Getting spatial in R

Soil Security Laboratory

2017

R has a very rich capacity to work with, analyse, manipulate and map spatial data. Many procedures one would carry out in a GIS software, can more-or-less be performed relatively easy in R. The application of spatial data analysis in R is well documented in Bivand et al. (2008). Naturally, in DSM, we constantly work with spatial data in one form or another e.g., points, polygons, rasters. We need to to such things as import, view, and export points to, in, and from a GIS. Similarly for polygons and rasters. In this chapter we will cover the fundamentals for doing these basic operations as they are very handy skills, particularly if we want to automate procedures.

Many of the functions used for working with spatial data do not come with the base function suite installed with the R software. Thus we need to use specific functions from a range of different contributed R packages. Probably the most important and most frequently used are:

`sp`	contains many functions for handling vector (polygon) data.
`raster`	very rich source of functions for handling raster data.
`rgdal`	function for projections and spatial data I/O.

Consult the help files and online documentation regarding these packages, and you will quickly realize that we are only scratching the surface of what spatial data analysis functions these few packages are able to perform.

1 Basic GIS operations using R

1.1 Points

We will be working with a small data set of soil information that was collected from the Hunter Valley, NSW in 2010 called `HV100`. This data set is contained in the `ithir` package. So first load it in:

```
library(ithir)
data(HV100)
str(HV100)
```

```
## 'data.frame': 100 obs. of  6 variables:
##  $ site: Factor w/ 100 levels "a1","a10","a11",..: 1 2 3 4 5 6 7 8 9 10 ...
##  $ x   : num  337860 344060 347035 338235 341760 ...
```

```
##  $ y   : num  6372416 6376716 6372741 6368766 6366016 ...
##  $ OC  : num  2.03 2.6 3.42 4.1 3.04 4.07 2.95 3.1 4.59 1.77 ...
##  $ EC  : num  0.129 0.085 0.036 0.081 0.104 0.138 0.07 0.097 0.114 0.031 ...
##  $ pH  : num  6.9 5.1 5.9 6.3 6.1 6.4 5.9 5.5 5.7 6 ...
```

Now load the necessary R packages (you may have to install them onto your computer first):

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

Using the `coordinates` function from the `sp` package we can define which columns in the data frame refer to actual spatial coordinates— here the coordinates are listed in columns `x` and `y`.

```
coordinates(HV100) <- ~x + y
str(HV100)
```

```
## Formal class 'SpatialPointsDataFrame' [package "sp"] with 5 slots
##   ..@ data       :'data.frame': 100 obs. of  4 variables:
##   .. ..$ site: Factor w/ 100 levels "a1","a10","a11",..: 1 2 3 4 5 6 7 8 9 10 ...
##   .. ..$ OC  : num [1:100] 2.03 2.6 3.42 4.1 3.04 4.07 2.95 3.1 4.59 1.77 ...
##   .. ..$ EC  : num [1:100] 0.129 0.085 0.036 0.081 0.104 0.138 0.07 0.097 0.114 0.031 .
##   .. ..$ pH  : num [1:100] 6.9 5.1 5.9 6.3 6.1 6.4 5.9 5.5 5.7 6 ...
##   ..@ coords.nrs : int [1:2] 2 3
##   ..@ coords     : num [1:100, 1:2] 337860 344060 347035 338235 341760 ...
##   .. ..- attr(*, "dimnames")=List of 2
##   .. .. ..$ : chr [1:100] "1" "2" "3" "4" ...
##   .. .. ..$ : chr [1:2] "x" "y"
##   ..@ bbox       : num [1:2, 1:2] 335160 6365091 350960 6382816
##   .. ..- 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 NA
```

Note now that by using the `str` function, the class of `HV100` has now changed from a `dataframe` to a `SpatialPointsDataFrame`. We can do a spatial plot of these points using the `spplot` plotting function in the `sp` package. There are a number of plotting options available, so it will be helpful to consult the help file. Here we are plotting the SOC concentration observed at each location (Figure 1).

```
spplot(HV100, "OC", scales = list(draw = T), cuts = 5,
col.regions = bpy.colors(cutoff.tails = 0.1,
    alpha = 1), cex = 1)
```

The `SpatialPointsDataFrame` structure is essentially the same data frame, except that additional "spatial" elements have been added or partitioned into slots. Some important ones being the bounding box (sort of like the spatial extent of the data), and the coordinate reference system `(proj4string)`,

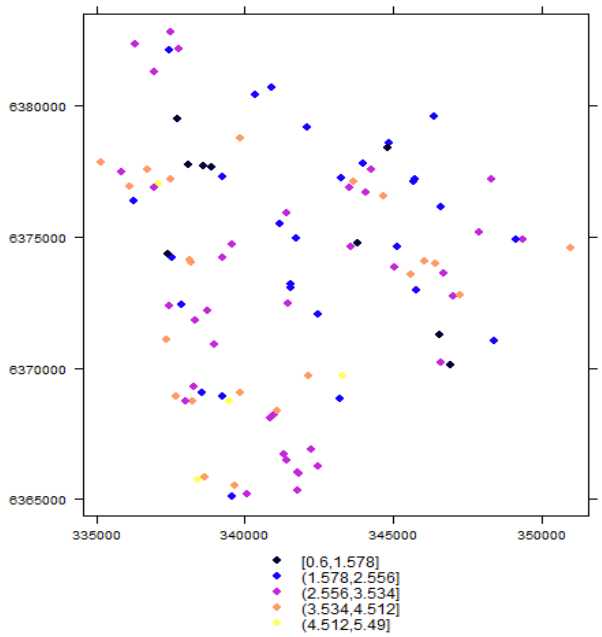

Figure 1: A plot of the site locations with reference to SOC concentration for the 100 points in the `HV100` data set

which we need to define for our data set. To define the CRS, we have to know some things about where our data are from, and what was the corresponding CRS used when recording the spatial information in the field. For this data set the CRS used was WGS1984 UTM Zone 56. To explicitly tell `R` this information we define the CRS as a character string which describes a reference system in a way understood by the PROJ.4 projection library `http://trac.osgeo.org/proj/`. An interface to the PROJ.4 library is available in the `rgdal` package. Alternative to using Proj4 character strings, we can use the corresponding yet simpler EPSG code (European Petroleum Survey Group). `rgdal` also recognizes these codes. If you are unsure of the Proj4 or EPSG code for the spatial data that you have, but know the CRS, you should consult `http://spatialreference.org/` for assistance. The EPSG code for texttttWGS1984 UTM Zone 56 is: 32556. So lets define to CRS for this data.

```
proj4string(HV100) <- CRS("+init=epsg:32756")
HV100@proj4string
```

```
## CRS arguments:
##  +init=epsg:32756 +proj=utm +zone=56 +south +datum=WGS84 +units=m
## +no_defs +ellps=WGS84 +towgs84=0,0,0
```

We need to define the CRS so that we can perform any sort of spatial analysis. For example, we may wish to use these data in a GIS environment such as Google Earth, ArcGIS, SAGA GIS etc. This means we need to export the `SpatialPointsDataFrame` of `HV100` to an appropriate spatial data format (for vector data) such as a shapefile or KML. `rgdal` is again used for this via the `writeOGR` function. To export the data set as a shapefile:

```
writeOGR(HV100, ".", "HV_dat_shape", "ESRI Shapefile")
# Check yor working directory for presence of this file
```

Note that the object we wish to export needs to be a `spatial points data frame`. You should try opening up this exported shapefile in a GIS software of your choosing.

To look at the locations of the data in Google Earth, we first need to make sure the data is in the WGS84 geographic CRS. If the data is not in this CRS (which is not the case for this data), then we need to perform a coordinate transformation. This is facilitated by using the `spTransform` function in `sp`. The EPSG code for WGS84 geographic is: 4326. We can then export out our transformed HV100 data set to a KML file and visualize it in Google Earth.

```
HV100.ll <- spTransform(HV100, CRS("+init=epsg:4326"))
writeOGR(HV100.ll, "HV100.kml", "ID", "KML")
# Check yor working directory for presence of this file
```

Sometimes to conduct further analysis of spatial data, we may just want to import it into R directly. For example, read in a shapefile (this includes both points and polygons too). So lets read in that shapefile that was created just before and saved to the working directory "`HV_dat_shape.shp`":

```
imp.HV.dat <- readOGR(".", "HV_dat_shape")
```

```
## OGR data source with driver: ESRI Shapefile
## Source: ".", layer: "HV_dat_shape"
## with 100 features
## It has 4 fields
```

```
imp.HV.dat@proj4string
```

```
## CRS arguments:
##  +proj=utm +zone=56 +south +datum=WGS84 +units=m +no_defs
## +ellps=WGS84 +towgs84=0,0,0
```

The imported shapefile is now a `SpatialPointsDataFrame`, just like the `HV100` data that was worked on before, and is ready for further analysis.

1.2 Rasters

Most of the functions needed for handling raster data are contained in the `raster` package. There are functions for reading and writing raster files from and to different raster formats. In DSM we work quite a deal with data in table format and then rasterise this data so that we can make a map. To do

this in R, lets bring in a **data frame**. This could be either from a text-file, but as for the previous occasions the data is imported from the **ithir** package. This data is a digital elevation model with 100m grid resolution, from the Hunter Valley, NSW, Australia.

```
library(ithir)
data(HV_dem)
str(HV_dem)
```

```
## 'data.frame': 21518 obs. of  3 variables:
##  $ X        : num  340210 340310 340110 340210 340310 ...
##  $ Y        : num  6362641 6362641 6362741 6362741 6362741 ...
##  $ elevation: num  177 175 178 172 173 ...
```

As the data is already a raster (such that the row observation indicate locations on a regular spaced grid), but in a table format, we can just use the **rasterFromXYZ** function from **raster**. Also we can define the CRS just like we did with the **HV100** point data we worked with before.

```
r.DEM <- rasterFromXYZ(HV_dem)
proj4string(r.DEM) <- CRS("+init=epsg:32756")
```

So lets do a quick plot of this raster and overlay the **HV100** point locations (Figure 2).

```
plot(r.DEM)
points(HV100, pch = 20)
```

So we may want to export this raster to a suitable format for further work in a standard GIS environment. See the help file for **writeRaster** to get information regarding the supported grid types that data can be exported. For demonstration, we will export our data to ESRI Ascii ascii, as it is a common and universal raster format.

```
writeRaster(r.DEM, filename = "HV_dem100.asc", format = "ascii", overwrite = TRUE)
```

What about exporting raster data to KML file? Here you could use the KML function. Remember that we need to reproject our data because it is in the UTM system, and need to get it to WGS84 geographic. The raster re-projection is performed using the **projectRaster** function. Look at the help file for this function. Probably the most important parameters are **crs**, which takes the CRS string of the projection you want to convert the existing raster to, assuming it already has a defined CRS. The other is **method** which controls the interpolation method. For continuous data, "bilinear" would be suitable, but for categorical, "ngb", (which is nearest neighbor interpolation) is probably better suited. **KML** is a handy function from **raster** for exporting grids to kml format.

```
p.r.DEM <- projectRaster(r.DEM, crs = "+init=epsg:4326", method = "bilinear")
KML(p.r.DEM, "HV_DEM.kml", col = rev(terrain.colors(255)), overwrite = TRUE)
# Check yor working directory for presence of the kml file
```

Now visualize this in Google Earth and overlay this map with the points that were created before.

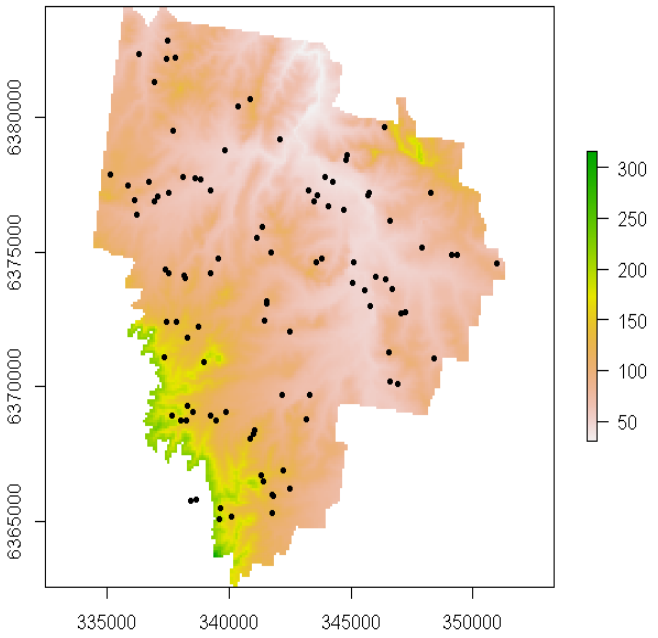

Figure 2: Digital elevation model for the Hunter Valley, overlayed with the `HV100` sampling sites

The other useful procedure we can perform is to import rasters directly into `R` so we can perform further analyses. `rgdal` interfaces with the GDAL library, which means that there are many supported grid formats that can be read into `R` `http://www.gdal.org/formats_list.html`. Here we will load in the "HV_dem100.asc" raster that was made just before.

```
read.grid <- readGDAL("HV_dem100.asc")
```

```
## HV_dem100.asc has GDAL driver AAIGrid
## and has 215 rows and 169 columns
```

The imported raster `read.grid` is a `SpatialGridDataFrame`, which is a formal class of the `sp` package. To be able to use the raster functions from `raster` we need to convert it to the `RasterLayer` class.

```
grid.dem <- raster(read.grid)
grid.dem
```

```
## class       : RasterLayer
## dimensions  : 215, 169, 36335  (nrow, ncol, ncell)
## resolution  : 100, 100  (x, y)
## extent      : 334459.8, 351359.8, 6362591, 6384091  (xmin, xmax, ymin, ymax)
```

```
## coord. ref. : NA
## data source : in memory
## names       : band1
## values      : 29.61407, 315.6837  (min, max)
```

You will notice from the R generated output indicating the data source, it says it is loaded into memory. This is fine for small rasters, but can become a problem when very large rasters need to be handled. A really powerful feature of the `raster` package is the ability to point to the location of a raster/s without the need to load it into memory. It is only very rarely that one needs to use all the data contained in a raster at one time. As will be seen later on this useful feature makes for a very efficient way to perform digital soil mapping across very large spatial extents. To point to the "HV_dem100.asc" raster that was created earlier we would use the following or similar command (where `getwd()` is the function to return the address string of the working directory):

```
grid.dem <- raster(paste(paste(getwd(), "/", sep = ""), "HV_dem100.asc", sep = ""))
grid.dem
```

```
## class       : RasterLayer
## dimensions  : 215, 169, 36335  (nrow, ncol, ncell)
## resolution  : 100, 100  (x, y)
## extent      : 334459.8, 351359.8, 6362591, 6384091  (xmin, xmax, ymin, ymax)
## coord. ref. : NA
## data source : C:\Users\bmalone\Dropbox\2017\DSM_workshopMaterials\HV_dem100.asc
## names       : HV_dem100
```

```
# plot(grid.dem)
```

2 Advanced Work: Creating interactive maps in R

A step beyond creating kml files of your digital soil information is the creation of customized interactive mapping products that can be visualized within your web browser. Interactive mapping makes sharing your data with colleagues simpler, and importantly improves the visualization experience via customization features that are difficult to achieve via the Google Earth software platform. The interactive mapping is made possible via the Leaflet R package. Leaflet is one of the most popular open-source JavaScript libraries for interactive maps. The Leaflet R package makes it easy to integrate and control Leaflet maps in R. More detailed information about Leaflet can be found at `http://leafletjs.com/`, and information specifically about the R package is at `https://rstudio.github.io/leaflet/`.

There is a common workflow for creating Leaflet maps in R. First is the creation of a map widget (calling `leaflet()`); followed by the adding of layers or features to the map by using layer functions (e.g. `addTiles`, `addMarkers`, `addPolygons`) to modify the map widget. The map can then be printed and

visualized in the R image window or saved to HTML file for visualization within a web browser. The following R script is a quick taste of creating an interactive Leaflet map. It is assumed that the `leaflet` and `magrittr` are installed.

```
library(leaflet)
library(magrittr)

leaflet() %>%
addMarkers(lng = 151.210558, lat = -33.852543,
popup = "The view from here is amazing!") %>%
    addProviderTiles("Esri.WorldImagery")
```

You should now see in your plot window a map of an iconic Australian landmark. Interactive features of this map include markers with text, plus ability to zoom and map panning. More will be discussed about the layer functions of the leaflet map further on. What has not been encountered yet is the forward pipe operator %¿%. This operator will forward a value, or the result of an expression, into the next function call or expression. To use this operator the `magrittr` package is required. The example script below shows the same example using and not using the forward pipe operator.

```
# Draw 100 random uniformly distributed numbers between 0 and 1
x <- runif(100)
sqrt(sum(range(x)))

## ..is equivalent to (using forward pipe operator)
x %>% range %>% sum %>% sqrt
```

Sometimes what we want to do in R can get lost within a jumble of brackets, whereas using the forward pipe operator the process of operations is a lot clearer. So lets begin to construct some Leaflet mapping using the data from a little earlier regarding the point (`HV100.ll`) and raster data (`p.r.DEM`). Note that these data have already been converted to WGS coordinate reference system, which is necessary for the creation of the interactive mapping outputs. Firstly, lets create a basic map — example of not using and then using the forward pipe operator.

```
# Basic map without piping operator
addMarkers(addTiles(leaflet()), data = HV100.ll)

# with forward pipe operator
leaflet() %>% addTiles() %>% addMarkers(data = HV100.ll)
```

With the above, we are calling upon a pre-existing base map via the `addTiles()` function. Leaflet supports base maps using map tiles, popularized by Google Maps and now used by nearly all interactive web maps. By default, OpenStreetMap `https://www.openstreetmap.org/#map=13/-33.7760/150.6528&layers=C` tiles are used. Alternatively, many popular free third-party base maps can be added using the `addProviderTiles()` function, which is implemented using the leaflet-providers plugin. For example, previously we used the

`Esri.WorldImagery` base mapping. The full set of possible base maps can be found at `http://leaflet-extras.github.io/leaflet-providers/preview/index.html`. Note that an internet connection is required for access to the base maps and map tiling. The last function used above the the `addMarkers` function, we we simply call up the point data we used previously, which are those soil point observations and measurements from the Hunter Valley, NSW. A basic map will have been created with your plot window. For the next step, lets populate the markers we have created with some of the data that was measured, then add the `Esri.WorldImagery` base mapping.

```
# Populate pop-ups
my_pops <- paste0("<strong>Site: </strong>"
, HV100.ll$site,
"<br>\n  <strong> Organic Carbon (%): </strong>",
HV100.ll$OC,
"<br>\n  <strong> soil pH: </strong>"
, HV100.ll$pH)

# Create interactive map
leaflet() %>% addProviderTiles("Esri.WorldImagery") %>%
addMarkers(data = HV100.ll, popup = my_pops)
```

Further, we can colour the markers and add a map legend. Here we will get the quantiles of the measured SOC% and color the markers accordingly. Note that you will need the colour ramp package `RColorBrewer` installed.

```
library(RColorBrewer)

# Colour ramp
pal1 <- colorQuantile("YlOrBr", domain = HV100.ll$OC)

# Create interactive map
leaflet() %>% addProviderTiles("Esri.WorldImagery") %>%
addCircleMarkers(data = HV100.ll,
    color = ~pal1(OC), popup = my_pops) %>%
addLegend("bottomright", pal = pal1,
    values = HV100.ll$OC, title = "Soil Organic Carbon (%) quantiles"
, opacity = 0.8)
```

It is very worth consulting the help files associated with the `leaflet` R package for further tips on creating further customized maps. The website dedicated to that package, which was mentioned above is also a very helpful resource too.

Raster maps can also be featured in our interactive mapping too, as illustrated in the following script.

```
# Colour ramp
pal2 <- colorNumeric(brewer.pal(n = 9, name = "YlOrBr"),
domain = values(p.r.DEM), na.color = "transparent")
```

```
# interactive map
leaflet() %>% addProviderTiles("Esri.WorldImagery") %>%
addRasterImage(p.r.DEM,
    colors = pal2, opacity = 0.7) %>%
addLegend("topright", opacity = 0.8, pal = pal2,
    values = values(p.r.DEM), title = "Elevation")
```

Lastly, we can create an interactive map that allows us to switch between the different mapping that we have created.

```
# layer switching
leaflet() %>% addTiles(group = "OSM (default)") %>%
addProviderTiles("Esri.WorldImagery") %>%

addCircleMarkers(data = HV100.ll, color = ~pal1(OC),
group = "points", popup = my_pops) %>%
addRasterImage(p.r.DEM, colors = pal2, group = "raster", opacity = 0.8) %>%
addLayersControl(baseGroups = c("OSM (default)", "Imagery"),
overlayGroups = c("points", "raster"))
```

With the created interactive mapping, we can then export these as a web page in HTML format. This can be done via the export menu within the R-Studio plot window, where you want to select the option for "Save as Web page". This file can then be easily shared and viewed by your colleagues.

3 Some R packages that are useful for digital soil mapping

Notwithstanding to the rich statistical and analytical resource provide through the R base functionality, the following R packages (and their contained functions) are what *I* think are an invaluable resource for DSM. As with all things in R, one discovers new tricks all the time, which subsequently means that what functions and analyses are useful now, are superseded or made obsolete later on. There are four main groups of tasks that are critical for implementing DSM in general. These are: (1) Soil science and pedometric type tasks; (2) Using GIS tools and related GIS tasks; (4) Modelling; (4) Making maps, plotting etc. The following are short introductions about those packages that fall into these categories.

Soil science and pedometrics

- aqp: Algorithms for quantitative pedology. http://cran.r-project.org/web/packages/aqp/index.html. A collection of algorithms related to modeling of soil resources, soil classification, soil profile aggregation, and visualization.
- GSIF: Global soil information facility. http://cran.r-project.org/web/packages/GSIF/index.html. Tools, functions and sample datasets for digital soil mapping.

GIS

- sp: http://cran.r-project.org/web/packages/sp/index.html. A package that provides classes and methods for spatial data. The classes document where the spatial location information resides, for 2D or 3D data. Utility functions are provided, e.g. for plotting data as maps, spatial selection, as well as methods for retrieving coordinates, for sub-setting, print, summary, etc.
- raster: http://cran.r-project.org/web/packages/raster/index.html. Reading, writing, manipulating, analyzing and modeling of gridded spatial data. The package implements basic and high-level functions and processing of very large files is supported.
- rgdal: http://cran.r-project.org/web/packages/rgdal/index.html. Provides bindings to Frank Warmerdam's Geospatial Data Abstraction Library (GDAL) (¿= 1.6.3) and access to projection/transformation operations from the PROJ.4 library. Both GDAL raster and OGR vector map data can be imported into R, and GDAL raster data and OGR vector data exported. Use is made of classes defined in the sp package.
- RSAGA: http://cran.r-project.org/web/packages/RSAGA/index.html. RSAGA provides access to geocomputing and terrain analysis functions of SAGA GIS http://www.saga-gis.org/en/index.html from within R by running the command line version of SAGA. RSAGA furthermore provides several R functions for handling ASCII grids, including a flexible framework for applying local functions (including predict methods of fitted models) and focal functions to multiple grids.

Modeling

- caret: http://cran.r-project.org/web/packages/caret/index.html. Extensive range of functions for training and plotting classification and regression models. See the caret website for more detailed information http://topepo.github.io/caret/index.html.
- Cubist: http://cran.r-project.org/web/packages/Cubist/index.html. Regression modeling using rules with added instance-based corrections. Cubist models were developed by Ross Quinlan. Further information can be found at Rulequest https://www.rulequest.com/
- C5.0: http://cran.r-project.org/web/packages/C50/index.html. C5.0 decision trees and rule-based models for pattern recognition. Another model structure developed by Ross Quinlan.
- gam: http://cran.r-project.org/web/packages/gam/index.html. Functions for fitting and working with generalized additive models.
- nnet: http://cran.r-project.org/web/packages/nnet/index.html. Software for feed-forward neural networks with a single hidden layer, and

for multinomial log-linear models.

- `gstat`: `http://cran.r-project.org/web/packages/gstat/`. Variogram modelling; simple, ordinary and universal point or block (co)kriging, sequential Gaussian or indicator (co)simulation; variogram and variogram map plotting utility functions. A related and useful package is `automap` (`http://cran.r-project.org/web/packages/automap/index.html`), which performs an automatic interpolation by automatically estimating the variogram and then calling `gstat`.

Mapping and plotting

- Both `raster` and `sp` have handy functions for plotting spatial data. Besides using the base plotting functionality, another useful plotting package is `ggplot2` (`http://cran.r-project.org/web/packages/ggplot2/index.html`). This package is an implementation of the grammar of graphics in `R`. It combines the advantages of both base and lattice graphics: conditioning and shared axes are handled automatically, and you can still build up a plot step by step from multiple data sources. It also implements a sophisticated multidimensional conditioning system and a consistent interface to map data to aesthetic attributes. See the ggplot2 website for more information, documentation and examples (`http://ggplot2.org/`).

References

Bivand, R. S., E. J. Pebesma, and V. Gomez-Rubio
2008. *Applied Spatial Data Analysis with R.* UseR! Series, Springer.