Using Spatial Data with R

Claudia A Engel

Last updated: May 08, 2018

Contents

Prerequisites and Preparations **5**
References . . . 5
Acknowledgements . . . 6

1 Introduction to spatial data in R **7**
1.1 Conceptualizing spatial vector objects in R . . . 7
1.2 Creating a spatial object from a lat/lon table . . . 13
1.3 Loading shape files into R . . . 15
1.4 Raster data in R . . . 20

2 Spatial data manipulation in R **31**
2.1 Attribute Join . . . 31
2.2 Reprojecting . . . 34
2.3 Points in Polygons . . . 36
2.4 Select Polygons by Location . . . 38

3 Making Maps in R **43**
3.1 Choropleth mapping with `spplot` . . . 43
3.2 Plotting simple features (`sf`) with `plot` . . . 49
3.3 Choropleth mapping with `ggplot2` . . . 52
3.4 Adding basemaps with `ggmap` . . . 55
3.5 Choropleth with `tmap` . . . 57
3.6 Web mapping with `leaflet` . . . 59

Prerequisites and Preparations

To get the most out of this workshop you should have:

- a **basic knowledge** of R and/or be familiar with the topics covered in the Introduction to R.
- have a recent version of R and RStudio installed.

Recommended:

- Create a new RStudio project `R-spatial` in a new folder `R-spatial`.
- Create a new folder under `R-spatial` and call it `data`.
- If you have your working directory set to `R-spatial` which contains a folder called `data` you can copy, paste, and run the following lines in R:

```
download.file("https://github.com/cengel/R-spatial/raw/master/data/R-spatial-data.zip", "R-spatial-data
unzip("R-spatial-data.zip", exdir = "data")
```

You can also download the data manually here R-spatial-data.zip and extract them.

- Open up a new R Script file `R-spatial.R` for the code you'll create during the workshop.
- Install and load the following libraries:
  - `sp` (version 1.2-7)
  - `rgdal` (version 1.2-18)
  - `sf` (Mac use binary of version 0.6-1)
  - `raster` (version 2.6-7)
  - `rgeos` (version 0.3-26)
- For the mapping section install and load these additional libraries:
  - `classInt`
  - `RColorBrewer`
  - `broom`
  - `ggplot2`
  - `ggmap`
  - `tmap`
  - `leaflet`

References

Bivand, RS., Pebesma, E., Gómez-Rubio, V. (2013): Applied Spatial Data Analysis with R

Brunsdon, C. and Comber, L. (2015): An Introduction to R for Spatial Analysis and Mapping

Lovelace, R., Nowosad, J., Muenchow. J. (forthcoming): Geocomputation with R

CRAN Task View: Analysis of Spatial Data

Acknowledgements

Some of the materials for this tutorial are adapted from http://datacarpentry.org

Chapter 1

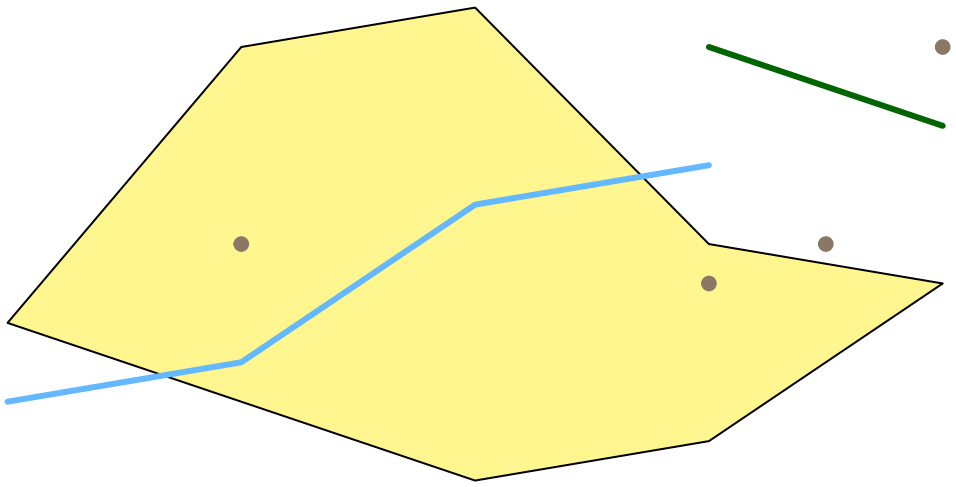
Introduction to spatial data in R

Learning Objectives

- Create point, line, and polygon shapefiles as `sp` and `sf` objects.
- Read shapefiles into `sp` and `sf` objects
- Examine `sp` and `sf` objects
- Read GeoTiff single and multiband into a `raster` object.
- Examine `raster` objects

1.1 Conceptualizing spatial vector objects in R

In vector GIS we deal with, points, lines, and polygons, like so:



Challenge

Discuss with your neighbor: What information do we need to store in order to define points, lines, polygons in geographic space?

There are currently two main approaches in R to handle geographic vector data:

1.1.1 The sp package

The first package to provide classes and methods for spatial data types in R is called `sp`[1]. Development of the `sp` package began in the early 2000s in an attempt to standardize how spatial data would be treated in R and to allow for better interoperability between different analysis packages that use spatial data. The package (first release on CRAN in 2005) provides classes and methods to create *points*, *lines*, *polygons*, and *grids* and to operate on them. About 350 of the spatial analysis packages use the spatial data types that are implemented in `sp` i.e. they "depend" on the `sp` package and many more are indirectly dependent.

The foundational structure for any spatial object in `sp` is the `Spatial` class. It has two "slots" (new-style S4 class objects in R have pre-defined components called slots):

- a **bounding box**
- a **CRS class object** to define the Coordinate Reference System

This basic structure is then extended, depending on the characteristics of the spatial object (point, line, polygon).

To build up a spatial object in `sp` we could follow these steps:

I. Create geometric objects (topology)

Points (which may have 2 or 3 dimensions) are the most basic spatial data objects. They are generated out of either a single coordinate or a set of coordinates, like a two-column matrix or a dataframe with a column for latitude and one for longitude.
Lines are generated out of `Line` objects. A `Line` object is a spaghetti collection of 2D coordinates[2] and is generated out of a two-column matrix or a dataframe with a column for latitude and one for longitude. A `Lines` object is a **list** of one or more `Line` objects, for example all the contours at a single elevation.
Polygons are generated out of `Polygon` objects. A `Polygon` object is a spaghetti collection of 2D coordinates with equal first and last coordinates and is generated out of a two-column matrix or a dataframe with a column for latitude and one for longitude. A `Polygons` object is a **list** of one or more `Polygon` objects, for example islands belonging to the same country.

See here for a very simple example for how to create a `Line` object:

```
ln1 <- Line(matrix(runif(6), ncol=2))
str(ln1)
```

```
#> Formal class 'Line' [package "sp"] with 1 slot
#>   ..@ coords: num [1:3, 1:2] 0.963 0.434 0.826 0.605 0.161 ...
```

See here for a very simple example for how to create a `Lines` object:

```
lns <- Lines(list(ln1), ID = c("hwy1")) # this contains just one Line!
str(lns)
```

```
#> Formal class 'Lines' [package "sp"] with 2 slots
#>   ..@ Lines:List of 1
#>   .. ..$ :Formal class 'Line' [package "sp"] with 1 slot
#>   .. .. .. ..@ coords: num [1:3, 1:2] 0.963 0.434 0.826 0.605 0.161 ...
#>   ..@ ID   : chr "hwy1"
```

II. Create spatial objects `Spatial*` object (`*` stands for Points, Lines, or Polygons).

This step adds the bounding box (automatically) and the slot for the Coordinate Reference System or CRS (which needs to be filled with a value manually). `SpatialPoints` can be directly generated out of the coordinates. `SpatialLines` and `SpatialPolygons` objects are generated using lists of `Lines` or `Polygons` objects respectively (more below).

See here for how to create a `SpatialLines` object:

[1] R Bivand (2011) Introduction to representing spatial objects in R
[2] Coordinates should be of type double and will be promoted if not.

```
sp_lns <- SpatialLines(list(lns))
str(sp_lns)
```

```
#> Formal class 'SpatialLines' [package "sp"] with 3 slots
#>   ..@ lines      :List of 1
#>   .. ..$ :Formal class 'Lines' [package "sp"] with 2 slots
#>   .. .. .. ..@ Lines:List of 1
#>   .. .. .. .. ..$ :Formal class 'Line' [package "sp"] with 1 slot
#>   .. .. .. .. .. .. ..@ coords: num [1:3, 1:2] 0.963 0.434 0.826 0.605 0.161 ...
#>   .. .. .. ..@ ID   : chr "hwy1"
#>   ..@ bbox       : num [1:2, 1:2] 0.434 0.161 0.963 0.685
#>   .. ..- attr(*, "dimnames")=List of 2
#>   .. .. ..$ : chr [1:2] "x" "y"
#>   .. .. ..$ : chr [1:2] "min" "max"
#>   ..@ proj4string:Formal class 'CRS' [package "sp"] with 1 slot
#>   .. .. ..@ projargs: chr NA
```

III. Add attributes (*Optional*:)

Add a data frame with attribute data, which will turn your `Spatial*` object into a `Spatial*DataFrame` object. The points in a `SpatialPoints` object may be associated with a row of attributes to create a `SpatialPointsDataFrame` object. The coordinates and attributes may, but do not have to be keyed to each other using ID values.
`SpatialLinesDataFrame` and `SpatialPolygonsDataFrame` objects are defined using `SpatialLines` and `SpatialPolygons` objects and data frames. The ID fields are here required to match the data frame row names.

See here for how to create a `SpatialLinesDataframe`:

```
dfr <- data.frame(id = "hwy1", use = "road", cars_per_hour = 10) # note how we use the ID from above!
sp_lns_dfr <- SpatialLinesDataFrame(sp_lns, dfr, match.ID = "id")
str(sp_lns_dfr)
```

```
#> Formal class 'SpatialLinesDataFrame' [package "sp"] with 4 slots
#>   ..@ data       :'data.frame':  1 obs. of  3 variables:
#>   .. ..$ id           : Factor w/ 1 level "hwy1": 1
#>   .. ..$ use          : Factor w/ 1 level "road": 1
#>   .. ..$ cars_per_hour: num 10
#>   ..@ lines      :List of 1
#>   .. ..$ :Formal class 'Lines' [package "sp"] with 2 slots
#>   .. .. .. ..@ Lines:List of 1
#>   .. .. .. .. ..$ :Formal class 'Line' [package "sp"] with 1 slot
#>   .. .. .. .. .. .. ..@ coords: num [1:3, 1:2] 0.963 0.434 0.826 0.605 0.161 ...
#>   .. .. .. ..@ ID   : chr "hwy1"
#>   ..@ bbox       : num [1:2, 1:2] 0.434 0.161 0.963 0.685
#>   .. ..- attr(*, "dimnames")=List of 2
#>   .. .. ..$ : chr [1:2] "x" "y"
#>   .. .. ..$ : chr [1:2] "min" "max"
#>   ..@ proj4string:Formal class 'CRS' [package "sp"] with 1 slot
#>   .. .. ..@ projargs: chr NA
```

A number of spatial methods are available for the classes in `sp`. Among the ones I use more frequently are:

function	and what it does
`bbox()`	returns the bounding box coordinates
`proj4string()`	sets or retrieves projection attributes using the CRS object.

Geometry primitives (2D)

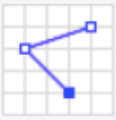
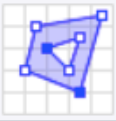
Type	Examples	
Point		`POINT (30 10)`
LineString		`LINESTRING (30 10, 10 30, 40 40)`
Polygon		`POLYGON ((30 10, 40 40, 20 40, 10 20, 30 10))`
		`POLYGON ((35 10, 45 45, 15 40, 10 20, 35 10), (20 30, 35 35, 30 20, 20 30))`

Figure 1.1: Well-Known-Text Geometry primitives (wikipedia)

function	and what it does
`CRS()`	creates an object of class of coordinate reference system arguments
`spplot()`	plots a separate map of all the attributes unless specified otherwise
`coordinates()`	set or retrieve the spatial coordinates. For spatial polygons it returns the centroids.
`over(a, b)`	used for example to retrieve the polygon or grid indices on a set of points
`spsample()`	sampling of spatial points within the spatial extent of objects

1.1.2 The `sf` package

The second package, first released on CRAN in late October 2016, is called `sf`[3]. It implements a formal standard called "Simple Features" that specifies a storage and access model of spatial geometries (point, line, polygon). A feature geometry is called simple when it consists of points connected by straight line pieces, and does not intersect itself. This standard has been adopted widely, not only by spatial databases such as PostGIS, but also more recent standards such as GeoJSON.

If you work with PostGis or GeoJSON you may have come across the WKT (well-known text) format, for example like these:

`sf` implements this standard natively in R. Data are structured and conceptualized very differently from the `sp` approach.

In `sf` spatial objects are stored as a simple data frame with a special column that contains the information for the geographic coordinates. That special column is a list with the same length as the number of rows in the data frame. Each of the individual list elements then can be of any length needed to hold the coordinates that correspond to an individual feature.

To create a spatial object manually the basic steps would be:

I. Create geometric objects (topology)

[3]E. Pebesma & R. Bivand (2016)Spatial data in R: simple features and future perspectives

Multipart geometries (2D)

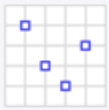
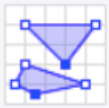
Type	Examples	
MultiPoint		MULTIPOINT ((10 40), (40 30), (20 20), (30 10))
		MULTIPOINT (10 40, 40 30, 20 20, 30 10)
MultiLineString		MULTILINESTRING ((10 10, 20 20, 10 40), (40 40, 30 30, 40 20, 30 10))
MultiPolygon		MULTIPOLYGON (((30 20, 45 40, 10 40, 30 20)), ((15 5, 40 10, 10 20, 5 10, 15 5)))
		MULTIPOLYGON (((40 40, 20 45, 45 30, 40 40)), ((20 35, 10 30, 10 10, 30 5, 45 20, 20 35), (30 20, 20 15, 20 25, 30 20)))

Figure 1.2: Well-Known-Text Multipart geometries (wikipedia)

Geometric objects (simple features) can be created from a numeric vector, matrix or a list with the coordinates. They are called `sfg` objects for Simple Feature Geometry.

See here for an example of how a LINESTRING `sfg` object is created:

```
lnstr_sfg <- st_linestring(matrix(runif(6), ncol=2))
class(lnstr_sfg)
```

```
#> [1] "XY"         "LINESTRING" "sfg"
```

II. Combine all individual single feature objects for the special column.

In order to work our way towards a data frame for all features we create what is called an `sfc` object with all individual features, which stands for Simple Feature Collection. The `sfc` object also holds the bounding box and the projection information.

See here for an example of how a `sfc` object is created:

```
(lnstr_sfc <- st_sfc(lnstr_sfg)) # just one feature here
```

```
#> Geometry set for 1 feature
#> geometry type:  LINESTRING
#> dimension:      XY
#> bbox:           xmin: 0.2956037 ymin: 0.2102633 xmax: 0.9555443 ymax: 0.3919366
#> epsg (SRID):    NA
#> proj4string:    NA
```

```
#> LINESTRING (0.9555443 0.3315015, 0.3090872 0.21...
```

```
class(lnstr_sfc)
```

```
#> [1] "sfc_LINESTRING" "sfc"
```

III. Add attributes.

We now combine the dataframe with the attributes and the simple feature collection. See here how its done.

```
(lnstr_sf <- st_sf(dfr , lnstr_sfc))
```

```
#> Simple feature collection with 1 feature and 3 fields
#> geometry type:  LINESTRING
#> dimension:      XY
#> bbox:           xmin: 0.2956037 ymin: 0.2102633 xmax: 0.9555443 ymax: 0.3919366
#> epsg (SRID):    NA
#> proj4string:    NA
#>     id  use cars_per_hour                       lnstr_sfc
#> 1 hwy1 road            10 LINESTRING (0.9555443 0.331...
```

```
class(lnstr_sf)
```

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

There are many methods available in the `sf` package, to find out use

```
methods(class="sf")
```

```
#>  [1] [                    [[<-                 $<-
#>  [4] aggregate            as.data.frame        cbind
#>  [7] coerce               extent               extract
#> [10] identify             initialize           mask
#> [13] merge                plot                 print
#> [16] rasterize            rbind                show
#> [19] slotsFromS3          st_agr               st_agr<-
#> [22] st_as_sf             st_bbox              st_boundary
#> [25] st_buffer            st_cast              st_centroid
#> [28] st_collection_extract st_convex_hull      st_coordinates
#> [31] st_crs               st_crs<-             st_difference
#> [34] st_geometry          st_geometry<-        st_intersection
#> [37] st_is                st_line_merge        st_node
#> [40] st_point_on_surface  st_polygonize        st_precision
#> [43] st_segmentize        st_set_precision     st_simplify
#> [46] st_snap              st_sym_difference    st_transform
#> [49] st_triangulate       st_union             st_voronoi
#> [52] st_wrap_dateline     st_write             st_zm
#> see '?methods' for accessing help and source code
```

Here are some of the other highlights of `sf` you might be interested in:

- provides **fast** I/O, particularly relevant for large files
- directly reads from and writes to spatial **databases** such as PostGIS
- stay tuned for a new `ggplot` release that will be able to read and plot the `sf` format without the need of conversion to a data frame, like the `sp` format

Note that `sp` and `sf` are not the only way spatial objects are conceptualized in R. Other spatial packages may use their own class definitions for spatial data (for example `spatstat`). Usuallly you can find functions that convert `sp` and increasingly `sf` objects to and from these formats.

Challenge

Similarly to the example above generate a Point object in R. Use both, the `sp` and the `sf` "approach".

1. Create a matrix `pts` of random numbers with two columns and as many rows as you like. These are your points.
2. Create a dataframe `attrib_df` with the same number of rows as your `pts` matrix and a column that holds an attribute. You can make up any attribute.

3. Use the appropriate commands and `pts` to create

- a `SpatialPointsDataFrame` and
- an `sf` object with a gemoetry column of class `sfc_POINT`.

4. Try to subset your spatial object using the attribute you have added and the way you are used to from regular data frames.
5. How do you determine the bounding box of your spatial object?

1.2 Creating a spatial object from a lat/lon table

Often in your research might have a spreadsheet that contains latitude, longitude and perhaps some attribute values. You know how to read the spreadsheet into a data frame with `read.table` or `read.csv`. We can then very easily convert the table into a spatial object in R.

A `SpatialPointsDataFrame` object can be created directly from a table by specifying which columns contain the coordinates. This can be done in one step by using the `coordinates()` function. As mentioned above this function can be used not only to retrieve spatial coordinates but also to set them, which is done in R fashion with:

```
coordinates(myDataframe) <- value
```

`value` can have different forms – in this context needs to be a character vector which specifies the data frame's columns for the longitude and latitude (x,y) coordinates.

If we use this on a data frame it automatically converts the data frame object into a `SpatialPointsDataFrame` object.

An `sf` object can be created from a data frame in the following way. We take advantage of the `st_as_sf()` function which converts any foreign object into an `sf` object. Similarly to above, it requires an argument `coords`, which in the case of point data needs to be a vector that specifies the data frame's columns for the longitude and latitude (x,y) coordinates.

```
my_sf_object <- st_as_sf(myDataframe, coords)
```

Note that `coordinates()` replaces the original data frame, while `st_as_sf()` creates a new object and leaves the original data frame untouched.

We use `read.csv()` to read `philly_homicides.csv` into a dataframe in R and name it `ph_df`.

```
ph_df <- read.csv("data/philly_homicides.csv")
head(ph_df)
```

```
#>   DC_DIST SECTOR DISPATCH_DATE DISPATCH_TIME          LOCATION_BLOCK
#> 1      22      1    2014-09-14      16:00:00 1800 BLOCK W MONTGOMERY
#> 2       1      B    2006-01-14      00:00:00   2000 BLOCK MIFFLIN ST
#> 3       1      B    2006-04-01      16:05:00   S 22ND ST /SNYDER AVE
#> 4       1      B    2006-05-10      11:13:00   2100 BLOCK MC KEAN ST
#> 5       1      E    2006-07-01      12:42:00   2100 BLOCK S HICKS ST
#> 6       1      F    2006-07-09      19:13:00   1800 BLOCK SNYDER AVE
#>   UCR_GENERAL OBJ_ID    TEXT_GENERAL_CODE   POINT_X  POINT_Y
#> 1         100      1  Homicide - Criminal -75.15680 39.98804
#> 2         100      1  Homicide - Criminal -75.17873 39.92801
#> 3         100      1  Homicide - Criminal -75.18275 39.92607
#> 4         100      1  Homicide - Criminal -75.18092 39.92704
#> 5         100      1  Homicide - Criminal -75.17204 39.92463
#> 6         100      1 Homicide - Criminal  -75.17612 39.92517
```

```
class(ph_df)
```

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

We convert the `ph_df` data frame into an `sf` object with `st_as_sf()`

```
ph_sf <- st_as_sf(ph_df , coords = c("POINT_X", "POINT_Y"))
class(ph_sf)
```

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

```
head(ph_sf)
```

```
#> Simple feature collection with 6 features and 8 fields
#> geometry type:  POINT
#> dimension:      XY
#> bbox:           xmin: -75.18275 ymin: 39.92463 xmax: -75.1568 ymax: 39.98804
#> epsg (SRID):    NA
#> proj4string:    NA
#>   DC_DIST SECTOR DISPATCH_DATE DISPATCH_TIME          LOCATION_BLOCK
#> 1      22      1    2014-09-14      16:00:00 1800 BLOCK W MONTGOMERY
#> 2       1      B    2006-01-14      00:00:00   2000 BLOCK MIFFLIN ST
#> 3       1      B    2006-04-01      16:05:00   S 22ND ST /SNYDER AVE
#> 4       1      B    2006-05-10      11:13:00   2100 BLOCK MC KEAN ST
#> 5       1      E    2006-07-01      12:42:00   2100 BLOCK S HICKS ST
#> 6       1      F    2006-07-09      19:13:00   1800 BLOCK SNYDER AVE
#>   UCR_GENERAL OBJ_ID    TEXT_GENERAL_CODE                   geometry
#> 1         100      1  Homicide - Criminal  POINT (-75.1568 39.98804)
#> 2         100      1  Homicide - Criminal POINT (-75.17873 39.92801)
#> 3         100      1  Homicide - Criminal POINT (-75.18275 39.92607)
#> 4         100      1  Homicide - Criminal POINT (-75.18092 39.92704)
#> 5         100      1  Homicide - Criminal POINT (-75.17204 39.92463)
#> 6         100      1 Homicide - Criminal  POINT (-75.17612 39.92517)
```

Alternatively, we convert the `ph_df` data frame into a spatial object with using the `coordinates` function and check with `class(ph_df)`again to examine which object class the table belongs to now.

```
coordinates(ph_df) <- c("POINT_X", "POINT_Y")
class(ph_df) # !!
```

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

A brief, but important word about projection:

Note that both the `SpatialPointsDataFrame` and the `sf` POINTS object you just created **do not** have a projection defined. It is ok to plot, but be aware that for any meaningful spatial operation you will need to define a projection.

This is how it's done:

```
is.projected(ph_df) # see if a projection is defined
```

```
#> [1] NA
```

```
proj4string(ph_df) <- CRS("+proj=longlat +ellps=WGS84 +datum=WGS84 +no_defs") # this is WGS84
is.projected(ph_df) # voila! hm. wait a minute..
```

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

```
# For the `sf` object you want to use
st_crs(ph_sf)
```

```
#> Coordinate Reference System: NA
```

```
st_crs(ph_sf) <- 4326 # we can use EPSG as numeric here
st_crs(ph_sf)
```

```
#> Coordinate Reference System:
#>   EPSG: 4326
#>   proj4string: "+proj=longlat +datum=WGS84 +no_defs"
```

We will save this for later use.

```
st_write(ph_sf, "data/PhillyHomicides", driver = "ESRI Shapefile")

# to save out using writeOGR from rgdal
# note that we need to save the ph_df, which we converted to sp object!
# writeOGR(ph_df, "data/PhillyHomicides", "PhillyHomcides", driver = "ESRI Shapefile")
```

1.3 Loading shape files into R

1.3.1 How to work with `rgdal`

In order to read spatial data into R and turn them into `Spatial*` family objects we rely on the `rgdal` package. It provides us direct access to the powerful GDAL library from within R.

We can read in and write out spatial data using:

```
readOGR() and writeOGR() (for vector)
readGDAL() and writeGDAL() (for raster/grids)
```

The parameters provided for each function vary depending on the exact spatial file type you are reading. We will take an ESRI shapefile as an example. A shapefile - as you know - consists of various files of the same name, but with different extensions. They should all be in one directory and that is what R expects.

When reading in a shapefile, `readOGR()` requires the following two arguments:

```
datasource name (dsn)  # the path to the folder that contains the files
                       # this is a path to the folder, not a filename!
layer name (layer)     # the shapefile name WITHOUT extension
                       # this is not a path but just the name of the file!
```

Setting these arguments correctly can be cause of much headache for beginners, so let me spell it out:

- Firstly, you obviously need to know the name of shapefile.
- Secondly, you need to know the name and location of the folder that contains all the shapefile parts.
- Lastly, `readOGR` only reads the file and dumps it on your screen. But similarly when reading csv tables you want to actually work with the file, so you need to assign it to an R object.

Now let's do this.

We load the `rgdal` package and read `PhillyTotalPopHHinc` into an object called `philly`. We can also examine the object, for example with `summary()` or `class()`.

```
library(rgdal)
philly_sp <- readOGR("data/Philly/", "PhillyTotalPopHHinc")
```

```
#> OGR data source with driver: ESRI Shapefile
#> Source: "/Users/cengel/Anthro/R_Class/R_Workshops/R-spatial/data/Philly", layer: "PhillyTotalPopHHinc"
#> with 384 features
#> It has 17 fields
```

```
# side note: unlike read.csv readOGR does not understand the ~ as valid element of a path. This (on Mac
# philly <- readOGR("~/Desktop/R-data-viz/data/Philly/", "PhillyTotalPopHHinc")
summary(philly_sp)
```

```
#> Object of class SpatialPolygonsDataFrame
#> Coordinates:
#>         min       max
#> x 1739496.5 1764029.7
#> y  457343.7  490544.9
#> Is projected: TRUE
#> proj4string :
#> [+proj=aea +lat_1=29.5 +lat_2=45.5 +lat_0=37.5 +lon_0=-96 +x_0=0
#> +y_0=0 +ellps=GRS80 +units=m +no_defs]
#> Data attributes:
#>  STATEFP10 COUNTYFP10   TRACTCE10          GEOID10        NAME10
#>  42:384    101:384    000100 :  1   42101000100:  1   1      :  1
#>                       000200 :  1   42101000200:  1   10.01  :  1
#>                       000300 :  1   42101000300:  1   10.02  :  1
#>                       000401 :  1   42101000401:  1   100    :  1
#>                       000402 :  1   42101000402:  1   101    :  1
#>                       000500 :  1   42101000500:  1   102    :  1
#>                       (Other):378   (Other)    :378   (Other):378
#>               NAMELSAD10   MTFCC10    FUNCSTAT10    ALAND10
#>  Census Tract 1    :  1   G5020:384   S:384      Min.   :   99958
#>  Census Tract 10.01:  1                         1st Qu.:  397457
#>  Census Tract 10.02:  1                         Median :  600362
#>  Census Tract 100  :  1                         Mean   :  904482
#>  Census Tract 101  :  1                         3rd Qu.:  955452
#>  Census Tract 102  :  1                         Max.   :17228698
#>  (Other)           :378
#>     AWATER10             INTPTLAT10         INTPTLON10
#>  Min.   :      0   +39.8798897:  1   -074.9667387:  1
#>  1st Qu.:      0   +39.8898768:  1   -074.9702250:  1
#>  Median :      0   +39.8904539:  1   -074.9742967:  1
#>  Mean   :  58045   +39.8988328:  1   -074.9781805:  1
#>  3rd Qu.:      0   +39.9024981:  1   -074.9789137:  1
#>  Max.   :3463789   +39.9051799:  1   -074.9805151:  1
#>                    (Other)    :378   (Other)     :378
#>            GISJOIN      Shape_area         Shape_len        medHHinc
#>  G4201010000100:  1   Min.   :  105512   Min.   : 1321   Min.   :  9286
#>  G4201010000200:  1   1st Qu.:  398512   1st Qu.: 2734   1st Qu.: 24946
#>  G4201010000300:  1   Median :  601061   Median : 3426   Median : 35365
#>  G4201010000401:  1   Mean   :  962528   Mean   : 4239   Mean   : 38509
#>  G4201010000402:  1   3rd Qu.:  966639   3rd Qu.: 4603   3rd Qu.: 49474
#>  G4201010000500:  1   Max.   :20692491   Max.   :30881   Max.   :130139
#>  (Other)       :378                                      NA's   :9
#>     totalPop
#>  Min.   :   0
#>  1st Qu.:2799
#>  Median :3914
#>  Mean   :3974
#>  3rd Qu.:5111
#>  Max.   :8322
#>
```

```
class(philly_sp)
```

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

Let's check out the attribute data and plot a subset of polygons with a median household income (`medHHinc`) of over 60000 on top of the plot of the entire city.

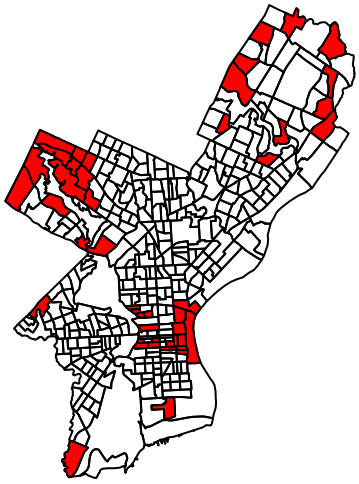
```
names(philly_sp)
```

```
#>  [1] "STATEFP10"  "COUNTYFP10" "TRACTCE10"  "GEOID10"    "NAME10"
#>  [6] "NAMELSAD10" "MTFCC10"    "FUNCSTAT10" "ALAND10"    "AWATER10"
#> [11] "INTPTLAT10" "INTPTLON10" "GISJOIN"    "Shape_area" "Shape_len"
#> [16] "medHHinc"   "totalPop"
```

```
head(philly_sp)
```

```
#>   STATEFP10 COUNTYFP10 TRACTCE10     GEOID10 NAME10          NAMELSAD10
#> 0        42        101    036301 42101036301 363.01 Census Tract 363.01
#> 1        42        101    036400 42101036400    364    Census Tract 364
#> 2        42        101    036600 42101036600    366    Census Tract 366
#> 3        42        101    034803 42101034803 348.03 Census Tract 348.03
#> 4        42        101    034702 42101034702 347.02 Census Tract 347.02
#> 5        42        101    036202 42101036202 362.02 Census Tract 362.02
#>   MTFCC10 FUNCSTAT10 ALAND10 AWATER10  INTPTLAT10   INTPTLON10
#> 0   G5020          S 2322732    66075 +40.0895349 -074.9667387
#> 1   G5020          S 4501110     8014 +40.1127747 -074.9789137
#> 2   G5020          S 1004313  1426278 +39.9470272 -075.1404472
#> 3   G5020          S 1271533     8021 +40.0619427 -075.0023705
#> 4   G5020          S 1016206        0 +40.0570427 -075.0283288
#> 5   G5020          S 1116115     2329 +40.0838623 -074.9781805
#>          GISJOIN Shape_area Shape_len medHHinc totalPop
#> 0 G4201010036301    2388806  6850.541    54569     3695
#> 1 G4201010036400    4509124 10567.331       NA      703
#> 2 G4201010036600    2430591  9256.983   130139     1643
#> 3 G4201010034803    1279556  4927.632    56667     4390
#> 4 G4201010034702    1016207  5919.885    69981     3807
#> 5 G4201010036202    1118443  5899.099    61513     6138
```

```
plot(philly_sp)
philly_sp_rich <- subset(philly_sp, medHHinc > 60000)
plot(philly_sp_rich, add=T, col="red")
```



GDAL supports over 200 raster formats and vector formats. Use `ogrDrivers()` and `gdalDrivers()` (without arguments) to find out which formats your `rgdal` install can handle.

1.3.2 How to do this in `sf`

`sf` also relies on GDAL, but we don't need to load a separate R library to read data in. We can use `st_read()`, which simply takes the path of the directory with the shapefile as argument.

So let's do the same as above using the `sf` package.

```
# read in
philly_sf <- st_read("data/Philly/")
```

```
#> Reading layer `PhillyTotalPopHHinc' from data source `/Users/cengel/Anthro/R_Class/R_Workshops/R-spatia
#> Simple feature collection with 384 features and 17 fields
#> geometry type:  MULTIPOLYGON
#> dimension:      XY
#> bbox:           xmin: 1739497 ymin: 457343.7 xmax: 1764030 ymax: 490544.9
#> epsg (SRID):    NA
#> proj4string:   +proj=aea +lat_1=29.5 +lat_2=45.5 +lat_0=37.5 +lon_0=-96 +x_0=0 +y_0=0 +ellps=GRS80 +unit
```

```
# take a look at what we've got
names(philly_sf)
```

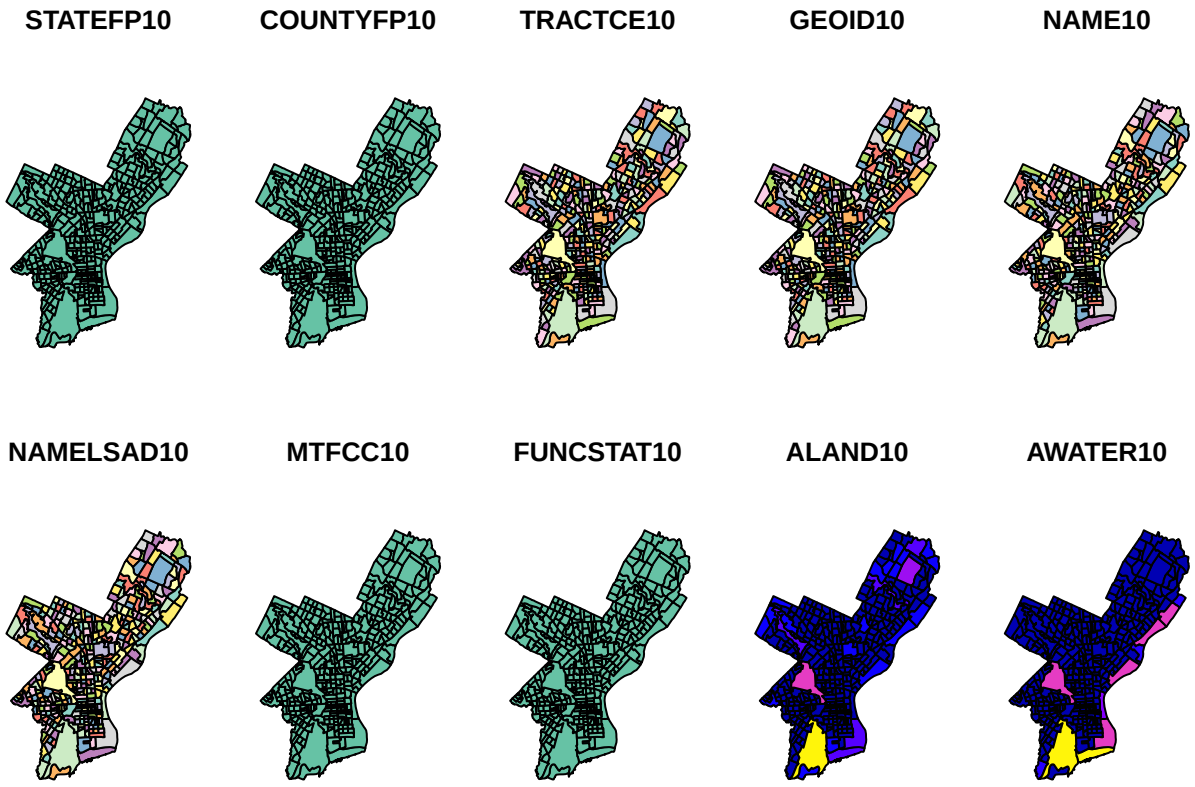
```
#>  [1] "STATEFP10"  "COUNTYFP10" "TRACTCE10"  "GEOID10"    "NAME10"
#>  [6] "NAMELSAD10" "MTFCC10"    "FUNCSTAT10" "ALAND10"    "AWATER10"
#> [11] "INTPTLAT10" "INTPTLON10" "GISJOIN"    "Shape_area" "Shape_len"
#> [16] "medHHinc"   "totalPop"   "geometry"
```

```
# note the added geometry column, as compared to:
names(philly_sp)
```

```
#>  [1] "STATEFP10"  "COUNTYFP10" "TRACTCE10"  "GEOID10"    "NAME10"
#>  [6] "NAMELSAD10" "MTFCC10"    "FUNCSTAT10" "ALAND10"    "AWATER10"
#> [11] "INTPTLAT10" "INTPTLON10" "GISJOIN"    "Shape_area" "Shape_len"
#> [16] "medHHinc"   "totalPop"
```

```
# plot works differently here:
plot(philly_sf)
```

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




```
# to do the same as above we need to directly print the geometry column
st_geometry(philly_sf)        # use this method to retreive geometry
```

```
#> Geometry set for 384 features
#> geometry type:  MULTIPOLYGON
#> dimension:      XY
#> bbox:           xmin: 1739497 ymin: 457343.7 xmax: 1764030 ymax: 490544.9
#> epsg (SRID):    NA
#> proj4string:   +proj=aea +lat_1=29.5 +lat_2=45.5 +lat_0=37.5 +lon_0=-96 +x_0=0 +y_0=0 +ellps=GRS80 +unit
#> First 5 geometries:

#> MULTIPOLYGON (((1763647 484837.3, 1763473 48519...

#> MULTIPOLYGON (((1761348 489213.5, 1761372 48918...

#> MULTIPOLYGON (((1752887 468814.9, 1752808 46863...

#> MULTIPOLYGON (((1761207 482777.8, 1761634 48258...

#> MULTIPOLYGON (((1759301 482266.6, 1759120 48186...
```

```
plot(st_geometry(philly_sf))

# subset the familar way
philly_sf_rich <- subset(philly_sf, medHHinc > 60000)
plot(st_geometry(philly_sf_rich), add=T, col="red")
```

1.4 Raster data in R

Raster files, as you might know, have a much more compact data structure than vectors. Because of their regular structure the coordinates do not need to be recorded for each pixel or cell in the rectangular extent. A raster is defined by:

- a CRS
- coordinates of its origin
- a distance or cell size in each direction
- a dimension or numbers of cells in each direction
- an array of cell values

Given this structure, coordinates for any cell can be computed and don't need to be stored.

The `raster` package[4] is a major extension of spatial data classes to access large rasters and in particular to process very large files. It includes object classes for `RasterLayer`, `RasterStacks`, and `RasterBricks`, functions for converting among these classes, and operators for computations on the raster data. Conversion from `sp` type objects into `raster` type objects is possible.

If we wanted to do create a raster object from scratch we would do the following:

```
# specify the RasterLayer with the following parameters:
# - minimum x coordinate (left border)
# - minimum y coordinate (bottom border)
# - maximum x coordinate (right border)
# - maximum y coordinate (top border)
# - resolution (cell size) in each dimension
r <- raster(xmn=-0.5, ymn=-0.5, xmx=4.5, ymx=4.5, resolution=c(1,1))
r
```

```
#> class       : RasterLayer
#> dimensions  : 5, 5, 25  (nrow, ncol, ncell)
#> resolution  : 1, 1  (x, y)
#> extent      : -0.5, 4.5, -0.5, 4.5  (xmin, xmax, ymin, ymax)
#> coord. ref. : +proj=longlat +datum=WGS84 +ellps=WGS84 +towgs84=0,0,0
```

Note that this raster object **has a CRS defined!** If the crs argument is missing when creating the Raster object, the x coordinates are within -360 and 360 and the y coordinates are within -90 and 90, the WGS84 projection is used by default!

Good to know.

To add some values to the cells we could the following.

```
class(r)
```

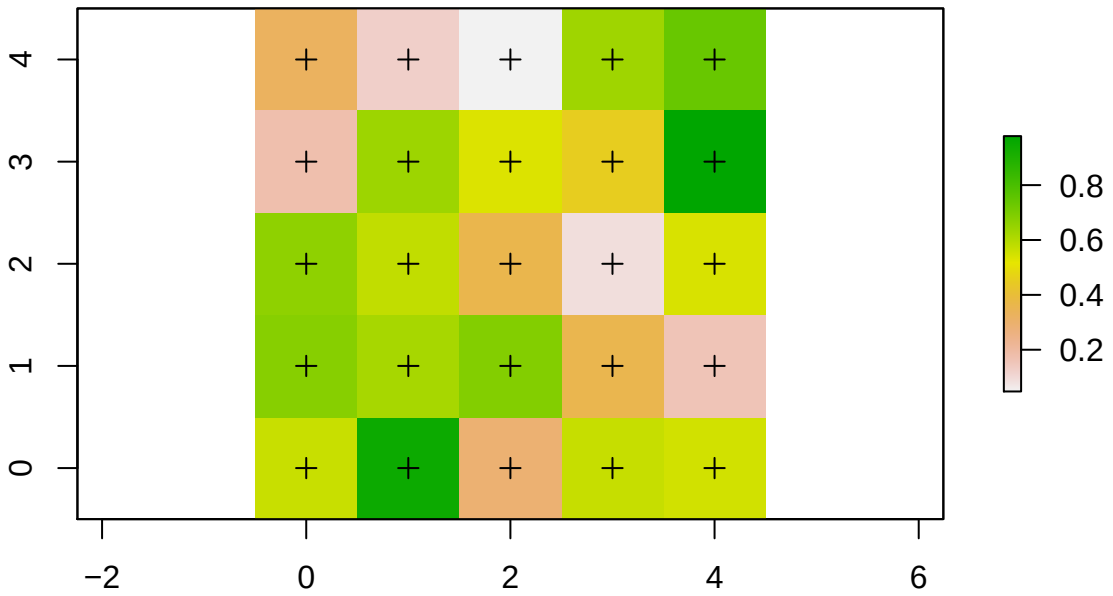
[4]The `geo_join()` command from the `tigris` package also provides a convenient way to merge a data frame to a spatial data frame.

```
#> [1] "RasterLayer"
#> attr(,"package")
#> [1] "raster"
```

```
r <- setValues(r, runif(25))
class(r)
```

```
#> [1] "RasterLayer"
#> attr(,"package")
#> [1] "raster"
```

```
plot(r); points(coordinates(r), pch=3)
```



(See the `rasterVis` package for more advanced plotting of `Raster*` objects.)

RasterLayer objects can also be created from a matrix.

```
class(volcano)
```

```
#> [1] "matrix"
```

```
volcano.r <- raster(volcano)
class(volcano.r)
```

```
#> [1] "RasterLayer"
#> attr(,"package")
#> [1] "raster"
```

And to read in a raster file we can use the `raster()` function. This raster is generated as part of the NEON Harvard Forest field site.

```
library(raster)
HARV <- raster("data/HARV_RGB_Ortho.tif")
```

Typing the name of the object will give us what's in there:

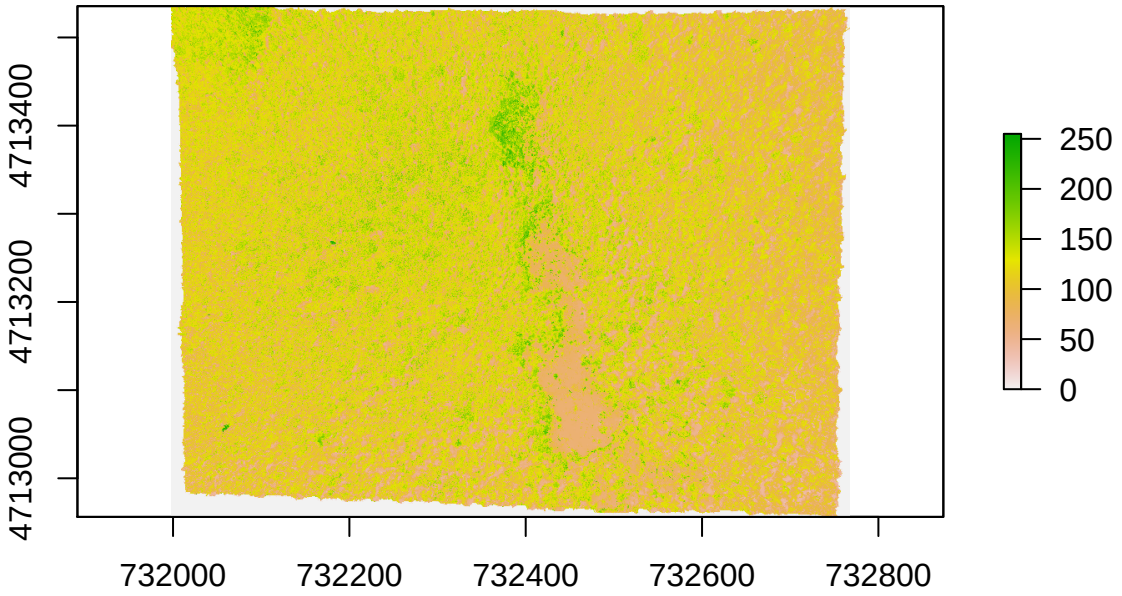
```
HARV
```

```
#> class       : RasterLayer
#> band        : 1  (of  3  bands)
#> dimensions  : 2317, 3073, 7120141  (nrow, ncol, ncell)
#> resolution  : 0.25, 0.25  (x, y)
```

```
#> extent      : 731998.5, 732766.8, 4712956, 4713536  (xmin, xmax, ymin, ymax)
#> coord. ref. : +proj=utm +zone=18 +datum=WGS84 +units=m +no_defs +ellps=WGS84 +towgs84=0,0,0
#> data source : /Users/cengel/Anthro/R_Class/R_Workshops/R-spatial/data/HARV_RGB_Ortho.tif
#> names       : HARV_RGB_Ortho
#> values      : 0, 255  (min, max)
```

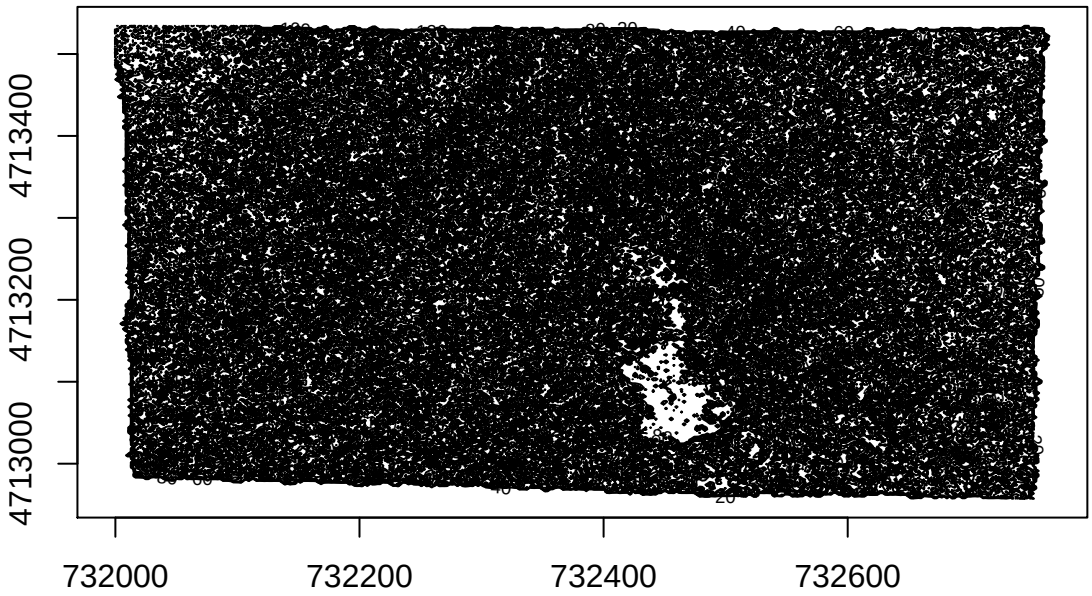
We can plot it like this:

```
plot(HARV)
```



And we can apply functions, like this one:

```
contour(HARV)
```



We can find out about the Coordinate Reference System with this:

```
crs(HARV)
```

```
#> CRS arguments:
#>  +proj=utm +zone=18 +datum=WGS84 +units=m +no_defs +ellps=WGS84
```

```
#> +towgs84=0,0,0
```

See what you can do with such an object:

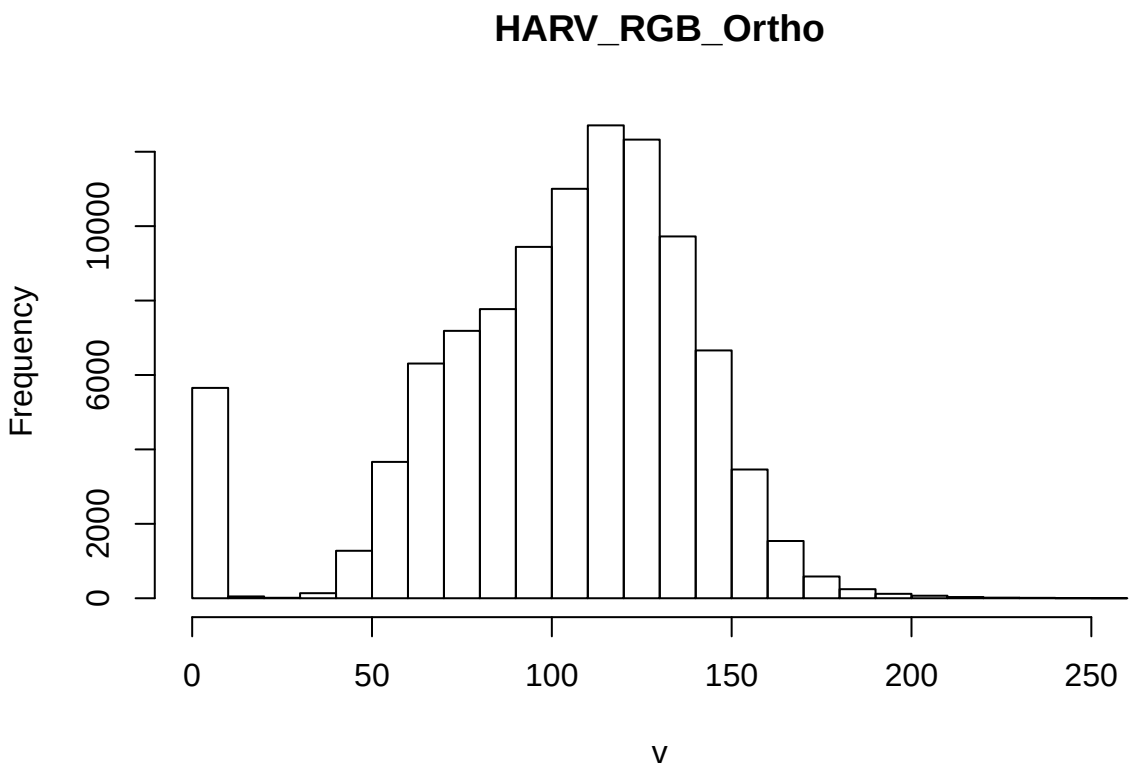
```
methods(class=class(HARV))
```

```
#>   [1] !                 !=                [                 [[
#>   [5] [<-               %in%              ==                $
#>   [9] $<-               addLayer          aggregate         all.equal
#>  [13] area              Arith             as.array          as.data.frame
#>  [17] as.factor         as.integer        as.list           as.logical
#>  [21] as.matrix         as.raster         as.vector         asFactor
#>  [25] atan2             bandnr            barplot           bbox
#>  [29] boundaries        boxplot           brick             buffer
#>  [33] calc              cellStats         clamp             click
#>  [37] clump             coerce            colSums           Compare
#>  [41] contour           coordinates       corLocal          cover
#>  [45] crop              crosstab          cut               cv
#>  [49] density           dim               dim<-             direction
#>  [53] disaggregate      distance          extend            extent
#>  [57] extract           flip              focal             freq
#>  [61] getValues         getValuesBlock    getValuesFocal    gridDistance
#>  [65] head              hist              image             interpolate
#>  [69] intersect         is.factor         is.finite         is.infinite
#>  [73] is.na             is.nan            isLonLat          KML
#>  [77] labels            layerize          length            levels
#>  [81] levels<-          lines             localFun          log
#>  [85] Logic             mask              match             Math
#>  [89] Math2             maxValue          mean              merge
#>  [93] minValue          modal             mosaic            names
#>  [97] names<-           ncell             ncol              nlayers
#> [101] nrow              origin            origin<-          overlay
#> [105] persp             plot              predict           print
#> [109] proj4string       proj4string<-     quantile          raster
#> [113] rasterize         readAll           readStart         readStop
#> [117] reclassify        res               resample          RGB
#> [121] rotate            rowSums           sampleRandom      sampleRegular
#> [125] sampleStratified  scale             select            setMinMax
#> [129] setValues         shift             show              spplot
#> [133] stack             stackSelect       subs              subset
#> [137] Summary           summary           t                 tail
#> [141] text              trim              unique            update
#> [145] values            values<-          Which             which.max
#> [149] which.min         writeRaster       writeStart        writeStop
#> [153] writeValues       xmax              xmin              xres
#> [157] ymax              ymin              yres              zonal
#> [161] zoom
#> see '?methods' for accessing help and source code
```

We can explore the distribution of values contained within our raster using the hist() function which produces a histogram. Histograms are often useful in identifying outliers and bad data values in our raster data.

```
hist(HARV)
```

```
#> Warning in .hist1(x, maxpixels = maxpixels, main = main, plot = plot, ...):
#> 1% of the raster cells were used. 100000 values used.
```



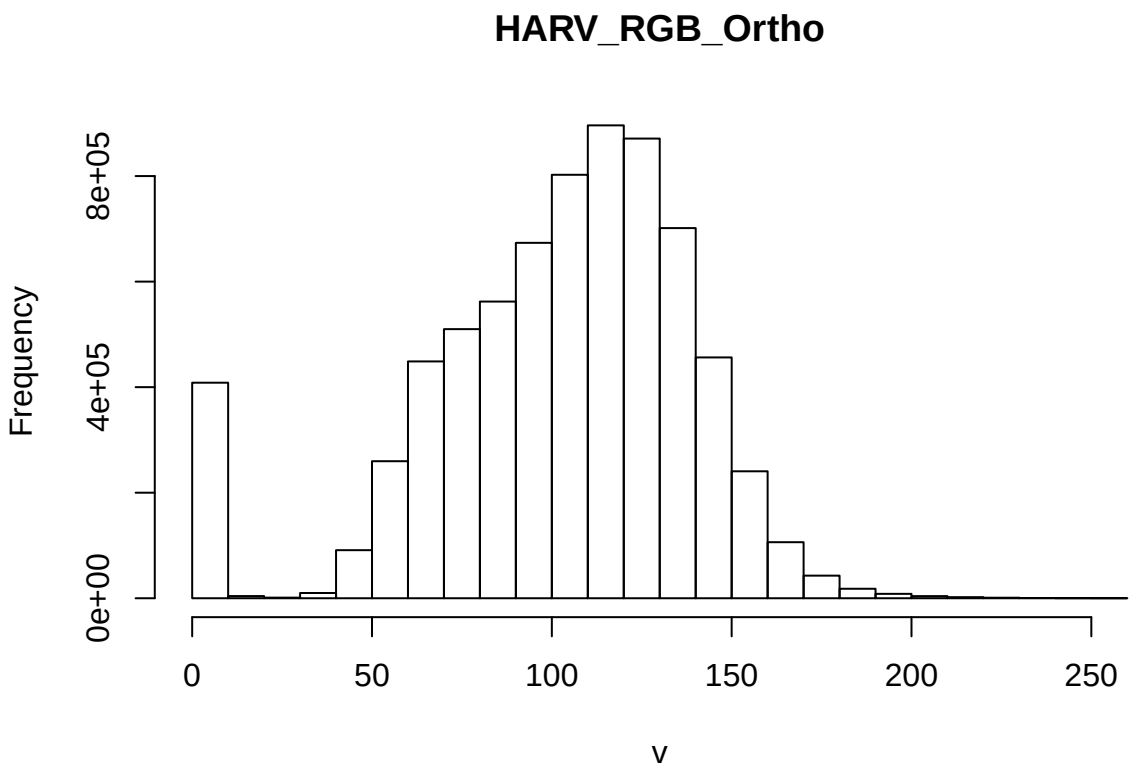
Notice that an warning message is thrown when R creates the histogram.

This warning is caused by the default maximum pixels value of 100,000 associated with the hist function. This maximum value is to ensure processing efficiency as our data become larger!

```
ncell(HARV)
```

```
#> [1] 7120141
```

```
hist(HARV,
     maxpixels = ncell(HARV))
```



At times it may be useful to explore raster metadata before loading them into R. This can be done with:

```
GDALinfo("path-to-raster-here")
```

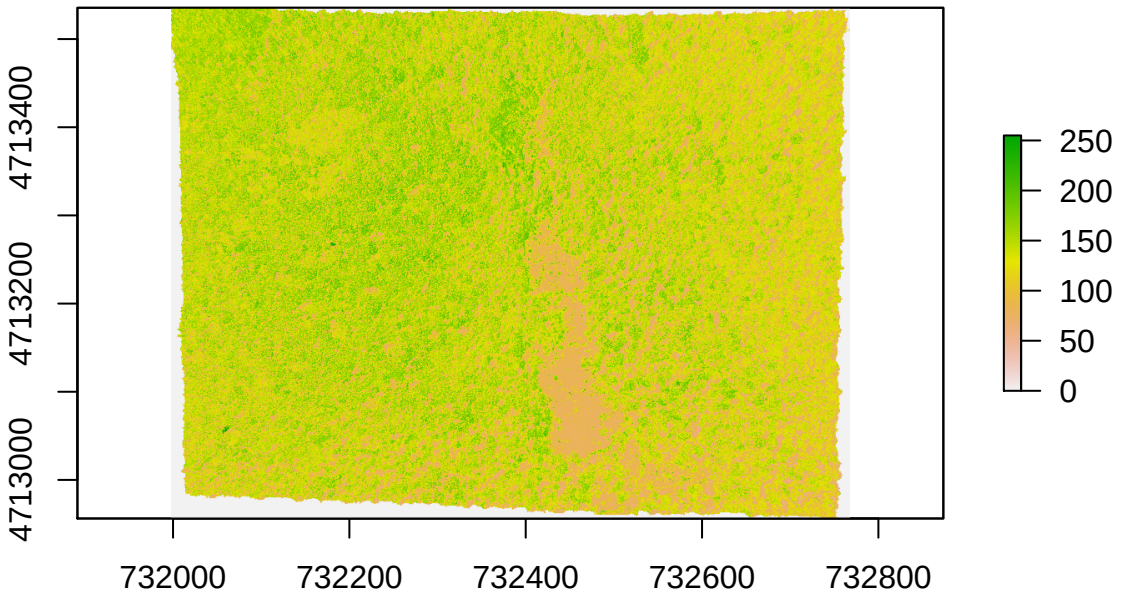
A raster dataset can contain one or more bands. We can view the number of bands in a raster using the `nlayers()` function.

```
nlayers(HARV)
```

```
#> [1] 1
```

We can use the `raster()` function to import one single band from a *single* **OR** from a *multi-band* raster. For multi-band raster, we can specify which band we want to read in.

```
HARV_Band2 <-
  raster("data/HARV_RGB_Ortho.tif", band = 2)

plot(HARV_Band2)
```



To bring in all bands of a multi-band raster, we use the `stack()` function.

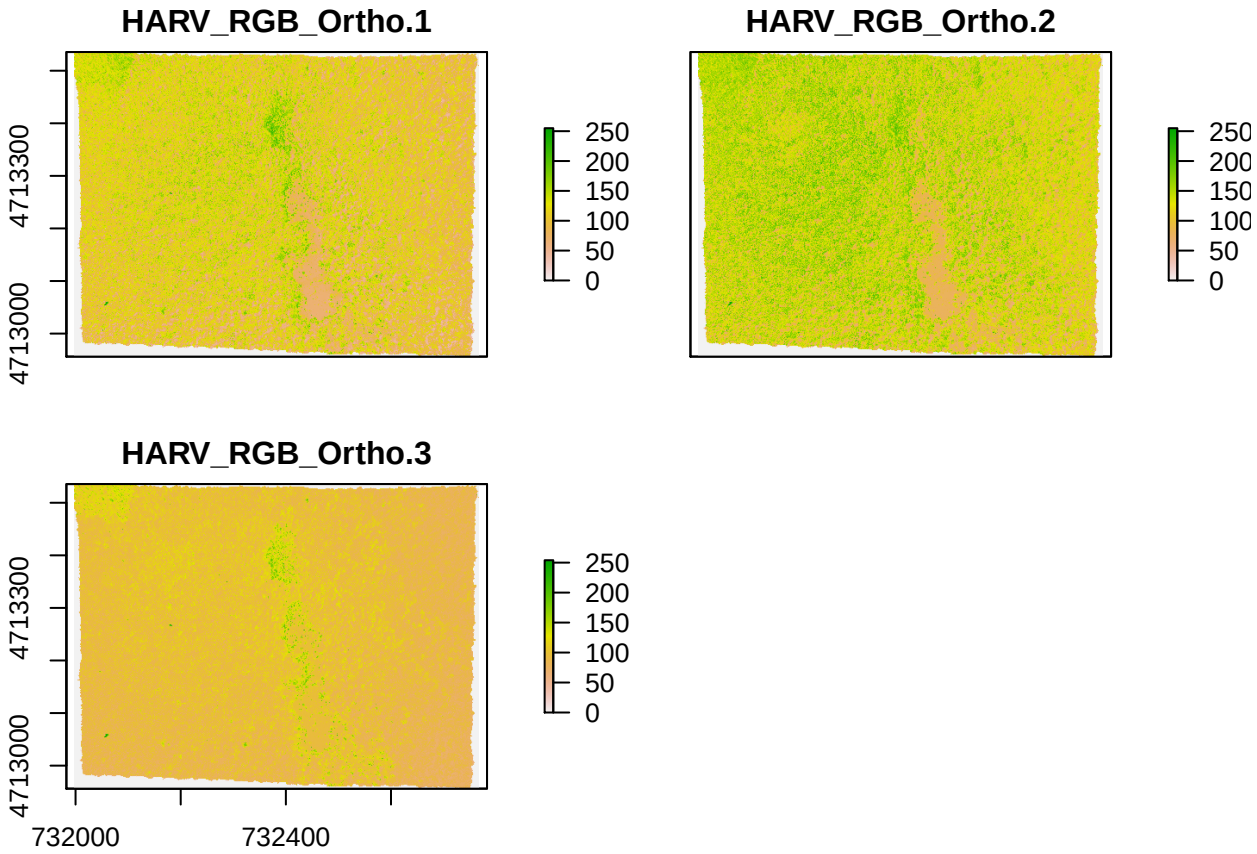
```
HARV_stack <-
  stack("data/HARV_RGB_Ortho.tif")

# how many layers?
nlayers(HARV)
```

```
#> [1] 1
```

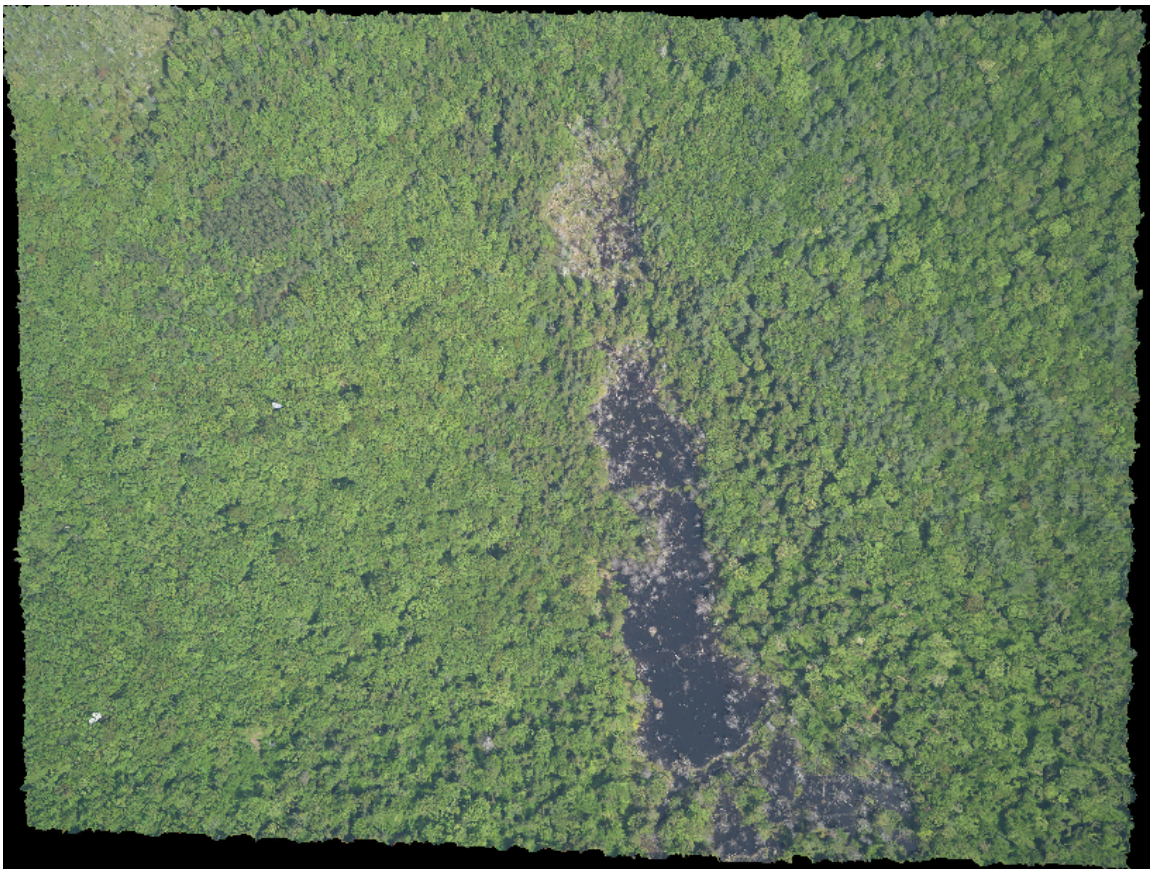
```
# view attributes of stack object
HARV_stack
```

```
#> class       : RasterStack
#> dimensions  : 2317, 3073, 7120141, 3  (nrow, ncol, ncell, nlayers)
#> resolution  : 0.25, 0.25  (x, y)
#> extent      : 731998.5, 732766.8, 4712956, 4713536  (xmin, xmax, ymin, ymax)
#> coord. ref. : +proj=utm +zone=18 +datum=WGS84 +units=m +no_defs +ellps=WGS84 +towgs84=0,0,0
#> names       : HARV_RGB_Ortho.1, HARV_RGB_Ortho.2, HARV_RGB_Ortho.3
#> min values  :                 0,                 0,                 0
#> max values  :               255,               255,               255
```

```
# what happens when we plot?
plot(HARV_stack)
```




```
# if we know that it is an RGB multiband raster we can plot them all in one
plotRGB(HARV_stack)
```



1.4.1 RasterStack vs RasterBrick in R

The R `RasterStack` and `RasterBrick` object types can both store multiple bands. However, how they store each band is different. The bands in a `RasterStack` are stored as links to raster data that is located somewhere on our computer. A `RasterBrick` contains all of the objects stored within the actual R object. Since in the `RasterBrick`, all of the bands are stored within the actual object its object size is much larger than the `RasterStack` object.

In most cases, we can work with a `RasterBrick` in the same way we might work with a `RasterStack`. However, a `RasterBrick` is often more efficient and faster to process - which is important when working with larger files.

We can turn a `RasterStack` into a `RasterBrick` in R by using `brick(StackName)`. Use the `object.size()` function to compare stack and brick R objects.

```
object.size(HARV_stack)
```

```
#> 41712 bytes
```

```
HARV_brick <- brick(HARV_stack)
object.size(HARV_brick)
```

```
#> 170896376 bytes
```

A simple grid can be built like this:

```
# specify the grid topology with the following parameters:
# - the smallest coordinates for each dimension, here: 0,0
# - cell size in each dimension, here: 1,1
```

```
# - number of cells in each dimension, here: 5,5
gtopo <- GridTopology(c(0,0), c(1,1), c(5,5)) # create the grid
datafr <- data.frame(runif(25)) # make up some data
SpGdf <- SpatialGridDataFrame(gtopo, datafr) # create the grid data frame
summary(SpGdf)
```

Chapter 2

Spatial data manipulation in R

Learning Objectives

- Join attribute data to a polygon vector file
- Reproject a vector file
- Select polygons of a vector by location

In this section we will look at some libraries and commands that allow us to process spatial data in R and perform a few examples of commonly used operations.

2.1 Attribute Join

An attribute join brings tabular data into a geographic context. It refers to the process of joining data in tabular format to data in a format that holds the geometries (polygon, line, or point).

If you have done attribute joins of shapefiles in GIS software like *ArcGIS* or *QGis* you know that you need a **unique identifier** in both the attribute table of the shapefile and the table to be joined.

In order to combine a `Spatial*Dataframe` with another table (which would be a dataframe in R) we do exactly the same. We have a `Spatial*Dataframe`[1] that contains the geometries and an identifying index variable for each. We combine it with a dataframe, that includes the same index variable with additional variables.

The `sp` package has a `merge` command which extends the base `merge` command to works with `Spatial*` objects as argument.

Assume we have:

- a `SpatialPolygonObject` named *worldCountries*, and
- a dataframe called *countryData* with the attribute data to join

where:

- *"id-number"* is the colum that contains the unique identifier in *worldCountries*, and
- *"countryID"* is the column that contains the unique identifier in *countryData*.

We would then say:

```
library(sp) # make sure that is loaded
worldCountries <- merge(worldCountries, countryData, by.x = "id-number", by.y = "countryID")
```

[1] Per the ESRI specification a shapefile always has an attribute table, so when we read it into R with the `readOGR` command from the `sp` package it automatically becomes a `Spatial*Dataframe` and the attribute table becomes the dataframe.

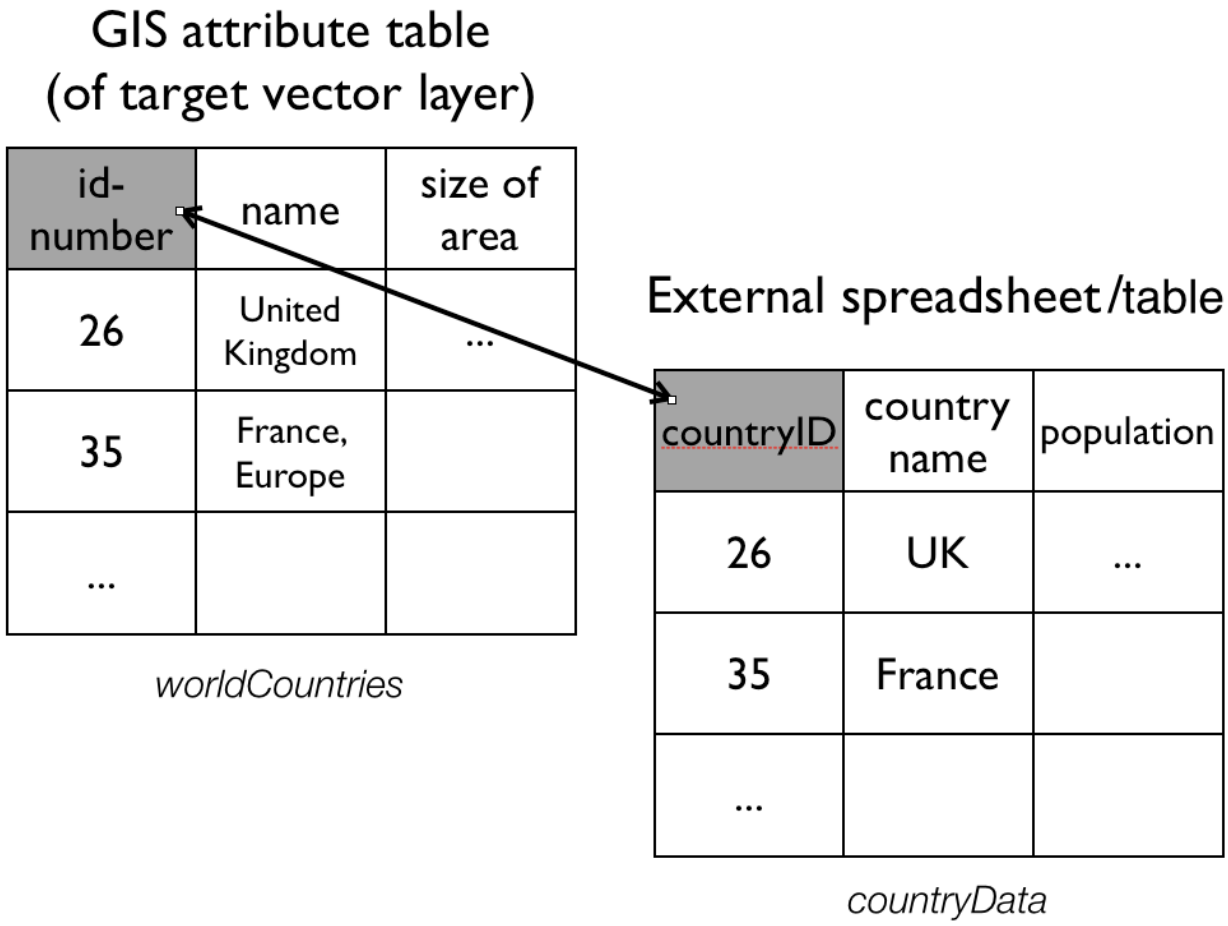


Figure 2.1: Attribute Join of countryData table to worldCountries using unique ID variables

(You may come across alternative suggestions for joins that operate on the data slot `@data` of the Spatial* object. While they may work, we don't suggest them here, as good practice suggests not to use the slot explicitly if at all possible.)

Load the CSV table `PhiladelphiaEduAttain.csv` into a dataframe in R and name it `ph_edu`.

```
ph_edu <- read.csv("data/PhiladelphiaEduAttain.csv")
names(ph_edu)
```

Read the `PhillyTotalPopHHinc` shapefile into an object named `philly_sf`. Check out the column names of `philly_sf` and and of `ph_edu` to determine which one might contain the unique identifier for the join.

```
## sf ##
# if you need to read in again:
# philly_sf <- st_read("data/Philly/")
names(philly_sf)
```

Join the `ph_edu` data frame with `philly` using `merge` as described above. Use the `names()` command to see if the join was successful.

```
# this is base::merge()
philly_sf_merged <- merge(philly_sf, ph_edu, by.x = "GEOID10", by.y = "GEOID")
names(philly_sf_merged) # note the geometry column
```

```
#>  [1] "GEOID10"         "STATEFP10"       "COUNTYFP10"
#>  [4] "TRACTCE10"       "NAME10"          "NAMELSAD10"
#>  [7] "MTFCC10"         "FUNCSTAT10"      "ALAND10"
#> [10] "AWATER10"        "INTPTLAT10"      "INTPTLON10"
#> [13] "GISJOIN"         "Shape_area"      "Shape_len"
#> [16] "medHHinc"        "totalPop"        "NAME"
#> [19] "fem_bachelor"    "fem_doctorate"   "fem_highschool"
#> [22] "fem_noschool"    "fem_ovr_25"      "male_bachelor"
#> [25] "male_doctorate"  "male_highschool" "male_noschool"
#> [28] "male_ovr_25"     "pop_ovr_25"      "geometry"
```

The same with `sp`

```
## sp ##
# if you need to read in again:
# philly_sp <- readOGR("data/Philly/", "PhillyTotalPopHHinc")

# this is sp::merge()
philly_sp_merged <- merge(philly_sp, ph_edu, by.x = "GEOID10", by.y = "GEOID")

names(philly_sp_merged) # no geometry column here
```

```
#>  [1] "GEOID10"         "STATEFP10"       "COUNTYFP10"
#>  [4] "TRACTCE10"       "NAME10"          "NAMELSAD10"
#>  [7] "MTFCC10"         "FUNCSTAT10"      "ALAND10"
#> [10] "AWATER10"        "INTPTLAT10"      "INTPTLON10"
#> [13] "GISJOIN"         "Shape_area"      "Shape_len"
#> [16] "medHHinc"        "totalPop"        "NAME"
#> [19] "fem_bachelor"    "fem_doctorate"   "fem_highschool"
#> [22] "fem_noschool"    "fem_ovr_25"      "male_bachelor"
#> [25] "male_doctorate"  "male_highschool" "male_noschool"
#> [28] "male_ovr_25"     "pop_ovr_25"
```

2.2 Reprojecting

Not unfrequently you may have to reproject spatial objects that you perhaps have acquired from various sources and that you need to be in the same Coordinate Reference System (CRS). The functions that do this typically take the following two arguments:

- the spatial object to reproject
- a CRS object with the new projection definition

You can reproject

- a `Spatial*` object with `spTransform()`
- a `sf` object with `st_transform()`
- a `raster` object with `projectRaster()`

The perhaps trickiest part here is to determine the definition of the projection, which needs to be a character string in proj4 format. You can look it up online. For example for UTM zone 33N (EPSG:32633) the string would be:

```
+proj=utm +zone=33 +ellps=WGS84 +datum=WGS84 +units=m +no_defs
```

You can retrieve the CRS:

- from an existing `Spatial*` object with `proj4string()`
- from an `sf` object with `st_crs()`
- from a `raster` object with `crs()`

Let us now go back to the homicide shapefile we exported to `"PhillyHomicides"`. Let's read it back in and transform it so it matches the projection of the Philadelphia Census tracts. We will assign it to a new object called `ph_homic_aea_`.

First we read it in and check the CRS for both files. Then we use the respective transformation functions to reproject.

```
## sf ##
ph_homic_sf <- st_read("data/PhillyHomicides/")
```

```
#> Reading layer `PhillyHomicides' from data source `/Users/cengel/Anthro/R_Class/R_Workshops/R-spatial/da
#> Simple feature collection with 3883 features and 8 fields
#> geometry type:  POINT
#> dimension:      XY
#> bbox:           xmin: -75.26809 ymin: 39.87503 xmax: -74.95874 ymax: 40.13086
#> epsg (SRID):    4326
#> proj4string:    +proj=longlat +datum=WGS84 +no_defs
```

```
st_crs(philly_sf)
```

```
#> Coordinate Reference System:
#>   No EPSG code
#>  proj4string: "+proj=aea +lat_1=29.5 +lat_2=45.5 +lat_0=37.5 +lon_0=-96 +x_0=0 +y_0=0 +ellps=GRS80 +uni
```

```
st_crs(ph_homic_sf)
```

```
#> Coordinate Reference System:
#>   EPSG: 4326
#>   proj4string: "+proj=longlat +datum=WGS84 +no_defs"
```

```
ph_homic_aea_sf <- st_transform(ph_homic_sf, st_crs(philly_sf))

## sp ##
ph_homic_sp <- readOGR("data/PhillyHomicides/", "PhillyHomicides")
```

```
#> OGR data source with driver: ESRI Shapefile
#> Source: "/Users/cengel/Anthro/R_Class/R_Workshops/R-spatial/data/PhillyHomicides", layer: "PhillyHomic
#> with 3883 features
#> It has 8 fields
```

```
proj4string(philly_sp)
```

```
#> [1] "+proj=aea +lat_1=29.5 +lat_2=45.5 +lat_0=37.5 +lon_0=-96 +x_0=0 +y_0=0 +ellps=GRS80 +units=m +no_de
```

```
proj4string(ph_homic_sp)
```

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

```
ph_homic_aea_sp <- spTransform(ph_homic_sp, CRS(proj4string(philly_sp)))
```

We can use the `range()` command from the R base package to compare the coordinates before and after reprojection and confirm that you actually have transformed them. `range()` simply returns the *min* and *max* value of a vector of numbers that you give it. So you can check with:

```
range(st_coordinates(ph_homic_aea_sf))
```

```
#> [1]  457489.7 1763671.8
```

```
range(st_coordinates(ph_homic_sf))
```

```
#> [1] -75.26809  40.13086
```

```
range(coordinates(ph_homic_aea_sp))
```

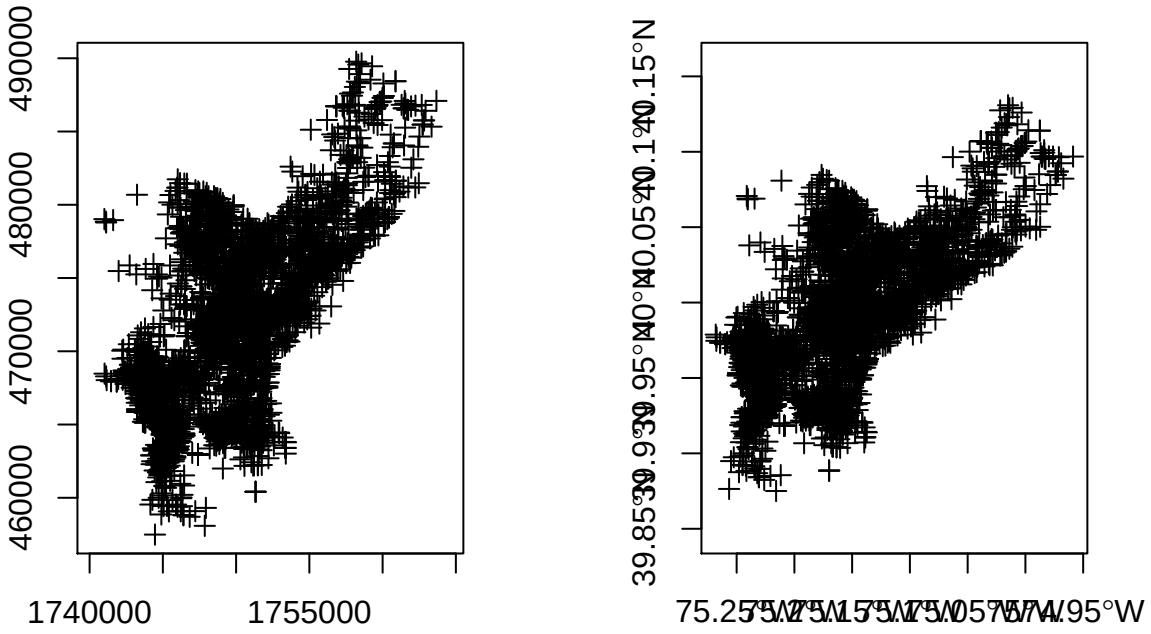
```
#> [1]  457489.7 1763671.8
```

```
range(coordinates(ph_homic_sp))
```

```
#> [1] -75.26809  40.13086
```

We can also compare them visually with:

```
par(mfrow=c(1,2))
plot(ph_homic_aea_sp, axes=TRUE)
plot(ph_homic_sp, axes=TRUE)
```

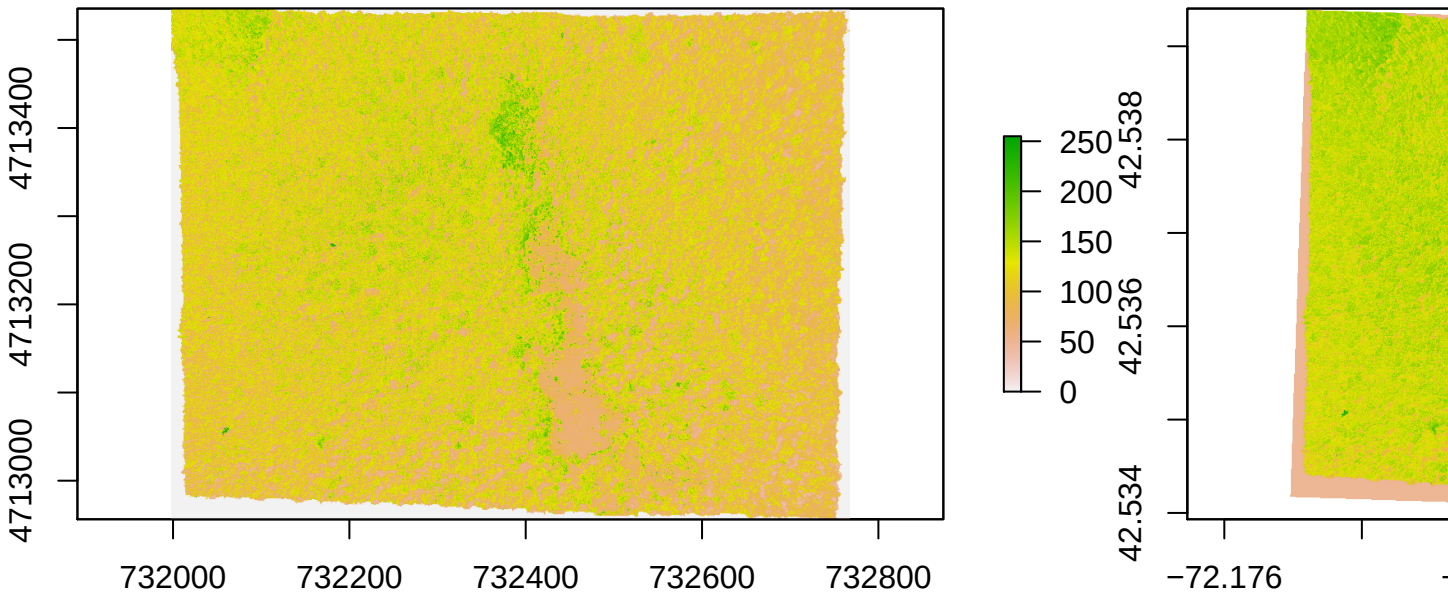


Here is what it would look like to reproject the HARV raster used earlier to a WGS84 projection.

```
# if you need to load again:
#HARV <- raster("data/HARV_RGB_Ortho.tif")
crs(HARV)
```

```
#> CRS arguments:
#>  +proj=utm +zone=18 +datum=WGS84 +units=m +no_defs +ellps=WGS84
#> +towgs84=0,0,0
```

```
HARV.WGS84 <- projectRaster(HARV, crs="+proj=longlat +ellps=WGS84 +datum=WGS84 +no_defs")
plot(HARV); plot(HARV.WGS84)
```



2.3 Points in Polygons

For the next exercise we want to calculate the density of homicides for each census tract in Philadelphia as

`number of homicides per census tract / area per census tract`

To achieve this this we join the points of homicide incidence to the census tract polygon. You might be familiar with this operation from other GIS packages.

For `sp` objects we can use the `aggregate()` function[2]. Here are the arguments that it needs:

- the `SpatialPointDataframe`with the homicide incidents as point locations,
- the `SpatialPolygonDataframe` with the census tract polygons to aggregate on, and
- an aggregate function. Since we are interested in counting the points (i.e. the rows of all the points that belong to a certain polygon), we can use length (of the respective vectors of the aggregated data).

Let's do this.

To count homicides per census tract we use the `OBJ_ID` field from `ph_homic_aea` for homicide incidents and `philly` polygons to aggregate on and save the result as `ph_hom_count`. Use `length` as aggregate function.

```
ph_hom_count_sp <- aggregate(x = ph_homic_aea_sp["OBJ_ID"], by = philly_sp, FUN = length)
# make sure we understand this error message:
# aggregate(x = ph_homic_sp, by = philly_sp, FUN = length)
```

[2]There is also an `aggregate()` function in the `stats` package that comes with the R standard install. Note that `sp` extends this function so it can take `Spatial*` objects and aggregate over the geometric features.

Now let us investigate the object we created.

```
class(ph_hom_count_sp)
```

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

```
names(ph_hom_count_sp)
```

```
#> [1] "OBJ_ID"
```

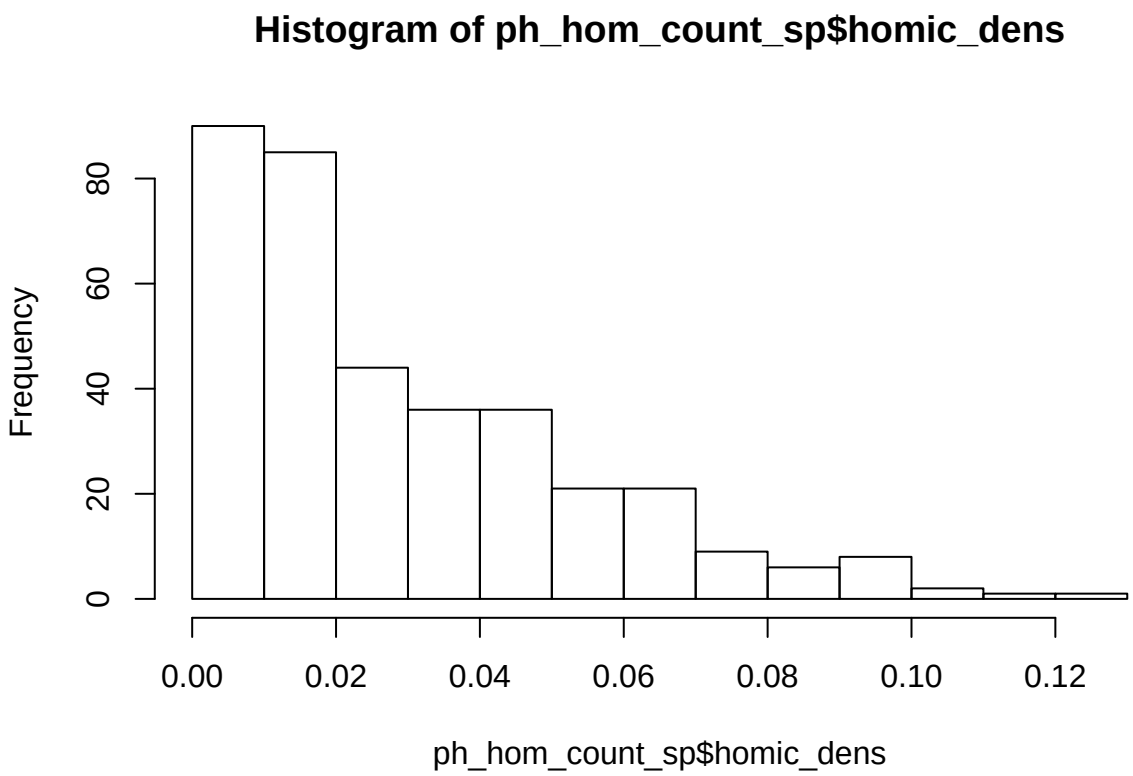
```
head(ph_hom_count_sp)
```

```
#>   OBJ_ID
#> 0      2
#> 1      3
#> 2     11
#> 3      3
#> 4      4
#> 5      5
```

Now we can calculate the density of homicides in Philadelphia, normalized over the area for each census tract.

We use `gArea()` from the `rgeos` library. `gArea`, when given a `SpatialPolygon`, calculates the size of the area covered. If we need that calculation for each polygon, we set `byid = TRUE`. Units are in map units.

```
library(rgeos)
# we multiply by by 1000000 to get sq km.
ph_hom_count_sp$homic_dens <- 1e6 * (ph_hom_count_sp$OBJ_ID/gArea(ph_hom_count_sp, byid = FALSE))

hist(ph_hom_count_sp$homic_dens)
```



We will write it out for later. (Note that this will produce an error if the file already exists. You can force it to write out with the option `overwrite_layer = TRUE`)

```
writeOGR(ph_hom_count_sp, "data/PhillyCrimerate", "PhillyCrimerate", driver = "ESRI Shapefile")
```

There might be other instances where we don't want to aggregate, but might only want to know which polygon a point falls into. In that case we can use `over()`. In fact, the `aggregate()` function used above makes use of `over()`. See https://cran.r-project.org/web/packages/sp/vignettes/over.pdf for more details on the over-methods. `point.in.poly()` from the `spatialEco` package intersects point and polygons and adds polygon attributes to points. There is also `point.in.polygon()` from the `sp` package which tests if a point or set of points fall in a given polygon.

For `sf` objects we need to add one more step. We first use `st_within()` to determine which polygon a points falls into. We can then use the result to aggregate.

Need to add this

2.4 Select Polygons by Location

For the next example our goal is to select all Philadelphia census tracts within a range of 2 kilometers from the city center.

Think about this for a moment – what might be the steps you'd follow?

```
## How about:

# 1. Get the census tract polygons.
# 2. Find the Philadelphia city center coordinates.
# 3. Create a buffer around the city center point.
# 4. Select all census tract polygons that intersect with the center buffer
```

2.4.1 Using the sp package

In order to perform those operations on an `sp` object we will need to make use of an additional package, called `rgeos`. Make sure you have it loaded.

```
library(rgeos)
# if you need to read it in again
# philly_sp <- readOGR("data/Philly/", "PhillyTotalPopHHinc", verbose = F)
```

We will use `philly_sp` for the census tract polygons.

In addition, we need to create a `SpatialPoints` object with the Philadelphia city center coordinates.

Lat is 39.95258 and Lon is -75.16522. This is in WGS84.

With this information, we create a `SpatialPoints` object named `philly_ctr`.

```
coords <- data.frame(x = -75.16522, y = 39.95258) # set the coordinates
prj <- CRS("+proj=longlat +ellps=WGS84 +datum=WGS84 +no_defs") # the projection string for WGS84
philly_ctr <- SpatialPoints(coords, proj4string = prj) # create the spatialPoints
```

Next, we create a buffer around the city center point.
Here is where we will use the `gBuffer()` function from the `rgeos` package. For this purpose we will need to provide two arguments: the **sp object** and the **width** of the buffer, which is assumed to be in map units. The function returns a `SpatialPolygons` object to you with the buffer - name it `philly_buf`.
So our command would look something like

```
philly_buf <- gBuffer(the_spatial_point_object, width = a_number_here)
```

Now – before we create this buffer, think about what you need to do to `philly_ctr` before you proceed.

```
philly_ctr_aea <- spTransform(philly_ctr, CRS(proj4string(philly_sp))) # reproject!!
philly_buf <- gBuffer(philly_ctr_aea, width=2000)  # create buffer around center
```

Ok. Now we can use that buffer to select all census tract polygons that intersect with the center buffer.

We will use the `gIntersects()` function from the `rgeos` package for this. The function tests if two geometries (let's name them *spgeom1* and *spgeom2*) have points in common or not. `gIntersects` returns TRUE if *spgeom1* and *spgeom2* have at least one point in common.

Here is where we determine if the census tracts fall within the buffer. In addition to our two `sp` objects (`philly_buf` and `philly_sp`) we need to provide one more argument, `byid`. It determines if the function should be applied across ids (TRUE) or the entire object (FALSE) for *spgeom1* and *spgeom2*. The default setting is FALSE. Since we want to compare *every single* census tract polygon in our `philly_sp` object we need to set it to TRUE.

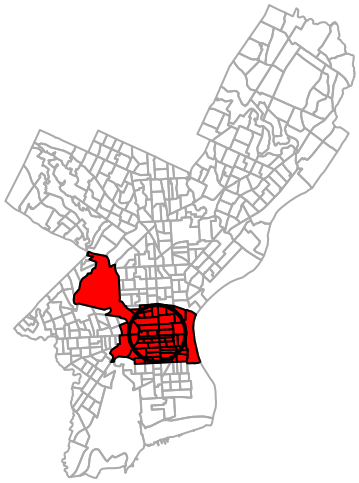
```
philly_buf_intersects <-  gIntersects (philly_buf, philly_sp, byid=TRUE) # determine which census tract

# what kind of object is this?
class(philly_buf_intersects)
```

```
#> [1] "matrix"
```

```
# subset
philly_sel <- philly_sp[as.vector(philly_buf_intersects),]
```

Finally, we plot it all.

```
plot (philly_sp, border="#aaaaaa")
plot (philly_sel, add=T, col="red")
plot (philly_buf, add=T, lwd = 2)
```



2.4.2 Using the `sf` package

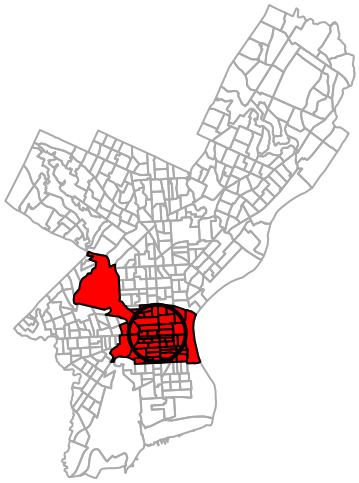
To give you a sense of how this might be done using the `sf` package we will reproduce here the same example as above.

For the spatial operations we can recur to the suite of geometric operations that come with the `sf` package , in particular we will use `st_buffer()` and `st_intersects()`

```
library(sf)
philly_sf <- st_read("data/Philly/", quiet = T)

# make a simple feature point with CRS
philly_ctr_sfc <- st_sfc(st_point(c(-75.16522, 39.95258)), crs = 4326)

# reproject
philly_ctr_aea_sf <- st_transform(philly_ctr_sfc, st_crs(philly_sf))

# create buffer
philly_buf_sf <- st_buffer(philly_ctr_aea_sf, 2000)

# find intersection between buffer and census polygons
philly_buf_intersects <- st_intersects(philly_buf_sf, philly_sf)
class(philly_buf_intersects)
```

```
#> [1] "sgbp"
```

```
# subset
philly_sel_sf <- philly_sf[unlist(philly_buf_intersects),]

# plot
plot(st_geometry(philly_sf), border="#aaaaaa")
plot(st_geometry(philly_sel_sf), add=T, col="red")
plot(st_geometry(philly_buf_sf), add=T, lwd = 2)
```



2.4.3 `sp` - `sf` comparison

how to..	for `sp` objects	for `sf` objects
join attributes	`sp::merge()`	`base::merge()`
reproject	`spTransform()`	`st_transform()`
retrieve (or assign) CRS	`proj4string()`	`st_crs()`
count points in polygons	`over()`	`st_within` and `aggregate()`
buffer	`rgeos::gBuffer()` (separate package)	`st_buffer()`
select by location	`g*` functions from `rgeos`	geos functions in `sf`

2.4.4 `raster` operations

to come

Chapter 3

Making Maps in R

Learning Objectives

- create a choropleth map with `spplot`
- cprepare an `sp` object to be plotted with `ggplot`
- create a choropleth map with `ggplot`
- add a basemap with `ggmap`
- use `RColorBrewer` to improve legend colors
- use `classInt`to improve legend breaks
- create a choropleth map with `tmap`
- create an interactive map with `leaflet`
- customize a `leaflet` map with popups and layer controls

In the preceding examples we have used the base `plot` command to take a quick look at our spatial objects.

In this section we will explore several alternatives to map spatial data with R. For more packages see the "Visualisation" section of the CRAN Task View.

Mapping packages are in the process of keeping up with the development of the new `sf` package, so they typicall accept both `sp` and `sf` objects. However, there are a few exceptions.

Of the packages shown here `spplot()`, which is part of the good old `sp` package, only takes `sp` objects. The development version of `ggplot2` can take `sf` objects, though `ggmap` seems to still have issues with `sf`. Both `tmap` and `leaflet` can also handle both `sp` and `sf` objects.

3.1 Choropleth mapping with `spplot`

`sp` comes with a plot command `spplot()`, which takes `Spatial*` objects to plot. `spplot()` is one of the earlier functions around to plot geographic objects.

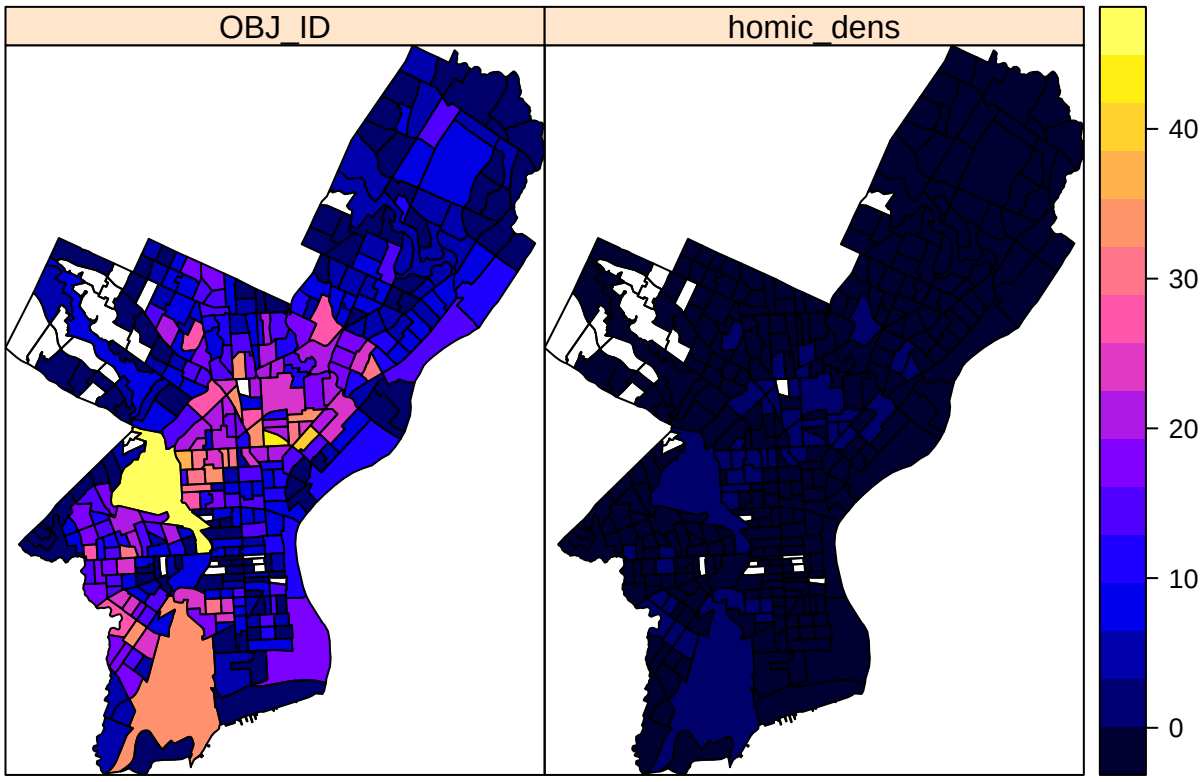
We will use `readOGR()` from the `rgdal` libryary to read back in the `PhillyCrimerate` shapefile we created earlyer back into an object named `philly_crimer_sp`.

```
philly_crimer_sp <- readOGR("data/PhillyCrimerate/", "PhillyCrimerate", verbose = FALSE) # verbose = FA
names(philly_crimer_sp)
```

```
#> [1] "OBJ_ID"     "homic_dens"
```

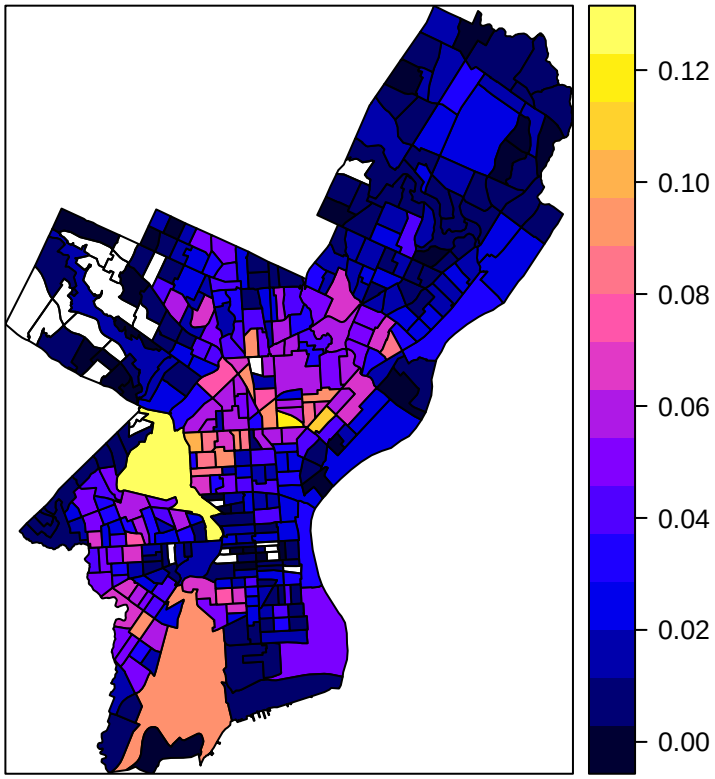
By default `spplot` tries to map everything it can find in the attribute table. Sometimes this does not work, depending on the data types in the attribute table. The variables being compared must have a range of values (or levels if they are factors). The following works, but probably does not make too much sense:

```
spplot(philly_crimer_sp)
```



In order to select specific values to map we can provide the `spplot` function with the name (or names) of the attribute variable(s) we want to plot. It is the name of the column of the `Spatial*Dataframe` as character string (or a vector if several).

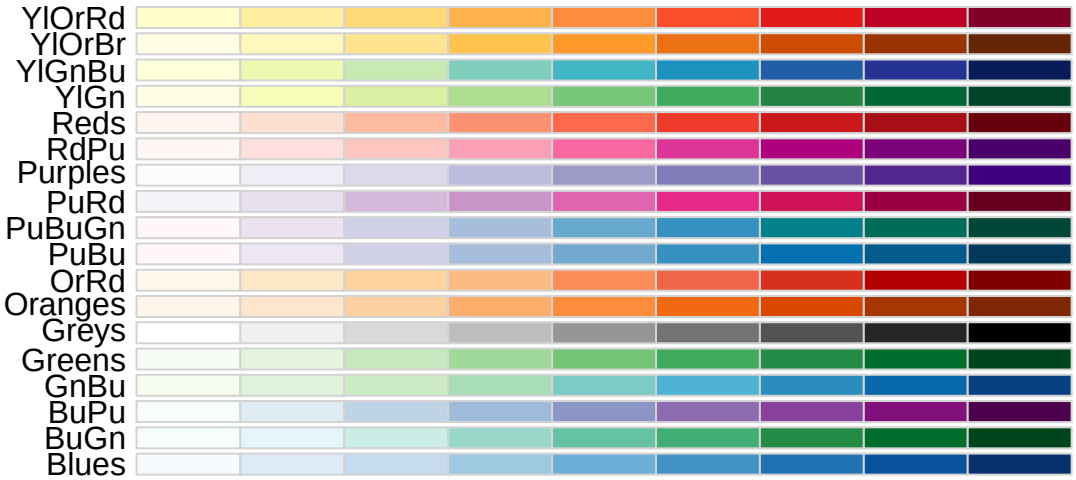
```
spplot(philly_crimer_sp, "homic_dens")
```



Let us try to improve this a little.

First off, we want to change the color palette. For this we use a library called `RColorBrewer`[1]. For more about ColorBrewer palettes read this. Load the `RColorBrewer` library and explore all sequential color schemes.

```
library(RColorBrewer)
display.brewer.all(type="seq")
```



To make the color palettes from ColorBrewer available as R palettes we use the `brewer.pal()` funcdtion. It takes two arguments: - the number of different colors desired and - the name of the palette as character string.

We select 5 colors from the 'Orange-Red' plaette and assign it to an object `pal`.

[1] This is not the only way to provide color palettes. You can create your customized palette in many different ways or simply as a vector of hexbin color codes, like `c( "#FDBB84" "#FC8D59" "#EF6548")`.

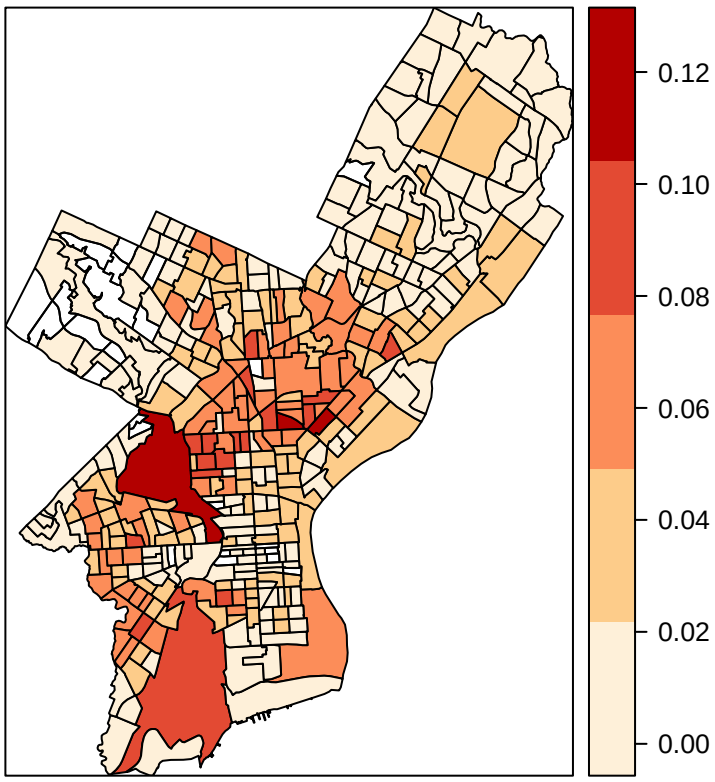
```
pal <- brewer.pal(5, "OrRd") # we select 5 colors from the palette
class(pal)
```

```
#> [1] "character"
```

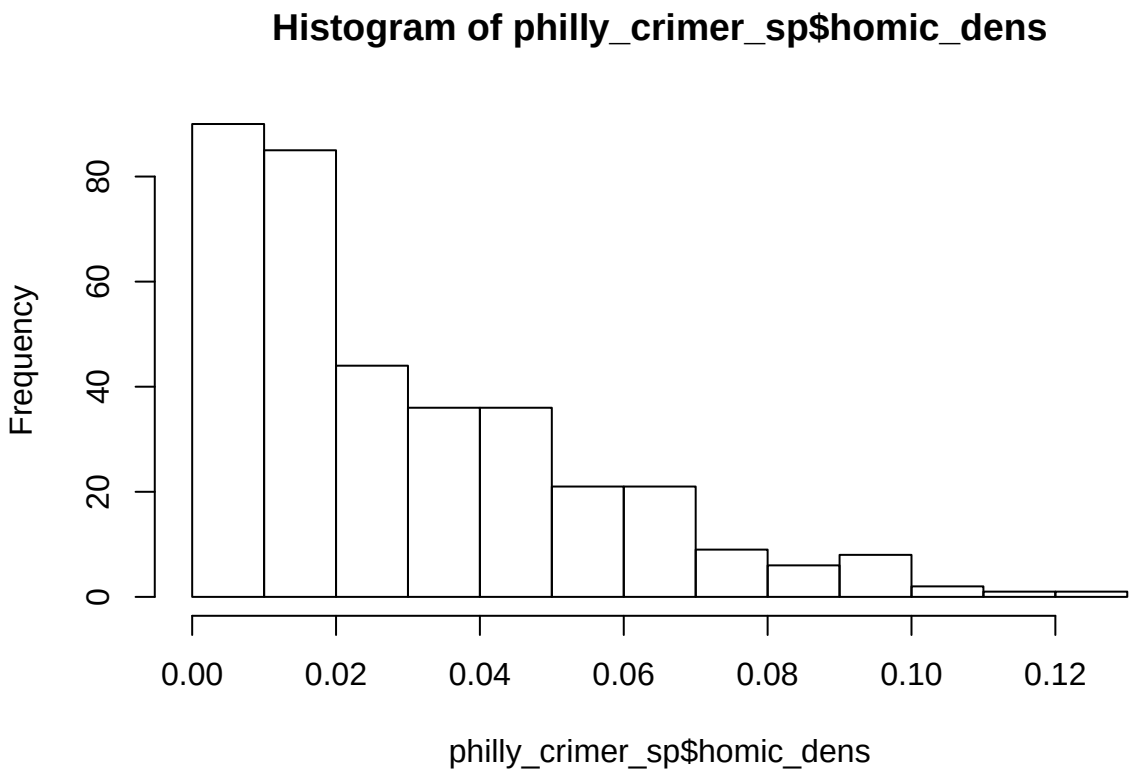
Now we pass this information on to `spplot`. We need to provide two more arguments:

- `col.regions` which we set to the palette we just created and
- `cuts` which in our case is 4. It bins our continous variable into 5 brackets and will make our colors match up with those class brackets.

```
spplot(philly_crimer_sp, "homic_dens", col.regions = pal, cuts = 4)
```



Looks better already. However, since are unevenly distributed (see below), in order to better distinguish the census tracts with low values, we might want to set the breaks to quantiles.



We will use `classIntervals` from the `classInt` library. This returns an object of type `classIntervals`, where we can extract the values for the breaks. (Because of a the way spplot handles break values[2] we need to shift them slightly.)

```
library(classInt)
breaks_qt <- classIntervals(philly_crimer_sp$homic_dens, n = 5, style = "quantile") # quantile is the d
```

```
#> Warning in classIntervals(philly_crimer_sp$homic_dens, n = 5, style =
#> "quantile"): var has missing values, omitted in finding classes
```

```
str(breaks_qt)  # structure of object
```

```
#> List of 2
#>  $ var : num [1:384] 0.00547 0.00821 0.03009 0.00821 0.01094 ...
#>  $ brks: num [1:6] 0.00274 0.00821 0.01641 0.02735 0.04923 ...
#>  - attr(*, "style")= chr "quantile"
#>  - attr(*, "nobs")= int 37
#>  - attr(*, "call")= language classIntervals(var = philly_crimer_sp$homic_dens, n = 5, style = "quantile")
#>  - attr(*, "intervalClosure")= chr "left"
#>  - attr(*, "class")= chr "classIntervals"
```

```
breaks_qt$brks  # break values
```

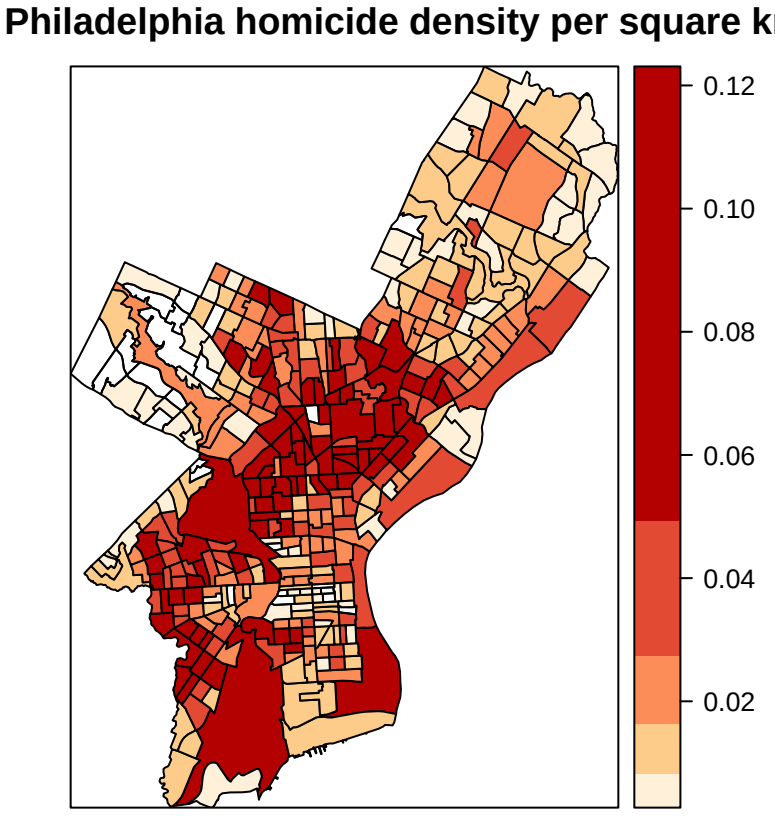
```
#> [1] 0.002735248 0.008205745 0.016411491 0.027352484 0.049234472 0.123086179
```

```
# add a very small value to the top breakpoint, and subtract from the bottom for symmetry
br <- breaks_qt$brks
offs <- 0.0000001
br[1] <- br[1] - offs
br[length(br)] <- br[length(br)] + offs
```

[2]For the correction of breaks after using classIntervals with spplot/levelplot see here http://r.789695.n4.nabble.com/SpatialPolygon-with-the-max-value-gets-no-color-assigned-in-spplot-function-when-using-quot-at-quot-r-td4654672.html

We use the breaks to set the `at=` argument in `spplot()` and also set `main=` to add a title.

```
spplot(philly_crimer_sp, "homic_dens", col.regions=pal, at=br,  main = "Philadelphia homicide density p
```



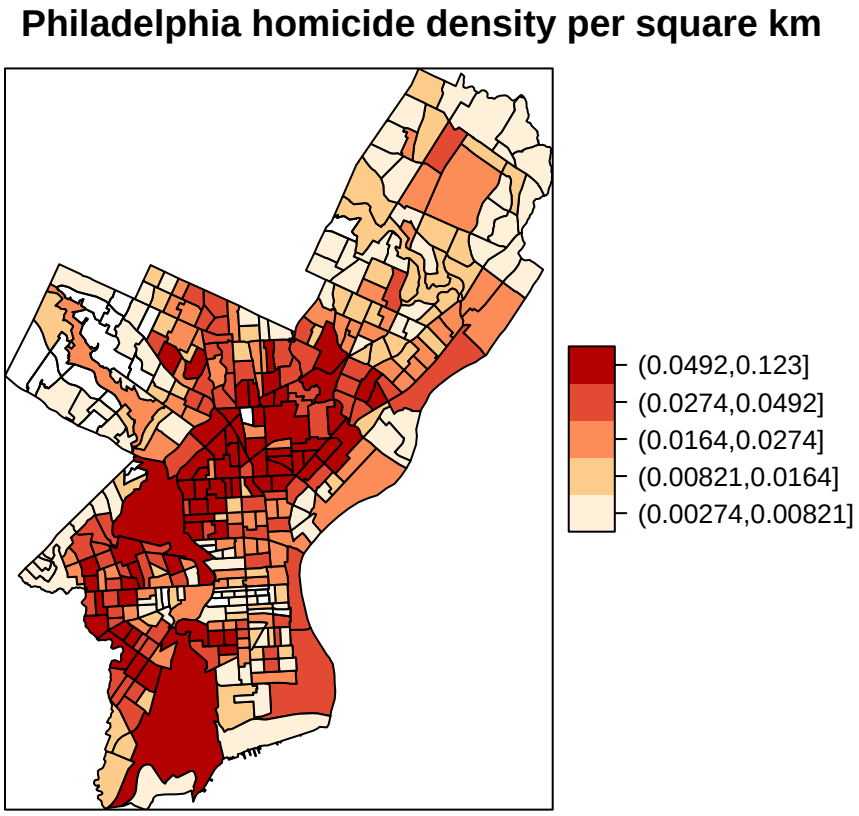
The final issue we will address is the legend, which shows as a graduated color, since we provided a vector of continuous values to map. Here is how we can change this:

- Use the `cut()` function from the base package with the values from `philly_crimer_sp$homic_dens` and the corrected breaks `br` to return a vector with the respective boundaries of the brackets. Use `?cut` if you need help.
- Assign the output vector you get as a new column `homic_dens_bracket` to the `philly_crimer_sp` attributes table. It will help to map the color based on the breaks. Take a look at the values. What object class is that vector?
- Remove the `at=` parameter in `spplot()` (which is only needed for continuous variables) and tell it to plot `homic_dens_bracket`.

```
philly_crimer_sp$homic_dens_bracket <- cut(philly_crimer_sp$homic_dens, br)
head(philly_crimer_sp$homic_dens_bracket)
class(philly_crimer_sp$homic_dens_bracket)
spplot(philly_crimer_sp, "homic_dens_bracket", col.regions=pal, main = "Philadelphia homicide density p
```

Now, this is what you should see:

```
#> Warning in classIntervals(philly_crimer_sp$homic_dens, n = 5, style =
#> "quantile"): var has missing values, omitted in finding classes
```



There are many more arguments for this function to provide additional plot parameters, like the legend position, labels, scales, etc. but we'll leave this for now.

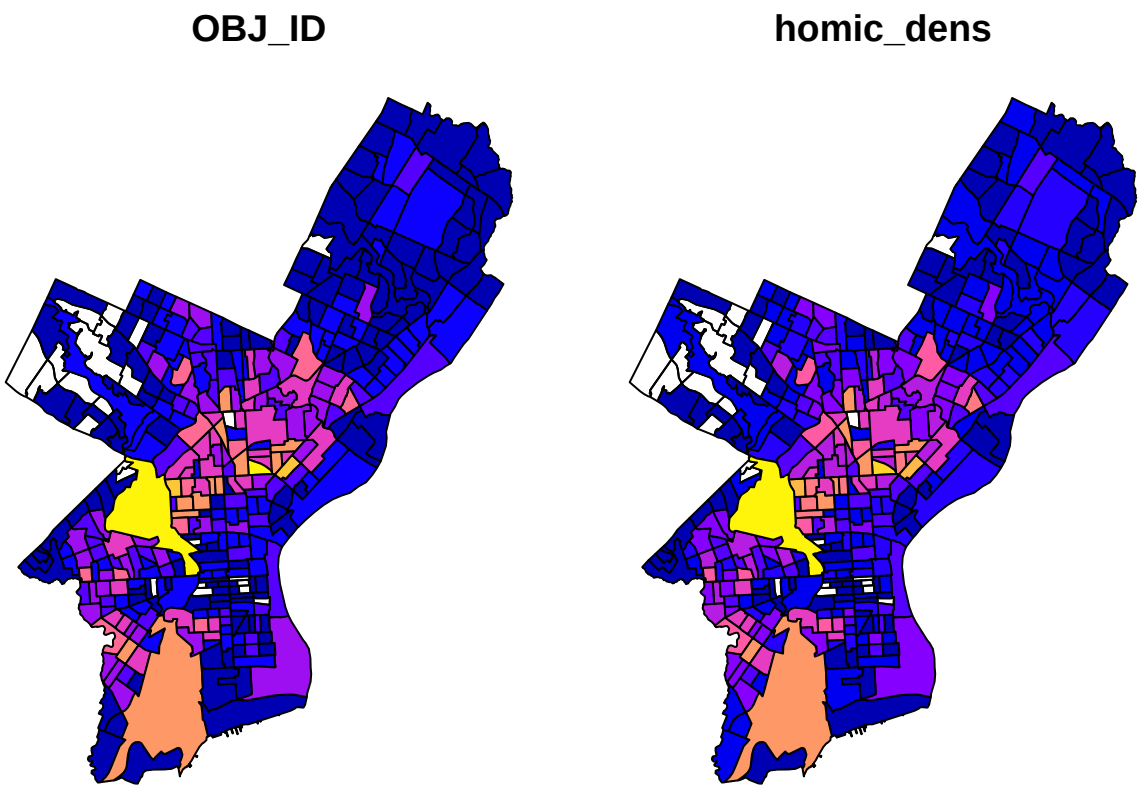
3.2 Plotting simple features (sf) with plot

The sf package extends the base plot command, so it can be used on sf objects. If used without any arguments it will plot all the attributes, like spplot does.

```
philly_crimer_sf <-  st_read("data/PhillyCrimerate/")
```

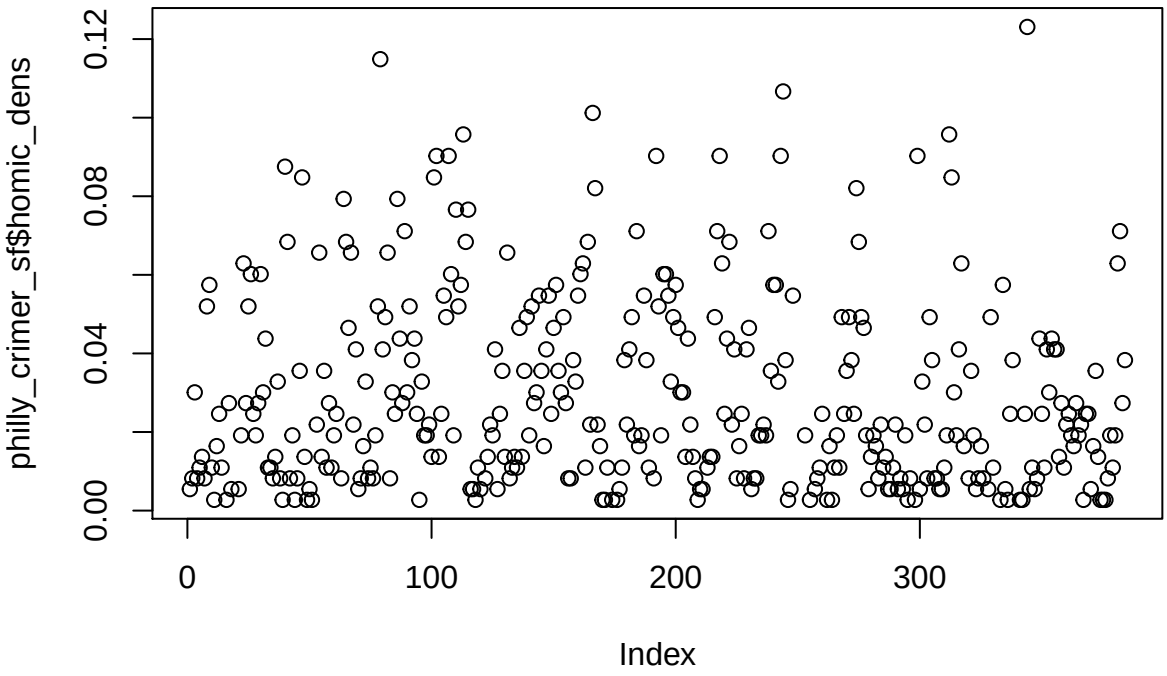
```
#> Reading layer `PhillyCrimerate' from data source `/Users/cengel/Anthro/R_Class/R_Workshops/R-spatial/da
#> Simple feature collection with 384 features and 2 fields
#> geometry type:  MULTIPOLYGON
#> dimension:      XY
#> bbox:           xmin: 1739497 ymin: 457343.7 xmax: 1764030 ymax: 490544.9
#> epsg (SRID):    NA
#> proj4string:   +proj=aea +lat_1=29.5 +lat_2=45.5 +lat_0=37.5 +lon_0=-96 +x_0=0 +y_0=0 +ellps=GRS80 +unit
```

```
plot(philly_crimer_sf)
```

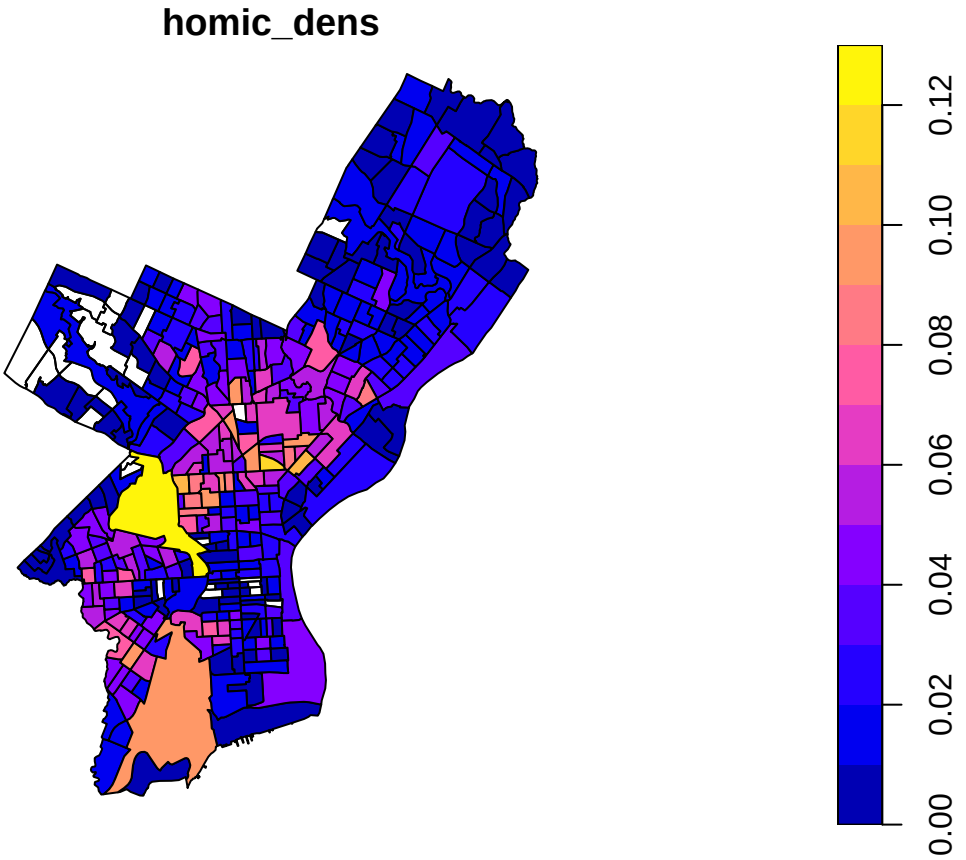


To plot a single attribute we need to provide an object of class `sf`, like so:

```
plot(philly_crimer_sf$homic_dens) # this is a numeric vector!
```

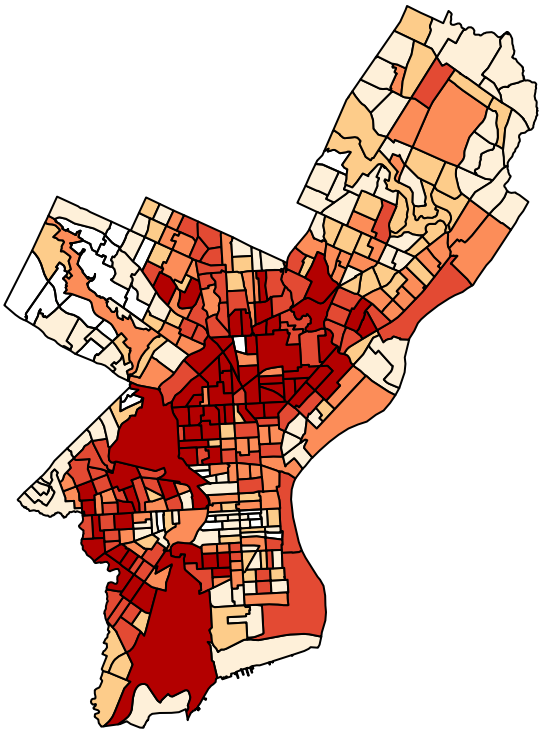



```
plot(philly_crimer_sf["homic_dens"])
```



If we wanted to add our own colors, legend and title we would recur to basic plot parameters to do this.

```
hr_cuts <- cut(philly_crimer_sf$homic_dens, br)
plot(philly_crimer_sf["homic_dens"], main = "Philadelphia homicide density per square km", col = pal[as
legend(1760000, 471000, legend = paste("<", round(br[-1])), fill = pal)
```



3.3 Choropleth mapping with `ggplot2`

`ggplot2` is a widely used and powerful plotting library for R. It is not specifically geared towards mapping, but one can generate great maps.

The `ggplot()` syntax is different from the previous as a plot is built up by adding components with a `+`. You can start with a layer showing the raw data then add layers of annotations and statistical summaries. This allows to easily superimpose either different visualizations of one dataset (e.g. a scatterplot and a fitted line) or different datasets (like different layers of the same geographical area)[3].

For an introduction to `ggplot` check out this book by the package creator or this for more pointers.

In order to build a plot you start with initializing a ggplot object. In order to do that `ggplot()` takes:

- a data argument usually a **dataframe** and
- a mapping argument where x and y values to be plotted are supplied.

In addition, minimally a geometry to be used to determine how the values should be displayed. This is to be added after an `+`.

```
ggplot(data = my_data_frame, mapping = aes(x = name_of_column_with_x_value, y = name_of_column_with_y_value
  geom_point()
```

Or shorter:

```
ggplot(my_data_frame, aes(name_of_column_with_x_value, name_of_column_with_y_value)) +
  geom_point()
```

So if we wanted to map polygons, like census tract boundaries, we would use longitude and latitude of their vertices as our `x` and `y` values and `geom_polygon()` as our geometry.

[3]See Wilkinson L (2005): "The grammar of graphics". Statistics and computing, 2nd ed. Springer, New York.

To plot the equivalent to the map we created with `spplot` above we need to convert `philly_crimer_sp`, which is a `SpatialPolygonsDataframe`, to a regular dataframe. `broom` is a general purpose package which provides functions to turn the messy output of built-in functions in R, such as lm, nls, or t.test, into tidy data frames. We use the `tidy()` command for the conversion[4].

We use use the `tidy`function from the `broom` package for the conversion and create a new object, `philly_crimer_df` for the output.

```
library(broom)
philly_crimer_df <- tidy(philly_crimer_sp)
```

```
#> Regions defined for each Polygons
```

```
head(philly_crimer_df)
```

```
#>      long      lat order  hole piece group id
#> 1 1763647 484837.3     1 FALSE     1   0.1  0
#> 2 1763473 485194.5     2 FALSE     1   0.1  0
#> 3 1763366 485341.0     3 FALSE     1   0.1  0
#> 4 1763378 485353.9     4 FALSE     1   0.1  0
#> 5 1763321 485414.1     5 FALSE     1   0.1  0
#> 6 1762968 485731.8     6 FALSE     1   0.1  0
```

Here are the important components of the output.

- *long* and *lat* of the vertices.
- *order.* This just shows in which order ggplot should "connect the dots"
- *hole* Is this polygon a hole or not?
- *group.* This is very important! `geom_` functions can take a group argument which controls (amongst other things) whether adjacent points should be connected by lines. If they are in the same group, then they get connected, but if they are in different groups then they don't. Essentially, having to points in different groups means that ggplot "lifts the pen" when going between them.

But wait. `tidy()` made us loose the attributes that we want to map, so we have to take care of that. We extract the polygon IDs from `philly_crimer_sp` and add them to its dataframe as a column, named, for example, `polyID`. This requires a bit of understanding of the internal structure of `philly_crimer_sp`. You can take a peek with `str(philly_crimer_sp, max.level = 2)`.

I use `slot(philly_crimer_sp, "polygons")` as argument to `sapply()` to iterate over the polygons slots and then extract the ID slot for each polygon, also with `slot()`.

Then we can use the polygon IDs with `merge()` to combine the attribute table from `philly_crimer_sp` with `philly_crimer_df`.

```
philly_crimer_sp$polyID <-
  sapply(slot(philly_crimer_sp, "polygons"), function(x) slot(x, "ID"))
philly_crimer_df <-
  merge(philly_crimer_df, philly_crimer_sp, by.x = "id", by.y="polyID")
head(philly_crimer_df)
```

```
#>   id    long      lat order  hole piece group OBJ_ID  homic_dens
#> 1  0 1763647 484837.3     1 FALSE     1   0.1      2 0.005470497
#> 2  0 1763473 485194.5     2 FALSE     1   0.1      2 0.005470497
#> 3  0 1763366 485341.0     3 FALSE     1   0.1      2 0.005470497
#> 4  0 1763378 485353.9     4 FALSE     1   0.1      2 0.005470497
#> 5  0 1763321 485414.1     5 FALSE     1   0.1      2 0.005470497
#> 6  0 1762968 485731.8     6 FALSE     1   0.1      2 0.005470497
#>   homic_dens_bracket
```

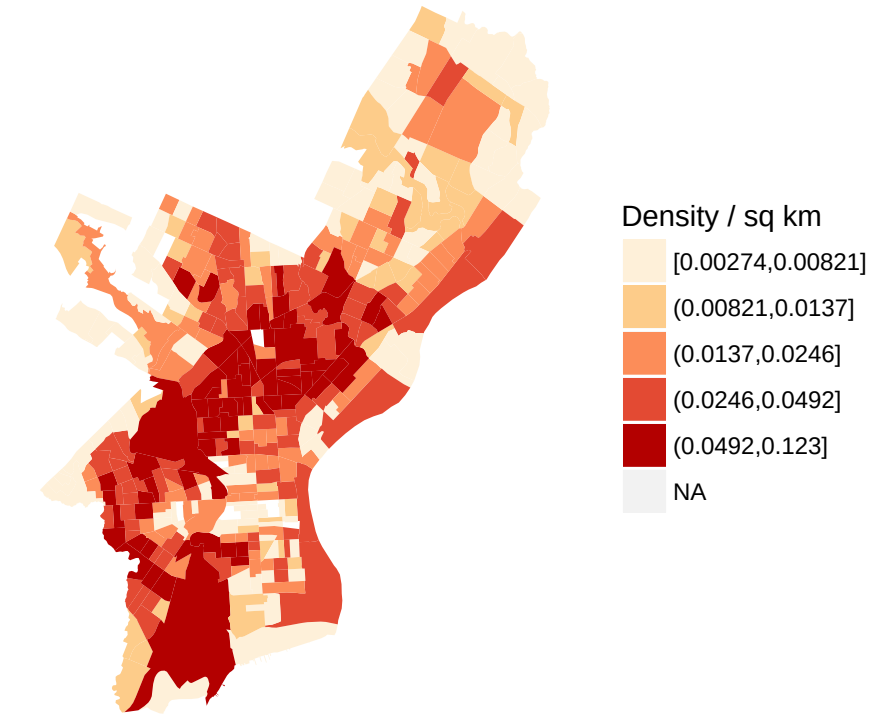
[4] You may still see examples that use `ggplot2::fortify`. This will likely be deprecated in the future.

```
#> 1  (0.00274,0.00821]
#> 2  (0.00274,0.00821]
#> 3  (0.00274,0.00821]
#> 4  (0.00274,0.00821]
#> 5  (0.00274,0.00821]
#> 6  (0.00274,0.00821]
```

OK. All set to plot.

There is a lot going on in this command, so I have provided comments in the code.

```
library(ggplot2)

ggplot() +                                                  # initialize ggplot object
  geom_polygon(                                             # make a polygon
    data = philly_crimer_df,                                # data frame
    aes(x = long, y = lat, group = group,                   # coordinates, and group them by polygons
        fill = cut_number(homic_dens, 5))) +                   # variable to use for filling
  scale_fill_brewer("Density / sq km", palette = "OrRd") + # fill with brewer colors
  ggtitle("Philadelphia homicides") +    # add title
  theme(line = element_blank(),                             # remove axis lines ..
        axis.text=element_blank(),                          # .. tickmarks..
        axis.title=element_blank(),                         # .. axis labels..
        panel.background = element_blank()) +               # .. background gridlines
  coord_equal()                                             # both axes the same scale
```



ggplot will soon be able to plot sf objects directly. This will look like:

```
ggplot(philly_sf) + geom_sf(aes(fill=homic_dens))
```

3.4 Adding basemaps with ggmap

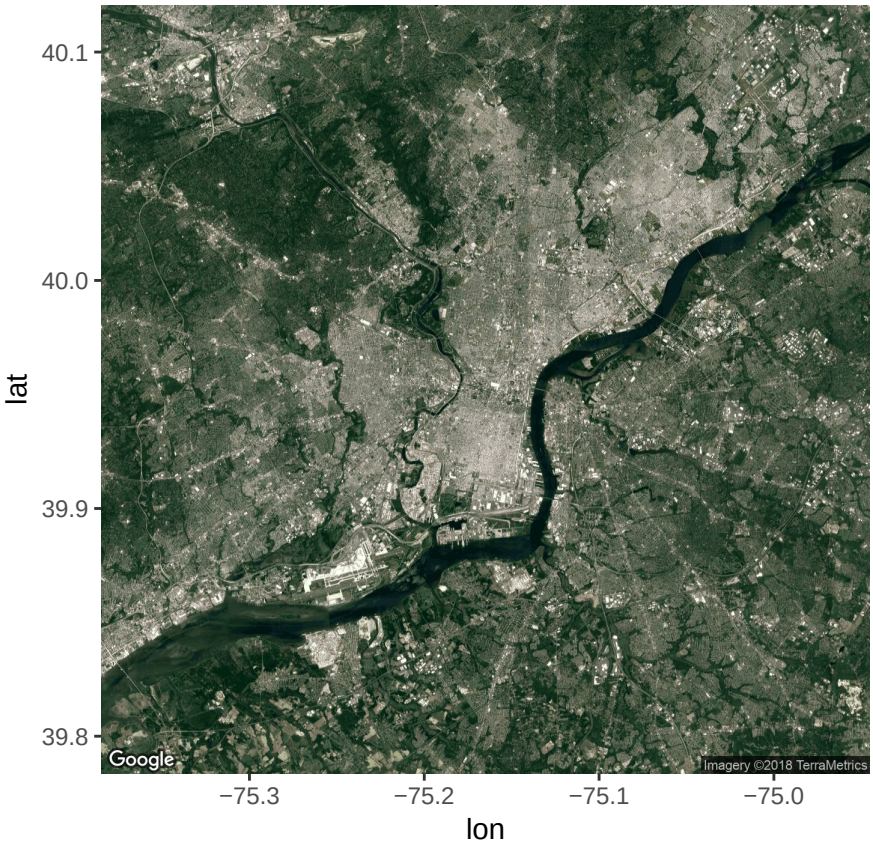
ggmap builds on ggplot and allows to pull in tiled basemaps from different services, like Google Maps and OpenStreetMaps[5].

So let's overlay the map from above on a google satellite basemap.

First we use the get_map() command from ggmap to pull down the basemap. We need to tell it the location or the boundaries of the map, the zoom level, and what kind of map service we like (default is Google terrain). It will actually download the tile. get_map() returns a ggmap object, name it ph_basemap.

In order to view the map we then use ggmap(ph_basemap).

```
library(ggmap)

# Philadelphia Lat 39.95258 and Lon is -75.16522
ph_basemap <- get_map(location=c(lon = -75.16522, lat = 39.95258), zoom=11, maptype = 'satellite')

ggmap(ph_basemap)
```



Then we can reuse the code from the ggplot example above, just replacing the first line, where we initialized a ggplot object above

```
ggplot() +
```

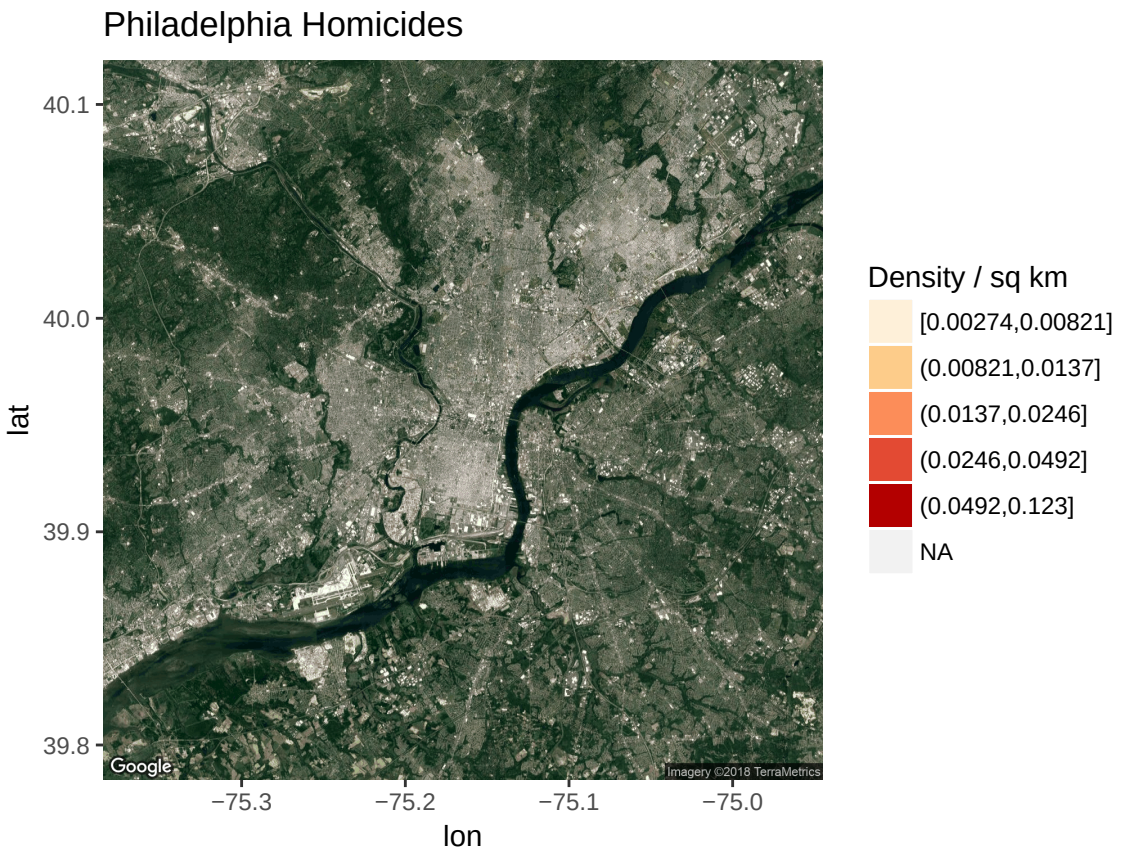
with the line to call our basemap:

```
ggmap(ph_basemap) +
```

[5]Note that the use of Stamen Maps currently only works with a patch and that Cloudmade maps retired its API so it is no longer possible to be used as basemap. RgoogleMaps is another library that provides an interface to query the Google server for static maps.

We also can get rid of the `theme()` and `coord_equal()` parts, as `ggmap` takes care of most of it.

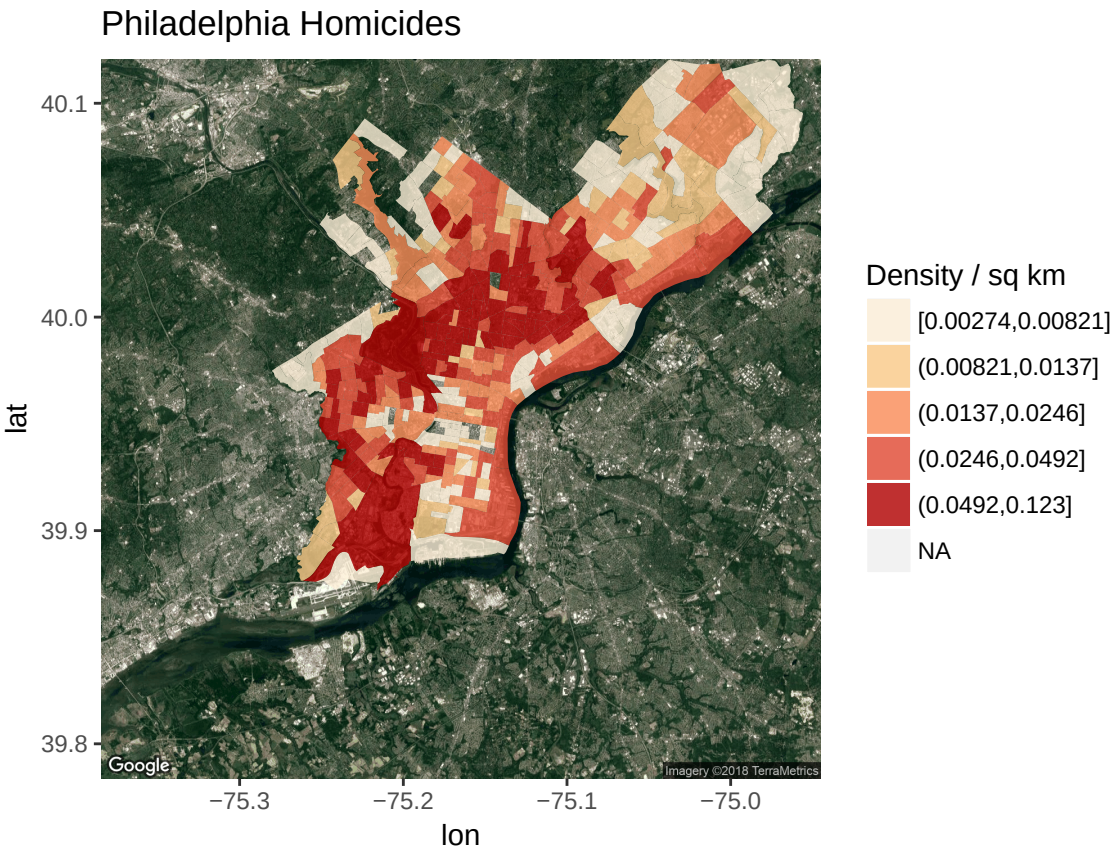
```
ggmap(ph_basemap) +
  geom_polygon(data = philly_crimer_df, aes(x=long, lat, group = group, fill = cut_number(homic_dens, 5)
  scale_fill_brewer("Density / sq km", palette = "OrRd") +
  ggtitle("Philadelphia Homicides")
```



Oops. Any idea what might be going on?

Unfortunately we have to go back to our original `philly_crimer_sp` object and reproject it to the CRS (WGS84) that works with Google maps. We then have to recreate our dataframe for `ggplot`.

```
# reproject
philly_WGS84 <- spTransform(philly_crimer_sp, CRS("+init=epsg:4326"))
# create data frame
philly_df_WGS84 <- tidy(philly_WGS84)
# add attribute data back in
philly_WGS84$polyID <- sapply(slot(philly_WGS84, "polygons"), function(x) slot(x, "ID"))
philly_df_WGS84 <- merge(philly_df_WGS84, philly_WGS84, by.x = "id", by.y="polyID")

# plot
ggmap(ph_basemap) +
  geom_polygon(data = philly_df_WGS84, aes(x=long, lat, group = group, fill = cut_number(homic_dens, 5)
  scale_fill_brewer("Density / sq km", palette = "OrRd") +
  ggtitle("Philadelphia Homicides")
```



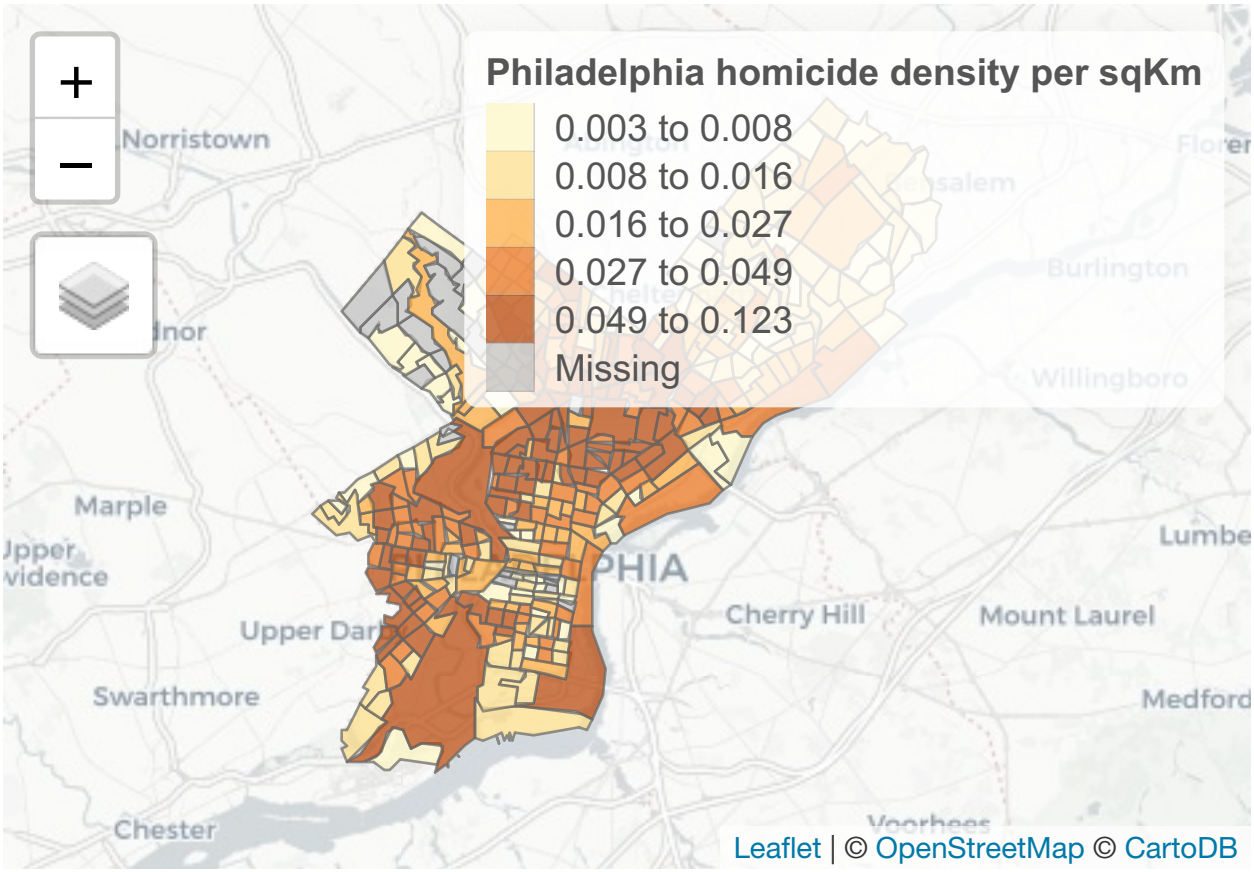
The ggmap package also includes functions for distance calculations, geocoding, and calculating routes.

3.5 Choropleth with tmap

tmap is specifically designed to make creation of thematic maps more convenient. It borrows from teh ggplot syntax and takes care of a lot of the styling and aesthetics. This reduces our amount of code significantly. We only need:

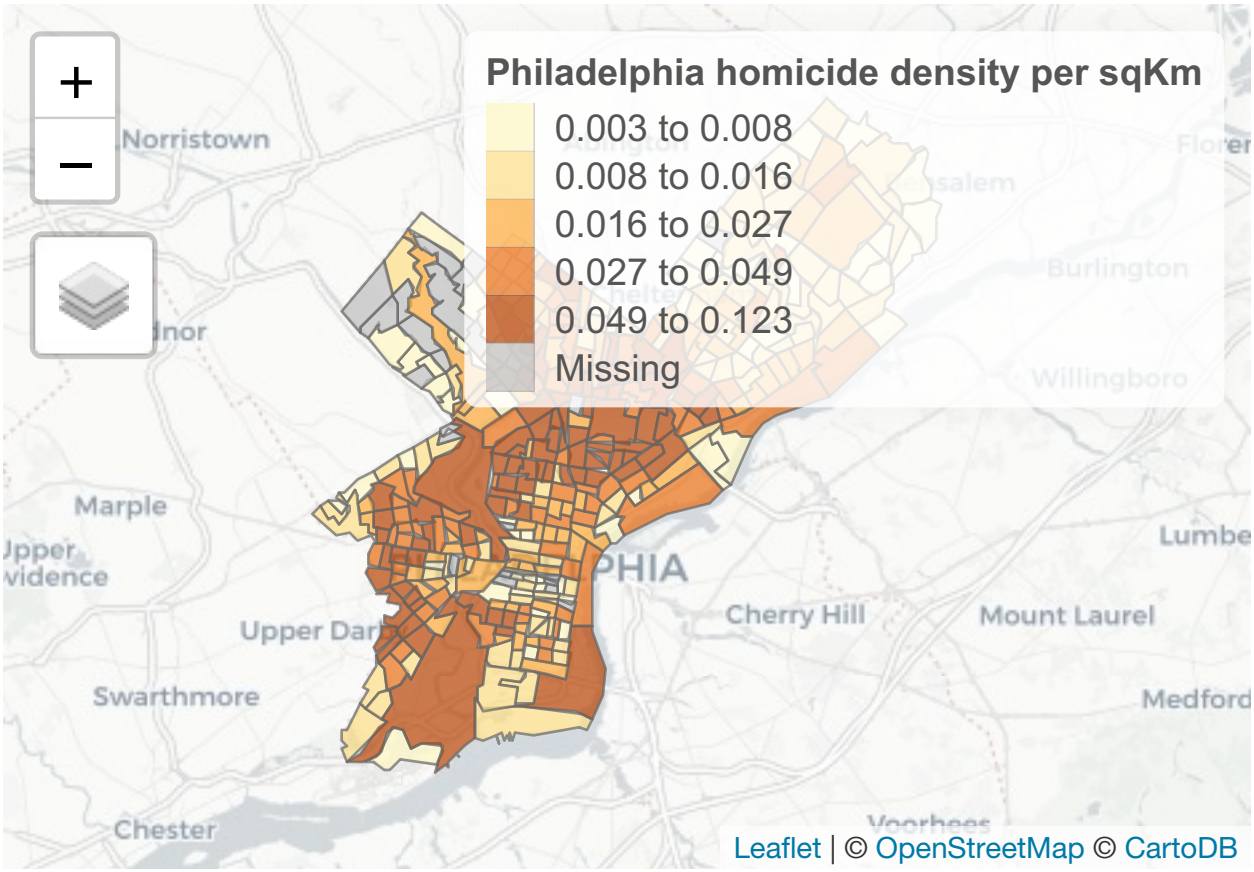
- tm_shape() where we provide
  - the SpatialPolygonsDataframe (we could also provide an sf object)
- tm_polygons() where we set
  - the attribute variable to map,
  - the break style, and
  - a title.

```
library(tmap)
tm_shape(philly_crimer_sp) +
  tm_polygons("homic_dens",
              style="quantile",
              title="Philadelphia \nhomicide density \nper sqKm")
```



`tmap` has a very nice feature that allows us to give basic interactivity to the map. We can switch from "plot" mode into "view" mode and call the last plot, like so:

```
tmap_mode("view")
last_map()
```



Cool huh?

The `tmap` library also includes functions for simple spatial operations, geocoding and reverse geocoding using OSM. For more check `vignette("tmap-nutshell")`.

3.6 Web mapping with `leaflet`

`leaflet` provides bindings to the 'Leaflet' JavaScript library, "the leading open-source JavaScript library for mobile-friendly interactive maps". We have already seen a simple use of leaflet in the `tmap` example.

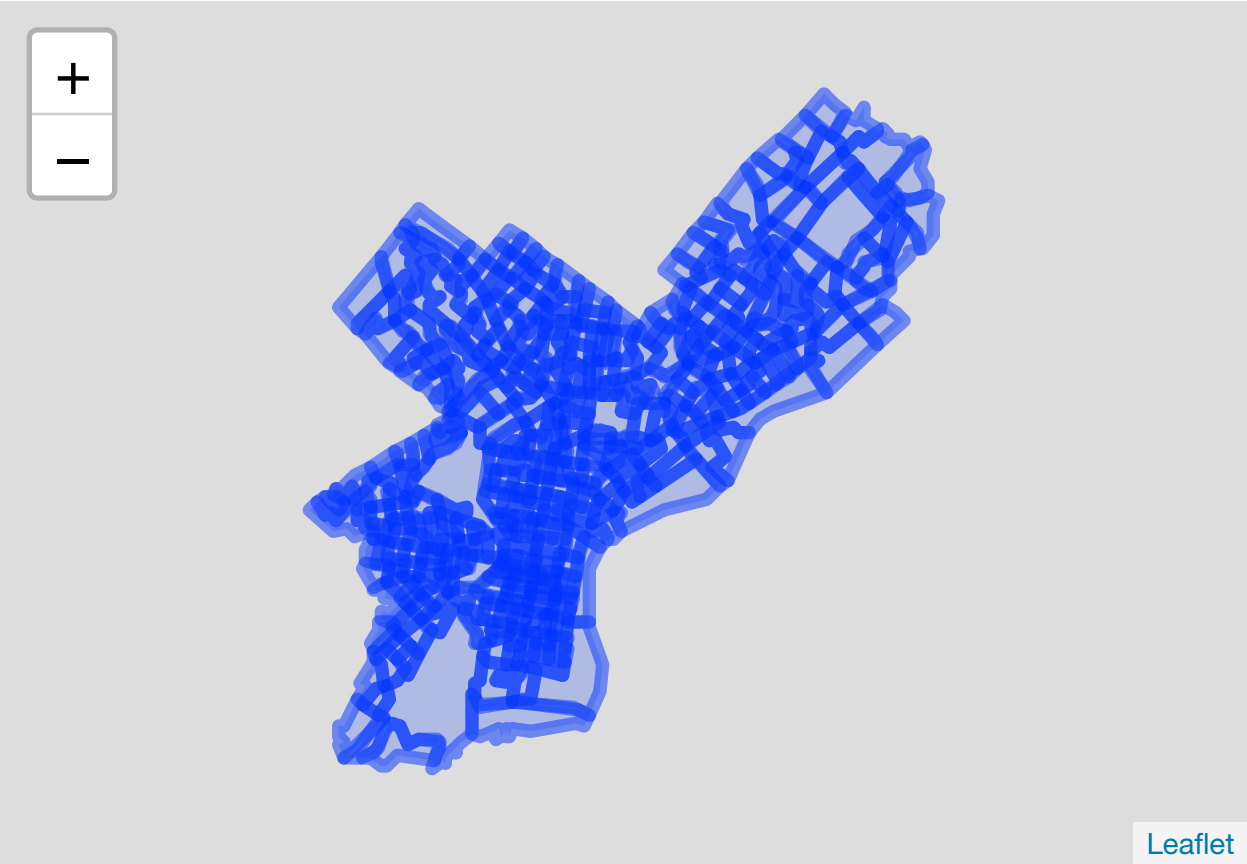
The good news is that the `leaflet` library gives us loads of options to customize the web look and feel of the map.

The bad news is that the `leaflet` library gives us loads of options to customize the web look and feel of the map.

Let's build up the map step by step.

First we load the `leaflet` library. Use the `leaflet()` function with a `Spatial*` or `sp` object and pipe it to `addPolygons()` function. It is not required, but improves readability if you use the pipe operator `%>%` to chain the elements together when building up a map with `leaflet`.

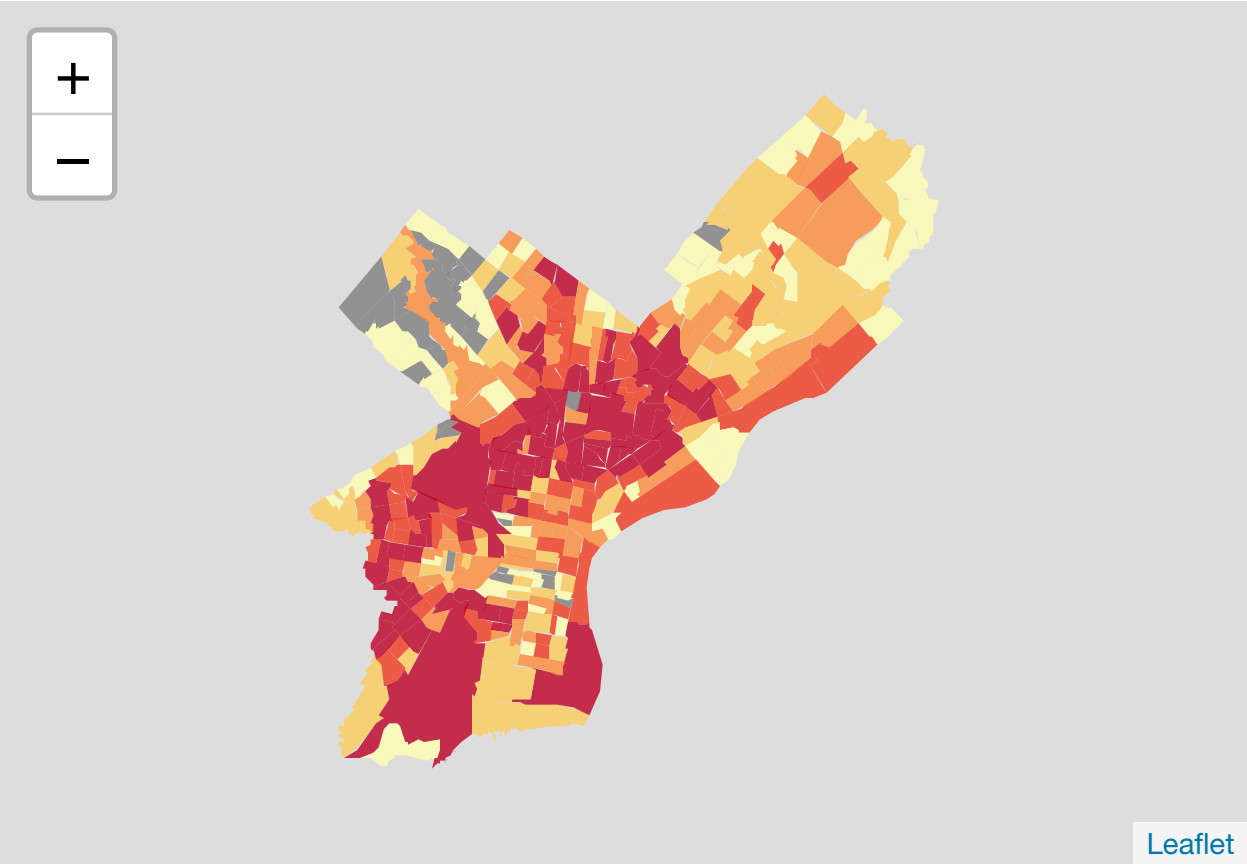
```
library(leaflet)

leaflet(philly_WGS84) %>%
  addPolygons()
```



To map the homicide density we use `addPolygons()` and:

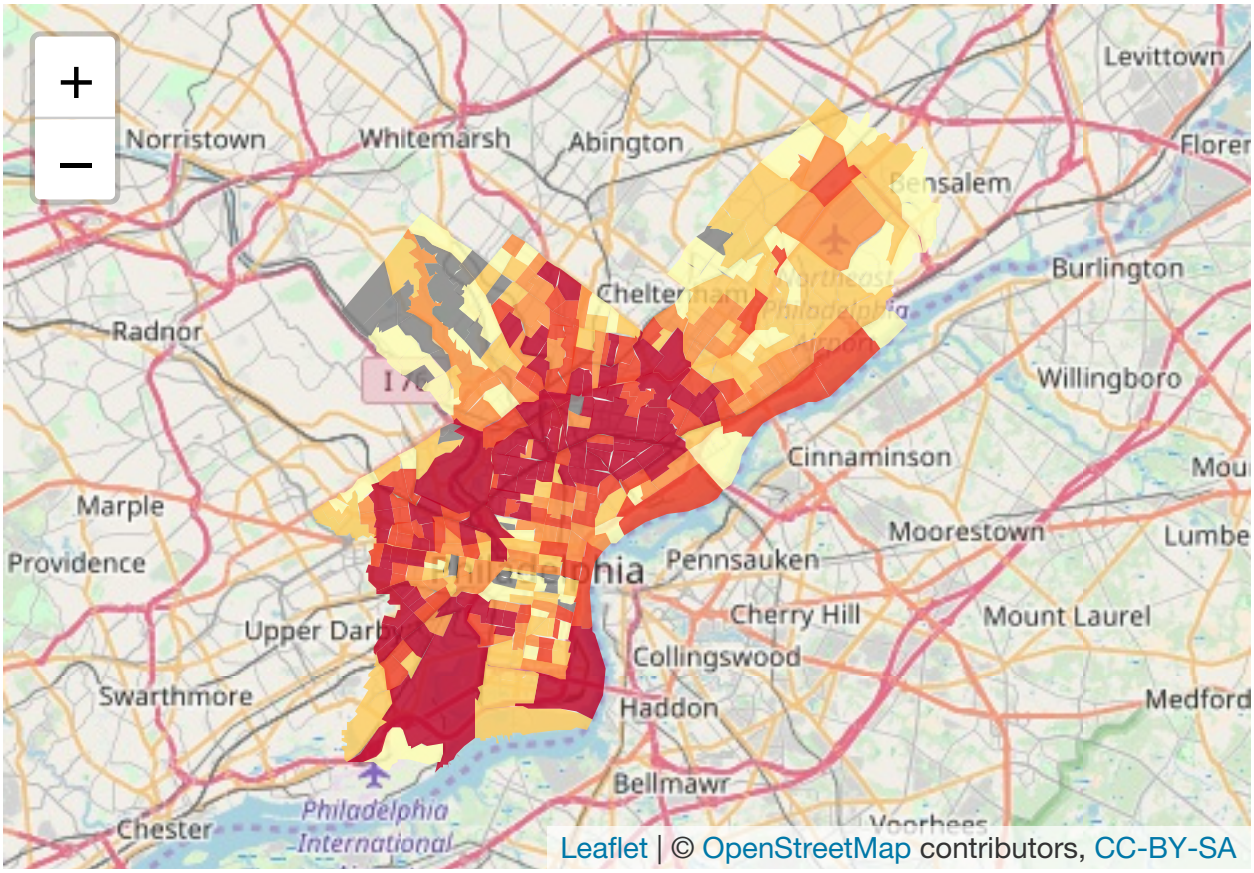
- remove stroke (polygon borders)
- set a fillColor for each polygon based on `homic_dens` and make it look nice by adjusting fillOpacity and smoothFactor (how much to simplify the polyline on each zoom level). The fill color is generated using `leaflet`'s `colorQuantile()` function, which takes the color scheme and the desired number of classes. To constuct the color scheme `colorQuantile()` returns a function that we supply to `addPolygons()` together with the name of the attribute variable to map.
- add a popup with the `homic_dens` values. We will create as a vector of strings, that we then supply to `addPolygons()`.

```
pal_fun <- colorQuantile("YlOrRd", NULL, n = 5)

p_popup <- paste0("<strong>Homicide Density: </strong>", philly_WGS84$homic_dens)

leaflet(philly_WGS84) %>%
  addPolygons(
    stroke = FALSE, # remove polygon borders
    fillColor = ~pal_fun(homic_dens), # set fill color with function from above and value
    fillOpacity = 0.8, smoothFactor = 0.5, # make it nicer
    popup = p_popup)  # add popup
```



Here we add a basemap, which defaults to OSM, with `addTiles()`

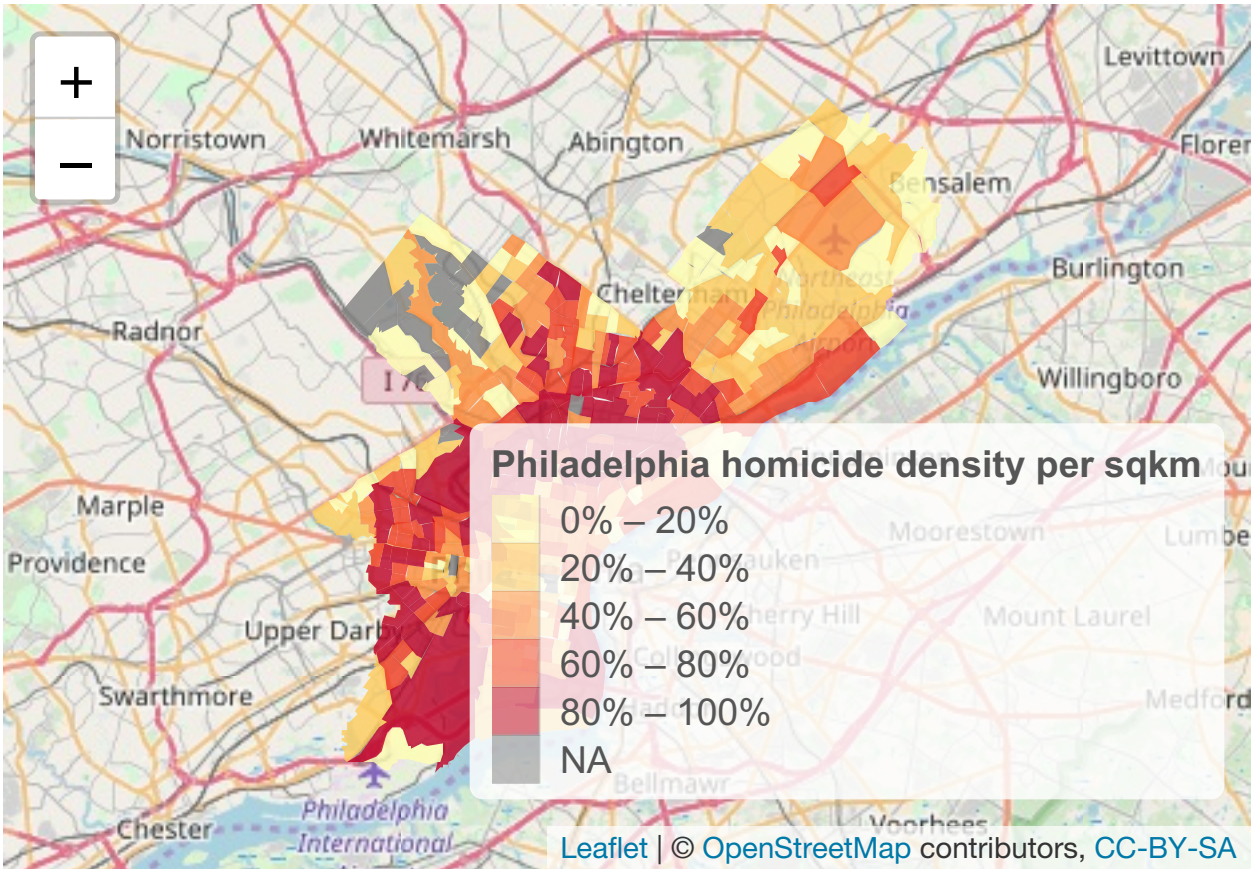
```
leaflet(philly_WGS84) %>%
  addPolygons(
    stroke = FALSE,
    fillColor = ~pal_fun(homic_dens),
    fillOpacity = 0.8, smoothFactor = 0.5,
    popup = p_popup) %>%
  addTiles()
```



Lastly, we add a legend. We will provide the `addLegend()` function with:

- the location of the legend on the map
- the function that creates the color palette
- the value we want the palette function to use
- a title

```
leaflet(philly_WGS84) %>%
  addPolygons(
    stroke = FALSE,
    fillColor = ~pal_fun(homic_dens),
    fillOpacity = 0.8, smoothFactor = 0.5,
    popup = p_popup) %>%
  addTiles() %>%
  addLegend("bottomright",  # location
            pal=pal_fun,    # palette function
            values=~homic_dens,  # value to be passed to palette function
            title = 'Philadelphia homicide density per sqkm') # legend title
```

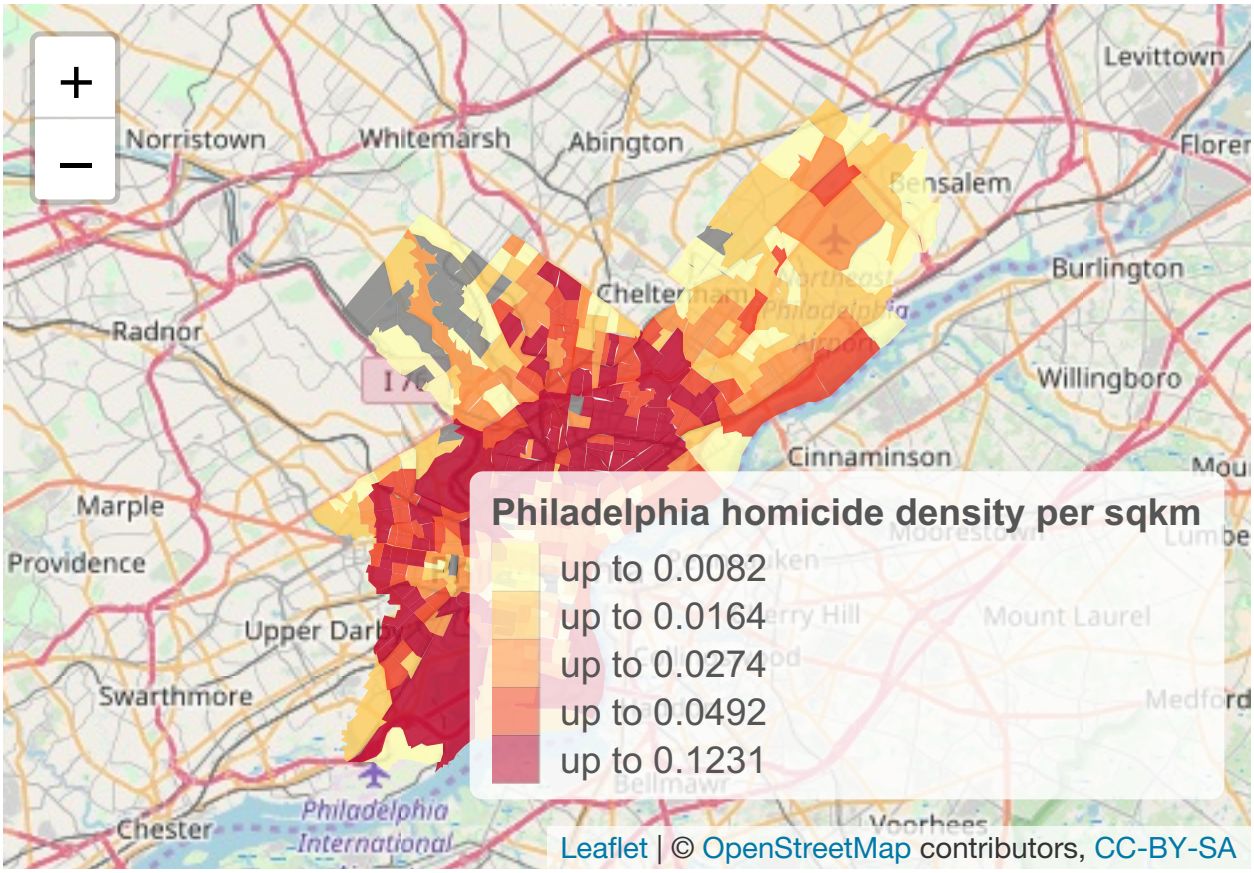


The labels of the legend show percentages instead of the actual value breaks[6].

To set the labels for our breaks manually we replace the `pal` and `values` with the `colors` and `labels` arguments and set those directly using `brewer.pal()` and `breaks_qt` from an earlier section above.

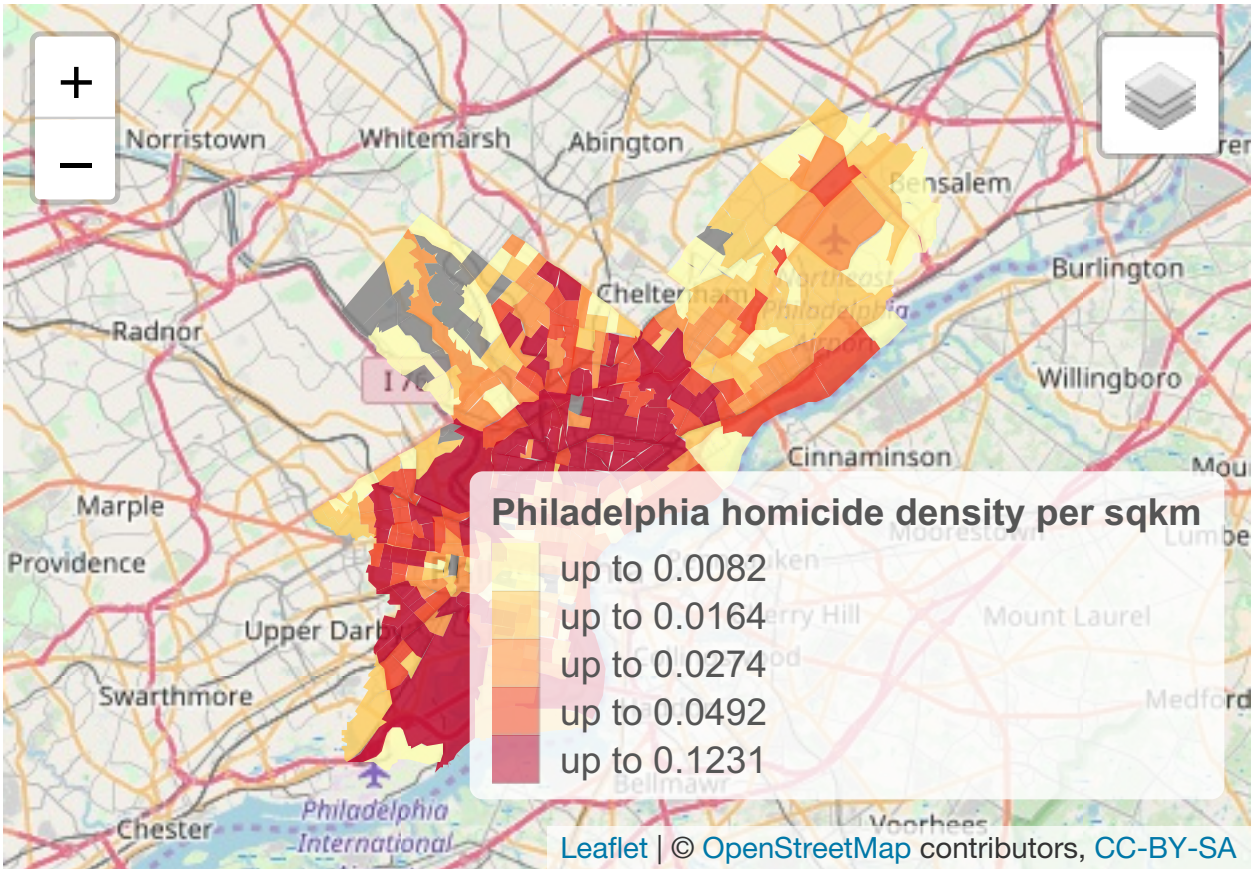
```
leaflet(philly_WGS84) %>%
  addPolygons(
    stroke = FALSE,
    fillColor = ~pal_fun(homic_dens),
    fillOpacity = 0.8, smoothFactor = 0.5,
    popup = p_popup) %>%
  addTiles() %>%
  addLegend("bottomright",
            colors = brewer.pal(5, "YlOrRd"),
            labels = paste0("up to ", format(breaks_qt$brks[-1], digits = 2)),
            title =  'Philadelphia homicide density per sqkm')
```

[6]The formatting is set with `labFormat()` and in the documentation we discover that: "By default, `labFormat` is basically `format(scientific = FALSE,big.mark = ',')` for the numeric palette, `as.character()` for the factor palette, and a function to return labels of the form `x[i] - x[i + 1]` for bin and quantile palettes (**in the case of quantile palettes, x is the probabilities instead of the values of breaks**)."



That's more like it. Finally, I have added for you a control to switch to another basemap and turn the philly polygon off and on. Take a look at the changes in the code below.

```
leaflet(philly_WGS84) %>%
  addPolygons(
    stroke = FALSE,
    fillColor = ~pal_fun(homic_dens),
    fillOpacity = 0.8, smoothFactor = 0.5,
    popup = p_popup,
    group = "philly") %>%
  addTiles(group = "OSM") %>%
  addProviderTiles("CartoDB.DarkMatter", group = "Carto") %>%
  addLegend("bottomright",
            colors = brewer.pal(5, "YlOrRd"),
            labels = paste0("up to ", format(breaks_qt$brks[-1], digits = 2)),
            title = 'Philadelphia homicide density per sqkm') %>%
  addLayersControl(baseGroups = c("OSM", "Carto"),
                   overlayGroups = c("philly"))
```



If you'd like to take this further here are a few pointers.

- Leaflet for R
- Creating maps in R
- Maps in R

Here is an example using `ggplot`, `leaflet`, `shiny`, and RStudio's flexdashboard template to bring it all together.

Due to continous deleopment on this front the R mapping package landscape is a bit volatile. Here are a few others, that I have come across, but don't use.

An alternative you may want to be aware of the `GISTools` package. It has not been updated for a while, but it can be quite convenient for choropleth plotting. It's `choropleth()` function is a convenience functions that wraps around `spplot()` Currently `GISTools` cannot understand `sf` objects.

The `leafletR` package does similar things, but requires the spatial object to be in GeoJSON/TopoJSON format. It also has not been updated for a while. My reason for using `leaflet` is that it integrates well with RStudio, Shiny, and R Markdown and can take `sf` objects.