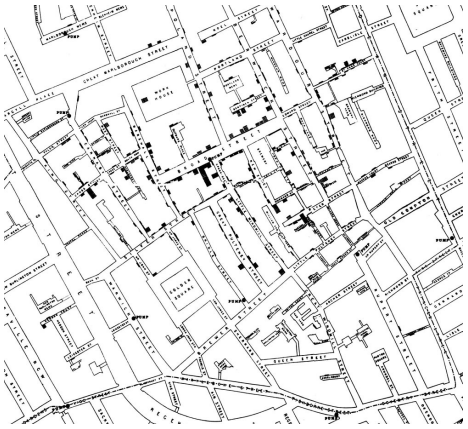
EPI 563: Spatial Epidemiology, Fall 2020

Michael Kramer

Last updated: 2020-08-09

Contents

How to use this eBook



Welcome to Concepts & Applications in Spatial Epidemiology (EPI 563)! This eBook is one of several sources of information and support for your progress through the semester. For an overview of the course, expectations, learning objectives, assignments, and grading, please review the full course syllabus on Canvas. This eBook serves to provide a 'jumping off point' for the content to be covered each week. Specifically, the content herein will introduce key themes, new vocabulary, and provide some additional detail that is complementary to the asynchronous (pre-recorded) video lectures, and foundational to the synchronous (in class) work.

0.1 Strategy for using this eBook

There is a separate module or chapter for eac week's content. In general, the content within each week's section is divided into two sections focusing on spatial thinking and spatial analysis. This dichotomy does not always hold, but in broad terms you can expect these sections to be more specific to content in class on Tuesday versus Thursday respectively.

- Spatial thinking for epidemiology: This section introduces vocabulary, concepts, and themes that are important to the incorporation of spatialized or geo-referenced data into epidemiologic work. At a minimum, plan to read this content prior to class Tuesday, although you will likely benefit from reading both sections before Tuesday.
- Spatial analysis for epidemiology: This section is more focused on data management, visualization, spatial statistics, and interpretation. This content is relevant for our work together on Tuesday's, but is essential for succesful work in the Thursday lab activities.

You can read the content on the web via your browser, or optionally, you can download a PDF of the current content. However please note that I will be continually updating the eBook throughout the semester, so if you choose to download, please double-check the **Last updated** date to be sure you have the most recent version.

Part I

Getting ready...

Software installation

The information in this module follow on the pre-class video on setting up R and RStudio on your computer.

Installing `R` on your computer

As of August 2020, the most up-to-date version of R is 4.0. The R Project for Statistical Computing are continually working to update and improve R, and as a result there are new versions 1-2 times per year.

If you already have R installed, you can open the console and check your current version by doing this: R.Version()$version.string

If you do not have R or have an older version than 3.5.1 you can install R by going to the R repository: https://www.r-project.org/. Note that there are many 'mirrors' or servers where the software is stored. Generally it is wise to select one that is geographically close to you, although any should work in theory. One mirror that is relatively close to Atlanta is here: http://archive.linux.duke.edu/cran/

Installing RStudio on your computer

The current version of RStudio 1.3.1056. If you do not have RStudio or have a version older than 1.2 please install/update.

TO INSTALL: go to https://www.rstudio.com/products/rstudio/download/

TO UPDATE: Open RStudio and go to Help Menu and choose 'Check for Updates'

Installing packages for this course

While base R has a great deal of essential functionality, most of the power of R comes from the rapidly growing list of user-created and contributed 'packages'. A package is simply a bundle of functions and tools, sometimes also including example datasets, basic documentation, and even tutorial 'vignettes'. You can see all the official R packages by going here: https://cran.r-project.org/web/packages/.

The most common way to install package in R is with the install.pacakges() command. For instance to install the package ggplot2 you do this:

```
install.packages("ggplot2")
```

Notice that for install.packages() you need quotes around the package name. Remember that you only need to install a package once (although you may have to update packages occasionally – see the green Update button in the Packages tab in R Studio). When you want to actually use a package (for example ggplot2) you call it like this:

```
library(ggplot2)
```

Notice that for the library() function you **do not** need quotes around the package name (unlike the install.packages() above). If your call to library() is working, nothing visible happens. However if you see errors, they might be because your package is out of date (and thus needs to be updated/reinstalled), or because some important dependencies are missing. Dependencies are other packages on which this package depends. Typically these are installed by default, but sometimes something is missing. If so, simply install the missing package and then try calling library(ggplot2) again.

While **most** packages can be installed as mentioned above (e.g. using install.packages()), there are instances where an installation requires additional tools, for instance to install from source or from github. Luckily there is a package for that! It is called Rtools, and you should install that **before** you install the packages below.

As you submit each installation request, note the output. If you get a warning that says installation was not possible because you are missing a package 'namespace', that suggests you are missing a dependency. Try installing the pacakge mentioned in the error. If you have trouble, reach out to the TA's!

Installing `Rtools40`

Navigate to this website: https://cran.r-project.org/bin/windows/Rtools/ and follow the instructions specific to your operating system.

Installing packages used for general data science

For the rest of this page, copy and paste the provided code in order to install packages necessary for this course.

These packages will support some of our general work in R, including working with RMarkdown and R Notebooks, as well as data manipulation tools from the tidyverse. You can learn more about the tidyverse here: https://tidyverse.tidyverse.org/. The tidyverse is actually a collection of data science tools including the visualization/plotting pakcage ggplot2 and the data manipulation package dplyr. The package tinytex, rmarkdown, and knitr are all necessary for creating R Notebooks, which is the format by which many assignments will be submitted.

```
install.packages('tidyverse')
install.packages(c('tinytex', 'rmarkdown', 'knitr'))
tinytex::install_tinytex()
# this function installs the tinytex LaTex on your
#  computer which is necessary for rendering (creating)
    PDF's
```

Installing packages use for geographic data

There are many ways to get the data we want for spatial epidemiology into R. Because we often (but don't always) use census geographies as aggregating units, and census populations as denominators, the following packages will be useful. They are designed to quickly extract both geographic boundary files as well as attribute data from the US Census website via an API. **NOTE**: For these to work you have to request a free Census API key. Notice the help() function below to get instructions on how to do this.

```
install.packages(c('tidycensus','tigris'))

help('census_api_key','tidycensus')
```

Installing packages used for spatial data manipulation & visualization

This section installs a set of tools specific to our goals of importing, exporting, manipulating, visualizing, and analyzing spatial data. The first line of packages have functions for defining, importing, exporting, and manipulating spatial data. The second line has some tools we will use for visualizing spatial data (e.g. making maps!).

```
install.packages(c('sp', 'sf', 'rgdal', 'rgeos', '
    maptools', 'OpenStreetMap'))
install.packages(c('tmap', 'tmaptools', 'ggmap', 'shinyjs
    ', 'shiny', 'micromap'))
```

Installing packages used for spatial analysis

Finally these are packages specifically to spatial analysis tasks we'll carry out.

```
install.packages(c('spdep', 'CARBayes', 'sparr'))
install.packages(c('GWmodel', 'spgwr') )
```

Part II

Weekly Modules

Chapter 1

Locating Spatial Epidemiology

1.1 Getting Ready

1.1.1 Learning objectives

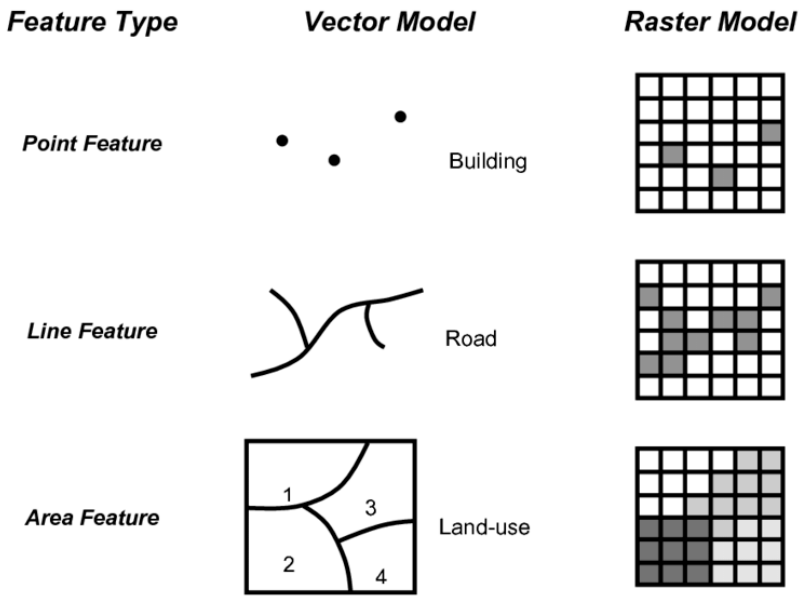
Table 1.1: Learning objectives by weekly module

After this module you should be able to...
Explain the potential role of spatial analysis for epidemiologic thinking and practice.
Produce simple thematic maps of epidemiologic data in R.

1.1.2 Additional Resources

- Geocompution with R by Robin Lovelace. This will be a recurring 'additional resource' as it provides lots of useful insight and strategy for working with spatial data in R.
- A basic introduction to the ggplot2 package. This is just one of dozens of great online resources introducing the grammar of graphics approach to plotting in R.
- A basic introduction to the tmap package This is also only one of many introductions to the tmap mapping package. tmap builds on the grammar of graphics philosophy of ggplot2, but brings a lot of tools useful for thematic mapping!

1.1.3 Important Vocabulary



1.2 Spatial Thinking in Epidemiology

When first learning epidemiology, it can be difficult to distinguish between the skills and concepts that are foundational to defining our profession or our professional identity. For instance are we data analysts, scientists, data scientists, technicians or something else? These questions are bigger than we can address in this class, but they are important when beginning to learn spatial epidemiology, because there is a tendency to focus on the data and the methods without understanding how each of those relate to the scientific questions and population health we are ultimately responsible for.

Take, for example, **data**. Data is central to quantitative analysis, including epidemiologic analysis. So how is data different in spatial epidemiology? One thing that may be distinct about data in spatial epidemiology is the unit of analysis. While in many other examples in your epidemiology coursework, the explicit (or sometimes implicit) unit of analysis has been the individual person. It is certainly possible for individuals to be the unit of analysis in spatial epidemiology. However oftentimes the units we observe and measure in spatial epidemiology – and therefore the units that compose our **data** – are not individuals but instead are geographic units (e.g. census tract, county, state, etc) and by extension the collection or aggregation of all the individuals therein.

One implication of the above discussion is that you should always be able to

answer a fundamental question about any dataset you wish to analyze: "what does one row of data represent?" A row of data is one way to think of the unit of analysis, and often (but not always) in spatial epidemiology a row of data is a summary of the population contained by a geographic unit.

The importance of having clarity about the unit of analysis cannot be overstated, because it has implications for measurement, understanding of risks for bias, and ultimately for the available tools for analysis and inference that we derive from those analyses.

1.3 Spatial Analysis in Epidemiology

1.3.1 Spatial data storage foramts

If you have worked with spatial or GIS data using ESRI's ArcMap, you will be familiar with what are called shapefiles. This is one very common format for storing geographic data on computers. ESRI shapefiles are not actually a single file, but are anywhere from four to eight different files all with the same file name but different extensions. Each different file (corresponding to an extension) contains a different portion of the data ranging from the geometry data, the attribute data, the projection data, an index connecting it all together, etc.

What you may not know is that shapefiles are not the only (and in my opinion **definitely not the best**) way to store geographic data. In this class I recommend storing data in a format called geopackages indicated by the .gpkg extension. Geopackages are an open source format that were developed to be functional on mobile devices. They are useful when we are storing individual files in an efficient and compact way. It is worth noting that many GIS programs including ArcMap and QGIS can both read and write the geopackage format; so there is no constraint or limitation in terms of software when data are stored in .gpkg format.

1.3.2 Representing spatial data in `R`

Just as our conceptualization of, or thinking about data in spatial epidemiology requires some reflection, the actual storage and representation of that data with a computer tool such as R also requires some attention. Specifically spatial data in R is not exactly like the conventional aspatial epidemiologic data, but it is also need not be as complex as spatial data in software platforms like ESRI's ArcMap.

First, it may be obvious, but spatial data is more complex than simple rectangular attribute data (e.g. data tables where a row is an observation and a

column is a variable). To be spatial, a dataset must have a representation of geogrphy, spatial location, or spatial relatedness, and that is most commonly done with either a vector or raster data model (see description above in vocabulary). Those spatial or geographic representations must be stored on your computer and/or held in memory, hopefully with a means for relating or associating the individual locations with their corresponding attributes.

Over the past 10+ years, R has increasingly been used to analyze and visualize spatial data. Early on, investigators tackling the complexities of spatial data analysis developed a number of ad hoc, one-off approaches to these data. This helped in the short term but created other problems as users needed to chain together steps and had to convert one data format to another. An eventual response to this early tumult was a thoughtful and systematic approach to defining a class of data that tackled the unique challenges of spatial data in R. Roger Bivand, Edzer Pebesma and others developed the sp package which defined spatial data classes, and provided functional tools to interact with them.

The sp package defined specific data classes to hold points, lines, and polygons, as well as raster/grid data; each of these data classes can contain geometry only (these have names like SpatialPoints or SpatialPolygons) or could contain geometry plus related data attributes (these have names like SPatialPointsDataFrame or SpatialPolygonsDataFrame). Each spatial object can contain all the information spatial data might include: the spatial extent (min/max x, y values), the coordinate system or spatial projection, the geometry information, the attribute information, etc.

Because of the flexibility and power of the sp$*$ class of objects, they became a standard up until the last few years. sp$*$ classes continue to be the only format allowed by a few of the packages we will use this semester. However analysts sometimes find the complexity of the sp$*$ objects to be a hindrance to efficient processing of geographic data. Specifically the information is stored in numerous 'slots' (not a formal list, but conceptually a little like list elements).

As the number of ways to visualize data increases, there was a desire to make spatial data behave more like tabular data. This led the same team (e.g. Bivand, Pebesma, others) to develop the Simple Features set of spatial data classes for R. Loaded with the sf package, this data format is almost surely going to become the standard for the coming years. Recognizing that many users and functions prefer the familiar sp$*$ objects, the sf package includes a number of utility functions for easily converting back and forth.

In this class we will use sf$*$ objects as the preferred data class, but because some of the tools we'll learn require sp$*$ we will go back and forth.

sf$*$ data classes are designed to hold all the essential spatial information (projection, extent, geometry), but do so with an easy to evaluate data frame format that integrates the attribute information and the geometry information together. The result is more intuitive sorting, selecting, aggregating, and visualizing.

1.3.3 Benefits of `sf` data classes

As Robin Lovelace writes in his online eBook, Gecomputation in R, sf data classes offer an approach to spatial data that is compatible with QGIS and PostGIS, important non-ESRI open source GIS platforms, and sf functionality compared to sp provides:

1. Fast reading and writing of data
2. Enhanced plotting performance
3. sf objects can be treated as data frames in most operations
4. sf functions can be combined using %>% pipe operator and works well with the tidyverse collection of R packages (we'll talk about this more later in the semester)
5. sf function names are relatively consistent and intuitive (all begin with st_)

1.3.4 Working with spatial data in `R`

Here and in lab, one example dataset we will use quantifies the counts and rates of death from motor vehicle crashes in each of Georgia's $n = 159$ counties. The dataset is vector in that it represents counties as polygons with associated attributes (e.g. the mortality information, county names, etc).

1.3.4.1 Importing spatial data into `R`

Although spatial data can be stored on your computer in many formats, it is possible to import almost any format into R. That means if you received data as a .shp shapefile, as a .gpkg geopackage, or as a .tif raster file, each can be easily imported.

All sf functions that act on spatial objects begin with the prefix st_. Therefore to import (read) data we will use st_read(). This function determines **how** to import the data based on the extension of the file name you specify. Look at the help documentation for st_read(). Notice that the first argument dsn=, might be a complete file name (e.g. myData.shp), or it might be a folder name (e.g. mygeodatabase.gdb). So if you had a the motor vehicle crash data saved as both a shapefile (mvc.shp, which is actually six different files on your computer), and as a geopackage (mvc.gpkg) you can read them in like this:

```
# this is the shapefile
mvc.a <- st_read('GA_MVC/ga_mvc.shp')

# this is the geopackage
mvc.b <- st_read('GA_MVC/ga_mvc.gpkg')
```

We can take a look at the defined data class of the imported objects within R:

```
class(mvc.a)
```

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

```
class(mvc.b)
```

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

First, note that when we use the st_read() function, the data class (e.g. the way the data are defined and organized within R) is both as an sf and data.frame class. That is incredibly important! One thing that means is that our complex spatial dataset is held within R as a relatively simple-seeming object: a rectangular data.frame. How does that work? We will explore this more in lab but essentially each dataset has rows (observations) and columns (variables). We can see the variable/column names like this:

```
names(mvc.a)
```

```
## [1] "GEOID"      "NAME"       "MVCRATE_17" "geometry"
```

```
names(mvc.b)
```

```
## [1] "GEOID"      "NAME"       "MVCRATE_17" "geom"
```

We can see that each dataset has the same attribute variables (e.g. GEOID , NAME, MVCRATE_17), and then a final column called geometry in one and called geom in another. These geometry columns are unique in that they don't hold a single value like the other columns; each 'cell' in those columns actually contains an embedded list of x, y coordinates defining the vertices of the polygons for each of Georgia's counties.

Combining these two observations, we now know that we can work with a wide range of spatial data formats, and that once imported we can conceive of (and manipulate!) these data almost as if they were simple rectangular datasets. This has implications for subsetting, recoding, merging, and aggregating data as we'll learn in the coming weeks.

1.3.4.2 Exporting spatial data from R

While importing is often the primary challenge with spatial data and R, it is not uncommon that you might modify or alter a spatial dataset and wish to save it for future use, or to write it out to disk to share with a colleague. Luckily the sf package has the same functionality to write an sf spatial object to disk in a wide variety of formats including shapefiles (.shp) and geopackages (.gpkg). Again, R uses the extension you specify in the filename to determine the target format.

```
# Write the file mvc to disk as a shapefile format
st_write(mvc, 'GA_MVC/ga_mvc_v2.shp')

# Write the file mvc to disk as a geopackage format
st_write(mvc, 'GA_MVC/ga_mvc_v2.gpkg')
```

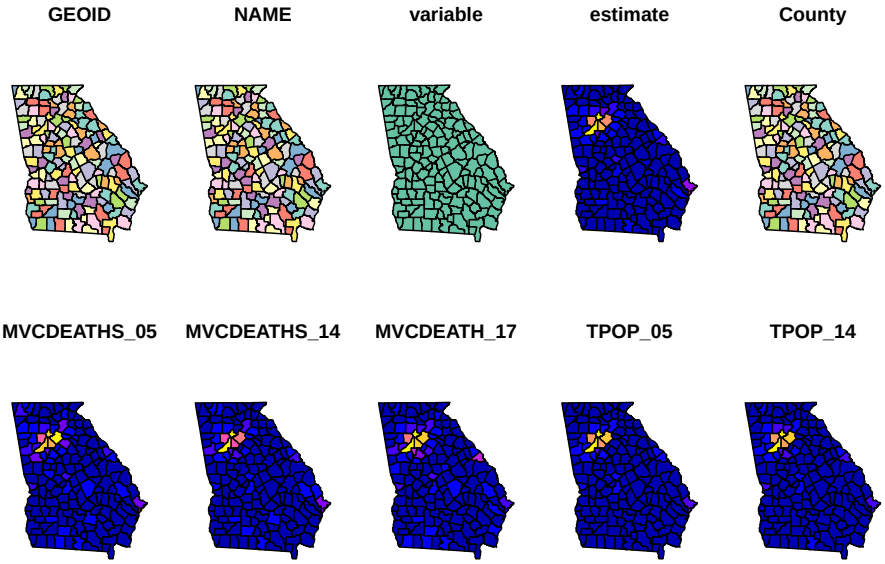
1.3.5 Basic visual inspection/plots

The base-R plot() function is extended by the sf package. That means that if you call plot() on a spatial object **without having loaded** sf, the results will be different than if plot() called **after loading** sf.

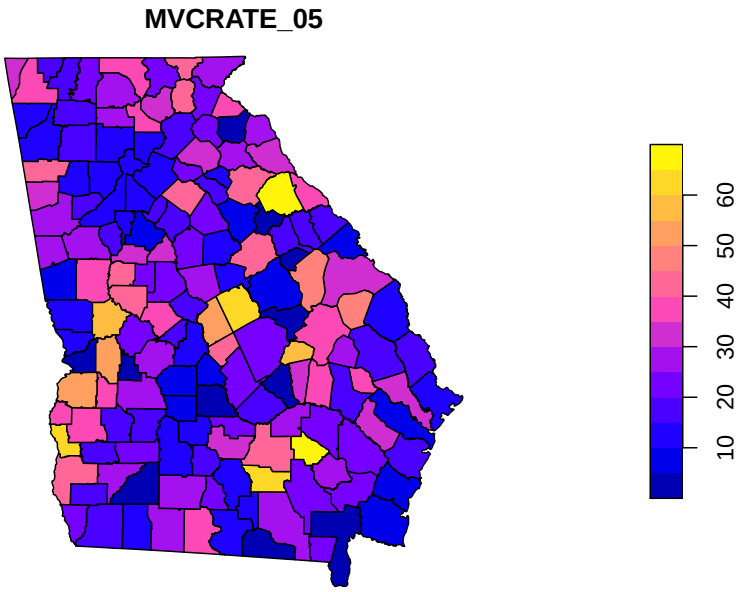
When you plot() with sf, by default it will try to make a map **for every variable in the data frame**! Try it once. If this is not what you want, you can force it to only plot some variables by providing a vector of variable names.

```
plot(mvc) # this plots a panel for every column - or
    actually the first 10 columns
```

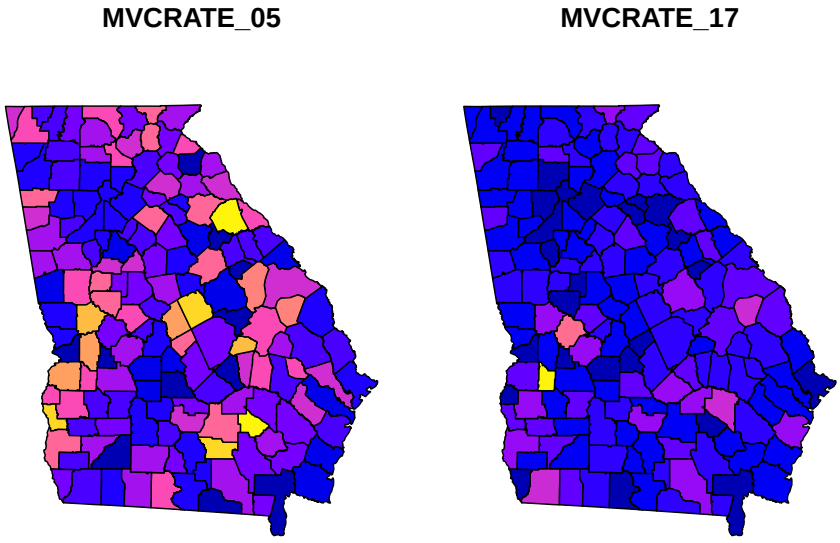
```
## Warning: plotting the first 10 out of 17 attributes;
    use max.plot = 17 to plot
## all
```




```
plot(mvc['MVCRATE_05']) # this plots only a single
    variable, the MVC mortality rate for 2005
```




```
plot(mvc['MVCRATE_05', 'MVCRATE_17']) # this plots two variables: MVC rate in 2005 & 2017
```

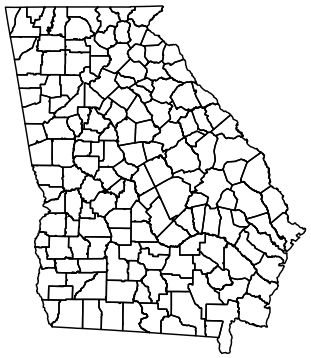


You might only want to see the geometry of the spatial object (e.g. not attributes) if you are checking its extent, the scale, or otherwise confirming some-

thing about the spatial aspects of the object. Here are two approaches to quickly plot the geometry:

```
plot(st_geometry(mvc)) # st_geometry() returns the geom
    information to plot

plot(mvc$geom)  # this is an alternative approach...
    directly plot the 'geom' column
```



1.3.6 Working with CRS and projection

If CRS (coordinate reference system) and projection information was contained in the original file you imported, it will be maintained. **If there is NO CRS information imported it is critical that you find out the CRS information from the data source!**. The most unambiguous way to describe a projection is by using the **EPSG** code, which stands for European Petroleum Survey Group (!). This consortium has standardized the projection definitions in a manner adopted by several R packages including rgdal and sf.

- A useful overview/review of coordinate reference systems in R
- Robin Lovelace's Geocompuation in R on projections with sf
- EPSG website: This link is to a searchable database of valid ESPG codes
- Here are some useful EPSG codes

Important: It is important to distinguish between defining the current projection of data and the act of projecting or transforming data from one known system to a new CRS/projection. **We cannot transform data until we correctly define its current CRS/projection status.** The above function tells us what the current status is. In some cases data do not have associated CRS information and this might be completely blank (for instance if you read in numerical x, y points from a geocoding or GPS process). In those cases you can **set** the underlying CRS using st_set_crs() to define it, but this assumes you **know** what it is. There are two arguments to this function: the first is x = objectName, and the second is value = xxx where 'xxx' is a valid EPSG code.

We already saw the CRS/projection information when we used the head() function above; it was at the top and read WGS 84. Recall there are two main types of CRS: purely **geographic** which is to say coordinate locations are represented as latitude and longitude degrees; and **projected** which means the coordinate values have been transformed for representation of the spherical geoid onto a planar (Euclidean) coordinate system. WGS 84 is a ubiquitous geographic coordinate system common to boundary files retrieved from the U.S. Census bureau.

An important question when you work with a spatial dataset is to understand whether it is primarily a geographic or projected CRS, and if so which one.

```
st_is_longlat(mvc)
```

```
## [1] TRUE
```

This quick logical test returns TRUE or FALSE to answer the question "Is the sf object simply a longitude/latitude geographic CRS?". The answer in this case is TRUE because WGS 84 is a geographic (longlat) coordinate system. But what if it were FALSE or we wanted to know more about the CRS/projection?

```
st_crs(mvc)
```

```
## Coordinate Reference System:
##   User input: WGS 84
##   wkt:
## GEOGCRS["WGS 84",
##     DATUM["World Geodetic System 1984",
##         ELLIPSOID["WGS 84",6378137,298.257223563,
##             LENGTHUNIT["metre",1]]],
##     PRIMEM["Greenwich",0,
##         ANGLEUNIT["degree",0.0174532925199433]],
##     CS[ellipsoidal,2],
##         AXIS["geodetic latitude (Lat)",north,
##             ORDER[1],
##             ANGLEUNIT["degree",0.0174532925199433],
```

```
##             AXIS["geodetic longitude (Lon)",east,
##                 ORDER[2],
##                 ANGLEUNIT["degree",0.0174532925199433]],
##         USAGE[
##             SCOPE["unknown"],
##             AREA["World"],
##             BBOX[-90,-180,90,180]],
##         ID["EPSG",4326]]
```

This somewhat complicated looking output is a summary of the CRS stored with the spatial object. There are two things to note about this output:

- At the top, the User input is WGS 84
- At the bottom of the section labeled GEOGCRS it says ID["EPSG",4326"]

While there are literally hundreds of distinct EPSG codes describing different geographic and projected coordinate systems, for this semester there are three worth remembering:

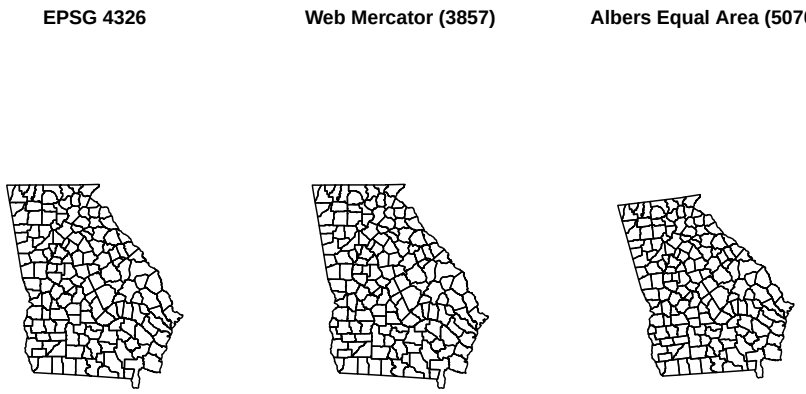
- **EPSG: 4326** is a common geographic (unprojected or long-lat) CRS
- **EPSG: 3857** is also called WGS 84/Web Mercator, and is the dominant CRS used by Google Maps
- **EPSG: 5070** is the code for a projected CRS called Albers Equal Area which has the benefit of representing the visual area of maps in an equal manner.

Once the CRS/projection is clearly defined, you may choose to transform or project the data to a different system. The sf package has another handy function called st_transform() that takes in a spatial object (dtaaset) with one CRS and outputs that object transformed to a new CRS.

```
# This uses the Albers equal area USA,
mvc.aea <- st_transform(mvc, 5070)

# This uses the Web Mercator CRS (EPSG 3857) which is
    just barely different from EPSG 4326
mvc.wm <- st_transform(mvc, 3857)

# Now let's look at them side-by-side
plot(st_geometry(mvc), main = 'EPSG 4326')
plot(st_geometry(mvc.wm), main = 'Web Mercator (3857)')
plot(st_geometry(mvc.aea), main = 'Albers Equal Area
    (5070)')
```



Do you see the difference between the three? In general we will prefer to use 'projected' rather than 'unprojected' data for both visualization and analysis. That means that whenever you bring in a new dataset you will need to check the CRS and project if necessary.

Table 1.2: Vocabulary for Week 1

Term	Definition
Unit of analysis	The unit or object that is measured, analyzed, and about which you wish to make inference. Examples of units of analysis are person, neighborhood, city, state, or hospital.
Attribute data	Nonspatial information about a geographic feature in a GIS, usually stored in a table and linked to the feature by a unique identifier. For example, attributes of a county might include the population size, density, and birth rate for the resident population
Geometry data	Spatial information about a geogrpahic feature. This could include the x,y coordinates for points or for vertices of lines or polygons, or the cell coordinates for raster data
Spatial data model: vector	A coordinate-based data model that represents geographic features as points, lines, and polygons. Each point feature is represented as a single coordinate pair, while line and polygon features are represented as ordered lists of vertices. Attributes are associated with each vector feature, as opposed to a raster data model, which associates attributes with grid cells (see figure below)
Spatial data model: raster	A spatial data model that defines space as an array of equally sized cells arranged in rows and columns, and composed of single or multiple bands. Each cell contains an attribute value and location coordinates. Unlike a vector structure, which stores coordinates explicitly, raster coordinates are contained in the ordering of the matrix. Groups of cells that share the same value represent the same type of geographic feature (see Figure below)
	A reference system that uses latitude and longitude to define the locations of points on the

Chapter 2

Cartography for Epidemiology I

2.1 Learning objectives

Table 2.1: Learning objectives by weekly module

After this module you should be able to...
Design a cartographic representation of epidemiologic data that is consistent with best practices in public health data visualization.
Apply data processing functions to accomplish foundational data management and preparation for spatial epidemiology (e.g. summarize, aggregate, combine, recode, etc)

2.2 Additional Resources

2.3 Spatial Thinking in Epidemiology

Include Vocabulary Concepts and themes as they relate to epidemiology and spatial thinking

2.4 Spatial Analysis in Epidemiology

Include additional vocabulary (if necessary) Mostly includes

Chapter 3

Cartography for Epidemiology II

3.1 Learning objectives

Table 3.1: Learning objectives by weekly module

After this module you should be able to...
Describe potential threats to privacy and research ethics that arise when population health data is represented geographically
Critique spatial epidemiologic literature based on consistency with ethical principles of privacy, avoidance of harm through stigmatization, and balance of benefit and risk

3.2 Additional Resources

3.3 Spatial Thinking in Epidemiology

Include Vocabulary Concepts and themes as they relate to epidemiology and spatial thinking

3.4 Spatial Analysis in Epidemiology

Include additional vocabulary (if necessary) Mostly includes

Chapter 4

Disease Mapping I

4.1 Learning objectives

Table 4.1: Learning objectives by weekly module

After this module you should be able to...
Determine and defend appropriate disease mapping strategies consistent with basic epidemiologic concepts (e.g. study design, sampling strategy, measurement error, and systematic bias)
Create statistically smoothed, age-adjusted disease maps of epidemiologic parameters including SMR, disease risk or rate, and measures of estimate precision/stability

4.2 Additional Resources

4.3 Spatial Thinking in Epidemiology

Content coming soon...

4.4 Spatial Analysis in Epidemiology

Content coming soon...

Chapter 5

Disease Mapping II

5.1 Learning objectives

Table 5.1: Learning objectives by weekly module

After this module you should be able to...
Compare and contrast the operationalization of distance or contiguity in spatial statistics to sociologic and demographic theories of health relevant processes and relationships in space
Apply and justify contrasting definitions of spatial weights matrix in estimation of statistically smoothed disease maps

5.2 Additional Resources

5.3 Spatial Thinking in Epidemiology

Content coming soon...

5.4 Spatial Analysis in Epidemiology

Content coming soon...

Chapter 6

Disease Mapping III

6.1 Learning objectives

Table 6.1: Learning objectives by weekly module

After this module you should be able to...
Discuss the meaning and interpretation of basic functions of spatial point processes including intensity, stationarity, heterogeneity
Produce spatially smoothed estimates of epidemiologic parameters using kernel density estimators for point and polygon data

6.2 Additional Resources

6.3 Spatial Thinking in Epidemiology

Content coming soon...

6.4 Spatial Analysis in Epidemiology

Content coming soon...

Chapter 7

Disease Mapping IV

7.1 Learning objectives

Table 7.1: Learning objectives by weekly module

After this module you should be able to...
Summarize the basic conceptual and statistical benefits of implementing fully Bayesian modeling for epidemiologic disease maping.
Estimate geographically-referenced posteriors from spatial Bayesian model

7.2 Additional Resources

7.3 Spatial Thinking in Epidemiology

Content coming soon...

7.4 Spatial Analysis in Epidemiology

Content coming soon...

Chapter 8

Spatial Structure and Clustering I

8.1 Learning objectives

Table 8.1: Learning objectives by weekly module

After this module you should be able to...
Compare and contrast the statistical, epidemiologic, and policy meaning of geospatial 'clustering' of disease
Calculate and visually summarize global and local tests for spatial clustering

8.2 Additional Resources

8.3 Spatial Thinking in Epidemiology

Content coming soon...

8.4 Spatial Analysis in Epidemiology

Content coming soon...

Chapter 9

Spatial Structure and Clustering II

9.1 Learning objectives

Table 9.1: Learning objectives by weekly module

After this module you should be able to...
Evaluate statistical estimation of spatial clustering in population health to generate epidemiologic hypotheses
Apply spatial scan statistics to epidemiologic data and interpret results

9.2 Additional Resources

9.3 Spatial Thinking in Epidemiology

Content coming soon...

9.4 Spatial Analysis in Epidemiology

Content coming soon...

Chapter 10

Spatial Regression I

10.1 Learning objectives

Table 10.1: Learning objectives by weekly module

After this module you should be able to...
Choose and justify spatial analytic methods that aligns with the epidemiologic research question or objective
Describe the modfiable areal unit problem and discuss strategies for evaluating bias arising from MAUP
Calculate and interpret spatial patterns of residuals from an aspatial multivariable regression model

10.2 Additional Resources

10.3 Spatial Thinking in Epidemiology

Content coming soon...

10.4 Spatial Analysis in Epidemiology

Content coming soon...

Chapter 11

Spatial Regression II

11.1 Learning objectives

Table 11.1: Learning objectives by weekly module

After this module you should be able to...
Explain and relate spatial non-stationarity to epidemiologic concepts of heterogeneity
Use geographically weighted regression to produce and interpret epidemiologic parameters from point and polygon data

11.2 Additional Resources

11.3 Spatial Thinking in Epidemiology

Content coming soon...

11.4 Spatial Analysis in Epidemiology

Content coming soon...

Chapter 12

Spatial Regression III

12.1 Learning objectives

Table 12.1: Learning objectives by weekly module

After this module you should be able to...

12.2 Additional Resources

12.3 Spatial Thinking in Epidemiology

Content coming soon...

12.4 Spatial Analysis in Epidemiology

Content coming soon...

Chapter 13

Wrap-up & Project

13.1 Learning objectives

Table 13.1: Learning objectives by weekly module

After this module you should be able to...

13.2 Additional Resources

13.3 Spatial Thinking in Epidemiology

Content coming soon...

13.4 Spatial Analysis in Epidemiology

Content coming soon...

Part III

Appendices

Chapter 14

Tips for R Notebooks

Additional Resources

- R Markdown Cheatsheet
- Comprehensive guide to using R Markdown
  - Chapter within the R Markdown guide specific to Notebooks

Why R Notebooks?

For most assignments in this course, at least a portion of the deliverable will be a fully-functional, annotated R Notebook. These notebooks are actually a specific case of rmarkdown which itself is a format for creating reproducible documents with interspersed R code, analytic results and text. For example this eBook, and many other resources in this course are created using rmarkdown or related packages such as bookdown.

But as I said, R Notebooks are a specific instance or case of markdown that is incorporated into R Studio and has some nice features for the applied data analyst.

- Notebooks contain text that explains what is happening, interprets findings, or notes areas in need of further exploration
- Notebooks contain functional R code that when run places the results **inside** the Notebook document
- Notebooks work in an interactive mode. This means that as you are coding and working you can see the results in the document. When you save the Notebook the text, the code **and the results** are saved!

So the reason for using Notebooks is that they provide a means for clear annotation and documentation combined with ready reproducitiblity. Reproducibility means that someone else (or a future you!) could come back and get the same result again.

To benefit from the advantages above, I recommend you gain familiarity with the basic (and perhaps the optional) formatting described below. I also recommend you develop a knack for rich annotation and documentation, not just brief (often cryptic) comments that we are all used to writing in SAS and other code! Document what you **plan to do**. Document what you **did**. Document what the **results means**. Document what else **needs to be done**.

What you need to know

This file summarizes both **important** and just a small handful of **optional** functions for effectively using R Notebooks. The **important** functions are those necessary to effectively intersperse narrative text and description communicating what you did and what it means, with clear R code and the resulting output. The **optional** parts are about some simple formatting tools. These **are not** necessary for your homework (our goal is documentation of analytic process not being 'pretty'), but you may find them useful.

Important R Notebook functions

The YAML

```
---
title: "Title of your notebook"
author: "Your Name Here"
date: "Submission date here"
output:
  html_notebook:
    number_sections: yes
    toc: yes
    toc_float: yes
---
```

When you create a new R Notebook or R Markdown file from within R Studio (e.g. via File > New File > R Notebook), a 'YAML' will automatically be created at the top of the script delineated by three dash lines ---. YAML stands for "yet another markup language" and it is a set of instructions about how the finished notebook will look and be structured. You can accept the default YAML structure (of course modifying the title) or copy/paste the YAML from the top

of this script. You can also read more online about additional customizations to the YAML, but none are necessary for this course.

Typing text

The utility of R Notebooks is the ability to more completely document your thinking and your process as you carry out analyses. It is not necessary to be wordy just for the sake of taking up space, but this is an opportunity to clearly delineate goals, steps, data sources, interpretations, etc.

You can just start typing text in the script to serve this purpose. Some text formatting functions are summarized later in this document, and in Cheatsheets and online resources linked to elsehwhere.

Adding `R` Code

R Notebooks let you write R code within your Markdown file, and then run that code, seeing the results appear right under the code (rather than only in the Console, where they usually appear).

There are 2 ways to add a new chunk of R code:

1. Click the green C—Insert button at the top of the editor panel in R Studio. The top option is for R code.

2. Use a keyboard short cut:

    - Mac Command + Shift + I
    - Windows Ctrl + Alt + I

Notice these R code chunks are delineated by three back-ticks (sort of like apostrophers)…these back-ticks are typically on the same key as the tilde (~) on the upper left of most keyboards. The space between the sets of 3 back-ticks is where the R code goes and this is called a code chunk.

Inside these 'chunks' you can type R code just as you would in a regular R script.

When you want to run it you can either:

1. Place your cursor on a line and click Ctrl+enter (Windows) or CMD+Return (Mac), or you can click the Run button at th etop of the editor pane in R Studio.
2. To run all of the code within a chunk click the green Run Current Chunk button at the upper-right of the code chunk.

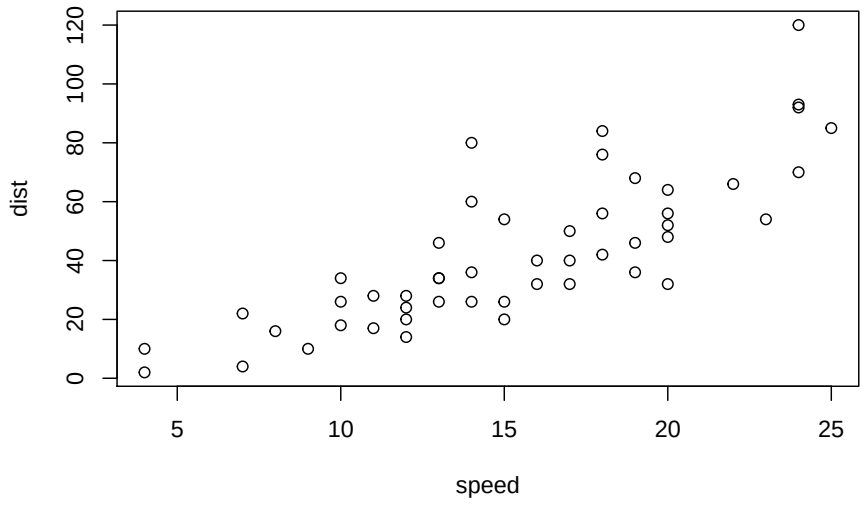
Below is some code and the corresponding results.

```
head(mtcars)
```

Table 14.1

mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
21	6	160	110	3.9	2.62	16.5	0	1	4	4
21	6	160	110	3.9	2.88	17	0	1	4	4
22.8	4	108	93	3.85	2.32	18.6	1	1	4	1
21.4	6	258	110	3.08	3.21	19.4	1	0	3	1
18.7	8	360	175	3.15	3.44	17	0	0	3	2
18.1	6	225	105	2.76	3.46	20.2	1	0	3	1

```
plot(cars)
```



In this way you can iterate through your analytic process…switching between running code, viewing output, documenting in free text.

If you want to see what your current work-in-progress looks like in HTML, you

can click the Preview button at the top of the panel. This will both save your document, and open the Viewer panel.

Making tables

While not required, you may want to summarize data in a table in R Markdown. There are packages devoted to creating tables, but you can create a quick-and-dirty table just using keyboard symbols.

1. First start by making a header row. Separate each column name with a 'pipe' symbol, |
2. Put a continuous line of dashes (−−−−−) under each column name, separating columns with pipe symbole (|)
3. Now type text corresponding to each row and column. Separate columns with pipe (|) and separate rows by carriage return/Enter

So the following text typed directly into the Markdown file (e.g. not inside a code chunk):

```
Column 1   | Column 2  | Column 3
-----------|-----------|----------
Text 1     | Text 2    | Text 3
Next line  | Next line 2 | Next line 3
```

Will produce the following output:

Column 1	Column 2	Column 3
Text 1	Text 2	Text 3
Next line	Next line 2	Next line 3

Workflow

The benefit of Notebooks (slightly different from regular Markdown) is that you can work interactively with your code, seeing results immediately just as you would with a regular script. In contrast a 'regular' Markdown file doesn't run the code until you click 'Knit'.

Here is what I recommend for workflow:

1. Name your script and save it to the desired location. Wherever you save it will become the default Working Directory for any code run from

within the Notebook.

2. Carry out your analysis, inserting code chunks, running them, and documenting them as you go.

3. As a final check of reproducibility (the assurance that your code is self-contained and not dependent on steps you did outside the script) I recommend you always end by clicking the RUN button at the top of the panel. Specifically, choose **Restart R and Run all Chunks**. If there is an error when you did this, then something is missing in your code.

Optional functions

The list of formatting functions is long. I include only a couple I find useful (but not mandatory) here:

Customizing your YAML

While the default YAML is perfectly fine, the YAML at the top of this script includes a few added functions including:

1. Specify a table of contents - this only works if you use headers
2. Specify section numbering
3. Specify that the table of contents should be 'floating' which means that in html it is visible even when you scroll. For PDF rendering, 'float' is not an option.

Simple formatting of your Notebook

Often it is helpful use headers to separate tasks or steps in your code. You can easily create headers using the hastag/pound sign #. Specifically…

- # at the beginning of the line denotes a top-level (level-1) header that will be large and bold.
- ## at the beginning of the line denotes level-2 header
- ### unsurprisingly is a level-3 header!
- Make sure there is a space between the # and the text
- Always leave a blank line (return/enter) between the header text and the 'regular' text.

You can also make numbered or bulleted lists if that is helpful. A line that begins with either an asterisk (∗) or a number will begin a bulleted or numbered list.

Headers are populated into the table of contents, if specified.

Text formatting

The R Markdown Cheatsheets have lots of examples of formatting. Three things that I use more frequently are bold, italics, and numbered or bulleted lists.

Key stroke	Result
∗ italics ∗	italics
∗∗bold∗∗	**bold**

1. Numbered lists start with number, and each line must end with 2 space (or have blank line between).
2. Instead of numbers you can use letters

- Bulleted lists can be initiated with an asterisk or +, and also must have 2 spaces (or blank carriage return) at end of each item.

Final Note

Remember that a final step when you think you are done with a project, is to Click Restart R and Run all Chunks, and then save/preview the Notebook **after** doing this to be sure it is what you expect.