

Applied Spatial Data Analysis with R

Roger Bivand¹

12 June 2014

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

Overview

1 Spatial data in R

- Study area
- Representing spatial data
- Coordinate reference systems
- Raster data
- Exporting data

2 Handling spatial data in R: GRASS interface

3 Handling spatial data in R: methods

- Topology predicates and operations
- Overlay

4 Worked examples in spatial statistics

- Disease mapping
- Spatial autocorrelation
- Regression
- Bayesian spatial regression with INLA

R refresher

- Many objects of interest in data analysis can be expressed as lists of numbers
- R sees the world this way too, and almost everything is expressed as vectors or lists of one kind or another
- R at its simplest behaves like an overgrown calculator, so that:

```
> 2 + 2
```

```
[1] 4
```

```
> 2 * pi * 10
```

```
[1] 62.83185
```

R refresher

- Examples are made up of the R command line prompt > followed by the instruction in italics, and then the output
- The output of a vector printed to the screen is preceded by the number in sequence of the vector value in square brackets; here the vector only has a length of one
- R lets us assign values to symbolic variables, so we can also do:

```
> a <- 2
> a
[1] 2
> a + a
[1] 4
```

R refresher

- Assignment does not print any output (usually), but we can see what is inside a symbolic variable (object) by entering its name
- R is built up using functions, some of which we can use on our symbolic variable (or object) a
- str() shows the structure of a in summary, it is numeric — here num, and has the value 2; length() tells us how long it is

```
> str(a)  
num 2  
> length(a)  
[1] 1
```

R refresher

- `class()` tells us what class it belongs to
- The values returned by functions can be assigned to new symbolic variables, here `ca`, which is a character vector

```
> ca <- class(a)
> ca
[1] "numeric"
> str(ca)
chr "numeric"
```

R refresher

R can express arithmetic and other operations directly, or through functions, which most often are vectorised. Let's generate some data:

```
> a <- c(1, 1, 1, 1, 1, 2, 2, 2, 2, 2)  
> b <- c(1, 2, 1, 2, 1, 2, 1, 2, 1, 2)  
> z <- c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

where `c()` is a function used to join objects of the same kind together into a vector.

R refresher

Doing the addition element for element is tiresome, so the `sum()` function helps — there are functions for most things in R

```
> a[1] + a[2] + a[3] + a[4] + a[5] + a[6] + a[7] + a[8] + a[9] +
+      a[10]

[1] 15

> sum(a)

[1] 15

> sum(a^2)

[1] 25

> sum(a * b)

[1] 23
```

R refresher

- When we talk of arguments to a function, we mean the (possibly named) arguments passed to the function for evaluation
- We can access the arguments using the function help pages, or briefly using `args()`:

```
> args(sum)
function (..., na.rm = FALSE)
NULL
```

- Now we will look at why data-analytical software should be used to analyse data
- Problems with the data are very common, respondents who do not answer a particular question, equipment that fails occasionally, rain smudging field data collection sheets, etc., and the software needs to be able to handle them

R refresher

- As we see, `sum()` has an argument called `na.rm`, which is set to `FALSE` by default
- This means that `NA` — not available — values will not be removed by default, and the summation will fail, returning an `NA`, and alerting the analyst to problems with the data

```
> a[3] <- NA
> a
[1]  1  1 NA  1  1  2  2  2  2
> sum(a)
[1] NA
> sum(a, na.rm = TRUE)
[1] 14
```

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 (packages are kept in libraries and extend R's basic functions and classes):

```
> 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"    "SETOR_"      "SETOR_ID"     "VAR5"        "DENS_DEMO"
[7] "SET"          "CASES"        "POP"         "DEPRIV"
```

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

Map of deprivation variable



Data frames

- Very often data are stored in a data frame with rows representing observations, here 1990 census districts in Olinda
- In data frames, the columns can be of different classes — in a matrix, all the columns have to have the same class
- The columns are accessed using the \$ operator; the columns are list elements of the data frame
- We can also use square brackets to access data frame values, like look-up in a data base

Olinda data

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

'data.frame': 243 obs. of 10 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
```

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.

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

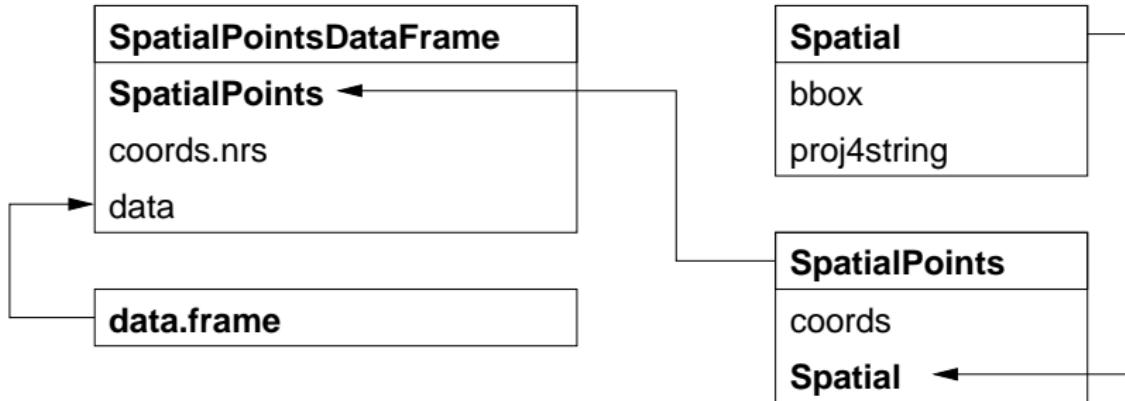
Coordinate reference systems

- 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

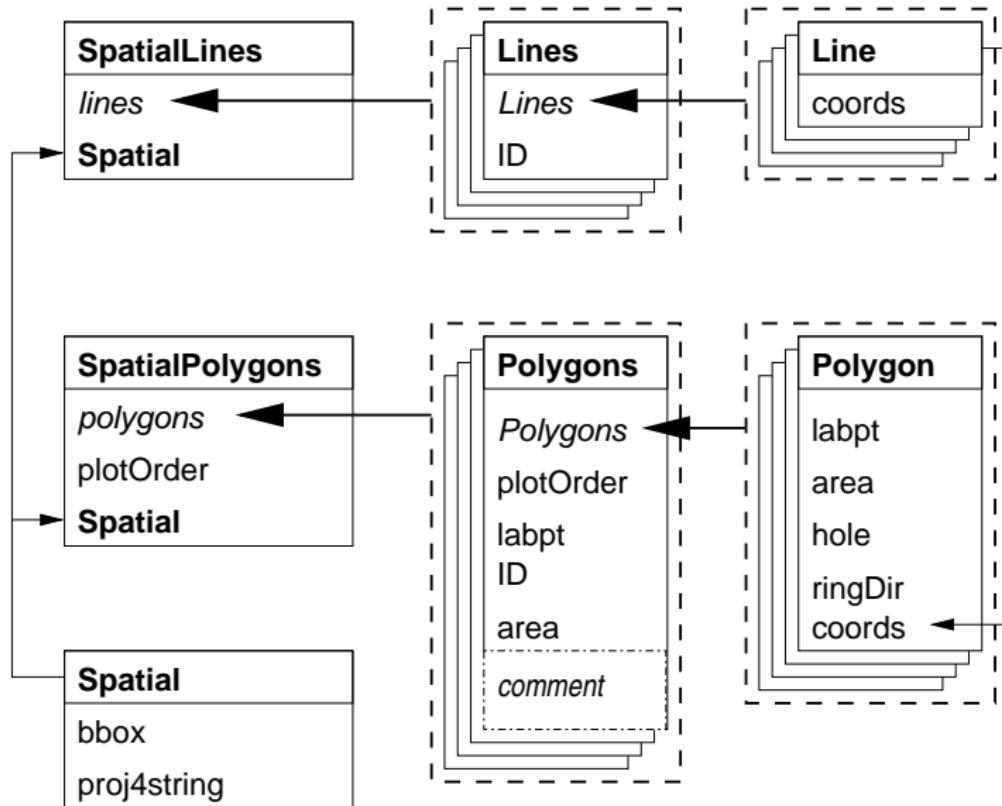
Spatial*DataFrames are constructed as Janus-like objects, looking to mapping and GIS people as “their” kind of object, as a collection of geometric features, with data associated with each feature. But to data analysts, the object “is” a data frame, because it behaves like one.



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 ...
```

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:

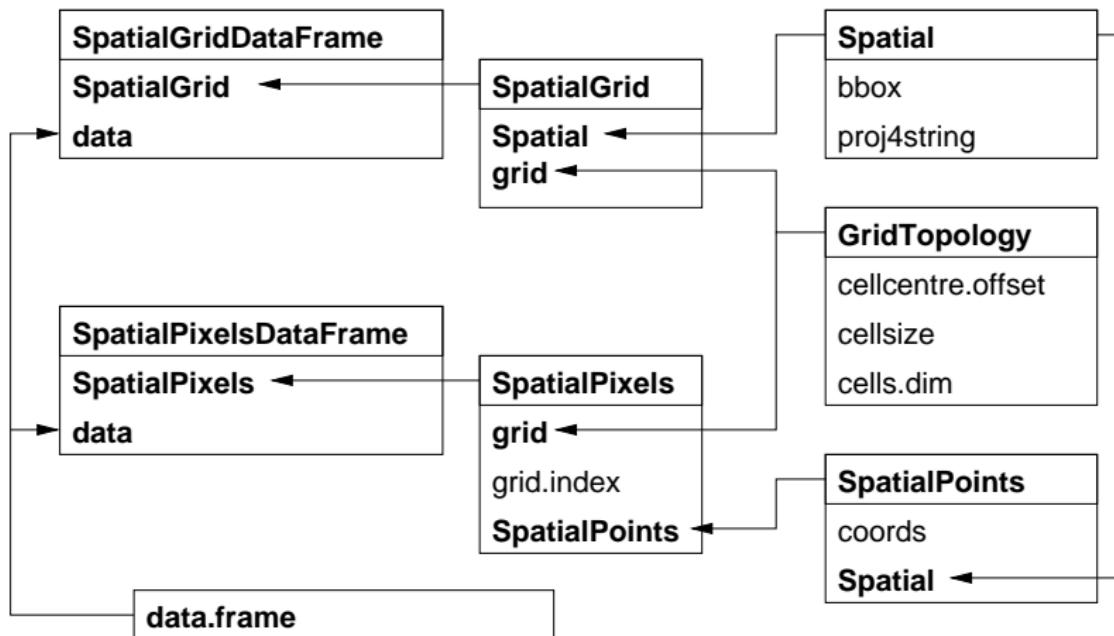
```
> library(sp)
> getClass("GridTopology")
Class "GridTopology" [package "sp"]

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

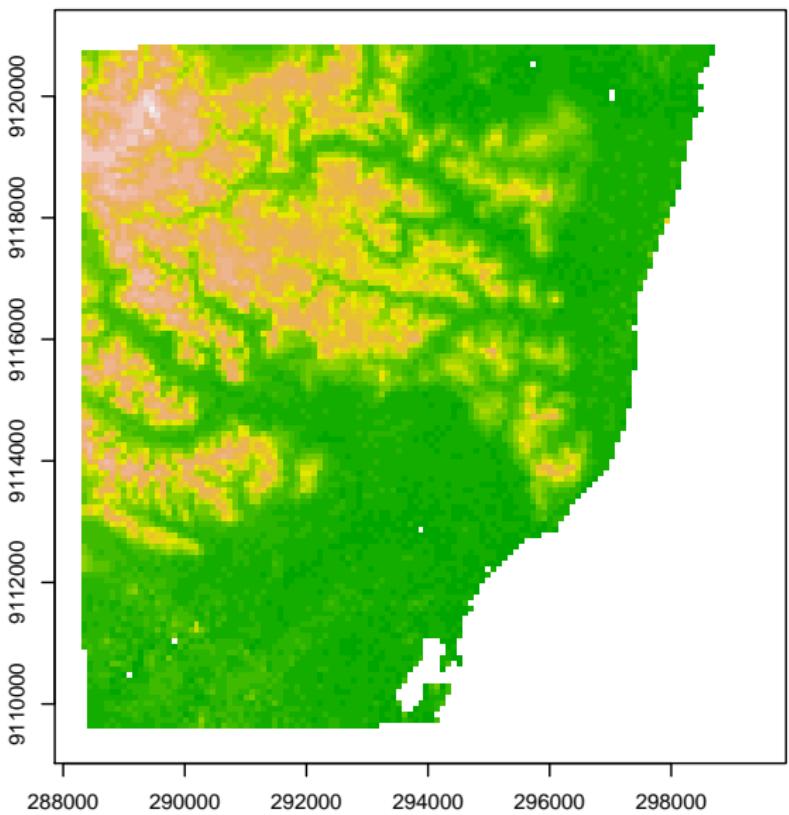
> summary(DEM)

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



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
2741 5536      # Corrego Alegre 1961 / UTM zone 21S
2742 5537      # Corrego Alegre 1961 / UTM zone 22S
2743 5538      # Corrego Alegre 1961 / UTM zone 23S
2744 5539      # Corrego Alegre 1961 / UTM zone 24S
3140 22521 # Corrego Alegre 1970-72 / UTM zone 21S
3141 22522 # Corrego Alegre 1970-72 / UTM zone 22S
3142 22523 # Corrego Alegre 1970-72 / UTM zone 23S
3143 22524 # Corrego Alegre 1970-72 / UTM zone 24S
3144 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"

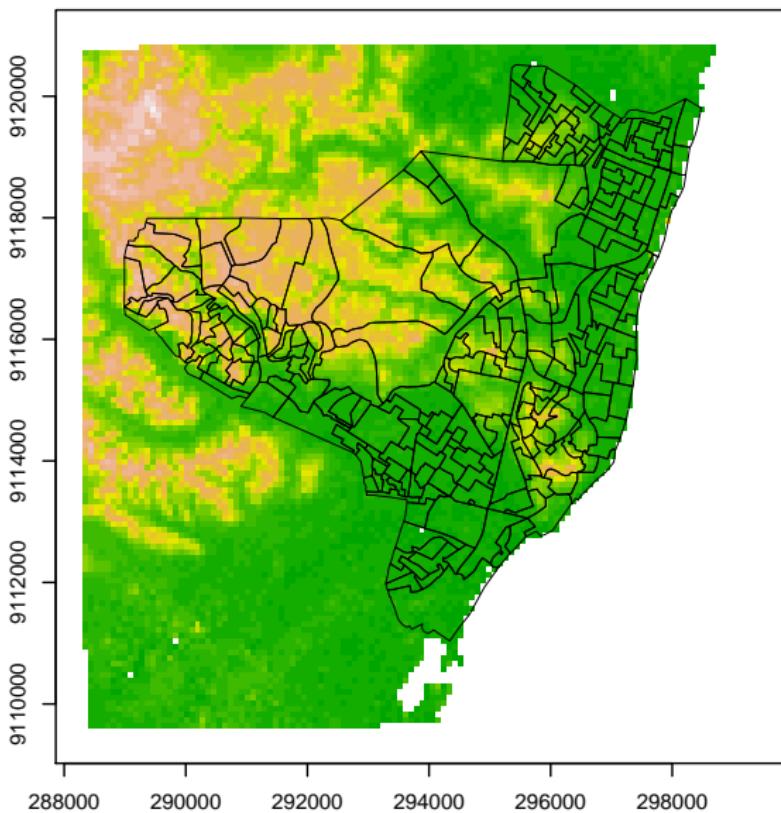
> set_ReplCRS_warn(TRUE)
[1] TRUE
```

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



Reading rasters

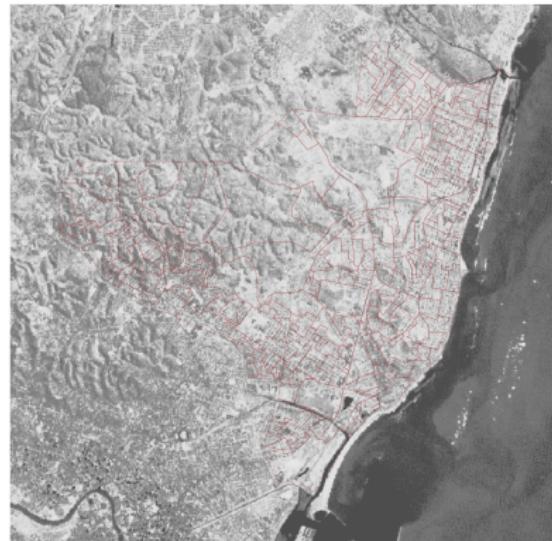
- There are very many raster and image formats; some allow only one band of data, others think data bands are RGB, while yet others are flexible
- There is a simple `readAsciiGrid` function in **maptools** that reads ESRI Arc ASCII grids into `SpatialGridDataFrame` objects; it does not handle CRS and 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")
```

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)
```



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

Writing objects

- In **rgdal**, `writeGDAL` can write for example multi-band GeoTiffs, but there are fewer write than read drivers; in general CRS and georeferencing 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)
```

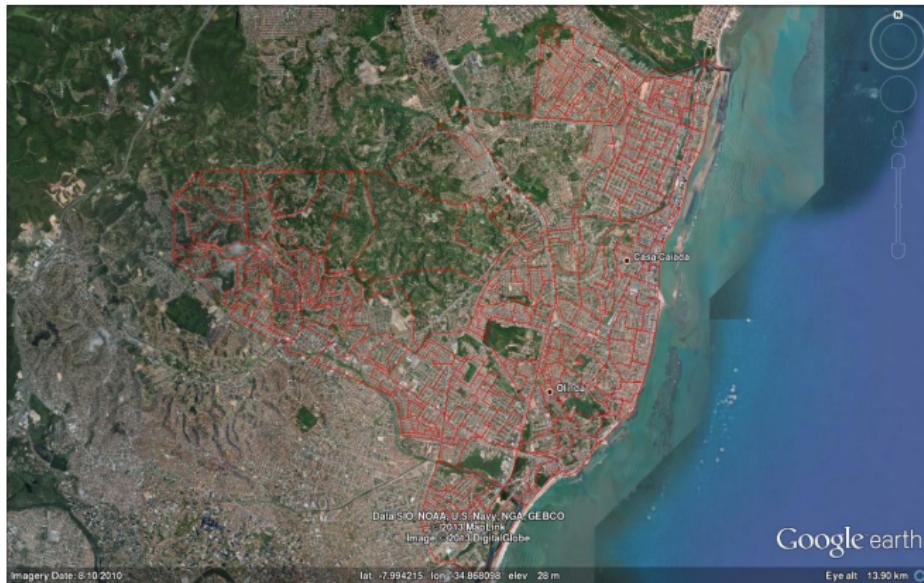
Spatial data in R

Handling spatial data in R: GRASS interface
Handling spatial data in R: methods
Worked examples in spatial statistics

Study area

Representing spatial data
Coordinate reference systems
Raster data
Exporting data

Olinda census district boundaries in GE



GIS interfaces

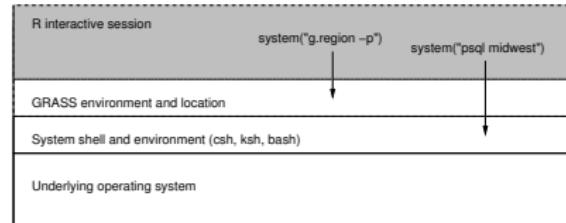
- Because GIS can be used as databases, and their tools can be better suited to some analyses and operations, it may be sensible to use one in addition to data analysis software
- There is an extra effort required when using linked software systems, because they may not integrate easily
- Since R is 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 seems 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

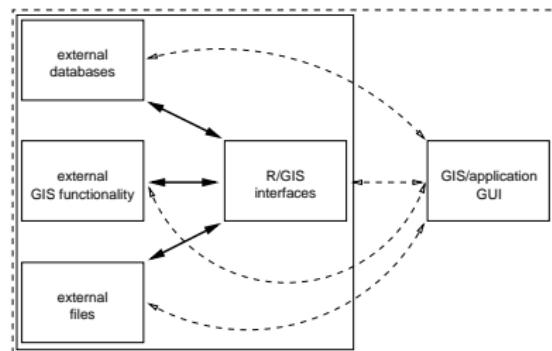
Layering of shells

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 preferred 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-3.1.0/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/g643/grass-6.4.3"
> 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

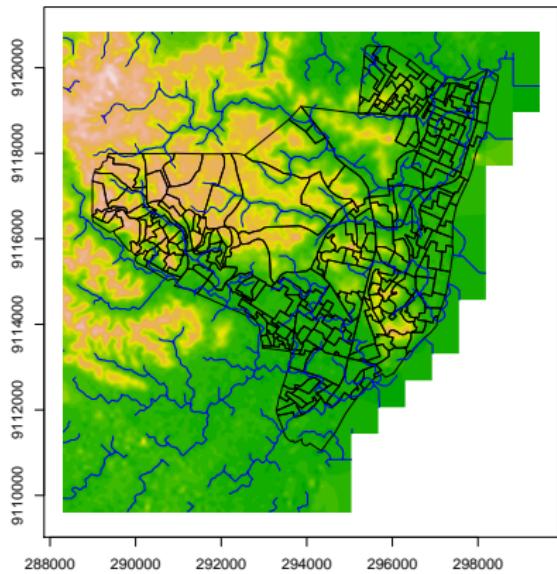
This is where the work gets done, with GRASS command `r.watershed` analysing the resampled DEM to detect depressions through which streams may flow; it can also output the boundaries of watersheds, but these are not needed here. The output is in raster form, and may also be thinned:

```
> 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")
```



Using Jasiewicz/Metz r.stream.extract

Using the same resampled input raster

DEM, we can also use the

r.stream.extract GRASS add-on.

Add-ons are not distributed with GRASS itself, but can be compiled and installed for if GRASS is installed on the system, and the same build tools. We will be using this computed stream line version here:

```
> execGRASS("r.stream.extract",
+   elevation = "DEM_resamp",
+   stream_vect = "v_stream",
+   threshold = 1000)
> stream1 <- readVECT6("v_stream")
```

Operating on geometries

```
> library(rgeos)
> getScale()
[1] 1e+08
> pols <- as(olinda1, "SpatialPolygons")
> Area <- gArea(pols, byid = TRUE)
> olinda1$area <- sapply(slot(olinda1,
+     "polygons"), slot, "area")
> all.equal(unname(Area), olinda1$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.

Intersections - channels and tracts

```
> lns <- as(stream1, "SpatialLines")
> system.time(GI <- gIntersects(lns,
+     pols, byid = TRUE))

  user  system elapsed
0.031   0.000   0.031

> unname(which(GI[2, ]))

[1] 50

> res <- numeric(length = nrow(GI))
> system.time(for (i in seq(along = res)) {
+   if (any(GI[i, ])) {
+     resi <- gIntersection(lns[GI[i,
+       ], pols[i]]
+     res[i] <- gLength(resi)
+   }
+ })
  user  system elapsed
0.305   0.000   0.307

> 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

```
> res0 <- numeric(length = length(pols))
> length(res0)

[1] 243

> system.time(lnsInt <- gIntersection(lns,
+   pols, byid = c(FALSE, TRUE)))
    user  system elapsed
  5.261   0.000  5.274

> length(lnsInt)

[1] 142

> res0[as.integer(row.names(lnsInt)) +
+   1] <- gLength(lnsInt, byid = TRUE)
> all.equal(res, res0)

[1] TRUE
```

We can see that using `gIntersection` directly gives the same sums of lengths, but at a greater cost in time, because each tract is intersected with all streams in turn, not just the ones that do intersect each other.

Intersections - channels and tracts

```
> system.time(tree <- gBinarySTRtreeQuery(lns,
+      pols))

  user  system elapsed
0.007  0.000  0.007

> tree[[2]]
[1] 46 47 48 50

> res1 <- numeric(length = length(tree))
> system.time(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))
+   }
+ })
[1] TRUE

  user  system elapsed
0.465  0.000  0.467

> 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, and tree query does make a big difference for big data sets. Here, the cost of taking intersections for tracts and streams with intersecting envelopes but which do not intersect makes this approach slower than using gIntersects.

Intersections - buffered channels and tracts

```
> buf50m <- gBuffer(lns, width = 50)
> GI1 <- gIntersects(buf50m, pols,
+   byid = TRUE)
> res <- numeric(length = nrow(GI1))
> system.time(for (i in seq(along = res)) {
+   if (any(GI1[i, ])) {
+     out <- gIntersection(buf50m[GI1[i,
+       ]], pols[i])
+     res[i] <- gArea(out)
+   }
+ })
```

user	system	elapsed
2.999	0.000	3.008

```
> 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.

Intersections - buffered channels and tracts

```
> system.time(bufInt <- gIntersection(buf50m,
+   pols, byid = c(FALSE, TRUE)))
    user  system elapsed
 3.339   0.000   3.347

> length(bufInt)
[1] 166

> res0 <- numeric(length = length(pols))
> res0[as.integer(row.names(bufInt)) +
+   1] <- gArea(bufInt, byid = TRUE)
> all.equal(res, res0)

[1] TRUE
```

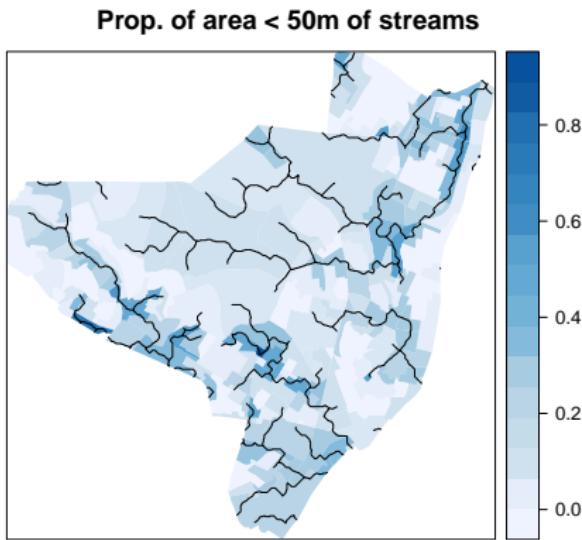
As we can see, because the buffer is defined as a single entity, running `gIntersection` directly takes about the same amount of time as subsetting; we still have to put the shorter output into the correct rows of the polygons object.

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)
```

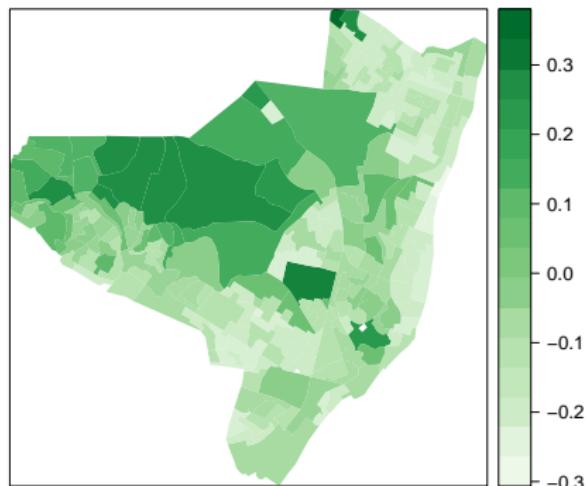


Overlay operations

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 tracts with missing population:

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

Normalized difference vegetation index



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
```

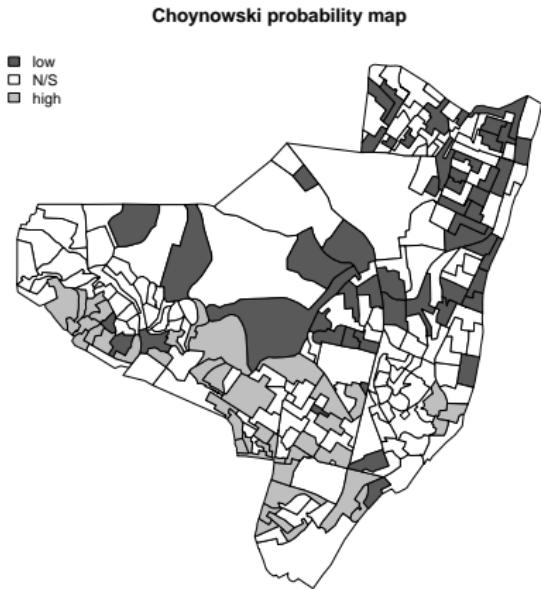
Probability maps and smoothing

- 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

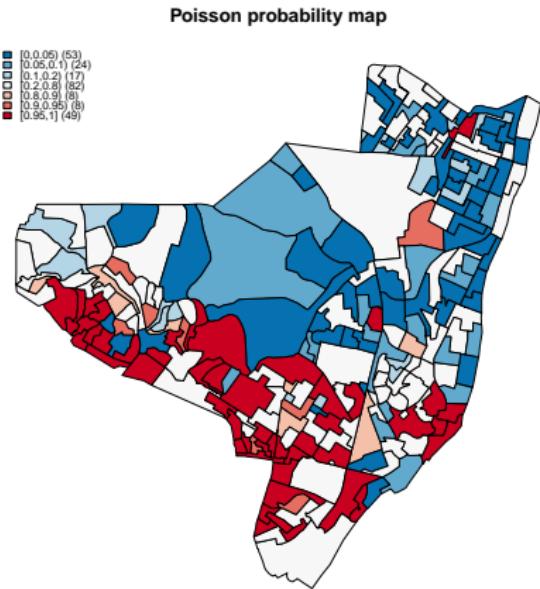
```
> library(spdep)
> ch <- choynowski(olinda2$CASES,
+    olinda2$POP)
```



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
```

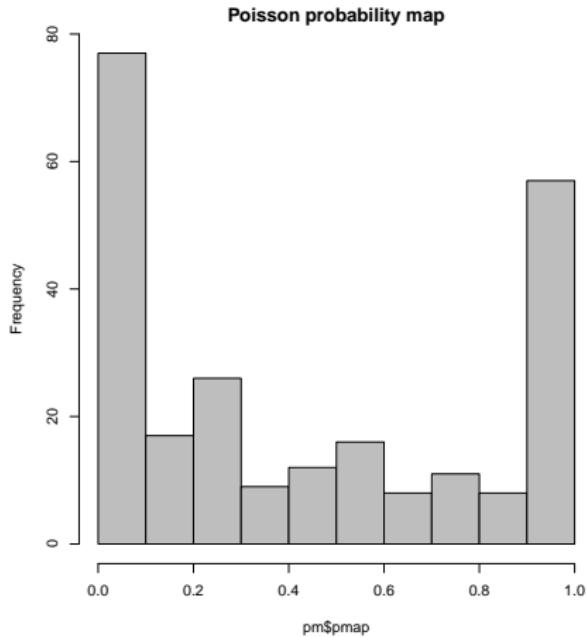


Weaknesses of probability maps

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

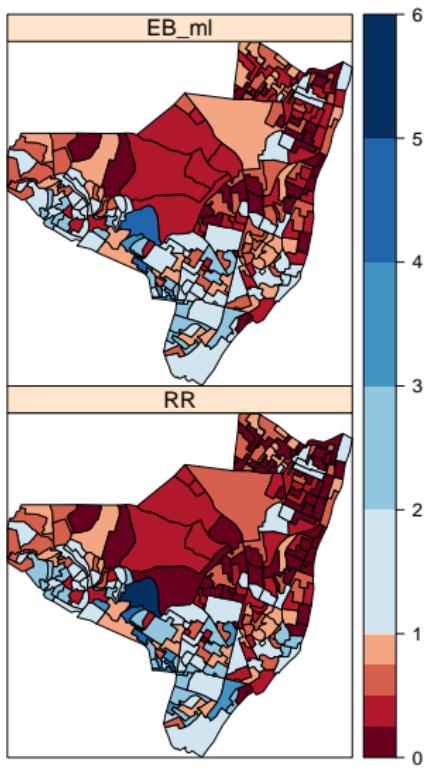


Some Empirical Bayes rates?

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)))
```

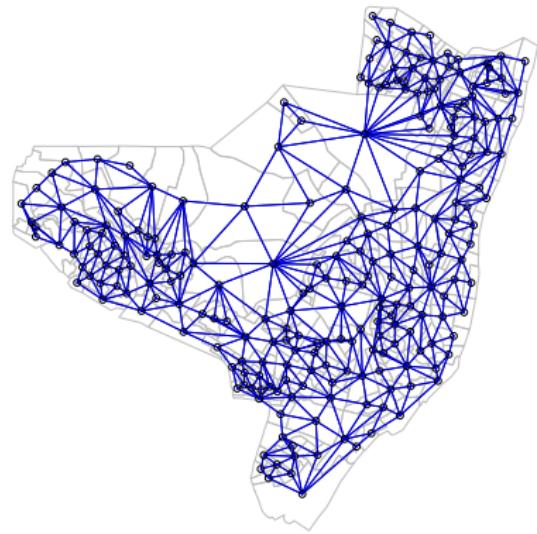


Neighbours

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

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

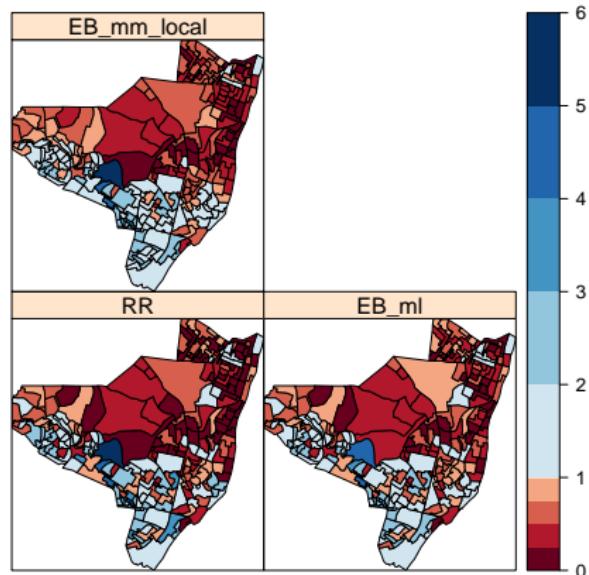
```
Neighbour list object:
Number of regions: 241
Number of nonzero links: 1324
Percentage nonzero weights: 2.279575
Average number of links: 5.493776
```



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:

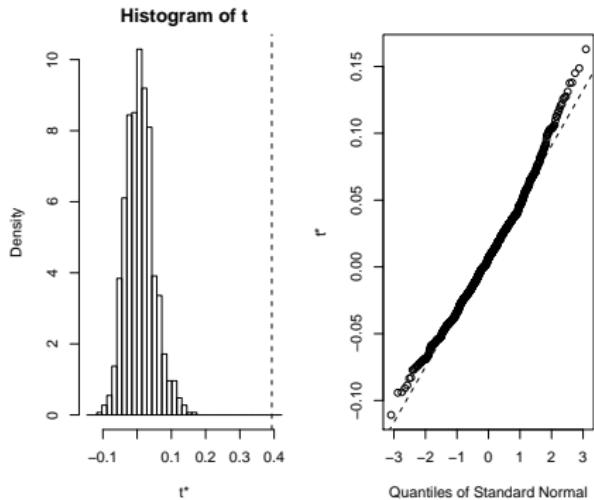
```
> olinda2$Observed <- olinda2$CASES
> eb2 <- EBlocal(olinda2$Observed,
+   olinda2$Expected, nb)
> olinda2$EB_mm_local <- eb2$est
```



Moran's I

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

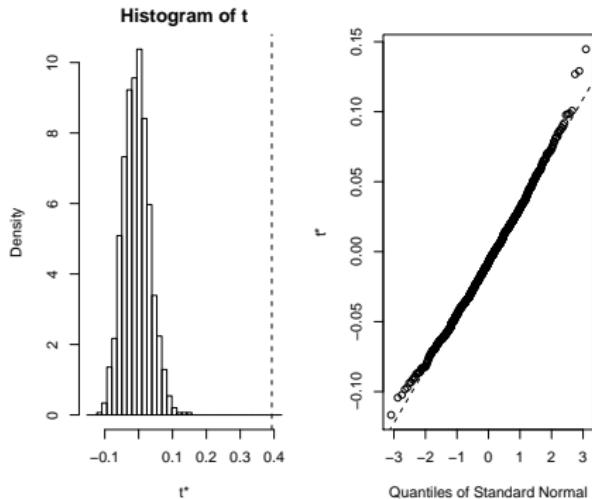
```
> lw <- nb2listw(nb)
> set.seed(130709)
> moran.boot <- boot(as.olinda2,
+   "data.frame"), statistic = moranI.boot,
+   R = 999, listw = lw, n = length(nb),
+   S0 = Szero(lw))
```



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:

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

Dean's P_B test for overdispersion

```
data: mOp
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))

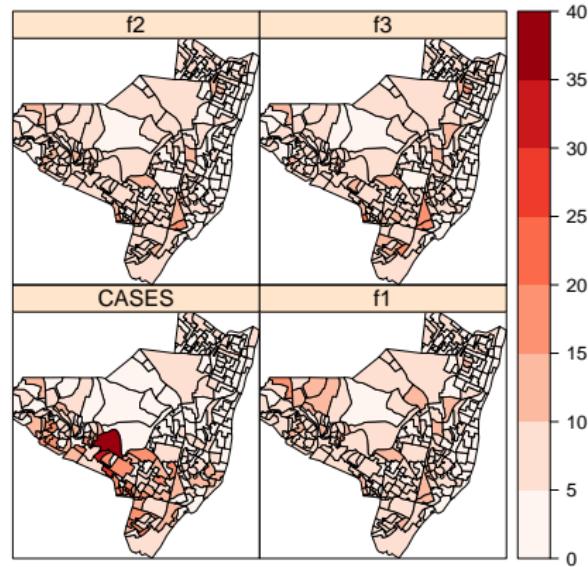
  Resid. Df Resid. Dev Df Deviance      F   Pr(>F)
1       240     1212.5
2       239     1096.6  1   115.871 24.9914 1.12e-06 ***
3       238     1059.4  1    37.183  8.0197 0.005025 **
4       237     1038.2  1    21.281  4.5900 0.033179 *
---
Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 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)

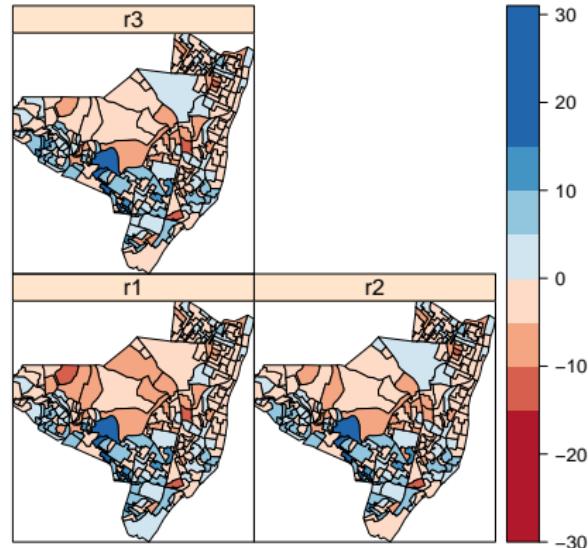
> spplot(olinda2, c("CASES", "f1",
+   "f2", "f3"), col.regions = brewer.pal(8,
+   "Reds"), at = seq(0, 40,
+   5))
```



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))
```



Bayesian spatial regression

- 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 Molliè (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
0.1771           0.7633           0.0490           0.9894

Fixed effects:
            mean      sd 0.025quant 0.5quant 0.975quant mode kld
(Intercept) -0.2314 0.1860    -0.5995 -0.2306    0.1318 -0.2289  0
DEPRIV       0.4183 0.3438    -0.2577  0.4184    1.0934  0.4185  0
ndvi         0.6982 0.5930    -0.4660  0.6977    1.8642  0.6967  0
DEM_resamp   -0.0126 0.0063    -0.0249 -0.0126   -0.0002 -0.0127  0

Random effects:
Name    Model
INLA_ID BYM model

Model hyperparameters:
                           mean      sd 0.025quant 0.5quant 0.975quant mode
Precision for INLA_ID (iid component) 5.4605 1.9042  2.7427   5.1196 10.0987  4.5074
Precision for INLA_ID (spatial component) 1.5733 0.5878  0.6928   1.4862  2.9701  1.3210

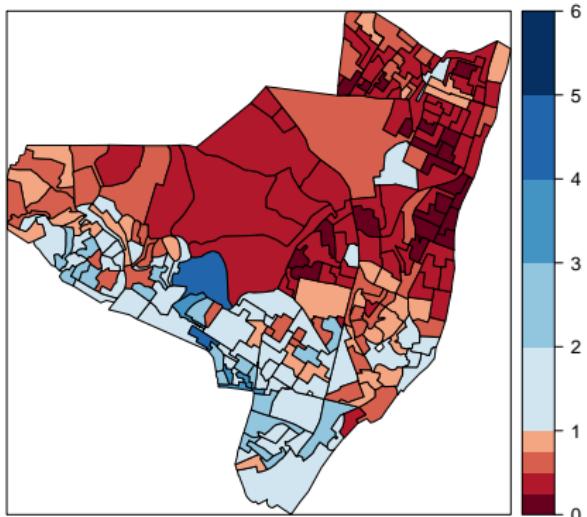
Expected number of effective parameters(std dev): 145.62(7.199)
Number of equivalent replicates : 1.655

Marginal Likelihood: -616.09
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)))
```



Summary

- Using spatial data (in R) permits attention to be given to the important steps of data handling and representation — rushing to analysis always leads to problems
- Handling spatial data is never “easy”, because representational choices lead to support consequences, which qualify or undermine inferential outcomes
- Being close to the data and close to the code may give considerable insight and freedom, and permits co-working with disciplines sharing the same frame of understanding
- Just publishing is not going to be enough; we will need to be able to demonstrate how our conclusions have been reached by releasing code and data (much data is public already).