Chapter 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 Goodal, 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) and Hall and Leahy (2008). 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.

[1] http://www.osgeo.org/.

[2] http://www.gdal.org/.

[3] http://trac.osgeo.org/proj/.

R.S. Bivand et al., *Applied Spatial Data Analysis with R*, Use R! 10,
DOI 10.1007/978-1-4614-7618-4_4,

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

```
> library(rgdal)
```

```
rgdal: version: 0.8-5, (SVN revision 449)
Geospatial Data Abstraction Library extensions to R successfully loaded
Loaded GDAL runtime: GDAL 1.9.2, released 2012/10/08
Path to GDAL shared files: /usr/local/share/gdal
Loaded PROJ.4 runtime: Rel. 4.8.0, 6 March 2012, [PJ_VERSION: 480]
Path to PROJ.4 shared files: (autodetected)
```

We see that the release version numbers and dates of the external dynamically loaded libraries are reported. In addition, the values of the package version, code revision number, and paths to GDAL and PROJ.4 metadata directories are reported.[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 not unusual.

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

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

The PROJ.4 library does not report the version of the EPSG list distributed with releases of PROJ.4, but this may be discovered by reading the NEWS file and extracting lines reporting the PROJ.4 release from newest to oldest:

```
> NEWS <- "http://svn.osgeo.org/metacrs/proj/trunk/proj/NEWS"
> PROJ4_NEWS <- readLines(url(NEWS))

> lns <- grep("Release Notes|EPSG", PROJ4_NEWS)
> head(PROJ4_NEWS[lns])

[1] "4.8.0 Release Notes"
[2] " o Upgrade to EPSG 7.9.  Some changes in ideal datum selection."
[3] "4.7.0 Release Notes"
[4] " o Regenerated nad/epsg init file with EPSG 7.1 database,
     including new"
[5] "   support for Google Mercator (EPSG:3857)."
[6] "4.6.1 Release Notes"
```

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. Copies of the list are provided in GDAL and PROJ.4, and in Windows and OSX binary

[5] http://www.epsg.org/.

rgdal packages,[6] because the list 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 very much 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 formerly given by the Royal Netherlands Navy entitled 'From ED50 towards WGS84, or does your GPS receiver tell you the truth?' 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. 84). In this case to save space, the search string has been chosen to match exactly the row needed; entering just `ED50` gives 35 hits:

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

    code   note
159 4230 # ED50
                                                                    prj4
159 +proj=longlat +ellps=intl +towgs84=-87,-98,-121,0,0,0,0 +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.[7] 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

[6] See installation note at chapter end, p. 125.

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

each other.[8] 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
+towgs84=-87,-98,-121,0,0,0,0 +no_defs
```

The three tags that are known in this version of the EPSG list are `+proj` – projection, which takes the value `longlat` for geographical coordinates – `+ellps` – ellipsoid, with value `intl` for the International Ellipsoid of 1909 (Hayford), and `+towgs84`, with a vector of three non-zero parameters for spatial translation (in geocentric space, ΔX, ΔY, ΔZ). Had seven parameters been given, they would permit shifting by translation + rotation + scaling; note that sources may vary in the signs of the parameters. There was no `+towgs84` tag given for this EPSG code in the first edition of this book, using EPSG 6.13,[9] and so without further investigation it was then 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; adding $\Delta X = 87 \pm 3$ m, $\Delta Y = -96 \pm 3$ m, $\Delta Z = -120 \pm 3$ m, with a sign change on ΔX, gives an alternative specification (note that the EPSG `+towgs84` values are within the tolerances of the Grids & Datums values):

```
> 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
```

When **rgdal** is loaded in the running R session, the proposed tags are verified against the valid set, and additions, as here, override those drawn from the EPSG list. 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. 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.

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

[9] The version shipped with PROJ.4 4.8.0 is EPSG 7.9 as we saw above.

[10] http://www.asprs.org/Grids-Datums.html.

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),
+     proj4string = 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))

        x
-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. If we use the three translation parameters provided by the current EPSG list instead of those given in Grids and Datums, we find that the distance is almost 2 m greater, and the azimuth is slightly changed:

```
> proj4string(IJ.ED50) <- CRS("+init=epsg:4230")
> res <- spTransform(IJ.ED50, CRS("+proj=longlat +datum=WGS84"))
> spDistsN1(coordinates(IJ.ED50), coordinates(res), longlat = TRUE) *
+     1000

[1] 125.8692

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

        x
-133.8915
```

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
626  2163       # US National Atlas Equal Area
2328 3978       # NAD83 / Canada Atlas Lambert
2329 3979 # NAD83(CSRS) / Canada Atlas Lambert

> 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
52 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 open source GIS (GRASS Development Team, 2012) from version 6.0 and subsequent releases 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.[11]

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 2.8, among others in earlier chapters, have been made using these sources.

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

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™.[12] 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. 131 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**,[13] 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 OSGeo4W Windows releases, which typically include a wide range of drivers.[14] On the system used for building this book, the first 10 drivers listed by `ogrDrivers` are:

```
> head(ogrDrivers(), n = 10)

         name write
1  AeronavFAA FALSE
2      ARCGEN FALSE
3      AVCBin FALSE
4      AVCE00 FALSE
5         BNA  TRUE
```

[12] The format is fully described in this white paper:
http://www.esri.com/library/whitepapers/pdfs/shapefile.pdf.

[13] See installation note at chapter end.

[14] http://trac.osgeo.org/osgeo4w/.

```
6      CouchDB  TRUE
7          CSV  TRUE
8          DGN  TRUE
9          DXF  TRUE
10      EDIGEO FALSE
```

Functions using OGR are based on the concepts of *data source name* and *layer*, both required to access data and metadata. In some circumstances, we need to find out which layers are offered by a data source, using the function `ogrListLayers` taking a data source name argument; an example is given on p. 97. The ways in which the data source name and layer arguments are specified and may differ forms for different drivers and it is worth reading the relevant web pages[15] for the format being imported. In some cases, a data source name contains only one layer, but in other cases many layers may be present.

Recent releases of OGR have included facilities for handling the encoding of strings, both the values of string fields and of field names. These are diffusing to drivers for different file formats slowly, but have already caused difficulties for Windows users of CP1252 (a codepage specifying encoding) and the ESRI™ Shapefile driver. An analysis of this issue is provided in a vignette:

```
> vignette("OGR_shape_encoding", package = "rgdal")
```

which explains how to prevent the OGR driver trying to modify the encoding, thus making it possible to keep the data representation the same in R and ArcGIS™.

The `readOGR` and `ogrInfo` functions take at least two arguments – the data source name (`dsn`) and the layer (`layer`). 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 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 functions 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).[16] 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):

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

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

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

We can use the ogrInfo function to show a summary of the layer before we read it. The function returns an object containing information, shown with a print method, and reporting the data source name, the layer name, the driver, number of features, and featyre type. If a coordinate reference system is associated with the layer, it is shown, but here, none is available. The shapefile appears to be in geographical coordinates, as we see from its extent, showing the coordinates of the bounding box of the vector features. The LDID value for ESRI™ Shapefiles is explaining in the OGR encoding vignette mentioned on p. 93.

```
> ogrInfo(".", "scot")
```

```
Source: ".", layer: "scot"
Driver: ESRI Shapefile number of rows 56
Feature type: wkbPolygon with 2 dimensions
Extent: (-8.621389 54.62722) - (-0.7530556 60.84444)
LDID: 0
Number of fields: 2
  name type length typeName
1 NAME    4     16   String
2   ID    2     16     Real
```

We also see that ogrInfo retrieves information on the fields of attribute data provided in the layer. The name of the field will be used in the "data" slot of the imported object, and its type will be "integer" or "numeric" for matching numeric input data, although many drivers return "numeric" where "integer" would be more appropriate. In readOGR, input strings (including all date and time fields read as strings) are converted to factors if the stringsAsFactors argument is TRUE; the argument defaults to the value returned by default.stringsAsFactors.

No *.prj file is present, so, after importing from the working directory with a missing CRS value, we assign a suitable coordinate reference system.

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

```
OGR data source with driver: ESRI Shapefile
Source: ".", layer: "scot"
with 56 features and 2 fields
Feature type: wkbPolygon with 2 dimensions
```

```
> proj4string(scot_LL)
```

```
[1] NA
```

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

As indicated above, when we get the classes of variables in the "data" slot of scot_LL, we see that the input string field has been converted to factor representation, and the numeric ID field could just as well been an integer:

```
> sapply(slot(scot_LL, "data"), class)
```

```
     NAME        ID
 "factor" "numeric"
```

```
> 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:

```
> 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
```

```
> ID_D <- match(scot_LL$ID, scot_dat$District)
> scot_dat1 <- scot_dat[ID_D, ]
> row.names(scot_dat1) <- row.names(scot_LL)
> library(maptools)
> scot_LLa <- spCbind(scot_LL, scot_dat1)
> all.equal(scot_LLa$ID, scot_LLa$District)
```

```
[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. 10, 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"))
```

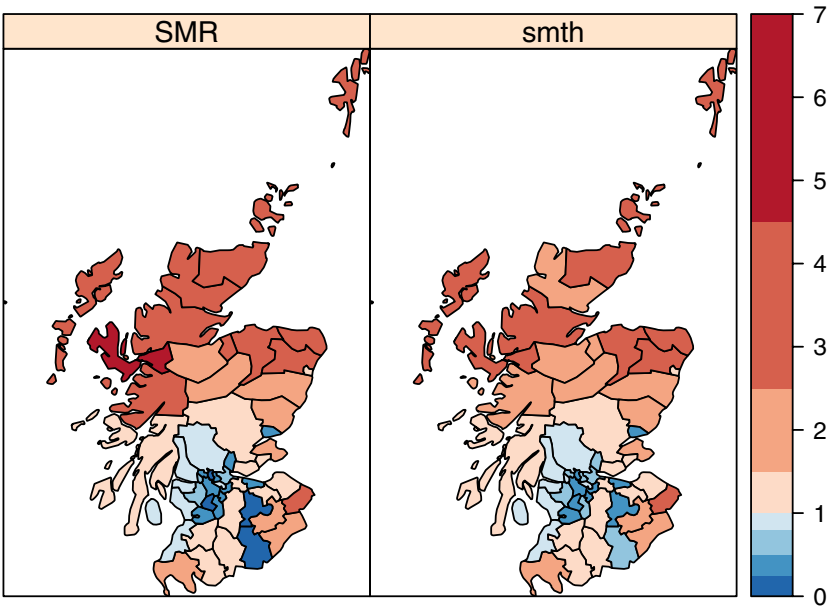

Fig. 4.1 Comparison of relative risk (SMR) and EB smoothed relative risk (smth) for Scottish lip cancer

We may export `SpatialPointsDataFrame`, `SpatialLinesDataFrame` and `SpatialPolygonsDataFrame` objects using the `writeOGR` function in **rgdal** to file formats for which output drivers are implemented. As an example, we can export the projected Scottish data set, including our added results, to a shapefile, using `driver="ESRI Shapefile"`, or to other file formats:

```
> 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"
```

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. As mentioned above, the `ogrDrivers` function can be used to see which drivers are available. Some, like `driver="ESRI Shapefile"`, can represent points, lines or polygons, and are intended for general vector data exchange. The OpenGIS® specifications include several XML-based formats, initially the Geography Markup Language (GML), but more recently followed by the Keyhole Markup Language (KML) brought forward by Google™. The GML format is also used within the OpenGIS®Web Feature Service (WFS), in which the server is the data source name, and the layer is a table of geographical data. Let us use the OGR WFS driver to access European Forest

Fire Information System data from the Joint Research Centre of the European Commission, adding a driver-specific prefix to the URL of the service; we can use `ogrListLayers` to check for available layers:

```
> dsn <- "WFS:http://geohub.jrc.ec.europa.eu/effis/ows"
> ogrListLayers(dsn)

[1] "EFFIS:FireNews"      "EFFIS:Fires30Days"   "EFFIS:Fires7Days"
[4] "EFFIS:FiresAll"      "EFFIS:Hotspots1Day"  "EFFIS:Hotspots7Days"
[7] "EFFIS:HotspotsAll"

> Fires <- readOGR(dsn, "EFFIS:FiresAll")

OGR data source with driver: WFS
Source: "WFS:http://geohub.jrc.ec.europa.eu/effis/ows", layer:
     "EFFIS:FiresAll"
with 1770 features and 11 fields
Feature type: wkbPoint with 2 dimensions

> names(Fires)

 [1] "gml_id"     "FireDate"   "Country"    "Province"   "Commune"
 [6] "Area_HA"    "CountryFul" "Class"      "X"          "Y"
[11] "LastUpdate"
```

The data downloaded are those in the current database,[17] and will change as new incidents accrue. First we create a bounding box covering the relevant parts of Europe, and use it to make a spatial selection of only the coastlines and national boundaries taken from the `wrld_simpl` data set included in **maptools**, falling within the box with `gIntersection`, a binary topological operator in the **rgeos** package. Subsetting the coastlines reduces plotting times, because the plotting method does not need to discard data far outside its data window.

```
> x <- c(-15, -15, 38, 38, -15)
> y <- c(28, 62, 62, 28, 28)
> crds <- cbind(x = x, y = y)
> bb <- SpatialPolygons(list(Polygons(list(Polygon(coords = crds)),
+     "1")))
> library(maptools)
> data(wrld_simpl)
> proj4string(bb) <- CRS(proj4string(wrld_simpl))
> library(rgeos)
> slbb <- gIntersection(bb, as(wrld_simpl, "SpatialLines"))
> spl <- list("sp.lines", slbb, lwd = 0.7, col = "khaki4")
```

Next we convert the input fire date to a `Date` object, then discard any incidents on Réunion. Using the new space-time classes introduced in Chap. 6 and specified in the **spacetime** package, we can plot three incident maps using the `stplot` method, conditioned by time quantiles, as shown in Fig. 4.2:

[17] Those used here were accessed on 2012-01-04.

```
> Fires$dt <- as.Date(as.character(Fires$FireDate), format = "%d-%m-%Y")
> Fires0 <- Fires[-which(coordinates(Fires)[, 2] < 0),
+     ]
> Fires1 <- Fires0[order(Fires0$dt), ]
> library(spacetime)
> Fires2 <- STIDF(as(Fires1, "SpatialPoints"), Fires1$dt,
+     as(Fires1, "data.frame"))
> stplot(Fires2, number = 3, sp.layout = spl, cex = 0.5)
```

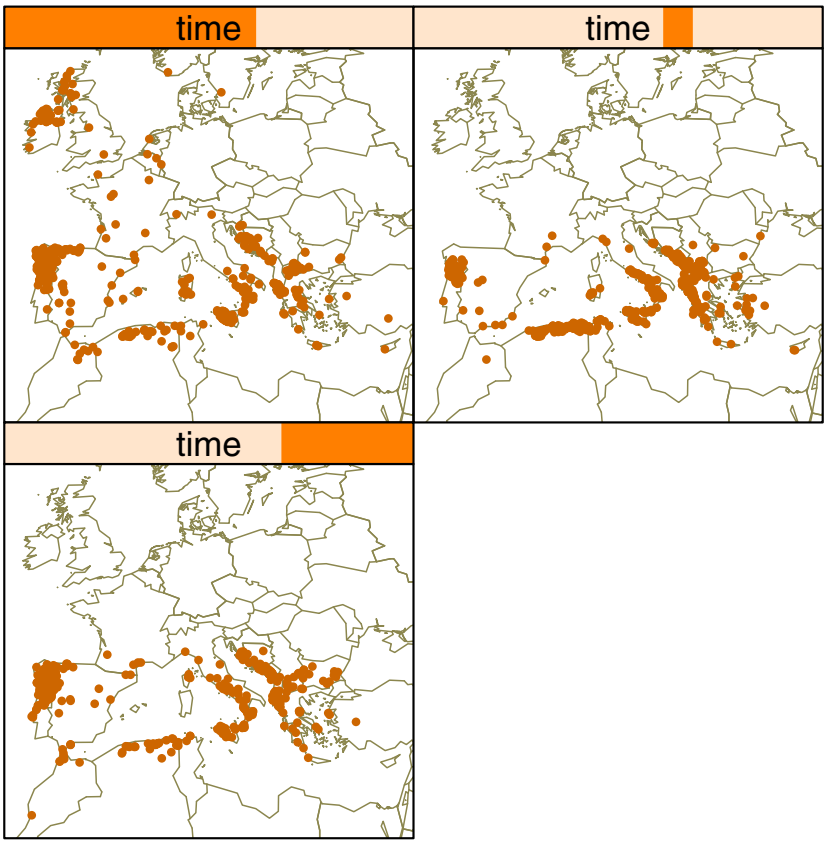

Fig. 4.2 Space-time plot of forest fire incidents, conditioned on time by quantiles: *upper left panel* – first third of incidents, *upper right panel* – second third, *lower left panel* – final third

The OGR GPX driver may be used to exchange data with GPS devices, using a simple XML representation. We may here prepare a file for uploading the fire locations recorded for Greece, for example to permit sites to be surveyed at fixed post-fire intervals; we must re-name the ID column to the GPX mandatory `"name"`:

```
> names(Fires1)[1] <- "name"
> GR_Fires <- Fires1[Fires1$Country == "GR", ]
> writeOGR(GR_Fires, "EFFIS.gpx", "waypoints", driver = "GPX",
+     dataset_options = "GPX_USE_EXTENSIONS=YES")
```

The use of the `dataset_options` argument permits the inclusion of identifying data in the GPX file, which may be accessed on suitable GPS devices. In this case, the retrieved values for the first incident may be shown as:

```
> GR <- readOGR("EFFIS.gpx", "waypoints")

OGR data source with driver: GPX
Source: "EFFIS.gpx", layer: "waypoints"
with 89 features and 34 fields
Feature type: wkbPoint with 2 dimensions

> GR[1, c(5, 24:28)]

        coordinates          name ogr_FireDate ogr_Country
1 (22.261, 39.4258) FiresAll.1552   26-06-2011          GR
  ogr_Province     ogr_Commune ogr_Area_HA
1       Larisa Dimos Krannonos         376
```

The Keyhole Markup Language (KML) is a further XML-based OGR driver, to which we return in Sect. 4.4 below.

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 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 is a function to read vector data from shapefiles: `readShapeSpatial`. It is matched by an equivalent exporting function: `writeSpatialShape`, 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 **ReadImages** 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 +datum=WGS84 +no_defs +ellps=WGS84
+towgs84=0,0,0]
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: 0x3145730>

> getDriverLongName(xx)

[1] "GeoTIFF"

> x

An object of class "GDALReadOnlyDataset"
Slot "handle":
<pointer: 0x85ad850>

> 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
lower left origin.x        174.2
lower left origin.y        -37.5
res.x       0.0008333333
res.y       0.0008333333
ysign       -1
oblique.x   0
oblique.y   0
driver      GTiff
projection  +proj=longlat +datum=WGS84 +no_defs
```

```
file        70042108.tif
apparent band summary:
   GDType hasNoDataValue NoDataValue blockSize1 blockSize2
1 Float32          FALSE           0          1       1320
apparent band statistics:
         Bmin       Bmax Bmean Bsd
1 -4294967295 4294967295    NA  NA
Metadata:
AREA_OR_POINT=Area
TIFFTAG_RESOLUTIONUNIT=1 (unitless)
TIFFTAG_SOFTWARE=IMAGINE TIFF Support
Copyright 1991 - 1999 by ERDAS, Inc. All Rights Reserved
@(#)$RCSfile: etif.c $ $Revision: 1.11 $ $Date$
TIFFTAG_XRESOLUTION=1
TIFFTAG_YRESOLUTION=1
```

While `Spatial` objects do not contain a record of their symbolic representation (see p. 29), it is possible to quantise numerical bands and associate them with colour tables when exporting raster data using some drivers. We can see here that the same colour table is retrieved from a `GTiff` file; the ways in which colour tables are handled varies considerably from driver to driver. The colour table is placed in a `list` because they are associated with raster bands, possibly one for each band in a multi-band raster, so the `colorTable=` argument to `writeGDAL` must be `NULL` or a list of length equal to the number of bands. Note that the integer raster values pointing to the colour table are zero-based, that is, the index for looking up values in the colour table starts at zero for the first colour, and so on. Care is needed to move the missing value beyond values pointing to the colour table; note that the colours have lost their alpha-channel settings. More examples are given in the `writeGDAL` help page.

```
> brks <- c(0, 10, 20, 50, 100, 150, 200, 300, 400, 500,
+     600, 700)
> pal <- terrain.colors(11)
> pal

 [1] "#00A600FF" "#2DB600FF" "#63C600FF" "#A0D600FF" "#E6E600FF"
 [6] "#E8C727FF" "#EAB64EFF" "#ECB176FF" "#EEB99FFF" "#F0CFC8FF"
[11] "#F2F2F2FF"

> length(pal) == length(brks) - 1

[1] TRUE

> auck_el1$band1 <- findInterval(auck_el1$band1, vec = brks,
+     all.inside = TRUE) - 1
> writeGDAL(auck_el1, "demIndex.tif", drivername = "GTiff",
+     type = "Byte", colorTable = list(pal), mvFlag = length(brks) -
+         1)
> Gi <- GDALinfo("demIndex.tif", returnColorTable = TRUE)
> CT <- attr(Gi, "ColorTable")[[1]]
> CT[CT > "#000000"]
```

```
 [1] "#00A600" "#2DB600" "#63C600" "#A0D600" "#E6E600" "#E8C727"
 [7] "#EAB64E" "#ECB176" "#EEB99F" "#F0CFC8" "#F2F2F2"
```

We use the Meuse grid data set to see how data may be written out using GDAL.[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 <- idw(log(zinc) ~ 1, meuse, meuse.grid)["var1.pred"]

> summary(log_zinc)

Object of class SpatialPixelsDataFrame
Coordinates:
     min    max
x 178440 181560
y 329600 333760
Is projected: TRUE
proj4string :
[+init=epsg:28992 +proj=sterea +lat_0=52.15616055555555
+lon_0=5.38763888888889 +k=0.9999079 +x_0=155000 +y_0=463000
+ellps=bessel
+towgs84=565.417,50.3319,465.552,-0.398957,0.343988,-1.8774,4.0725
+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", drivername = "GTiff",
+     type = "Float32", options = "INTERLEAVE=PIXEL")

> GDALinfo("log_zinc.tif")

rows        104
columns     78
bands       1
lower left origin.x        178440
lower left origin.y        329600
res.x       40
res.y       40
ysign       -1
oblique.x   0
```

[18] The current EPSG list provides `+towgs84` parameter values, which were not present in earlier versions of that list.

```
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
+towgs84=565.417,50.3319,465.552,-0.398957,0.343988,-1.8774,4.0725
+units=m +no_defs
file        log_zinc.tif
apparent band summary:
   GDType hasNoDataValue NoDataValue blockSize1
1 Float32          FALSE           0         26
  blockSize2
1         78
apparent band statistics:
         Bmin       Bmax Bmean Bsd
1 -4294967295 4294967295    NA  NA
Metadata:
AREA_OR_POINT=Area
```

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

In much the same way that `writeGDAL` can write per-band colour tables to exported files for some drivers (see p. 102), the same can be done with category names. The same remarks with respect to zero-based indexing also apply, so that zero in the raster points to the first category name. On import using `readGDAL` with the default value of the `as.is=` argument of `FALSE`, the integer band will be associated with the category names in the file and converted to `"factor"`:

```
> Soil <- meuse.grid["soil"]
> table(Soil$soil)

   1    2    3
1665 1084  354

> Soil$soil <- as.integer(Soil$soil) - 1
> Cn <- c("Rd10A", "Rd90C/VII", "Bkd26/VII")
> writeGDAL(Soil, "Soil.tif", drivername = "GTiff", type = "Byte",
+     catNames = list(Cn), mvFlag = length(Cn))
> Gi <- GDALinfo("Soil.tif", returnCategoryNames = TRUE)
> attr(Gi, "CATlist")[[1]]

[1] "Rd10A"     "Rd90C/VII" "Bkd26/VII"

> summary(readGDAL("Soil.tif"))

Soil.tif has GDAL driver GTiff
and has 104 rows and 78 columns
Input level values and names
0 Rd10A
1 Rd90C/VII
2 Bkd26/VII
Object of class SpatialGridDataFrame
```

```
Coordinates:
     min    max
x 178440 181560
y 329600 333760
Is projected: TRUE
proj4string :
[+proj=sterea +lat_0=52.15616055555555 +lon_0=5.38763888888889
+k=0.9999079 +x_0=155000 +y_0=463000 +ellps=bessel
+towgs84=565.417,50.3319,465.552,-0.398957,0.343988,-1.8774,4.0725
+units=m +no_defs]
Grid attributes:
  cellcentre.offset cellsize cells.dim
x            178460       40        78
y            329620       40       104
Data attributes:
    Rd10A Rd90C/VII Bkd26/VII      NA's
     1665      1084       354      5009
```

The range of drivers available for raster data is vast, and steadily increasing. The `gdalDrivers` function shows those available, including chosen properties; here are the first 10 on the book production platform:

```
> head(gdalDrivers(), n = 10)
```

```
      name                             long_name create  copy
1  AAIGrid                 Arc/Info ASCII Grid  FALSE  TRUE
2     ACE2                                ACE2  FALSE FALSE
3     ADRG        ARC Digitized Raster Graphics   TRUE FALSE
4      AIG                Arc/Info Binary Grid  FALSE FALSE
5   AirSAR           AirSAR Polarimetric Image  FALSE FALSE
6      BAG          Bathymetry Attributed Grid  FALSE FALSE
7   BIGGIF   Graphics Interchange Format (.gif)  FALSE FALSE
8      BLX                Magellan topo (.blx)  FALSE  TRUE
9      BMP MS Windows Device Independent Bitmap   TRUE FALSE
10     BSB          Maptech BSB Nautical Charts  FALSE FALSE
```

For example, there is now an R driver for storing portable `SpatialGridDataFrame` objects:

```
> writeGDAL(log_zinc, fname = "log_zinc.rda", drivername = "R")
```

```
> GDALinfo("log_zinc.rda")
```

```
rows        104
columns     78
bands       1
lower left origin.x        178440
lower left origin.y        329600
res.x       40
res.y       40
ysign       -1
oblique.x   0
oblique.y   0
driver      R
projection  +proj=sterea +lat_0=52.15616055555555
```

```
+lon_0=5.38763888888889 +k=0.9999079 +x_0=155000
+y_0=463000 +ellps=bessel
+towgs84=565.417,50.3319,465.552,-0.398957,0.343988,-1.8774,4.0725
+units=m +no_defs
file        log_zinc.rda
apparent band summary:
   GDType hasNoDataValue NoDataValue blockSize1
1 Float64          FALSE           0          1
  blockSize2
1         78
apparent band statistics:
         Bmin       Bmax Bmean Bsd
1 -4294967295 4294967295    NA  NA
Metadata:
R_OBJECT_NAME=gg
```

As in the vector case, GDAL now supports a range of web services. The OpenGIS®Web Map Service (WMS) driver can be used with a local XML file describing the service, and customised offset, region size (both in raster cells, ordered: northings, eastings), and output size. This example reads a raster version of OpenStreetMap[19] data for the centre of Bergen, Norway:

```
> service_xml <- "frmt_wms_openstreetmap_tms.xml"
> offset <- c(19339000, 34546000)
> osm <- readGDAL(service_xml, offset = offset, region.dim = c(2000,
+     2000), output.dim = c(1000, 1000))

frmt_wms_openstreetmap_tms.xml has GDAL driver WMS
and has 67108864 rows and 67108864 columns

> summary(osm)

Object of class SpatialGridDataFrame
Coordinates:
      min       max
x  592129  593323.3
y 8487754 8488948.3
Is projected: TRUE
proj4string :
[+proj=merc +lon_0=0 +k=1 +x_0=0 +y_0=0 +a=6378137 +b=6378137
+units=m +no_defs]
Grid attributes:
  cellcentre.offset cellsize cells.dim
x          592129.6 1.194329      1000
y         8487754.5 1.194329      1000
Data attributes:
     band1           band2           band3
 Min.   :  0.0   Min.   :  0.0   Min.   :  0.0
 1st Qu.:202.0   1st Qu.:208.0   1st Qu.:208.0
 Median :240.0   Median :232.0   Median :227.0
 Mean   :223.5   Mean   :219.3   Mean   :215.1
 3rd Qu.:241.0   3rd Qu.:238.0   3rd Qu.:232.0
 Max.   :255.0   Max.   :255.0   Max.   :255.0
```

[19] http://www.openstreetmap.org/, this selection corresponds to http://www.openstreetmap.org/?lat=60.39542&lon=5.32233&zoom=16&layers=C.

Figure 4.3 shows the retrieved data, which has been extracted from the tiled OpenStreetMap global raster database.

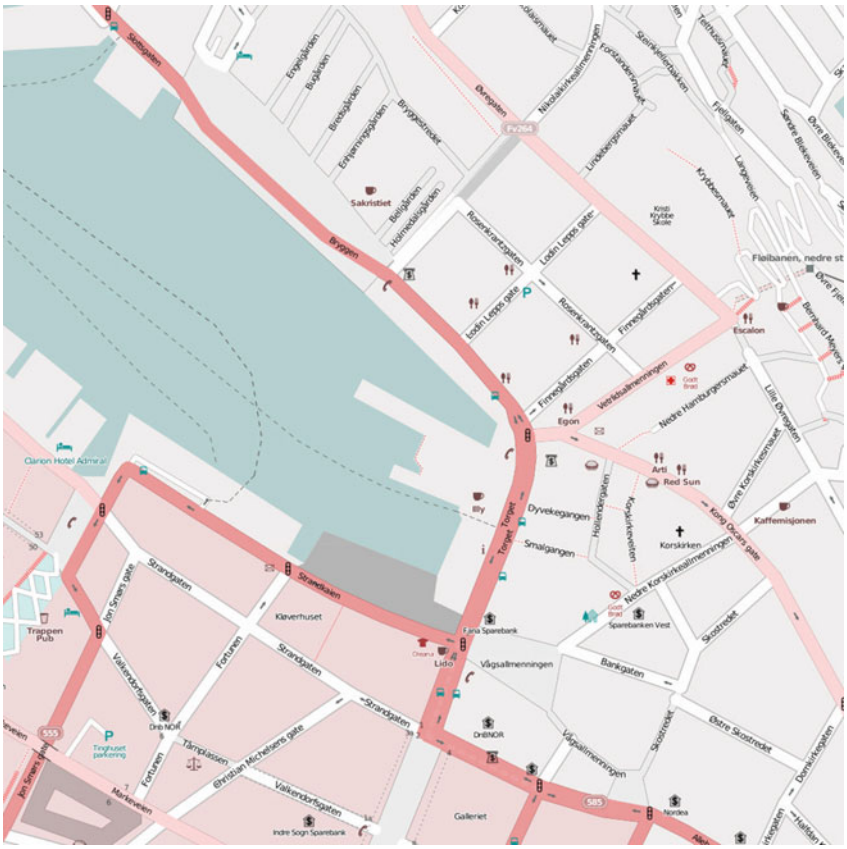

Fig. 4.3 Use of the WMS GDAL driver to retrieve OpenStreetMap raster data for the centre of Bergen, Norway (©OpenStreetMap contributors, CC-BY-SA)

4.3.2 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 Google Earth™, Google Maps™ and Other Formats

As we have seen above, web-based services are becoming ever more important channels for exchanging spatial data. We will examine two directions for transfer, first the import of background maps into R, and next the export of data for display on web-based mapping platforms.

The **RgoogleMaps** package provides tools to access Google Maps™data in image form using the Google Static Maps API, in order to permit background maps to be used in R. The HTTP request issued specifies the image required, which is then composed and downloaded for display in R graphics devices. The object uses screen coordinates internally, but may be reshaped as a `SpatialGridDataFrame`, permitting standard **sp** methods to be used for overplotting. The package has been extended to issue HTTP requests to OpenStreetMap, but as yet without geographical registration, so that we can retrieve images for the street layout of the centre of Bergen, Norway in this way, in addition to using WMS from OpenStreetMap; the results are shown in Fig. 4.4:

```
> library(RgoogleMaps)
> myMap <- GetMap(center = c(60.395, 5.322), zoom = 16,
+     destfile = "MyTile2.png", maptype = "mobile")

> BB <- do.call("rbind", myMap$BBOX)
> dBB <- rev(diff(BB))
> DIM12 <- dim(myMap$myTile)[1:2]
> cs <- dBB/DIM12
> cc <- c(BB[1, 2] + cs[1]/2, BB[1, 1] + cs[2]/2)
> GT <- GridTopology(cc, cs, DIM12)
> p4s <- CRS("+proj=longlat +datum=WGS84")
> SG_myMap <- SpatialGridDataFrame(GT, proj4string = p4s,
+     data = data.frame(r = c(t(myMap$myTile[, , 1])) *
+         255, g = c(t(myMap$myTile[, , 2])) * 255, b = c(t(myMap$myTile[,
+         , 3])) * 255))

> myMap1 <- GetMap.OSM(lonR = c(5.319, 5.328), latR = c(60.392,
+     60.398), scale = 4000, destfile = "MyTile.png")
```

Vector data from OpenStreetMap is also available for download from the recently contributed package **osmar**; the package is under active development. If we retrieve a similar area to that sourced from Google Maps™, we should be able to overlay the vector data after conversion to an **sp** class; we can also see who had made most contributions of lines to OSM at the time this snapshot was downloaded:

```
> library(osmar)
> api <- osmsource_api()
> box <- corner_bbox(5.319, 60.392, 5.328, 60.398)
> torget <- get_osm(box, source = api)

> torget1 <- as_sp(torget, "lines")

> sort(table(torget1$user), decreasing = TRUE)[1:3]

Karl Ove Hufthammer             M E Menk           Bård Aase
                 47                   39                  29
```

The package also provides methods for selection of particular kinds of objects, so that we can find and select the city terminus of the light rail route for overplotting in this way (using data downloaded as this chapter was being revised, but not necessarily valid before or after, for example, the tag `"light_rail"` appears to have been changed to `"railway"` at some stage, and an inoperative museum tramway added):

```
> bybane <- find(torget, way(tags(k == "light_rail")))
> bybane <- find_down(torget, way(bybane))
> bybane <- subset(torget, ids = bybane)
> bybane <- as_sp(bybane, "lines")
```

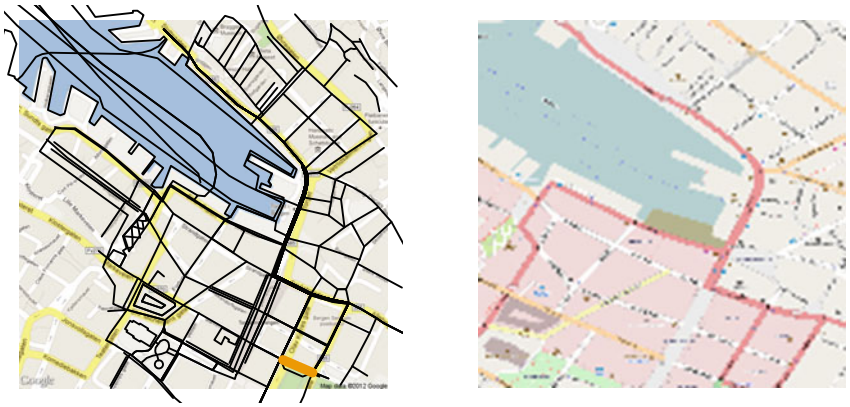

Fig. 4.4 Background maps imported with functions in **RgoogleMaps** from Google Maps™, overplotted with line data imported with functions in **osmar** with the light rail terminus shown as an *orange line*; and OpenStreetMap for the centre of Bergen, Norway (©Google™and ©OpenStreetMap contributors, CC-BY-SA)

Data may be exported for display with Google Earth™and other systems in the Keyhole Markup Language (KML) format. Vector data can be exported directly in a number of ways, given that it is in geographical coordinates and in the WGS84 datum. Point data may use the KML driver in OGR through `writeOGR`, with the values of attributes shown in bubbles when displayed points are clicked:

Fig. 4.5 Forest fires in Europe from JRC database shown in Google Earth™

```
> writeOGR(Fires[, c("gml_id", "FireDate", "Area_HA")],
+     dsn = "fires.kml", layer = "fires", driver = "KML")
```

Figure 4.5 shows the KML file displayed in Google Earth™, and the reader may try out the attribute query mechanism, as well as zooming in to regions of interest in a running Google Earth application. Lines and Polygons may also be exported with the KML driver and `writeOGR`, but there is only limited support for styles and attribute data. There are also three functions in **maptools**, `kmlPoints`, `kmlLine` and `kmlPolygon`, that permit more control over style but at the cost of more work setting arguments. The **plotKML** package is now available on CRAN, providing unified methods for setting styles and other display qualities, and handling space-time data.

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)
```

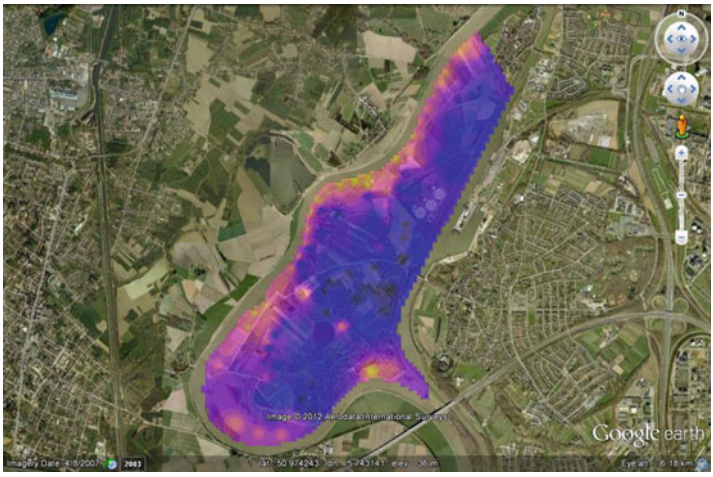

Fig. 4.6 Interpolated log zinc ppm for the Meuse Bank data set shown in Google Earth™

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 `over` 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 <- over(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 (the need for the `::` notation is explained on Sect. 8.3.1):

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

Since we have used `GE_SpatialGrid` to set up the size of an `Rpng` 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. 4.6:

```
> 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")
```

4.5 Geographical Resources Analysis Support System (GRASS)

GRASS[20] 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; for further reviews, see Neteler et al. (2008, 2012) and Jolma et al. (2012).

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 6.4.2 released in early 2012, 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 **rgdal** for moving data, 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; a custom binary interface using `r.in.bin` and `r.out.bin` is also available, and may be faster than using a GDAL file format. 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

[20] http://grass.osgeo.org/.

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.

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**, and drivers may be chosen by the user. 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. It does not yet seem to be possible to use mechanisms in **rgdal** to interface colour tables or category names for rasters. Native Windows GRASS is now firmly established for GRASS 6.4.*, and the interface functions well on that platform; and **spgrass6** binaries for Windows and Mac OSX are on CRAN.

Each GRASS program takes a `--interface-description` flag, which when run returns an XML description of its flags and parameters. These descriptions are used by the GRASS GUI to populate its menus, and are also used in **spgrass6** to check that GRASS programs are used correctly. This also means that the `parseGRASS` function can set up an object in a searchable list on the R side of the interface, to avoid re-parsing interface descriptions that have already been encountered in a session. The middle function is `doGRASS`, which takes the flags and parameters chosen, checks their validity, and constructs a command string. Finally, `execGRASS` uses the `system` function to execute the GRASS program with the chosen flag and parameter values.

The package may be used in two ways, either in an R session started from within a GRASS session from the command line, or with the `initGRASS` function. The function may be used with an existing GRASS location and mapset, or with a one-time throw-away location, and takes the GRASS installation directory as its first argument; an example is given on p. 136. It then starts a GRASS session within the R session, and is convenient for scripting GRASS in R, rather than Python, which is be the GRASS scripting language in development version GRASS 7.

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

```
> library(spgrass6)

> execGRASS("g.region", flags = "p")

projection: 1 (UTM)
zone:       13
datum:      nad27
ellipsoid:  clark66
north:      4928000
south:      4914020
west:       590010
east:       609000
nsres:      30
ewres:      30
rows:       466
cols:       633
cells:      294978
```

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. 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
[1,]  590010  609000
[2,] 4914020 4928000
Is projected: TRUE
proj4string :
[+proj=utm +zone=13 +a=6378206.4 +rf=294.9786982 +no_defs
+nadgrids=/home/rsb/topics/grass/g642/grass-6.4.2/etc/nad/conus
+to_meter=1.0]
Grid attributes:
  cellcentre.offset cellsize cells.dim
1            590025       30       633
2           4914035       30       466
Data attributes:
 elevation.dem       geology
 Min.   :1066   sandstone:75062
 1st Qu.:1200   limestone:61278
 Median :1316   shale    :46207
 Mean   :1354   sand     :36706
 3rd Qu.:1488   igneous  :36394
 Max.   :1840   (Other)  :37561
 NA's   :2661   NA's     : 1770
```

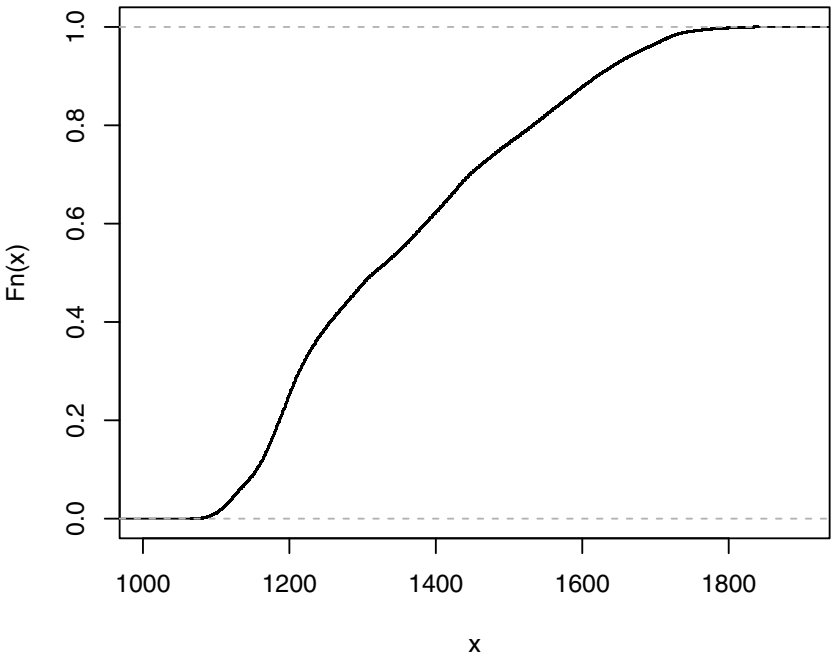

Fig. 4.7 Empirical cumulative distribution function of elevation for the Spearfish location

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
      11556         142       36394       75062       61278
      shale sandy shale    claysand        sand
      46207       11340       14523       36706

> execGRASS("r.stats", input = "geology", flags = c("quiet",
+     "c", "l"))

1 metamorphic 11556
2 transition 142
3 igneous 36394
4 sandstone 75062
5 limestone 61278
6 shale 46207
7 sandy shale 11340
8 claysand 14523
9 sand 36706
* no data 1770
```

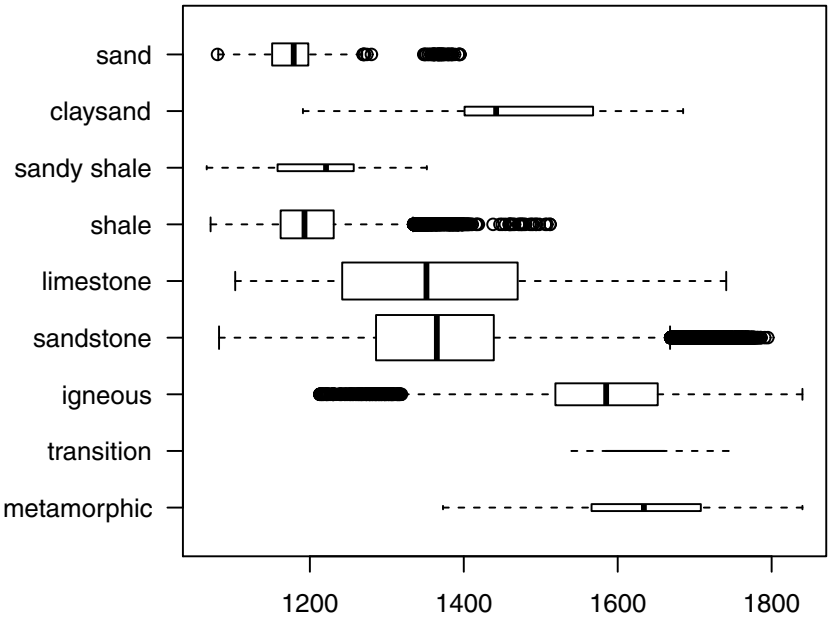

Fig. 4.8 Boxplots of elevation by geology category, Spearfish location

Figure 4.7 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. 4.8 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")

     nodes     points      lines boundaries  centroids      areas
       139          0        104         12          4          4
   islands      faces    kernels primitives      map3d
         4          0          0        120          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 +datum=NAD27 +units=m +no_defs
+ellps=clrk66
+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.

The interface between GRASS 6 and R has been used in research in a number of fields, for example by Carrera-Hernández and Gaskin (2008) in implementing the Basin of Mexico hydrogeological database, and by Grohmann and Steiner (2008) in SRTM resampling using short distance kriging. The work by Haywood and Stone (2011) is interesting in that it uses the interface to apply the Weka machine learning software suite, itself interfaced to R through the **RWeka** package, to GIS data in GRASS; R then becomes a convenient bridge between applications, with the GRASS–R interface opening up other possibilities beyond R.

4.5.1 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.

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

> buildings <- readVECT6("vsnow4")
> proj4string(sohoSG) <- CRS(proj4string(buildings))
```

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 <- over(deaths, sohoSG)
> library(maptools)
> deaths <- spCbind(deaths, o)
> deaths$b_nearer <- deaths$snowcost_broad < deaths$snowcost_not_broad

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

deaths$b_nearer: FALSE
[1] 221
------------------------------------------------------
deaths$b_nearer: TRUE
[1] 357


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

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. 4.9, 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. 4.10.

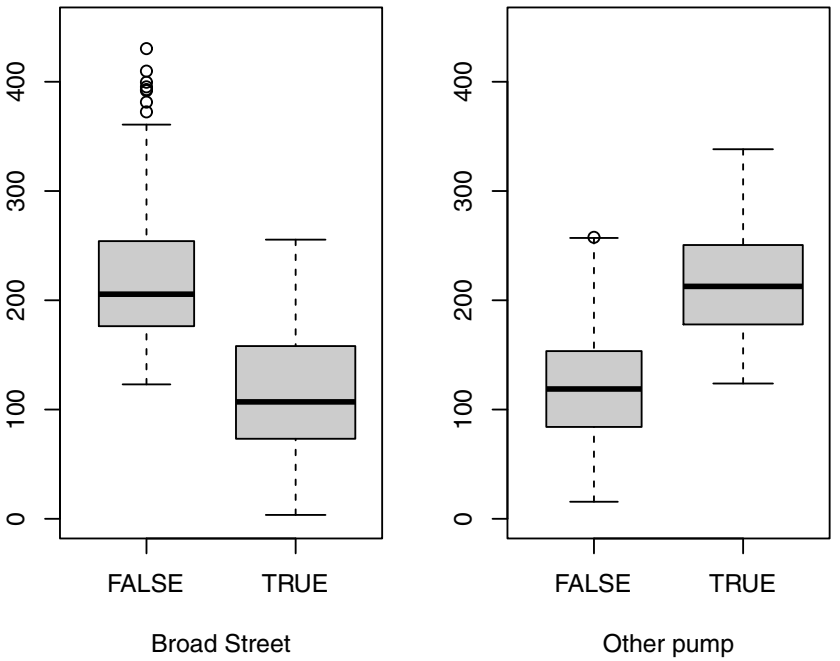

Fig. 4.9 Comparison of walking distances from homes of fatalities to the Broad Street pump or another pump by whether the Broad Street pump was closer or not

The colour 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.

It is possible to reproduce some of the analysis on the R side using **rgeos** and **gdistance**, by importing the digitised building outlines from GRASS into R, using `gBuffer` to buffer them in from the street so that the house points with mortalities fall in the street, then an `over` method to define a street raster.

```
> library(rgeos)
> vsnow4buf <- gBuffer(buildings, width = -4)
> GRD <- gmeta2grd()
> SG <- SpatialGrid(GRD, proj4string = CRS(proj4string(vsnow4buf)))
> o <- over(SG, vsnow4buf)
> crs <- CRS(proj4string(vsnow4buf)
> SGDF <- SpatialGridDataFrame(GRD, proj4string = crs),
+     data = data.frame(o = o))
> SGDF$o[is.na(SGDF$o)] <- 2.5
> SGDF$o[SGDF$o == 1] <- NA
```

Finally, `rSPDistance` may be used to find cost distances from houses with mortalities to pumps:

Fig. 4.10 The 1854 London cholera outbreak near Golden Square

```
> library(gdistance)
> r <- as(SGDF, "RasterLayer")
> tr <- transition(r, mean, 8)
> d_b_pump <- rSPDistance(tr, deaths, b_pump, theta = 1e-12)
> d_nb_pump <- rSPDistance(tr, deaths, nb_pump, theta = 1e-12)

> deaths$g_snowcost_broad <- d_b_pump[, 1]
> deaths$g_snowcost_not_broad <- apply(d_nb_pump, 1, min)
> deaths$g_b_nearer <- deaths$g_snowcost_broad < deaths
  $g_snowcost_not_broad
> by(deaths$Num_Cases, deaths$g_b_nearer, sum)

deaths$g_b_nearer: FALSE
[1] 272
------------------------------------------------------
deaths$g_b_nearer: TRUE
[1] 306
```

Neither the values nor the distances are the same as those yielded by the GRASS module `r.cost`, but the conclusion is the same despite the differences in implementations of cost distances.

4.6 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 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 and from **spatstat** `ppp`, `owin`, `im` and `psp` 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).

The four successor packages to the **adehabitat** package: **adehabitatHR**, **adehabitatHS**, **adehabitatLT**, and **adehabitatMA**, all depend on **sp** and use **sp** classes directly. The original package was documented in Calenge (2006), and includes many tools for the analysis of space and habitat use by animals.

4.6.1 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,[21] 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

[21] http://www.stata.com/search.cgi?query=tmap.

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 (Laurent et al., 2012). The R graphics facilities are perhaps better suited to non-interactive use, however, especially as it is easy to write data out to Mondrian (Theus, 2002; Theus and Urbanek, 2009).[22] 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.

4.6.2 TerraLib and aRT

The **aRT** package[23] provides an advanced modular interface to TerraLib.[24] 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.

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**.[25] All of

[22] http://rosuda.org/Mondrian/.

[23] http://leg.ufpr.br/aRT/.

[24] http://www.terralib.org/.

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

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.

4.6.3 Other GIS Systems

An interface package – **RSAGA** – has been provided for SAGA GIS[26]; like the GRASS 6 interface, it uses `system` to pass commands to external software. The R interface with SAGA has been used by Brenning (2009) for integrating terrain analysis and multispectral remote sensing in automatic rock glacier detection, using modern regression techniques – the availability of many varied techniques in R permitted them to be evaluated rapidly. Goetz et al. (2011) follow this up in integrating physical and empirical landslide models. In a paper on geostatistical modelling of topography, Hengl et al. (2008) use the interface between R and SAGA to benefit from the strengths of both software components. Hengl et al. (2010) address the associated problem of stream network uncertainty, when the stream networks are derived from interpolated elevation data, again using the interface between SAGA and R; R is also used extensively for scripting SAGA.

In connection with a comprehensive book on spatial statistical data analysis based on ArcGIS™, Krivoruchko (2011, pp. 767–801) devotes Appendix 2 to the use of R with ArcGIS™, using both file transfer, and (D)COM and Python interfaces. In addition, Chap. 16, pp. 676–715 covers a range of useful examples of R/ArcGIS use for spatial data analysis, with many code examples. A link to updated code may be found on the website of this book.

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 `RStatConnector` (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.

[26] http://www.saga-gis.org

The marine geospatial ecology tools project[27] follows up the work begun in the concluded ArcRstats project, providing for execution in many environments (Roberts et al., 2010). 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.

4.7 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. Thanks to sustained contributions by Brian Ripley and Uwe Ligges, CRAN publishes a self-contained Windows binary **rgdal** package for 32-bit and 64-bit architectures, with a substantial range of drivers available. A similar range of drivers is available for Intel architectures for Mac OSX in binary **rgdal** packages published on CRAN thanks to continuing help from Simon Urbanek.

For Linux/Unix, 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[28] a useful place to start should additional drivers be required. More information is available by searching the archives of the R-sig-geo mailing list, but frequent changes in **rgdal** may render older postings less relevant.

Windows users needing other drivers, and for whom conversion using programs in the OSGeo4W[29] binary for Windows is not useful, may choose to install **rgdal** from source, compiling the **rgdal** DLL with VC++ and linking against the OSGeo4W DLLs – see the `inst/README.windows` file in the source package for details.

[27] http://code.env.duke.edu/projects/mget.

[28] http://www.kyngchaos.com/software/frameworks.

[29] http://trac.osgeo.org/osgeo4w/.