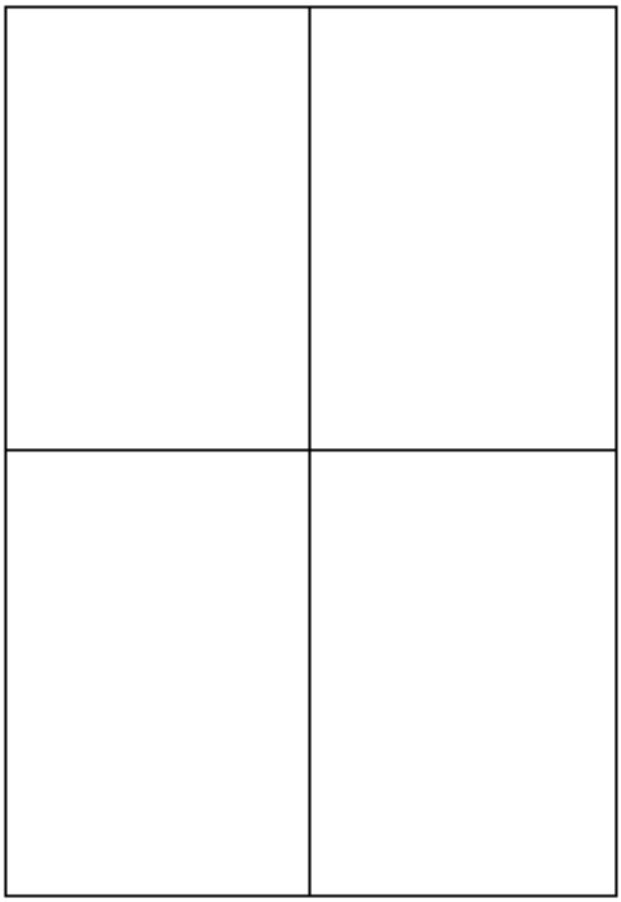
```
pa <- aggregate(p, by='NAME_1')
za <- aggregate(z)
plot(za, col='light gray', border='light gray', lwd=5)
plot(pa, add=TRUE, col=rainbow(3), lwd=3, border='white')
```



You can also aggregate by providing a second Spatial object (see `?sp::aggregate`)

Aggregate without dissolve

```
zag <- aggregate(z, dissolve=FALSE)
zag
## class       : SpatialPolygons
## features    : 1
## extent      : 5.74414, 6.528252, 49.44781, 50.18162  (xmin, xmax, ymin, ymax)
## coord. ref. : +proj=longlat +datum=WGS84 +no_defs +ellps=WGS84 +towgs84=0,0,0
plot(zag)
```



This is a structure that is similar to what you may get for an archipelago: multiple polygons represented as one entity (one row). Use `disaggregate` to split these up into their parts.

```
zd <- disaggregate(zag)
zd
## class       : SpatialPolygons
## features    : 4
## extent      : 5.74414, 6.528252, 49.44781, 50.18162  (xmin, xmax, ymin, ymax)
## coord. ref. : +proj=longlat +datum=WGS84 +no_defs +ellps=WGS84 +towgs84=0,0,0
```

7.5 7.3 Overlay

7.5.1 7.3.1 Erase

Erase a part of a SpatialPolygons object

```
e <- erase(p, z2)
```

This is equivalent to

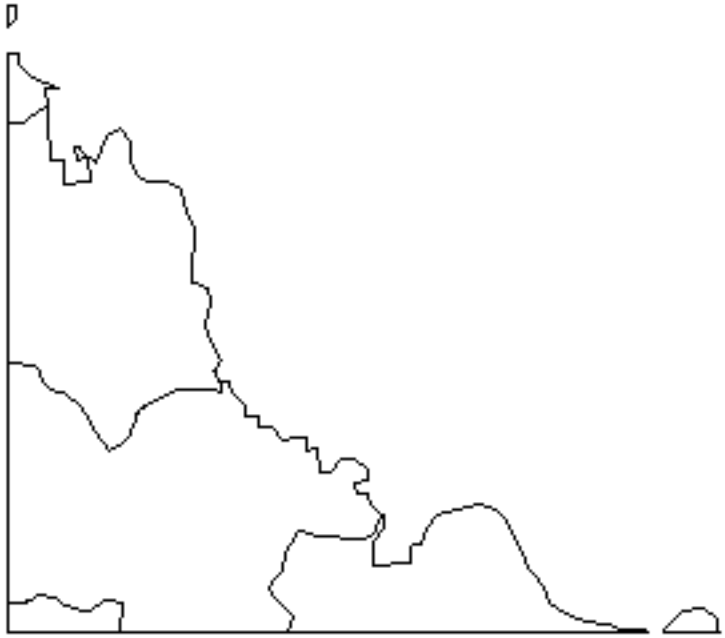
```
e <- p - z2
plot(e)
```

7.5.2 7.3.2 Intersect

Intersect SpatialPolygons

```
i <- intersect(p, z2)
plot(i)
```

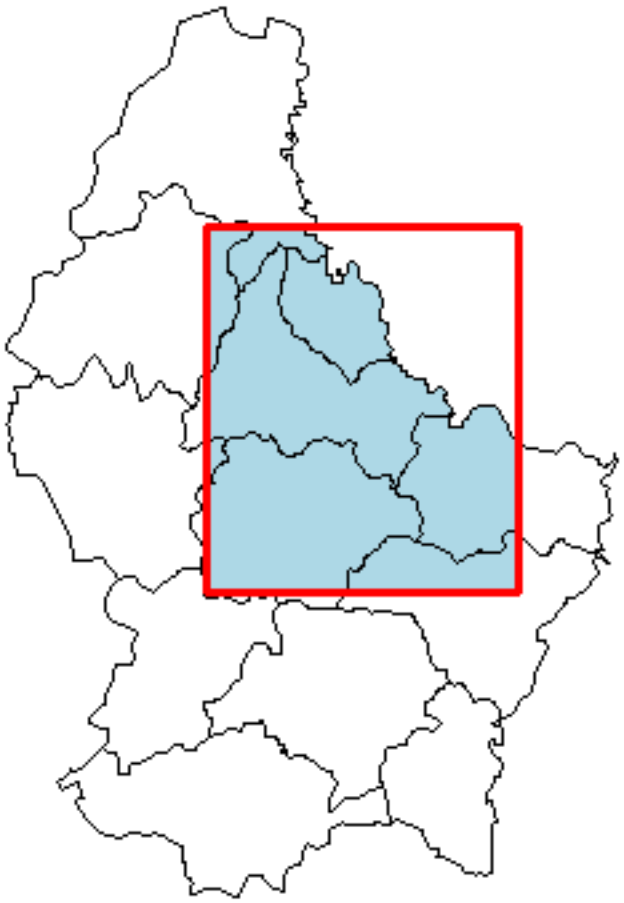
This is equivalent to



```
i <- p * z2
```

You can also intersect with an Extent (rectangle).

```
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')
```



7.5.3 7.3.3 Union

Get the union of two SpatialPolygon* objects.

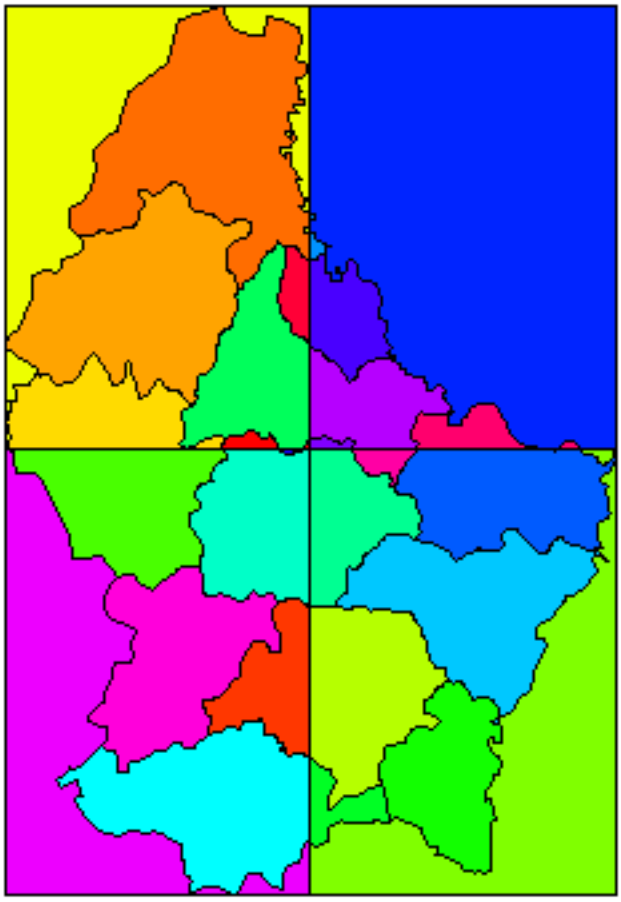
```
u <- union(p, z)
```

This is equivalent to

```
u <- p + z
```

Note that there are many more polygons now. One for each unique combination of polygons (and attributes in this case).

```
u
## class       : SpatialPolygonsDataFrame
## features    : 28
## extent      : 5.74414, 6.528252, 49.44781, 50.18162  (xmin, xmax, ymin, ymax)
## coord. ref. : +proj=longlat +datum=WGS84 +no_defs +ellps=WGS84 +towgs84=0,0,0
## variables   : 6
## names       : Zone, ID_1,     NAME_1, ID_2,    NAME_2, AREA
## min values  :    1,    1,   Diekirch,    1, Capellen,   76
## max values  :    4,    3, Luxembourg,   12,    Wiltz,  312
set.seed(5)
plot(u, col=sample(rainbow(length(u))))
```

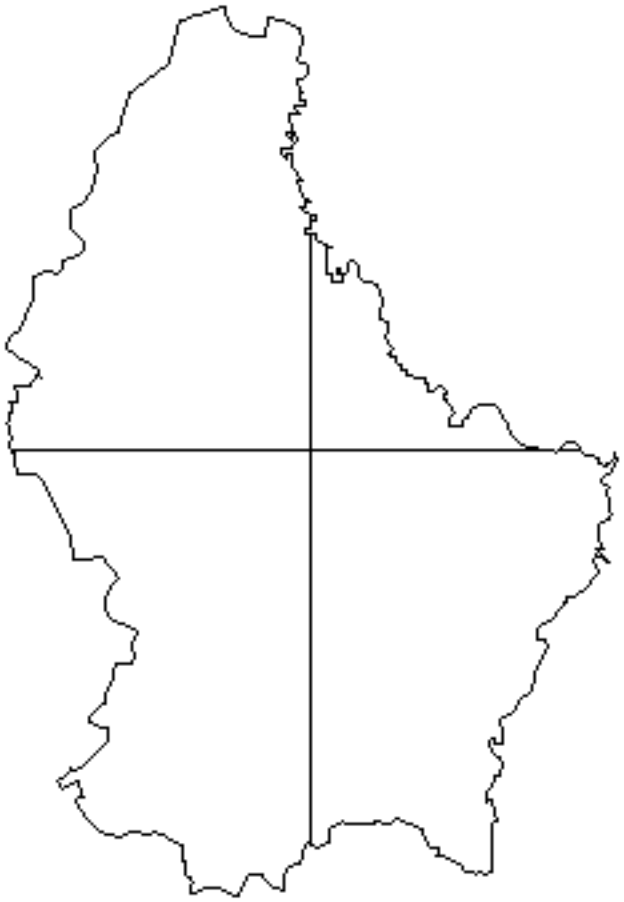


7.5.4 7.3.4 Cover

Cover is a combination of intersect and union

```
cov <- cover(p, z)
cov
## class       : SpatialPolygonsDataFrame
## features    : 6
## extent      : 5.74414, 6.528252, 49.44781, 50.18162  (xmin, xmax, ymin, ymax)
## coord. ref. : +proj=longlat +datum=WGS84 +no_defs +ellps=WGS84 +towgs84=0,0,0
## variables   : 6
```

```
## names       : ID_1,       NAME_1, ID_2,     NAME_2, AREA, Zone
## min values  :    1,     Diekirch,    1,   Clervaux,  188,    1
## max values  :    2, Grevenmacher,    6, Echternach,  312,    4
plot(cov)
```

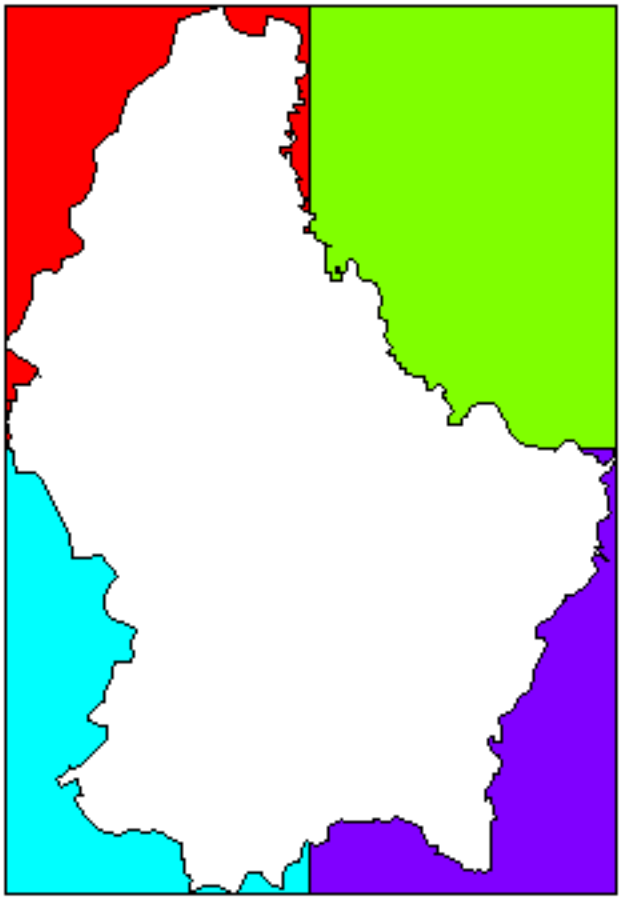


7.5.5 7.3.4 Difference

The symmetrical difference of two SpatialPolygons* objects

```
dif <- symdif(z,p)
plot(dif, col=rainbow(length(dif)))
```

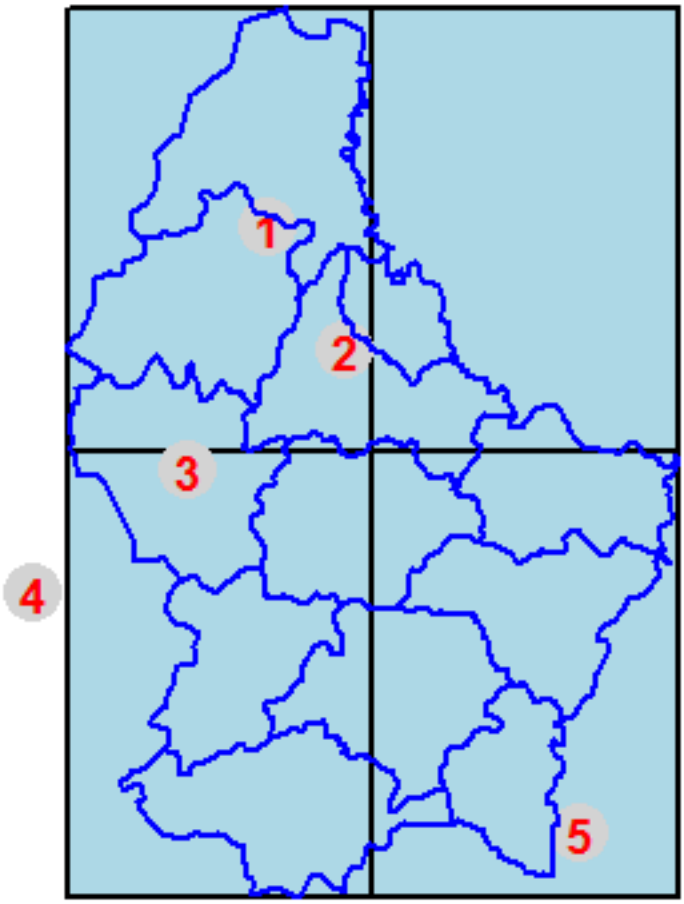
```
dif
## class       : SpatialPolygonsDataFrame
## features    : 4
## extent      : 5.74414, 6.528252, 49.44781, 50.18162  (xmin, xmax, ymin, ymax)
## coord. ref. : +proj=longlat +datum=WGS84 +no_defs +ellps=WGS84 +towgs84=0,0,0
## variables   : 1
## names       : Zone
## min values  :    1
## max values  :    4
```



7.6 7.4 Spatial queries

Query polygons with points.

```
pts <- matrix(c(6, 6.1, 5.9, 5.7, 6.4, 50, 49.9, 49.8, 49.7, 49.5), ncol=2)
spts <- SpatialPoints(pts, proj4string=crs(p))
plot(z, col='light blue', lwd=2)
points(spts, col='light gray', pch=20, cex=6)
text(spts, 1:nrow(pts), col='red', font=2, cex=1.5)
lines(p, col='blue', lwd=2)
```



Use `over` for queries between Spatial* objects

```
over(spts, p)
##   ID_1   NAME_1 ID_2   NAME_2 AREA
## 1    1 Diekirch    5    Wiltz  263
## 2    1 Diekirch    2 Diekirch  218
## 3    1 Diekirch    3  Redange  259
## 4   NA      <NA>   NA     <NA>   NA
## 5   NA      <NA>   NA     <NA>   NA
over(spts, z)
##   Zone
## 1    1
## 2    1
```

```
## 3     3
## 4    NA
## 5     4
```

`extract` is generally used for queries between Spatial* and Raster* objects, but it can also be used here.

```
extract(z, pts)
##   point.ID poly.ID Zone
## 1        1       1    1
## 2        2       1    1
## 3        3       3    3
## 4        4      NA   NA
## 5        5       4    4
```

CHAPTER

EIGHT

8. RASTER DATA MANIPULATION

8.1 Introduction

In this chapter general aspects of the design of the `raster` package are discussed, notably the structure of the main classes, and what they represent. The use of the package is illustrated in subsequent sections. `raster` has a large number of functions, not all of them are discussed here, and those that are discussed are mentioned only briefly. See the help files of the package for more information on individual functions and `help("raster-package")` for an index of functions by topic.

8.2 Creating Raster* objects

A `RasterLayer` can easily be created from scratch using the function `raster`. The default settings will create a global raster data structure with a longitude/latitude coordinate reference system and 1 by 1 degree cells. You can change these settings by providing additional arguments such as `xmn`, `nrow`, `ncol`, and/or `crs`, to the function. You can also change these parameters after creating the object. If you set the projection, this is only to properly define it, not to change it. To transform a `RasterLayer` to another coordinate reference system (projection) you can use the function 1projectRaster1.

Here is an example of creating and changing a `RasterLayer` object 'r' from scratch.

```
library(raster)
## Loading required package: sp
# RasterLayer with the default parameters
x <- raster()
x
## 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
```

With some other parameters

```
x <- raster(ncol=36, nrow=18, xmn=-1000, xmx=1000, ymn=-100, ymx=900)
```

These parameters can be changed. Resolution:

```
res(x)
## [1] 55.55556 55.55556
res(x) <- 100
res(x)
## [1] 100 100
```

Change the number of columns (this affects the resolution).

```
ncol(x)
## [1] 20
ncol(x) <- 18
ncol(x)
## [1] 18
res(x)
## [1] 111.1111 100.0000
```

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

```
projection(x) <- "+proj=utm +zone=48 +datum=WGS84"
x
## class       : RasterLayer
## dimensions  : 10, 18, 180  (nrow, ncol, ncell)
## resolution  : 111.1111, 100  (x, y)
## extent      : -1000, 1000, -100, 900  (xmin, xmax, ymin, ymax)
## coord. ref. : +proj=utm +zone=48 +datum=WGS84 +ellps=WGS84 +towgs84=0,0,0
```

The objects `x` created in the examples above only consist of the raster 'geometry', that is, we have defined the number of rows and columns, and where the raster is located in geographic space, but there are no cell-values associated with it. Setting and accessing values is illustrated below.

First another example empty raster geometry.

```
r <- raster(ncol=10, nrow=10)
ncell(r)
## [1] 100
hasValues(r)
## [1] FALSE
```

Use the 'values' function.

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

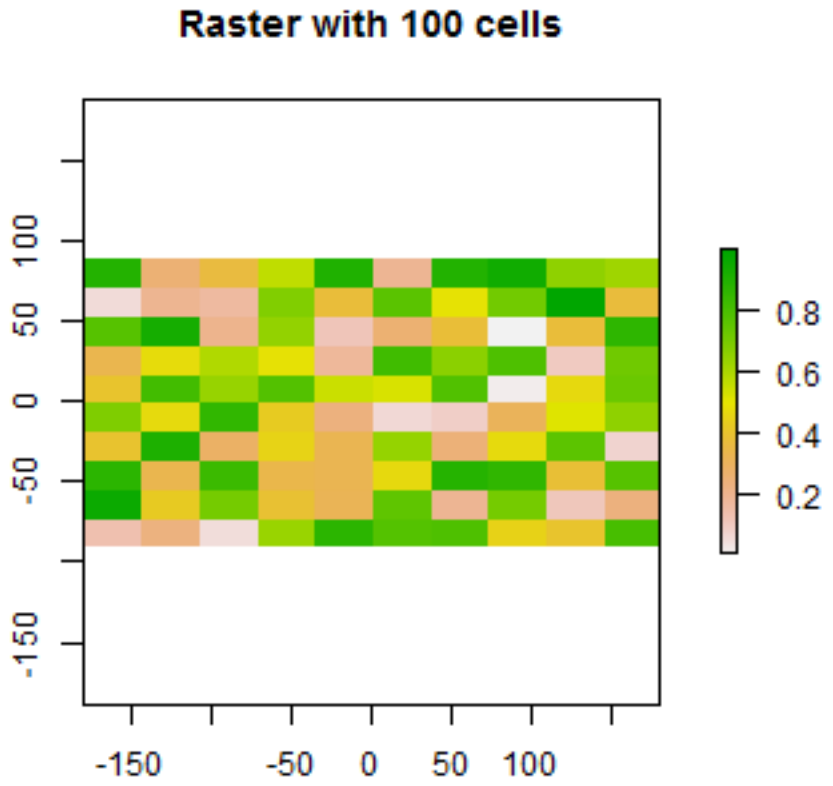
Another example.

```
set.seed(0)
values(r) <- runif(ncell(r))

hasValues(r)
## [1] TRUE
inMemory(r)
## [1] TRUE
values(r)[1:10]
##  [1] 0.8966972 0.2655087 0.3721239 0.5728534 0.9082078 0.2016819 0.8983897
##  [8] 0.9446753 0.6607978 0.6291140
plot(r, main='Raster with 100 cells')
```

In some cases, for example when you change the number of columns or rows, you will lose the values associated with the `RasterLayer` if there were any (or the link to a file if there was one). The same applies, in most cases, if you change the resolution directly (as this can affect the number of rows or columns). Values are not lost when changing the extent as this change adjusts the resolution, but does not change the number of rows or columns.

```
hasValues(r)
## [1] TRUE
res(r)
## [1] 36 18
dim(r)
```


Raster with 100 cells
100
50
0
-50
-150
-150
-50
0
50
100
0.8
0.6
0.4
0.2

```
## [1] 10 10  1
xmax(r)
## [1] 180
```

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

```
xmax(r) <- 0
hasValues(r)
## [1] TRUE
res(r)
## [1] 18 18
dim(r)
## [1] 10 10  1
```

And the number of columns (the values disappear)

```
ncol(r) <- 6
hasValues(r)
## [1] FALSE
res(r)
## [1] 30 18
dim(r)
## [1] 10  6  1
xmax(r)
## [1] 0
```

The function `raster` also allows you to create a `RasterLayer` from another object, including another `RasterLayer`, `RasterStack` and `RasterBrick` , as well as from a `SpatialPixels*` and `SpatialGrid*` object (defined in the `sp` package), an `Extent` object, a matrix, an `im` object (spatstat package), and others.

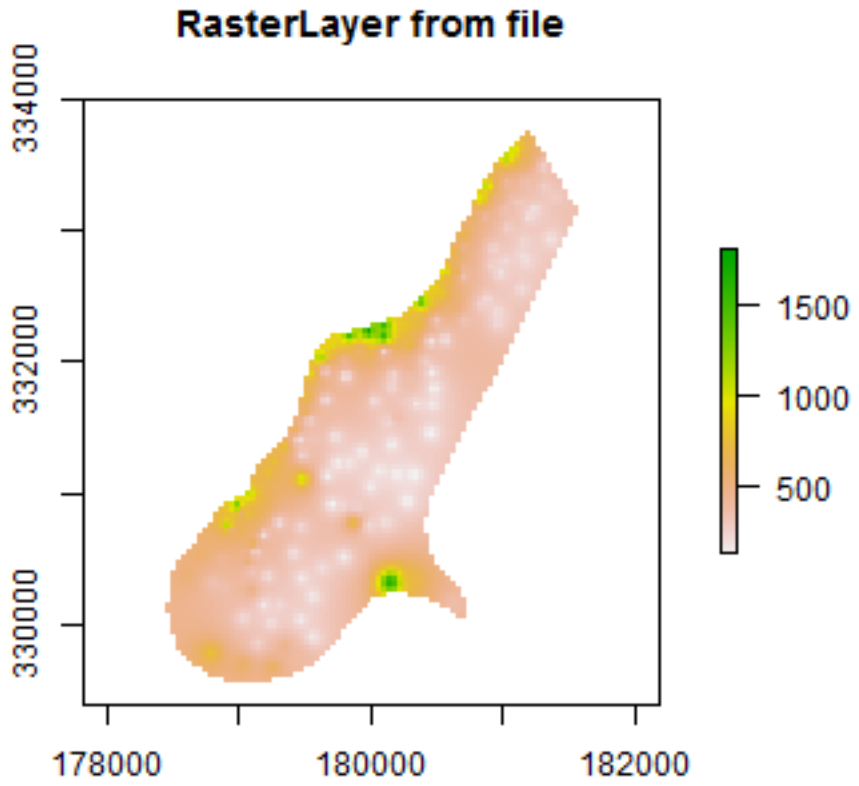
It is more common, however, to create a `RasterLayer` object from a file. The raster package can use raster files in several formats, including some 'natively' supported formats and other formats via the `rgdal` package. Supported formats for reading include GeoTiff, ESRI, ENVI, and ERDAS. Most formats supported for reading can also be written to. Here is an example using the 'Meuse' dataset (taken from the `sp` package), using a file in the native 'raster-file' format.

A notable feature of the `raster` package is that it can work with raster datasets that are stored on disk and are too large to be loaded into memory (RAM). The package can work with large files because the objects it creates from these files only contain information about the structure of the data, such as the number of rows and columns, the spatial extent, and the filename, but it does not attempt to read all the cell values in memory. In computations with these objects, data is processed in chunks. If no output filename is specified to a function, and the output raster is too large to keep in memory, the results are written to a temporary file.

For this example, we first we get the name of an example file installed with the package. Do **not** use this `system.file` construction of your own files (just type the file name; don't forget the forward slashes).

```
filename <- system.file("external/test.grd", package="raster")
filename
## [1] "C:/soft/R/R-3.3.1/library/raster/external/test.grd"
```

```
r <- raster(filename)
filename(r)
## [1] "C:\\soft\\R\\R-3.3.1\\library\\raster\\external\\test.grd"
hasValues(r)
## [1] TRUE
inMemory(r)
## [1] FALSE
plot(r, main='RasterLayer from file')
```


RasterLayer from file
334000
332000
330000
178000
180000
182000
1500
1000
500

Multi-layer objects can be created in memory (from `RasterLayer` objects) or from files.

Create three identical RasterLayer objects

```
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))
```

Combine three RasterLayer objects into a RasterStack.

```
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)
## coord. ref. : +proj=longlat +datum=WGS84 +ellps=WGS84 +towgs84=0,0,0
## names       :    layer.1,    layer.2,    layer.3
## min values  : 0.01307758, 0.02778712, 0.06380247
## max values  :  0.9926841,  0.9815635,  0.9960774
nlayers(s)
## [1] 3
```

Or combine the RasterLayer objects into a RasterBrick.

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

This is equivalent to:

```
b2 <- brick(s)
```

You can also create a RasterBrick from a file.

```
filename <- system.file("external/rlogo.grd", package="raster")
filename
## [1] "C:/soft/R/R-3.3.1/library/raster/external/rlogo.grd"
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)
## coord. ref. : +proj=merc +datum=WGS84 +ellps=WGS84 +towgs84=0,0,0
## data source : C:\soft\R\R-3.3.1\library\raster\external\rlogo.grd
## names       : red, green, blue
## min values  :   0,     0,    0
## max values  : 255,   255,  255
nlayers(b)
## [1] 3
```

Extract a single RasterLayer from a RasterBrick (or RasterStack).

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

In this case, that would be equivalent to creating it from disk with a `band=2` argument.

```
r <- raster(filename, band=2)
```

8.3 Raster algebra

Many generic functions that allow for simple and elegant raster algebra have been implemented for `Raster` objects, including the normal algebraic operators such as {}, logical operators such as >, >=, <, ==, ! and functions like `abs`, `round`, `ceiling`, `floor`, `trunc`, `sqrt`, `log`, `log10`, `exp`, `cos`, `sin`, `atan`, `tan`, `max`, `min`, `range`, `prod`, `sum`, `any`, `all`. In these functions you can mix `raster` objects with numbers, as long as the first argument is a `raster` object.

Create an empty RasterLayer and assign values to cells.

```
r <- raster(ncol=10, nrow=10)
values(r) <- 1:ncell(r)
```

Now some algebra.

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

You can also use replacement functions.

```
s[r] <- -0.5
s[!r] <- 5
s[s == 5] <- 15
```

If you use multiple `Raster` objects (in functions where this is relevant, such as range), these must have the same resolution and origin. The origin of a `Raster` object is the point closest to (0, 0) that you could get if you moved from a corners of a `Raster` object toward that point in steps of the `x` and `y` resolution. Normally these objects would also have the same extent, but if they do not, the returned object covers the spatial intersection of the objects used.

When you use multiple multi-layer objects with different numbers or layers, the 'shorter' objects are 'recycled'. For example, if you multiply a 4-layer object (a1, a2, a3, a4) with a 2-layer object (b1, b2), the result is a four-layer object (a1b1, a2b2, a3b1, a3b2).

```
r <- raster(ncol=5, nrow=5)
r[] <- 1
s <- stack(r, r+1)
q <- stack(r, r+2, r+4, r+6)
x <- r + s + q
x
## class       : RasterBrick
## dimensions  : 5, 5, 25, 4  (nrow, ncol, ncell, nlayers)
## resolution  : 72, 36  (x, y)
## extent      : -180, 180, -90, 90  (xmin, xmax, ymin, ymax)
## coord. ref. : +proj=longlat +datum=WGS84 +ellps=WGS84 +towgs84=0,0,0
## data source : in memory
## names       : layer.1, layer.2, layer.3, layer.4
## min values  :       3,       6,       7,      10
## max values  :       3,       6,       7,      10
```

Summary functions (min, max, mean, prod, sum, Median, cv, range, any, all) always return a `RasterLayer` object. Perhaps this is not obvious when using functions like `min`, `sum` or `mean`.

```
a <- mean(r,s,10)
b <- sum(r,s)
st <- stack(r, s, a, b)
```

```
sst <- sum(st)
sst
## class       : RasterLayer
## dimensions  : 5, 5, 25  (nrow, ncol, ncell)
## resolution  : 72, 36  (x, y)
## extent      : -180, 180, -90, 90  (xmin, xmax, ymin, ymax)
## coord. ref. : +proj=longlat +datum=WGS84 +ellps=WGS84 +towgs84=0,0,0
## data source : in memory
## names       : layer
## values      : 11.5, 11.5  (min, max)
```

Use `cellStats` if instead of a `RasterLayer` you want a single number summarizing the cell values of each layer.

```
cellStats(st, 'sum')
## layer.1.1 layer.1.2 layer.2.1 layer.2.2   layer.3
##      25.0      25.0      50.0      87.5     100.0
cellStats(sst, 'sum')
## [1] 287.5
```

8.4 'High-level' functions

Several 'high level' functions have been implemented for `RasterLayer` objects. 'High level' functions refer to functions that you would normally find in a computer program that supports the analysis of raster data. Here we briefly discuss some of these functions. All these functions work for raster datasets that cannot be loaded into memory. See the help files for more detailed descriptions of each function.

The high-level functions have some arguments in common. The first argument is typically 'x' or 'object' and can be a `RasterLayer`, or, in most cases, a `RasterStack` or `RasterBrick`. It is followed by one or more arguments specific to the function (either additional `RasterLayer` objects or other arguments), followed by a filename="" and "..." arguments.

The default filename is an empty character "". If you do not specify a filename, the default action for the function is to return a `raster` object that only exists in memory. However, if the function deems that the `raster` object to be created would be too large to hold memory it is written to a temporary file instead.

The "..." argument allows for setting additional arguments that are relevant when writing values to a file: the file format, datatype (e.g. integer or real values), and a to indicate whether existing files should be overwritten.

8.4.1 Modifying a Raster* object

There are several functions that deal with modifying the spatial extent of `Raster` objects. The `crop` function lets you take a geographic subset of a larger `raster` object. You can crop a `Raster` by providing an extent object or another spatial object from which an extent can be extracted (objects from classes deriving from Raster and from Spatial in the sp package). An easy way to get an extent object is to plot a `RasterLayer` and then use `drawExtent` to visually determine the new extent (bounding box) to provide to the crop function.

`trim` crops a `RasterLayer` by removing the outer rows and columns that only contain `NA` values. In contrast, `extend` adds new rows and/or columns with `NA` values. The purpose of this could be to create a new `RasterLayer` with the same Extent of another larger `RasterLayer` such that the can be used together in other functions.

The `merge` function lets you merge 2 or more `Raster` objects into a single new object. The input objects must have the same resolution and origin (such that their cells neatly fit into a single larger raster). If this is not the case you can first adjust one of the `Raster` objects with use `(dis)aggregate` or `resample`.

`aggregate` and `disaggregate` allow for changing the resolution (cell size) of a `Raster` object. In the case of `aggregate`, you need to specify a function determining what to do with the grouped cell values `mean`. It is possible

to specify different (dis)aggregation factors in the x and y direction. `aggregate` and `disaggregate` are the best functions when adjusting cells size only, with an integer step (e.g. each side 2 times smaller or larger), but in some cases that is not possible.

For example, you may need nearly the same cell size, while shifting the cell centers. In those cases, the `resample` function can be used. It can do either nearest neighbor assignments (for categorical data) or bilinear interpolation (for numerical data). Simple linear shifts of a Raster object can be accomplished with the `shift` function or with the `extent` function. `resample` should not be used to create a Raster* object with much larger resolution. If such adjustments need to be made then you can first use aggregate.

With the `projectRaster` function you can transform values of `Raster` object to a new object with a different coordinate reference system.

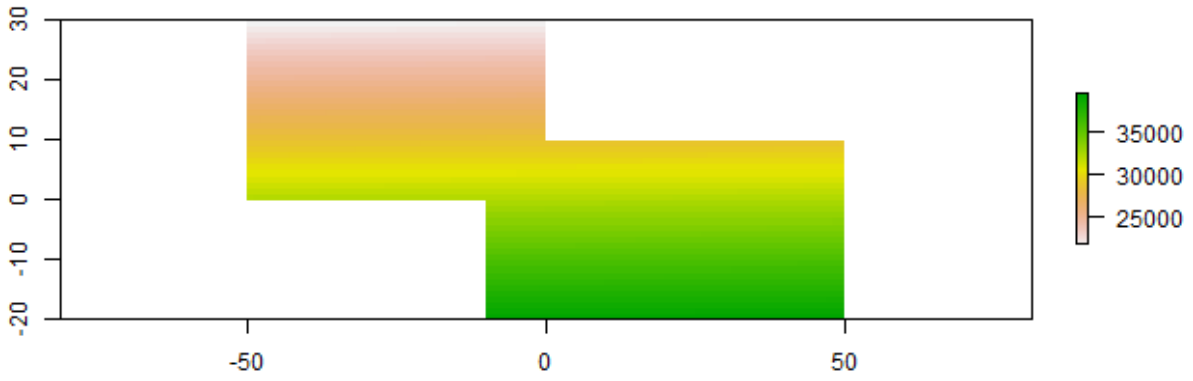
Here are some simple examples.

Aggregate and disaggregate.

```
r <- raster()
r[] <- 1:ncell(r)
ra <- aggregate(r, 20)
rd <- disaggregate(ra, 20)
```

Crop and merge example.

```
r1 <- crop(r, extent(-50,0,0,30))
r2 <- crop(r, extent(-10,50,-20, 10))
m <- merge(r1, r2, filename='test.grd', overwrite=TRUE)
plot(m)
```



`flip` lets you flip the data (reverse order) in horizontal or vertical direction – typically to correct for a 'communication problem' between different R packages or a misinterpreted file. `rotate` lets you rotate longitude/latitude rasters that have longitudes from 0 to 360 degrees (often used by climatologists) to the standard -180 to 180 degrees system. With `t` you can rotate a `Raster` object 90 degrees.

8.4.2 Overlay

The `overlay` function can be used as an alternative to the raster algebra discussed above. Overlay, like the functions discussed in the following subsections provide either easy to use short-hand, or more efficient computation for large

(file based) objects.

With `overlay` you can combine multiple Raster objects (e.g. multiply them). The related function `mask` removes all values from one layer that are `NA` in another layer, and `cover` combines two layers by taking the values of the first layer except where these are `NA`.

8.4.3 Calc

`calc` allows you to do a computation for a single `raster` object by providing a function. If you supply a `RasterLayer`, another `RasterLayer` is returned. If you provide a multi-layer object you get a (single layer) `RasterLayer` if you use a summary type function (e.g. `sum` but a `RasterBrick` if multiple layers are returned. `stackApply` computes summary type layers for subsets of a `RasterStack` or `RasterBrick`.

8.4.4 Reclassify

You can use `cut` or `reclassify` to replace ranges of values with single values, or `subs` to substitute (replace) single values with other values.

```
r <- raster(ncol=3, nrow=2)
r[] <- 1:ncell(r)
getValues(r)
## [1] 1 2 3 4 5 6
```

Set all values above 4 to `NA`

```
s <- calc(r, fun=function(x){ x[x < 4] <- NA; return(x)} )
as.matrix(s)
##      [,1] [,2] [,3]
## [1,]   NA   NA   NA
## [2,]    4    5    6
```

Divide the first raster with two times the square root of the second raster and add five.

```
w <- overlay(r, s, fun=function(x, y){ x / (2 * sqrt(y)) + 5 } )
as.matrix(w)
##      [,1]     [,2]     [,3]
## [1,]   NA       NA       NA
## [2,]    6 6.118034 6.224745
```

Remove from `r` all values that are `NA` in `w`.

```
u <- mask(r, w)
as.matrix(u)
##      [,1] [,2] [,3]
## [1,]   NA   NA   NA
## [2,]    4    5    6
```

Identify the cell values in `u` that are the same as in `s`.

```
v <- u==s
as.matrix(v)
##      [,1] [,2] [,3]
## [1,]   NA   NA   NA
## [2,] TRUE TRUE TRUE
```

Replace `NA` values in `w` with values of `r`.

```
cvr <- cover(w, r)
as.matrix(w)
##       [,1]     [,2]     [,3]
## [1,]   NA       NA       NA
## [2,]    6 6.118034 6.224745
```

Change value between 0 and 2 to 1, etc.

```
x <- reclassify(w, c(0,2,1,  2,5,2, 4,10,3))
as.matrix(x)
##       [,1] [,2] [,3]
## [1,]   NA   NA   NA
## [2,]    3    3    3
```

Substitute 2 with 40 and 3 with 50.

```
y <- subs(x, data.frame(id=c(2,3), v=c(40,50)))
as.matrix(y)
##       [,1] [,2] [,3]
## [1,]   NA   NA   NA
## [2,]   50   50   50
```

8.4.5 Focal functions

The `focal` function currently only work for (single layer) RasterLayer objects. They make a computation using values in a neighborhood of cells around a focal cell, and putting the result in the focal cell of the output RasterLayer. The neighborhood is a user-defined matrix of weights and could approximate any shape by giving some cells zero weight. It is possible to only computes new values for cells that are `NA` in the input RasterLayer.

8.4.6 Distance

There are a number of distance related functions. `distance` computes the shortest distance to cells that are not `NA`. `pointDistance` computes the shortest distance to any point in a set of points. `gridDistance` computes the distance when following grid cells that can be traversed (e.g. excluding water bodies). `direction` computes the direction toward (or from) the nearest cell that is not `NA`. `adjacency` determines which cells are adjacent to other cells. See the `gdistance` package for more advanced distance calculations (cost distance, resistance distance)

8.4.7 Spatial configuration

Function `clump` identifies groups of cells that are connected. `boundaries` identifies edges, that is, transitions between cell values. `area` computes the size of each grid cell (for unprojected rasters), this may be useful to, e.g. compute the area covered by a certain class on a longitude/latitude raster.

```
r <- raster(nrow=45, ncol=90)
r[] <- round(runif(ncell(r))*3)
a <- area(r)
zonal(a, r, 'sum')
##      zone       sum
## [1,]    0  93604336
## [2,]    1 168894837
## [3,]    2 158110025
## [4,]    3  87822040
```

8.4.8 Predictions

The package has two functions to make model predictions to (potentially very large) rasters. `predict` takes a multilayer raster and a fitted model as arguments. Fitted models can be of various classes, including glm, gam, and RandomForest. The function `interpolate` is similar but is for models that use coordinates as predictor variables, for example in Kriging and spline interpolation.

8.4.9 Vector to raster conversion

The raster packages supports point, line, and polygon to raster conversion with the `rasterize` function. For vector type data (points, lines, polygons), objects of Spatial* classes defined in the `sp` package are used; but points can also be represented by a two-column matrix (x and y).

Point to raster conversion is often done with the purpose to analyze the point data. For example to count the number of distinct species (represented by point observations) that occur in each raster cell. `rasterize` takes a `Raster` object to set the spatial extent and resolution, and a function to determine how to summarize the points (or an attribute of each point) by cell.

Polygon to raster conversion is typically done to create a `RasterLayer` that can act as a mask, i.e. to set to `NA` a set of cells of a `raster` object, or to summarize values on a raster by zone. For example a country polygon is transferred to a raster that is then used to set all the cells outside that country to `NA`; whereas polygons representing administrative regions such as states can be transferred to a raster to summarize raster values by region.

It is also possible to convert the values of a `RasterLayer` to points or polygons, using `rasterToPoints` and `rasterToPolygons`. Both functions only return values for cells that are not `NA`. Unlike `rasterToPolygons`, `rasterToPoints` is reasonably efficient and allows you to provide a function to subset the output before it is produced (which can be necessary for very large rasters as the point object is created in memory).

8.5 Summarizing functions

When used with a `Raster` object as first argument, normal summary statistics functions such as min, max and mean return a RasterLayer. You can use cellStats if, instead, you want to obtain a summary for all cells of a single `Raster` object. You can use `freq` to make a frequency table, or to count the number of cells with a specified value. Use `zonal` to summarize a `Raster` object using zones (areas with the same integer number) defined in a `RasterLayer` and `crosstab` to cross-tabulate two `RasterLayer` objects.

```
r <- raster(ncol=36, nrow=18)
r[] <- runif(ncell(r))
cellStats(r, mean)
## [1] 0.5179682
```

Zonal stats

```
s <- r
s[] <- round(runif(ncell(r)) * 5)
zonal(r, s, 'mean')
##      zone      mean
## [1,]    0 0.5144431
## [2,]    1 0.5480089
## [3,]    2 0.5249257
## [4,]    3 0.5194031
## [5,]    4 0.4853966
## [6,]    5 0.5218401
```

Count cells

```
freq(s)
##      value count
## [1,]     0    54
## [2,]     1   102
## [3,]     2   139
## [4,]     3   148
## [5,]     4   133
## [6,]     5    72
freq(s, value=3)
## [1] 148
```

Cross-tabulate

```
ctb <- crosstab(r*3, s)
head(ctb)
##   Var1 Var2 Freq
## 1    0    0    8
## 2    1    0   17
## 3    2    0   19
## 4    3    0   10
## 5 <NA>    0    0
## 6    0    1   13
```

8.6 Helper functions

The cell number is an important concept in the raster package. Raster data can be thought of as a matrix, but in a `RasterLayer` it is more commonly treated as a vector. Cells are numbered from the upper left cell to the upper right cell and then continuing on the left side of the next row, and so on until the last cell at the lower-right side of the raster. There are several helper functions to determine the column or row number from a cell and vice versa, and to determine the cell number for x, y coordinates and vice versa.

```
library(raster)
r <- raster(ncol=36, nrow=18)
ncol(r)
## [1] 36
nrow(r)
## [1] 18
ncell(r)
## [1] 648
rowFromCell(r, 100)
## [1] 3
colFromCell(r, 100)
## [1] 28
cellFromRowCol(r,5,5)
## [1] 149
xyFromCell(r, 100)
##       x  y
## [1,] 95 65
cellFromXY(r, c(0,0))
## [1] 343
colFromX(r, 0)
## [1] 19
rowFromY(r, 0)
## [1] 10
```

8.7 Accessing cell values

Cell values can be accessed with several methods. Use `getValues` to get all values or a single row; and `getValuesBlock` to read a block (rectangle) of cell values.

```
r <- raster(system.file("external/test.grd", package="raster"))
v <- getValues(r, 50)
v[35:39]
## [1] 456.878 485.538 550.788 580.339 590.029
getValuesBlock(r, 50, 1, 35, 5)
## [1] 456.878 485.538 550.788 580.339 590.029
```

You can also read values using cell numbers or coordinates (xy) using the `extract` method.

```
cells <- cellFromRowCol(r, 50, 35:39)
cells
## [1] 3955 3956 3957 3958 3959
extract(r, cells)
## [1] 456.878 485.538 550.788 580.339 590.029
xy <- xyFromCell(r, cells)
xy
##           x      y
## [1,] 179780 332020
## [2,] 179820 332020
## [3,] 179860 332020
## [4,] 179900 332020
## [5,] 179940 332020
extract(r, xy)
## [1] 456.878 485.538 550.788 580.339 590.029
```

You can also extract values using `SpatialPolygons*` or `SpatialLines*`. The default approach for extracting raster values with polygons is that a polygon has to cover the center of a cell, for the cell to be included. However, you can use argument "weights=TRUE" in which case you get, apart from the cell values, the percentage of each cell that is covered by the polygon, so that you can apply, e.g., a "50% area covered" threshold, or compute an area-weighted average.

In the case of lines, any cell that is crossed by a line is included. For lines and points, a cell that is only 'touched' is included when it is below or to the right (or both) of the line segment/point (except for the bottom row and right-most column).

In addition, you can use standard *R* indexing to access values, or to replace values (assign new values to cells) in a `raster` object. If you replace a value in a `raster` object based on a file, the connection to that file is lost (because it now is different from that file). Setting raster values for very large files will be very slow with this approach as each time a new (temporary) file, with all the values, is written to disk. If you want to overwrite values in an existing file, you can use `update` (with caution!)

```
r[cells]
## [1] 456.878 485.538 550.788 580.339 590.029
r[1:4]
## [1] NA NA NA NA
filename(r)
## [1] "C:\\soft\\R\\R-3.3.1\\library\\raster\\external\\test.grd"
r[2:3] <- 10
r[1:4]
## [1] NA 10 10 NA
filename(r)
## [1] ""
```

Note that in the above examples values are retrieved using cell numbers. That is, a raster is represented as a (one-dimensional) vector. Values can also be inspected using a (two-dimensional) matrix notation. As for *R* matrices, the first index represents the row number, the second the column number.

```
r[1]
##
## NA
r[2,2]
##
## NA
r[1,]
##  [1] NA 10 10 NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA
## [24] NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA
## [47] NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA
## [70] NA NA NA NA NA NA NA NA NA NA NA
r[,2]
##   [1]  10.000      NA      NA      NA      NA      NA      NA      NA
##   [9]      NA      NA      NA      NA      NA      NA      NA      NA
##  [17]      NA      NA      NA      NA      NA      NA      NA      NA
##  [25]      NA      NA      NA      NA      NA      NA      NA      NA
##  [33]      NA      NA      NA      NA      NA      NA      NA      NA
##  [41]      NA      NA      NA      NA      NA      NA      NA      NA
##  [49]      NA      NA      NA      NA      NA      NA      NA      NA
##  [57]      NA      NA      NA      NA      NA      NA      NA      NA
##  [65]      NA      NA      NA      NA      NA      NA      NA      NA
##  [73]      NA      NA      NA      NA      NA      NA      NA      NA
##  [81]      NA      NA      NA      NA      NA      NA      NA      NA
##  [89]      NA      NA      NA      NA      NA      NA      NA 473.833
##  [97] 471.573      NA      NA      NA      NA      NA      NA      NA
## [105]      NA      NA      NA      NA      NA      NA      NA      NA
## [113]      NA      NA      NA
r[1:3,1:3]
## [1] NA 10 10 NA NA NA NA NA NA

# keep the matrix structure
r[1:3,1:3, drop=FALSE]
## class       : RasterLayer
## dimensions  : 3, 3, 9  (nrow, ncol, ncell)
## resolution  : 40, 40  (x, y)
## extent      : 178400, 178520, 333880, 334000  (xmin, xmax, ymin, ymax)
## coord. ref. : +init=epsg:28992 +towgs84=565.237,50.0087,465.658,-0.406857,0.350733,-1.87035,4.081
## data source : in memory
## names       : layer
## values      : 10, 10  (min, max)
```

Accessing values through this type of indexing should be avoided inside functions as it is less efficient than accessing values via functions like `getValues`.

8.8 Coercion to other classes

Although the `raster` package defines its own set of classes, it is easy to coerce objects of these classes to objects of the `Spatial` family defined in the `sp` package. This allows for using functions defined by sp (e.g. spplot) and for using other packages that expect Spatial* objects. To create a Raster object from variable n in a SpatialGrid* x use `raster(x, n)` or `stack(x)` or `brick(x)`. Vice versa use `as( , )`. You can also convert objects of class `im` (spatstat) and others to a `RasterLayer` using the `raster`, `stack` or `brick` functions.

```
r1 <- raster(ncol=36, nrow=18)
r2 <- r1
r1[] <- runif(ncell(r1))
r2[] <- runif(ncell(r1))
s <- stack(r1, r2)
sgdf <- as(s, 'SpatialGridDataFrame')
newr2 <- raster(sgdf, 2)
news <- stack(sgdf)
```

CHAPTER

NINE

9. MAPS

Like for other plots, there are different approaches in R to make maps. You can use "base plot" in many cases. Alternatively use levelplot, either via the `spplot` function (implemented in `sp` and `raster`) or via the `rasterVis` package.

Here are some brief examples about making maps. You can also look elsewhere on the Internet, like here, or this for spplot and rasterVis.

9.1 Vector data

9.1.1 Base plots

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

```
n <- length(p)
plot(p, col=rainbow(n))
```

One colour per region (NAME_1)

```
u <- unique(p$NAME_1)
u
## [1] "Diekirch"     "Grevenmacher" "Luxembourg"
m <- match(p$NAME_1, u)
plot(p, col=rainbow(n)[m])
text(p, 'NAME_2', cex=.75, halo=TRUE)
```

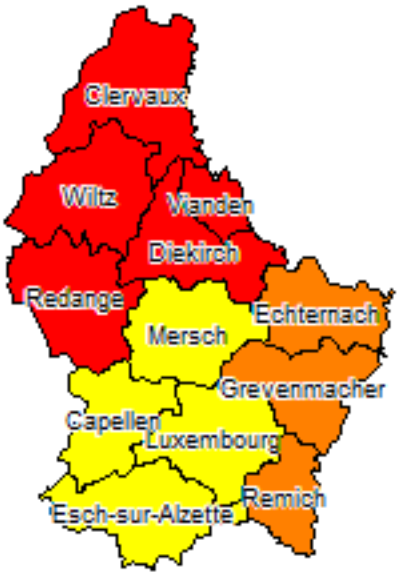
9.1.2 spplot

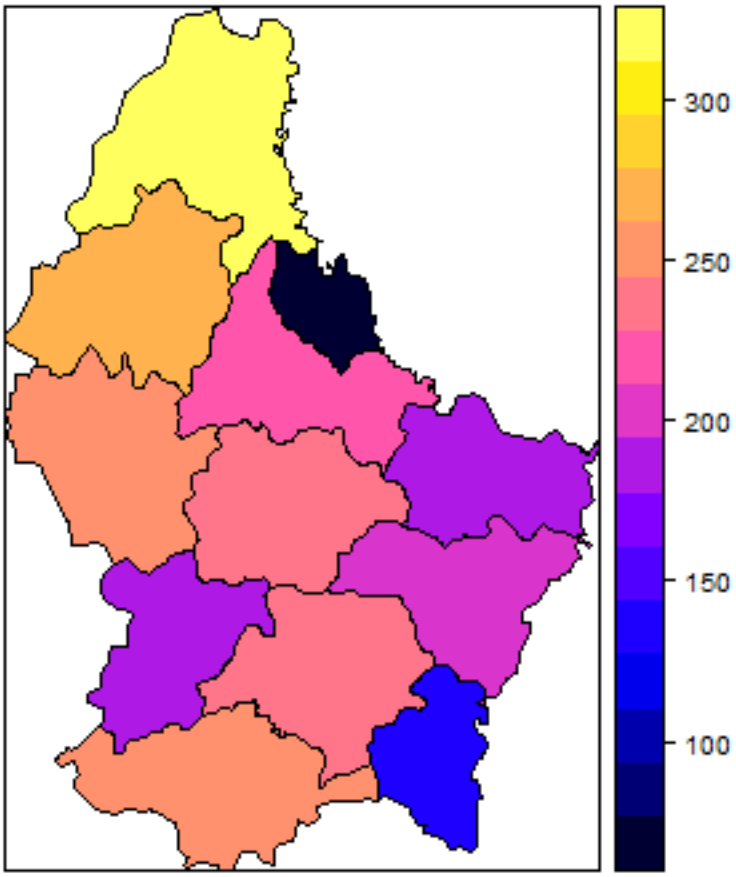
```
spplot(p, 'AREA')
```

9.2 Raster

Example data

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


Clervaux
Wiltz
Vianden
Diekirch
Redange
Mersch
Echternach
Grevenmacher
Capellen
Luxembourg
Remich
Esch-sur-Alzette

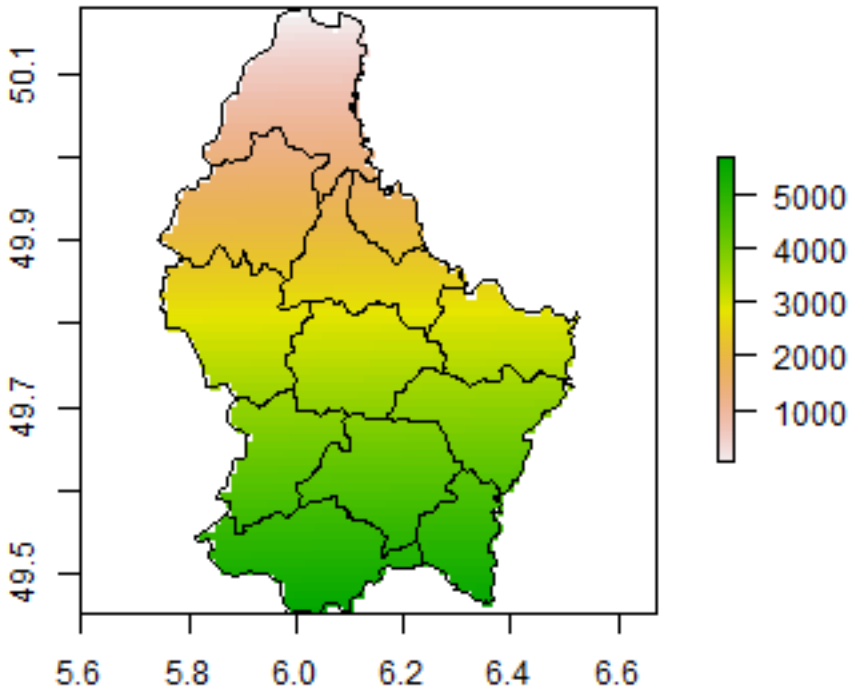

300
250
200
150
100

```
r <- raster(p, res=0.01 )
values(r) <- 1:ncell(r)
r <- mask(r, p)
```

Several generic functions have been implemented for Raster* objects to create maps and other plot types. Use 'plot' to create a map of a Raster* object. When plot is used with a `RasterLayer`, it calls the function 'rasterImage' (but, by default, adds a legend; using code from `fields::image.plot`). It is also possible to directly call `image`. You can zoom in using 'zoom' and clicking on the map twice (to indicate where to zoom to). With `click` it is possible to interactively query a Raster* object by clicking once or several times on a map plot.

After plotting a `RasterLayer` you can add vector type spatial data (points, lines, polygons). You can do this with functions points, lines, polygons if you are using the basic R data structures or plot(object, add=TRUE) if you are using Spatial* objects as defined in the sp package. When plot is used with a multi-layer Raster* object, all layers are plotted (up to 16), unless the layers desired are indicated with an additional argument.

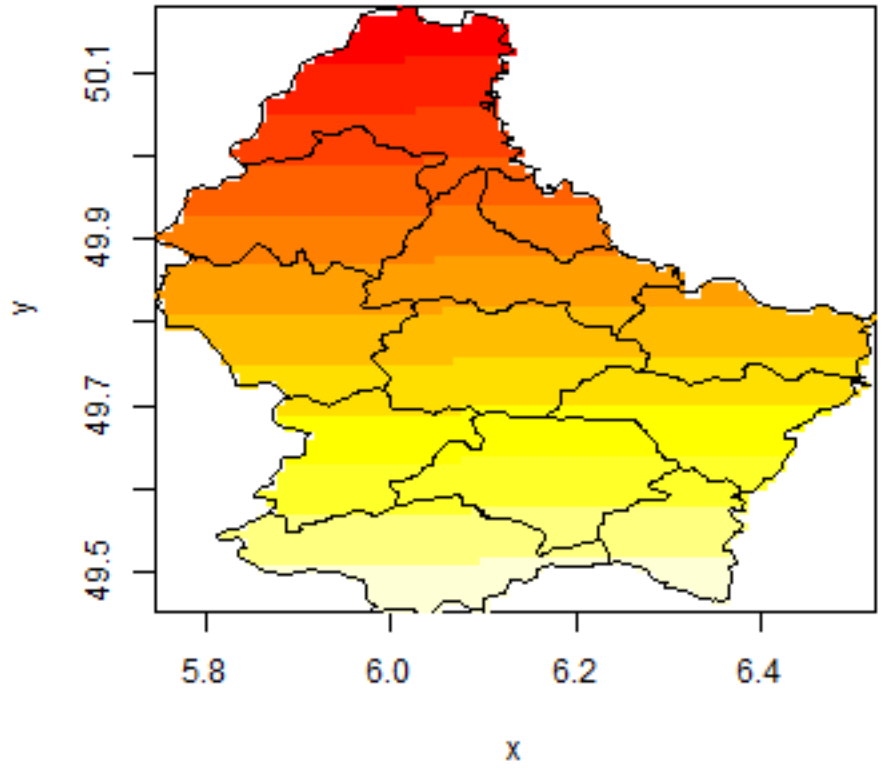
```
plot(r)
plot(p, add=TRUE)
```



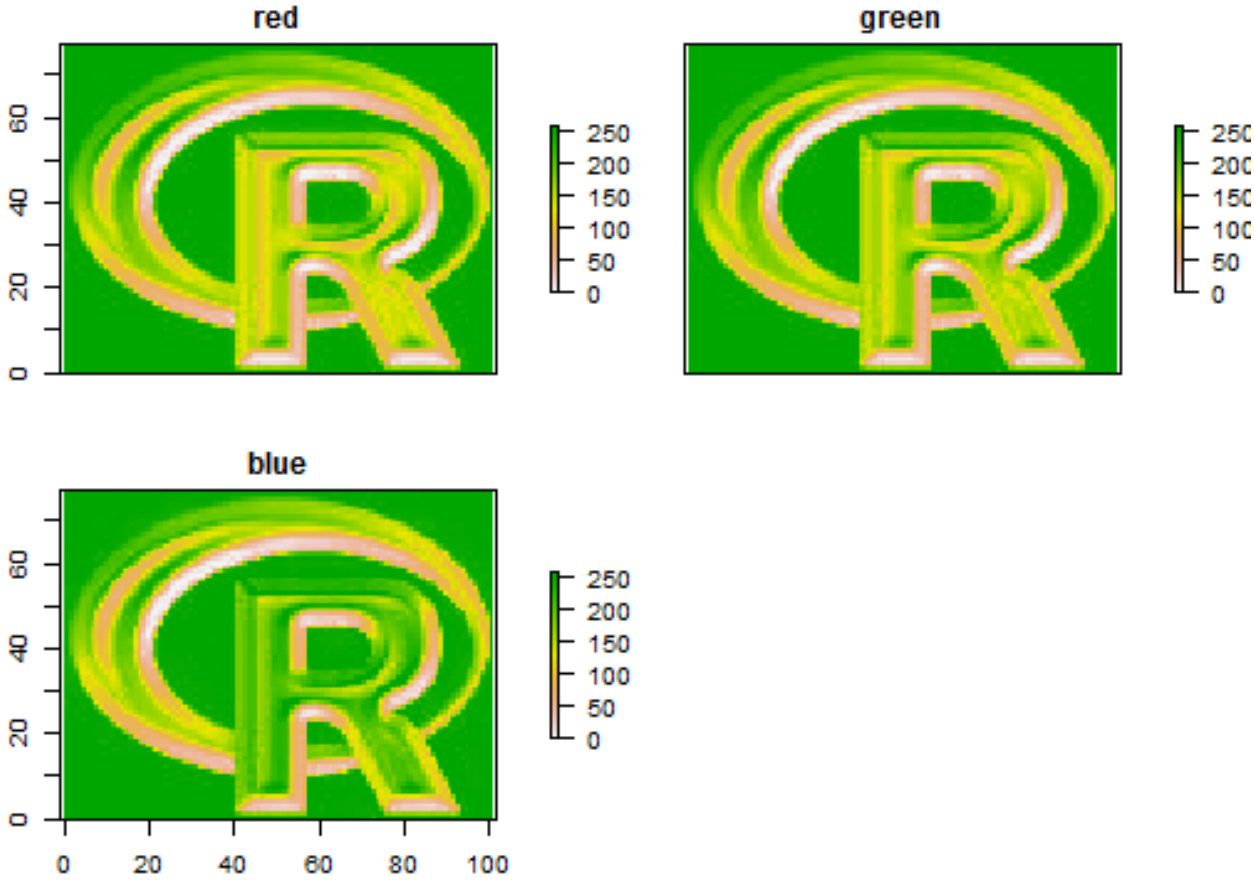
`image` does not provide a legend and that can be advantageous in some cases.

```
image(r)
plot(p, add=TRUE)
```

Multi-layer Raster objects can be plotted as individual layers


50.1
49.9
49.7
49.5
y
5.8
6.0
6.2
6.4
x

```
plot(b)
```



They can also be combined into a single image, by assigning individual layers to one of the three color channels (red, green and blue):

```
plotRGB(b, r=1, g=2, b=3)
```

You can also plot `Raster*` objects with spplot.

```
bounds <- list("sp.polygons", p)
spplot(r, sp.layout=bounds)
```

```
spplot(b, layout=c(3,1))
```

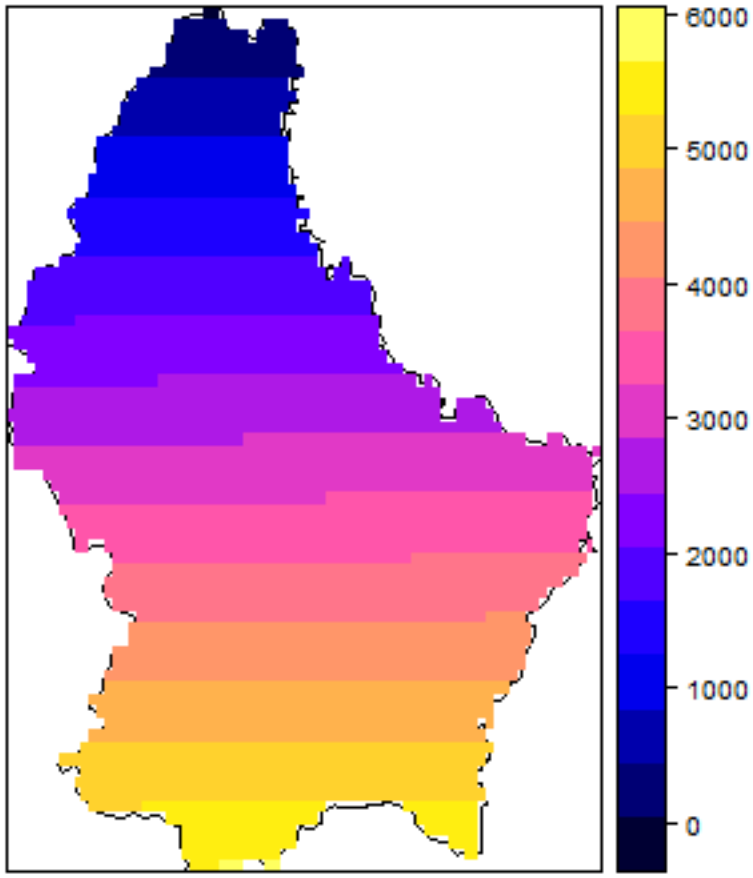
The `rasterVis` package has several other `lattice` based plotting functions for `Raster*` objects. The rasterVis package also facilitates creating a map from a `RasterLayer` with the `ggplot2` package.

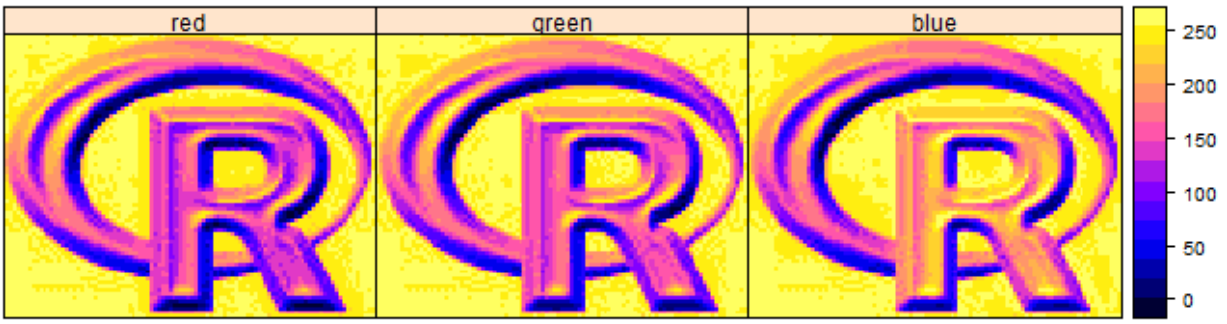
You can also use the a number of other plotting functions with a `raster` object as argument, including `hist`, `persp`, `contour`}, and `density`. See the help files for more info.

9.3 Basemaps

From Google and others...

Get a google map.


6000
5000
4000
3000
2000
1000
0


red
green
blue
250
200
150
100
50
0

```
library(dismo)
g <- gmap("Belgium")
## Loading required namespace: XML
```

Plot it

```
plot(g, interpolate=TRUE)
```

```
brus <- geocode('Brussels, Belgium')
merc <- Mercator(brus[, c('longitude', 'latitude')])
points(merc, pch='*', col='red', cex=5)
```

9.4 Specialized packages

coming soon....


The Hague
Netherlands
Arnhem
Münster
Rotterdam
Dortmund
Eindhoven
Essen
Antwerp
Düsseldorf
Bruges
Ghent
Dunkirk
Calais
Cologne
Brussels
Bonn
Aachen
Liège
Lille
Belgium
Koblenz
Arras
Charleroi
Amiens
Luxembourg City
Trier
Reims
Saarbrücken
Metz
Paris
Versailles
Nancy
Google
Map data ©2016 GeoBasis-DE/BKG (©2009), Google