SISMID Spatial Statistics in Epidemiology and Public Health
2016 R Notes: Introduction

Jon Wakefield
Departments of Statistics and Biostatistics, University of Washington

2016-07-22

R for Spatial Analysis

R has extensive spatial capabilities, the Spatial task view is here

Chris Fowler's R GIS class site is here

Other GIS resources here

Some of the notes that follow are build on Roger Bivand's notes taken from the latter site, and these are based on Bivand et al (2013), which is the reference book!

To get R code alone then load the `knitr` library and then type `purl("SISMID-Introduction.Rmd")` from the directory with this file in.

Representing Spatial Data

Spatial classes[1] were defined to represent and handle spatial data, so that data can be exchanged between different classes.

The `sp` library is the workhorse for representing spatial data.

The most basic spatial object is a 2d or 3d point: a set of coordinates may be used to define a `SpatialPoints` object.

From the help function:

```
SpatialPoints(coords, proj4string=CRS(as.character(NA)),bbox = NULL)
```

- PROJ.4 is a library for performing conversions between cartographic projections.
- The points in a `SpatialPoints` object may be associated with a set of attributes to give a `SpatialPointsDataFrame` object.

[1] Class definitions are objects that contain the formal definition of a class of R objects, usually referred to as an S4 class

Creating a Spatial Object

As an example, the splancs library was pre-sp and so does not use spatial objects.

splancs contains a number of useful functions for analyzing spatial referenced point data.

```
library(sp)
library(splancs)
data(southlancs)  # case control data
summary(southlancs)
##         x                 y                 cc
##  Min.   :346475   Min.   :412437   Min.   :0.00000
##  1st Qu.:353031   1st Qu.:417358   1st Qu.:0.00000
##  Median :355870   Median :421900   Median :0.00000
##  Mean   :355526   Mean   :421518   Mean   :0.05852
##  3rd Qu.:358202   3rd Qu.:425984   3rd Qu.:0.00000
##  Max.   :364435   Max.   :428987   Max.   :1.00000
```

Creating a Spatial Object

We convert into a `SpatialPoints` object and then create a `SpatialPointsDataFrame` data frame.

```
SpPtsObj <- SpatialPoints(southlancs[, c("x", "y")])
summary(SpPtsObj)
## Object of class SpatialPoints
## Coordinates:
##      min    max
## x 346475 364435
## y 412437 428987
## Is projected: NA
## proj4string : [NA]
## Number of points: 974
SpPtsDFObj <- SpatialPointsDataFrame(coords = SpPtsObj,
    data = as.data.frame(southlancs$cc))
```

Spatial Lines and Polygons

A `Line` object is just a collection of 2d coordinates while 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.

`SpatialLines` and `SpatialPolygons` objects are made using lists of `Lines` and `Polygons` objects, respectively.

Spatial Data Frames

`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.

For data on rectangular grids (oriented N-S, E-W) there are two representations: `SpatialPixels` and `SpatialGrid`.

`Spatial*DataFrame` family objects usually behave like data frames, so most data frame techniques work with the spatial versions, e.g. `[]` or \$.

Visualizing Spatial Data

We demonstrate how points and polygons can be plotted on the same graph.

Note that the default is for axes not to be included.

The `meuse` data (in the `sp` library) have been extensively used to illutrate spatial modeling.

```
data(meuse)  # A regular data frame
coords <- SpatialPoints(meuse[, c("x", "y")])
summary(coords)
## Object of class SpatialPoints
## Coordinates:
##      min    max
## x 178605 181390
## y 329714 333611
## Is projected: NA
## proj4string : [NA]
## Number of points: 155
```

Visualizing Spatial Data

```
meuse1 <- SpatialPointsDataFrame(coords, meuse)
data(meuse.riv)
river_polygon <- Polygons(list(Polygon(meuse.riv)),
    ID = "meuse")
rivers <- SpatialPolygons(list(river_polygon))
summary(rivers)
## Object of class SpatialPolygons
## Coordinates:
##        min      max
## x 178304.0 182331.5
## y 325698.5 337684.8
## Is projected: NA
## proj4string : [NA]
```

Visualizing Spatial Data

```
plot(as(meuse1, "Spatial"), axes = T)
plot(meuse1, add = T)
plot(rivers, add = T)
```

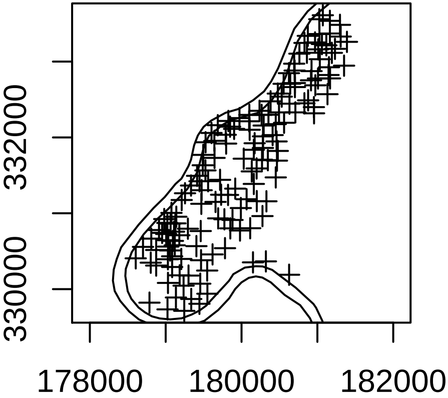


Figure 1: The Meuse river and sampling points

Spatial Pixels and Grids

For data on rectangular grids (oriented N-S, E-W) there are two representations: `SpatialPixels` and `SpatialGrid`.

`SpatialPixels` are like `SpatialPoints` objects, but the coordinates have to be regularly spaced. Coordinates and grid indices are stored.

`SpatialPixelDataFrame` 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.

Visualizaing Spatial Data

Plotting spatial data can be provided in a variety of ways, see Chapter 3 of Bivand et al. (2013).

The most obvious is to use the regular plotting functions, by converting `Spatial` dataframes to regular dataframes, for example using `as.data.frame`.

Trellis graphics (which produce conditional plots) are particularly useful for plotting maps over time.

Visualizing Spatial Data

We construct a `SpatialPixelsDataFrame` object for the Meuse river grid data provided.

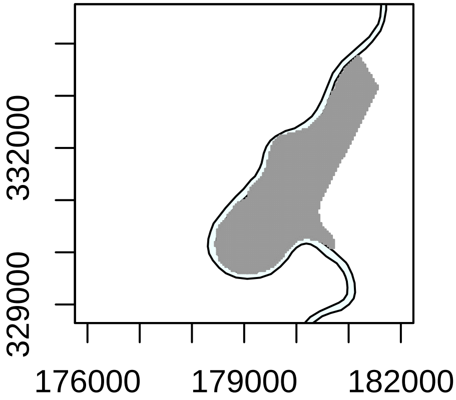
`meuse.grid` is a grid with 40 m x 40 m spacing that covers the Meuse study area

```
data(meuse.grid)
coords <- SpatialPixels(SpatialPoints(meuse.grid[,
    c("x", "y")]))
meuseg1 <- SpatialPixelsDataFrame(coords, meuse.grid)
```

Visualizing Spatial Data

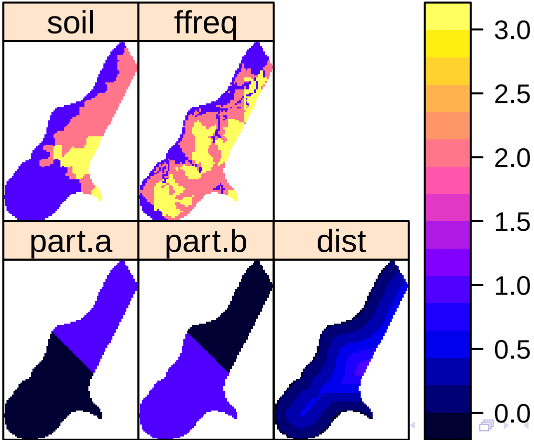
Plotting a grid by the Meuse river.

```
plot(rivers, axes = T, col = "azure1", xlim = c(176000,
    182000), ylim = c(329400, 334000))
box()
plot(meuseg1, add = T, col = "grey60", cex = 0.15)
```



Plotting the variables in meuse.grid

```
data(meuse.grid)
names(meuse.grid)
## [1] "x"        "y"        "part.a" "part.b" "dist"    "soil"    "ffreq"
coordinates(meuse.grid) = ~x + y
proj4string(meuse.grid) <- CRS("+init=epsg:28992")
gridded(meuse.grid) = TRUE
spplot(meuse.grid)
```



Mapping a continous variable

Now we plot a continuous variable, using a particular class inteval style.

The “Fisher-Jenks” style uses the “natural breaks” of class intervals bases on minimizing the within-class variance.

```
library(classInt)
library(RColorBrewer)
pal <- brewer.pal(3, "Blues")
fj5 <- classIntervals(meuse1$zinc, n = 5, style = "fisher")
fj5cols <- findColours(fj5, pal)
```

Mapping a continous variable

```
plot(fj5, pal = pal, main = "")
```

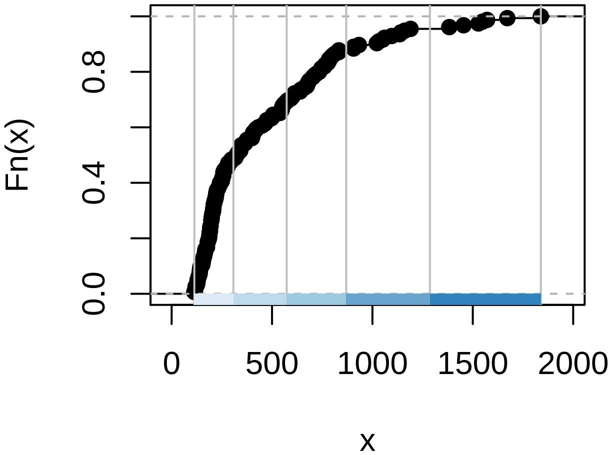


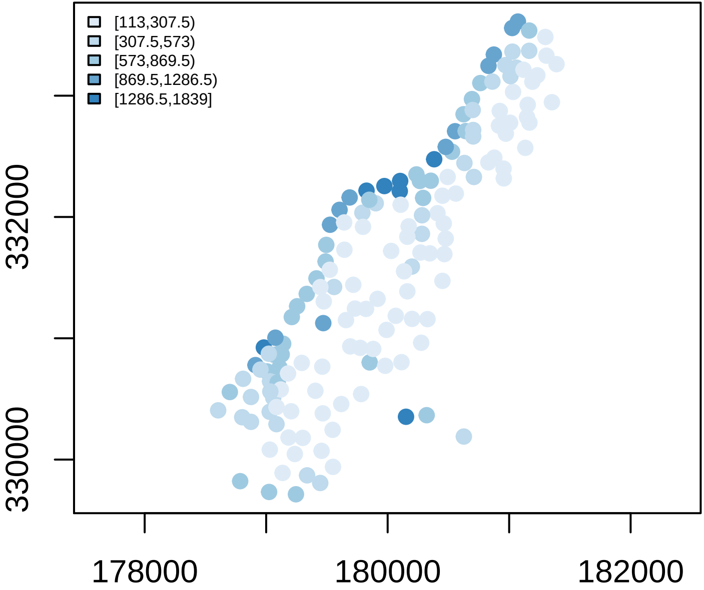
Figure 2: Illustration of Fisher-Jenks natural breaks with five classes, grey vertical lines denote the break points.

Mapping a continous variable

Map the zinc levels in the study region.

```
plot(as(meuse1, "Spatial"), axes = T)
plot(meuse1, col = fj5cols, pch = 19, add = T)
legend("topleft", fill = attr(fj5cols, "palette"),
    legend = names(attr(fj5cols, "table")), bty = "n")
```

Mapping a continous variable

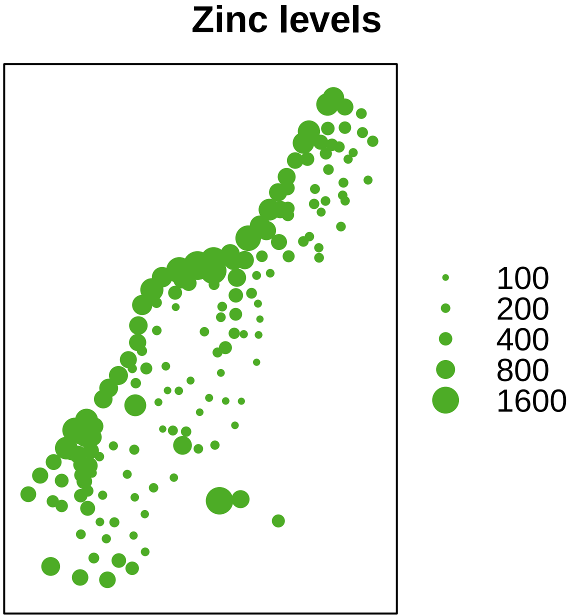


Mapping a continous variable

An alternative display as a “bubble” plot.

```
library(lattice)
bubble(meuse1, zcol = "zinc", main = "Zinc levels",
    maxsize = 0.5, key.entries = 100 * 2^(0:4))
```

Mapping a continous variable



John Snow

For fun, let's look at the poster child of health mapping.

The Snow data consists of the relevant 1854 London streets, the location of 578 deaths from cholera, and the position of 13 water pumps (wells) that can be used to re-create John Snow's map showing deaths from cholera in the area surrounding Broad Street, London in the 1854 outbreak.

The following code was taken from here

```
library(HistData)
data(Snow.deaths)
data(Snow.pumps)
data(Snow.streets)
data(Snow.polygons)
```

John Snow

We first create a SpatialLines object containing the coordinates of the streets using the Lines function

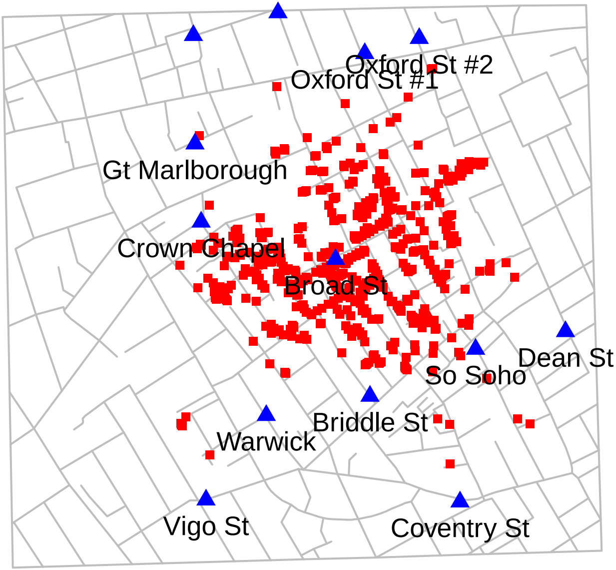
```
# Streets
slist <- split(Snow.streets[, c("x", "y")],
    as.factor(Snow.streets[, "street"]))
Ll1 <- lapply(slist, Line)
Lsl1 <- Lines(Ll1, "Street")
Snow.streets.sp <- SpatialLines(list(Lsl1))
```

John Snow

Display the streets and then add the deaths and pumps (with labels).

```
plot(Snow.streets.sp, col = "gray")
# Deaths
Snow.deaths.sp = SpatialPoints(Snow.deaths[, c("x",
    "y")])
plot(Snow.deaths.sp, add = TRUE, col = "red", pch = 15,
    cex = 0.6)
# Pumps
spp <- SpatialPoints(Snow.pumps[, c("x", "y")])
Snow.pumps.sp <- SpatialPointsDataFrame(spp, Snow.pumps[,
    c("x", "y")])
plot(Snow.pumps.sp, add = TRUE, col = "blue", pch = 17,
    cex = 1)
text(Snow.pumps[, c("x", "y")], labels = Snow.pumps$label,
    pos = 1, cex = 0.8)
```

John Snow: red squares are deaths, blue triangles are pumps



Reading Shapefiles

ESRI (a company one of whose products is ArcGIS) shapefiles consist of three files, and this is a common form.

The first file (*.shp) contains the geography of each shape.

The second file (*.shx) is an index file which contains record offsets.

The third file (*.dbf) contains feature attributes with one record per feature.

Reading Shapefiles

The Washington state Geospatial Data Archive here contains data that can be read into R.

As an example, consider Washington county data that was downloaded from wagda.

The data consists of the three files: wacounty.shp, wacounty.shx, wacounty.dbf.

The following code reads in these data and then draws a county level map of 1990 populations, and a map with centroids.

Reading Shapefiles

First load the libraries.

```
library(maps)
library(shapefiles)
library(maptools)
```

Reading Shapefiles

```
# The following is useful to see if you have the
# versions you want
sessionInfo()
## R version 3.2.3 (2015-12-10)
## Platform: x86_64-apple-darwin13.4.0 (64-bit)
## Running under: OS X 10.10.5 (Yosemite)
##
## locale:
## [1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets  methods   base
##
## other attached packages:
##  [1] maptools_0.8-39    shapefiles_0.7     foreign_0.8-66
##  [4] maps_2.3-10        HistData_0.7-5     lattice_0.20-33
##  [7] RColorBrewer_1.1-2 classInt_0.1-22    splancs_2.01-37
## [10] sp_1.1-1           knitr_1.10.5
##
## loaded via a namespace (and not attached):
##  [1] codetools_0.2-14 class_7.3-14     digest_0.6.9     grid_3.2.3
##  [5] formatR_1.2      magrittr_1.5     e1071_1.6-4      evaluate_0.7
##  [9] stringi_1.0-1    rmarkdown_0.7    rgdal_1.0-4      tools_3.2.3
## [13] stringr_1.0.0    yaml_2.1.13      htmltools_0.2.6
```

Reading Shapefiles

Note that there are problems with the files, which are sorted by using the `repair=T` argument.

The data can be saved from here: here

```
wacounty <- readShapePoly(fn = "examples/wacounty",
    proj4string = CRS("+proj=longlat"), repair = T)
# enter wacounty.shp
names(wacounty)
## [1] "AreaName" "AreaKey"  "INTPTLAT" "INTPTLNG" "TotPop90" "CNTY"
# Let's see what these variables look like
wacounty$AreaName[1:3]  # county names
## [1] WA, Adams County  WA, Asotin County WA, Benton County
## 39 Levels: WA, Adams County WA, Asotin County ... WA, Yakima County
wacounty$AreaKey[1:3]  # FIPS codes
## [1] 53001 53003 53005
## 39 Levels: 53001 53003 53005 53007 53009 53011 53013 53015 53017 ... 53077
```

Drawing a map

We look at some variables.

```
wacounty$INTPTLAT[1:3]  # latitude
## [1] 46.98899 46.18248 46.24764
wacounty$INTPTLNG[1:3]  # longitude
## [1] -118.5569 -117.1850 -119.5015
wacounty$CNTY[1:3]
## [1] 1 3 5
## 39 Levels: 1 11 13 15 17 19 21 23 25 27 29 3 31 33 35 37 39 4
wacounty$TotPop90[1:3]
## [1]  13603  17605 112560
```

Drawing a map

We look at some variables, and then set up the colors to map.

```
plotvar <- wacounty$TotPop90  # variable we want to map
nclr <- 8  # next few lines set up the color scheme for plotting
plotclr <- brewer.pal(nclr, "BuPu")
brks <- round(quantile(plotvar, probs = seq(0,
    1, 1/(nclr))), digits = 1)
colornum <- findInterval(plotvar, brks, all.inside = T)
colcode <- plotclr[colornum]
plot(wacounty)
plot(wacounty, col = colcode, add = T)
legend(-119, 46, legend = leglabs(round(brks,
    digits = 1)), fill = plotclr, cex = 0.4,
    bty = "n")
```

Drawing a map

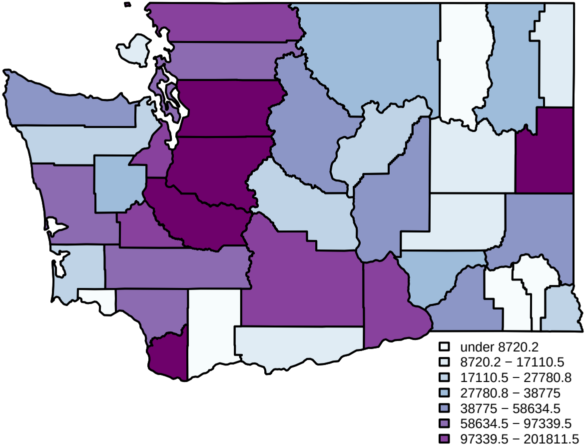


Figure 3: 1990 Washington population counts by census tracts

Drawing a map

As an alternative we can use the `spplot` function, which uses lattice (trellis) plot methods for spatial data with attributes.

```
spplot(wacounty, zcol = "TotPop90")
```

Drawing a map: an alternative way

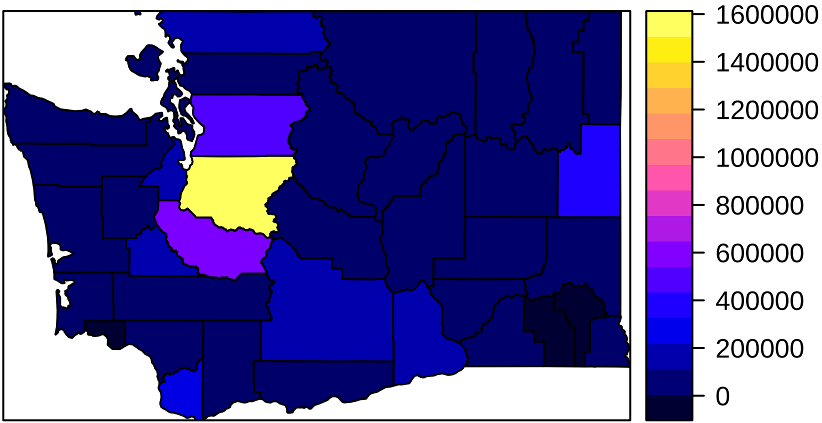


Figure 4: 1990 Washington population counts by county

Drawing a census tract map

We repeat but now map populations at the census tract level.

```
watract <- readShapePoly(fn = "examples/watract1",
    proj4string = CRS("+proj=longlat"), repair = T)  #
names(watract)
## [1] "SP_ID"    "AreaName" "AreaKey"  "INTPTLAT" "INTPTLNG" "TotPop90"
## [7] "TRACT"    "CNTY"
plotvar <- watract$TotPop90  # variable we want to map
brks <- round(quantile(plotvar, probs = seq(0,
    1, 1/(nclr))), digits = 1)
colornum <- findInterval(plotvar, brks, all.inside = T)
colcode <- plotclr[colornum]
plot(watract)
plot(watract, col = colcode, add = T)
legend(-119, 46, legend = leglabs(round(brks,
    digits = 1)), fill = plotclr, cex = 0.4,
    bty = "n")
```

Drawing a cenusus tract map

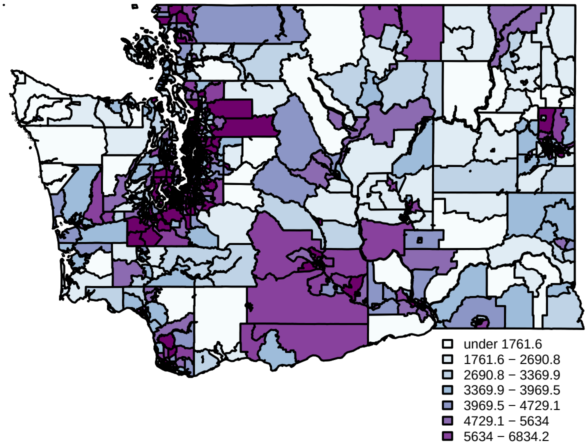
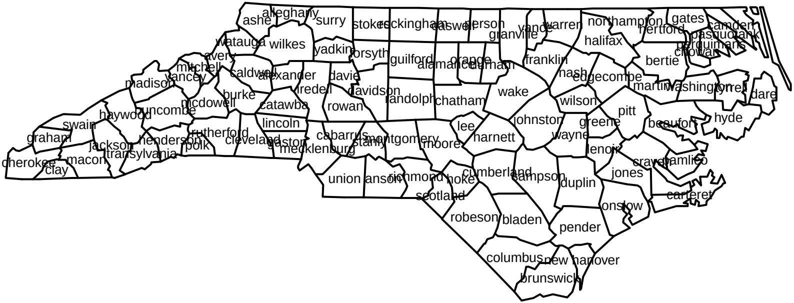


Figure 5: 1990 Washington populations by census tract

A county map of North Carolina with text

```
library(ggplot2)  # For map_data. It's just a wrapper; should just use maps.
library(sp)
library(maps)
getLabelPoint <- # Returns a county-named list of label points
function(county) {Polygon(county[c('long', 'lat')])@labpt}
df <- map_data('county', 'north carolina')  # NC region county data
centroids <- by(df, df$subregion, getLabelPoint)     # Returns list
centroids <- do.call("rbind.data.frame", centroids) # Convert to Data Frame
names(centroids) <- c('long', 'lat') # Appropriate Header
map('county', 'north carolina')
text(centroids$long, centroids$lat, rownames(centroids), offset=0, cex=0.4)
```

A county map of North Carolina with text



Getting fancy

Spatial data can be displayed on on interactive web-maps using the open-source JavaScript library Leaflet.

```
library(maptools)
library(sp)
library(leafletR)
SP <- readShapePoly(system.file("shapes/sids.shp",
    package = "maptools")[1], proj4string = CRS("+proj=long
SP4leaflet <- toGeoJSON(data = SP, dest = tempdir(),
    name = "BIR79")
SPleaflet <- leaflet(data = SP4leaflet, dest = tempdir(),
    title = "Trying to plot BIR79", base.map = "osm",
    popup = "*")
```

Getting fancy

This code produces a html with the births in 1979 in North Carolina plotted over a map.

```
brks <- seq(0, max(SP$BIR79), by = 5000)
clrs <- colorRampPalette(c("blue", "yellow",
    "red"))(7)
stl <- styleGrad(prop = "BIR79", breaks = brks,
    style.val = clrs, out = 1, leg = "BIR79")
SPleaflet <- leaflet(data = SP4leaflet, dest = tempdir(),
    title = "SISMID plot of BIR79", base.map = "osm",
    style = stl, popup = "*")
SPleaflet
```