Using spatial data

Roger Bivand[1]

9 July 2013

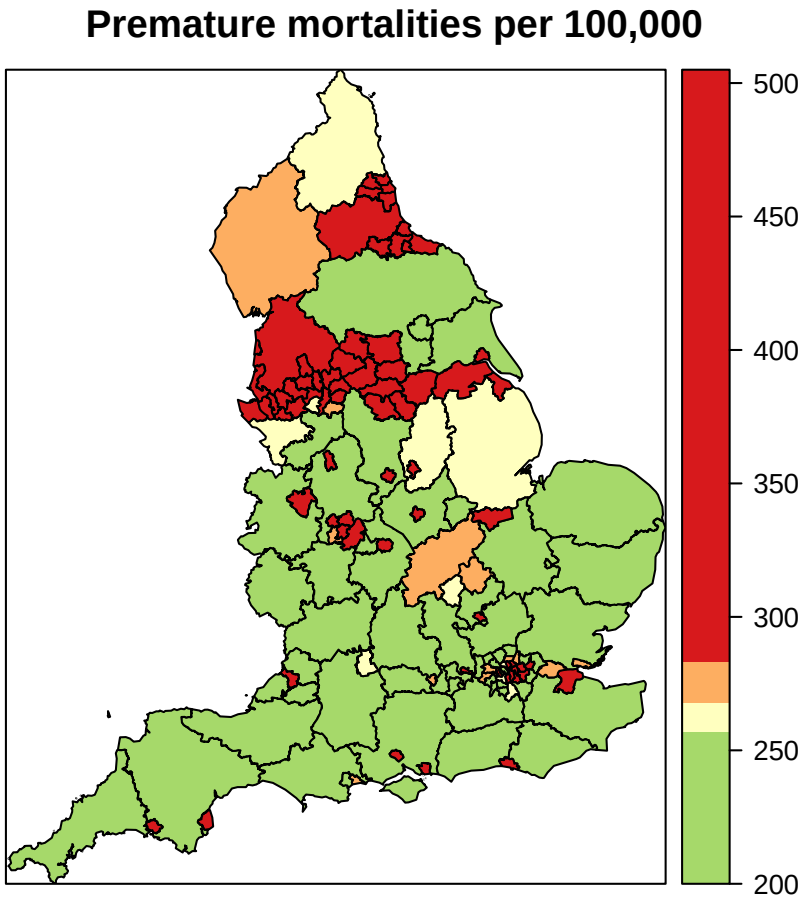
[1]Department of Economics, Norwegian School of Economics, Helleveien 30, N-5045 Bergen, Norway; Roger.Bivand@nhh.no

Overview

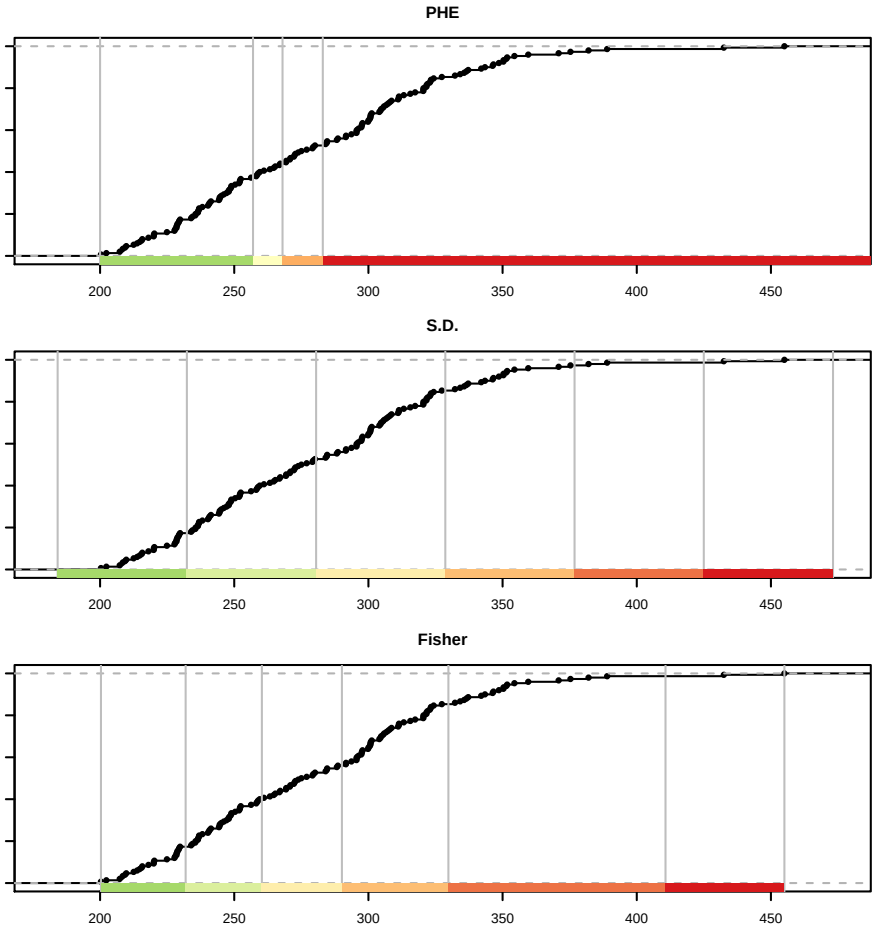
1. Why spatial data in R?
2. Representing spatial data
3. Handling spatial data
   - Interfaces — GRASS
   - Topology
   - Raster data
   - Exporting data
   - Overlay
4. Worked examples
   - Disease mapping
   - Spatial autocorrelation
   - Regression
   - Bayesian spatial regression with INLA

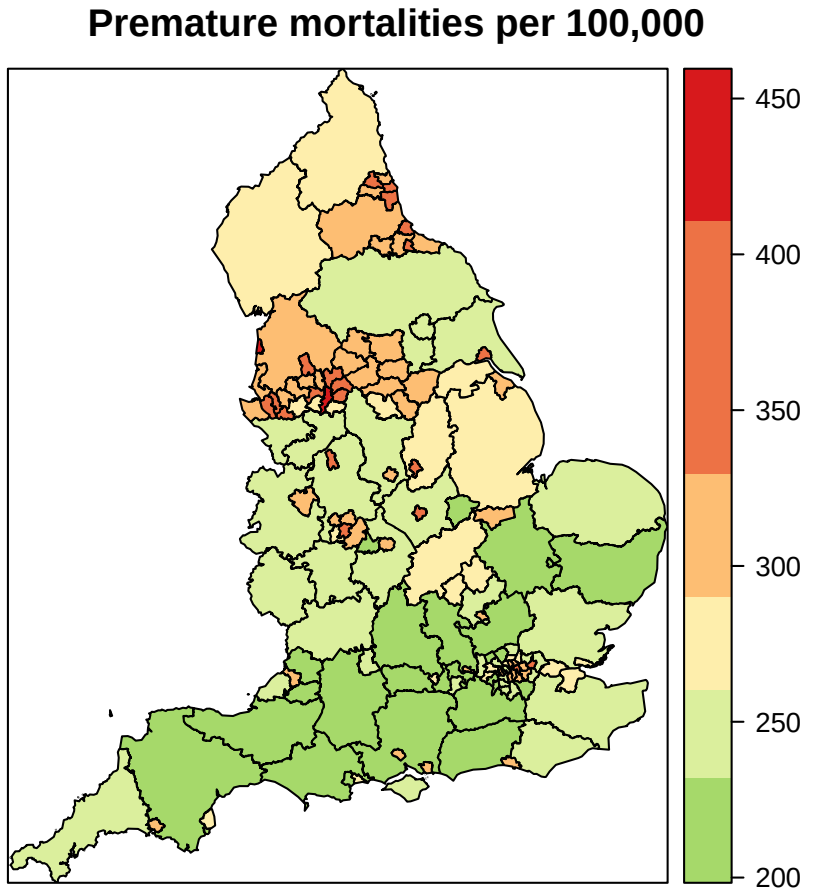
Visualizing longer lives I

A recent news item `http://www.bbc.co.uk/news/health-22844227` describes work on overall premature mortality (<75) in England. It links to Public Health England `http://longerlives.phe.org.uk/`, providing access to data. Using open boundary data (Contains Ordnance Survey data ©Crown copyright and database right 2013), the map can be reproduced:

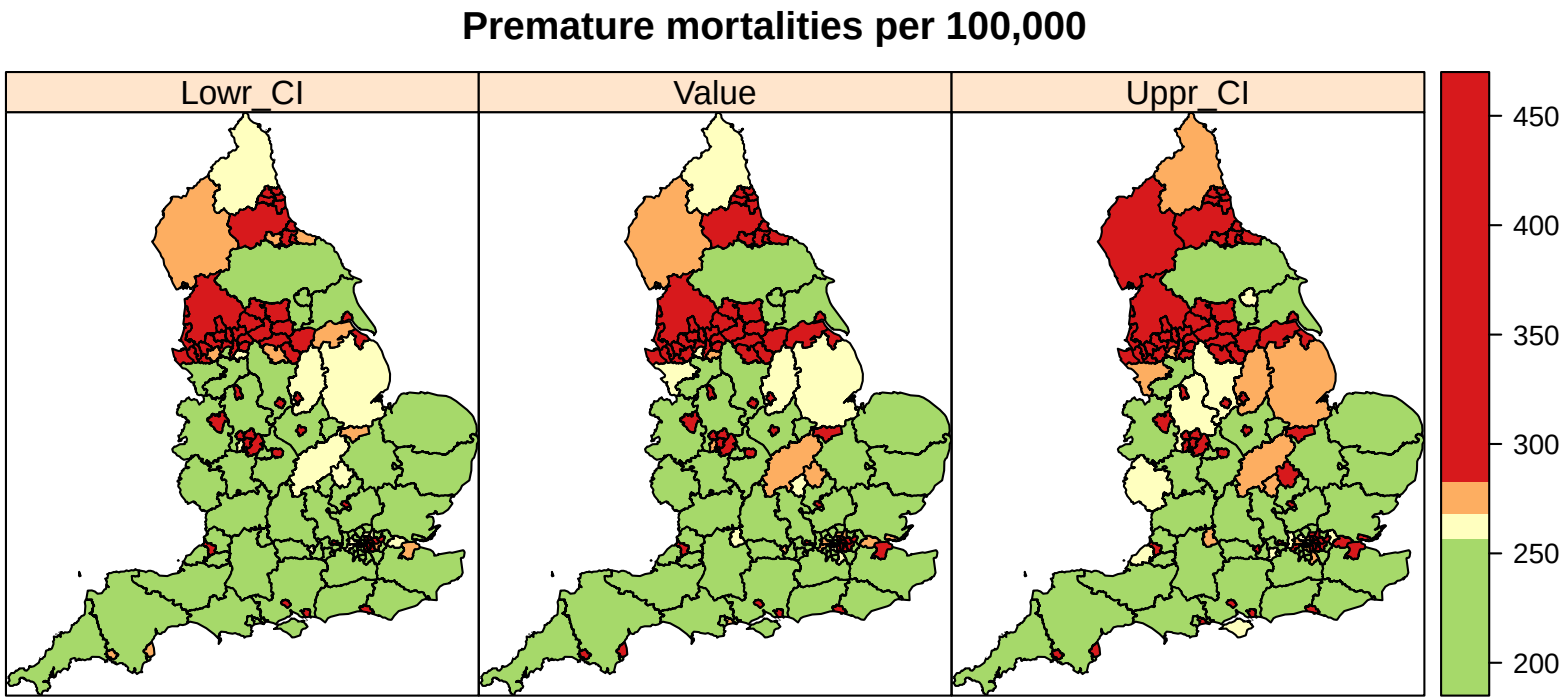


Visualizing longer lives II



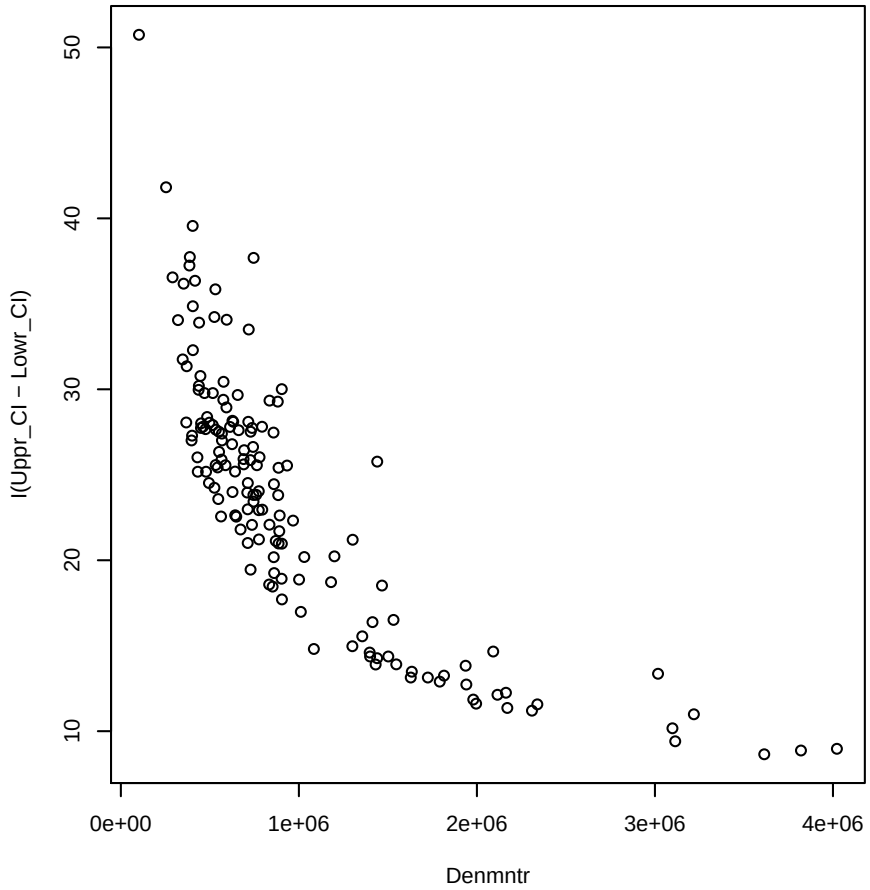


Visualizing longer lives III

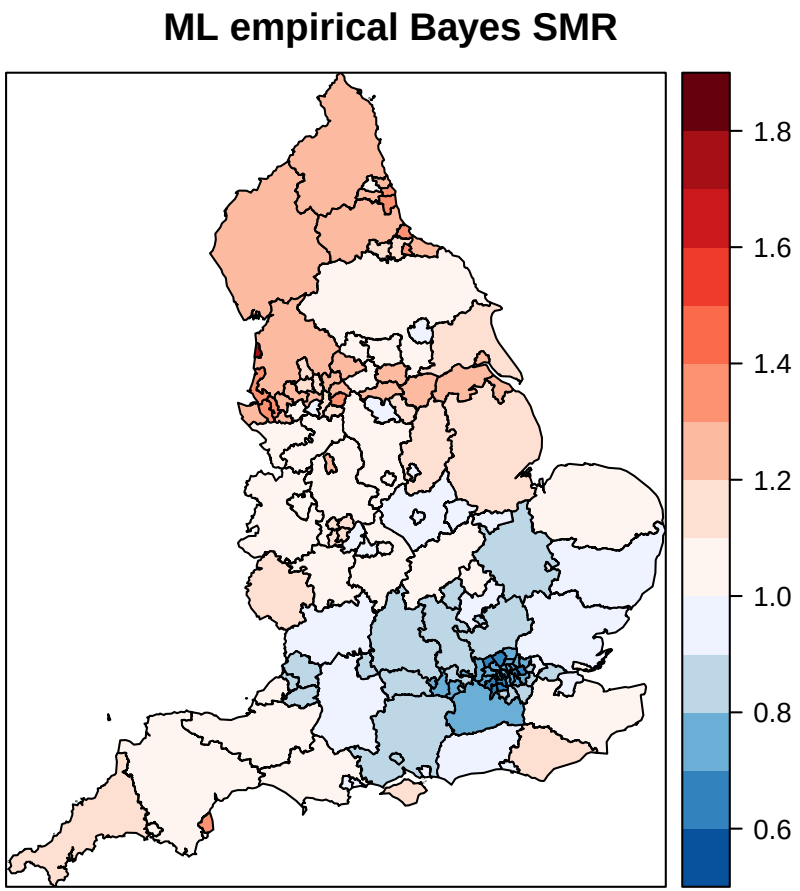


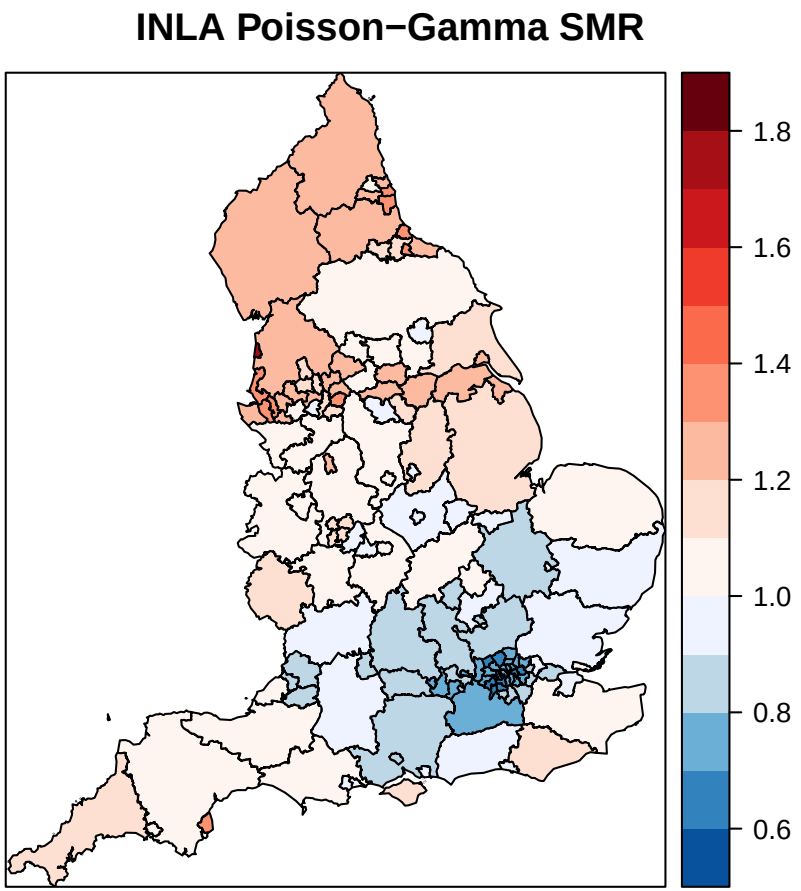
Visualizing longer lives IV

The website does not explain what statistical methods have been used to get the CI values, other than saying that "directly standardised rates" were used. We do not know what the denominator given is — maybe it is age-adjusted? If that is the case, we can use INLA to fit a Poisson-Gamma model, to see if that is possible. First, let us see if the CI interval is related to the denominator:

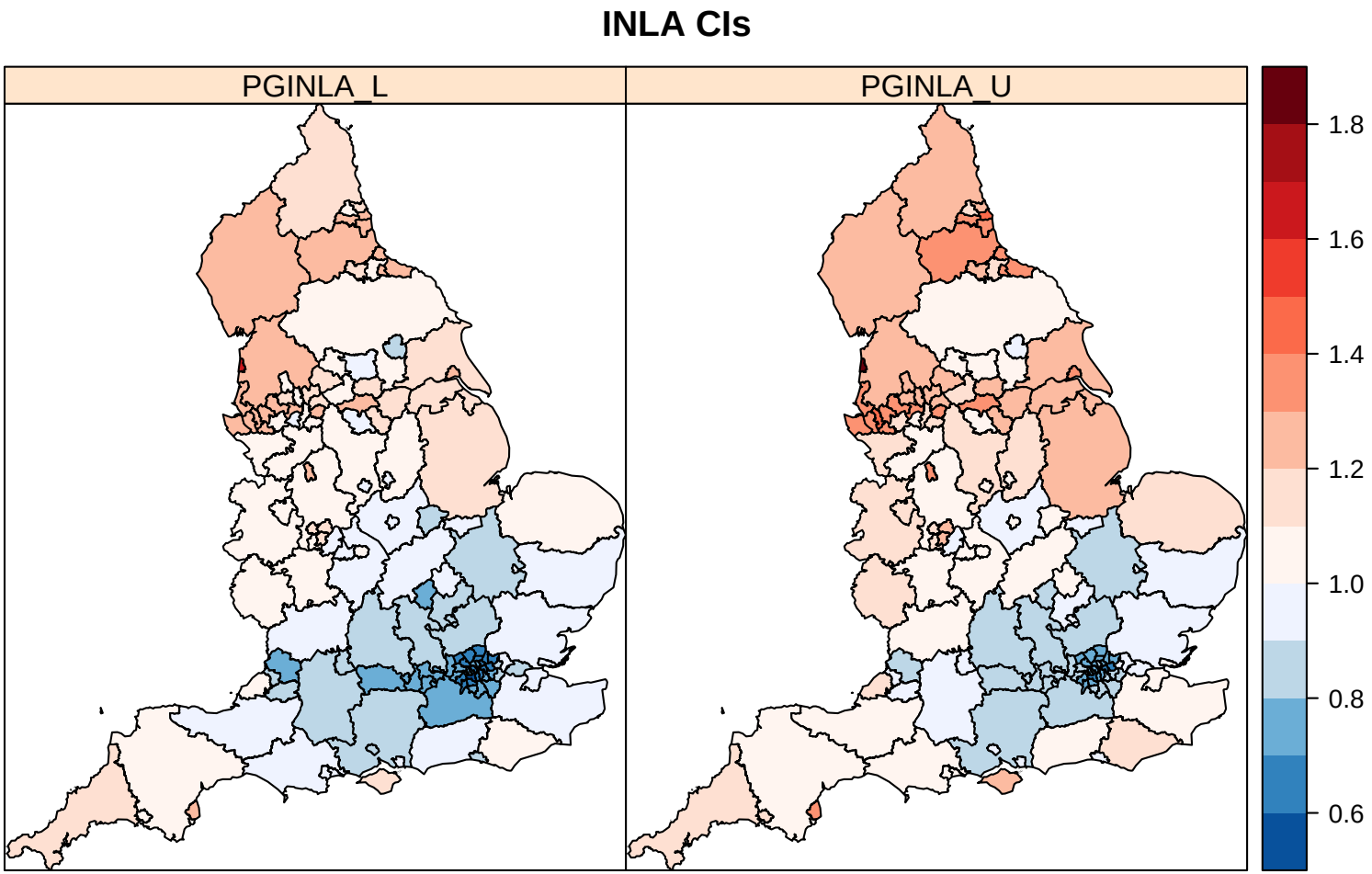


Visualizing longer lives V





Visualizing longer lives VI



Olinda Hansen's Disease data set

- The data are discussed in Lara, T. et al. (2001) Leprosy surveillance in Olinda, Brazil, using spatial analysis techniques. *Cadernos de Saúde Pública*, 17(5): 1153–1162.
- They use 243 1990 census districts, and include the 1991-1996 counts of new cases (CASES), 1993 year-end population figures (POP) and a deprivation index (DEPRIV)
- The data have been made available in cooperation with Marilia Sa Carvalho, and their use here is partly based on teaching notes that we wrote together some years ago
- The data now take the form of a shapefile, but were initially assembled from a Mapinfo file of census district borders and text files of variable values

Olinda Hansen's Disease data set

So that we've got something to go on, let's read in the data set, using an interface function in the **rgdal** package:

```
> library(rgdal)
```

The function as we will see later takes at least two arguments, a data source name, here the current directory, and a layer name, here the name of the shapefile without extension. Once we have checked the names in the imported object, we can use the `spplot` method in **sp** to make a map:

```
> olinda <- readOGR(".", "setor1")

OGR data source with driver: ESRI Shapefile
Source: ".", layer: "setor1"
with 243 features and 10 fields
Feature type: wkbPolygon with 2 dimensions

> names(olinda)

 [1] "AREA"      "PERIMETER"
 [3] "SETOR_"    "SETOR_ID"
 [5] "VAR5"      "DENS_DEMO"
 [7] "SET"       "CASES"
 [9] "POP"       "DEPRIV"

> spplot(olinda, "DEPRIV", col.regions = grey.colors(20,
+     0.9, 0.3))
```

Map of deprivation variable

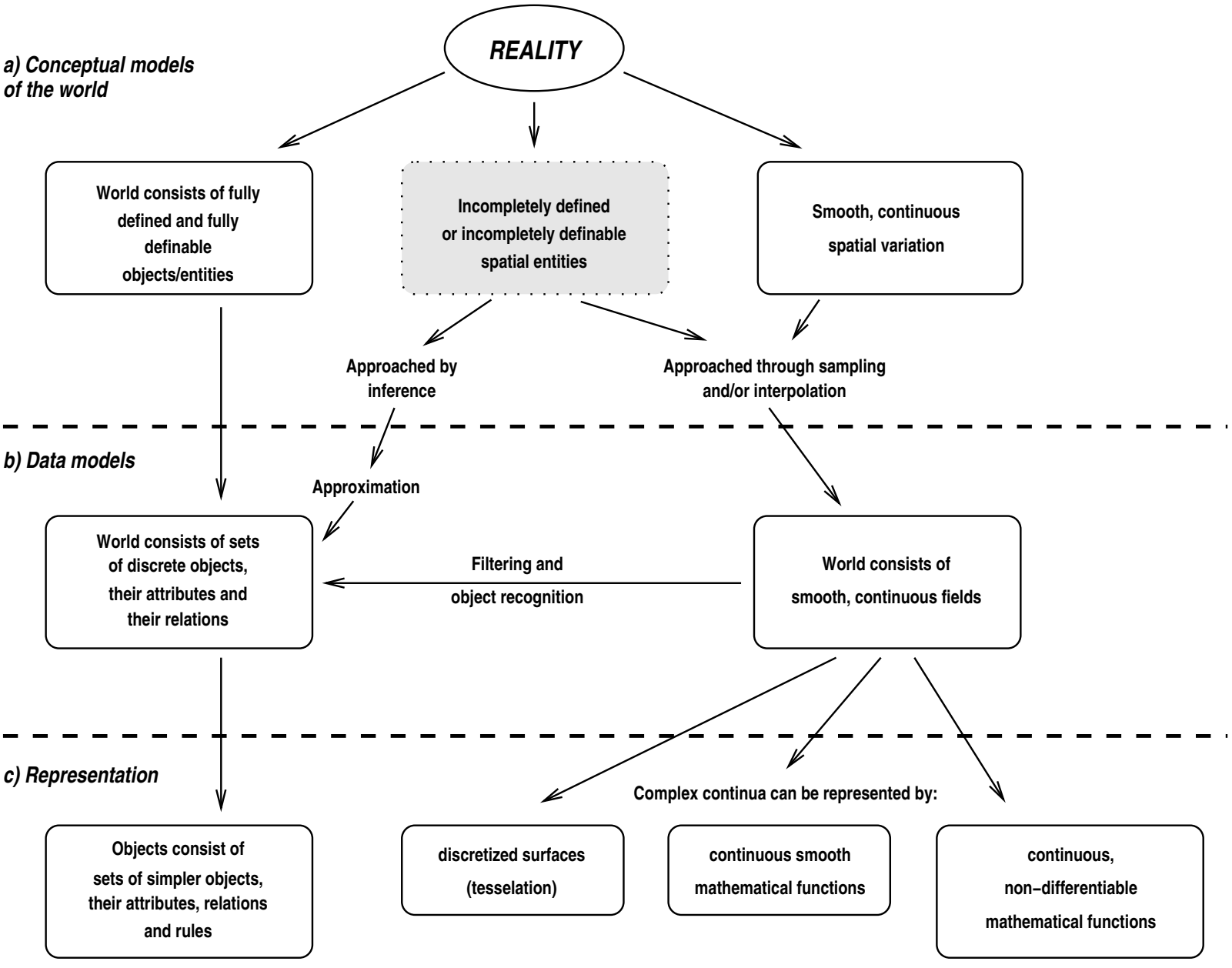
Object framework

- To begin with, all contributed packages for handling spatial data in R had different representations of the data.
- This made it difficult to exchange data both within R between packages, and between R and external file formats and applications.
- The result has been an attempt to develop shared classes to represent spatial data in R, allowing some shared methods and many-to-one, one-to-many conversions.
- We chose to use new-style classes to represent spatial data, and are confident that this choice was justified.

GIS data models

- Spatial reference systems are used to register different data in the same framework
- Geographical metadata includes projection, ellipsoid, and datum, as temporal metadata includes series origin and time zone
- Spatial data regarding position are subject to measurement error and to numerical artifacts as a result of transformation
- The two major data models: raster and vector, handle resolution differently; neither handle "crispness" well

Representation in GIS



"Combining incompatible spatial data"

- Ten years ago, Gotway and Young (2002) pointed up serious statistical questions involved in spatial data analysis
- The underlying issue is the change of support problem, where the measurement may not capture the phenomenon under analysis well
- This is endemic for example in the marine domain, almost all "measurements" involve error processes, be they spatially structured or otherwise
- For prediction (and particularly for putting confidence intervals on predictions), these must be carried through

The change of support problem

- They start by defining *support* as: "the size or volume associated with each data value", but it "also includes the geometrical size, shape, and spatial orientation of regions associated with the measurements"
- "Changing the support of a variable ... creates a new variable (which) is related to the original, but has different statistical and spatial properties"
- "... how the spatial variation in one variable associated with a given support relates to that of the other variable with a different support is the change of support problem"
- Typically, GIS data models require some change of support between raw input and data that can be used for analysis, making entitation a research decision of some importance

Spatial objects

- The foundation object is the `Spatial` class, with just two slots (new-style class objects have pre-defined components called slots)
- The first is a bounding box, and is mostly used for setting up plots
- The second is a `CRS` class object defining the coordinate reference system, and may be set to `CRS(as.character(NA))`, its default value.
- Operations on `Spatial*` objects should update or copy these values to the new `Spatial*` objects being created

Coordinate reference systems

- 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 prime meridian and a datum, and projected CRS expressed in a measure of length, and a chosen position on the earth, as well as the underlying ellipse, prime meridian and datum.
- Most countries have multiple CRS, and where they meet there is usually a big mess — this led to the collection by the European Petroleum Survey Group (EPSG, now Oil & Gas Producers (OGP) Surveying & Positioning Committee) of a geodetic parameter dataset

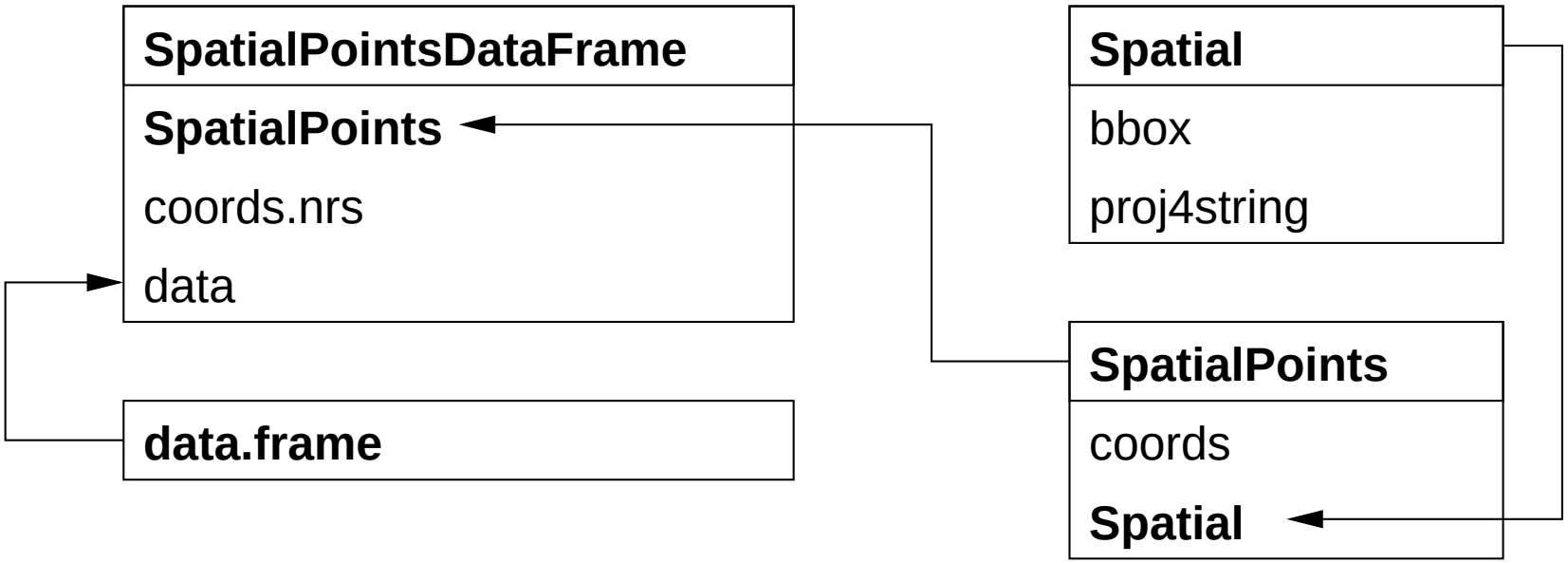
Coordinate reference systems

- The EPSG list among other sources is used in the workhorse PROJ.4 library, which as implemented by Frank Warmerdam handles transformation of spatial positions between different CRS
- This library is interfaced with R in the **rgdal** package, and the `CRS` class is defined partly in **sp**, partly in **rgdal**
- A `CRS` object is defined as a character NA string or a valid PROJ.4 CRS definition
- The validity of the definition can only be checked if **rgdal** is loaded

Spatial points

- The most basic spatial data object is a point, which may have 2 or 3 dimensions
- A single coordinate, or a set of such coordinates, may be used to define a `SpatialPoints` object; coordinates should be of mode `double` and will be promoted if not
- The points in a `SpatialPoints` object may be associated with a row of attributes to create a `SpatialPointsDataFrame` object
- The coordinates and attributes may, but do not have to be keyed to each other using ID values

Spatial points classes and their slots



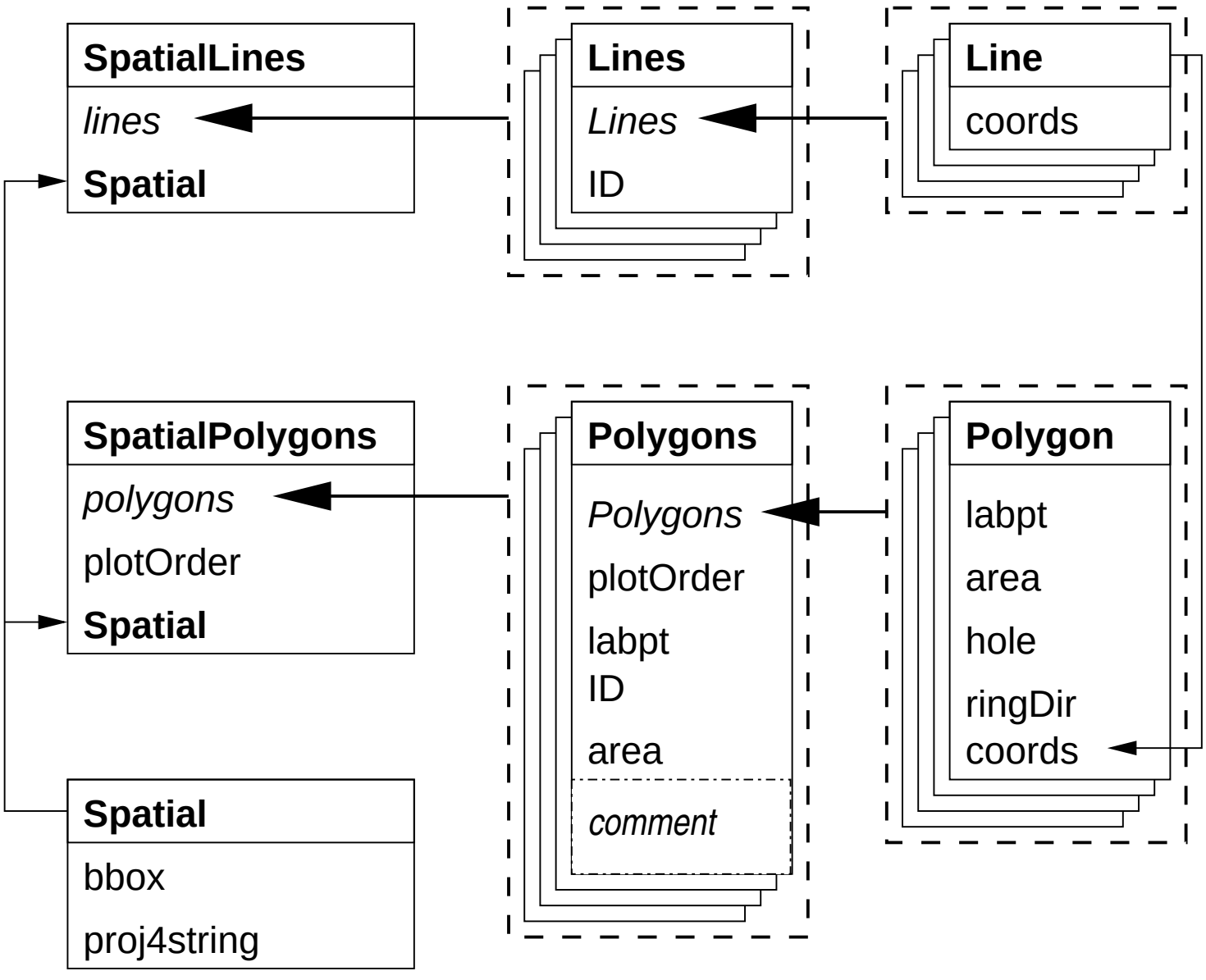
Spatial*DataFrames

`Spatial*DataFrames` are constructed as Janus-like objects, looking to mapping and GIS people as "their" kind of object, as a collection of geometric freatures, with data associated with each feature. But to data analysts, the object "is" a data frame, because it behaves like one.

Spatial lines and polygons

- A `Line` object is just a spaghetti collection of 2D coordinates; a `Polygon` object is a `Line` object with equal first and last coordinates
- A `Lines` object is a list of `Line` objects, such as all the contours at a single elevation; the same relationship holds between a `Polygons` object and a list of `Polygon` objects, such as islands belonging to the same county
- A `comment` is added to each `Polygons` object to indicate which interior ring belongs to which exterior ring (SFS)
- `SpatialLines` and `SpatialPolygons` objects are made using lists of `Lines` or `Polygons` objects respectively
- `SpatialLinesDataFrame` and `SpatialPolygonsDataFrame` objects are defined using `SpatialLines` and `SpatialPolygons` objects and standard data frames, and the `ID` fields are here required to match the data frame row names

Spatial Polygons classes and slots



Back to Olinda

So we can treat `olinda` as a data frame, for example adding an expected cases variable. The object also works within model fitting functions, like `glm`.

```
> olinda$Expected <- olinda$POP * sum(olinda$CASES, na.rm = TRUE)/sum(olinda$POP,
+     na.rm = TRUE)
> str(olinda, max.level = 2)

Formal class 'SpatialPolygonsDataFrame' [package "sp"] with 5 slots
  ..@ data       :'data.frame': 243 obs. of  11 variables:
  ..@ polygons   :List of 243
  .. .. [list output truncated]
  ..@ plotOrder  : int [1:243] 243 56 39 231 224 60 58 232 145 66 ...
  ..@ bbox       : num [1:2, 1:2] 288955 9111044 298446 9120528
  .. ..- attr(*, "dimnames")=List of 2
  ..@ proj4string:Formal class 'CRS' [package "sp"] with 1 slots

> str(as(olinda, "data.frame"))

'data.frame': 243 obs. of  11 variables:
 $ AREA     : num  79139 151153 265037 137696 121873 ...
 $ PERIMETER: num  1270 2189 2818 2098 3353 ...
 $ SETOR_   : num  2 3 4 5 6 7 8 9 10 11 ...
 $ SETOR_ID : num  1 2 3 4 5 6 9 10 11 12 ...
 $ VAR5     : int  242 224 223 229 228 222 218 225 227 221 ...
 $ DENS_DEMO: num  4380 10563 6642 13174 13812 ...
 $ SET      : num  242 224 223 229 228 222 218 225 227 221 ...
 $ CASES    : num  1 6 5 1 1 4 6 2 1 6 ...
 $ POP      : num  337 1550 1711 1767 1638 ...
 $ DEPRIV   : num  0.412 0.168 0.192 0.472 0.302 0.097 0.141 0.199 0.228 0.223 ...
 $ Expected : num  1.4 6.43 7.1 7.33 6.8 ...
```

Back to Olinda

The object also works within model fitting functions, like `glm`; note the number of rows.

```
> str(model.frame(CASES ~ DEPRIV + offset(log(POP)), olinda), give.attr = FALSE)

'data.frame': 241 obs. of  3 variables:
 $ CASES           : num  1 6 5 1 1 4 6 2 1 6 ...
 $ DEPRIV          : num  0.412 0.168 0.192 0.472 0.302 0.097 0.141 0.199 0.228 0.223 ...
 $ offset(log(POP)): num  5.82 7.35 7.44 7.48 7.4 ...
```

Creating objects within R

- **maptools** includes `ContourLines2SLDF()` to convert contour lines to `SpatialLinesDataFrame` objects
- **maptools** also allows lines or polygons from **maps** to be used as **sp** objects
- **maptools** can export **sp** objects to **PBSmapping**
- **maptools** uses **gpclib** to check polygon topology and to dissolve polygons
- **maptools** converts some **sp** objects for use in **spatstat**
- **maptools** can read GSHHS high-resolution shoreline data into `SpatialPolygon` objects

Reading vectors

- GIS vector data are points, lines, polygons, and fit the equivalent **sp** classes
- There are a number of commonly used file formats, all or most proprietary, and some newer ones which are partly open
- GIS are also handing off more and more data storage to DBMS, and some of these now support spatial data formats
- Vector formats can also be converted outside R to formats that are easier to read

The GridTopology class

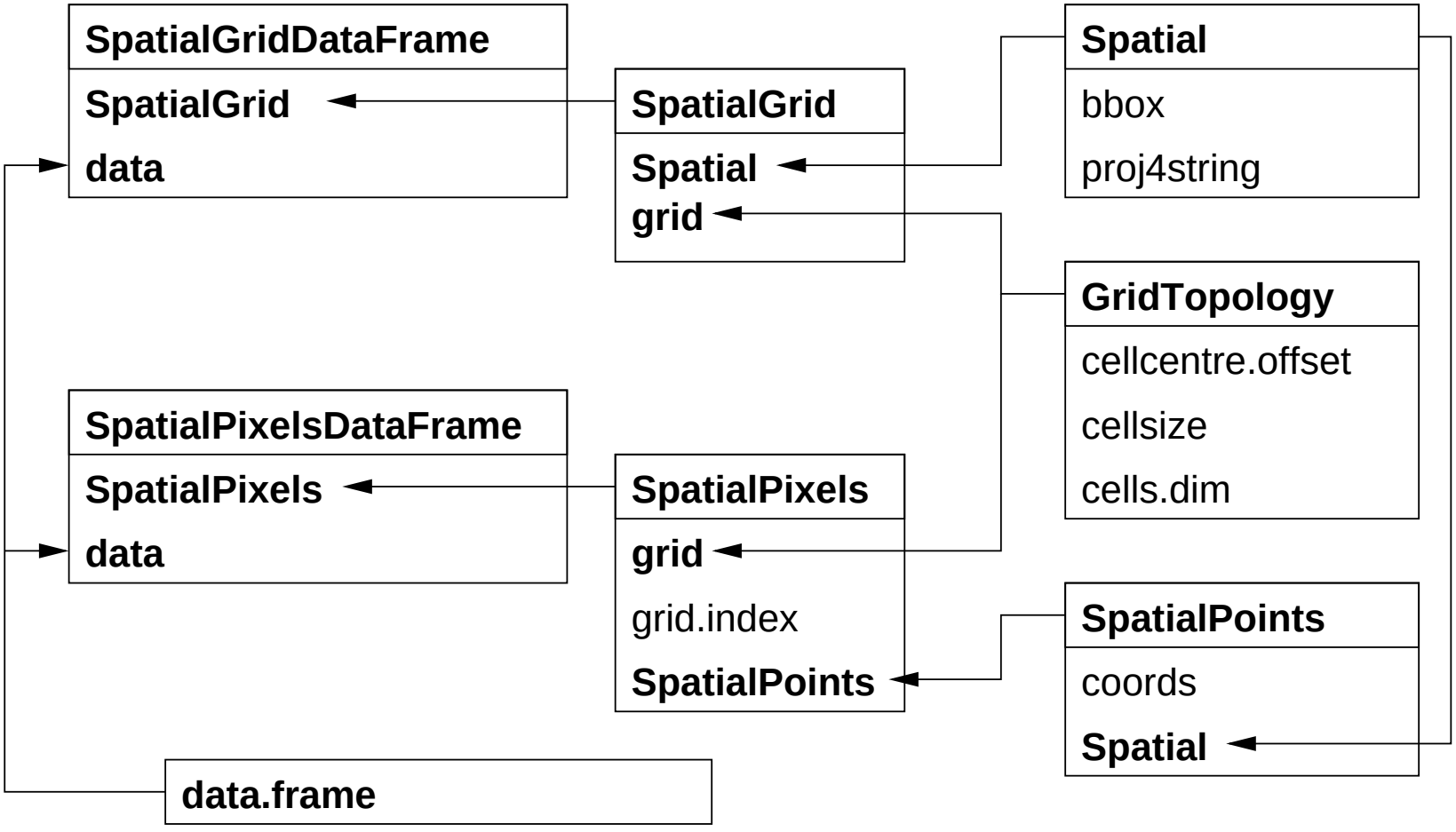
The GridTopology class is the key constituent of raster representations, giving the coordinates of the south-west raster cell, the two cell sizes in the metric of the coordinates, giving the step to successive centres, and the numbers of cells in each dimension. We'll return to examples after importing some data:

```
> library(sp)
> getClass("GridTopology")

Class "GridTopology" [package "sp"]

Slots:

Name:  cellcentre.offset
Class:           numeric

Name:           cellsize
Class:           numeric

Name:          cells.dim
Class:           integer
```

Spatial grids and pixels

- There are two representations for data on regular rectangular grids (oriented N-S, E-W): `SpatialPixels` and `SpatialGrid`
- `SpatialPixels` are like `SpatialPoints` objects, but the coordinates have to be regularly spaced; the coordinates are stored, as are grid indices
- `SpatialPixelsDataFrame` objects only store attribute data where it is present, but need to store the coordinates and grid indices of those grid cells
- `SpatialGridDataFrame` objects do not need to store coordinates, because they fill the entire defined grid, but they need to store `NA` values where attribute values are missing

Spatial grid and pixels classes and their slots

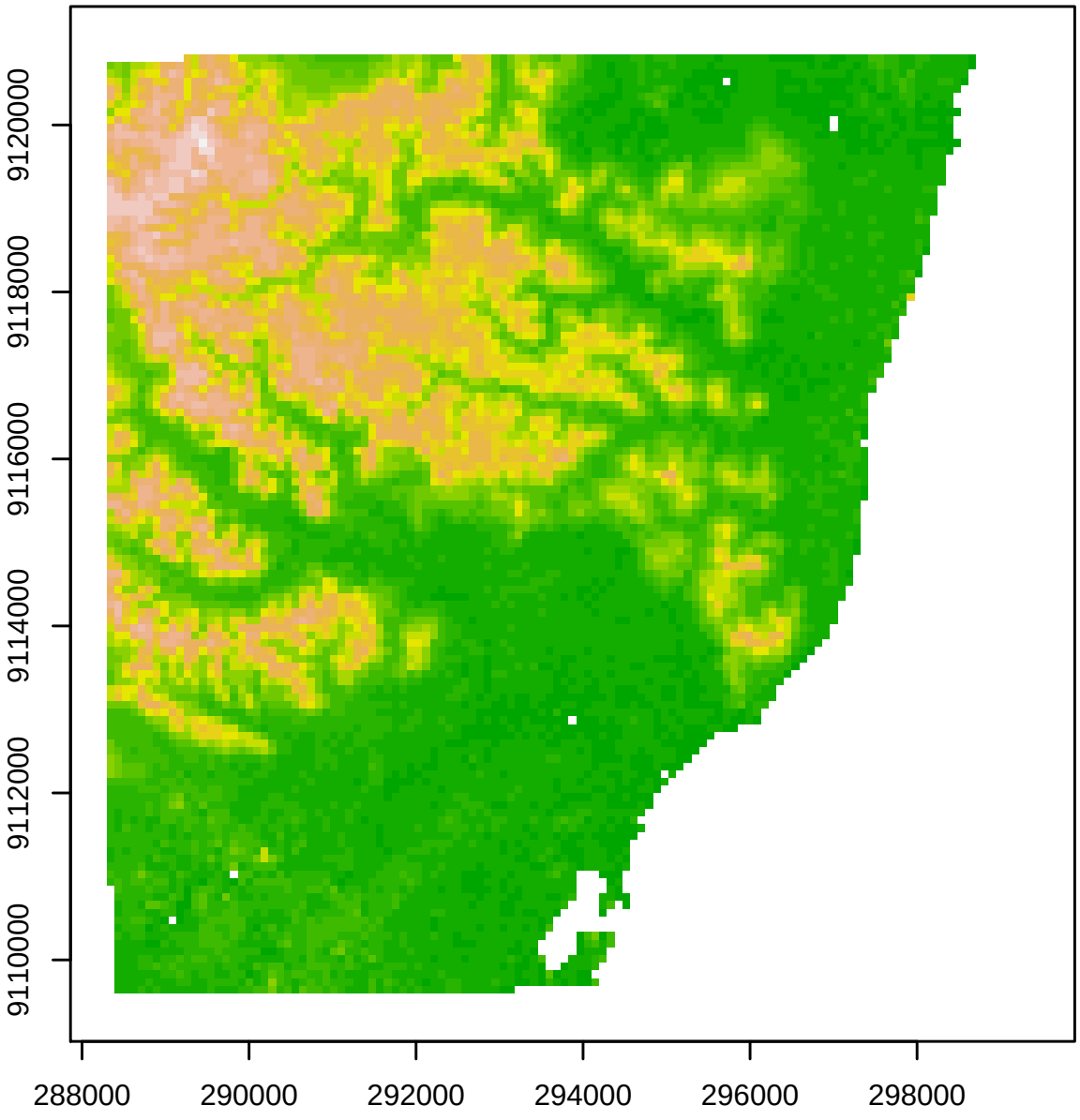


A SpatialGridDataFrame object

The space shuttle flew a radar topography mission in 2000, giving 90m resolution elevation data for most of the world. The data here have been warped to a UTM projection, but for the WGS84 datum - we'll see below how to project and if need be datum transform `Spatial` objects:

```
> DEM <- readGDAL("UTM_dem.tif")

UTM_dem.tif has GDAL driver GTiff
and has 122 rows and 121 columns

> summary(DEM)

Object of class SpatialGridDataFrame
Coordinates:
        min       max
x  288307.3  299442.7
y 9109607.8 9120835.2
Is projected: TRUE
proj4string :
[+proj=utm +zone=25 +south +ellps=WGS84 +units=m +no_defs]
Grid attributes:
  cellcentre.offset cellsize cells.dim
x          288353.4 92.02797       121
y         9109653.8 92.02797       122
Data attributes:
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  -1.00    3.00   11.00   19.45   31.00   88.00

> is.na(DEM$band1) <- DEM$band1 < 1

> image(DEM, "band1", axes = TRUE, col = terrain.colors(20))
```

Elevation at Olinda



Spatial classes provided by **sp**

This table summarises the classes provided by **sp**, and shows how they build up to the objects of most practical use, the `Spatial*DataFrame` family objects:

data type	class	attributes	extends
points	`SpatialPoints`	none	`Spatial`
points	`SpatialPointsDataFrame`	`data.frame`	`SpatialPoints`
pixels	`SpatialPixels`	none	`SpatialPoints`
pixels	`SpatialPixelsDataFrame`	`data.frame`	`SpatialPixels` `SpatialPointsDataFrame`
full grid	`SpatialGrid`	none	`SpatialPixels`
full grid	`SpatialGridDataFrame`	`data.frame`	`SpatialGrid`
line	`Line`	none	
lines	`Lines`	none	`Line` list
lines	`SpatialLines`	none	`Spatial`, `Lines` list
lines	`SpatialLinesDataFrame`	`data.frame`	`SpatialLines`
polygon	`Polygon`	none	`Line`
polygons	`Polygons`	none	`Polygon` list
polygons	`SpatialPolygons`	none	`Spatial`, `Polygons` list
polygons	`SpatialPolygonsDataFrame`	`data.frame`	`SpatialPolygons`

Methods provided by **sp**

This table summarises the methods provided by **sp**:

method	what it does
`[`	select spatial items (points, lines, polygons, or rows/cols from a grid) and/or attributes variables
`$`, `$<-`, `[[`, `[[<-`	retrieve, set or add attribute table columns
`spsample`	sample points from a set of polygons, on a set of lines or from a gridded area
`bbox`	get the bounding box
`proj4string`	get or set the projection (coordinate reference system)
`coordinates`	set or retrieve coordinates
`coerce`	convert from one class to another
`over`	combine two different spatial objects

Using `Spatial` family objects

- Very often, the user never has to manipulate `Spatial` family objects directly, as we have been doing here, because methods to create them from external data are also provided
- Because the `Spatial*DataFrame` family objects behave in most cases like data frames, most of what we are used to doing with standard data frames just works — like `[` or `$` (but no `merge`, etc., yet)
- These objects are very similar to typical representations of the same kinds of objects in geographical information systems, so they do not suit spatial data that is not geographical (like medical imaging) as such
- They provide a standard base for analysis packages on the one hand, and import and export of data on the other, as well as shared methods, like those for visualisation

Coordinate reference systems

- 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 prime meridian and a datum, and projected CRS expressed in a measure of length, and a chosen position on the earth, as well as the underlying ellipse, prime meridian and datum.
- Most countries have multiple CRS, and where they meet there is usually a big mess — this led to the collection by the European Petroleum Survey Group (EPSG, now Oil & Gas Producers (OGP) Surveying & Positioning Committee) of a geodetic parameter dataset

Coordinate reference systems

- The EPSG list among other sources is used in the workhorse PROJ.4 library, which as implemented by Frank Warmerdam handles transformation of spatial positions between different CRS
- This library is interfaced with R in the **rgdal** package, and the `CRS` class is defined partly in **sp**, partly in **rgdal**
- A `CRS` object is defined as a character NA string or a valid PROJ.4 CRS definition
- The validity of the definition can only be checked if **rgdal** is loaded

The original data set CRS

Very often, a good deal of work is needed to find the actual coordinate reference system of a data set. Typically, researchers use what is to hand, without thinking that incomplete CRS specifications limit their ability to integrate other data. Here we were told that the map was from the 1990 census, that it was most likely UTM zone 25 south, but not its datum. Investigation showed that until very recently, the most used datum in Brazil has been Corrego Alegre:

```
> proj4string(olinda)

[1] NA

> EPSG <- make_EPSG()
> EPSG[grep("Corrego Alegre", EPSG$note), 1:2]

      code                                        note
154   4225                  # Corrego Alegre 1970-72
460   5524                     # Corrego Alegre 1961
2736  5536    # Corrego Alegre 1961 / UTM zone 21S
2737  5537    # Corrego Alegre 1961 / UTM zone 22S
2738  5538    # Corrego Alegre 1961 / UTM zone 23S
2739  5539    # Corrego Alegre 1961 / UTM zone 24S
2948 22521 # Corrego Alegre 1970-72 / UTM zone 21S
2949 22522 # Corrego Alegre 1970-72 / UTM zone 22S
2950 22523 # Corrego Alegre 1970-72 / UTM zone 23S
2951 22524 # Corrego Alegre 1970-72 / UTM zone 24S
2952 22525 # Corrego Alegre 1970-72 / UTM zone 25S
```

The original data set CRS

In order to insert the parameters for the transformation to the WGS84 datum, we have to override the initial imported values:

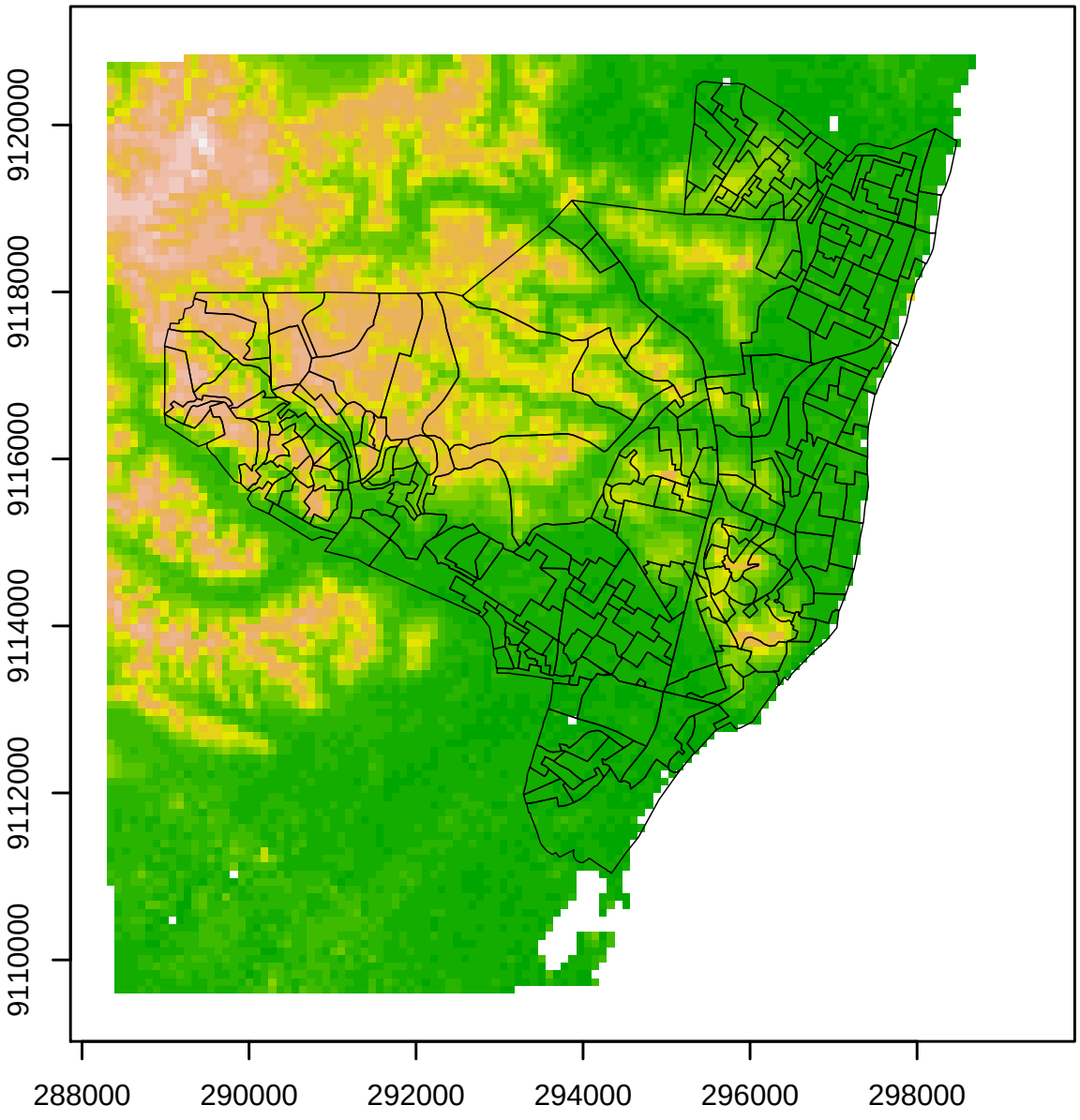
```
> set_ReplCRS_warn(FALSE)

[1] FALSE

> proj4string(olinda) <- CRS("+init=epsg:22525")
> proj4string(olinda)

[1] "+init=epsg:22525 +proj=utm +zone=25 +south +ellps=intl +towgs84=-206,172,-6,0,0,0,0 +units=m +no_defs

> proj4string(DEM)

[1] "+proj=utm +zone=25 +south +ellps=WGS84 +units=m +no_defs"

> proj4string(DEM) <- CRS("+init=epsg:31985")
> proj4string(DEM)

[1] "+init=epsg:31985 +proj=utm +zone=25 +south +ellps=GRS80 +towgs84=0,0,0,0,0,0,0 +units=m +no_defs"

> set_ReplCRS_warn(TRUE)

[1] TRUE
```

Getting olinda to WGS84

As we see, although both olinda and DEM are UTM zone 25 south, they differ in their ellipsoid and datum. Using `spTransform` methods in **rgdal** we can undertake a datum shift for `olinda`, making it possible to overplot in register:

```
> olinda1 <- spTransform(olinda, CRS(proj4string(DEM)))

> image(DEM, "band1", axes = TRUE, col = terrain.colors(20))
> plot(olinda1, add = TRUE, lwd = 0.5)
```

Getting olinda to WGS84



CRS are muddled

- If you think CRS are muddled, you are right, like time zones and daylight saving time in at least two dimensions
- But they are the key to ensuring positional interoperability, and "mashups" — data integration using spatial position as an index must be able to rely on data CRS for integration integrity
- The situation is worse than TZ/DST because there are lots of old maps around, with potentially valuable data; finding correct CRS values takes time
- On the other hand, old maps and odd choices of CRS origins can have their charm ...

GIS interfaces

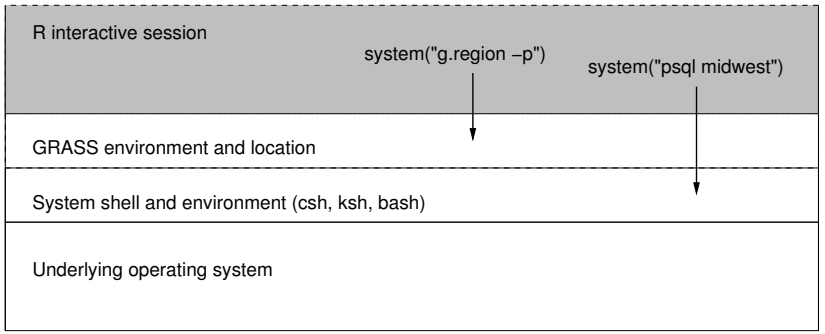
- Because GIS can be used as databases, and their tools can be better suited to some analyses and operations, it may be sensible to use one in addition to data analysis software
- There is an extra effort required when using linked software systems, because they may not integrate easily
- Since R is open source, and R spatial packages use open source components, staying with open source GIS means that many of the underlying software components are shared
- This certainly applies to R and GRASS, and through GRASS, also to R and QGIS — QGIS seemes to be more file-based than GRASS, which has an underlying data storage specification

R — GIS interfaces

- GIS interfaces can be as simple as just reading and writing files — loose coupling, once the file formats have been worked out, that is
- Loose coupling is less of a burden than it was with smaller, slower machines, which is why the **GRASS** 5 interface was tight-coupled, with R functions reading from and writing to the GRASS database directly
- The GRASS 6 interface **spgrass6** on CRAN also runs R within GRASS, but uses intermediate temporary files; the package is under development but is quite usable
- Use has been made of COM and Python interfaces to ArcGIS; typical use is by loose coupling except in highly customised work situations

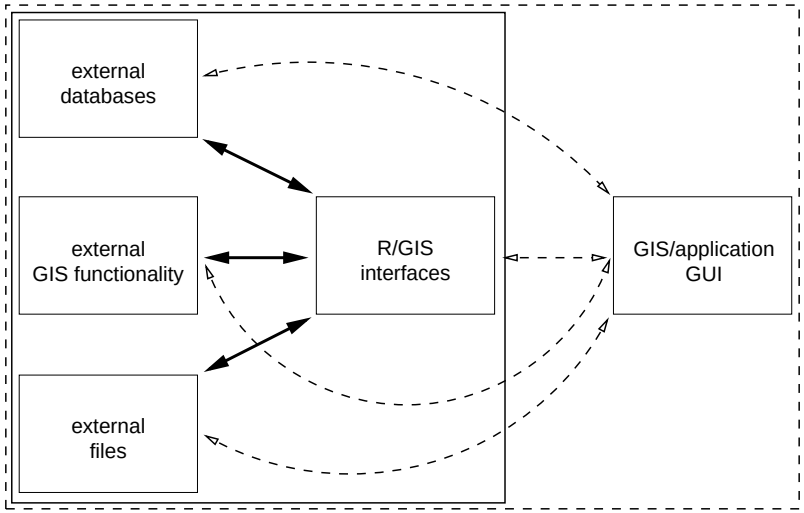
Layering of shells

The interface between R and GRASS uses the fact that GRASS modules can be run as command line programs at the shell prompt. The shell has certain environment variables for GRASS set, for example saying where the data is stored, but is otherwise a regular shell, from which R can be started. This instance of R inherits the environment variables set by GRASS



Which software component faces the user?

Finally, although for research purposes it may be prefered to have data analysis software, such as R, facing the user, it is possible to try to "embed" this component in the workflow, so that the end user does not need so much training — but then an "expert" has to codify the steps needed in advance



The GRASS location

- GRASS uses a structure of locations, with fixed projection and variable region (window) and resolution
- The PERMANENT mapset contains data that is not supposed to change, and users should create their own mapsets, where work-in-progress can be kept
- Each mapset can contain many directories, including directories for raster and vector data components, projection metadata, and various windows used in parts of the location
- Raster layers with different resolutions are resampled to the current region when used

Getting GRASS running

- The stable version of GRASS, 6.4.*, has a native Windows build, so GRASS modules and the GRASS shell are now much easier to install
- GRASS has very mature raster analysis modules, including a complete map calculator, as well as many other useful tools
- On the vector side, GRASS 6 uses a topological vector model, cleaning input data for oddities; keeping the geometries associated with the data can be challenging

Installing GRASS

- The easiest way is using the GRASS download, which is large, but contains all the dependencies on Windows.
- Type the full path to Rgui.exe at the shell prompt, for me `"C:/Program Files/R/R-3.0.1/bin/x64/Rgui.exe"`
- You may need to set the path to `R_LIBS` in user environment variables, or to add it to the command line in the GRASS shell prompt, or use `.libPaths()` within R

Using GRASS inside R

There are two ways of using GRASS inside R, by starting R inside a running GRASS session, or by initiating GRASS inside R, here using a throw-away location to show how the streams were generated, set to the extent and resolution of the elevation data from SRTM:

```
> library(spgrass6)
> myGRASS <- "/home/rsb/topics/grass/g642/grass-6.4.2"
> initGRASS(myGRASS, tempdir(),
+     SG = DEM, override = TRUE)
```

Moving data and running GRASS commands

Functions are provided in **spgrass6** for moving raster and vector data both ways. In addition, all GRASS commands replying to an interface description request can be run programmatically from within R using execGRASS. Here we resample the DEM to the resolution of the panchromatic image, and reset the region to the output raster:

```
> writeRAST6(DEM, "dem", flags = "o")
> execGRASS("g.region", rast = "dem")
> execGRASS("r.resamp.rst", input = "dem",
+     ew_res = 14.25, ns_res = 14.25,
+     elev = "DEM_resamp", slope = "slope",
+     aspect = "aspect")
> execGRASS("g.region", rast = "DEM_resamp")
> DEM_out <- readRAST6(c("DEM_resamp",
+     "slope", "aspect"))
```

Creating the stream network

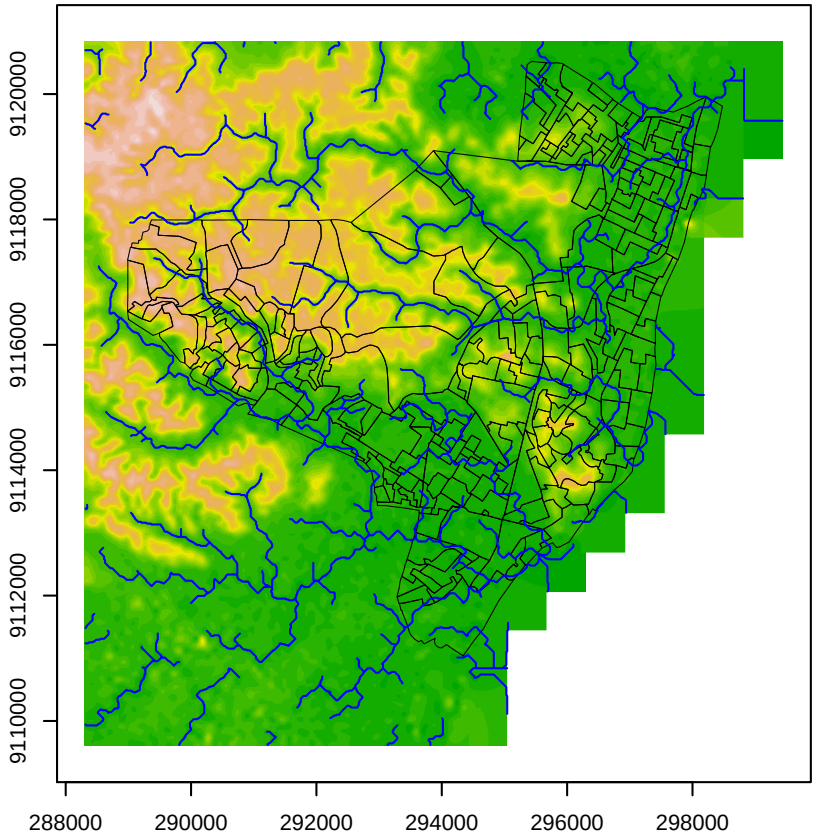
This is where the work gets done, with GRASS command `r.watershed` analysing the resampled DEM to detect depressions through which streams may flow; it can also output the boundaries of watersheds, but these are not needed here. The output is in raster form, and may also be thinned:

```
> execGRASS("r.watershed", elevation = "DEM_resamp",
+     stream = "stream", threshold = 1000L,
+     convergence = 5L, memory = 300L)
> execGRASS("r.thin", input = "stream",
+     output = "stream1", iterations = 200L)
```

Returning results

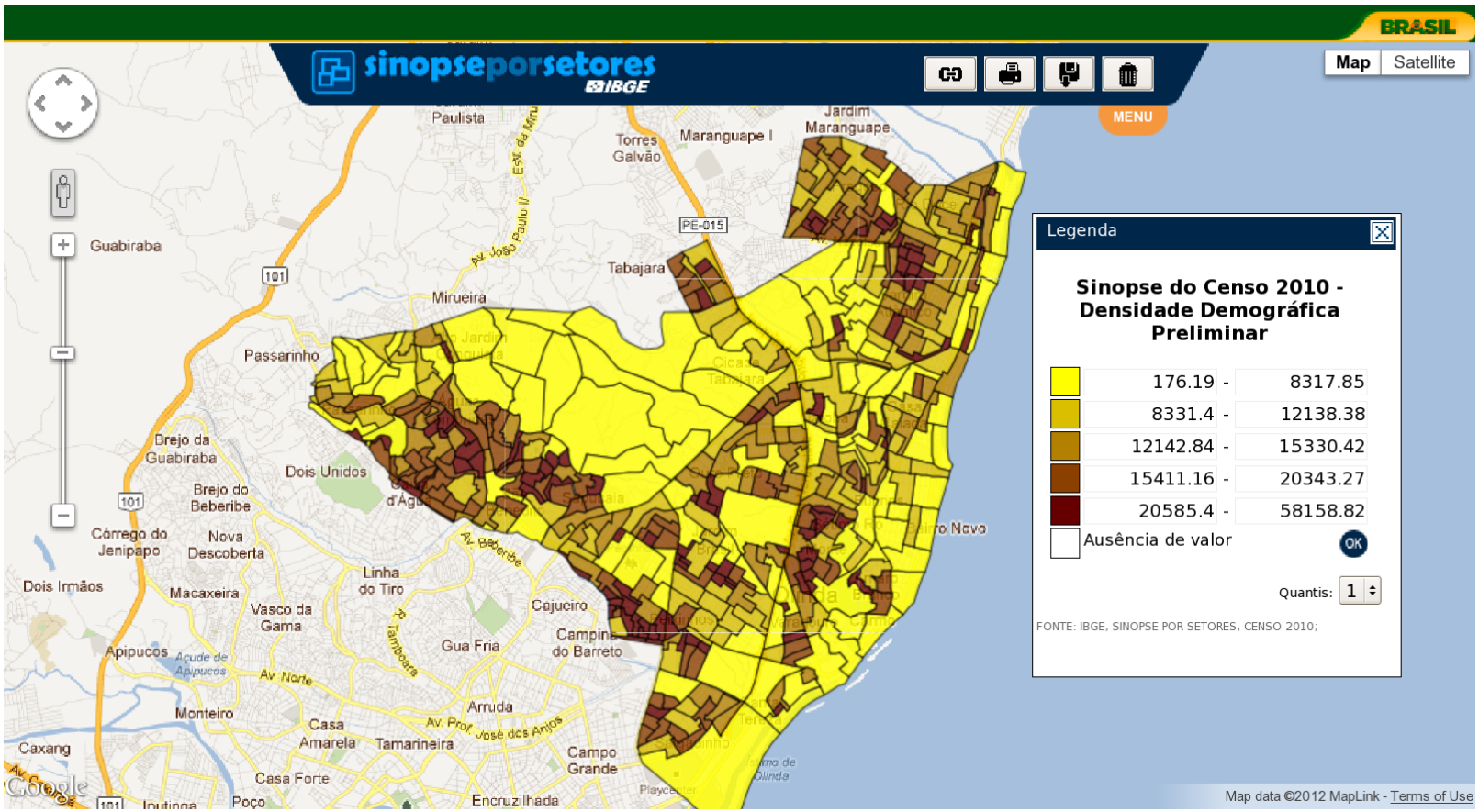
First we vectorise the raster stream network, and read it into the R session for display:

```
> execGRASS("r.to.vect", input = "stream1",
+     output = "stream", feature = "line")
> stream1 <- readVECT6("stream")
```



Population density in Olinda

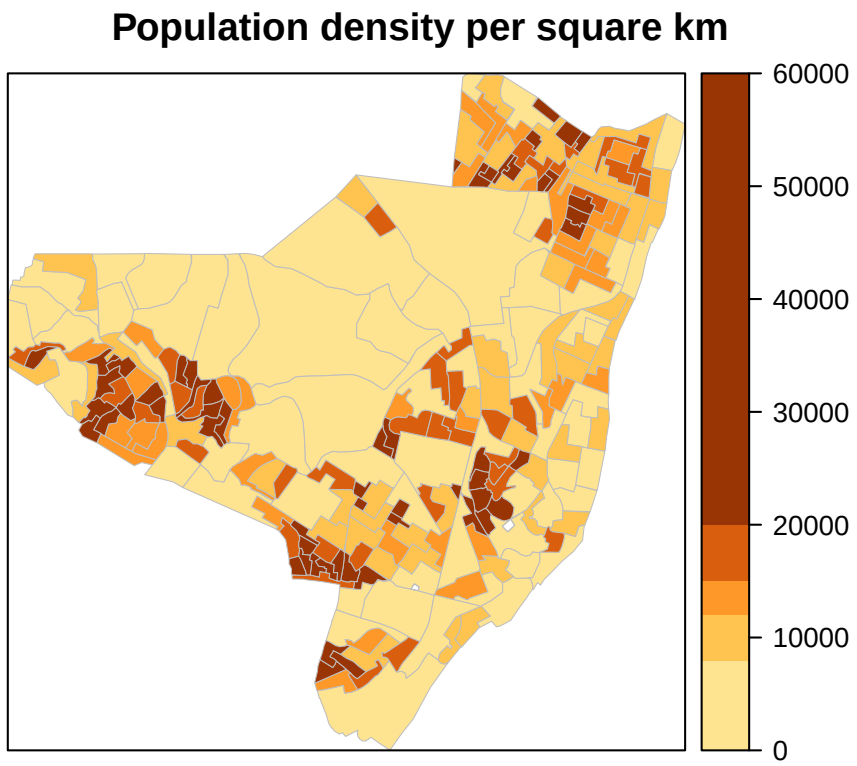
The 2010 census website publishes density data:



Population density in Olinda

So let's see how things go with the 1993 data for different tracts:

```
> olinda1$area <- sapply(slot(olinda1,
+     "polygons"), slot, "area")
> km2 <- olinda1$area/1e+06
> olinda1$dens <- olinda1$POP/km2
> library(RColorBrewer)

> tt <- "Population density per square km"
> spplot(olinda1, "dens", at = c(0,
+     8000, 12000, 15000, 20000,
+     60000), col.regions = brewer.pal(6,
+     "YlOrBr")[-1], main = tt,
+     col = "grey", lwd = 0.5)
```



Operating on geometries

```
> library(rgeos)

> getScale()

[1] 1e+08

> pols <- as(olinda1, "SpatialPolygons")
> Area <- gArea(pols, byid = TRUE)
> all.equal(unname(Area), olinda1$area)

[1] TRUE
```

The **rgeos** package interfaces the GEOS/JTS topology suite providing predicates and operations for geometries. The simplest are measures for planar geometries — only these are supported. A complication is that computational geometry may represent objects using different scaling factors, leading to geometries becoming "unclean"; GEOS uses a fixed precision representation.

Operating on geometries

```
> lns0 <- as(stream1, "SpatialLines")
> length(lns0)

[1] 321

> summary(gLength(lns0, byid = TRUE))

   Min. 1st Qu.  Median    Mean
  14.25   20.16  314.50  475.60
3rd Qu.    Max.
 710.80 2881.00

> t0 <- gTouches(lns0, lns0, byid = TRUE)
> any(t0)

[1] TRUE
```

We may start by assembling the lines created in GRASS as drainage lines into larger assemblages. The size distribution of the input data shows many segments at the resolution of the input elevation raster, but as they touch, they may be joined. We use the `gTouches` predicate to construct a matrix showing which objects do touch one another.

Operating on geometries

```
> library(spdep)

> lw <- mat2listw(t0)
> is.symmetric.nb(lw$neighbours)

[1] TRUE

> nComp <- n.comp.nb(lw$neighbours)
> IDs <- as.character(nComp$comp.id)
> lns <- gLineMerge(lns0, id = IDs)
> length(lns)

[1] 26

> summary(gLength(lns, byid = TRUE))

    Min.  1st Qu.   Median     Mean
   20.16   476.30  1197.00  5872.00
 3rd Qu.     Max.
 2839.00 65950.00
```

Once we have a matrix of touching lines, which is by definition symmetric, we can convert it to a sparse list representation and identify its graph components with functions in **spdep**. After merging the lines into graph component objects, we see that the length distribution has changed dramatically.

Intersections - channels and tracts

```
> GI <- gIntersects(lns, pols,
+     byid = TRUE)
> unname(which(GI[2, ]))

[1] 24

> res <- numeric(length = nrow(GI))
> for (i in seq(along = res)) {
+     if (any(GI[i, ])) {
+         resi <- gIntersection(lns[GI[i,
+             ]], pols[i])
+         res[i] <- gLength(resi)
+     }
+ }
> olinda1$stream_len <- res
```

When many geometries are involved in topological operations, like finding the drainage channels in each tract and measuring their length, it is often prudent to run the equivalent predicate first — here `gIntersects`, and then only operate on combinations of objects which meet that condition. We assign the drainage channel lengths to the tracts object.

Intersections - channels and tracts

```
> tree <- gBinarySTRtreeQuery(lns,
+     pols)
> tree[[2]]

[1]  2 24 25

> res1 <- numeric(length = length(tree))
> for (i in seq(along = res1)) {
+     if (!is.null(tree[[i]])) {
+         gi <- gIntersection(lns[tree[[i]]],
+             pols[i])
+         res1[i] <- ifelse(is.null(gi),
+             0, gLength(gi))
+     }
+ }
> all.equal(res, res1)

[1] TRUE
```

It is also possible to build and query a Sort-Tile-Recursive tree to check envelopes of objects for intersection. The advantages of doing this increase with the number of objects, as the predicate output matrix is returned dense, although many elements are `FALSE`. The list returned by STR tree query is sparse; for example, the **BARD** package needs contiguities of many irregular polygons to study election redistricting, and tree query does make a big difference.

Intersections - buffered channels and tracts

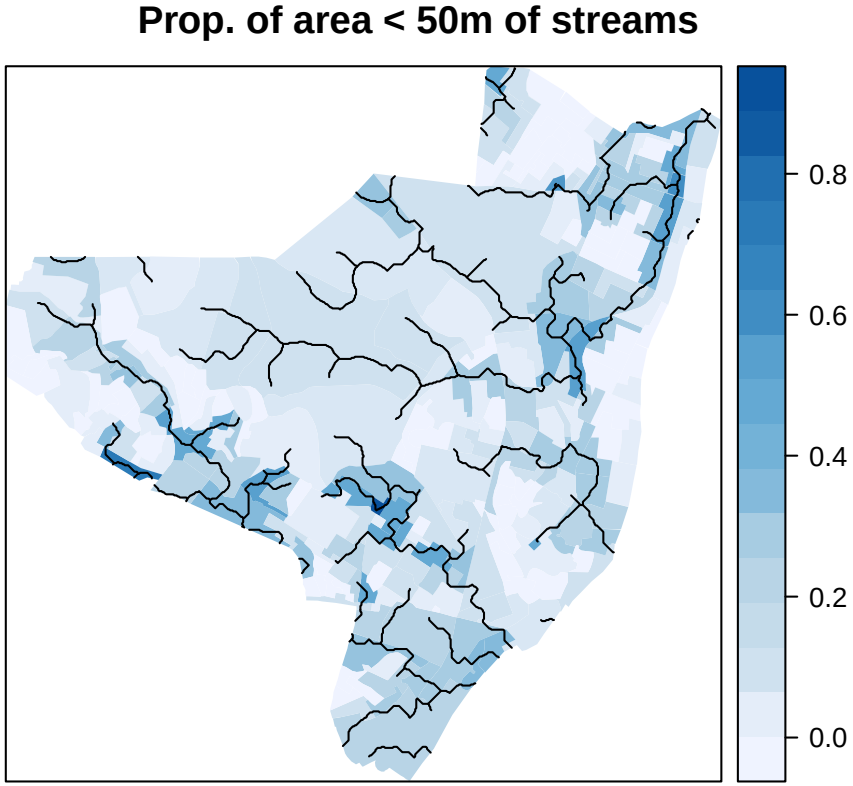
```
> buf50m <- gBuffer(lns, width = 50)
> GI1 <- gIntersects(buf50m, pols,
+     byid = TRUE)
> res <- numeric(length = nrow(GI1))
> for (i in seq(along = res)) {
+     if (any(GI1[i, ])) {
+         out <- gIntersection(buf50m[GI1[i,
+             ]], pols[i])
+         res[i] <- gArea(out)
+     }
+ }
> olinda1$buf_area <- res
> prop <- olinda1$buf_area/olinda1$area
> olinda1$prop_50m <- prop
```

Just knowing the lengths by tract may be less useful than knowing the proportion of areas within 50m of drainage channels. We can use `gBuffer` to create a single `SpatialPolygons` object, which we can intersect with the tracts — using an STR tree here will not lead to savings because we only have one buffer object.

Drainage channels in Olinda

If we merge the tracts within the city limits, we can make a cookie-cutter, and use it to clip/crop other objects to match. So now we can display the proportions of the areas of tracts within 50m of drainage channels in the city, omitting channels outside the city bounds:

```
> bounds <- gUnaryUnion(pols)
> stream_inside <- gIntersection(lns,
+     bounds)

> bl <- colorRampPalette(brewer.pal(5,
+     "Blues"))
> tt <- "Prop. of area < 50m of streams"
> spplot(olinda1, "prop_50m",
+     col.regions = bl(20), col = "transparent",
+     sp.layout = list("sp.lines",
+         stream_inside), main = tt)
```



Reading rasters

- There are very many raster and image formats; some allow only one band of data, others think data bands are RGB, while yet others are flexible
- 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
- Much more support is 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, which helps handle large rasters

But let's keep things local

Using Landsat 7 panchromatic imagery, we can get a similar contextual effect, here with histogram equalised 15m resolution:

```
> pan <- readGDAL("L8s.tif")

L8s.tif has GDAL driver GTiff
and has 800 rows and 800 columns

> bb0 <- set_ReplCRS_warn(FALSE)
> proj4string(pan) <- CRS("+init=epsg:31985")
> bb0 <- set_ReplCRS_warn(TRUE)
> brks <- quantile(pan$band1,
+     seq(0, 1, 1/255))
> pan$lut <- findInterval(pan$band1,
+     brks, all.inside = TRUE)

> image(pan, "lut", col = grey.colors(20))
> plot(olinda1, add = TRUE, border = "brown",
+     lwd = 0.5)
> title(main = "Landsat panchromatic channel 15m")
```

Constructing and using NDVI

Landsat ETM also lets us construct a 30m resolution normalised difference vegetation index (of more relevance for other diseases), and plot it using a ColorBrewer palette:

```
> letm <- readGDAL("L_ETMps.tif")

L_ETMps.tif has GDAL driver GTiff
and has 400 rows and 400 columns

> bb0 <- set_ReplCRS_warn(FALSE)
> proj4string(letm) <- CRS("+init=epsg:31985")
> bb0 <- set_ReplCRS_warn(TRUE)
> letm$ndvi <- (letm$band4 - letm$band3)/(letm$band4 +
+     letm$band3)
> library(RColorBrewer)
> mypal <- brewer.pal(5, "Greens")
> greens <- colorRampPalette(mypal)

> image(letm, "ndvi", col = greens(20))
> plot(olinda1, add = TRUE)
```

Writing objects

- In **rgdal**, `writeGDAL` can write for example multi-band GeoTiffs, but there are fewer write than read drivers; in general CRS and geogreferencing are supported — see `gdalDrivers`
- The **rgdal** function `writeOGR` can be used to write vector files, including those formats supported by drivers, including now KML — see `ogrDrivers`
- External software (including different versions) tolerate output objects in varying degrees, quite often needing tricks - see mailing list archives
- In **maptools**, there are functions for writing **sp** objects to shapefiles — `writeSpatialShape`, etc., as Arc ASCII grids — `writeAsciiGrid`, and for using the R PNG graphics device for outputting image overlays for Google Earth

Using the direct KML method

One of the nice things about linking to OSGeo software is that when someone contributes a driver, it is there for other software too. With the increasing availability of software like Google Earth (or Google Maps), the ability to display data or results in a context that is familiar to the user can be helpful, but we need geographical coordinates:

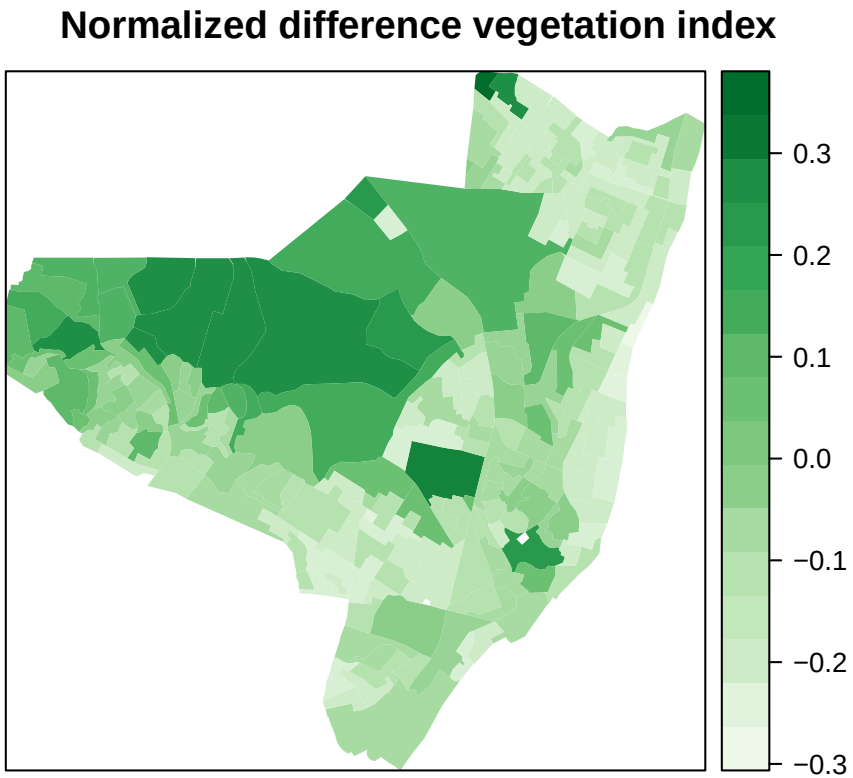
```
> olinda_ll <- spTransform(olinda1, CRS("+proj=longlat +datum=WGS84"))
> writeOGR(olinda_ll, dsn = "olinda_ll.kml", layer = "olinda_ll", driver = "KML",
+     overwrite_layer = TRUE)
```

Olinda census district boundaries in GE

Overlay operations

The basic approach is to query one spatial data object using a second spatial data object of the same or of a different class. The query result will typically be another spatial data object or an object pointing to one of the input objects. We also remove two census traccts with missing population:

```
> o_ndvi <- over(olinda1, letm[,
+     , "ndvi"], fn = median)
> o_alt <- over(olinda1, DEM_out[,
+     , "DEM_resamp"], fn = median)
> olinda1A <- spCbind(olinda1,
+     cbind(o_ndvi, o_alt))
> olinda2 <- olinda1A[!is.na(olinda1A$POP),
+     ]
```



Extract operations

It is possible to use `extract` methods from the **raster** package to aggregate queried values; note that the arguments are reversed in order compared to `over` methods in **sp**. We convert our `SpatialPixelsDataFrame` object to a `RasterStack` object, and obtain when querying with a `SpatialPolygons` object, a list of numeric matrices with the values of all variables in rasters in the `RasterStack` object, for all the cells with centres falling in each `Polygons` object. As we can see by comparison, `extract` methods from the **raster** package yield the same results as `over` methods in **sp** for 28.5m NDVI medians.

```
> library(raster)
```

```
> TMrs <- stack(letm)
> e1 <- extract(TMrs, as(olinda2, "SpatialPolygons"))
> all.equal(sapply(e1, function(x) median(x[, "ndvi"])), olinda2$ndvi)
```

```
[1] TRUE
```

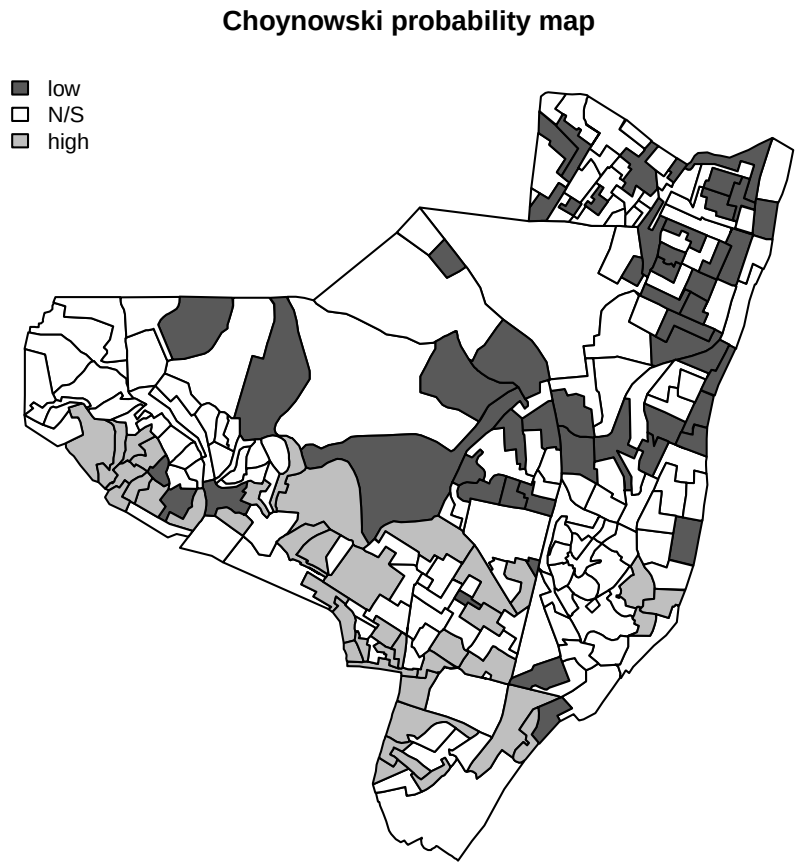
Probability maps and smoothing

- Choynowski proposed a method for making probability maps in 1959, still described by contemporary authors — it splits between upper and lower tails
- It is also possible to say just `ppois(Observed, Expected)` to get a similar result. If the observations are distributed as assumed, this will indicate which regions appear unusual
- Probability maps typically mirror the spatial pattern of relative risks; using Empirical Bayes smoothing will use the population at risk to adjust relative risks
- Empirical Bayes smoothing is implemented for the method-of-moments global and local cases in **spdep** and for the ML global case in **DCluster**

Choynowski probability map

The legacy Choynowski approach is implemented in spdep

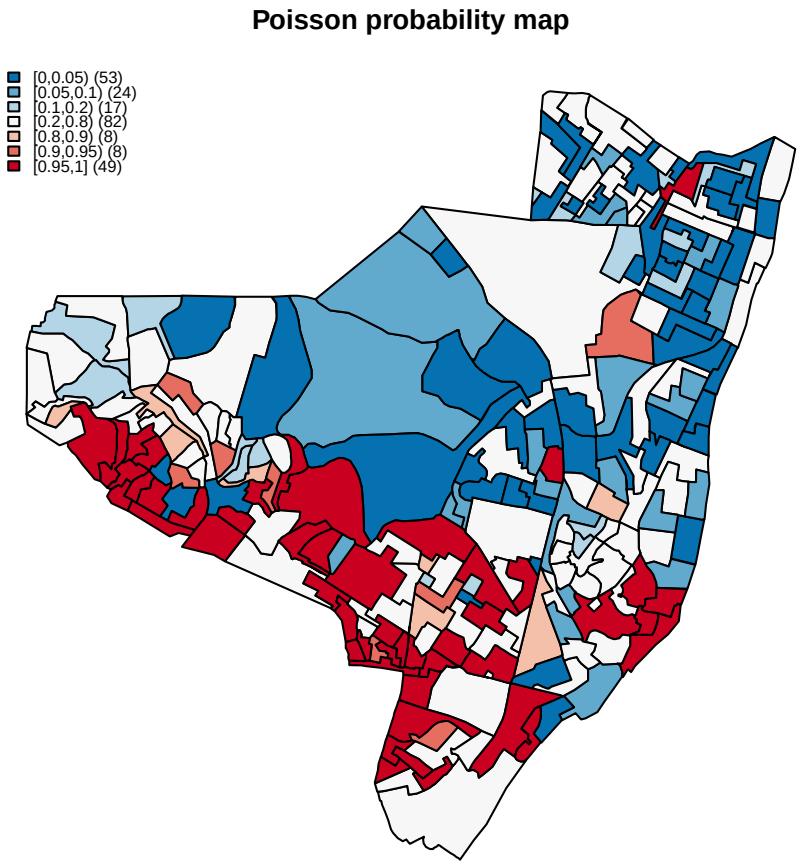
```
> library(spdep)
> ch <- choynowski(olinda2$CASES,
+     olinda2$POP)
```



Poisson probability map

The Poisson probability map in one tail is identical with the Choynowski approach, but shows both tails rather than folding them together

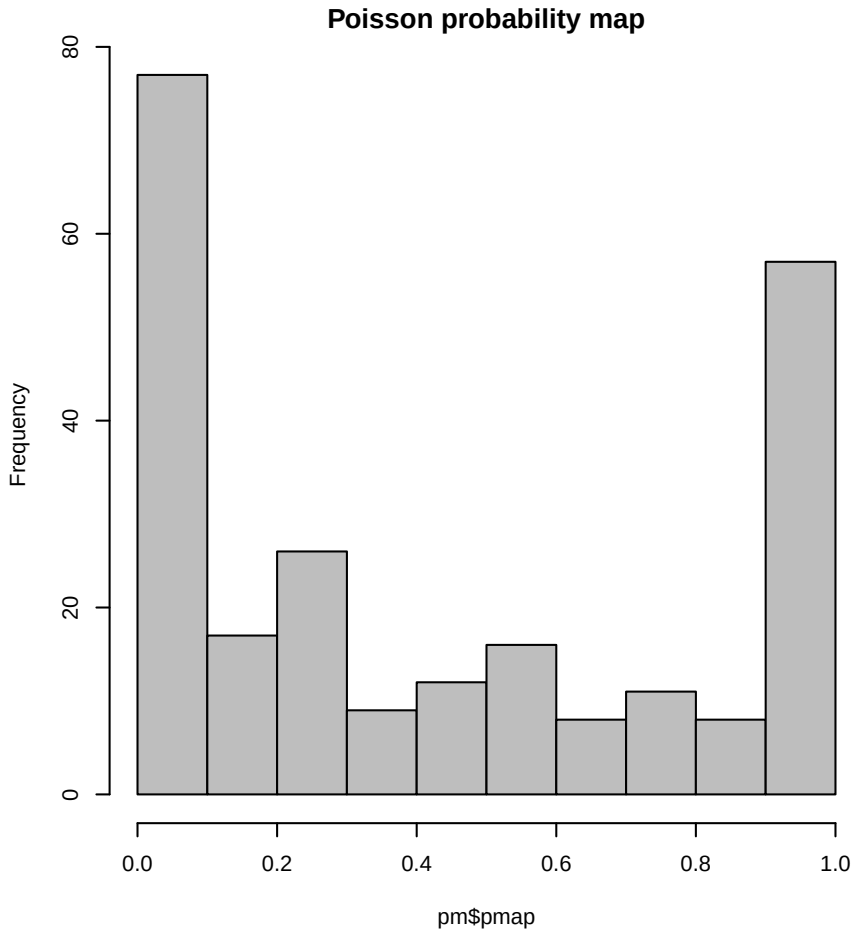
```
> pm <- probmap(olinda2$CASES,
+     olinda2$POP)
> names(pm)

[1] "raw"      "expCount" "relRisk"
[4] "pmap"

> olinda2$SMR <- pm$relRisk
```



Weaknesses of probability maps

If the underlying distribution of the data does not agree with our assumption, we may get several possible processes mixed up, overdispersion with spatial dependence:

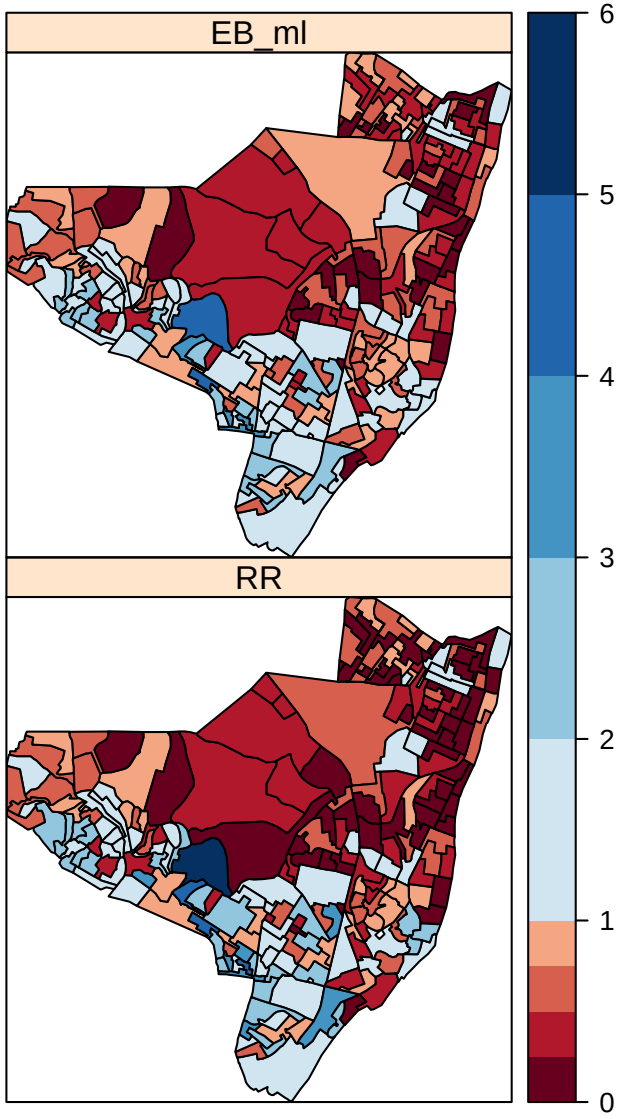
```
> table(findInterval(pm$pmap,
+     seq(0, 1, 1/10)))

 1  2  3  4  5  6  7  8  9 10
77 17 26  9 12 16  8 11  8 57
```



Some Empirical Bayes rates?

We can compare the relative rate and the EB relative rate:

```
> library(DCluster)
> olinda2$RR <- olinda2$CASES/olinda2$Expected
> olinda2$EB_ml <- empbaysmooth(olinda2$CASES,
+     olinda2$Expected)$smthrr

> spplot(olinda2, c("RR", "EB_ml"),
+     col.regions = brewer.pal(10,
+         "RdBu"), at = c(seq(0,
+         1, 0.25), seq(1, 6,
+         1)))
```



Spatial autocorrelation

- A wide range of scientific disciplines have encountered spatial autocorrelation among areal entities, with the term "Galton's problem" used in several
- In his exchange with Tyler in 1889, Galton questioned whether observations of marriage laws across areal entities constituted independent observations, since they could just reflect a general pattern from which they had all descended
- So positive spatial dependence tends to reduce the amount of information contained in the observations, because proximate observations can in part be used to predict each other
- Here we will be concerned with areal entities that are defined as neighbours, for chosen definitions of neighbours

Areal data

- Spatial data often take the form of polygon entities defined by boundaries for which observations are available
- The observed data are frequently aggregations within the boundaries, such as population counts
- Exceptionally, the areal entities themselves constitute the units of observation, for example when studying local government behaviour where decisions, such as local taxes, are taken at the level of the entity
- By and large, though, areal entities are aggregates used to tally measurements, like voting results at polling stations

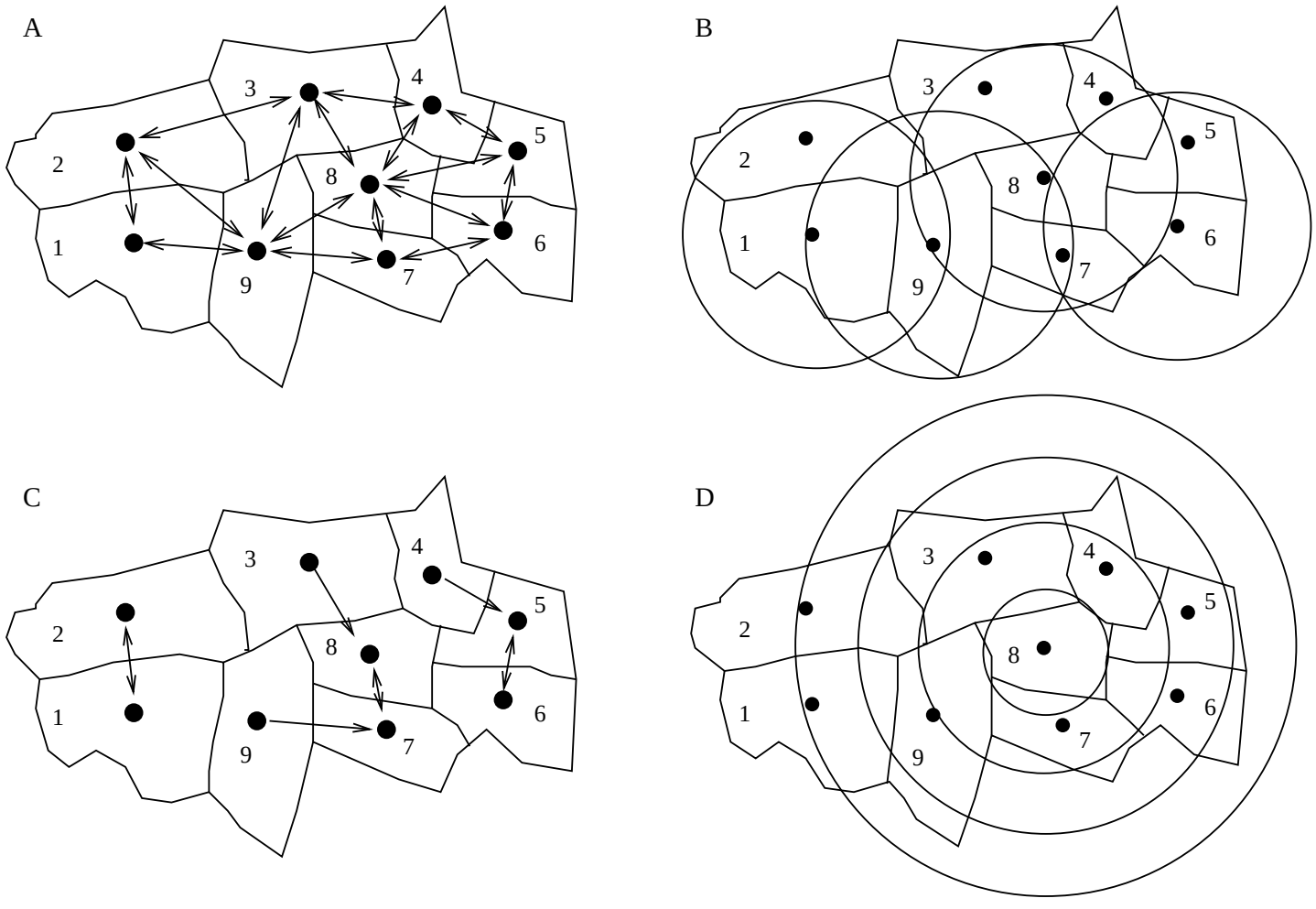
Spatial weights

- Creating spatial weights is a necessary step in using areal data, perhaps just to check that there is no remaining spatial patterning in residuals
- Making the weights is, however, not easy to do, so a number of functions are included in the **spdep** package to help
- Further functions are found in some ecology packages, such as the **ade4** package — this package also provides `nb2neig` and `neig2nb` converters for inter-operability
- The first step is to determine the sets of neighbours for each observation, the second to assign weights to each neighbour relationship

Neighbours

- In the **spdep** package, neighbour relationships between n observations are represented by an object of class `nb`
- This is a list of length n with the index numbers of neighbours of each component recorded as an integer vector
- If any observation has no neighbours, the component contains an integer zero
- It also contains attributes, typically a vector of character region indentifiers, and a logical value indicating whether the relationships are symmetric

Selected neighbourhood schemes



Neighbour relations

- It is usual in the literature to define the contiguity relation in terms of sets $N_{(i)}$ of neighbours of zone or site i, where $c_{ij} = 1$ if i is linked to j and $c_{ij} = 0$ otherwise
- This implies no use of other information than that of neighbourhood set membership
- Set membership may be defined on the basis of shared boundaries, of centroids lying within distance bands, or other a priori grounds

Neighbourhood sets

Zone	A: contiguity		B: distance	
	number	neighbours	number	neighbours
1	2	(2, 9)	2	(2, 9)
...				
6	3	(5, 7, 8)	2	(5, 7)
...				
8	6	(3, 4, 5, 6, 7, 9)	4	(3, 4, 7, 9)
9	5	(1, 2, 3, 7, 8)	3	(1, 7, 8)

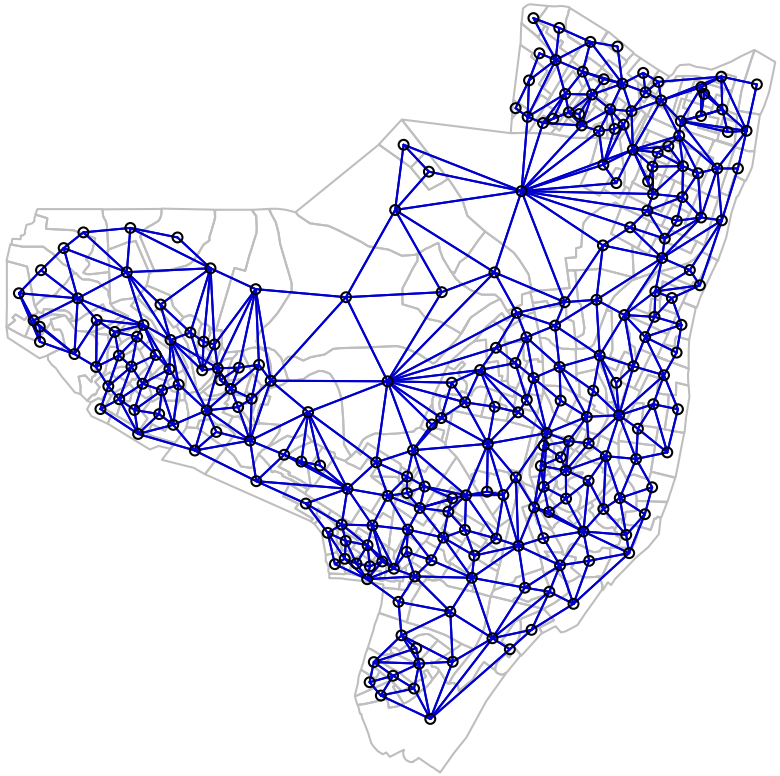
Spatial weights styles

- Spatial weights are in part difficult to use because the identified neighbour relationships need to be given a value
- Even when the values are binary 0/1, the issue of what to do with no-neighbour observations arises
- Then the weights can be standardised, for example so that row sums are unity, or that the weights sum to the number of observations
- Finally, the weights may be generalised to reflect something that we "know" about the spatial dependency process, for example that it falls with increasing distance

Neighbours

In order to investigate spatial dependence, we need a list of neighbours. We take Queen contiguities of the tract polygons:

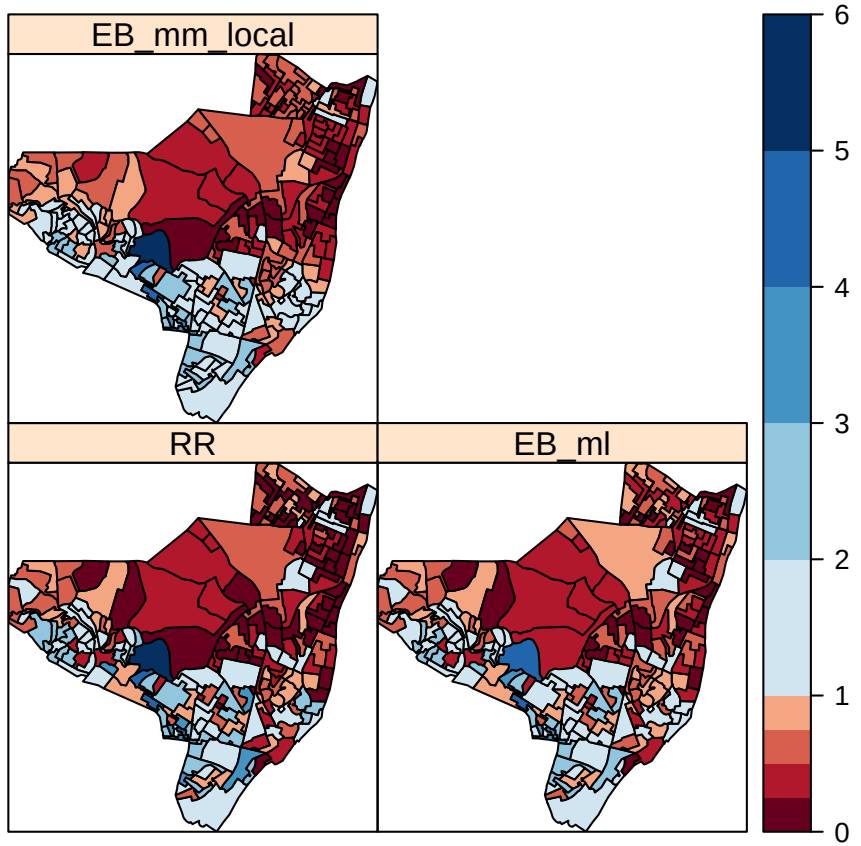
```
> nb <- poly2nb(olinda2)
> nb

Neighbour list object:
Number of regions: 241
Number of nonzero links: 1324
Percentage nonzero weights: 2.279575
Average number of links: 5.493776
```



Local Empirical Bayes smoothing

If instead of shrinking to a global rate, we shrink to a local rate, we may be able to take unobserved heterogeneity into account; here we use the list of neighbours:

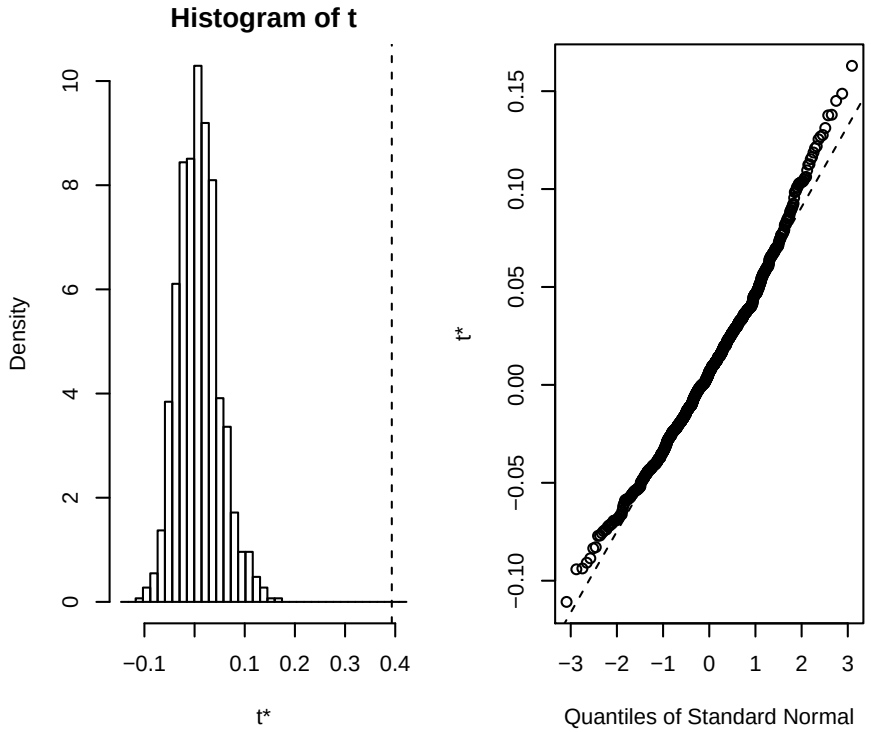
```
> olinda2$Observed <- olinda2$CASES
> eb2 <- EBlocal(olinda2$Observed,
+     olinda2$Expected, nb)
> olinda2$EB_mm_local <- eb2$est
```



Moran's *I*

DCluster provides a permutation bootstrap test for spatial autocorrelation of the difference between observed and expected counts:

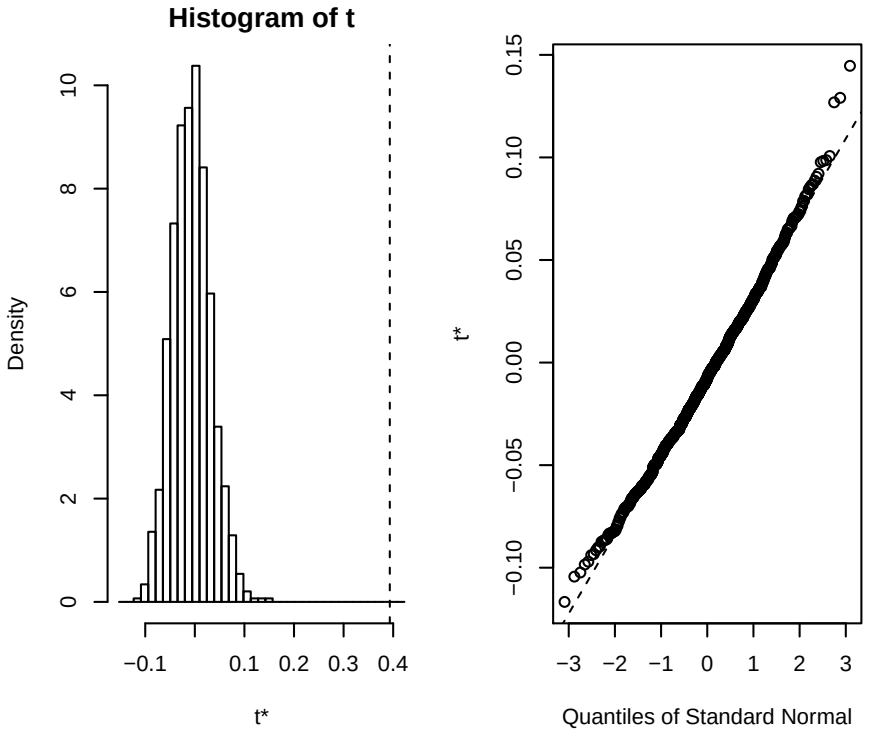
```
> lw <- nb2listw(nb)
> set.seed(130709)
> moran.boot <- boot(as(olinda2,
+     "data.frame"), statistic = moranI.boot,
+     R = 999, listw = lw, n = length(nb),
+     S0 = Szero(lw))
```



Moran's *I*

It also provides parametric bootstraps for variants, including the Negative Binomial:

```
> moran.pgboot <- boot(as(olinda2,
+     "data.frame"), statistic = moranI.pboot,
+     sim = "parametric", ran.gen = negbin.sim,
+     R = 999, listw = lw, n = length(nb),
+     S0 = Szero(lw))
```



Assunção and Reis' correction

The Assunção and Reis' correction to Moran's *I* is implemented in **spdep**:

```
> EBImoran.mc(olinda2$CASES, olinda2$Expected, lw, nsim = 999)

Monte-Carlo simulation of Empirical Bayes Index

data:  cases: olinda2$CASES, risk population: olinda2$Expected
weights: lw
number of simulations + 1: 1000

statistic = 0.3941, observed rank = 1000, p-value = 0.001
alternative hypothesis: greater
```

Maybe some modelling?

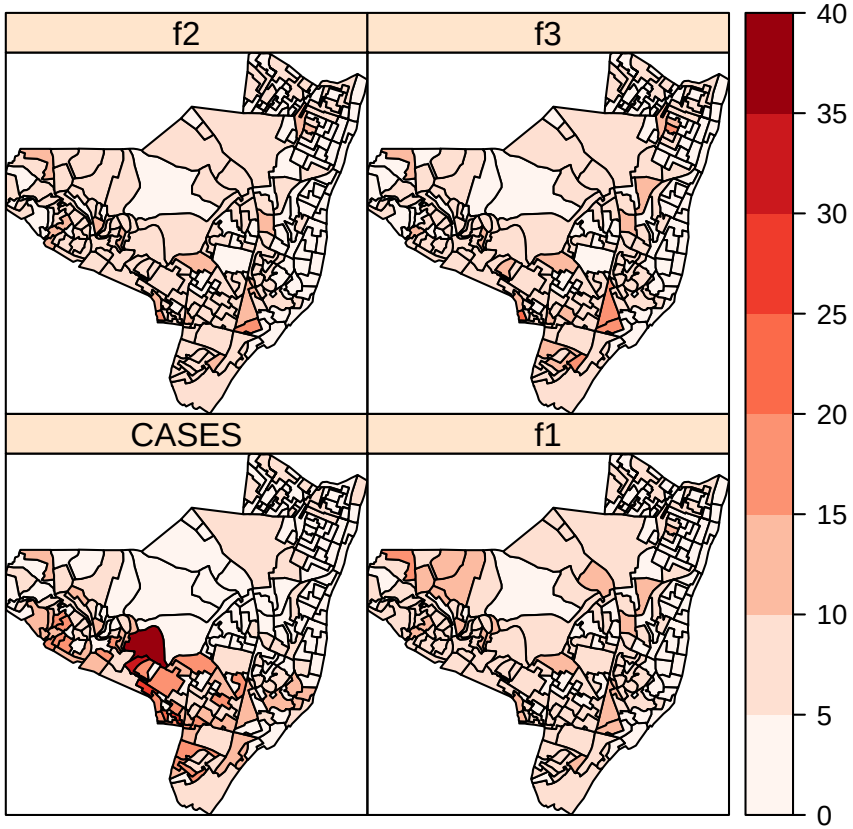
To show how modelling and cartography can be inter-woven, let us fit some quasipoisson models, after seeing that — once again — spatial dependence and overdispersion seem to be entwined:

```
> m0p <- glm(CASES ~ 1 + offset(log(Expected)), data = olinda2, family = poisson)
> DeanB(m0p)

Dean's P_B test for overdispersion

data:  m0p
P_B = 50.2478, p-value < 2.2e-16
alternative hypothesis: greater

> m0 <- glm(CASES ~ 1 + offset(log(Expected)), data = olinda2, family = quasipoisson)
> m1 <- update(m0, . ~ . + DEPRIV)
> m2 <- update(m1, . ~ . + ndvi)
> m3 <- update(m2, . ~ . + DEM_resamp)
> anova(m0, m1, m2, m3, test = "F")

Analysis of Deviance Table

Model 1: CASES ~ 1 + offset(log(Expected))
Model 2: CASES ~ DEPRIV + offset(log(Expected))
Model 3: CASES ~ DEPRIV + ndvi + offset(log(Expected))
Model 4: CASES ~ DEPRIV + ndvi + DEM_resamp + offset(log(Expected))
  Resid. Df Resid. Dev Df Deviance       F   Pr(>F)
1       240     1212.5
2       239     1096.6  1  115.871 24.9914 1.12e-06 ***
3       238     1059.4  1   37.183  8.0197 0.005025 **
4       237     1038.2  1   21.281  4.5900 0.033179 *
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

Plotting the fitted values

We can use `spplot` methods to plot sequences of variables on the same scale using lattice graphics, once more using a Color Brewer palette:

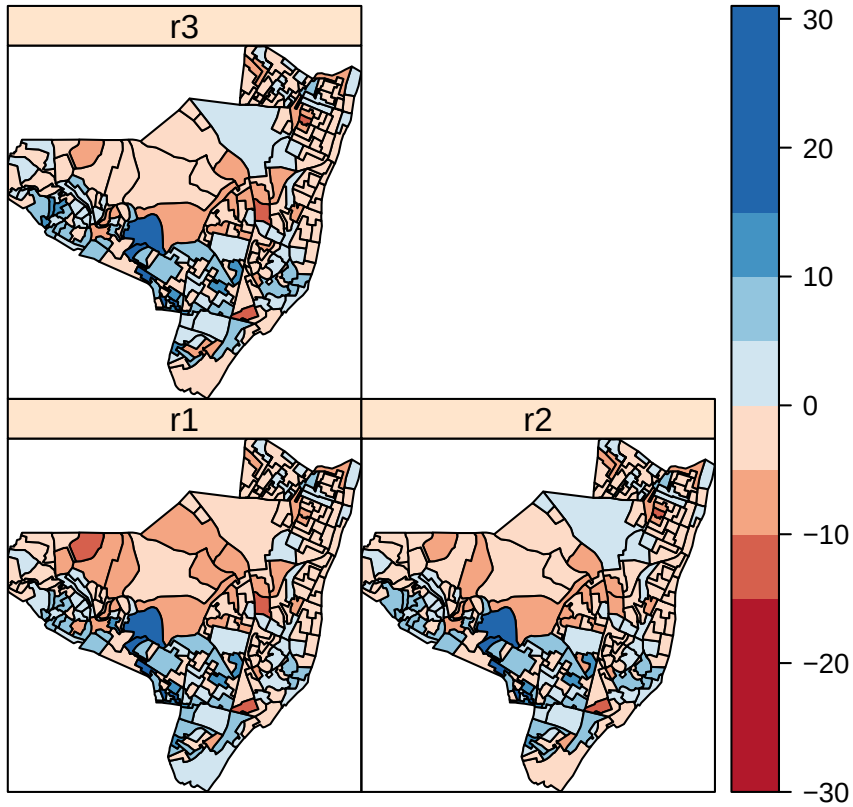
```
> olinda2$f1 <- fitted(m1)
> olinda2$f2 <- fitted(m2)
> olinda2$f3 <- fitted(m3)

> spplot(olinda2, c("CASES", "f1",
+     "f2", "f3"), col.regions = brewer.pal(8,
+     "Reds"), at = seq(0, 40,
+     5))
```



Residual maps

So we can also plot residual maps for the three nested models, but getting the palette divergence in the right place takes a little more care:

```
> olinda2$r1 <- residuals(m1,
+     type = "response")
> olinda2$r2 <- residuals(m2,
+     type = "response")
> olinda2$r3 <- residuals(m3,
+     type = "response")

> spplot(olinda2, c("r1", "r2",
+     "r3"), col.regions = brewer.pal(8,
+     "RdBu"), at = c(-30, -15,
+     -10, -5, 0, 5, 10, 15, 31))
```



Introduction

- In order to model the data, we need to try to fit random effects with the stipulated distributions
- The Poisson-Gamma model (PG) fits a model without spatially structured random effects
- The Besag, York and Mollič (BYM) model includes a spatially structured random effect
- The Besag structure is a conditional spatialautoregressive model

INLA — integrated nested Laplace approximation

- These models are not very complicated, but many WinBUGS models run for hours or days — making inference is impeded by time and computation constraints
- INLA — `http://www.r-inla.org/` provides ways to reach very similar inferences much faster, using integrated nested Laplace approximation instead of Gibbs sampling in MCMC
- The `inla` function in the **INLA** package (installed from the homepage of the INLA project, not yet CRAN) uses the familiar formula interface
- It adds an extra function in the formula to define which model is to be used

INLA BYM model

By using the standard White book framework of formula, data, and family, `inla` is easy to use, but only offers some built-in models:

```
> if (!("INLA" %in% .packages(all = TRUE))) source("http://www.math.ntnu.no/inla/givemeINLA.R")
> library(INLA)
> olinda2$INLA_ID <- 1:nrow(olinda2)
> INLA_BYM <- inla(CASES ~ DEPRIV +
+     ndvi + DEM_resamp + f(INLA_ID,
+     model = "bym", graph = nb2mat(nb,
+         style = "B")) + offset(log(Expected)),
+     family = "poisson", data = as(olinda2,
+         "data.frame"), control.predictor = list(compute = TRUE))
```

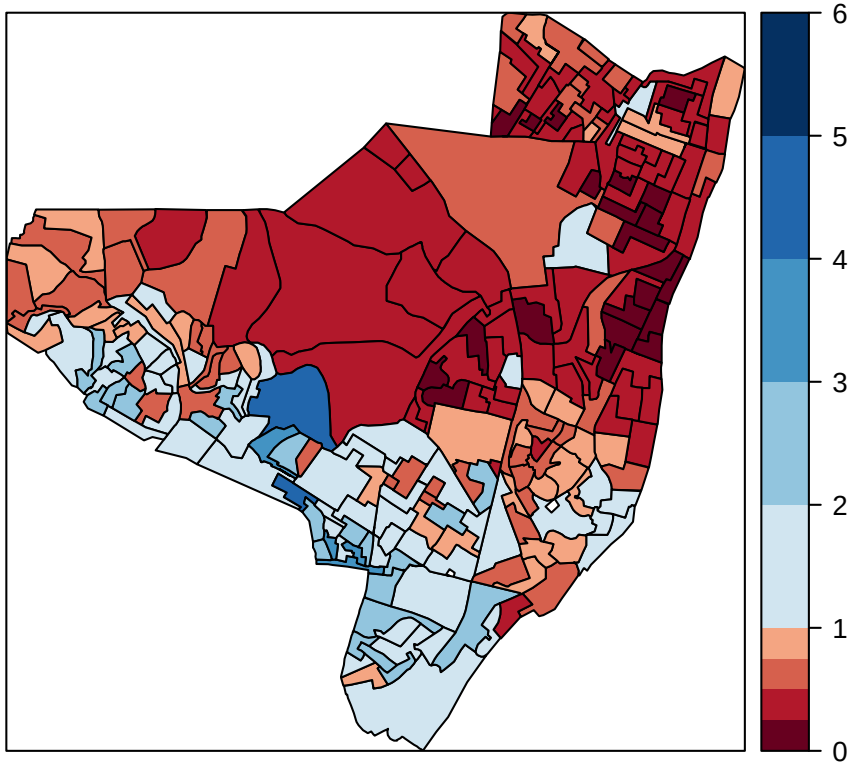
INLA BYM model

```
> summary(INLA_BYM)

Call:
c("inla(formula = CASES ~ DEPRIV + ndvi + DEM_resamp + f(INLA_ID, ",  "    model = \"bym\", graph = nb2mat

Time used:
 Pre-processing    Running inla Post-processing           Total
         0.2050          0.4448          0.0461          0.6959

Fixed effects:
               mean     sd 0.025quant 0.5quant 0.975quant kld
(Intercept) -0.2299 0.1869    -0.5990  -0.2290     0.1347   0
DEPRIV       0.4010 0.3443    -0.2756   0.4013     1.0767   0
ndvi         0.7181 0.5946    -0.4476   0.7175     1.8877   0
DEM_resamp  -0.0124 0.0063    -0.0247  -0.0124     0.0001   0

Random effects:
Name   Model
 INLA_ID   BYM model

Model hyperparameters:
                                          mean    sd      0.025quant 0.5quant 0.975quant
Precision for INLA_ID (iid component)      5.8902  2.1995  2.8640     5.4618  11.3476
Precision for INLA_ID (spatial component)  1.4401  0.5270  0.6358     1.3679   2.6753

Expected number of effective parameters(std dev): 145.78(7.159)
Number of equivalent replicates : 1.653

Marginal Likelihood:  -615.78
CPO and PIT are computed

Posterior marginals for linear predictor and fitted values computed
```

INLA BYM model

We can also extract the INLA BYM standardised rates:

```
> olinda2$INLA <- exp(INLA_BYM$summary.linear.predictor[,
+     1] - log(olinda2$Expected))

> spplot(olinda2, c("INLA"), col.regions = brewer.pal(10,
+     "RdBu"), at = c(seq(0, 1,
+     0.25), seq(1, 6, 1)))
```



CARBayes BYM model

The **CARBayes** package may be used to fit a BYM model using MCMC, but is typically much slower than `inla`, yielding the same results:

```
> library(CARBayes)

> df <- as(olinda2, "data.frame")
> attach(df)
> obj <- poisson.bymCAR(CASES ~ DEPRIV + ndvi + DEM_resamp + offset(log(Expected)),
+     W = nb2mat(nb, style = "B"), n.sample = 30000, burnin = 20000, thin = 10)
> detach(df)

> olinda2$CARBayes <- obj$fitted.values[, 1]/olinda2$Expected
```

INLA BYM model

We can confirm this by plotting the rates against each other, where the performance improvement is apparent — here we see the means, but the distributions are similar too:

```
> splom(as(olinda2, "data.frame")[,
+     c("RR", "INLA", "CARBayes")])
```