Applied geostatistics
Exercise 9: R and GIS

D G Rossiter
University of Twente, Faculty of Geo-Information Science & Earth Observation (ITC)

August 5, 2010

Contents

1 Introduction **1**

2 Projections and coördinate systems **2**
2.1 Spatial objects without coördinate systems 2
2.2 Specifying a coördinate reference system 4
2.2.1 Specifying a CRS with the EPSG database 4
2.2.2 * Specifying a CRS directly 6
2.2.3 Transformation of CRS 7
2.2.4 * Measurements on the elipsoid 8

3 Creating GoogleEarth layers **9**
3.1 Placemarks 10
3.2 PNG ground overlays 12
3.2.1 Interpolation grid 12
3.2.2 Kriging interpolation 12
3.2.3 Re-projecting to geographic coördinates 14
3.2.4 Resampling 15

4 Importing and exporting grids **24**
4.1 ESRI ASCII grids 24
4.2 Matrices as spatial objects 26

Version 1.2. Copyright © 2010 University of Twente, Faculty ITC. All rights reserved. Reproduction and dissemination of the work as a whole (not parts) freely permitted if this original copyright notice is included. Sale or placement on a web site where payment must be made to access this document is strictly prohibited.
To adapt or translate please contact the author (http://www.itc.nl/personal/rossiter).

5 Answers **29**

References **32**

Index of R concepts **33**

子曰:'三人行、必有我师焉、择其善者而从之、其不善者而改之'
– 论语 7.22
(Zǐ yuē, sān rén xíng, bìyǒu wǒ shī yān, zé qí shàn zhě ér cóng zhī, qí búshàn zhě ér gǎi zhī)
"When I walk along with two others, they surely can serve as teachers. I will select their good qualities and follow them, their bad qualities and avoid them."
– The Analects, 7.22

This exercise shows some ways to integrate R with GIS for spatial analysis, and for carrying out typical GIS operations directly within R. An excellent text, with many more applications, is "Applied Spatial Data Analysis with R" (ASDAR) by Bivand, Pebesma, and Gómez-Rubio [2] in the Springer UseR! series; here we can only scratch the surface. We've tried to explain some of the steps in more detail here than in the ASDAR book, to make these notes perhaps more suitable for a first exposure.

Note: The code in these exercises was tested with Sweave [6, 7] on R version 2.10.1 (2009-12-14), `sp` package Version: 0.9-52, `gstat` package Version: 0.9-66, `lattice` package Version: 0.17-26, `maptools` package Version: 0.7-29, `rgdal` package Title: Bindings for the Geospatial Data Abstraction Library, running on Mac OS X 10.5.6. So, the text and graphical output you see here was automatically generated and incorporated into LaTeX by running actual code through R and its packages. Then the LaTeX document was compiled into the PDF version you are now reading. Your output may be slightly different on different versions and on different platforms.

1 Introduction

Spatial data within R is a rapidly-developing field. Of particular importance is the `sp` package, which is an integrating framework for spatial data. We have used this extensively in the previous exercises. This package has a so-called *vignette* (small explanation), which can be displayed with the `vignette` command.

Task 1 : Display the vignette for the `sp` package. •

```
> vignette("sp")
```

Other relevant packages covered in this exercise include:

- the `rgdal` package for GDAL-standard data access to geographic data sources;
- the `maptools` package for direct access to some common geographic data formats (e.g. shapefiles), creation of topology, and conversion between R formats.
- the `gstat` package for geostatistical analysis;
- the `lattice` graphics system, which is used by `sp` for much of its graphical output.

Other packages of interest, but not covered here, include:

- the `raster` package for creating rasters (grids) and doing GIS map algebra operations on them.

Task 2 : Load the relevant spatial packages. •

Packages are loaded, if not already in the environment, with the`require` command:

```
> require(sp)
> require(rgdal)
> require(maptools)
> require(gstat)
> require(lattice)
```

2 Projections and coördinate systems

To this point in the exercises we have treated coördinates as metric numbers on a Euclidean plane. You may have noticed an unused slot in spatial objects, namely `proj4string`. We now explain how this is used to specify coördinates. These must be related to to the Earth's surface by some *projection* used to render the surface of an elipsoid as a plane; in addition the original of the system and the assumed figure of the earth must be given; these together are called the *datum*.

> **Note:** If you are unfamiliar with the concepts of projections and datums, they are explained in many surveying and geodesy texts, and in more detail in various works by Snyder and colleagues [e.g. 11, 3]. An excellent practical introduction to the concept of coordinate reference systems, with many explanatory colour figures, is by Iliffe and Lott [5]. A similar recent work is by van Sickle [12].

2.1 Spatial objects without coördinate systems

We will use as an example the famous Meuse River soil pollution dataset from the southern part of the Netherlands. This will have some complications as we attempt to get proper coördinate reference information; although these problems are specific to the Netherlands, they well illustrate the kind of detective work you will ususally have to do with a legacy dataset.

> **Note:** It is very important to fully understand the coördinate systems for the spatial objects in a project. Failure to bring different sources to a common system will usually cause gross errors in any analysis.

Task 3 : Load the Meuse River soil pollution dataset (supplied with the `sp` package), convert it to a spatial object, and examine its structure. •

According to the on-line documentation[1], the coördinates are in fields `x` and `y`:

[1] `?meuse; help(meuse)`

```
> data(meuse)
> coordinates(meuse) = ~x + y
> head(coordinates(meuse))

          x      y
[1,] 181072 333611
[2,] 181025 333558
[3,] 181165 333537
[4,] 181298 333484
[5,] 181307 333330
[6,] 181390 333260

> str(meuse)

Formal class 'SpatialPointsDataFrame' [package "sp"] with 5 slots
  ..@ data       :'data.frame':        155 obs. of  12 variables:
  .. ..$ cadmium: num [1:155] 11.7 8.6 6.5 2.6 2.8 3 3.2 2.8 2.4 1.6 ...
  .. ..$ copper : num [1:155] 85 81 68 81 48 61 31 29 37 24 ...
  .. ..$ lead   : num [1:155] 299 277 199 116 117 137 132 150 133 80 ...
  .. ..$ zinc   : num [1:155] 1022 1141 640 257 269 ...
  .. ..$ elev   : num [1:155] 7.91 6.98 7.8 7.66 7.48 ...
  .. ..$ dist   : num [1:155] 0.00136 0.01222 0.10303 0.19009 0.27709 ...
  .. ..$ om     : num [1:155] 13.6 14 13 8 8.7 7.8 9.2 9.5 10.6 6.3 ...
  .. ..$ ffreq  : Factor w/ 3 levels "1","2","3": 1 1 1 1 1 1 1 1 1 1 ...
  .. ..$ soil   : Factor w/ 3 levels "1","2","3": 1 1 1 2 2 2 2 1 1 2 ...
  .. ..$ lime   : Factor w/ 2 levels "0","1": 2 2 2 1 1 1 1 1 1 1 ...
  .. ..$ landuse: Factor w/ 15 levels "Aa","Ab","Ag",..: 4 4 4 11 4 11 4 2 2 15 ...
  .. ..$ dist.m : num [1:155] 50 30 150 270 380 470 240 120 240 420 ...
  ..@ coords.nrs : int [1:2] 1 2
  ..@ coords     : num [1:155, 1:2] 181072 181025 181165 181298 181307 ...
  .. ..- attr(*, "dimnames")=List of 2
  .. .. ..$ : NULL
  .. .. ..$ : chr [1:2] "x" "y"
  ..@ bbox       : num [1:2, 1:2] 178605 329714 181390 333611
  .. ..- attr(*, "dimnames")=List of 2
  .. .. ..$ : chr [1:2] "x" "y"
  .. .. ..$ : chr [1:2] "min" "max"
  ..@ proj4string:Formal class 'CRS' [package "sp"] with 1 slots
  .. .. ..@ projargs: chr NA
```

Q1 : *(1) Which slot is designed to hold information on the projection and datum?*

(2) What are the units of measure of the coördinates? *Jump to A1* •

The `proj4string` function retrieves the current projection:

```
> proj4string(meuse)

[1] NA
```

Q2 : *What is the listed projection and datum?* *Jump to A2* •

In this case, the coördinates are just numbers, with no specified projection or datum.

2.2 Specifying a coördinate reference system

The listed coördinates must come from some projection and datum together referred to as a a **coordinate reference system**, abbreviated CRS. The `sp` package provides the `proj4string` method to specify the CRS for opjects of any spatial class. Such a CRS object is created by calls to the `CRS` method, with one argument: a valid string of PROJ.4 arguments. These are documented in three manuals[2] provided by the PROJ.4 project.

> **Note:** PROJ.4 is an open-source of library of projection functions[3]; it is not part of R or the `sp` package, and may be used stand-alone or linked in another program.

PROJ.4 arguments must be entered **exactly** as in the PROJ.4 documentation, in particular there cannot be any white space in `+<arg>=<value>` strings, and successive such strings can only be separated by blanks.

There are two ways to specify a CRS:

1. With a reference to the EPSG database (§2.2.1);
2. Directly from the parameters (§2.2.2).

The first is much simpler, and we will explain it first. If you can find the CRS in the EPSG database (explained next), this is quicker and more reliable. If not, you will have to try the second method.

2.2.1 Specifying a CRS with the EPSG database

In our example, the metadata[4] tell us that the coördinates are in the RDH (Rijksdriehook = Dutch triangulation) CRS.

The parameters of this and many other systems are included in the European Petroleum Survey Group (**EPSG**) database[5], which is an attempt to collect and verify information on all the projection systems in use now and in the past. The database is supplied with the `rgdal` package[6], in file `library/rgdal/proj/epsg` in the R installation[7].

> **Note:** The `rgdal` package is an R interface to the open-source Geospatial Data Abstraction Library (GDAL)[8], which presents a single abstract data model for a large variety of spatial data.

The advantage of this approach is that you only need to know the reference number in the EPSG database, and the parameters are then all set automatically.

[2] `OF90-284.pdf`, `proj.4.3.pdf` and `proj.4.3.I2.pdf`
[3] http://trac.osgeo.org/proj/
[4] `?meuse`
[5] now maintained by the International Association of Oil & Gas producers (OGP)
[6] The latest version is also available on-line and as an MS-Access databse, at http://www.epsg.org
[7] if the `rgdal` package is installed, of course
[8] http://www.gdal.org/

Task 4 : Load the EPSG database into R and search for the "Amersfoort / RD" CRS. •

We use the `make_EPSG` utility function of the `rgdal` package to load the definitions into a list, and then search with `grep` for the string `Amersfoort`, which is the name of the origin of the Rijksdriehook (Dutch triangulation) coördinate system[9].

```
> EPSG <- make_EPSG()
> (EPSG[grep("Amersfoort", fixed = T, EPSG$note), ])
> rm(EPSG)

      code                  note
2985 28991 # Amersfoort / RD Old
2986 28992 # Amersfoort / RD New
                                                                    prj4
2985 +proj=sterea +lat_0=52.15616055555555 +lon_0=5.38763888888889
+k=0.9999079 +x_0=0 +y_0=0 +ellps=bessel +units=m +no_defs
2986 +proj=sterea +lat_0=52.15616055555555 +lon_0=5.38763888888889
+k=0.9999079 +x_0=155000 +y_0=463000 +ellps=bessel +units=m +no_defs
```

We see that there are two systems, "RD Old" and "RD New". For information on these we refer to the review of coordinate systems in the Netherlands by Mugnier [9]; this is one of a long series of "Grids & Datums" articles[10]; it should be your first stop for information on a country's coordinate systems.

From this article we see that the new system differs from the old only in the location of (0,0) point in the English Channel (instead of at Amersfoort); there are thus new false orgins, i.e. the `+x_0` and `+y_0` parameters.[11]

The list shows that the RD new system is EPSG reference 28992[12]

Task 5 : Set the projection information for the Meuse dataset to EPSG reference 28992. •

This is easily done with the `CRS` method, using a single argument string, `+init=`, i.e. "initialize" the entire CRS from the given EPSG entry.

```
> proj4string(meuse) <- CRS("+init=epsg:28992")
> proj4string(meuse)
> head(coordinates(meuse))

[1] " +init=epsg:28992 +proj=sterea +lat_0=52.15616055555555
+lon_0=5.38763888888889 +k=0.9999079 +x_0=155000 +y_0=463000 +ellps=bessel
+units=m +no_defs"
```

[9] the centre point of the projection is the top of the Onze Lieve Vrouwetoren in the city of Amersfoort

[10] http://www.asprs.org/resources/grids/ in *Photogrammetric Engineering & Remote Sensing*

[11] This ensures that within the national territory all East coördinates are less than 280 000 m, and all North coördinates greater than than 300 000 m; thus there can never be a confusion between N and E coordinates.

[12] This information is also found in the help text for the `writeOGR` method, where it is used as an example.

Note that the `+init` parameter is as we specified it, but this has been expanded with the list of parameters (`+proj`, `+lat_0` etc.) for this system, according to the EPSG database.

Q3 : *(1) Were the coördinates changed?*

(2) What are now the units of measure of the coördinates? *Jump to A3* •

Unfortunately, the EPSG is not (as of this writing) up-to-date; in particular it is missing a key issue: the offset of the centre of the Earth in the RD system to that in the "WGS84" CRS. This will not affect conversion to geographic coördinates in terms of the RDH's elipsoid, but will cause serious errors, on the order of 100's of meters, when converting to the WGS84 CRS used as a default by most consumer GPS receivers and Google Earth (§3).

The Netherlands Geodetic Commission [4] has published the following adjustment, which requires six parameters (see next §2.2.2 for details); they are also shown in an example in the ASDAR book [2, §4.3.1].

```
+towgs84=565.237,50.0087,465.658,-0.406857,0.350733,-1.87035,4.0812
```

Task 6 : Add the WGS84 displacement to the CRS definition for the Meuse dataset. •

We retrieve the current CRS with the `proj4string` method, paste it together with the additional information with the `paste` function, and then format the combined string into a CRS with the `CRS` method:

```
> proj4string(meuse) <- CRS(paste(proj4string(meuse),
+ "+towgs84=565.237,50.0087,465.658,-0.406857,0.350733,-1.87035,4.0812"))
> proj4string(meuse)

[1] " +init=epsg:28992 +proj=sterea +lat_0=52.15616055555555
+lon_0=5.38763888888889 +k=0.9999079 +x_0=155000 +y_0=463000 +ellps=bessel
+units=m +no_defs
+towgs84=565.237,50.0087,465.658,-0.406857,0.350733,-1.87035,4.0812"
```

2.2.2 * Specifying a CRS directly

If a CRS can't be found in the EPSG database, you must search for its parameters in some other source, most likely the survey agency responsible for the system. An excellent source of information on many countries and regions is the series of columns "Grids & Datums"[13] in *Photogrammetric Engineering & Remote Sensing* written by Clifford J. Mugnier.

In our example, the Rijksdriehook CRS is extensively documented (more completely than in Mugnier [9]) in a publication of the Netherlands Geodetic Commission [4]. Referring to this document, we discover that this CRS is a:

- stereographic projection (parameter `+proj`) ...

[13] http://www.asprs.org/resources/grids/

- on the Bessel ellipsoid (parameter `+ellps`) ...
- with a fixed origin (parameters `+lat_0` and `+lon_0`), and
- scale factor at the tangency point (parameter `+k`).
- In addition, the coördinate system has a false origin (parameters `+x_0` and `+y_0`).
- The centre of the ellipsoid is displaced with respect to the standard WGS84 ellipsid (parameter `+towgs84`, with three distances, three angles, and one scale factor).

Task 7 : Set the correct projection information for the Meuse dataset and check it, •

To both set and check the projection, we use the `proj4string` method, setting up the CRS from a text string with the `CRS` method:

```
> proj4string(meuse) <- CRS("+proj=stere
+     +lat_0=52.15616055555555 +lon_0=5.38763888888889
+     +k=0.999908 +x_0=155000 +y_0=463000
+     +ellps=bessel +units=m +no_defs
+     +towgs84=565.2369,50.0087,465.658,
+       -0.406857330322398,0.350732676542563,-1.8703473836068,
+       4.0812")
```

Note: Note that these `towgs84` parameters are more precise than those from the ASDAR book [2], which are:

```
+towgs84=565.237,50.0087,465.658,-0.406857,0.350733,-1.87035,4.0812
```

2.2.3 Transformation of CRS

The `rgdal` package provides methods to convert between projections and datums; this dual process we can call "transformation" between CRS. The most general is the generic `spTransform` method. It works on any spatial class with a defined CRS; the second argument is an object of class `CRS`, specifying the target CRS.

Task 8 : Transform the Meuse dataset from the Rijksdriehook CRS to geographic coördinates on the WGS84 elipsoid. •

Very important:

- To force `spTransform` to do a datum transformation, the target CRS **must** include the `+datum` parameter. Otherwise only a reprojection will be done, but on the same datum.
- In almost all cases the non-WGS84 system should specify a `+towgs84` parameter, with either 3 (origin shift) or 7 (origin shift + rotation) values.

```
> meuse.wgs84 <- spTransform(meuse, CRS("+proj=longlat +datum=WGS84"))
> proj4string(meuse.wgs84)
```

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

```
> head(coordinates(meuse.wgs84))
```

```
             x         y
[1,] 5.7585603 50.991526
[2,] 5.7578871 50.991051
[3,] 5.7598796 50.990856
[4,] 5.7617701 50.990374
[5,] 5.7618871 50.988989
[6,] 5.7630641 50.988356
```

Note: Note the projection is `longlat`, not `latlong`! So, E before N, as with metric coördinate systems.

Note: There is also an EPSG refererence for this system, 4326; so this could have been written as:

```
meuse.wgs84 <- spTransform(meuse, CRS("+init=epsg:4326"))
```

The coördinates are now in geographic coordinates on the WGS84 ellipsoid, as required.

Q4 : *What are the units of measure of these transformed coördinates?* *Jump to A4* •

2.2.4 * Measurements on the elipsoid

Spatially-aware methods automatically measure distances along a great-circle on the elipsoid, if the CRS is a geographic (long/lat) system and the elipsoid is known. An example is the `variogram` method of the `gstat` package.

Task 9 : Compute the default variogram on both the elipsoid and RD metric system. •

```
> (v.wgs84 <- variogram(log(zinc) ~ 1, loc = meuse.wgs84))
```

```
    np        dist      gamma dir.hor dir.ver   id
1   57 0.079269952 0.12344793       0       0 var1
2  299 0.163923879 0.21621849       0       0 var1
3  419 0.267284017 0.30278588       0       0 var1
4  457 0.372621281 0.41214476       0       0 var1
5  547 0.478324369 0.46341279       0       0 var1
6  533 0.585151621 0.56469327       0       0 var1
7  574 0.692926159 0.56896826       0       0 var1
8  565 0.796021090 0.61760666       0       0 var1
9  588 0.902937992 0.64822464       0       0 var1
10 543 1.010942686 0.69157049       0       0 var1
11 501 1.117588338 0.70273849       0       0 var1
12 477 1.221119674 0.60381181       0       0 var1
13 451 1.328812070 0.65240318       0       0 var1
14 458 1.436853980 0.56530875       0       0 var1
15 415 1.542891299 0.57480955       0       0 var1
```

```
> (v.rd <- variogram(log(zinc) ~ 1, loc = meuse))

    np        dist      gamma dir.hor dir.ver   id
1   57   79.292437 0.12344793       0       0 var1
2  299  163.973666 0.21621849       0       0 var1
3  419  267.364828 0.30278588       0       0 var1
4  457  372.735422 0.41214476       0       0 var1
5  547  478.476695 0.46341279       0       0 var1
6  533  585.340581 0.56469327       0       0 var1
7  574  693.145256 0.56896826       0       0 var1
8  564  796.183649 0.61867686       0       0 var1
9  589  903.146498 0.64714789       0       0 var1
10 543 1011.291773 0.69157049       0       0 var1
11 500 1117.862346 0.70339835       0       0 var1
12 477 1221.328099 0.60387704       0       0 var1
13 452 1329.164065 0.65171578       0       0 var1
14 457 1437.256203 0.56653178       0       0 var1
15 415 1543.202482 0.57482273       0       0 var1
```

Q5 : *(1) What is the same and what is different in the two variograms?*

(2) Which distances should be larger, on the elipsoid or metric?

(3) Which in fact are larger? Try to explain. *Jump to A5* •

Task 10 : Compute the differences between the average distances of the semivariogram bins, in centimetres. Compute the percent difference relative to the long dimension of the bounding box. •

```
> (v.rd$dist - v.wgs84$dist * 1000) * 100

 [1]  2.2485935  4.9786902  8.0810702 11.4140892 15.2325795 18.8959922
 [7] 21.9096532 16.2558547 20.8505890 34.9086958 27.4007407 20.8424737
[13] 35.1995466 40.2223240 31.1182715

> max(v.rd$dist - v.wgs84$dist * 1000) * 100

[1] 40.222324

> max(v.rd$dist - v.wgs84$dist * 1000)/diff(bbox(meuse)["y",
+     ]) * 100

        max
0.010321356
```

Q6 : *What is the maximum difference, in cm?* *Jump to A6* •

3 Creating GoogleEarth layers

Google Earth uses KML[14] ("Keyhole Markup Language") to specify how to display geographic data. From the web site:

[14] http://code.google.com/apis/kml/documentation/index.html

> "Just as web browsers display HTML files, Earth browsers such as Google Earth display KML files. Once you've properly configured your server and shared the URL (address) of your KML files, anyone who's installed Google Earth can view the KML files hosted on your public web server."

So to display maps created in R, there are five steps, three of which we have already done:

1. Create a map to export (§2.1);
2. Assign projection information to the map (§2.2);
3. Transform to geographic coördinates (Long/Lat) on the WGS84 datum, as required by Google Earth (§2.2.3);
4. Export as a KML file;
5. Open the KML file in Google Earth.

3.1 Placemarks

One type of KML is the **placemark**. It contains the location and (optionally) feature-space attributes for one or more geographic points.

We have a point coverage `meuse.wgs84`, already transformed to the correct CRS. So the next step is to export it.

The `rgdal` package provides the `writeOGR` method to export spatial vector data (e.g. points); one of the supported formats is KML.

Task 11 : Export the point coverage of Meuse Zn values as a KML file. •

We do this with `writeOGR`, specifying the KML driver:

```
> writeOGR(meuse.wgs84["zinc"], "meuseZnPoints.kml", "zinc",
+     driver = "KML")
```

For this next task you must have Google Earth installed.

Task 12 : Open the created KML file to view the points in Google Earth. •

The screen should look something like Figure 1; the points are written with their data values, which can be seen by clicking on them in Google Earth. An example is one of the most polluted sites, at the bend of the river just south of the village of Meers (Figure 2), and on the river side of the dike that protects that village.

Other parameters can be included in calls to `writeOGR`; as with other GDAL drivers, this is documented on the GDAL web page[15], not within R. We see

[15] http://www.gdal.org/ogr/drv_kml.html

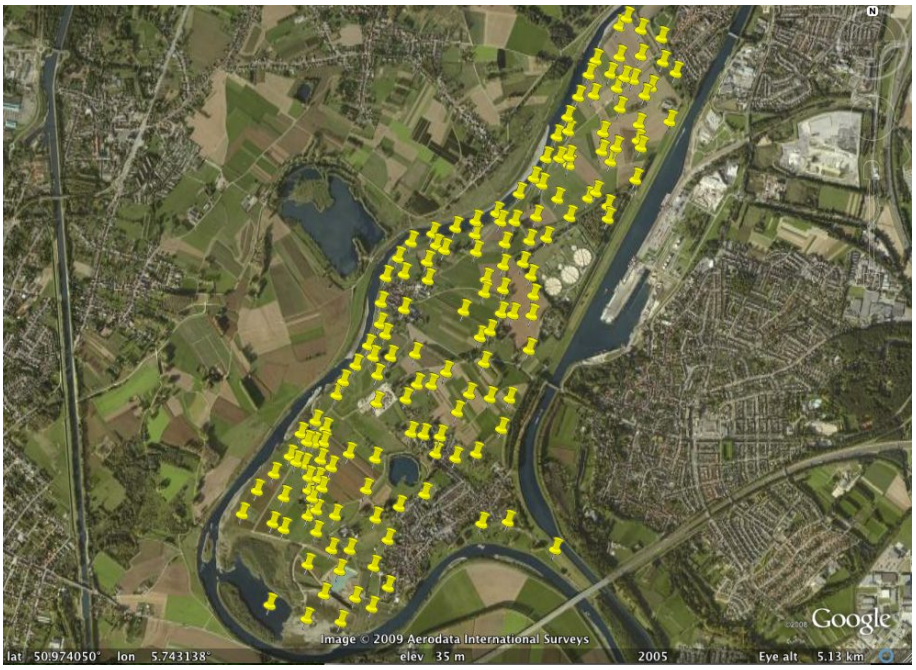


Figure 1: Meuse sample points, shown in Google Earth



Figure 2: Meuse sample points, detail

from this that several layers can be written to the same object and then selected within Google Earth.

3.2 PNG ground overlays

Another type of Google Earth layer is the **ground overlay**. This a graphics image (not a map or GIS layer), in Portable Network Graphics (PNG) format, along with a control KML file that specifies how the graphic is to be placed in Google Earth. So we must write two files: the image and the control.

This task is made possible by methods from the `maptools` package, in particular the `GE_SpatialGrid` method to define the size and position of a PNG image overlay in Google Earth, and the `kmlOverlay` method write the KML file including this PNG image overlay.

3.2.1 Interpolation grid

Task 13 : Create an interpolated map of log(Zn) concentration in the Meuse topsoils, on a regular grid in the original (RD) coördinate system. •

A regular grid has been supplied with the sample data; it is in the same coördinate system as the point data, and is converted to geo-referenced data the same way; we use the shortcut `gridded` method to specify both that the points are on a regular grid, and which fields give their coördinates. We then specify that the CRS is the same here as for the points.

```
> data(meuse.grid)
> gridded(meuse.grid) = ~x + y
> proj4string(meuse.grid) = proj4string(meuse)
```

3.2.2 Kriging interpolation

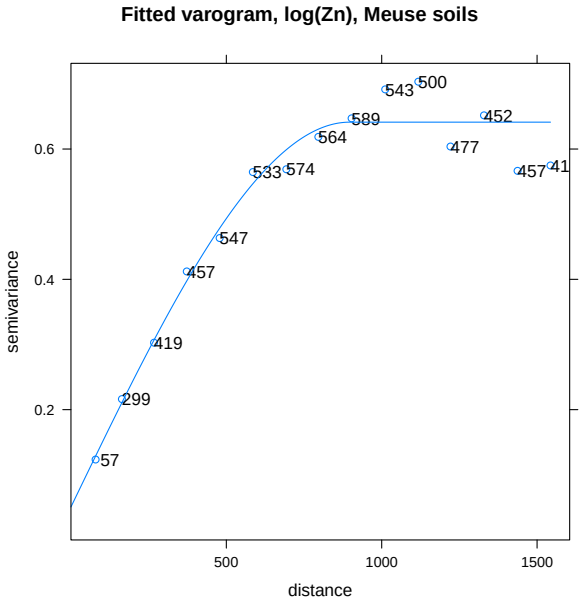
The next task is to interpolate; any method will do for our purposes. But since this variogram shows nice structure we'll model it and do a kriging interpolation.

Task 14 : Compute the empirical variogram for log(Zn) and model it. •

The `variogram` method computes the variogram; the `vgm` specifies an initial modelm and the `fit.variogram` method adjusts it with weighted least-squares:

```
> v <- variogram(log(zinc) ~ 1, loc = meuse)
> max(v$gamma)

[1] 0.70339835

> which(v$gamma == max(v$gamma))

[1] 11
```

```
> v[which(v$gamma == max(v$gamma)), "dist"]

[1] 1117.8623

> (vm <- fit.variogram(v, model = vgm(max(v$gamma), "Sph",
+     v[which(v$gamma == max(v$gamma)), "dist"], 0)))

  model       psill     range
1   Nug 0.050659365   0.00000
2   Sph 0.590604764 896.99857

> print(plot(v, pl = T, model = vm, main = "Fitted varogram, log(Zn), Meuse soils"))
```



Note: The automatic fit began with the initial variogram estimate of 0 nugget, partial sill equal to the maximum semivariance in the empirical variogram, and range equal to the bin average where that maximum semivariance was reached (note the use of `which`). The form of the model, here `"Sph"`, should be known from previous work. The automatic fit does not need very accurate initial values if the form of the variogram is good, as here.

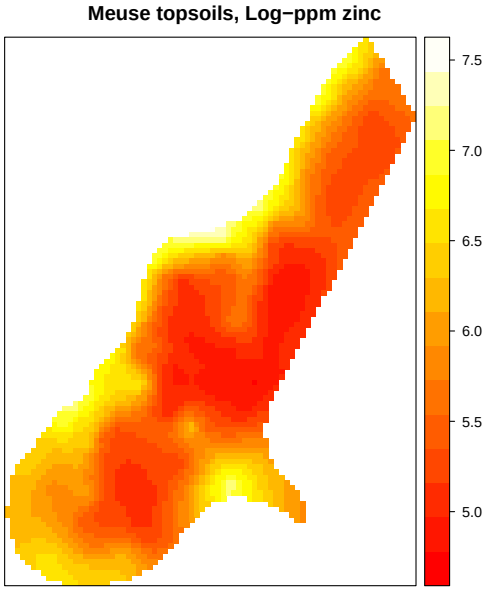
Q7 : *How much did the automatic fit adjust the automatic initial estimates of variogram parameters? How well does the automatic fit match the empirical variogram?* *Jump to A7*

•

Task 15 : Interpolate log(Zn) by ordinary kriging on the regular grid, using the fitted variogram model and the sample points. •

```
> kr <- krige(log(zinc) ~ 1, loc = meuse, newdata = meuse.grid,
+     model = vm)

[using ordinary kriging]
```

```
> print(spplot(kr, zcol = "var1.pred", col.regions = heat.colors(64),
+     main = "Meuse topsoils, Log-ppm zinc"))
```



3.2.3 Re-projecting to geographic coördinates

The interpolation has to be in geographic coördinates to be useful in Google Earth. So, we **re-project** the kriging prediction grid to geographic coördinates; this will take some time, since the new position of every pixel has to be separately computed.

Task 16 : Re-project the kriging prediction grid to geographic coördinates. •

```
> class(kr)

[1] "SpatialPixelsDataFrame"
attr(,"package")
[1] "sp"

> head(coordinates(kr))

          x      y
[1,] 181180 333740
[2,] 181140 333700
[3,] 181180 333700
[4,] 181220 333700
[5,] 181100 333660
[6,] 181140 333660

> kr.wgs84 <- spTransform(kr, CRS("+proj=longlat +datum=WGS84"))
> class(kr.wgs84)

[1] "SpatialPointsDataFrame"
attr(,"package")
[1] "sp"
```

```
> head(coordinates(kr.wgs84))

             x        y
[1,] 5.7601080 50.992680
[2,] 5.7595353 50.992322
[3,] 5.7601051 50.992321
[4,] 5.7606748 50.992319
[5,] 5.7589627 50.991965
[6,] 5.7595324 50.991963
```

Notice that the object was converted to spatial points (pixel centres), i.e. class `SpatialPointsDataFrame`, even though the source grid was of class `SpatialPixelsDataFrame`, i.e. a regular grid. This is because the regular centres on the metric grid are *not* regular centres in geographic coördinates.

Q8 : *How can you tell that the geographic coördinates are not regular centres?* *Jump to A8* •

3.2.4 Resampling

The image overlay has to be on a regular grid, but alligned with WGS84 longitude and latitude. So, we have to set up a regular grid in geographic coördinates and then **resample** to it.

> **Note:** Resampling is explained in remote sensing texts [e.g. 10, 8, 1].

So, the next step is to set up a grid in geographic coördinates (as required by Google Earth) as the target object for export. This is done with the `GE_SpatialGrid` function, which also sets up a container for the PNG file, and so returns information that can be used when opening the PNG graphics device (i.e. the output file). Note that a PNG is just a graphics file for visualization, it is not meant as a precise grid, as was the case with the points KML overlay. So we will sacrifice some precision for ease of visualization.

The resolution of the PNG should more or less match the cell size of the interpolation; this is specified with the `maxPixels` argument to `GE_SpatialGrid`, which gives the number of pixels in the longest dimension. Here we know it's the N-S dimension, which has been transformed but was 78 by 104:

```
> kr@grid@cells.dim

  x   y
 78 104
```

So, we should set up the grid accordingly[16]. A complication here is that the grid must be clipped to the study area; it is not a rectangle. Fortunately, we have the area covered by a grid already (in RD coördinates), which we used for interpolation. This grid has already been converted to class `SpatialPixelsDataFrame`; but `GE_SpatialGrid` expects a single polygon, into which it will create a grid suitable for Google Earth.

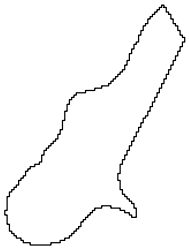
[16] This code adapted from §4.3.2 of Bivand et al. [2]

Task 17 : Create a single polygon surrounding the study area. •

To convert the gridded pixels to a polygon, we first convert them to individual small polygons with the `as` type-cast method, and then find the **topological union** of them with the `unionSpatialPolygons` method of the `maptools` package. The union discards all boundaries that are inside the largest polygon(s) that can be made from a set of polygons; here this will be the outside boundary of the study area, with a jagged edge corresponding to the pixels.

We follow this process with the `class` method to see the evolution of the class, and the `length` method to see how many polygons are in the original and union objects:

```
> class(meuse.grid)

[1] "SpatialPixelsDataFrame"
attr(,"package")
[1] "sp"

> length(meuse.grid@coords[, "x"])

[1] 3103

> grd <- as(meuse.grid, "SpatialPolygons")
> class(grd)

[1] "SpatialPolygons"
attr(,"package")
[1] "sp"

> length(grd@polygons)

[1] 3103

> grd.union <- unionSpatialPolygons(grd, rep("x", length(grd@polygons)))
> class(grd.union)

[1] "SpatialPolygons"
attr(,"package")
[1] "sp"

> length(grd.union@polygons)

[1] 1

> plot(grd.union, main = "Bounding polygon, Meuse interpolation grid")
```



The figure shows the single polygon; the jagged edges show that it was derived by the union of many small polygons (equivalent to single pixels).

Q9 : *What are the classes of the original and union objects, and how many polygons are in each?* *Jump to A9* •

This polygon must now be re-projected to the WGS84 CRS; for this we again use the `spTransform` method, as in §2.2.3.

Task 18 : Re-project the bounding polygon to the WGS84 CRS. •

```
> bbox(grd.union)

      min    max
r1 178440 181560
r2 329600 333760

> grd.union.wgs84 <- spTransform(grd.union, CRS("+proj=longlat +datum=WGS84"))
> bbox(grd.union.wgs84)

          min        max
r1  5.7208453  5.7654809
r2 50.9555501 50.9928609
```

We can see the effect of the transformation in the bounding box, with method `bbox`.

Task 19 : Create a spatial grid, with ancillary information for the Google Earth KML file, covering the study area. •

Now the `GE_SpatialGrid` method can be used. This can accept several kinds of spatial objects as an argument; in the case of `SpatialPolygons` it will fill the bounding box of the polygon with grid cells. The resolution is by default 600 pixels in the maximum dimension; this will result in a very fine overlay and large PNG graphic file. Recall that the dimensions of this

area, divided into 40x40 m pixels, were much smaller; the largest dimension is only 104. To make the image somewhat smooth we'll double this:

```
> GRD.wgs84 <- GE_SpatialGrid(grd.union.wgs84, maxPixels = 2 *
+     max(kr@grid@cells.dim))
> str(GRD.wgs84)

List of 6
 $ height: int 208
 $ width : int 157
 $ SG    :Formal class 'SpatialGrid' [package "sp"] with 5 slots
  .. ..@ grid       :Formal class 'GridTopology' [package "sp"] with 3 slots
  .. .. .. ..@ cellcentre.offset: Named num [1:2] 5.72 50.96
  .. .. .. .. ..- attr(*, "names")= chr [1:2] "x" "y"
  .. .. .. ..@ cellsize         : Named num [1:2] 0.000285 0.000179
  .. .. .. .. ..- attr(*, "names")= chr [1:2] "max" "max"
  .. .. .. ..@ cells.dim        : int [1:2] 157 208
  .. ..@ grid.index : int(0)
  .. ..@ coords     : num [1:2, 1:2] 5.72 5.77 50.96 50.99
  .. .. ..- attr(*, "dimnames")=List of 2
  .. .. .. ..$ : NULL
  .. .. .. ..$ : chr [1:2] "x" "y"
  .. ..@ bbox       : num [1:2, 1:2] 5.72 50.96 5.77 50.99
  .. .. ..- attr(*, "dimnames")=List of 2
  .. .. .. ..$ : chr [1:2] "x" "y"
  .. .. .. ..$ : chr [1:2] "min" "max"
  .. ..@ proj4string:Formal class 'CRS' [package "sp"] with 1 slots
  .. .. .. ..@ projargs: chr " +proj=longlat +datum=WGS84 +ellps=WGS84 +towgs84=0,0,0"
 $ asp   : num 1.33
 $ xlim  : Named num [1:2] 5.72 5.77
  ..- attr(*, "names")= chr [1:2] "min" "max"
 $ ylim  : Named num [1:2] 51 51
  ..- attr(*, "names")= chr [1:2] "min" "max"
 - attr(*, "class")= chr "GE_SG"
```

Notice that `GE_SpatialGrid` computed the proper map aspect from the geographic latitude:

```
> GRD.wgs84$asp

[1] 1.3275158
```

A grid created by `GE_SpatialGrid` has a triple purpose:

1. to contain the graphics file (PNG) to be displayed in Google Earth;
2. to give information for setting up the PNG graphics device in R, i.e. the number of pixels;
3. to provide display information for Google Earth, i.e. the geographic location (bounding box) and the aspect ratio.

We can see this triple role in its structure:

```
> str(GRD.wgs84)

List of 6
 $ height: int 208
 $ width : int 157
 $ SG    :Formal class 'SpatialGrid' [package "sp"] with 5 slots
```

```
  .. ..@ grid       :Formal class 'GridTopology' [package "sp"] with 3 slots
  .. .. .. ..@ cellcentre.offset: Named num [1:2] 5.72 50.96
  .. .. .. .. ..- attr(*, "names")= chr [1:2] "x" "y"
  .. .. .. ..@ cellsize         : Named num [1:2] 0.000285 0.000179
  .. .. .. .. ..- attr(*, "names")= chr [1:2] "max" "max"
  .. .. .. ..@ cells.dim        : int [1:2] 157 208
  .. ..@ grid.index : int(0)
  .. ..@ coords     : num [1:2, 1:2] 5.72 5.77 50.96 50.99
  .. .. ..- attr(*, "dimnames")=List of 2
  .. .. .. ..$ : NULL
  .. .. .. ..$ : chr [1:2] "x" "y"
  .. ..@ bbox       : num [1:2, 1:2] 5.72 50.96 5.77 50.99
  .. .. ..- attr(*, "dimnames")=List of 2
  .. .. .. ..$ : chr [1:2] "x" "y"
  .. .. .. ..$ : chr [1:2] "min" "max"
  .. ..@ proj4string:Formal class 'CRS' [package "sp"] with 1 slots
  .. .. .. ..@ projargs: chr " +proj=longlat +datum=WGS84 +ellps=WGS84 +towgs84=0,0,0"
 $ asp   : num 1.33
 $ xlim  : Named num [1:2] 5.72 5.77
  ..- attr(*, "names")= chr [1:2] "min" "max"
 $ ylim  : Named num [1:2] 51 51
  ..- attr(*, "names")= chr [1:2] "min" "max"
 - attr(*, "class")= chr "GE_SG"
```

This is a list with six items: `height` and `width` will be used for the PNG device; `xlim`, `ylim`, and `asp` for the KML file, and `SG` to hold the image.

The grid is created as a rectangle covering the bounding box; we want all the pixels outside the study area to be missing values (i.e. not interpolated). To do this we'll set up a vector of the same length as the grid, with each cell either `1` (cell is inside the study area) or `NA` (cell is outside). This is easily accomplished with the `overlay` method, which when applied to a grid and polygon returns the polygon number in which each grid cell is found:

```
> GRD.wgs84_in <- overlay(GRD.wgs84$SG, grd.union.wgs84)
> str(GRD.wgs84_in)

 int [1:32656] NA NA NA NA NA NA NA NA NA NA ...

> unique(GRD.wgs84_in)

[1] NA  1
```

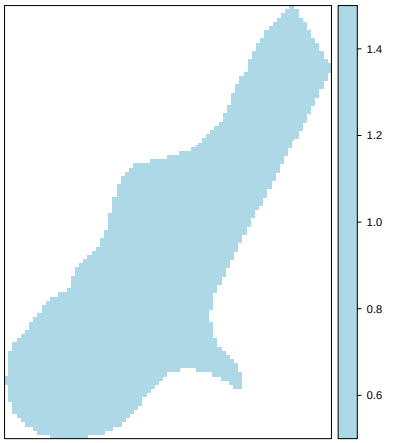
The `unique` function lists the unique values in a vector, here either `1` or `NA`.

Finally, we set up a `SpatialPixelsDataFrame` to receive the resampled values, with the same structure as the Google Earth grid. Note the use of the `as` type-casting method to convert the grid to individual pixels.

Note also the dummy `@data` slot, with one variable (`pred`), initialized to all 1's (the enclosing polygon number from the previous `overlay`). This is necessary for a spatial object with data, such as `SpatialGridDataFrame`. Without this there would be no data frame in which to write the resampled values.

```
> llSGDF <- SpatialGridDataFrame(grid = GRD.wgs84$SG@grid,
+     proj4string = CRS(proj4string(GRD.wgs84$SG)), data = data.frame(pred = GRD.wgs84_in))
```

```
> llSPix <- as(llSGDF, "SpatialPixelsDataFrame")
> print(spplot(llSPix, zcol = "pred", col.regions = "lightblue"))
```



Note that the `SpatialGridDataFrame` constructor method discarded the grid cells outside the bounding polygon, and only retained the grid cells from the source grid (`GRD.wgs84$SG@grid`) that had valid data, i.e. not `NA`. So now the grid covers only the study area.

> **Note:** For a rectangular grid there would be no need for the overlay, and the initial data would be specified by `pred = rep(0, GRD.wgs84$height*GRD.wgs84$width)`, i.e. a vector of any value of length equal to the product of the dimensions.

Now we have to **resample**, i.e. interpolate to the centres of the grid. We know the the `krige` method interpolates, and without a variogram model it defaults to linear inverse distance interpolation. However, the `idw` method allows the specification of an inverse-distance decay power, by default 2 (i.e. inverse-distance squared) which will give a smoother-looking picture[17]. We write the interpolation directly into the `pred` field in the grid's data frame, using the grid as locations to interpolate (`newdata` argument to `idw`).

This interpolation may take some time; we can time this by calling the interpolation inside a call to the `system.time` method:

```
> (st <- system.time(llSPix$pred <- idw(var1.pred ~ 1,
+     loc = kr.wgs84, newdata = llSPix, nmax = 8)$var1.pred))

[inverse distance weighted interpolation]
   user  system elapsed
 17.884   0.046  18.091

> summary(llSPix)

Object of class SpatialPixelsDataFrame
Coordinates:
         min       max
x  5.7208453  5.765571
y 50.9555501 50.992861
Is projected: FALSE
proj4string :
```

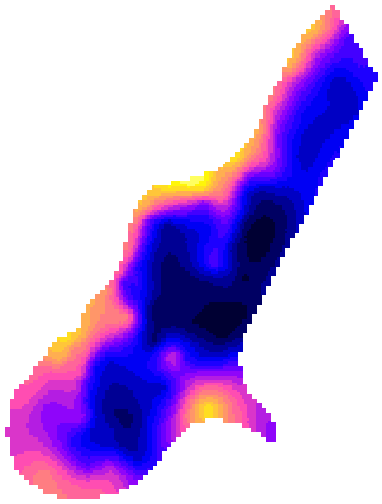
[17] Remember, this is for visualization, not quantitative prediction.

```
[+proj=longlat +datum=WGS84 +ellps=WGS84 +towgs84=0,0,0]
Number of points: 12410
Grid attributes:
  cellcentre.offset      cellsize cells.dim
x         5.7209877 0.00028487730       157
y        50.9556398 0.00017937874       208
Data attributes:
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 4.7853  5.2436  5.5770  5.7064  6.1691  7.4306
```

> **Note:** You can see the elapsed time on my system: 18.1 seconds; is yours faster or slower?

We display the smooth interpolation as an image (not a map), with the `image` function:

```
> image(llSPix, "pred", col = bpy.colors(20))
```



This can now be exported to the PNG graphics format, using the `png` method to open a PNG graphics file, and the `dev.off` method at the end to close it. The parameters of the file are set with `par`, and the pixels are written with the `image` graphics function:

```
> png(file = "MeuseZnInterpolation.png", width = GRD.wgs84$width,
+     height = GRD.wgs84$height, bg = "transparent")
> par(mar = c(0, 0, 0, 0), xaxs = "i", yaxs = "i")
> image(llSPix, "pred", col = bpy.colors(128))
> dev.off()
```

> **Note:** In the `png` function, note how the `width` and `height` arguments are taken from the control file which was created by the multi-purpose

`GE_SpatialGrid` method.

Note: The `par` function sets graphics device parameters; for the `png` device setting the margins to all 0's removes the default boundary, so that the image lines up with the coördinates specified in the KML file (see next), and setting the axis style to `"i"` also stops the driver from extending the image to show axis labels.

Note: If the image had been rectangular, it could have also been written as a matrix, specifying the rows and columns as arguments to the `matrix` function:

```
> image(matrix(llSPix$pred, nrow = GRD.wgs84$SG@grid@cells.dim[1],
+     ncol = GRD.wgs84$SG@grid@cells.dim[2]))
```

Finally, this PNG file is converted to a KML image overlay with the `kmlOverlay` method, which takes care of all the details of setting this up for use in Google Earth:

```
> kmlOverlay(obj = GRD.wgs84, kmlfile = "MeuseZn.kml",
+     imagefile = "MeuseZn.png", name = "Log(Zn) concentration, Meuse soils")

[1] "<?xml version='1.0' encoding='UTF-8'?>"
[2] "<kml xmlns='http://earth.google.com/kml/2.0'>"
[3] "<GroundOverlay>"
[4] "<name>Log(Zn) concentration, Meuse soils</name>"
[5] "<Icon><href>MeuseZn.png</href><viewBoundScale>0.75</viewBoundScale></Icon>"
[6] "<LatLonBox><north>50.992860922255</north><south>50.9555501437319</south><east>5.7655710
[7] "</GroundOverlay></kml>"
```

Notice the output, which shows each line of the KML file. We can check the file:

```
> file.show("MeuseZn.kml")
```

Note: The KML file does *not* contain the ground overlay image, it only specifies where (geographically) to plot it. Google Earth will look for the image in the same directory from which the KML file was launched. If the overlay is located on a server, the reference to it in the `<href> ... </href>` tag of the KML has to be adjusted to point to the URL where the image is stored.

Task 20 : View the ground overlay in Google Earth. •

The screen should look something like Figure 3; note that the overlay has been made partially transparent by using the transparency slider in Google Earth.

Task 21 : Clean up from this section. •

```
> rm(meuse, meuse.wgs84, meuse.grid, v, vm, v.rd, v.wgs84,
+     GRD.wgs84, GRD.wgs84_in, llSGDF, llSPix, grd, grd.union,
+     grd.union.wgs84, kr, kr.wgs84, st)
```

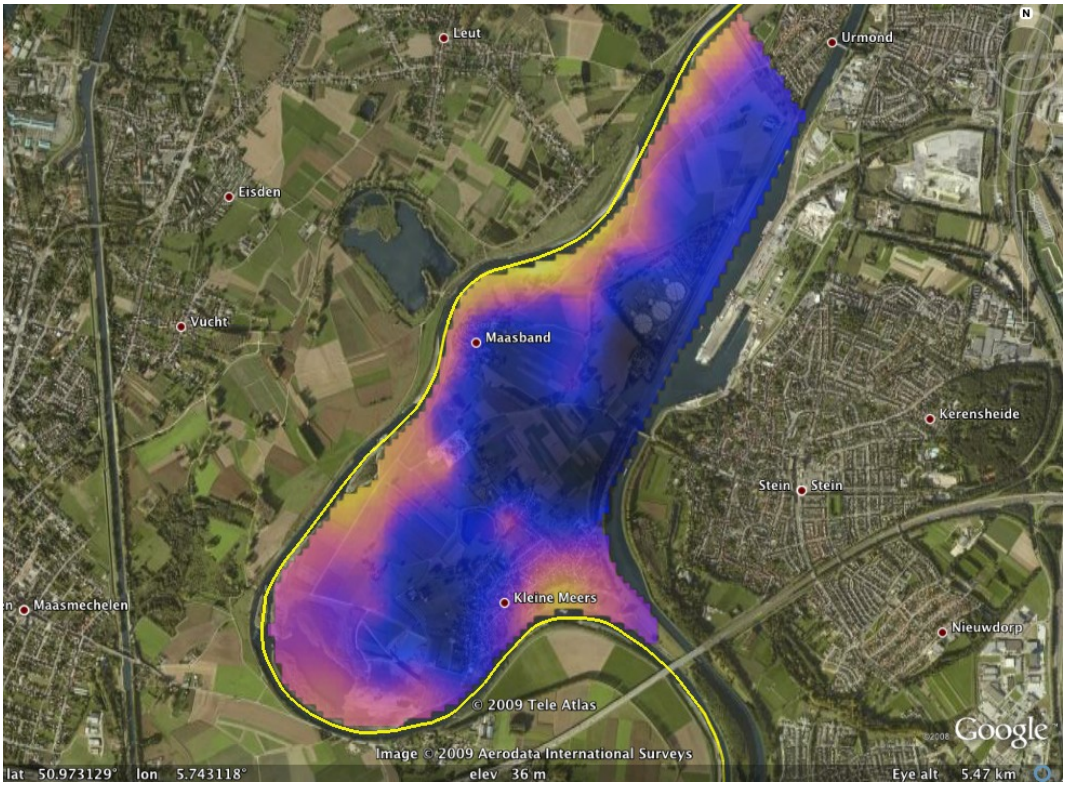


Figure 3: Interpolated Log(Zn) concentration, shown in Google Earth

4 Importing and exporting grids

Grids (rasters) are particularly easy to exchange. The major complication is preserving true coordinates and projection information.

4.1 ESRI ASCII grids

This is an interchange format developed by ESRI for image data. Most raster GIS (e.g. Imagine, ILWIS) can export grids in this format. An ESRI ASCII grid can be read and written directly as spatial objects (as defined in package `sp`) with methods `read.asciigrid` and `write.asciigrid`.

We use as an example a test image file which includes missing values, provided with the `sp` package:

Task 22 : Find the location on your system of the example `test.ag` file in the `sp` package. •

We use the `system.file` method to find the full path of a file distributed with a package, in this case `sp`:

```
> (f <- system.file("external/test.ag", package = "sp")[1])
```

```
[1] "/Library/Frameworks/R.framework/Resources/library/sp/external/test.ag"
```

Task 23 : View the contents of the file in a plain-text editor. You can also use the `file.show` function to display the file in a separate window within R. •

```
> file.show(f)
```

The output here has been abbreviated with `...`:

```
NCOLS 80
NROWS 115
XLLCORNER 178400.000000
YLLCORNER 329400.000000
CELLSIZE 40.000000
NODATA_VALUE 1.0e31
 1.0e31 1.0e31 1.0e31 1.0e31 1.0e31 ...
...
 1.0e31 1.0e31 1.0e31 1.0e31 1.0e31 1.0e31 1.0e31 1.0e31 1.0e31
 1.0e31 1.0e31 1.0e31 1.0e31 1.0e31 1.0e31 1.0e31 1.0e31 1.0e31
 1.0e31 1.0e31 1.0e31 1.0e31 1.0e31 1.0e31 1.0e31 1.0e31 1.0e31
 1.0e31 1.0e31 1.0e31 1.0e31 1.0e31 1.0e31 1.0e31 1.0e31 1.0e31
 1.0e31 1.0e31 1.0e31 1.0e31 1.0e31 1.0e31 1.0e31 1.0e31 1.0e31
 1.0e31 1.0e31   816.139   883.365   942.816   843.089   699.424
 616.902  580.08   557.019   552.757   525.807   460.971  406.23
 345.293   286.796   283.064   269.464   222.415   234.081   279.537
 305.188   318.359   326.189   333.156
...
```

Q10 : *How many header lines are there? What information do they give?*

Jump to A10 •

Task 24 : Import the file to a spatial object, examine its structure, and display it. •

Note the use of the optional `colname` argument to `read.asciigrid`; this names the single item in the dataframe, which otherwise would be given the name of the source file.

```
> test.grid <- read.asciigrid(f, colname = "z")
> str(test.grid)

Formal class 'SpatialGridDataFrame' [package "sp"] with 6 slots
  ..@ data       :'data.frame':        9200 obs. of  1 variable:
  .. ..$ z: num [1:9200] NA NA NA NA NA NA NA NA NA ...
  ..@ grid       :Formal class 'GridTopology' [package "sp"] with 3 slots
  .. .. ..@ cellcentre.offset: num [1:2] 178420 329420
  .. .. ..@ cellsize         : num [1:2] 40 40
  .. .. ..@ cells.dim        : int [1:2] 80 115
  ..@ grid.index : int(0)
  ..@ coords     : num [1:2, 1:2] 178420 181580 329420 333980
  .. ..- attr(*, "dimnames")=List of 2
  .. .. ..$ : NULL
  .. .. ..$ : chr [1:2] "coords.x1" "coords.x2"
  ..@ bbox       : num [1:2, 1:2] 178400 329400 181600 334000
  .. ..- attr(*, "dimnames")=List of 2
  .. .. ..$ : chr [1:2] "coords.x1" "coords.x2"
  .. .. ..$ : chr [1:2] "min" "max"
  ..@ proj4string:Formal class 'CRS' [package "sp"] with 1 slots
  .. .. ..@ projargs: chr NA
```

Q11 : *What is the data type of the imported object?* *Jump to A11* •

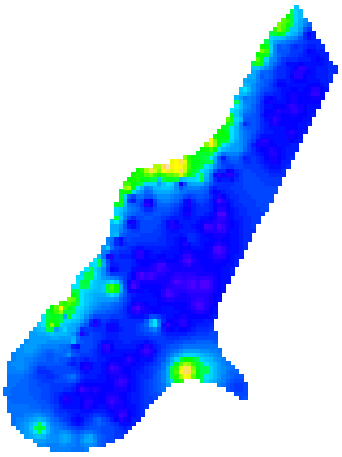
Q12 : *What is the coordinate system and projection?* *Jump to A12* •

Q13 : *What are the names of the two coordinates?* *Jump to A13* •

Task 25 : Display the imported grid. •

To display gridded data held in a spatial object, we can use the `image` method from `sp`.

```
> image(test.grid, col = topo.colors(64))
```



To go further with this, we need coordinate system, projection and datum information from metadata, which in this case we do not have.

```
> rm(f, test.grid)
```

4.2 Matrices as spatial objects

Sometimes images are provided only as a list of values. These can be read into R with the `scan` method and converted to a matrix with the `matrix` method.

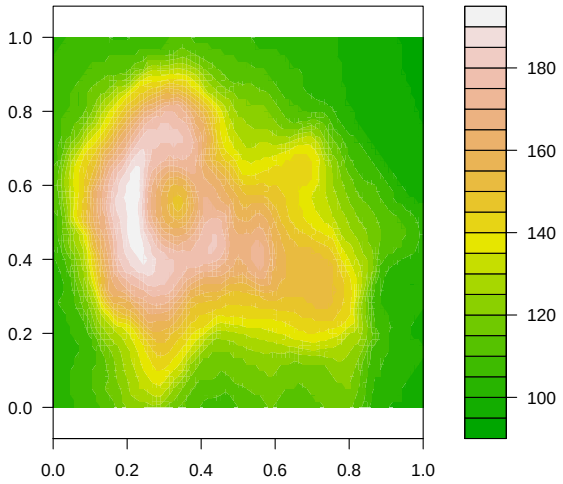
As an example, we use the `volcano` dataset, provided in the `datasets` package. This is a matrix of elevations on a regular grid.

Task 26 : Load the `volcano` dataset and examine its structure; also read its metadata and display it as a filled contour plot. •

The `filled.contour` plot displays a matrix as a sequence of solid colours along a colour ramp.

```
> data(volcano)
> str(volcano)

 num [1:87, 1:61] 100 101 102 103 104 105 105 106 107 108 ...

> dim(volcano)

[1] 87 61
```

```
> filled.contour(volcano, color.palette = terrain.colors,
+     asp = 1)
> help(volcano)
```



Note: In this case the matrix was already saved as an R object: a matrix with dimensions. In general the matrix must be created from a vector, using the `matrix` method.

Q14 : *What is the data type of this object? Does it have any coördinates? If so, what are they?* *Jump to A14* •

To use matrices as spatial objects, they must be provided with coordinates. The row and column numbers of the matrix can serve.

Task 27 : Convert the `volcano` object into a `SpatialPixelsDataFrame` object. •

First we convert to a data frame, using the `data.frame` method. The fields with the matrix row and column numbers must be computed from the matrix dimensions, extracted with the `dim` method.

Note the two uses of the `rep` method:

1. to repeat the sequence `1:dim(volcano)[1]` `dim(volcano)[2]` times for the columns;
2. to repeat each element of the sequence `1:dim(volcano)[2]` a `dim(volcano)[1]` number of times, with the `each` argument.

```
> volcano.sp <- data.frame(x = rep(1:dim(volcano)[1], dim(volcano)[2]),
+     y = rep(1:dim(volcano)[2], each = dim(volcano)[1]),
+     z = as.vector(volcano))
> str(volcano.sp)
```

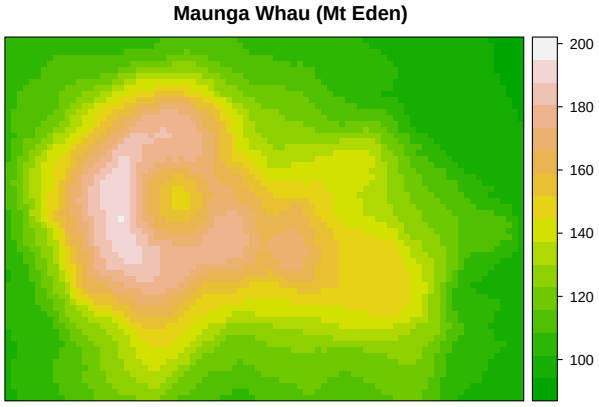
```
'data.frame':        5307 obs. of  3 variables:
 $ x: int  1 2 3 4 5 6 7 8 9 10 ...
 $ y: int  1 1 1 1 1 1 1 1 1 1 ...
 $ z: num  100 101 102 103 104 105 105 106 107 108 ...
```

Second, we convert this to a spatial object by specifying which columns are the coordinates, using the `coordinates` method:

```
> coordinates(volcano.sp) <- ~x + y
> str(volcano.sp)

Formal class 'SpatialPointsDataFrame' [package "sp"] with 5 slots
  ..@ data       :'data.frame':        5307 obs. of  1 variable:
  .. ..$ z: num [1:5307] 100 101 102 103 104 105 105 106 107 108 ...
  ..@ coords.nrs : int [1:2] 1 2
  ..@ coords     : num [1:5307, 1:2] 1 2 3 4 5 6 7 8 9 10 ...
  .. ..- attr(*, "dimnames")=List of 2
  .. .. ..$ : NULL
  .. .. ..$ : chr [1:2] "x" "y"
  ..@ bbox       : num [1:2, 1:2] 1 1 87 61
  .. ..- attr(*, "dimnames")=List of 2
  .. .. ..$ : chr [1:2] "x" "y"
  .. .. ..$ : chr [1:2] "min" "max"
  ..@ proj4string:Formal class 'CRS' [package "sp"] with 1 slots
  .. .. ..@ projargs: chr NA
```

Finally, since it is gridded (i.e. points are on a regular grid), we inform `sp` of that fact with the `gridded` method, thereby converting the `SpatialPointsDataFrame` to `SpatialPixelsDataFrame`.

```
> gridded(volcano.sp) <- TRUE
> str(volcano.sp)

Formal class 'SpatialPixelsDataFrame' [package "sp"] with 7 slots
  ..@ data       :'data.frame':        5307 obs. of  1 variable:
  .. ..$ z: num [1:5307] 100 101 102 103 104 105 105 106 107 108 ...
  ..@ coords.nrs : num(0)
  ..@ grid       :Formal class 'GridTopology' [package "sp"] with 3 slots
  .. .. ..@ cellcentre.offset: Named num [1:2] 1 1
  .. .. .. ..- attr(*, "names")= chr [1:2] "x" "y"
  .. .. ..@ cellsize         : Named num [1:2] 1 1
  .. .. .. ..- attr(*, "names")= chr [1:2] "x" "y"
  .. .. ..@ cells.dim        : Named int [1:2] 87 61
  .. .. .. ..- attr(*, "names")= chr [1:2] "x" "y"
  ..@ grid.index : int [1:5307] 5221 5222 5223 5224 5225 5226 5227 5228 5229 5230 ...
  ..@ coords     : num [1:5307, 1:2] 1 2 3 4 5 6 7 8 9 10 ...
  .. ..- attr(*, "dimnames")=List of 2
  .. .. ..$ : NULL
  .. .. ..$ : chr [1:2] "x" "y"
  ..@ bbox       : num [1:2, 1:2] 0.5 0.5 87.5 61.5
  .. ..- attr(*, "dimnames")=List of 2
  .. .. ..$ : chr [1:2] "x" "y"
  .. .. ..$ : chr [1:2] "min" "max"
  ..@ proj4string:Formal class 'CRS' [package "sp"] with 1 slots
  .. .. ..@ projargs: chr NA

> print(spplot(volcano.sp, col.regions = terrain.colors(64),
+     main = "Maunga Whau (Mt Eden)"))
```



To go further with geographic analysis, e.g. to write a Google Earth overlay, we'd need to **georeference** the image to some CRS; in this case there is no information on tie points.

Task 28 : Clean up from this section. •

```
> rm(volcano, volcano.sp)
```

5 Answers

A1 : *(1) The slot (marked with* `@`*)* `proj4string` *is of class* `CRS`*, this will hold the projection information.*

(2) The units of measure at this point are unknown, there is no specification. *Return to Q1* •

A2 : *It is now empty (*`NA` *meaning "not available" or "not applicable", take your pick).* *Return to Q2* •

A3 : *(1) No, we just specified what they mean.*

(2) We have now specified that these are meters on the Dutch national grid system, with respect to its $(0, 0)$ *origin.* *Return to Q3* •

A4 : *Decimal degrees of longitude (*`x`*) and latitude (*`y`*), with respect to Greenwich meridian (longitude) and the equator (latitude).* *Return to Q4* •

A5 : *(1) The number of point-pairs (field* `np`*) and the average semivariances (field*

`gamma`) *are exactly the same; the average separation between point-pairs in the bin (field* `dist`*) are different in two ways: (i) units of measure: on the elipsoid they are in km, in the metric system in the system's units, in this case meters; (ii) once the km are converted to m, there is still some difference in average separation.*

(2) It seems that the distance over a curved surface would be longer than over a flat one.

(3) The explanation may be that the study area (near Maastricht) is far from the centre of the stereographic projection (Amersfoort), so the metric coördinates are stretched here. *Return to Q5* •

A6 *:*

In fact, the measurements on the elipsoid are shorter in this region of the RDH system (considerably removed from the origin in Amersfoort). The largest difference is -40.2 cm, i.e. approximately the length of a small person's foot. This is 0.0103 % of the longest dimension of the bounding box; a very small relative error in this small area. *Return to Q6* •

A7 *: The nugget was raised from* `0` *to 0.0507.*

The partial sill was changed from 0.7034 to 0.5906.

The range was changed from 1118 to 897 meters.

The automatic fit looks excellent. *Return to Q7* •

A8 *: Points that have the same metric RD x-coördinates, e.g. 1 and 3, now have slightly different geographic longitudes (x-coördinates); similarly for the y-coördinates of points 2, 3, and 4.* *Return to Q8* •

A9 *: Both are* `SpatialPolygons`*, i.e. a list of* `Polygons`*, each itself a list of* `Polygon`*. The converted grid has 3103, i.e. one for each pixel in the original grid, and the union has only 1 polygon.* *Return to Q9* •

A10 *: Six header lines: (1, 2) number of rows and columns in the image; (3, 4) lower left corner coordinates in X (columns) and Y (rows); (5) cell size; (6) the value used for missing data.*

Note the units of measure are not specified. *Return to Q10* •

A11 *: The data type is SpatialGridDataFrame.* *Return to Q11* •

A12 *: The coordinate system is a grid with unspecified units; the projection is given in the* `@proj4string` *slot as* `@projargs = NA`*, i.e. unknown; this interchange format does not preserve projection information.* *Return to Q12* •

A13 : *This is given in the second attributes of the* `@coords` *slot: coords.x1.* *Return to Q13* •

A14 : *It is a two-dimensional matrix with no coördinates as such; however the matrix indices can serve as coördinates in an unspecified system for purposes of the* `image` *function.* *Return to Q14* •

References

[1] E. C. Barrett and L. F. Curtis. *Introduction to environmental remote sensing*. Stanley Thornes Publishers, Cheltenham, Glos., UK, 4th edition, 1999. 15

[2] Roger S. Bivand, Edzer J. Pebesma, and V. Gómez-Rubio. *Applied Spatial Data Analysis with R*. UseR! Springer, 2008. http://www.asdar-book.org/. 1, 6, 7, 15

[3] Lev M Bugayevskiy and John P. Snyder. *Map projections : a reference manual*. Taylor & Francis, 1995. 2

[4] Arnoud de Bruijne, Joop van Buren, Anton Kösters, and Hans van der Marel. *De geodetische referentiestelsels van Nederland : definitie en vastlegging van ETRS89, RD en NAP en hun onderlinge relaties; Geodetic reference frames in the Netherlands : definition and specification of ETRS89, RD and NAP, and their mutual relationships*. 43. NCG, Nederlandse Commissie voor Geodesie, Netherlands Geodetic Commission, Delft (NL), 2005. URL http://www.ncg.knaw.nl/Publicaties/Groen/pdf/43Referentie.pdf. 6

[5] Jonathan Iliffe and Roger Lott. *Datums and map projections for remote sensing, GIS, and surveying*. Whittles Pub.; CRC Press, Scotland, UK; Boca Raton, FL, 2nd edition, 2008. 2

[6] F Leisch. *Sweave User's Manual*. TU Wein, Vienna (A), 2.7.1 edition, 2006. URL http://www.stat.uni-muenchen.de/~leisch/Sweave/. 1

[7] F Leisch. Sweave, part I: Mixing R and LaTeX. *R News*, 2(3):28–31, December 2002. URL http://CRAN.R-project.org/doc/Rnews/. 1

[8] Thomas M. Lillesand, Ralph W. Kiefer, and Jonathan W. Chipman. *Remote sensing and image interpretation*. John Wiley Sons, Hoboken, NJ, 6th edition, 2007. 15

[9] Cliff Mugnier. Grids and datums: The Kingdom of the Netherlands. *Photogrammetric Engineering & Remote Sensing*, 69(2):117–119, 2003. 5, 6

[10] Robert A. Ryerson, American Society for Photogrammetry, and Remote Sensing. *Manual of remote sensing*. J. Wiley, New York, 3rd. edition, 1998. 15

[11] John P. Snyder. *Map projections: a working manual*. Geological Survey professional paper 1395. US Government Printing Office, Washington, DC, 1987. 2

[12] J. van Sickle. *GPS for land surveyors*. CRC, Boca Raton, third edition edition, 2008. 2

Index of R Concepts

as, 16, 19

bbox (package:sp), 17

class, 16
colname method argument, 25
coordinates (package:sp), 28
CRS (package:sp), 4–7
CRS (sp class), 7, 29

data.frame, 27
datasets package, 26
dev.off, 21
dim, 27

each method argument, 27

file.show, 24
filled.contour, 26
fit.variogram (package:gstat), 12

GE_SpatialGrid (package:maptools), 12, 15, 17, 18, 22
grep, 5
gridded (package:sp), 12, 28
gstat package, 1, 8

height graphics argument, 21

idw (package:gstat), 20
image (package:sp), 25
image, 21, 31

kmlOverlay (package:maptools), 12, 22
krige (package:gstat), 20

lattice package, 1
length, 16

make_EPSG (package:rgdal), 5
maptools package, 1, 12, 16
matrix, 22, 26, 27

overlay (package:sp), 19

par, 22
par method argument, 21
paste, 6
png, 21
Polygon (sp class), 30
Polygons (sp class), 30

proj4string (package:sp), 3, 4, 6, 7

raster package, 2
read.asciigrid (package:sp), 24, 25
rep, 27
require, 2
rgdal package, 1, 4, 5, 7, 10

scan, 26
sp package, 1, 2, 4, 24, 25, 28
SpatialGridDataFrame (package:sp), 20
SpatialGridDataFrame (sp class), 19
SpatialPixelsDataFrame (sp class), 15, 19, 27, 28
SpatialPointsDataFrame (sp class), 15, 28
SpatialPolygons (sp class), 17, 30
spTransform (package:rgdal), 7, 17
system.file, 24
system.time, 20

unionSpatialPolygons (package:maptools), 16
unique, 19

variogram (package:gstat), 8, 12
vgm (package:gstat), 12
vignette, 1
volcano dataset, 26, 27

which, 13
width graphics argument, 21
write.asciigrid (package:sp), 24
writeOGR (package:rgdal), 5, 10