Mapping in R

Geographic data

Spatial objects in R

R has a series of special object types for spatial data. For many mapping / GIS tasks, you will need your data to be in one of these objects.

Spatial objects:

- `SpatialPolygons`
- `SpatialPoints`
- `SpatialLines`

Spatial objects + dataframes:

- `SpatialPolygonsDataFrame`
- `SpatialPointsDataFrame`
- `SpatialLinesDataFrame`

Spatial objects in R

The `tigris` package lets you pull spatial data directly from the US Census. This data comes in directly as a spatial object.

```
library(tigris)
denver_tracts <- tracts(state = "CO", county = "031", cb = TRUE)
class(denver_tracts)
```

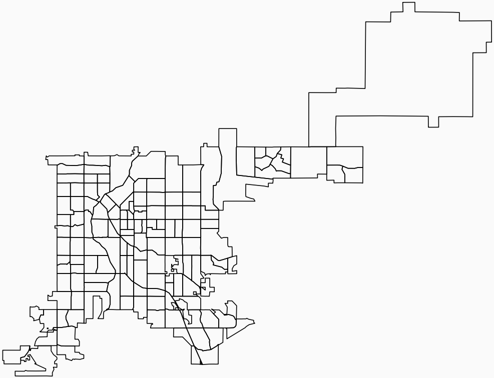
```
## [1] "SpatialPolygonsDataFrame"
## attr(,"package")
## [1] "sp"
```

For more on this package, see the related article in *The R Journal*: `https://journal.r-project.org/archive/accepted/walker.pdf`.

Spatial objects in R

You can plot a spatial object in R just by calling `plot`:

```
plot(denver_tracts)
```



Spatial objects in R

These spatial objects come with a number of special *methods*, or functions that work for the specific object type. You can list these methods using `name`:

```
names(summary(denver_tracts))
```

```
## [1] "class"        "bbox"         "is.projected" "proj4string
## [5] "data"
```

Spatial objects in R

For example, `bbox` will print out the *bounding box* of the spatial object (range of latitudes and longitudes included).

```
bbox(denver_tracts)
```

```
##          min        max
## x -105.10993 -104.60030
## y   39.61443   39.91425
```

Spatial objects in R

The `is.projected` and `proj4string` functions give you some information about the current Coordinate Reference System of the data.

```
is.projected(denver_tracts)
```

```
## [1] FALSE
```

```
proj4string(denver_tracts)
```

```
## [1] "+proj=longlat +datum=NAD83 +no_defs +ellps=GRS80 +towgs8
```

Spatial objects in R

You can access a "slot" in a spatial object with a dataframe to pull out the data. This is similar to indexing a list. Just use `@` instead of `$`. For example, here's the dataframe for the `denver_tracts` spatial object:

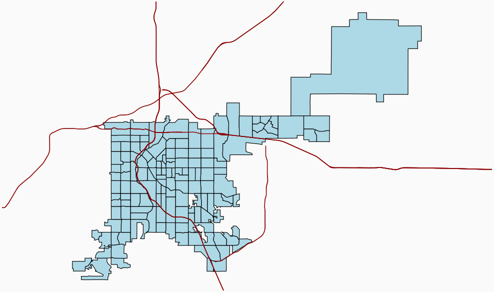
```
head(denver_tracts@data[ , 1:4])
```

```
##    STATEFP COUNTYFP TRACTCE             AFFGEOID
## 25      08      031  000201 1400000US08031000201
## 26      08      031  000302 1400000US08031000302
## 27      08      031  001101 1400000US08031001101
## 28      08      031  002802 1400000US08031002802
## 29      08      031  003300 1400000US08031003300
## 30      08      031  004006 1400000US08031004006
```

Spatial objects in R

You can add different layers of spatial objects onto the same plot. To do that, just use `add = TRUE` for added layers. For example, to add primary roads to the Denver census tract map, you could run:

```
denver_roads <- primary_roads()
plot(denver_tracts, col = "lightblue")
plot(denver_roads, add = TRUE, col = "darkred")
```



Spatial objects in R

If you read in a shapefile, it will automatically be one of these shape objects. However, you can also convert other data into shape objects.

- Functions from `sp` package convert data into spatial objects
- `fortify` converts from a spatial object to a dataframe (useful for `ggplot` plotting)

Spatial objects in R

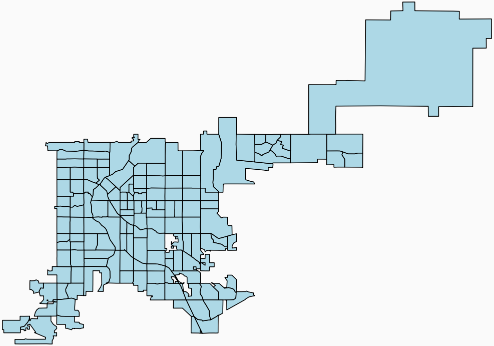
You can use the `fortify` function from ggplot2 to convert the spatial object into a dataframe, so you can plot it using polygons in ggplot2.

```
fortify(denver_tracts) %>%
  dplyr::select(1:4) %>% dplyr::slice(1:5)
```

```
## # A tibble: 5 x 4
##        long      lat order  hole
##       <dbl>    <dbl> <int> <lgl>
## 1 -105.0251 39.79400     1 FALSE
## 2 -105.0213 39.79398     2 FALSE
## 3 -105.0208 39.79109     3 FALSE
## 4 -105.0158 39.79107     4 FALSE
## 5 -105.0064 39.79105     5 FALSE
```

Spatial objects in R

```
denver_tracts %>%
  fortify() %>%
  ggplot(aes(x = long, y = lat, group = group)) +
  geom_polygon(fill = "lightblue", color = "black") +
  theme_void()
```



Projections

Spatial objects can have different Coordinate Reference Systems (CRSs). CRSs can be *geographic* (e.g., WGS84, for longitude-latitude data) or *projected* (e.g., UTM, NADS83).

There is a website that lists projection strings and can be useful in setting projection information or re-projecting data: `http://www.spatialreference.org`

Here is an excellent resource on projections and maps in R from Melanie Frazier: `https://www.nceas.ucsb.edu/~frazier/RSpatialGuides/OverviewCoordinateReferenceSystems.pdf`

Projections

To tell R the Coordinate Reference System of some data, set this attribute with `proj4string` (similar to setting column names with `colnames`):

```r
## Generic code
proj4string(my_spatial_object) <- "+proj=longlat +datum=NAD83"
```

This does not create a projection. Instead, this is just how you tell R what projection the data already is in.

Projections

The CRS function creates CRS class objects that can be used to specify projections. You input a character string of projection arguments into this function (for example, `CRS("+proj=longlat +datum=NAD27")`). You can also use, however, use a shorter EPSG code for a projection (for example, `CRS("+init=epsg:28992")`).

```
library(sp)
CRS("+proj=longlat +datum=NAD27")
```

```
## CRS arguments:
##  +proj=longlat +datum=NAD27 +ellps=clrk66
## +nadgrids=@conus,@alaska,@ntv2_0.gsb,@ntv1_can.dat
```

```
CRS("+init=epsg:28992")
```

```
## CRS arguments:
##  +init=epsg:28992 +proj=sterea +lat_0=52.15616055555555
## +lon_0=5.38763888888889 +k=0.9999079 +x_0=155000 +y_0=463000
```

Projections

To **change** the projection of a spatial object, you can use the spTransform function from the sp package.

```
## Generic code
my_spatial_object <- spTransform(my_spatial_object,
                                 proj4string(another_sp_object))
```

Projections

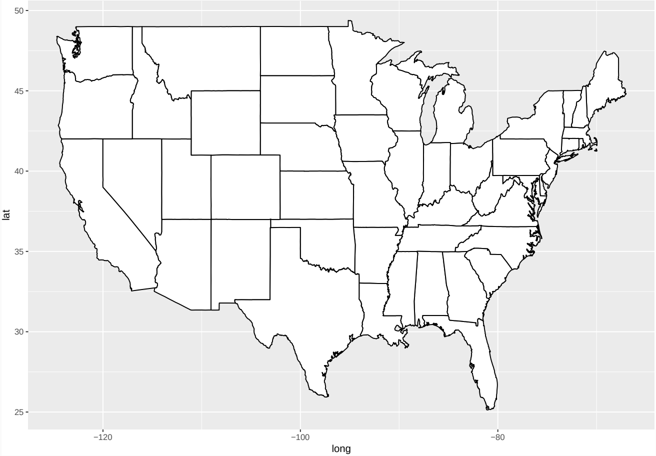
The `coord_map` function in `ggplot2` can help you with map projections, as well. Here's an example from the help file with a US map.

```
states <- map_data("state")
usamap <- ggplot(states,
                 aes(long, lat, group = group)) +
  geom_polygon(fill = "white", colour = "black")
```

Projections

The default is Cartesian coordinates:

```
usamap
```



Projections

Mercator projection:

```
usamap + coord_map("mercator")
```

Projections

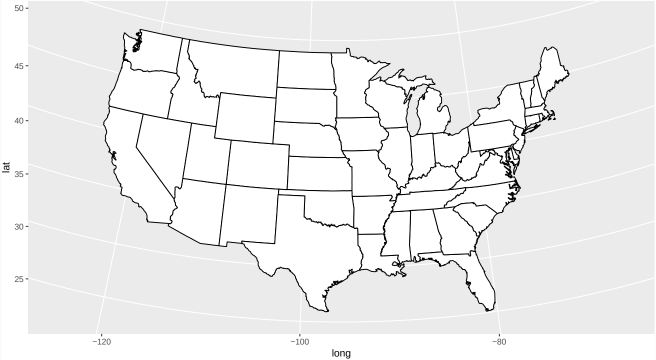
Gilbert projection

```
usamap + coord_map("gilbert")
```

Projections

Conic projection:

```
usamap + coord_map("conic", lat0 = 30)
```



Shapefiles

- File format (ESRI, but usuable by other software)
- Not a single file, but rather a directory of related files (e.g., `.shp`, `.shx`, `.dbf`, `.prj`)
- Typically includes both geographic information (e.g., locations of county boundaries) and attribute information (e.g., median income of each county)
- To read shapefiles into R, use the `readOGR` function from the `rgdal` package
- You can also write out shapefiles you've created or modified in R, using `writeOGR`.

R as GIS

Capabilities

You can use R for a number of GIS-style tasks:

- Clipping
- Creating buffers
- Measuring area of a polygon
- Counting points in polygon

Capabilities

There are some advantages to using R for this:

- R is free
- You can write all code in a script, so research is more reproducible
- You save time and effort by staying in one software system, not going between different software

There are some advantages to GIS, too, though:

- More user-friendly at the start (point-and-click)
- R spatial functionality is still spread over lots of packages, with different syntax and conventions.

Spatial Points

For an example, I've cleaned up some FARs data at the driver level for 2001–2010:

```
load("../data/fars_colorado.RData")
driver_data %>%
  dplyr::select(1:5) %>% dplyr::slice(1:5)
```

```
## Warning in as.POSIXlt.POSIXct(x, tz): unknown timezone 'defau
## Denver'
```

```
## # A tibble: 5 x 5
##   state st_case county                date latitude
##   <dbl>   <dbl>  <dbl>              <dttm>    <dbl>
## 1     8   80001     51 2001-01-01 10:00:00 39.10972
## 2     8   80002     31 2001-01-04 19:00:00 39.68215
## 3     8   80003     31 2001-01-03 07:00:00 39.63500
## 4     8   80004     31 2001-01-05 20:00:00 39.71304
## 5     8   80005     29 2001-01-05 10:00:00 39.09733
```

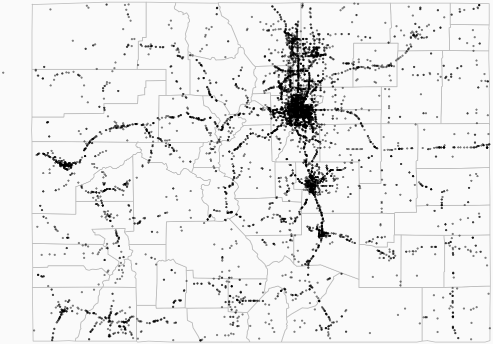
Spatial Points

Here is how you would plot fatal accidents (by driver) in Colorado without using spatial objects:

```
ggplot2::map_data("county", region = "Colorado") %>%
  ggplot2::ggplot(ggplot2::aes(x = long, y = lat,
                               group = subregion)) +
    ggplot2::geom_polygon(color = "gray", fill = NA) +
    ggplot2::theme_void() +
    ggplot2::geom_point(data = driver_data,
                        ggplot2::aes(x = longitud,
                                     y = latitude,
                                     group = NULL),
                        alpha = 0.5, size = 0.7 )
```

Spatial Points

Fatal accidents (by driver) in Colorado, 2001–2010:



Counting points in each polygon

You can also make a choropleth of county accident counts without using spatial data, by using the `choroplethr` package.

To do this, you'll first need to use `dplyr` functions to limit to unique accidents (rather than drivers) and add up the number of accidents in each county. In this case, it's possible to add up accidents by county because `county` is included as a column in our data.

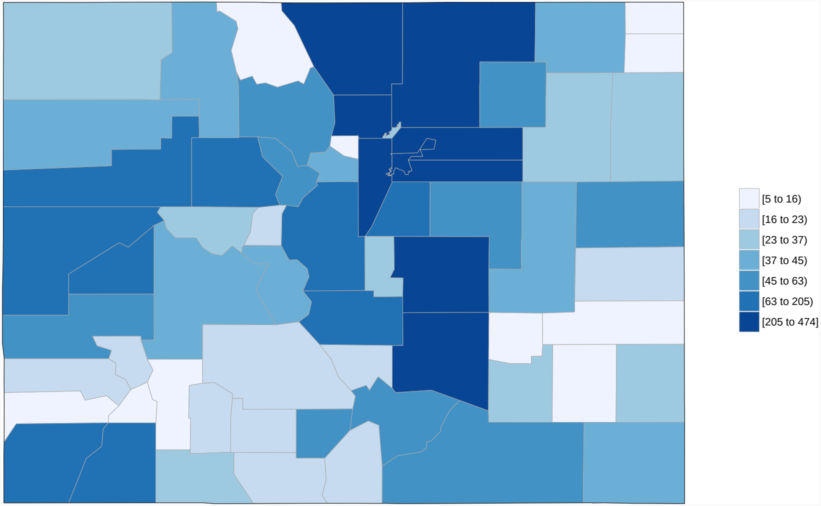
Counting points in each polygon

```
county_accidents <- driver_data %>%
  dplyr::select(state, st_case, county, latitude, longitud) %>%
  dplyr::distinct() %>%
  dplyr::mutate(county = str_pad(county, width = 3,
                                 side = "left", pad = "0")) %>%
  tidyr::unite(region, state, county, sep = "") %>%
  dplyr::group_by(region) %>%
  dplyr::summarize(value = n()) %>%
  dplyr::mutate(region = as.numeric(region))
county_accidents %>% slice(1:4)
```

```
## # A tibble: 4 x 2
##   region value
##    <dbl> <int>
## 1   8001   372
## 2   8003    47
## 3   8005   305
```

Counting points in each polygon

```r
county_choropleth(county_accidents, state_zoom = "colorado")
```



Counting points in each polygon

This "out-of-the-box" solution let us looks at accident counts by county, but what if we want to look at a geographical unit for which we don't have an identifying column?

For example, we might want to look at accident counts by census tract in Denver. To do this, we'll need to link each accident (point) to a census tract (polygon), and then we can count up the number of points linked to each polygon.

First, I've created a dataframe with only accidents in Denver (based on the `county` column in the accident data):

```
denver_fars <- driver_data %>%
  filter(county == 31)
```

Counting points in each polygon

To do this, both the census tracts and the accident data need to be in spatial objects.

```
library(sp)
denver_fars_sp <- denver_fars %>%
  dplyr::rename(longitude = longitud)
coordinates(denver_fars_sp) <- c("longitude", "latitude")
proj4string(denver_fars_sp) <- CRS("+init=epsg:4326")
```

Note that the dataframe is changed into a spatial object by changing its `coordinates` attribute, and that the CRS was set uas the `proj4string` attribute.

Counting points in each polygon

```r
summary(denver_fars_sp)
```

```
## Object of class SpatialPointsDataFrame
## Coordinates:
##                   min       max
## longitude -105.10973 -104.0122
## latitude    39.61715   39.8381
## Is projected: FALSE
## proj4string :
## [+init=epsg:4326 +proj=longlat +datum=WGS84 +no_defs +ellps=W
## +towgs84=0,0,0]
## Number of points: 695
## Data attributes:
##      state       st_case          county         date
##  Min.   :8   Min.   :80001   Min.   :31   Min.   :2001-01-03
##  1st Qu.:8   1st Qu.:80121   1st Qu.:31   1st Qu.:2003-01-06
##  Median :8   Median :80268   Median :31   Median :2005-01-29
```

Counting points in each polygon

To be able to pair up polygons and points, their spatial objects need to have the same CRS. To help later with calculating the area of each polygon, I'll use a projected CRS that is reasonable for Colorado.

```
proj4string(denver_tracts)
```

```
## [1] "+proj=longlat +datum=NAD83 +no_defs +ellps=GRS80 +towgs8
```

```
CRS(proj4string(denver_tracts))
```

```
## CRS arguments:
##  +proj=longlat +datum=NAD83 +no_defs +ellps=GRS80 +towgs84=0,
```

Counting points in each polygon

To reproject spatial data, you can use the `spTransform` function:

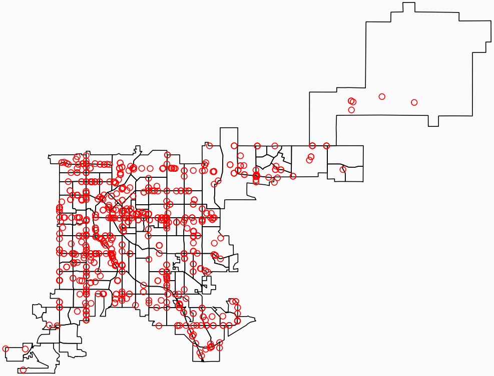
```
denver_tracts_proj <- spTransform(denver_tracts,
                                  CRS("+init=epsg:26954"))
denver_fars_proj <- spTransform(denver_fars_sp,
                                CRS(proj4string(denver_tracts)))
```

The `spTransform` function transforms the coordinates in a spatial object into a new coordinate reference system.

Counting points in each polygon

Here is a map of the tracts with the accidents overlaid:

```r
plot(denver_tracts)
plot(denver_fars_proj, add = TRUE, col = "red", pch = 1)
```



Counting points in each polygon

Now, the data's in a format where we can link spatial points to spatial polygons.

The `poly.counts` function in the `GISTools` package will measure the number of points that fall within each polygon.

It results in a vector with one element for each polygon (census tract in our example), where the element name identifies the polygon and the cell value gives the count within that polygon.

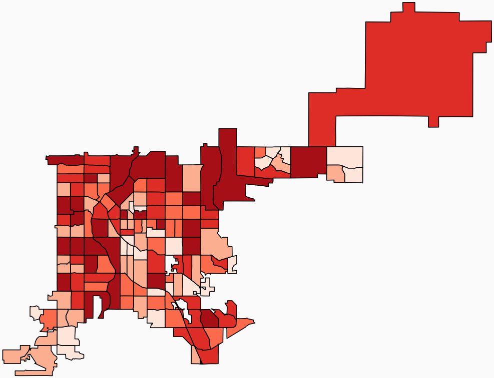
```
tract_counts <- poly.counts(denver_fars_proj, denver_tracts)
head(tract_counts)
```

```
## 25 26 27 28 29 30
##  7  2  2  0  0  4
```

Counting points in each polygon

You can use a choropleth to show these accident counts. The quickest way to do this is probably to use the `choropleth` function in the `GISTools` package.

```
choropleth(denver_tracts, tract_counts)
```



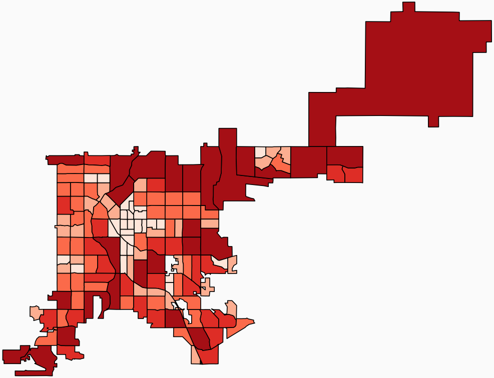
Determining area

There is another function in the package that calculates the area of each polygon.

```
library(GISTools)
head(poly.areas(denver_tracts_proj))
```

```
##        25        26        27        28        29        30
## 2100172.2 1442824.1  897886.3  881530.5 1282812.2 1948187.1
```

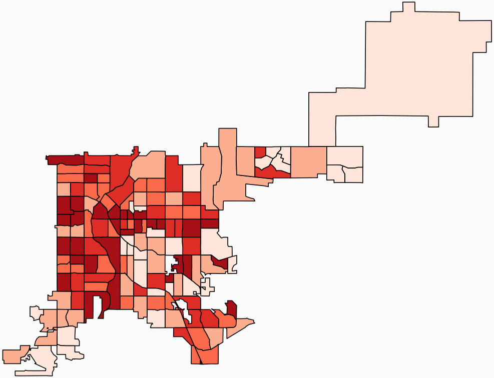
```r
choropleth(denver_tracts, poly.areas(denver_tracts_proj))
```



Counting points in each polygon

You can combine these ideas to create a choropleth of the rate of fatal accidents per area in Denver census tracts.

```
choropleth(denver_tracts, tract_counts /
             poly.areas(denver_tracts_proj))
```



Raster data

When mapping in R, you may also need to map *raster data*.

You can think of this as pixels— the graphing region is divided into even squares, and color is constant within each square.

There are functions that allow you to “rasterize” data. That is, you take spatial points data, divide the region into squares, and count the number of points (or other summary) within each square.

Raster data

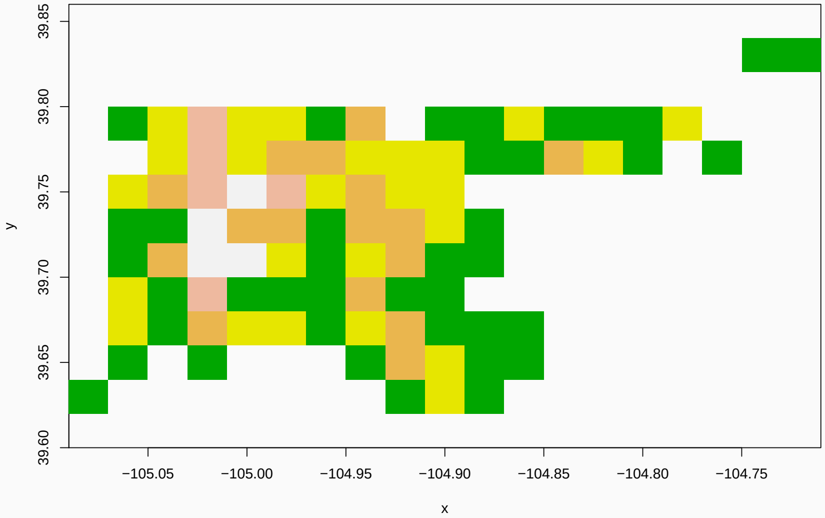
```r
bbox(denver_fars_sp)
```

```
##                  min       max
## longitude -105.10973 -104.0122
## latitude    39.61715   39.8381
```

```r
library(raster)
denver_raster <- raster(xmn = -105.09, ymn = 39.60,
                        xmx = -104.71, ymx = 39.86,
                        res = 0.02)
den_acc_raster <- rasterize(geometry(denver_fars_sp),
                            denver_raster,
                            fun = "count")
```

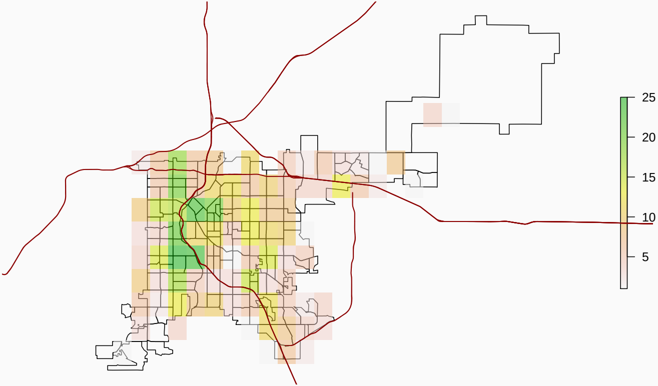
Raster data

```
image(den_acc_raster, col = terrain.colors(5))
```



Raster data

```
plot(denver_tracts)
plot(den_acc_raster, add = TRUE, alpha = 0.5)
plot(denver_roads, add = TRUE, col = "darkred")
```



Other capabilities

You can also use R for other spatial tasks:

- Kernel density estimation
- Identifying clusters
- Kriging
- Measuring spatial autocorrelation

Find out more

- *Applied Spatial Data Analysis with R* by Roger Bivand (available online through CSU library)
- *An Introduction to R for Spatial Analysis and Mapping* by Chris Brunsdon and Lex Comber
- CRAN Spatial Data Task View
- R Spatial Cheatsheet
- Great blog post (among many) by Zev Ross

Group project

- I'm inviting Anne Lenaerts on Wednesday. By then, create a simple pdf or HTML from RMarkdown that shows (brief) examples of any figures or other results your group has created. For Group 1, the examples of "tidy" data will work for this.
- For each person, write at least one function that inputs one of the cleaned data files and outputs a figure or a table of results (this is essentially wrapping your work so far into a function format). For Group 1, these will be functions to input an Excel template and output a clean dataset.
- Ultimately, each group should pick at least two functions to include in the Shiny App. You will describe all of the functions in your group report, as well as how you picked the two final functions.
- For one of these two functions, write a version of the function that uses `plotly` to create an interactive graphic.