4

Spatial Data Import and Export

Geographical information systems (GIS) and the types of spatial data they handle were introduced in Chap. 1. We now show how spatial data can be moved between **sp** objects in R and external formats, including the ones typically used by GIS. In this chapter, we first show how coordinate reference systems can be handled portably for import and export, going on to transfer vector and raster data, and finally consider ways of linking R and GIS more closely.

Before we begin, it is worth noting the importance of open source projects in making it possible to offer spatial data import and export functions in R. Many of these projects are now gathered in the Open Source Geospatial Foundation.[1] There are some projects which form the basis for the others, in particular the Geospatial Data Abstraction Library[2] (GDAL, pronounced Góodal, coordinated by Frank Warmerdam). Many of the projects also use the PROJ.4 Cartographic Projections library,[3] originally written by Gerald Evenden then of the United States Geological Survey, and modified and maintained by Frank Warmerdam. Without access to such libraries and their communities, it would not be possible to provide import or export facilities for spatial data in R. Many of the open source toolkits are also introduced in depth in Mitchell (2005). As we proceed, further links to relevant sources of information, such as mailing list archives, will be given.

In this chapter, we consider the representation of coordinate reference systems in a robust and portable way. Next, we show how spatial data may be read into R, and be written from R, using the most popular formats. The interface with GRASS GIS will be covered in detail, and finally the export of data for visualisation will be described.

First, we show how loading the package providing most of the interfaces to the software of these open source projects, **rgdal**, reports their status:

[1] `http://www.osgeo.org/`.
[2] `http://www.gdal.org/`.
[3] `http://proj.maptools.org/`.

```
> library(rgdal)

Geospatial Data Abstraction Library extensions to R successfully loaded
Loaded GDAL runtime: GDAL 1.5.1, released 2008/03/14
GDAL_DATA: /home/rsb/lib/R/rgdal/gdal
Loaded PROJ.4 runtime: Rel. 4.6.0, 21 Dec 2007
PROJ_LIB: /home/rsb/lib/R/rgdal/proj
```

We see that the release version numbers and dates of the external dynamically loaded libraries are reported. In addition, the values of the system environment variables `GDAL_DATA` and `PROJ_LIB` are set internally to support files bundled with **rgdal**, and are reset to their initial values when the package exits.[4]

4.1 Coordinate Reference Systems

Spatial data vary a great deal both in the ways in which their position attributes are recorded and in the adequacy of documentation of how position has been determined. This applies both to data acquired from secondary sources and to Global Positioning System input, or data capture from analogue maps by digitising. This also constitutes a specific difference from the analysis say of medical imagery, which in general requires only a local coordinate system; astronomy and the mapping of other planets also constitute a separate but linked field. Knowledge about the coordinate reference system is needed to establish the positional coordinates' units of measurement, obviously needed for calculating distances between observations and for describing the network topology of their relative positions. This knowledge is essential for integrating spatial data for the same study area, but coming from different sources. Waller and Gotway (2004, pp. 40–47) describe some of the key concepts and features to be dealt with here in an accessible fashion.

Coordinate reference systems (CRS) are at the heart of geodetics and cartography: how to represent a bumpy ellipsoid on the plane. We can speak of geographical CRS expressed in degrees and associated with an ellipse – a model of the shape of the earth, a prime meridian defining the origin in longitude, and a datum. The concept of a datum is arbitrary and anchors a specific geographical CRS to an origin point in three dimensions, including an assumed height above the assumed centre of the earth or above a standard measure of sea level. Since most of these quantities have only been subject to accurate measurement since the use of satellites for surveying became common, changes in ellipse and datum characteristics between legacy maps and newly collected data are common.

In contrast, projected CRS are expressed by a specific geometric model projecting to the plane and measures of length, as well as the underlying

[4] The report returned when loading **rgdal** may be suppressed by wrapping the call in `suppressPackageStartupMessages`.

ellipse, prime meridian, and datum. Most countries have multiple CRS, often for very good reasons. Surveyors in cities have needed to establish a local datum and a local triangulation network, and frequently these archaic systems continue to be used, forming the basis for property boundaries and other legal documents.

Cartography and surveying has seen the development of national triangulations and of stipulated national projections, or sub-national or zoned projections for larger countries. Typically, problems arise where these regimes meet. The choices of ellipse, prime meridian, and datum may differ, and the chosen projection and metric may also differ, or have different key parameters or origin offsets. On land, national borders tend to be described adequately with reference to the topography, but at sea, things change. It was because the coastal states around the North Sea basin had incompatible and not fully defined CRS that the European Petroleum Survey Group (EPSG; now Oil & Gas Producers (OGP) Surveying & Positioning Committee) began collecting a geodetic parameter data set[5] starting in 1986, based on earlier work in member companies.

4.1.1 Using the EPSG List

The EPSG list is under continuous development, with corrections being made to existing entries, and new entries being added as required. A copy of the list is provided in the **rgdal** package,[6] because it permits the conversion of a large number of CRS into the PROJ.4 style description used here. Since it allows for datum transformation as well as projection, the number of different coordinate reference systems is larger than that in the **mapproj** package. Datum transformation is based on transformation to the World Geodetic System of 1984 (WGS84), or inverse transformation from it to an alternative specified datum. WGS84 was introduced after measurements of earth from space had become very accurate, and forms a framework into which local and national systems may be fitted.

The **rgdal** package copy of the EPSG list can be read into a data frame and searched using `grep`, for example. We try to reproduce the example given by the Royal Netherlands Navy entitled 'From ED50 towards WGS84, or does your GPS receiver tell you the truth?'[7] A position has been read from a chart in the ED50 datum about a nautical mile west of the jetties of IJmuiden, but needs to be converted to the WGS84 datum for comparison with readings from a GPS satellite navigation instrument. We need to transform the chart coordinates in ED50 – ED50 is the European Datum 1950 – to coordinates in the WGS84 datum (the concept of a datum is described on p. 82). In this case to save space, the search string has been chosen to match exactly the row needed; entering just `ED50` gives 35 hits:

[5] `http://www.epsg.org/`.
[6] See installation note at chapter end, p. 111.
[7] `http://www.hydro.nl/articles/artikel2_en.htm`.

```
> EPSG <- make_EPSG()
> EPSG[grep("^# ED50$", EPSG$note), ]

    code   note                                   prj4
149 4230 # ED50 +proj=longlat +ellps=intl +no_defs
```

The EPSG code is in the first column of the data frame and the PROJ.4 specification in the third column, with the known set of tags and values.

4.1.2 PROJ.4 CRS Specification

The PROJ.4 library uses a 'tag=value' representation of coordinate reference systems, with the tag and value pairs enclosed in a single character string. This is parsed into the required parameters within the library itself. The only values used autonomously in `CRS` class objects are whether the string is a character NA (missing) value for an unknown CRS, and whether it contains the string `longlat`, in which case the CRS contains geographical coordinates.[8] There are a number of different tags, always beginning with `+`, and separated from the value with `=`, using white space to divide the tag/value pairs from each other.[9] If we use the special tag `+init` with value `epsg:4230`, where `4230` is the EPSG code found above, the coordinate reference system will be populated from the tables supplied with the libraries (PROJ.4 and GDAL) and included in **rgdal**.

```
> CRS("+init=epsg:4230")

CRS arguments:
 +init=epsg:4230 +proj=longlat +ellps=intl +no_defs
```

The two tags that are known are `+proj` – projection, which takes the value `longlat` for geographical coordinates – and `+ellps` – ellipsoid, with value `intl` for the International Ellipsoid of 1909 (Hayford). There is, however, no `+towgs84` tag, and so without further investigation it will not be possible to make the datum transformation. Lots of information about CRS in general can be found in *Grids & Datums*,[10] a regular column in Photogrammetric Engineering & Remote Sensing. The February 2003 number covers the Netherlands and gives a three-parameter transformation – in some cases seven parameters are given to specify the shift between datums.[11] Adding these values gives a full specification:

[8] The value `latlong` is not used, although valid, because coordinates in **sp** class objects are ordered with eastings first followed by northings.

[9] In addition to the EPSG list, there are many examples at the PROJ.4 website, for example: `http://geotiff.maptools.org/proj_list/`.

[10] `http://www.asprs.org/resources/GRIDS/`.

[11] Searching the PROJ.4 mailing list can also provide useful hints: `http://news.gmane.org/gmane.comp.gis.proj-4.devel`.

```
> ED50 <- CRS("+init=epsg:4230 +towgs84=-87,-96,-120,0,0,0,0")
> ED50

CRS arguments:
 +init=epsg:4230 +towgs84=-87,-96,-120,0,0,0,0 +proj=longlat
+ellps=intl +no_defs
```

Datum transformation shifts coordinates between differently specified ellipsoids in all three dimensions, even if the data appear to be only 2D, because 2D data are assumed to be on the surface of the ellipsoid. It may seem unreasonable that the user is confronted with the complexities of coordinate reference system specification in this way. The EPSG list provides a good deal of help, but assumes that wrong help is worse than no help, and does not give transformation parameters where there is any ambiguity, and for the ED50 datum, parameter values do vary across Europe. Modern specifications are designed to avoid ambiguity, and so this issue will become less troublesome with time, although old maps are going to be a source of data for centuries to come.

4.1.3 Projection and Transformation

In the Dutch navy case, we do not need to project because the input and output coordinates are geographical:

```
> IJ.east <- as(char2dms("4d31'00\"E"), "numeric")
> IJ.north <- as(char2dms("52d28'00\"N"), "numeric")
> IJ.ED50 <- SpatialPoints(cbind(x = IJ.east, y = IJ.north),
+     ED50)
> res <- spTransform(IJ.ED50, CRS("+proj=longlat +datum=WGS84"))
> x <- as(dd2dms(coordinates(res)[1]), "character")
> y <- as(dd2dms(coordinates(res)[2], TRUE), "character")
> cat(x, y, "\n")

4d30'55.294"E 52d27'57.195"N

> spDistsN1(coordinates(IJ.ED50), coordinates(res), longlat = TRUE) *
+     1000

[1] 124.0994

> library(maptools)
> gzAzimuth(coordinates(IJ.ED50), coordinates(res))

[1] -134.3674
```

Using correctly specified coordinate reference systems, we can reproduce the example successfully, with a 124 m shift between a point plotted in the inappropriate WGS84 datum and the correct ED50 datum for the chart:

> 'For example: one who has read his position 52d28′00″N/ 4d31′00″E (ED50) from an ED50-chart, right in front of the jetties of IJmuiden, has to adjust this co-ordinate about 125 m to the Southwest .... The corresponding co-ordinate in WGS84 is 52d27′57″N/ 4d30′55″E.'

The work is done by the `spTransform` method, taking any `Spatial*` object, and returning an object with coordinates transformed to the target `CRS`. There is no way of warping regular grid objects, because for arbitrary transformations, the new positions will not form a regular grid. The solution in this case is to convert the object to point locations, transform them to the new CRS, and interpolate to a suitably specified grid in the new CRS.

Two helper functions are also used here to calculate the difference between the points in ED50 and WGS84: `spDistsN1` and `gzAzimuth`. Function `spDistsN1` measures distances between a matrix of points and a single point, and uses Great Circle distances on the WGS84 ellipsoid if the `longlat` argument is `TRUE`. It returns values in kilometres, and so we multiply by 1,000 here to obtain metres. `gzAzimuth` gives azimuths calculated on the sphere between a matrix of points and a single point, which must be geographical coordinates, with north zero, and negative azimuths west of north.

So far in this section we have used an example with geographical coordinates. There are many different projections to the plane, often chosen to give an acceptable representation of the area being displayed. There exist no all-purpose projections, all involve distortion when far from the centre of the specified frame, and often the choice of projection is made by a public mapping agency.

```
> EPSG[grep("Atlas", EPSG$note), 1:2]

    code                            note
578 2163 # US National Atlas Equal Area

> CRS("+init=epsg:2163")

+init=epsg:2163 +proj=laea +lat_0=45 +lon_0=-100 +x_0=0 +y_0=0
    +a=6370997 +b=6370997 +units=m +no_defs
```

For example, the US National Atlas has chosen a particular CRS for its view of the continental US, with a particular set of tags and values to suit. The projection chosen has the value `laea`, which, like many other values used to represent CRS in PROJ.4 and elsewhere, is rather cryptic. Provision is made to access descriptions within the PROJ.4 library to make it easier to interpret the values in the CRS. The `projInfo` function can return several kinds of information in tabular form, and those tables can be examined to shed a little more light on the tag values.

```
> proj <- projInfo("proj")
> proj[proj$name == "laea", ]

   name                  description
47 laea Lambert Azimuthal Equal Area
```

```
> ellps <- projInfo("ellps")
> ellps[grep("a=6370997", ellps$major), ]

     name       major         ell               description
42 sphere a=6370997.0 b=6370997.0 Normal Sphere (r=6370997)
```

It turns out that this CRS is in the Lambert Azimuthal Equal Area projection, using the sphere rather than a more complex ellipsoid, with its centre at 100° west and 45° north. This choice is well-suited to the needs of the Atlas, a compromise between coverage, visual communication, and positional accuracy.

All this detail may seem unnecessary, until the analysis we need to complete turns out to depend on data in different coordinate reference systems. At that point, spending time establishing as clearly as possible the CRS for our data will turn out to have been a wise investment. The same consideration applies to importing and exporting data – if their CRS specifications are known, transferring positional data correctly becomes much easier. Fortunately, for any study region the number of different CRS used in archived maps is not large, growing only when the study region takes in several jurisdictions. Even better, all modern data sources are much more standardised (most use the WGS84 datum), and certainly much better at documenting their CRS specifications.

4.1.4 Degrees, Minutes, and Seconds

In common use, the sign of the coordinate values may be removed and the value given a suffix of E or N for positive values of longitude or latitude and W or S for negative values. In addition, values are often recorded traditionally not as decimal degrees, but as degrees, minutes, and decimal seconds, or some truncation of this. These representations raise exactly the same questions as for time series, although time can be mapped onto the infinite real line, while geographical coordinates are cyclical – move 360° and you return to your point of departure. For practical purposes, geographical coordinates should be converted to decimal degree form; this example uses the Netherlands point that we have already met:

```
> IJ.dms.E <- "4d31'00\"E"
> IJ.dms.N <- "52d28'00\"N"
```

We convert these character strings to class ‘DMS’ objects, using function char2dms:

```
> IJ_east <- char2dms(IJ.dms.E)
> IJ_north <- char2dms(IJ.dms.N)
> IJ_east

[1] 4d31'E

> IJ_north
```

```
[1] 52d28'N
```

```
> getSlots("DMS")
```

```
       WS       deg       min       sec        NS
"logical" "numeric" "numeric" "numeric" "logical"
```

The DMS class has slots to store representations of geographical coordinates, however, they might arise, but the char2dms() function expects the character input format to be as placed, permitting the degree, minute, and second symbols to be given as arguments. We get decimal degrees by coercing from class 'DMS' to class 'numeric' with the as() function:

```
> c(as(IJ_east, "numeric"), as(IJ_north, "numeric"))
```

```
[1]  4.516667 52.466667
```

4.2 Vector File Formats

Spatial vector data are points, lines, polygons, and fit the equivalent **sp** classes. There are a number of commonly used file formats, most of them proprietary, and some newer ones which are adequately documented. GIS are also more and more handing off data storage to database management systems, and some database systems now support spatial data formats. Vector formats can also be converted outside R to formats for which import is feasible.

GIS vector data can be either topological or simple. Legacy GIS were topological, desktop GIS were simple (sometimes known as spaghetti). The **sp** vector classes are simple, meaning that for each polygon all coordinates are stored without checking that boundaries have corresponding points. A topological representation in principal stores each point only once, and builds arcs (lines between nodes) from points, polygons from arcs – the GRASS 6 open source GIS has such a topological representation of vector features. Only the **RArcInfo** package tries to keep some traces of topology in importing legacy ESRI™ ArcInfo™ binary vector coverage data (or e00 format data) – **maps** uses topology because that was how things were done when the underlying code was written. The import of ArcGIS™ coverages is described fully in Gómez-Rubio and López-Quílez (2005); conversion of imported features into **sp** classes is handled by the pal2SpatialPolygons function in **maptools**.

It is often attractive to make use of the spatial databases in the **maps** package. They can be converted to **sp** class objects using functions such as map2SpatialPolygons in the **maptools** package. An alternative source of coastlines is the Rgshhs function in **maptools**, interfacing binary databases of varying resolution distributed by the 'Global Self-consistent, Hierarchical, High-resolution Shoreline Database' project.[12]

[12] http://www.soest.hawaii.edu/wessel/gshhs/gshhs.html.

The best resolution databases are rather large, and so **maptools** ships only with the coarse resolution one; users can install and use higher resolution databases locally. Figures 2.3 and 2.7, among others in earlier chapters, have been made using these sources.

A format that is commonly used for exchanging vector data is the shapefile. This file format has been specified by ESRI™, the publisher of ArcView™ and ArcGIS™, which introduced it initially to support desktop mapping using ArcView™.[13] This format uses at least three files to represent the data, a file of geometries with an `*.shp` extension, an index file to the geometries `*.shx`, and a legacy `*.dbf` DBF III file for storing attribute data. Note that there is no standard mechanism for specifying missing attribute values in this format. If a `*.prj` file is present, it will contain an ESRI™ well-known text CRS specification. The shapefile format is not fully compatible with the OpenGIS® Simple Features Specification (see p. 122 for a discussion of this specification). Its incompatibility is, however, the same as that of the `SpatialPolygons` class, using a collection of polygons, both islands and holes, to represent a single observation in terms of attribute data.

4.2.1 Using OGR Drivers in rgdal

Using the OGR vector functions of the Geospatial Data Abstraction Library, interfaced in **rgdal**,[14] lets us read spatial vector data for which drivers are available. A driver is a software component plugged-in on demand – here the OGR library tries to read the data using all the formats that it knows, using the appropriate driver if available. OGR also supports the handling of coordinate reference systems directly, so that if the imported data have a specification, it will be read.

The availability of OGR drivers differs from platform to platform, and can be listed using the `ogrDrivers` function. The function also lists whether the driver supports the creation of output files. Because the drivers often depend on external software, the choices available will depend on the local computer installation. It is frequently convenient to convert from one external file format to another using utility programs such as `ogr2ogr` in binary FWTools releases, which typically include a wide range of drivers.[15]

The `readOGR` function takes at least two arguments – they are the data source name `(dsn)` and the layer `(layer)`, and may take different forms for different drivers. It is worth reading the relevant web pages[16] for the format being imported. For ESRI™ shapefiles, `dsn` is usually the name of the directory containing the three (or more) files to be imported (given as `"."` if the working

[13] The format is fully described in this white paper: `http://shapelib.maptools.org/dl/shapefile.pdf`.

[14] See installation note at chapter end.

[15] `http://fwtools.maptools.org`.

[16] `http://ogr.maptools.org/ogr_formats.html`.

directory), and `layer` is the name of the shapefile without the `".shp"` extension. Additional examples are given on the function help page for file formats, but it is worth noting that the same function can also be used where the data source name is a database connection, and the layer is a table, for example using PostGIS in a PostgreSQL database.

We can use the classic Scottish lip cancer data set by district downloaded from the additional materials page for Chap. 9 in Waller and Gotway (2004).[17] There are three files making up the shapefile for Scottish district boundaries at the time the data were collected – the original study and extra data in a separate text file are taken from Clayton and Kaldor (1987). The shapefile appears to be in geographical coordinates, but no `*.prj` file is present, so after importing from the working directory, we assign a suitable coordinate reference system.

```
> scot_LL <- readOGR(".", "scot")

OGR data source with driver: ESRI Shapefile
Source: ".", layer: "scot"
with  56  rows and  2  columns

> proj4string(scot_LL) <- CRS("+proj=longlat ellps=WGS84")
> scot_LL$ID

 [1] 12 13 19  2 17 16 21 50 15 25 26 29 43 39 40 52 42 51 34 54 36 46
[23] 41 53 49 38 44 30 45 48 47 35 28  4 20 33 31 24 55 18 56 14 32 27
[45] 10 22  6  8  9  3  5 11  1  7 23 37
```

The Clayton and Kaldor data are for the same districts, but with the rows ordered differently, so that before combining the data with the imported polygons, they need to be matched first (matching methods are discussed in Sect. 5.5.2):

```
> scot_dat <- read.table("scotland.dat", skip = 1)
> names(scot_dat) <- c("District", "Observed", "Expected",
+     "PcAFF", "Latitude", "Longitude")
> scot_dat$District

 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22
[23] 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44
[45] 45 46 47 48 49 50 51 52 53 54 55 56

> library(maptools)
> scot_dat1 <- scot_dat[match(scot_LL$ID, scot_dat$District),
+     ]
> row.names(scot_dat1) <- sapply(slot(scot_LL, "polygons"),
+     function(x) slot(x, "ID"))
> scot_LLa <- spCbind(scot_LL, scot_dat1)
> all.equal(scot_LLa$ID, scot_LLa$District)
```

[17] `http://www.sph.emory.edu/~lwaller/WGindex.htm`.

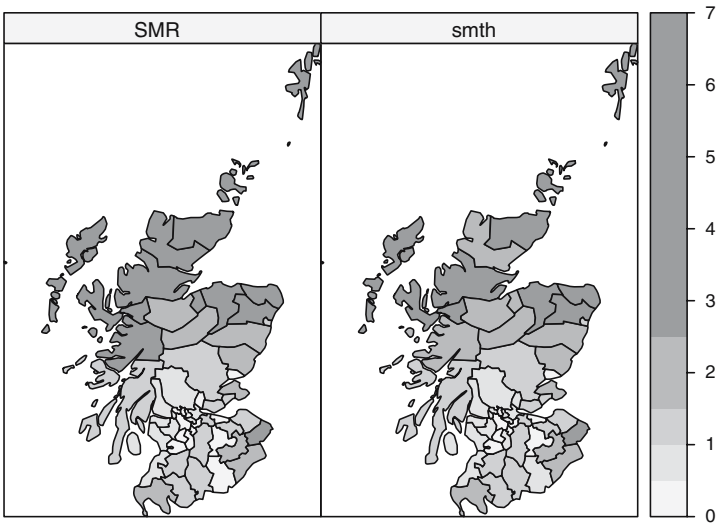

Fig. 4.1. Comparison of relative risk and EB smoothed relative risk for Scottish lip cancer

```
[1] TRUE
```

```
> names(scot_LLa)
```

```
[1] "NAME"      "ID"        "District"  "Observed"  "Expected"
[6] "PcAFF"     "Latitude"  "Longitude"
```

Figure 4.1 compares the relative risk by district with the Empirical Bayes smooth values – we return to the actual techniques involved in Chap. 11, here the variables are being added to indicate how results may be exported from R below. The relative risk does not take into account the possible uncertainty associated with unusual incidence rates in counties with relatively small populations at risk, while Empirical Bayes smoothing shrinks such values towards the rate for all the counties taken together.

```
> library(spdep)
> O <- scot_LLa$Observed
> E <- scot_LLa$Expected
> scot_LLa$SMR <- probmap(O, E)$relRisk/100
> library(DCluster)
> scot_LLa$smth <- empbaysmooth(O, E)$smthrr
```

Finally, we project the district boundaries to the British National Grid as described by Waller and Gotway (2004):

```
> scot_BNG <- spTransform(scot_LLa, CRS("+init=epsg:27700"))
```

We export these data in two forms, first as Keyhole Markup Language (KML) overlays for Google Earth™. The underlying coordinate reference system for Google Earth™ is geographical, in the WGS84 datum, so we can export the district boundaries as we imported them, using the `writeOGR` function, choosing only a single variable. This function, like `readOGR`, uses drivers to handle different data formats, with `driver="KML"` in this case. Next we take the district centroids and export them as a `SpatialPointsDataFrame`, with the district name, the observed and expected values, and the two rates:

```
> writeOGR(scot_LLa["ID"], dsn = "scot_district.kml",layer = "borders",
+     driver = "KML")
> llCRS <- CRS("+proj=longlat ellps=WGS84")
> scot_SP_LL <- SpatialPointsDataFrame(coordinates(scot_LLa),
+     proj4string = llCRS, data = as(scot_LLa, "data.frame")[c("NAME",
+         "Observed", "Expected", "SMR", "smth")])
> writeOGR(scot_SP_LL, dsn = "scot_rates.kml", layer = "rates",
+     driver = "KML")
```

The output format for Google Earth™ is fairly simple, and it will probably become possible in time to make the export objects more intelligent, but just being able to distribute spatial output permitting an online recipient simply to display results by opening a file does offer considerable opportunities, as Fig. 4.2 illustrates. We could also have written an image overlay, as we see later in Sect. 4.3.2.

Fig. 4.2. Scottish district boundaries and centroid pointers shown in Google Earth™

We can of course export to a shapefile, using `driver="ESRI Shapefile"`, or to other file formats for which output drivers are implemented:

```
> drv <- "ESRI Shapefile"
> writeOGR(scot_BNG, dsn = ".", layer = "scot_BNG", driver = drv)

> list.files(pattern = "^scot_BNG")

[1] "scot_BNG.dbf" "scot_BNG.prj" "scot_BNG.shp" "scot_BNG.shx"
[5] "scot_BNG.txt"
```

The output now contains a `*.prj` file with the fully specified coordinate reference system for the British National Grid, to which we projected the data object.

4.2.2 Other Import/Export Functions

If the **rgdal** package is not available, there are two other packages that can be used for reading and writing shapefiles. The **shapefiles** package is written without external libraries, using file connections. It can be very useful when a shapefile is malformed, because it gives access to the raw numbers. The **maptools** package contains a local copy of the library used in OGR for reading shapefiles (the DBF reader is in the **foreign** package), and provides a low-level import `read.shape` function, a helper function `getinfo.shape` to identify whether the shapefile contains points, lines, or polygons.

```
> getinfo.shape("scot_BNG.shp")

Shapefile type: Polygon, (5), # of Shapes: 56
```

There are three functions to read these kinds of data: `readShapePoints`, `readShapeLines`, and `readShapePoly`. They are matched by equivalent exporting functions: `writePolyShape`, `writeLinesShape`, `writePointsShape`, using local copies of shapelib functions otherwise available in **rgdal** in the OGR framework. The **RArcInfo** package also provides local access to OGR functionality, for reading ArcGIS™ binary vector coverages, but with the addition of a utility function for converting `e00` format files into binary coverages; full details are given in Gómez-Rubio and López-Quílez (2005).

4.3 Raster File Formats

There are very many raster and image formats; some allow only one band of data, others assume that data bands are Red-Green-Blue (RGB), while yet others are flexible and self-documenting. The simplest formats are just rectangular blocks of uncompressed data, like a matrix, but sometimes with row indexing reversed. Others are compressed, with multiple bands, and may be interleaved so that subscenes can be retrieved without unpacking the whole

image. There are now a number of R packages that support image import and export, such as the **rimage** and **biOps** packages and the **EBImage** package in the Bioconductor project. The requirements for spatial raster data handling include respecting the coordinate reference system of the image, so that specific solutions are needed. There is, however, no direct support for the transformation or 'warping' of raster data from one coordinate reference system to another.

4.3.1 Using GDAL Drivers in rgdal

Many drivers are available in **rgdal** in the `readGDAL` function, which – like `readOGR` – finds a usable driver if available and proceeds from there. Using arguments to `readGDAL`, subregions or bands may be selected, and the data may be decimated, which helps handle large rasters. The simplest approach is just to read all the data into the R workspace – here we will the same excerpt from the Shuttle Radar Topography Mission (SRTM) flown in 2000, for the Auckland area as in Chap. 2.

```
> auck_el1 <- readGDAL("70042108.tif")

70042108.tif has GDAL driver GTiff
and has 1200 rows and 1320 columns

> summary(auck_el1)

Object of class SpatialGridDataFrame
Coordinates:
    min   max
x 174.2 175.3
y -37.5 -36.5
Is projected: FALSE
proj4string :
[+proj=longlat +ellps=WGS84 +datum=WGS84 +no_defs
+towgs84=0,0,0]
Number of points: 2
Grid attributes:
  cellcentre.offset     cellsize cells.dim
x         174.20042 0.0008333333      1320
y         -37.49958 0.0008333333      1200
Data attributes:
      Min.    1st Qu.     Median       Mean    3rd Qu.       Max.
-3.403e+38  0.000e+00  1.000e+00 -1.869e+34  5.300e+01  6.860e+02

> is.na(auck_el1$band1) <- auck_el1$band1 <= 0 | auck_el1$band1 >
+     10000
```

The `readGDAL` function is actually a wrapper for substantially more powerful R bindings for GDAL written by Timothy Keitt. The bindings allow us to handle very large data sets by choosing sub-scenes and re-sampling, using the

`offset`, `region.dim`, and `output.dim` arguments. The bindings work by opening a data set known by GDAL using a `GDALDriver` class object, but only reading the required parts into the workspace.

```
> x <- GDAL.open("70042108.tif")
> xx <- getDriver(x)
> xx

An object of class "GDALDriver"
Slot "handle":
<pointer: 0x83945f0>

> getDriverLongName(xx)

[1] "GeoTIFF"

> x

An object of class "GDALReadOnlyDataset"
Slot "handle":
<pointer: 0x83d4708>

> dim(x)

[1] 1200 1320

> GDAL.close(x)
```

Here, `x` is a derivative of a `GDALDataset` object, and is the GDAL data set handle; the data are not in the R workspace, but all their features are there to be read on demand. An open GDAL handle can be read into a SpatialGridDataFrame, so that `readGDAL` may be done in pieces if needed. Information about the file to be accessed may also be shown without the file being read, using the GDAL bindings packaged in the utility function `GDALinfo`:

```
> GDALinfo("70042108.tif")

rows        1200
columns     1320
bands       1
ll.x        174.2
ll.y        -36.5
res.x       0.0008333333
res.y       0.0008333333
oblique.x   0
oblique.y   0
driver      GTiff
projection  +proj=longlat +ellps=WGS84 +datum=WGS84 +no_defs
file        70042108.tif
```

We use the Meuse grid data set to see how data may be written out using GDAL, adding a set of `towgs84` values from the GDAL bug tracker.[18] The `writeGDAL` function can be used directly for drivers that support file creation. For other file formats, which can be made as copies of a prototype, we need to create an intermediate GDAL data set using `create2GDAL`, and then use functions operating on the GDAL data set handle to complete. First we simply output inverse distance weighted interpolated values of Meuse Bank logarithms of zinc ppm as a GeoTiff file.

```
> library(gstat)
> log_zinc <- krige(log(zinc) ~ 1, meuse, meuse.grid)["var1.pred"]

[inverse distance weighted interpolation]

> proj4string(log_zinc) <- CRS(proj4string(meuse.grid))
> summary(log_zinc)

Object of class SpatialPixelsDataFrame
Coordinates:
     min    max
x 178440 181560
y 329600 333760
Is projected: TRUE
proj4string :
[+init=epsg:28992
+towgs84=565.237,50.0087,465.658,-0.406857,0.350733,-1.87035,4.0812
+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]
Number of points: 3103
Grid attributes:
  cellcentre.offset cellsize cells.dim
x            178460       40        78
y            329620       40       104
Data attributes:
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  4.791   5.484   5.694   5.777   6.041   7.482

> writeGDAL(log_zinc, fname = "log_zinc.tif", driver = "GTiff",
+     type = "Float32", options = "INTERLEAVE=PIXEL")

> GDALinfo("log_zinc.tif")

rows        104
columns     78
bands       1
ll.x        178440
ll.y        333760
```

[18] `http://trac.osgeo.org/gdal/ticket/1987`; newer versions of PROJ4/GDAL may include the correct `+towgs84` parameter values.

```
res.x       40
res.y       40
oblique.x   0
oblique.y   0
driver      GTiff
projection  +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
file        log_zinc.tif
```

The output file can for example be read into ENVI™ directly, or into ArcGIS™ via the 'Calculate statistics' tool in the Raster section of the Toolbox, and displayed by adjusting the symbology classification.

4.3.2 Writing a Google Earth™ Image Overlay

Our next attempt to export a raster will be more ambitious; in fact we can use this technique to export anything that can be plotted on a PNG graphics device. We export a coloured raster of interpolated log zinc ppm values to a PNG file with an alpha channel for viewing in Google Earth™. Since the target software requires geographical coordinates, a number of steps will be needed. First we make a polygon to bound the study area and project it to geographical coordinates:

```
> library(maptools)
> grd <- as(meuse.grid, "SpatialPolygons")
> proj4string(grd) <- CRS(proj4string(meuse))
> grd.union <- unionSpatialPolygons(grd, rep("x", length(slot(grd,
+     "polygons"))))
> ll <- CRS("+proj=longlat +datum=WGS84")
> grd.union.ll <- spTransform(grd.union, ll)
```

Next we construct a suitable grid in geographical coordinates, as our target object for export, using the `GE_SpatialGrid` wrapper function. This grid is also the container for the output PNG graphics file, so `GE_SpatialGrid` also returns auxiliary values that will be used in setting up the `png` graphics device within R. We use the `overlay` method to set grid cells outside the river bank area to NA, and then discard them by coercion to a SpatialPixelsDataFrame:

```
> llGRD <- GE_SpatialGrid(grd.union.ll)
> llGRD_in <- overlay(llGRD$SG, grd.union.ll)
> llSGDF <- SpatialGridDataFrame(grid = slot(llGRD$SG,
+     "grid"), proj4string = CRS(proj4string(llGRD$SG)),
+     data = data.frame(in0 = llGRD_in))
> llSPix <- as(llSGDF, "SpatialPixelsDataFrame")
```

We use `idw` from the **gstat** package to make an inverse distance weighted interpolation of zinc ppm values from the soil samples available, also, as here, when the points are in geographical coordinates; interpolation will be fully presented in Chap. 8:

Fig. 4.3. Interpolated log zinc ppm for the Meuse Bank data set shown in Google Earth™

```
> meuse_ll <- spTransform(meuse, CRS("+proj=longlat +datum=WGS84"))
> llSPix$pred <- idw(log(zinc) ~ 1, meuse_ll, llSPix)$var1.pred
```

```
[inverse distance weighted interpolation]
```

Since we have used `GE_SpatialGrid` to set up the size of an R `png` graphics device, we can now use it as usual, here with `image`. In practice, any base graphics methods and functions can be used to create an image overlay. Finally, after closing the graphics device, we use `kmlOverlay` to write a `*.kml` file giving the location of the overlay and which will load the image at that position when opened in Google Earth™, as shown in Fig. 4.3:

```
> png(file = "zinc_IDW.png", width = llGRD$width,height = llGRD$height,
+     bg = "transparent")
> par(mar = c(0, 0, 0, 0), xaxs = "i", yaxs = "i")
> image(llSPix, "pred", col = bpy.colors(20))
> dev.off()
> kmlOverlay(llGRD, "zinc_IDW.kml", "zinc_IDW.png")
```

4.3.3 Other Import/Export Functions

There is a simple `readAsciiGrid` function in **maptools** that reads ESRI™ Arc ASCII grids into SpatialGridDataFrame objects; it does not handle CRS and has a single band. The companion `writeAsciiGrid` is for writing Arc ASCII grids. It is also possible to use connections to read and write arbitrary binary files, provided that the content is not compressed. Functions in the R image analysis packages referred to above may also be used to read and write a number of image formats. If the grid registration slots in objects of classes defined in the **pixmap** package are entered manually, these objects may also be used to hold raster data.

4.4 Grass

GRASS[19] is a major open source GIS, originally developed as the Geographic Resources Analysis Support System by the U.S. Army Construction Engineering Research Laboratories (CERL, 1982–1995), and subsequently taken over by its user community. GRASS has traditional strengths in raster data handling, but two advances (floating point rasters and support for missing values) were not completed when development by CERL was stopped. These were added for many modules in the GRASS 5.0 release; from GRASS 6.0, new vector support has been added. GRASS is a very large but very simple system – it is run as a collection of separate programs built using shared libraries of core functions. There is then no GRASS 'program', just a script setting environment variables needed by the component programs. GRASS does interact with the OSGeo stack of applications, and GRASS functionality, including the R interface, is available in the Quantum GIS desktop GIS application.

An R package to interface with GRASS has been available on CRAN – **GRASS** – since the release of GRASS 5.0. It provided a compiled interface to raster and sites data, but not vector data, and included a frozen copy of the core GRASS GIS C library, modified to suit the fact that its functions were being used in an interactive, longer-running program like R. The **GRASS** package is no longer being developed, but continues to work for users of GRASS 5. The GRASS 5 interface is documented in Neteler and Mitasova (2004, pp. 333–354) and Bivand (2000).

The current GRASS releases, from GRASS 6.0, with GRASS 6.2 released in October 2006 and 6.3 due in 2008, have a different interface, using the **sp** classes presented in Chap. 2. Neteler and Mitasova (2008) describe GRASS 6 fully, and present this interface on pp. 353–364. The **spgrass6** package depends on other packages for moving data between GRASS and R, chiefly using **rgdal**, because GRASS also uses GDAL and OGR as its main import/export mechanisms. The interface works by exchanging temporary files in formats that both GRASS and **rgdal** know. This kind of loose coupling is less of a burden than it was before, with smaller, slower machines. This is why the GRASS 5 interface was tight-coupled, with R functions reading from and writing to the GRASS database directly. Using GRASS plug-in drivers in GDAL/OGR is another possibility for reading GRASS data directly into R through **rgdal**, without needing **spgrass6**; **spgrass6** can use these plug-in drivers if present for reading GRASS data.

GRASS uses the concept of a working region or window, specifying both the viewing rectangle and – for raster data – the resolution. The data in the GRASS database can be from a larger or smaller region and can have a different resolution, and are re-sampled to match the working region for analysis. This current window should determine the way in which raster data are retrieved and transferred.

[19] `http://grass.osgeo.org/`.

GRASS also uses the concepts of a location, with a fixed and uniform coordinate reference system, and of mapsets within the location. The location is typically chosen at the start of a work session, and with the location, the user will have read access to possibly several mapsets, and write access to some, probably fewer, to avoid overwriting the work of other users of the location.

Intermediate temporary files are the chosen solution for interaction between GRASS and R in **spgrass6**, using shapefiles for vector data and single band BIL (Band Interleaved by Line) binary files for raster data. Note that missing values are defined and supported for GRASS raster data, but that missing values for vector data are not uniformly defined or supported.

Support for GRASS under Cygwin is provided and known to function satisfactorily. Native Windows GRASS has been tested only with Quantum GIS, but is expected to become more widely available as GRASS 6.3 has now been released, using MSYS rather than Cygwin for Unix shell emulation (a Windows **spgrass6** binary is on CRAN). Mac OSX is treated as Unix, and **spgrass6** is installed from source (like **rgdal**). The **spgrass6** package should be installed with the packages it depends upon, **sp** and **rgdal**.

R is started from within a GRASS session from the command line, and the **spgrass6** loaded with its dependencies:

```
> system("g.version", intern = TRUE)

[1] "GRASS 6.3.cvs (2007) "

> library(spgrass6)
> gmeta6()

gisdbase    /home/rsb/topics/grassdata
location    spearfish60
mapset      rsb
rows        477
columns     634
north       4928010
south       4913700
west        589980
east        609000
nsres       30
ewres       30
projection  +proj=utm +zone=13 +a=6378206.4 +rf=294.9786982 +no_defs
+nadgrids=/home/rsb/topics/grass63/grass-6.3.cvs/etc/nad/conus
+to_meter=1.0
```

The examples used here are taken from the 'Spearfish' sample data location (South Dakota, USA, 103.86W, 44.49N), perhaps the most typical for GRASS demonstrations. The `gmeta6` function is simply a way of summarising the current settings of the GRASS location and region within which we are working. Data moved from GRASS over the interface will be given category labels if

present. The interface does not support the transfer of factor level labels from R to GRASS, nor does it set colours or quantisation rules. The `readRAST6` command here reads elevation values into a `SpatialGridDataFrame` object, treating the values returned as floating point and the geology categorical layer into a factor:

```
> spear <- readRAST6(c("elevation.dem", "geology"), cat = c(FALSE,
+     TRUE))

> summary(spear)

Object of class SpatialGridDataFrame
Coordinates:
              min     max
coords.x1  589980  609000
coords.x2 4913700 4928010
Is projected: TRUE
proj4string :
[+proj=utm +zone=13 +a=6378206.4 +rf=294.9786982 +no_defs
+nadgrids=/home/rsb/topics/grass63/grass-6.3.cvs/etc/nad/conus
+to_meter=1.0]
Number of points: 2
Grid attributes:
  cellcentre.offset cellsize cells.dim
1            589995       30       634
2           4913715       30       477
Data attributes:
 elevation.dem        geology
 Min.   : 1066   sandstone:74959
 1st Qu.: 1200   limestone:61355
 Median : 1316   shale    :46423
 Mean   : 1354   sand     :36561
 3rd Qu.: 1488   igneous  :36534
 Max.   : 1840   (Other)  :37636
 NA's   :10101   NA's     : 8950
```

When the `cat` argument is set to `TRUE`, the GRASS category labels are imported and used as factor levels; checking back, we can see that they agree:

```
> table(spear$geology)

metamorphic  transition     igneous   sandstone   limestone
      11693         142       36534       74959       61355
      shale sandy shale    claysand        sand
      46423       11266       14535       36561

> system("r.stats --q -cl geology", intern = TRUE)

 [1] "1 metamorphic 11693" "2 transition 142"    "3 igneous 36534"
 [4] "4 sandstone 74959"   "5 limestone 61355"   "6 shale 46423"
 [7] "7 sandy shale 11266" "8 claysand 14535"    "9 sand 36561"
[10] "* no data 8950"
```

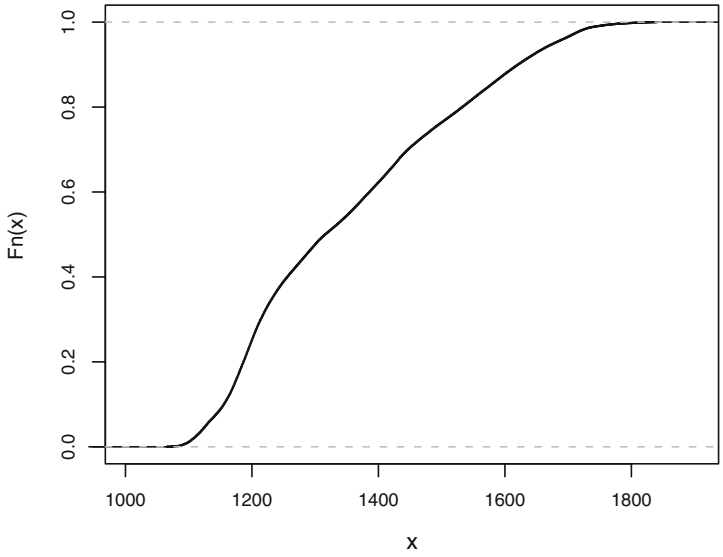

Fig. 4.4. Empirical cumulative distribution function of elevation for the Spearfish location

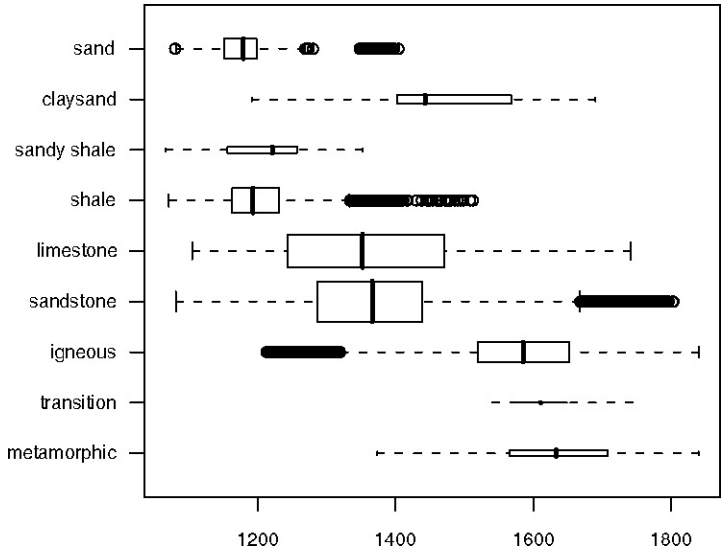

Fig. 4.5. Boxplots of elevation by geology category, Spearfish location

Figure 4.4 shows an empirical cumulative distribution plot of the elevation values, giving readings of the proportion of the study area under chosen elevations. In turn Fig. 4.5 shows a simple boxplot of elevation by geology category, with widths proportional to the share of the geology category in the total area.

We have used the `readRAST6` function to read from GRASS rasters into R; the `writeRAST6` function allows a single named column of a `SpatialGridDataFrame` object to be exported to GRASS.

The **spgrass6** package also provides functions to move vector features and associated attribute data to R and back again; unlike raster data, there is no standard mechanism for handling missing values. The `readVECT6` function is used for importing vector data into R, and `writeVECT6` for exporting to GRASS. The first data set to be imported from GRASS contains the point locations of sites where insects have been monitored, the second is a set of stream channel centre-lines:

```
> bugsDF <- readVECT6("bugsites")

> vInfo("streams")

    points      lines boundaries  centroids      areas    islands
         0        104         12          4          4          4
     faces    kernels
         0          0

> streams <- readVECT6("streams", type = "line,boundary",
+     remove.duplicates = FALSE)
```

The `remove.duplicates` argument is set to TRUE when there are only, for example lines or areas, and the number present is greater than the data count (the number of rows in the attribute data table). The `type` argument is used to override type detection when multiple types are non-zero, as here, where we choose lines and boundaries, but the function guesses areas, returning just filled water bodies.

Because different mechanisms are used for passing information concerning the GRASS location coordinate reference system for raster and vector data, the PROJ.4 strings often differ slightly, even though the actual CRS is the same. We can see that the representation for the point locations of beetle sites does differ here; the vector representation is more in accord with standard PROJ.4 notation than that for the raster layers, even though they are the same. In the summary of the `spear` object above, the ellipsoid was represented by `+a` and `+rf` tags instead of the `+ellps` tag using the `clrk66` value:

```
> summary(bugsDF)

Object of class SpatialPointsDataFrame
Coordinates:
              min     max
coords.x1  590232  608471
coords.x2 4914096 4920512
Is projected: TRUE
proj4string :
[+proj=utm +zone=13 +ellps=clrk66 +datum=NAD27 +units=m
+no_defs +nadgrids=@conus,@alaska,@ntv2_0.gsb,@ntv1_can.dat]
```

```
Number of points: 90
Data attributes:
      cat                 str1
 Min.   : 1.00   Beetle site:90
 1st Qu.:23.25
 Median :45.50
 Mean   :45.50
 3rd Qu.:67.75
 Max.   :90.00
```

This necessitates manual assignment from one representation to the other in some occasions, and is due to GRASS using non-standard but equivalent extensions to PROJ.4.

There are a number of helper functions in the **spgrass6** package, one `gmeta2grd` to generate a GridTopology object from the current GRASS region settings. This is typically used for interpolation from point data to a raster grid, and may be masked by coercion from a SpatialGrid to a SpatialPixels object having set cells outside the study area to NA. A second utility function for vector data uses the fact that GRASS 6 uses a topological vector data model. The `vect2neigh` function returns a data frame with the left and right neighbours of arcs on polygon boundaries, together with the length of the arcs. This can be used to modify the weighting of polygon contiguities based on the length of shared boundaries. Like GRASS, GDAL/OGR, PROJ.4, and other OSGeo projects, the functions offered by **spgrass6** are changing, and current help pages should be consulted to check correct usage.

4.4.1 Broad Street Cholera Data

Even though we know that John Snow already had a working hypothesis about cholera epidemics, his data remain interesting, especially if we use a GIS to find the street distances from mortality dwellings to the Broad Street pump in Soho in central London. Brody et al. (2000) point out that John Snow did not use maps to 'find' the Broad Street pump, the polluted water source behind the 1854 cholera epidemic, because he associated cholera with water contaminated with sewage, based on earlier experience. The accepted opinion of the time was that cholera was most probably caused by a 'concentrated noxious atmospheric influence', and maps could just as easily have been interpreted in support of such a point source.

The specific difference between the two approaches is that the atmospheric cause would examine straight-line aerial distances between the homes of the deceased and an unknown point source, while a contaminated water source would rather look at the walking distance along the street network to a pump or pumps. The basic data to be used here were made available by Jim Detwiler, who had collated them for David O'Sullivan for use on the cover of O'Sullivan and Unwin (2003), based on earlier work by Waldo Tobler and others. The files were a shapefile with counts of deaths at front doors of houses and a

georeferenced copy of the Snow map as an image; the files were registered in the British National Grid CRS. The steps taken in GRASS were to set up a suitable location in the CRS, to import the image file, the file of mortalities, and the file of pump locations.

To measure street distances, the building contours were first digitised as a vector layer, cleaned, converted to raster leaving the buildings outside the street mask, buffered out 4 m to include all the front door points within the street mask, and finally distances measured from each raster cell in the buffered street network to the Broad Street pump and to the nearest other pump. These operations in summary were as follows:

```
v.digit -n map=vsnow4 bgcmd="d.rast map=snow"
v.to.rast input=vsnow4 output=rfsnow use=val value=1
r.buffer input=rfsnow output=buff2 distances=4
r.cost -v input=buff2 output=snowcost_not_broad \
  start_points=vpump_not_broad
r.cost -v input=buff2 output=snowcost_broad start_points=vpump_broad
```

The main operation here is `r.cost`, which uses the value of 2.5 m stored in each cell of `buff2`, which has a resolution of 2.5 m, to cumulate distances from the start points in the output rasters. The operation is carried out for the other pumps and for the Broad Street pump. This is equivalent to finding the line of equal distances shown on the extracts from John Snow's map shown in Brody et al. (2000, p. 65). It is possible that there are passages through buildings not captured by digitising, so the distances are only as accurate as can now be reconstructed.

```
> buildings <- readVECT6("vsnow4")
> sohoSG <- readRAST6(c("snowcost_broad", "snowcost_not_broad"))
```

For visualisation, we import the building outlines, and the two distance rasters. Next we import the death coordinates and counts, and overlay the deaths on the distances, to extract the distances for each house with mortalities – these are added to the `deaths` object, together with a logical variable indicating whether the Broad Street pump was closer (for this distance measure) or not:

```
> deaths <- readVECT6("deaths3")
> o <- overlay(sohoSG, deaths)
> deaths <- spCbind(deaths, as(o, "data.frame"))
> deaths$b_nearer <- deaths$snowcost_broad < deaths$snowcost_not_broad

> by(deaths$Num_Cases, deaths$b_nearer, sum)

INDICES: FALSE
[1] 221
------------------------------------------------
INDICES: TRUE
[1] 357
```

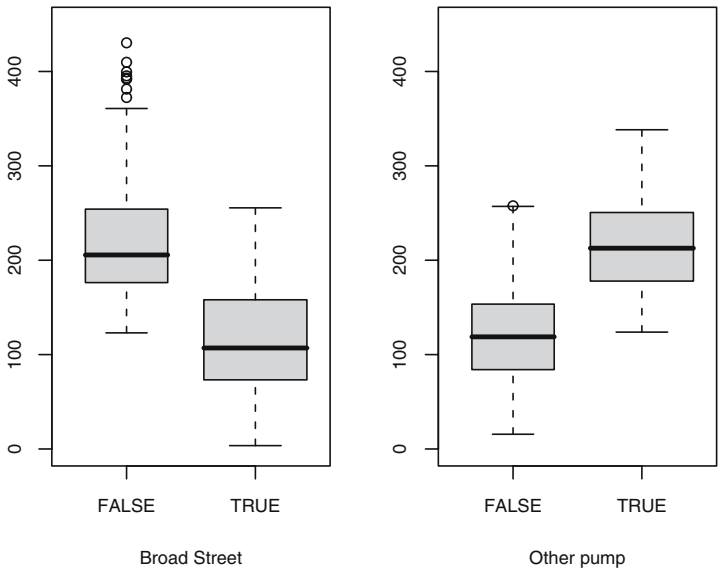

Fig. 4.6. Comparison of walking distances from homes of fatalities to the Broad Street pump or another pump by whether the Broad Street pump was closer or not

There are not only more mortalities in houses closer to the Broad Street pump, but the distributions of distances are such that their inter-quartile ranges do not overlap. This can be seen in Fig. 4.6, from which a remaining question is why some of the cases appear to have used the Broad Street pump in spite of having a shorter distance to an alternative. Finally, we import the locations of the pumps to assemble a view of the situation, shown in Fig. 4.7. The grey scaled streets indicate the distance of each 2.5 m raster cell from the Broad Street pump along the street network. The buildings are overlaid on the raster, followed by proportional symbols for the number of mortalities per affected house, coded for whether they are closer to the Broad Street pump or not, and finally the pumps themselves.

```
> nb_pump <- readVECT6("vpump_not_broad")
> b_pump <- readVECT6("vpump_broad")
```

4.5 Other Import/Export Interfaces

The classes for spatial data introduced in **sp** have made it easier to implement and maintain the import and export functions described earlier in this chapter. In addition, they have created opportunities for writing other interfaces, because the structure of the objects in R is better documented. In this section, a number of such interfaces will be presented, with others to come in

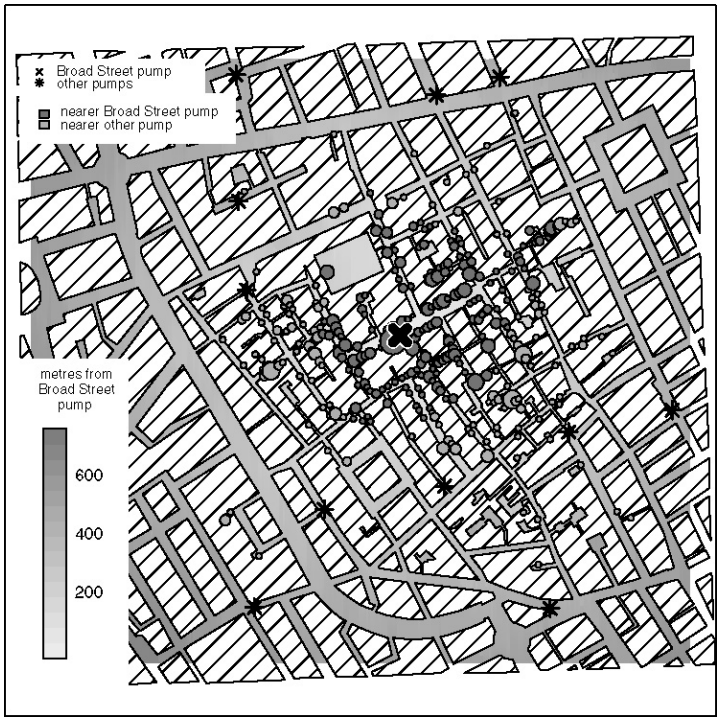

Fig. 4.7. The 1854 London cholera outbreak near Golden Square

the future, hosted in **maptools** or other packages. Before going on to discuss interfaces with external applications, conversion wrappers for R packages will be mentioned.

The **maptools** package contains interface functions to convert selected **sp** class objects to classes used in the **spatstat** for point pattern analysis – these are written as coercion methods to **spatstat** `ppp` and `owin` classes. **maptools** also contains the `SpatialLines2PolySet` and `SpatialPolygons2PolySet` functions to convert **sp** class objects to `PolySet` class objects as defined in the **PBSmapping** package, and a pair of matching functions in the other direction. This package provides a number of GIS procedures needed in fisheries research (PBS is the name of the Pacific Biological Station in Nanaimo, British Columbia, Canada).

There are also interface functions in the **adehabitat** package for conversion between **sp** class objects and **adehabitat** `kasc` and `asc` gridded objects, **adehabitat** `area` polygon objects, and **adehabitat** `traj` and `ltraj` trajectory objects. The package itself is documented in Calenge (2006), and includes many tools for the analysis of space and habitat use by animals.

4.5.1 Analysis and Visualisation Applications

While many kinds of data analysis can be carried out within the R environment, it is often very useful to be able to write out files for use in other applications or for sharing with collaborators not using R. These functions live in **maptools** and will be extended as required. The `sp2tmap` function converts a `SpatialPolygons` object for use with the Stata™ tmap contributed command,[20] by creating a data frame with the required columns. The data frame returned by the function is exported using `write.dta` from the **foreign** package, which should also be used to export the attribute data with the polygon tagging key. The `sp2WB` function exports a `SpatialPolygons` object as a text file in S-PLUS™ map format to be imported by WinBUGS.

The **GeoXp** package provides some possibilities for interactive statistical data visualisation within R, including mapping. The R graphics facilities are perhaps better suited to non-interactive use, however, especially as it is easy to write data out to Mondrian.[21] Mondrian provides fully linked multiple plots, and although the screen can become quite 'busy', users find it easy to explore their data in this environment. The function `sp2Mondrian` in **maptools** writes out two files, one with the data, the other with the spatial objects from a `SpatialPolygonsDataFrame` object for Mondrian to read; the polygon format before Mondrian 1.0 used a single file and may still be used, controlled by an additional argument.

```
> sp2Mondrian(scot_BNG, "scot_BNG.txt")
```

The example we use here is the Scottish lip cancer `SpatialPolygons-DataFrame` object in the British National Grid projection (Mondrian assumes a planar map). A screen shot of two Mondrian plots is shown in Fig. 4.8, with a map view and a parallel boxplot, where a group of districts has been selected on the map, and is shown as a subset on the linked display.

4.5.2 TerraLib and aRT

The **aRT** package[22] provides an advanced modular interface to TerraLib.[23] TerraLib is a GIS classes and functions library intended for the development of multiple GIS tools. Its main aim is to enable the development of a new generation of GIS applications, based on the technological advances on spatial databases. TerraLib defines the way that spatial data are stored in a database system, and can use MySQL, PostgreSQL, Oracle, or Access as a back-end. The library itself can undertake a wide range of GIS operations on the data stored in the database, as well as storing and retrieving the data as spatial objects from the database system.

[20] `http://www.stata.com/search.cgi?query=tmap`.
[21] `http://rosuda.org/Mondrian/`.
[22] `http://leg.ufpr.br/aRT/`.
[23] `http://www.terralib.org/`.

Fig. 4.8. Screen shot of two linked Mondrian plots: a map of the Empirical Bayes smooth and a parallel boxplot for four variables, with the selected districts on the map (three northern mainland counties, Outer Hebrides, Orkney, and Shetland) split out as overlay boxplots

The **aRT** package interfaces **sp** classes with TerraLib classes, permitting data to flow between R, used as a front-end system interacting with the user, through TerraLib and the back-end database system. One of the main objectives of **aRT** is to do spatial queries and operations in R. Because these operations are written to work efficiently in TerraLib, a wide range of overlay and buffering operations can be carried out, without them being implemented in R itself. Operations on the geometries, such as whether they touch, how far apart they are, whether they contain holes, polygon unions, and many others, can be handed off to TerraLib.

A further innovation is the provision of a wrapper for the R compute engine, allowing R with **aRT** to be configured with TerraLib between the back-end database system and a front-end application interacting with the user. This application, for example TerraView, can provide access through menus to spatial data analysis functionality coded in R using **aRT**.[24] All of this software is released under open source licences, and offers considerable opportunities for building non-proprietary customised systems for medium and larger organisations able to commit resources to C++ programming. Organisations running larger database systems are likely to have such resources anyway, so **aRT** and TerraLib provide a real alternative for fresh spatial data handling projects.

[24] Andrade Neto and Ribeiro Jr. (2005).

4.5.3 Other GIS and Web Mapping Systems

The **Rpad** package[25] is a general-purpose interactive, web-based analysis program, and Rpad pages are interactive workbook-type sheets based on R. Some of the examples[26] use spatial data directly, and it is quite practical to handle smaller or moderate-sized point data without needing bulky client-side applets. Integration with R results is automatic, without the need for much extra software on the server side.

An interface package – **RSAGA** – has been provided for SAGA GIS;[27] like the GRASS 6 interface, it uses `system` to pass commands to external software.

For raster or polygon data, it may be sensible to use MapServer[28] or alternative software. MapServer integrates and renders spatial data for the web, and because it uses the same open source geospatial toolset as the packages described in this chapter (PROJ.4, GDAL/OGR), it is possible to add R as a compute engine to websites running MapServer with little difficulty. It is also possible to set up MapServer as a Web Feature Server, which serves the actual map data encapsulated in XML wrappers instead of rendering the map for display on a web browser; Mitchell (2005) contains a good deal of information on providing web mapping facilities.

In the discussion above, integration between R and GIS has principally taken the form of file transfer. It is possible to use other mechanisms, similar in nature to the embedding of R in TerraView using **aRT**. One example is given by Tait et al. (2004), using the R `StatConnector` (D)COM mechanism to use R as a back-end from ArcGIS™. The specific context is the need to provide epidemiologists using ArcGIS™ for animal disease control and detection with point pattern analysis tools, using a GIS interface. The prototype was a system using **splancs** running in R to calculate results from data passed from ArcGIS™, with output passed back to ArcGIS™ for display. A practical difficulty of embedding both R and **splancs** on multiple workstations is that of software installation and maintenance.

A second example is ArcRstats,[29] for producing multivariate habitat prediction rasters using ArcGIS™ and R for interfacing classification and regression trees, generalised linear models, and generalised additive models. It is implemented using the Python interface introduced into ArcGIS™ from version 9, and then the Python `win32com.client` module to access the R `StatConnector` (D)COM mechanism. The current release of ArcRstats uses the **sp**, **maptools**, and **rgdal** packages to interface spatial data, and **RODBC** to work with Access formated geodatabases, in addition to a wide range of analysis functions, including those from the **spatstat** package.

[25] `http://www.rpad.org/Rpad/`.
[26] `http://www.rpad.org/Rpad/InterruptionMap.Rpad`.
[27] `http://www.saga-gis.uni-goettingen.de/html/index.php`.
[28] `http://mapserver.gis.umn.edu/`.
[29] `http://www.env.duke.edu/geospatial/software/`.

The marine geospatial ecology tools project[30] follows up the work begun in ArcRstats, providing for execution in many environments, and using the Python route through the COM interface to ArcGIS™. It is not hard to write small Python scripts to interface R and ArcGIS™ through temporary files and the `system` function. This is illustrated by the **RPyGeo** package, which uses R to write Python scripts for the ArcGIS™ geoprocessor. The use of R Python interface is not as fruitful as it might be, because ArcGIS™ bundles its own usually rather dated version of Python, and ArcGIS™ itself only runs on Windows and is rather expensive, certainly compared to GRASS.

4.6 Installing rgdal

Because **rgdal** depends on external libraries, on GDAL and PROJ.4, and particular GDAL drivers may depend on further libraries, installation is not as easy as with self-contained R packages. Only the Windows binary package is self-contained, with a basic set of drivers available. For Linux/Unix and MacOSX, it is necessary to install **rgdal** from source, after first having installed the external dependencies. Users of open source GIS applications such as GRASS will already have GDAL and PROJ.4 installed anyway, because they are required for such applications.

In general, GDAL and PROJ.4 will install from source without difficulty, but care may be required to make sure that libraries needed for drivers are available and function correctly. If the programs `proj`, `gdalinfo`, and `ogrinfo` work correctly for data sources of interest after GDAL and PROJ.4 have been installed, then **rgdal** will also work correctly. Mac OSX users may find William Kyngesburye's frameworks[31] a useful place to start, if installation from source seems forbidding. More information is available on the 'maps' page at the Rgeo website,[32] and by searching the archives of the R-sig-geo mailing list.

Windows users needing other drivers, and for whom conversion using programs in the FWTools[33] binary for Windows is not useful, may choose to install **rgdal** from source, compiling the **rgdal** DLL with VC++ and linking against the FWTools DLLs – see the `inst/README.windows` file in the source package for details.

[30] `http://code.env.duke.edu/projects/mget`.
[31] `http://www.kyngchaos.com/software/unixport/frameworks`.
[32] `http://www.r-project.org/Rgeo`.
[33] `http://fwtools.maptools.org`.