Introduction to the analysis of spatial data using R

Ana I. Moreno-Monroy

13-16 June 2017

Chapter 2: Spatial data

- Spatial data: definition and types
- Spatial objects and projections
- Merging statistical information into spatial objects
- Additional operations with polygons
- Rethinking spatial analysis: the sf package

Spatial data

- Spatial data is data that contains information about geographical locations and/or about the shapes of the geographical features (geometry)
- Spatial data can be stored as precise (point) coordinates (lat-lon/XY) or topology (i.e., the geometric boundaries of municipalities, countries, etc.)
- A (postal) address or zip-code is not spatial data. It needs to be geolocated in order to obtain point coordinates
- In order to visually represent (i.e. map) an administrative unit, we need information about its geometry (lines/polygons)

Shapefiles

- Polygon boundaries are a collection of lines that form shapes. These are usually contained in **shapefiles**
- The shapefile format is a geospatial vector data format for geographic information system (GIS) software.
- Shapefile folders include at least four files with extensions: .shp (the feature geometry itself); .shx (a positional index of the feature geometry); .dbf (columnar attributes). They can also include a .prj file (with the coordinate system and projection information)

R and GIS

- Spatial data is usually managed in a Geographic Information System (GIS) software (e.g. ArcGIS, QGIS)
- Some packages in R make it possible to handle and analyze spatial data. These include **sp**, **rgdal**, **matpools**, **rgeos** and **spatstat**
- In R it is possible to geolocate addresses, manipulate and visualize spatial data and perform spatial statistical analyses

Spatial data: an example

- Super Neighborhoods division polygon data is freely available at the City of Houston GIS open data website
- We can use the function **readOGR** from the package **rgdal** to import this data

```
library(rgdal)
neigh <- readOGR(dsn = "Super_Neighborhoods", layer = "Super_Neighborhoods")
```

```
## OGR data source with driver: ESRI Shapefile
## Source: "Super_Neighborhoods", layer: "Super_Neighborhoods"
## with 88 features
## It has 11 fields
```

- Other options for importing shapefiles include the readSpatialPoly function of the **maptools** package (without projection data)

Inspection

- Neigh is a SpatialPolygonDataFrame with several **slots**, two of which are @data (containing variables associated to each polygon) and @polygons
- We can work with @data the same way we work with a regular data object

```
str(neigh@data)
```

```
## 'data.frame':    88 obs. of  11 variables:
##  $ OBJECTID  : int  1 2 3 4 5 6 7 8 10 11 ...
##  $ PERIMETER : num  16572 43119 39256 59785 75759 ...
##  $ POLYID    : int  60 63 61 59 21 5 78 73 17 22 ...
##  $ SNBNAME   : Factor w/ 88 levels "ACRES HOME","ADDICKS PARK TEN",..: 26 68 16 14 36 33 32 27 22 82
##  $ cohgisCOHG: int  0 0 0 0 0 0 0 0 0 0 ...
##  $ cohgisCO_1: int  0 0 0 0 0 0 0 0 0 0 ...
##  $ COUNCIL_AC: Factor w/ 2 levels "No","Yes": NA 2 NA 2 NA 2 NA 2 2 2 ...
##  $ RECOGNITIO: Factor w/ 20 levels "2000-03-06T00:00:00.000Z",..: NA 14 NA 5 NA 2 NA 6 20 1 ...
##  $ SnbrInfoUR: Factor w/ 88 levels "http://www.houstontx.gov/superneighborhoods/profiles/SN_1.htm",.
##  $ WeCan     : Factor w/ 1 level "Y": 1 1 NA 1 NA NA NA NA NA NA ...
##  $ Top10     : Factor w/ 1 level "Y": NA 1 NA NA NA NA NA NA NA NA ...
```

NA values

- The **is.na()** function is useful to handle NA values

```
table(is.na(neigh@data$COUNCIL_AC))
```

```
##
## FALSE  TRUE
##    60    28
```

- The **complete.cases** function reports TRUE for cases without any NA values (it can be used to subset as well)

```
table(complete.cases(neigh@data))
```

```
##
## FALSE  TRUE
##    80     8
```

Map projections

- Every spatial data has its own Coordinate System, which can be either a lat-lon system (Geographic Coordinate System) or a projected coordinate system (map projection)
- A map projection is always necessary to create a map. Projections are a way to represent the 3-D surface of the earth on a 2-D plane
- Different spatial operations require either lat-lon or projected coordinate systems. In any case, the systems should be the same when working with super-imposed layers

- Please take a moment to read these Projection Basics

Finding out the Coordinate System

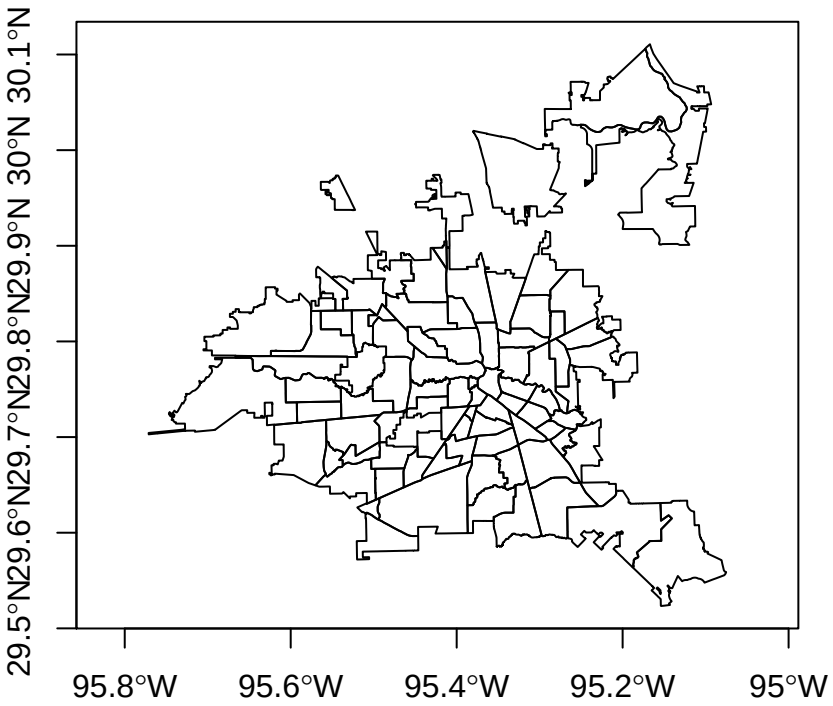
- To find out the CRS of a spatial object, open .prj file in shapefile folder, or in R type

```
proj4string(neigh)
```

```
## [1] "+proj=longlat +datum=WGS84 +no_defs +ellps=WGS84 +towgs84=0,0,0"
```

- A Geographic Coordinate System (Lat/Lon coordinates) does not have a projection. It is represented in degrees, so distance is not measured in usual distance units (meters, kilometers)
- The following map shows the axes values in degrees for the unprojected spatial data

```
plot(neigh, axes=TRUE)
```



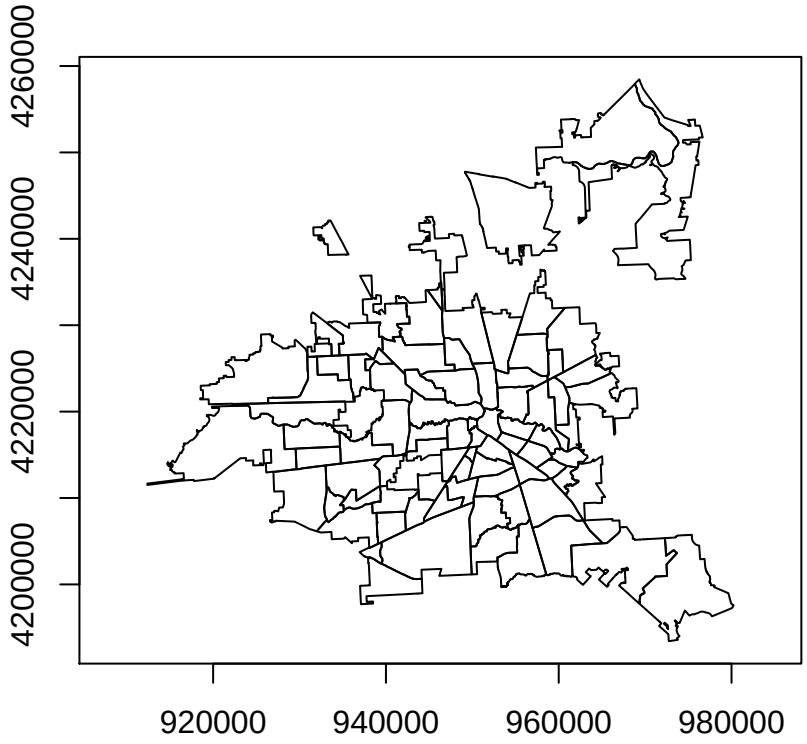
Projecting coordenates

- To find out the equivalent projection to a GCS, see this Spatial Reference website
- Transformations from unprojected to projected systems (and vice-versa) can be made in R using the **spTransform** function of the **sp** package

```
library(sp)
neigh <- spTransform(neigh, CRS("+init=epsg:3673"))
```

- The axes of the map are now in a different unit

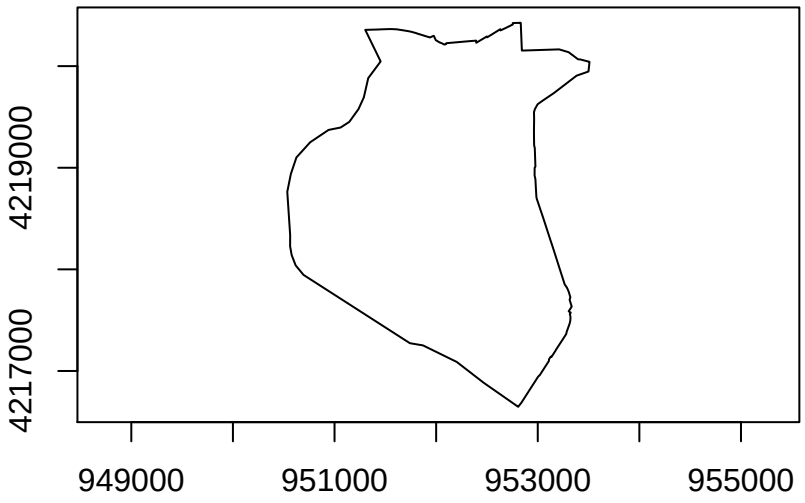
```
plot(neigh, axes=TRUE)
```



Selecting a single area

Suppose we are only interested in the area DOWNTOWN. We can construct a TRUE/FALSE variable to subset the spatial object

```
sel <- neigh@data$SNBNAME=="DOWNTOWN"
downtown<-neigh[sel, ]
plot(downtown, axes=TRUE)
```



Merging external datasets

- More information by Super Nieghborhood from the 2010 US Census is available at the City of Houston GIS open data website in .csv format
- We can import this csv into R using the **read.csv** function (see ?read.csv)

```
SN_data<-read.csv("Census_2010_By_SuperNeighborhood.csv", header=TRUE)
```

- Alternatively, we can download and import the data directly using **read.csv2**

```
SN_data<-read.csv2("http://arcg.is/2dZ7naE", sep=",")
```

- The variable POLYID in SN_data is also in neigh@data. It is a unique identifier of the spatial units (Super Neighborhoods)
- We can merge the SN_data into the neigh SpatialPointsDataFrame using this unique identifier

```
neigh<-merge(neigh, SN_data, by="POLYID")
```

- We can subset to keep the variables we are interested in

```
keep<-c("POLYID", "SNBNAME", "SUM_TotPop")
neigh1<-neigh[,colnames(neigh@data) %in% keep]
```

- The new shapefile can be exported using the **writeOGR** function

Handling merged data

- We can make basic queries on the merged data the same way we would with a regular data object. For instance, we could find out how many people in all neighborhoods, or how many people live Downtown

```
sum(neigh@data$SUM_TotPop)
```

```
## [1] 2066749
```

```
neigh@data$SUM_TotPop[neigh@data$SNBNAME=="DOWNTOWN"]
```

```
## [1] 16716
```

- To add more criteria (e.g., more neighborhoods) use & for AND and | for OR

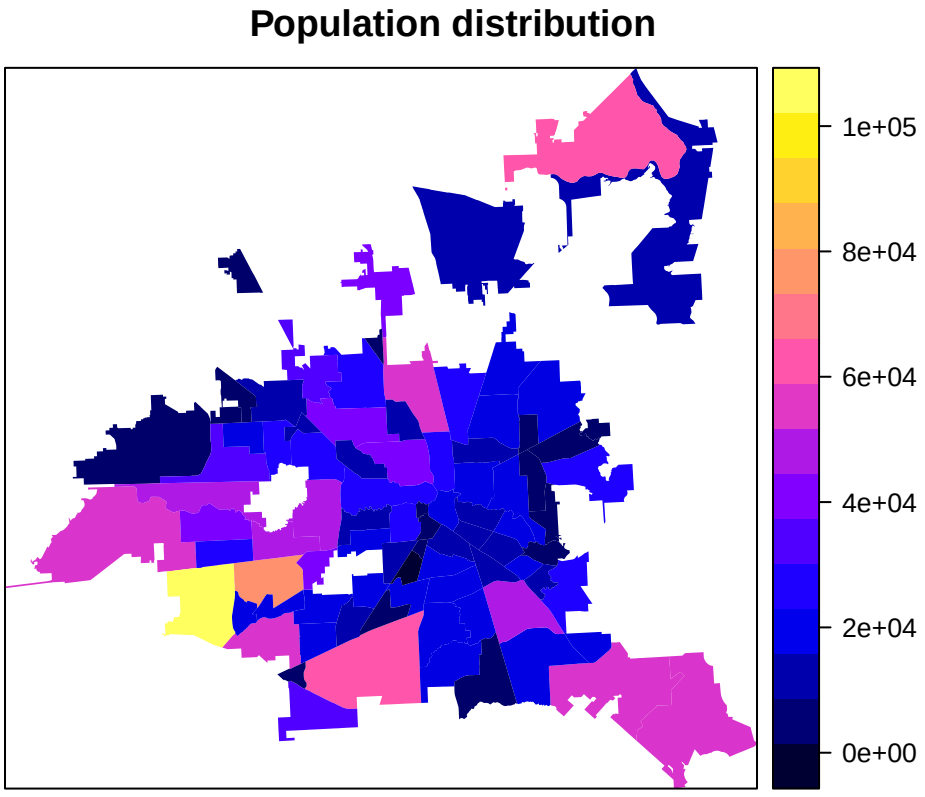
```
neigh@data$SUM_TotPop[neigh@data$SNBNAME=="DOWNTOWN" | neigh@data$SNBNAME=="MEMORIAL"]
```

```
## [1] 16716 45294
```

Simple cloropeth map

We can use the **spplot** to get a simple cloropeth map of any continuos variable

```
spplot(neigh, "SUM_TotPop", main = "Population distribution", col = "transparent")
```

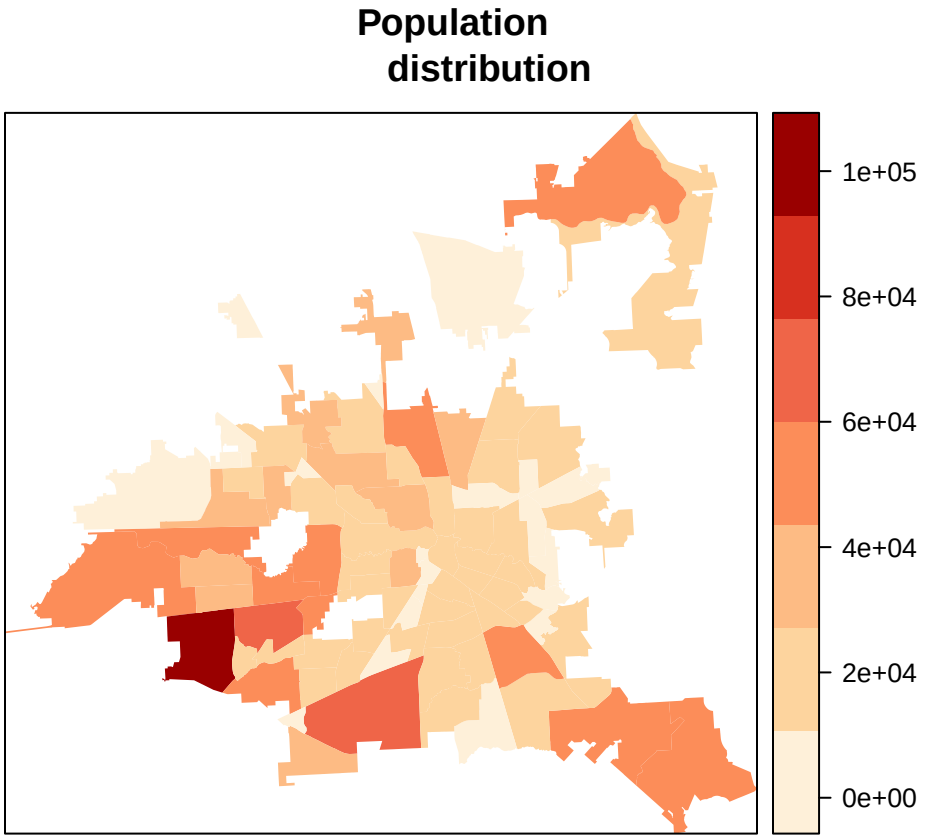


Cloropeth map: more options

- We can manipulate all the attributes of the cloropeth map by modifying different options. Unfortunately, there is no friendly Map editor (as in QGIS or ArcMap) for these tasks

- To change the colors, we have to first choose a palette from the RColorBrewer package options (type display.brewer.all() to see the options) and then define the number of cuts. In this case the palette is "Orange Red" and the number of cuts 7 (which becomes 6 when passed on the **spplot** function)

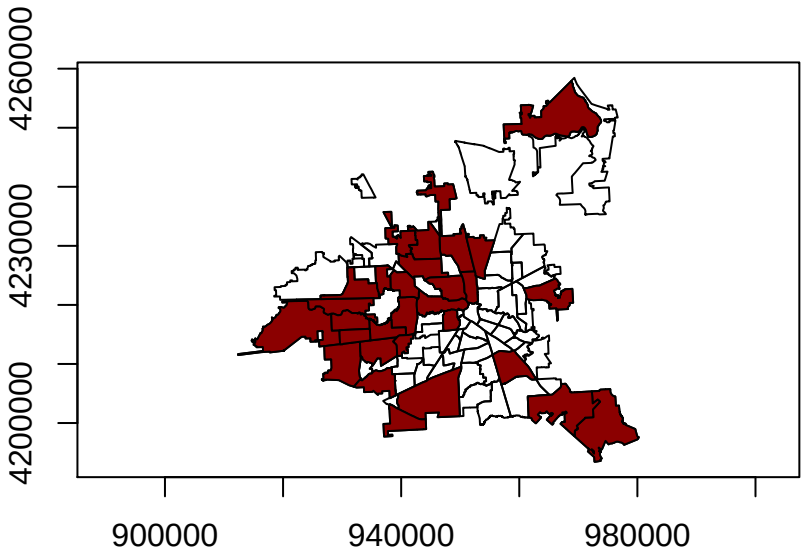
```
library(RColorBrewer)
my.palette <- brewer.pal(n = 7, name = "OrRd")
spplot(neigh, "SUM_TotPop", col.regions = my.palette, main = "Population
       distribution", cuts = 6, col = "transparent")
```



Identify specific areas in a plot

We can subset polygons in the same way we subset data based on a condition

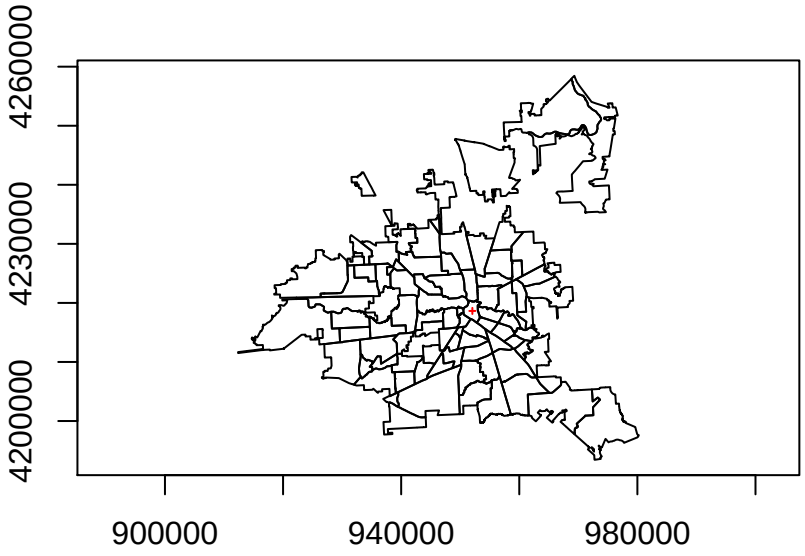
```
plot(neigh, axes=TRUE)
sel <- neigh@data$SUM_TotPop>mean(neigh@data$SUM_TotPop)
plot(neigh[sel, ], axes=TRUE, col="dark red", add=TRUE)
```



Find centroid of an area

- The function **gCentroid** of the **rgeos** package allows us to find the geometric center (centroid) of a map or a single polygon

```
library(rgeos)
plot(neigh, axes = TRUE)
center <- gCentroid(neigh[neigh$SNBNAME == "DOWNTOWN",])
plot(center, cex = 0.3, col="red", add=TRUE)
```



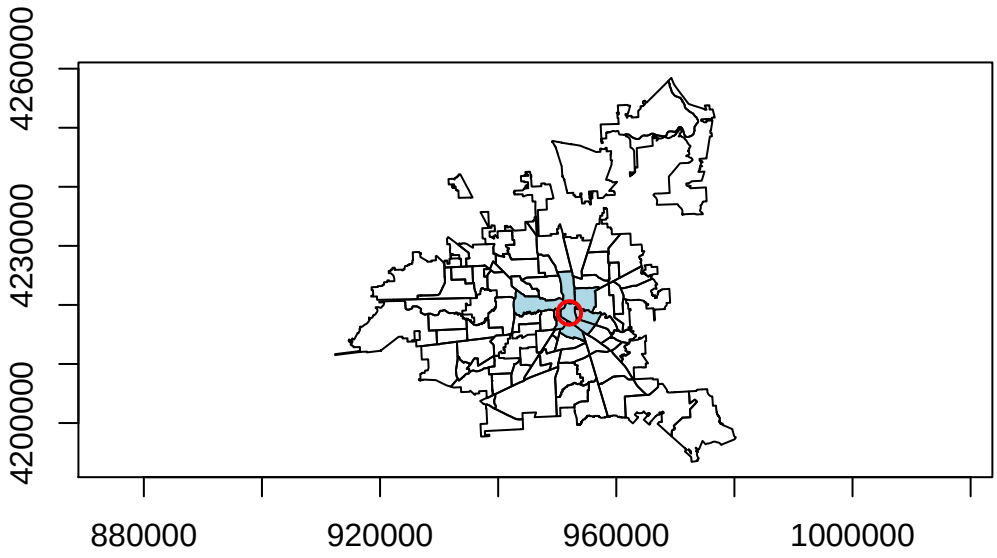
- Use the option "byid=TRUE" to obtain the centroids of each polygon

```
center_all <- gCentroid(neigh, byid=TRUE)
```

Select areas touching a buffer

Buffers around a point are often used to define areas of interest. The function **gBuffer** of the **rgeos** package allows us to calculate buffers of different width (in meters) around a given spatial geometry

```
plot(neigh, axes=T)
buffer_2k <- gBuffer(center, width = 2000)
plot(neigh[buffer_2k,], col = "lightblue", add = T)
plot(buffer_2k, add = T, border = "red", lwd = 2)
```



Calculating a distance matrix between centroids

We can calculate the euclidian distance between every pair of centroids (as an approximation for the distance between areas) using the **distm** function of the **geosphere** package. As it requires points in degrees (lon/lat), we have to transform back our spatial object to the original CRS

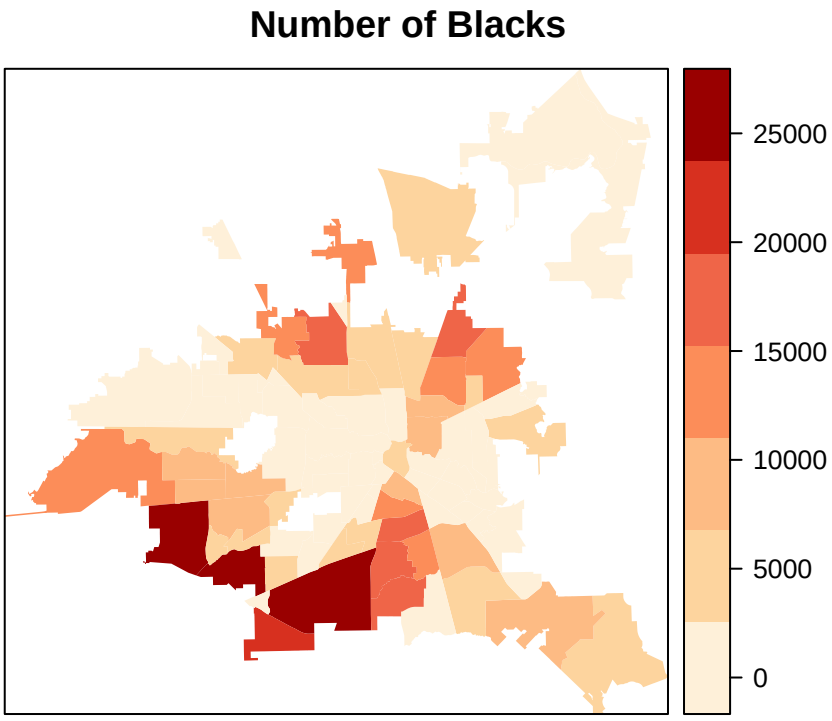
```
library(geosphere)
neigh <- spTransform(neigh, CRS("+proj=longlat +datum=WGS84 +no_defs +ellps=WGS84 +towgs84=0,0,0"))
center_all <- gCentroid(neigh, byid=TRUE)
dist_matrix <- distm(center_all)
dist_matrix[1:5,1:5]
```

```
##           [,1]      [,2]      [,3]      [,4]     [,5]
## [1,]     0.000  5151.180  2094.728 11450.322  9166.35
## [2,]  5151.180     0.000  3077.924  6310.274 14285.42
## [3,]  2094.728  3077.924     0.000  9360.479 11260.36
## [4,] 11450.322  6310.274  9360.479     0.000 20595.69
## [5,]  9166.350 14285.421 11260.359 20595.694     0.00
```

Testing for spatial autocorrelation

Variables may have a "spatial" pattern, meaning that they seem to be more clustered or dispersed than would be expected by random location. To check for this, we first plot the spatial distribution of the variable of interest (in this case Black population)

```
spplot(neigh, "SUM_NH_Black", col.regions = my.palette, main = "Number of Blacks",
       cuts = 6, col = "transparent")
```



Constructing a neighbors' list

- To find out the extent of spatial autocorrelation in our variable, we need to define some criteria for what we consider "being a neighbor". The first step in this direction is to construct a neighbors list using the **poly2nb** function of the **spdep** package.

```
library(spdep)
neigh_nb <- poly2nb(neigh)
head(neigh_nb, n=3)
```

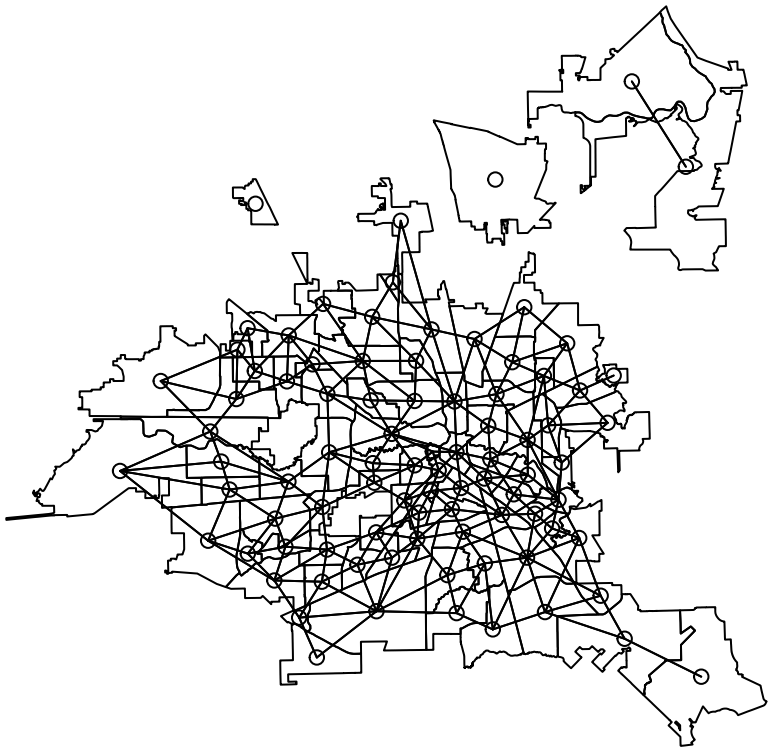
```
## [[1]]
## [1]  3 10 78 85
##
## [[2]]
## [1]  3 11 12 73 77 82
##
## [[3]]
## [1]  1  2 10 11 19 75 77 78
```

- Here the default contiguity condition is "Queen", meaning that areas sharing any boundary point are considered neighbors
- The object neigh_nb contains a list with the ids of neighbors of each polygon

Plot neighbors

We can visualize the nb object in a plot, together with the coordinates of the centroids of each polygon

```
coords<-coordinates(neigh)
plot(neigh)
plot(neigh_nb, coords, add=T)
```



Construct weights between neighbors

The next step is to construct weights between neighbors. There are many coding schemes that can be chosen to represent the spatial relationship between areas. In this simple example, we assume row-standardized weights between neighbors, so that areas with more neighbors weight more than those with less. We do this using the **nb2listw** function

```
neigh_lw_W <- nb2listw(neigh_nb, zero.policy = T)
print(neigh_lw_W, zero.policy=T)
```

```
## Characteristics of weights list object:
## Neighbour list object:
```

```
## Number of regions: 88
## Number of nonzero links: 430
## Percentage nonzero weights: 5.552686
## Average number of links: 4.886364
## 2 regions with no links:
## 36 38
##
## Weights style: W
## Weights constants summary:
##    n   nn S0       S1       S2
## W 86 7396 86 40.11216 353.6972
```

We have to introduce the option "zero.policy = TRUE" because we have 2 areas (green areas/parks) with no neighbors in the original shapefile

Calculating a measure of spatial autocorrelation

Finally we are ready to calculate measures of spatial autocorrelation. In this example we calculate the Moran's I statistic under normality

```
moran_u <- moran.test(neigh@data$SUM_NH_Black, listw = neigh_lw_W,
                      zero.policy = T, randomisation=FALSE)
moran_u
```

```
##
##  Moran I test under normality
##
## data:  neigh@data$SUM_NH_Black
## weights: neigh_lw_W
##
## Moran I statistic standard deviate = 5.1102, p-value = 1.609e-07
## alternative hypothesis: greater
## sample estimates:
## Moran I statistic       Expectation          Variance
##       0.354440899      -0.011764706       0.005135344
```

According to these results, we can reject the hypothesis of no spatial autocorrelation in the distribution of blacks in Houston

Rethinking spatial analysis: the sf package

The Simple Features model, implemented by the **sf** package in R, is a recent development that simplifies geographic data and brings R closer other alternatives for spatial data analysis such as PostGIS.

This model greatly increases the efficiency of spatial operations because it condenses different geographic forms into a single geometry class, which is included in a data frame as a list. It has a much more compact, intuitive syntax, and also integrates the syntax of **dpylr** for handling data.

To try it out, we use the function **st__read** from the package **sf** to import the Super Neighborhoods shapefile

```
library(sf)
neigh <- st_read("Super_Neighborhoods/Super_Neighborhoods.shp")
```

```
## Reading layer `Super_Neighborhoods' from data source `C:\Users\juanita82\Dropbox\R-Course\data\Super
## converted into: POLYGON
## Simple feature collection with 88 features and 11 fields
```

```
## geometry type:  POLYGON
## dimension:      XY
## bbox:           xmin: -95.77118 ymin: 29.52338 xmax: -95.07527 ymax: 30.11074
## epsg (SRID):    4326
## proj4string:    +proj=longlat +datum=WGS84 +no_defs
```

```
class(neigh)
```

```
## [1] "sf"         "data.frame"
```

As you can see, the class of the object "neigh" is no longer SpatialPolygonsDataFrame, but simple data frame.

Plotting with the sf package

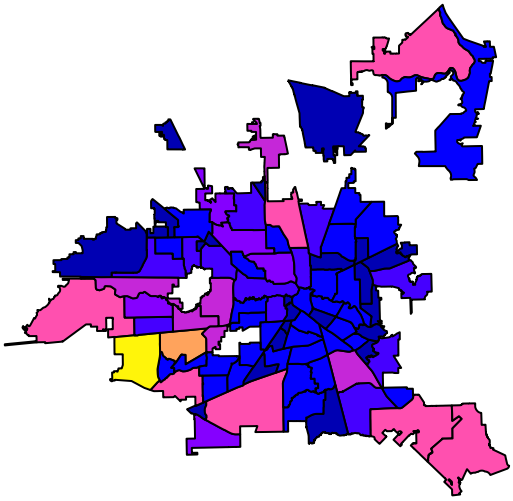
To see how plotting with **sf** works, we come back to our previous 2010 Census Data example. We use this opportunity to introduce another way to merge databases using the **left_join** function of the **dplyr** package (see ?join for different join types):

```
library(dplyr)
SN_data<-read.csv("Census_2010_By_SuperNeighborhood.csv", header=TRUE)
neigh<-left_join(neigh, SN_data, by="POLYID")
```

Plotting total population and hispanic population is now as simple as typing:

```
plot(neigh[c("SUM_TotPop", "SUM_HispPop")])
```

SUM_TotPop



SUM_HispPop