Guide to making map shapefiles for ACER purposes using R

Reinier Overmaat

May 2021

This document will outline the process of creating map geometries programatically using R. These map geometries can be used to create many different types of files (such as Shapefile `.shp`, GeoJSON, or topoJSON), which can be used for custom maps in Power BI, can be used in GIS applications or other software, and can serve as the basis to produce map figures for ACER publications and communications. The goal of this guide is to show a variety of tools and methods for working with geospatial data in R, after which the user can find their own data sources, and use the tools, methods and knowledge to make maps fitting their own need.

1 Preparations

1.1 Setting up the R environment

The first step would be to set up your environment. Install R and Rstudio if you have not done so already, create a new script, and include the following packages at the beginning of your script:

- rnaturalearth
  - rnaturalearthdata
  - rnaturalearthhires
- rgdal
- tidyverse
- raster
- rgeos
- sp
- geojsonio

Some packages are dependent on other packages, which should also be installed automatically.

1.2 Making a list of data

Next, think about which geometries you would need to make your map. You probably have an idea already of what your map will show, so just make a list of what you need. Do you need all the countries in Europe for example, or just the EU member states? Do you need the country borders, or also provincial/regional borders?

2 Working with geospatial data in R

2.1 Loading the required geometries using rnaturalearth

The next step is loading all the geometries using rnaturalearth. The Natural Earth project (www.naturalearthdata.com) is a repository of vector maps of all parts of the world. We can access those maps directly in R using the rnaturalearth package. There are two commands you can use:

ne_countries will give you country boundaries of the whole world. You can specify parameters to narrow down the returned data. For instance, `ne_countries(continent = "Europe")` or `ne_countries(country = "Italy")`. There is also the parameter *scale* which defines the resolution. Scale can take the values "small", "medium", or "large" (equivalent to 1:110m, 1:50m, or 1:10m map scale), where "small" is the coarsest map, and "large" is the most detailed. As with any geometry, more detail yields better-looking maps, but also increases computation times and memory requirements. In figure 2, compare the output of `ne_countries(country = "Italy", scale = "small")`, which is on the left, and the output of `ne_countries(country = "Italy", scale = "large")`, which is on the right.

Figure 1: Countries of continent Europe

Figure 2: Difference between small and large scale

ne_states does the same as `ne_countries`, except it also returns the first level of regional boundaries. For example `ne_states(country = "Germany")` returns the Germany border as well as the borders of the 16 federal subdivisions (Bundersländer). `ne_states` will always return the geometries in "large" detail.

You can use `plot()` at any time to visualise your geometry variables. More details can be found in this introduction: https://cran.r-project.org/web/packages/rnaturalearth/vignettes/rnaturalearth.html

2.2 Filtering geometries

Now the interesting bit starts. When you have loaded your data using the previous section, you will see they are stored in a data format called `SpatialPolygonsDataFrame` (sp is the abbreviation for SpatialPolygons). This `SpatialPolygonsDataFrame` format contains a list of polygons which defines the geographic part of the geometry, and data part. This data comprises of a list of attributes, and for each attribute, there is a value for each sub-geometry. Below is an example.

```
my_geometry <- ne_states(country = "Germany")
plot(my_geometry)
```

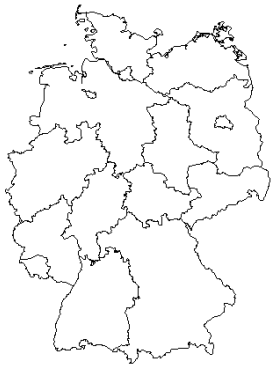


Figure 3: State division of Germany

Now `my_geometry` is a SpatialPolygonsDataFrame. It contains 16 polygon collections corresponding to the 16 states of Germany. Then there is the list of data, which contains 83 attributes. One of the attributes for instance is `Name`. The attribute `Name` consists of 16 character vectors, which contains a name for all of the 16 states. We can use the data portion of the SpatialPolygonsDataFrame to filter our geometries based on the value of an attribute. Say we want to plot France

```
my_france_geometry <- ne_states(country = "France")
plot(my_france_geometry)
```

We see that not only mainland France, but also the overseas territories are included (figure 4a). If we wanted to have only the provinces of mainland France, we would need to find a way to filter out those overseas territories. Looking in the data list of `my_france_geometry`, we see there is an attribute called `geounit`. To filter out the overseas territories we can do subsetting using a logical check, employing our `geounit` attribute. This is the same syntax that you may be familiar with when subsetting a regular `data.frame` object:

```
mainland_france <- my_france_geometry[my_france_geometry$geonunit == "France", ]
plot(mainland_france)
```

Here, we filter out any geounit which is not part of "France", as the overseas geounits have a different name. The result is displayed in figure 4b, where we have only the states of mainland France displayed.

NOTE that there is a typo in the original data here so that it says "geoNunit" instead of "geounit"

In this way, you can also select only certain countries in Europe based on a name attribute. In the following example you see it's possible to make a list beforehand, and use the `%in%` operator to make the logical check instead of writing each check as a separate subsetting statement:

```
list_of_v4_countries <- c("Czech Republic", "Hungary", "Poland", "Slovakia")
europe_countries <- ne_countries(continent = "Europe")
v4_countries <- europe_countries[europe_countries$name_long %in% list_of_v4_countries, ]
plot(v4_countries)
```

(a) All of France's states, including overseas

(b) Only mainland France's states

Figure 4: Filtering out France's overseas territories

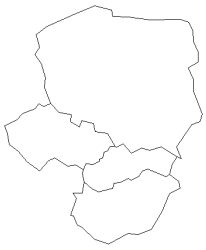


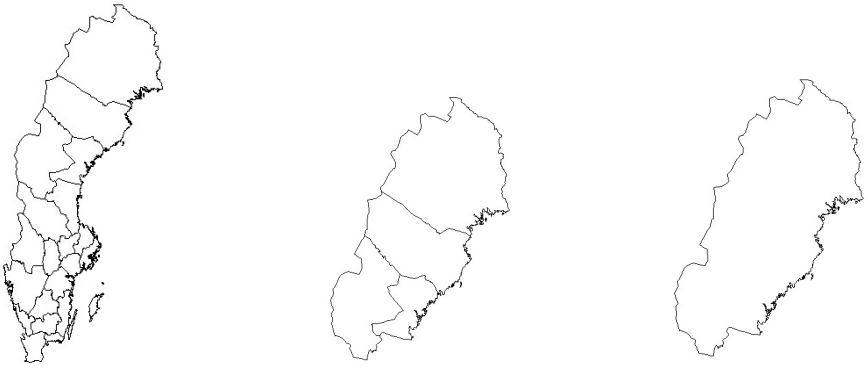
Figure 5: Countries of the Visegrád group (V4)

2.3 Dissolving internal boundaries in a set of geometries

If you have selected and filtered geometries, but now would like to merge those geometries into one, we can dissolve the borders between those geometries using the command `rgeos::gUnaryUnion`. It takes all geometries in your `SpatialPolygonDataFrame` and removes any shared borders between them, and consolidates them into one big geometry.

As an example, say that we want to select the most northern few provinces of Sweden and join them together to have a geometry for "North Sweden". We could make it like this:

```
sweden_states <- ne_states(country = "Sweden")
# these are the names of the four north most provinces
north_sweden_states <- Sweden_states[Sweden_states$name == "Norrbotten" |
                                     Sweden_states$name == "Västerbotten" |
                                     Sweden_states$name == "Västernorrland" |
                                     Sweden_states$name == "Jämtland", ]
grouped_north_sweden <- rgeos::gUnaryUnion(north_sweden_states)
# now compare
plot(sweden_states)
plot(north_sweden_states)
plot(grouped_north_sweden)
```



(a) Provinces of Sweden

(b) Four northern most provinces of Sweden

(c) Those four provinces with dissolved internal boundaries

Figure 6: Dissolving internal boundaries

There is an additional argument in the `rgeos::gUnaryUnion` command which we left empty here, but which can be used to indicate which geometries to dissolve. If we go back to the example of states of mainland France in our `mainland_france SpatialPolygonsDataFrame`, there is also an attribute called *Region*, which lists the region that this state belongs to. If we wanted to dissolve all boundaries within the regions, but not between the regions, we can add the *Region* attribute as ID in our `rgeos::gUnaryUnion` command

```
regions_of_france <- rgeos::gUnaryUnion(mainland_france, ID = mainland_france$region)
```

Now, only the boundaries of the states that belong to the same region have been dissolved, but not the boundaries between the different regions (figure 7).

Note that the resulting geometry is of the type SpatialPolygons, and not SpatialPolygonsDataFrame. This will be addressed in the following steps.

(a) States of mainland France as we saw before

(b) Boundaries inside the regions are dissolved

Figure 7: Dissolving boundaries in France's regions based on the region attribute

2.4 Binding sets of geometries together

To combine multiple elements so they can belong to the same map, we can use the command `raster::bind`. Simply list the geometries you want to combine and the result will be a combined geometry. Notice that the geometries need to be of the same type for this to work, so either all `SpatialPolygonsDataFrame`, or just `SpatialPolygons`. Notice that when you execute a `rgeos::gUnaryUnion`, the result is no longer a `SpatialPolygonsDataFrame`, but a `SpatialPolygons`.

As an example, if we want to have a map of the Benelux region, we could of course take all countries of Europe, and then subset using the method in the filtering step. Instead, using `raster::bind` it would look like this:

```
be_country <- ne_countries(country = "Belgium")
nl_country <- ne_countries(country = "Netherlands")
# filter out Caribbean part of NL
nl_mainland_country <- nl_country[nl_country$geounit == "Netherlands"]
lux_country <- ne_countries(country = "Luxembourg")
benelux <- raster::bind(be_country, nl_mainland_country, lux_country)
plot(benelux)
```

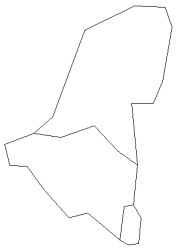


Figure 8: BeNeLux countries (notice how the scale is "small")

NOTE: if you use geospatial data from another source than rnaturalearth (such as from the raster package, or from EU NUTS/LAU sources), it might be they have a different coordinate reference system. Always harmonize the reference systems before binding geometries together. This can be done using the `spTransform` command

2.5 Adding information to your newly created geometries

Notice that when you execute a `rgeos::gUnaryUnion`, the result is no longer a `SpatialPolygonsDataFrame`, but a `SpatialPolygons`. One is still able to plot the `SpatialPolygons`, but the data which was contained in the `data.frame` part is removed in the process. We can manually assign only the relevant data that we need to the new geometries, for example to the region of North Sweden which we created earlier.

You can do this simply by creating a `data.frame` object and giving it the attributes you wish to be included in your final map. There can be as little or as many attributes as you want, but attributes should have a number of records equal to the number of geometries that you bound together in the previous step. Since our Sweden example consist of only one geometry, only one value is given for each attribute in the example below. Now, we can convert the `SpatialPolygons` back into a `SpatialPolygonsDataFrame` with our own data using the `SpatialPolygonsDataFrame()` command.

```
north_data <- data.frame(
    ACER_name = c("North Sweden"),
```

```
    direction = c("North"),
    is_EU = c(TRUE)
)
north_sweden_spdf <- SpatialPolygonsDataFrame(grouped_north_sweden, data = north_data)
```

In case you did not do a `rgeos::gUnaryUnion` and still have the `SpatialPolygonsDataFrame`, you can add information in the way you would normally add an attribute to a `data.frame`, using the $ operator.

```
V4_countries$eic_2letter_code <- c("CZ", "HU", "PL", "SK")
```

NOTE: when adding a data frame back to your `SpatialPolygons`, be sure that the info you add is in the same order as the geometries in your `SpatialPolygons`. If you have filtered geometries using the steps in section 2.2 for example, it might be that the order has changed. You can match your info with the geometries using matching commands, such as `left_join`.

3 Exporting formats

The final step is to export your file. There are many ways to do this and you can export to a lot of different file types. To export as GeoJSON I found the following way the fastest and results in a smaller file than with other methods I tried. First we convert the `SpatialPolygonsDataFrame` into a GeoJSON object in R. Then we export this GeoJSON object to an external file.

```
europe_geojson <- geojson_json(Europe_regions_with_information)
# write to file. I'm using a subdirectory of the working directory called "maps"
geojson_write(europe_geojson, file = paste0(getwd(), "maps/my_europe_map.geojson"))
```

Using some of the links provided at the end of the document, you can export to many other formats and using different commands, too. Pick the method which works best for your purpose!

4 Making an Example map

Now we can use what we've learned and apply it to a real-world example.

Let's say we take an arbitrary division of Europe into regions, such as the one displayed in figure 9, and want to recreate those regions. The code below goes through all the steps discussed before in sequence and plots the result.

Figure 9: Example of a certain division of Europe's countries into regions

```r
# Importing rnaturalearth data
europe_countries <- ne_countries(continent = "Europe", scale = "medium")

# Making a list of which country should be included in which region
list_north = c("Norway", "Sweden", "Denmark", "Finland", "Iceland")
list_centre_east = c("Estonia", "Latvia", "Lithuania", "Poland", "Germany", "Luxembourg", "
    Switzerland", "Austria", "Slovenia", "Croatia", "Hungary", "Slovakia", "Czech Republic")
list_south_east = c("Romania", "Bulgaria", "Greece", "Albania", "Montenegro", "Bosnia and
    Herzegovina", "Serbia", "Kosovo", "Macedonia", "Moldova")
list_south = c("Italy", "Spain", "Portugal")
list_west = c("France", "Belgium", "Netherlands", "United Kingdom", "Ireland")

# Subsetting the regions based on the previously defined lists
countries_north <- europe_countries[europe_countries$name_long %in% list_north, ]
countries_centre_east <- europe_countries[europe_countries$name_long %in% list_centre_east,
    ]
countries_south_east <- europe_countries[europe_countries$name_long %in% list_south_east, ]
countries_south <- europe_countries[europe_countries$name_long %in% list_south, ]
countries_west <- europe_countries[europe_countries$name_long %in% list_west, ]

# Dissolving internal boundaries in the regions
region_north <- rgeos::gUnaryUnion(countries_north)
region_centre_east <- rgeos::gUnaryUnion(countries_centre_east)
region_south_east <- rgeos::gUnaryUnion(countries_south_east)
region_south <- rgeos::gUnaryUnion(countries_south)
region_west <- rgeos::gUnaryUnion(countries_west)

# Combining the regions
regions_combined <- raster::bind(region_north, region_centre_east, region_south_east,
    region_south, region_west)

regions_data <- data.frame(
  region_name = c("Northern Europe", "Central and Eastern Europe", "South-eastern Europe", "
    Southern Europe", "Western Europe"),
  region_letter = c("N", "CE", "SE", "S", "W")
)
```

```r
regions_europa <- SpatialPolygonsDataFrame(regions_combined, data = regions_data)

# Plot the result
plot(europe_countries,
     border = "gray70",
     xlim = c(-10, 30),
     ylim = c(35, 70)
)
plot(regions_europa,
     col = c("green", "cyan", "orange", "red", "purple"),
     add = TRUE
)
```

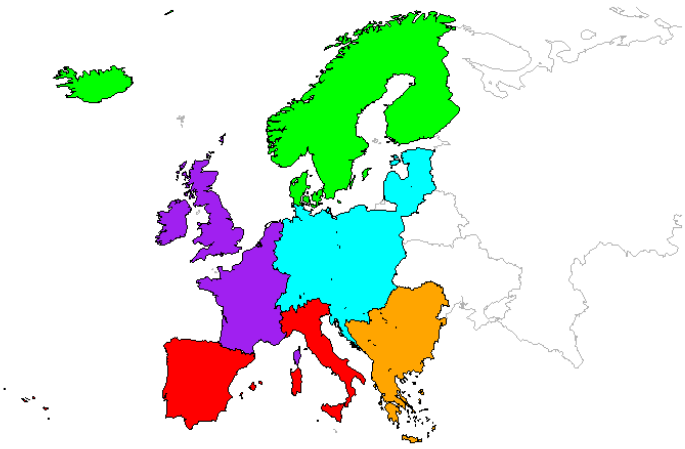


Figure 10: The final result of the executed code

5 Final words

Now you should have all the tools necessary to make your own map files. Of course this guide only covers making the map shapes. Actually producing good-looking maps is a completely different topic ;-). The packages and methods used in this guide are just one way of achieving custom shape maps, but there are plenty of other packages which offer similar commands for working with geospatial data. For example, there exists other packages which avoid the use of SpatialPolygons altogether, while others use the same underlying "engine" but package their own commands.

Further reading links

- CRAN Overview: Analysis of Spatial Data in R. Also for background on the used packages in this guide. https://cran.r-project.org/web/views/Spatial.html
- Introduction to *Simple Features*, another way of representing spatial data which integrates well with the "tidy" packages of R. https://mhallwor.github.io/_pages/Tidyverse_intro