

Using Spatial Data with R

Claudia A Engel

Last updated: February 11, 2019

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 | 14 |
| 1.3 Loading shape files into R | 16 |
| 1.4 Raster data in R | 20 |
| 2 Spatial data manipulation in R | 29 |
| 2.1 Attribute Join | 29 |
| 2.2 Topological Subsetting: Select Polygons by Location | 30 |
| 2.3 Reprojecting | 33 |
| 2.4 Spatial Aggregation: Points in Polygons | 37 |
| 2.5 <code>raster</code> operations | 40 |
| 3 Making Maps in R | 41 |
| 3.1 Plotting simple features (<code>sf</code>) with <code>plot</code> | 41 |
| 3.2 Choropleth mapping with <code>spplot</code> | 45 |
| 3.3 Choropleth mapping with <code>ggplot2</code> | 47 |
| 3.4 Adding basemaps with <code>ggmap</code> | 49 |
| 3.5 Choropleth with <code>tmap</code> | 52 |
| 3.6 Web mapping with <code>leaflet</code> | 53 |

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("http://bit.ly/R-spatial-data", "R-spatial-data.zip")
unzip("R-spatial-data.zip", exdir = "data")
```

You can also download the data manually here [R-spatial-data.zip](http://bit.ly/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:

- **sf**
 - **sp**
 - **rgdal**
 - **raster**
 - **rgeos**
 - **dplyr**
- For the mapping section install and load these additional libraries:
 - **classInt**
 - **RColorBrewer**
 - **ggplot2**
 - **ggmap**
 - **tmap**
 - **leaflet**(On Mac installing binary version is ok)

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. (2019): Geocomputation with R

Spatial Data Analysis and Modeling with R

CRAN Task View: Analysis of Spatial Data

Engel, C. (2019). R for Geospatial Analysis and Mapping. The Geographic Information Science & Technology Body of Knowledge (1st Quarter 2019 Edition), John P. Wilson (Ed.). DOI:10.22224/gistbok/2019.1.3.

For a quick introduction to all things geo check out map school.

Acknowledgements

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

Chapter 1

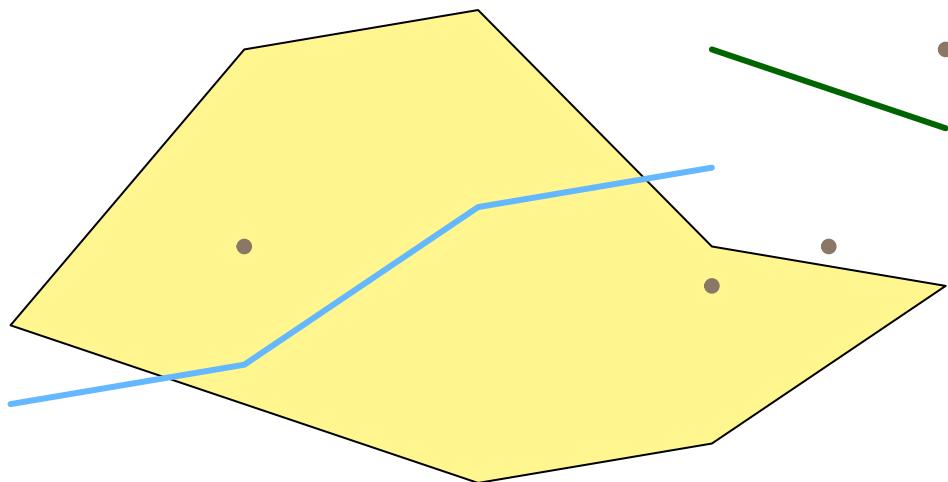
Introduction to spatial data in R

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 general package to provide classes and methods for spatial data types that was developed for 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 manually 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 coordinates² 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.

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).

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.

If, for example we wanted to build up an `sp` Object that would contain highways we would do the following.

First we would create a `Line` object that holds one highway. We use a matrix with two columns of arbitrary numbers, for x and y coordinates.

¹R Bivand (2011) Introduction to representing spatial objects in R

²Coordinates should be of type double and will be promoted if not.

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

#> Formal class 'Line' [package "sp"] with 1 slot
#> ..@ coords: num [1:3, 1:2] 0.449 0.94 0.249 0.435 0.208 ...
```

Note the @ coords slot which holds the coordinates.

Ok, let's create another Line object for another highway:

```
ln2 <- Line(matrix(runif(6), ncol=2))
```

Now we combine the two highways to a Lines object. Note how we add a unique ID for each highway. This step allows to combine generate multiple line strings, so we could add more lines under the same ID.

```
lns1 <- Lines(list(ln1, ID = c("hwy1"))
lns2 <- Lines(list(ln2, ID = c("hwy2")))
str(lns1)
```

```
#> 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] 0.449 0.94 0.249 0.435 0.208 ...
#> ... @ ID : chr "hwy1"
```

The Line objects are now in a list and we have an additional ID slot, which uniquely identifies each Line object.

Now we turn this into a geospatial object by creating a SpatialLines object:

```
sp_lns <- SpatialLines(list(lns1, lns2))
str(sp_lns)
```

```
#> Formal class 'SpatialLines' [package "sp"] with 3 slots
#> ..@ lines :List of 2
#> ... .$. :Formal class 'Lines' [package "sp"] with 2 slots
#> ... .$. .@ Lines:List of 1
#> ... .$. .$. :Formal class 'Line' [package "sp"] with 1 slot
#> ... .$. .$. .@ coords: num [1:3, 1:2] 0.449 0.94 0.249 0.435 0.208 ...
#> ... .$. .$. @ ID : chr "hwy1"
#> ... .$. :Formal class 'Lines' [package "sp"] with 2 slots
#> ... .$. .@ Lines:List of 1
#> ... .$. .$. :Formal class 'Line' [package "sp"] with 1 slot
#> ... .$. .$. .@ coords: num [1:3, 1:2] 0.3428 0.928 0.0464 0.6182 0.5911 ...
#> ... .$. .$. @ ID : chr "hwy2"
#> ... @ bbox : num [1:2, 1:2] 0.0464 0.208 0.9402 0.6182
#> ... .- 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
```

Note how this adds the @ bbox with the bounding box coordinates and @ proj4string to hold the Coordinate Reference System - in our case NA as we have not assigned any projection.

Finally we create some attributes to each highway and create a SpatialLinesDataframe. The way we do this is that we create a regular data.frame and join it to the spatial object via the unique ID.

```

dfr <- data.frame(id = c("hwy1", "hwy2"), # note how we use the same IDs from above!
                   cars_per_hour = c(78, 22))
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': 2 obs. of 2 variables:
#>     ...$ id       : Factor w/ 2 levels "hwy1","hwy2": 1 2
#>     ...$ cars_per_hour: num [1:2] 78 22
#>   ..@ lines     :List of 2
#>     ...$ :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] 0.449 0.94 0.249 0.435 0.208 ...
#>           ...$ ID   : chr "hwy1"
#>         ...$ :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] 0.3428 0.928 0.0464 0.6182 0.5911 ...
#>           ...$ ID   : chr "hwy2"
#>   ..@ bbox      : num [1:2, 1:2] 0.0464 0.208 0.9402 0.6182
#>   ..- 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

```

Note the additional `@ data` slot here, where we find the attribute information.

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

| function | and what it does |
|----------------------------|-----------------------------------------------------------------------------------------|
| <code>bbox()</code> | returns the bounding box coordinates |
| <code>proj4string()</code> | sets or retrieves projection attributes as object of the CRS class. |
| <code>CRS()</code> | creates an object of class of coordinate reference system arguments |
| <code>spplot()</code> | plots a separate map of all the attributes unless specified otherwise |
| <code>coordinates()</code> | set or retrieve the spatial coordinates. For spatial polygons it returns the centroids. |
| <code>over(a, b)</code> | used for example to retrieve the polygon or grid indices on a set of points |
| <code>spsample()</code> | 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 (Fig 1.1 and 1.2)

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

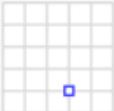
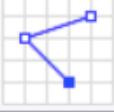
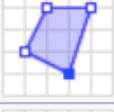
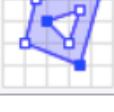
| Geometry primitives (2D) | | |
|--------------------------|-----------------------------------------------------------------------------------|-----------------------------------------------------------------------------|
| Type | Examples | |
| Point |  | POINT (30 10) |
| LineString |  | LINESTRING (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)

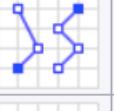
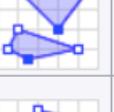
| Multipart geometries (2D) | | |
|---------------------------|-------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 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))), ((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)

`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 geometry 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 `sf` object manually the basic steps would be:

I. Create geometric objects (topology)

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.b Similarly to `sp` there are functions that help create simple feature geometries, like `st_point()`, `st_linestring()`, `st_polygon()` and more.

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

The feature geometries are then combined into a Simple Feature Collection with `st_sfc()`. which is nothing other than a simple feature geometry list-column. The `sfc` object also holds the bounding box and the projection information.

III. Add attributes.

Lastly, we add the attributes to the the simple feature collection with the `st_sf()` function. This function extends the well known data frame in R with a column that holds the simple feature collection.

So if we created the same highway object from above as `sf` object we would first generate LINESTRINGS as simple feature geometries out of a matrix with coordinates:

```
lnstr_sfg1 <- st_linestring(matrix(runif(6), ncol=2))
lnstr_sfg2 <- st_linestring(matrix(runif(6), ncol=2))
class(lnstr_sfg1)
```

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

We would then combine this into a simple feature collection :

```
(lnstr_sfc <- st_sfc(lnstr_sfg1, lnstr_sfg2)) # just one feature here
```

```
#> Geometry set for 2 features
#> geometry type:  LINESTRING
#> dimension:      XY
#> bbox:            xmin: 0.1472823 ymin: 0.03795971 xmax: 0.6951138 ymax: 0.9930134
#> epsg (SRID):    NA
#> proj4string:    NA

#> LINESTRING (0.6951138 0.3978693, 0.2638811 0.03...
#> LINESTRING (0.4910172 0.1953684, 0.4145963 0.30...
```

And lastly use our data frame from above to generate the `sf` object:

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

```
#> Simple feature collection with 2 features and 2 fields
#> geometry type:  LINESTRING
#> dimension:      XY
#> bbox:            xmin: 0.1472823 ymin: 0.03795971 xmax: 0.6951138 ymax: 0.9930134
#> epsg (SRID):    NA
#> proj4string:    NA
#>   id cars_per_hour          lnstr_sfc
#> 1 hwy1             78 LINESTRING (0.6951138 0.397...
```

```
#> 2 hwy2           22 LINESTRING (0.4910172 0.195...
```

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

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

```
#> [1] [           [[<-          $<-
#> [4] aggregate      anti_join      arrange
#> [7] as.data.frame  cbind         coerce
#> [10] dbDataType     dbWriteTable   distinct
#> [13] extent         extract        filter
#> [16] full_join     group_by      identify
#> [19] initialize    inner_join    left_join
#> [22] mask          merge         mutate
#> [25] plot          print         raster
#> [28] rasterize     rbind         rename
#> [31] right_join    sample_frac  sample_n
#> [34] select         semi_join    show
#> [37] slice          slotsFromS3 st_agr
#> [40] st_agr<-
#> [43] st_bbox         st_area        st_as_sf
#> [46] st_cast         st_boundary   st_buffer
#> [49] st_convex_hull st_centroid   st_collection_extract
#> [52] st_crs          st_coordinates st_crop
#> [55] st_geometry     st_crs<-
#> [58] st_is           st_geometry<-
#> [61] st_node         st_line_merge st_intersection
#> [64] st_precision    st_point_on_surface st_nearest_points
#> [67] st_simplify    st_segmentize st_polygonize
#> [70] st_transform    st_snap        st_set_precision
#> [73] st_voronoi     st_triangulate st_sym_difference
#> [76] st_zm           st_wrap_dateline st_union
#> [79] ungroup         summarise    st_write
#> 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
- spatial functions that rely on GEOS and GDAL and PROJ external libraries are directly linked into the package, so no need to load additional external packages (like in `sp`)
- `sf` objects can be plotted directly with `ggplot`
- `sf` directly reads from and writes to spatial **databases** such as PostGIS
- `sf` is compatible with the `tidyverse` approach, (but see some pitfalls here)

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`).

There are packages specifically for the GeoJSON and for that reason are more lightweight, for example:

- `geojson` and
- `geoops` - (demo)

Usually you can find functions that convert 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 geometry 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.

1.2.1 With `sf`

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)
```

`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 data frame in R and name it `philly_homicides_df`.

```
philly_homicides_df <- read.csv("data/philly_homicides.csv")
str(philly_homicides_df)
```

```
#> 'data.frame': 3883 obs. of 10 variables:
#> $ DC_DIST      : int 22 1 1 1 1 1 1 1 1 ...
#> $ SECTOR       : Factor w/ 30 levels "1","2","3","4",...: 1 6 6 6 9 10 6 14 14 5 ...
#> $ DISPATCH_DATE: Factor w/ 2228 levels "2006-01-01","2006-01-02",...: 2139 10 62 90 125 132 133 ...
#> $ DISPATCH_TIME: Factor w/ 1263 levels "00:00:00","00:01:00",...: 799 1 804 561 633 981 1224 86 ...
#> $ LOCATION_BLOCK: Factor w/ 3305 levels " 100 E CHAMPLOST AVE",...: 595 745 3135 806 828 588 579 ...
#> $ UCR_GENERAL   : int 100 100 100 100 100 100 100 100 100 ...
#> $ OBJ_ID        : int 1 1 1 1 1 1 1 1 1 ...
#> $ TEXT_GENERAL_CODE: Factor w/ 4 levels "Homicide - Criminal",...: 1 1 1 1 1 2 1 1 1 ...
#> $ POINT_X        : num -75.2 -75.2 -75.2 -75.2 -75.2 ...
#> $ POINT_Y        : num 40 39.9 39.9 39.9 39.9 ...
```

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

```
philly_homicides_sf <- st_as_sf(philly_homicides_df, coords = c("POINT_X", "POINT_Y"))
str(philly_homicides_sf)
```

```
#> Classes 'sf' and 'data.frame': 3883 obs. of 9 variables:
```

```
#> $ DC_DIST      : int  22 1 1 1 1 1 1 1 1 1 ...
#> $ SECTOR       : Factor w/ 30 levels "1","2","3","4",...: 1 6 6 6 9 10 6 14 14 5 ...
#> $ DISPATCH_DATE : Factor w/ 2228 levels "2006-01-01","2006-01-02",...: 2139 10 62 90 125 132 133 ...
#> $ DISPATCH_TIME : Factor w/ 1263 levels "00:00:00","00:01:00",...: 799 1 804 561 633 981 1224 86 ...
#> $ LOCATION_BLOCK : Factor w/ 3305 levels " 100 E CHAMPLOST AVE",...: 595 745 3135 806 828 588 579 8 ...
#> $ UCR_GENERAL   : int  100 100 100 100 100 100 100 100 100 ...
#> $ OBJ_ID        : int  1 1 1 1 1 1 1 1 1 1 ...
#> $ TEXT_GENERAL_CODE: Factor w/ 4 levels "Homicide - Criminal",...: 1 1 1 1 1 2 1 1 1 1 ...
#> $ geometry       :sfc_POINT of length 3883; first list element: 'XY' num -75.2 40
#> - attr(*, "sf_column")= chr "geometry"
#> - attr(*, "agr")= Factor w/ 3 levels "constant","aggregate",...: NA NA NA NA NA NA NA NA
#> .. - attr(*, "names")= chr "DC_DIST" "SECTOR" "DISPATCH_DATE" "DISPATCH_TIME" ...
```

Note the additional **geometry** list-column which now holds the simple feature collection with the coordinates of all the points.

To make it a complete geographical object we assign the WGS84 projection, which has the EPSG code 4326:

```
st_crs(philly_homicides_sf)
```

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

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

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

```
#>   EPSG: 4326
```

```
#>   proj4string: "+proj=longlat +datum=WGS84 +no_defs"
```

We will save this object as a shapefile on our hard drive for later use. (Note that by default **st_write** checks if the file already exists, and if so it will not overwrite it. If you need to force it to overwrite use the option **delete_layer = TRUE**.)

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

```
# to force the save:
```

```
st_write(philly_homicides_sf, "data/PhillyHomicides", driver = "ESRI Shapefile", delete_layer = TRUE)
```

1.2.2 With sp

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.

Below, we convert the **philly_homicides_df** data frame into a spatial object with using the **coordinates** function and check with **class(philly_homicides_df)** again to examine which object class the table belongs to now. Note that the **coordinates()** function if used in this way replaces the original data frame.

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

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

Assigning the projection:

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

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

```
proj4string(philly_homicides_df) <- CRS("+init=epsg:4326") # this is WGS84
is.projected(philly_homicides_df) # voila! hm. wait a minute..
```

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

To save the `sp` object out as a shapefile we need to load another library, called `rgdal` (more on this below.)

```
# to save out using writeOGR from rgdal
library(rgdal)
```

```
# note that we need to save the philly_homicides_df, which we converted to sp object!
writeOGR(philly_homicides_df, "data/PhillyHomicides", "PhillyHomcides", driver = "ESRI Shapefile")
# to force save:
writeOGR(philly_homicides_df, "data/PhillyHomicides", "PhillyHomcides", driver = "ESRI Shapefile", overv
```

1.3 Loading shape files into R

1.3.1 How to do this in sf

`sf` relies on the powerful GDAL library, which is automatically linked in when loading `sf`. We can use `st_read()`, which simply takes the path of the directory with the shapefile as argument.

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

```
#> Reading layer `PhillyTotalPopHHinc` from data source `/Users/cengel/Anthro/R_Class/R_Workshops/R-spatial-0.6-1`
#> 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
# take a look at what we've got
str(philly_sf) # note again the geometry column
```

```
#> Classes 'sf' and 'data.frame':  384 obs. of  18 variables:
#> $ STATEFP10 : Factor w/ 1 level "42": 1 1 1 1 1 1 1 1 1 ...
#> $ COUNTYFP10: Factor w/ 1 level "101": 1 1 1 1 1 1 1 1 1 ...
#> $ TRACTCE10 : Factor w/ 384 levels "000100","000200",...: 347 350 353 329 326 345 46 82 173 15 ...
#> $ GEOID10   : Factor w/ 384 levels "42101000100",...: 347 350 353 329 326 345 46 82 173 15 ...
#> $ NAME10    : Factor w/ 384 levels "1","10.01","10.02",...: 281 284 287 262 259 279 299 354 86 3 ...
#> $ NAMELSAD10: Factor w/ 384 levels "Census Tract 1",...: 281 284 287 262 259 279 299 354 86 3 ...
#> $ MTFCC10   : Factor w/ 1 level "G5020": 1 1 1 1 1 1 1 1 ...
#> $ FUNCSTAT10: Factor w/ 1 level "S": 1 1 1 1 1 1 1 1 ...
#> $ ALAND10   : num  2322732 4501110 1004313 1271533 1016206 ...
```

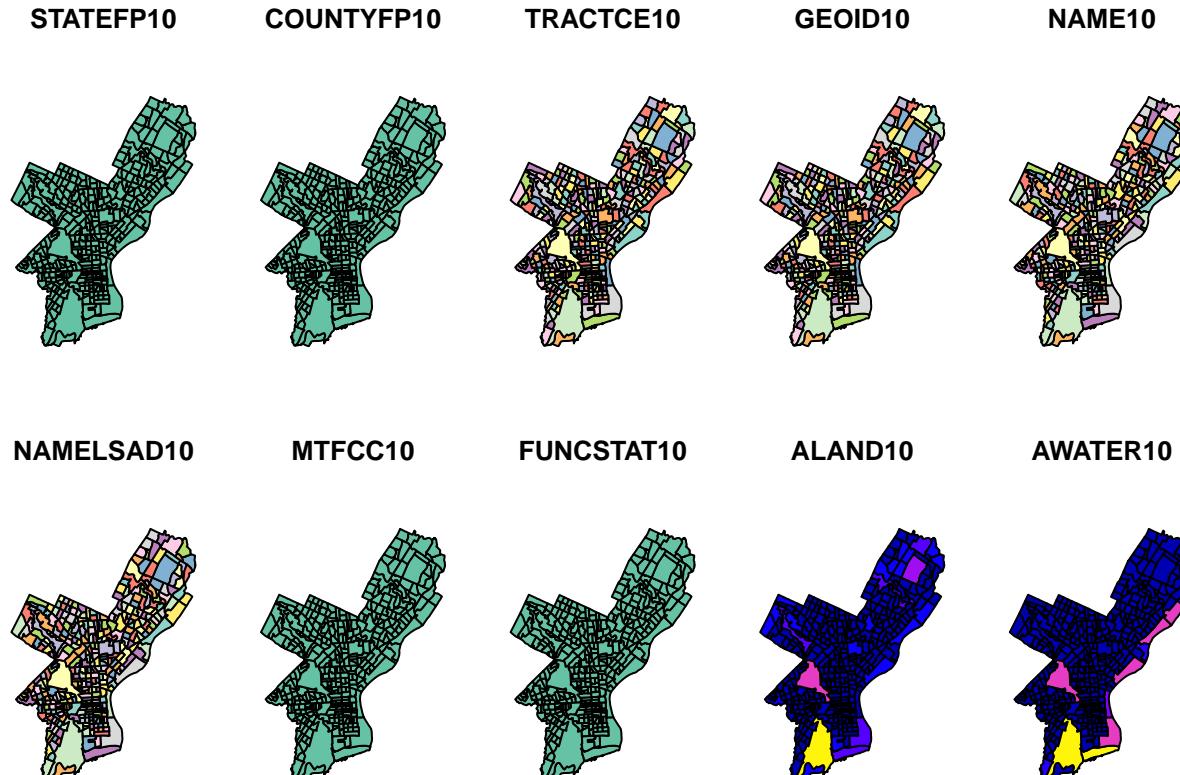
```
#> $ AWATER10 : num 66075 8014 1426278 8021 0 ...
#> $ INTPTLAT10: Factor w/ 384 levels "+39.8798897",...: 369 380 68 333 325 368 16 93 188 63 ...
#> $ INTPTLON10: Factor w/ 384 levels "-074.9667387",...: 1 5 137 14 27 4 272 367 119 147 ...
#> $ GISJOIN : Factor w/ 384 levels "G4201010000100",...: 347 350 353 329 326 345 46 82 173 15 ...
#> $ Shape_area: num 2388806 4509124 2430591 1279556 1016207 ...
#> $ Shape_len : num 6851 10567 9257 4928 5920 ...
#> $ medHHinc : num 54569 NA 130139 56667 69981 ...
#> $ totalPop : num 3695 703 1643 4390 3807 ...
#> $ geometry :sfc_MULTIPOLYGON of length 384; first list element: List of 1
#> ...$ :List of 1
#> ...$ : num [1:55, 1:2] 1763647 1763473 1763366 1763378 1763321 ...
#> ..- attr(*, "class")= chr "XY" "MULTIPOLYGON" "sfg"
#> - attr(*, "sf_column")= chr "geometry"
#> - attr(*, "agr")= Factor w/ 3 levels "constant", "aggregate", ...: NA NA NA NA NA NA NA NA NA ...
#> ..- attr(*, "names")= chr "STATEFP10" "COUNTYFP10" "TRACTCE10" "GEOID10" ...
```

Two more words about the geometry column: You can name this column any way you wish. Secondly, you can remove this column and revert to a regular, non-spatial data frame at any time with `st_drop_geometry()`.

The default plot of an `sf` object is a multi-plot of the first attributes, with a warning if not all can be plotted:

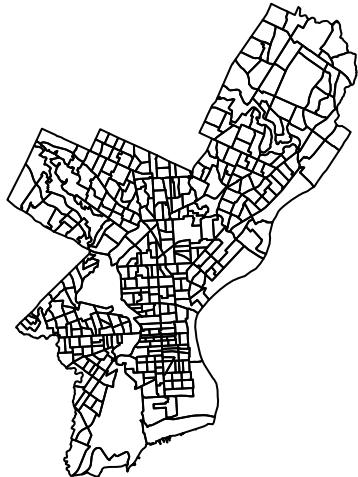
```
plot(philly_sf)
```

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



In order to only plot the polygon boundaries we need to directly use the geometry column. We use the `st_geometry()` function to extract it:

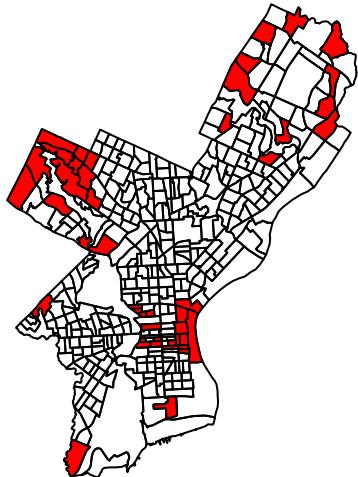
```
plot(st_geometry(philly_sf))
```



Let's add a subset of polygons with only the census tracts where the median household income is more than \$60,000. We can extract elements from an `sf` object based on attributes using your preferred method of subsetting data frames.

```
# subset the familiar way
philly_sf_rich <- philly_sf[philly_sf$medHHinc > 60000, ]
# or
philly_sf_rich <- subset(philly_sf, medHHinc > 60000)

plot(st_geometry(philly_sf_rich), add=T, col="red")
```



Piping works as well!

```
library(dplyr)
philly_sf %>%
  filter(medHHinc > 60000) %>%
  st_geometry() %>%
  plot(col="red", add=T)
```

1.3.2 How to work with rgdal and sp

In order to read spatial data into R and turn them into `Spatial*` family objects we require the `rgdal` package, which provides bindings to GDAL⁴.

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` by using the `readOGR` function⁵. We can also examine the object and confirm what it is with `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
class(philly_sp)

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

Very similarly to the above we can create a simple plot of the polygons with the `plot` command, which directly understands the `SpatialPolygonsDataFrame` object and then plot a subset of polygons with a median household income (`medHHinc`) of over \$60,000 on top of the plot of the entire city.

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

⁴GDAL supports over 200 raster formats and vector formats. Use `ogrDrivers()` and `gdalDrivers()` (without arguments) to find out which formats your `rgdal` install can handle.

⁵Unlike `read.csv` `readOGR` does not understand the `~` as valid element of a path. This (on Mac) will not work: `philly_sp <- readOGR("~/Desktop/data/Philly/", "PhillyTotalPopHHinc")`

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` package⁶ 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)

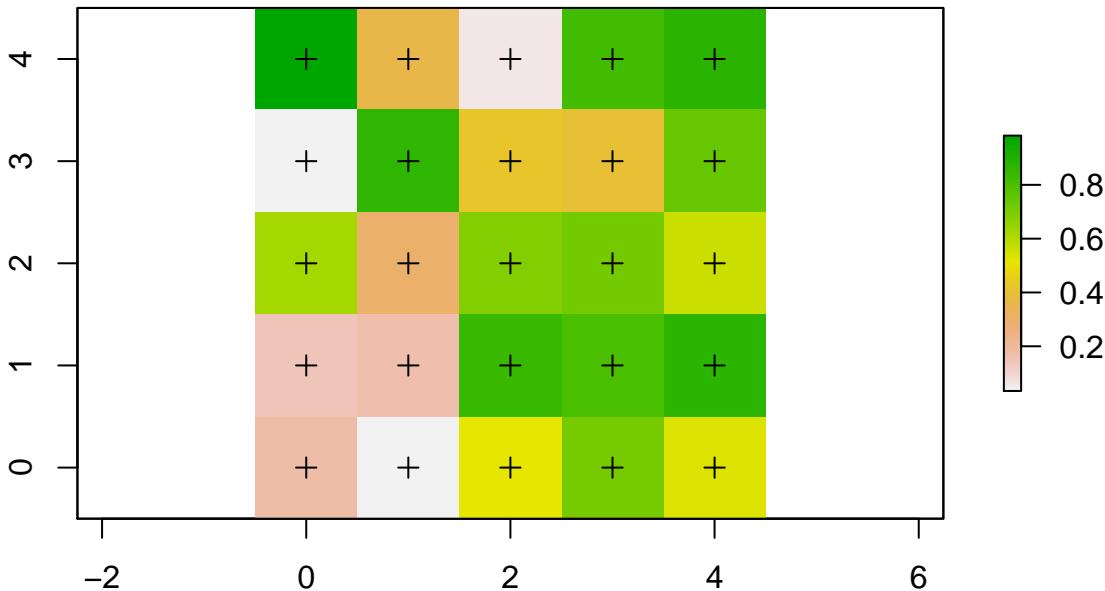
#> [1] "RasterLayer"
#> attr(,"package")
#> [1] "raster"

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

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

⁶Note that `sp` also allows to work with raster structures. The `GridTopology` class is the key element of raster representations. It contains: (a) the center coordinate pair of the south-west raster cell, (b) the two cell sizes in the metric of the coordinates, giving the step to successive centres, and (c) the numbers of cells for each dimension. There is also a `SpatialPixels` object which stores grid topology and coordinates of the actual points.

```
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")
```

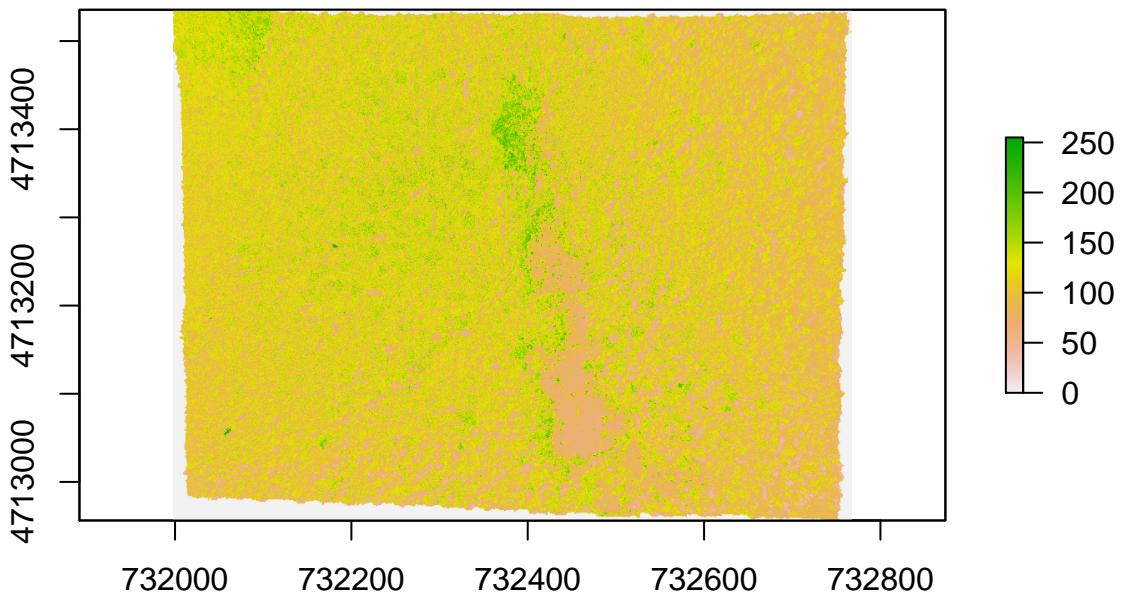
Typing the name of the object will give us what's in there:

```
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)
```



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   adjacent    aggregate  
#> [13] all.equal area       Arith       as.array  
#> [17] as.character as.data.frame as.factor  as.integer  
#> [21] as.list    as.logical  as.matrix  as.raster  
#> [25] as.vector  asFactor    atan2      bandnr  
#> [29] barplot   bbox       boundaries boxplot  
#> [33] brick     buffer     calc       cellFromRowCol  
#> [37] cellFromXY cellStats   clamp     click  
#> [41] clump     coerce     colFromCell colFromX  
#> [45] colSums   Compare    contour    coordinates  
#> [49] corLocal  cover      crop      crosstab  
#> [53] crs<-     cut       cv        density  
#> [57] dim       dim<-     direction  disaggregate  
#> [61] distance  extend    extent    extract  
#> [65] flip      focal     freq      getValues  
#> [69] getValuesBlock getValuesFocal gridDistance head  
#> [73] hist      image     interpolate intersect  
#> [77] is.factor  is.finite  is.infinite is.na  
#> [81] is.nan    isLonLat  KML       labels  
#> [85] layerize  length    levels    levels<-  
#> [89] lines     localFun  log       Logic
```

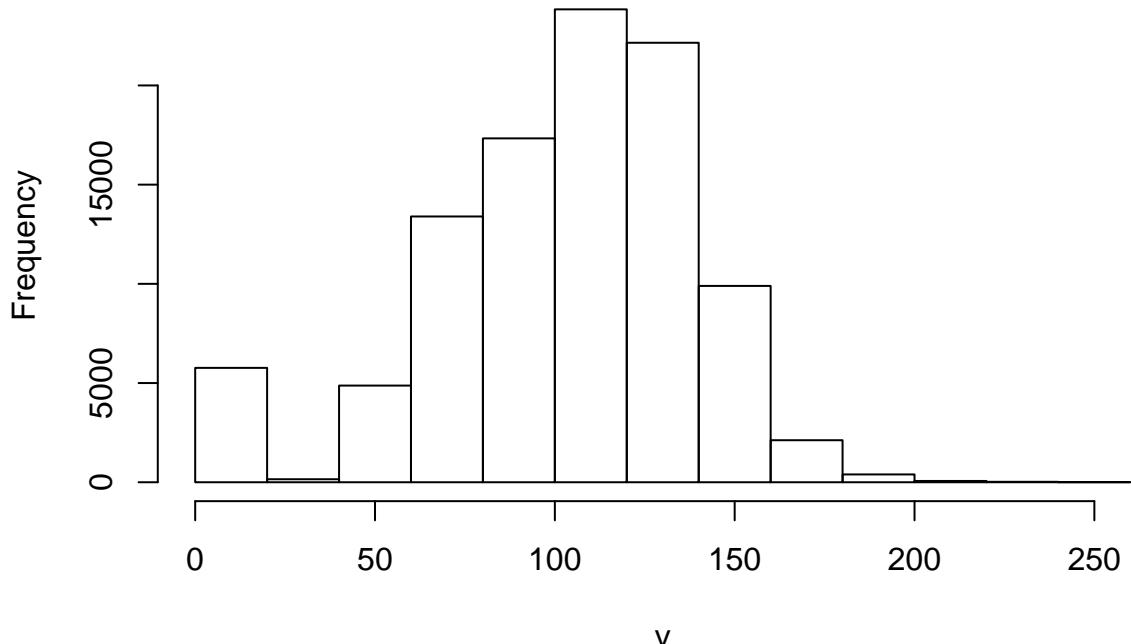
```
#> [93] mask           match       Math        Math2
#> [97] maxValue      mean        merge      minValue
#> [101] modal        mosaic     names      names<-
#> [105] ncell        ncol       ncol<-    nlayers
#> [109] nrow         nrow<-    origin    origin<-
#> [113] overlay      persp      plot      predict
#> [117] print        proj4string proj4string<- quantile
#> [121] raster       rasterize   readAll   readStart
#> [125] readStop     reclassify  res      resample
#> [129] RGB          rotate     rowColFromCell rowFromCell
#> [133] rowFromY    rowSums    sampleRandom sampleRegular
#> [137] sampleStratified scale    select    setMinMax
#> [141] setValues    shift     show      spplot
#> [145] stack        stackSelect subs    subset
#> [149] Summary      summary   t       tail
#> [153] text         trim     unique   update
#> [157] values       values<- Which    which.max
#> [161] which.min    writeRaster writeStart writeStop
#> [165] writeValues  xFromCell xFromCol  xmax
#> [169] xmin         xres      xyFromCell yFromCell
#> [173] yFromRow     ymax      ymin      yres
#> [177] zonal        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.
```

HARV_RGB_Ortho



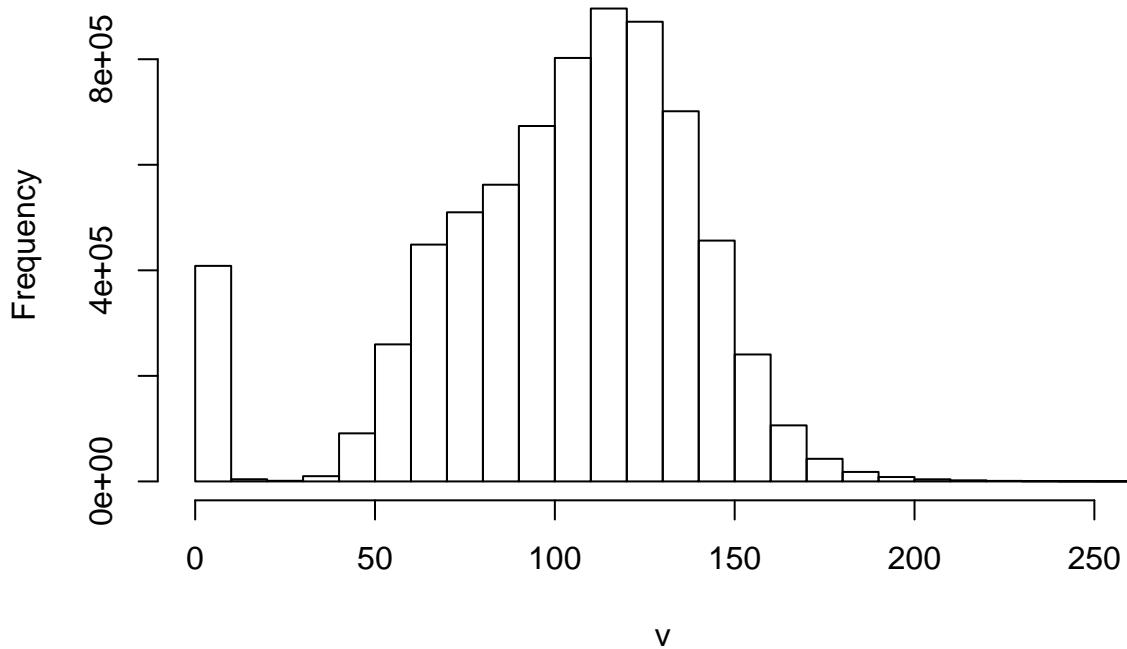
Notice that a warning message is produced when R creates the histogram.

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! We can force the `hist` function to use all cell values.

```
ncell(HARV)
```

```
#> [1] 7120141
hist(HARV, maxpixels = ncell(HARV))
```

HARV_RGB_Ortho



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")
```

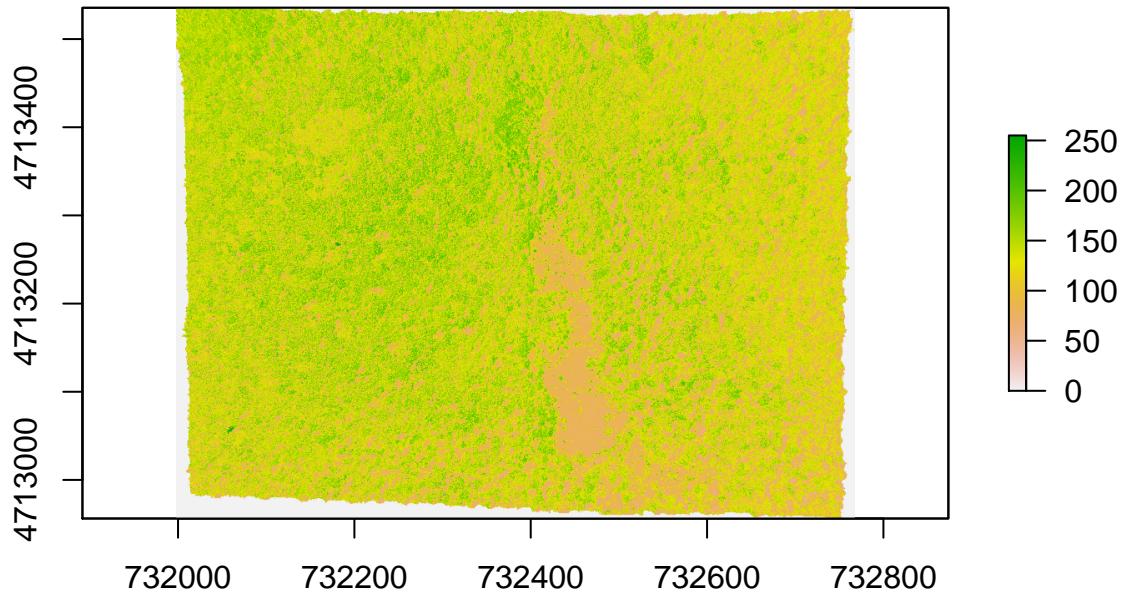
A raster dataset can contain one or more bands. We can view the number of bands in a raster using the `nlayers()` function.

```
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")

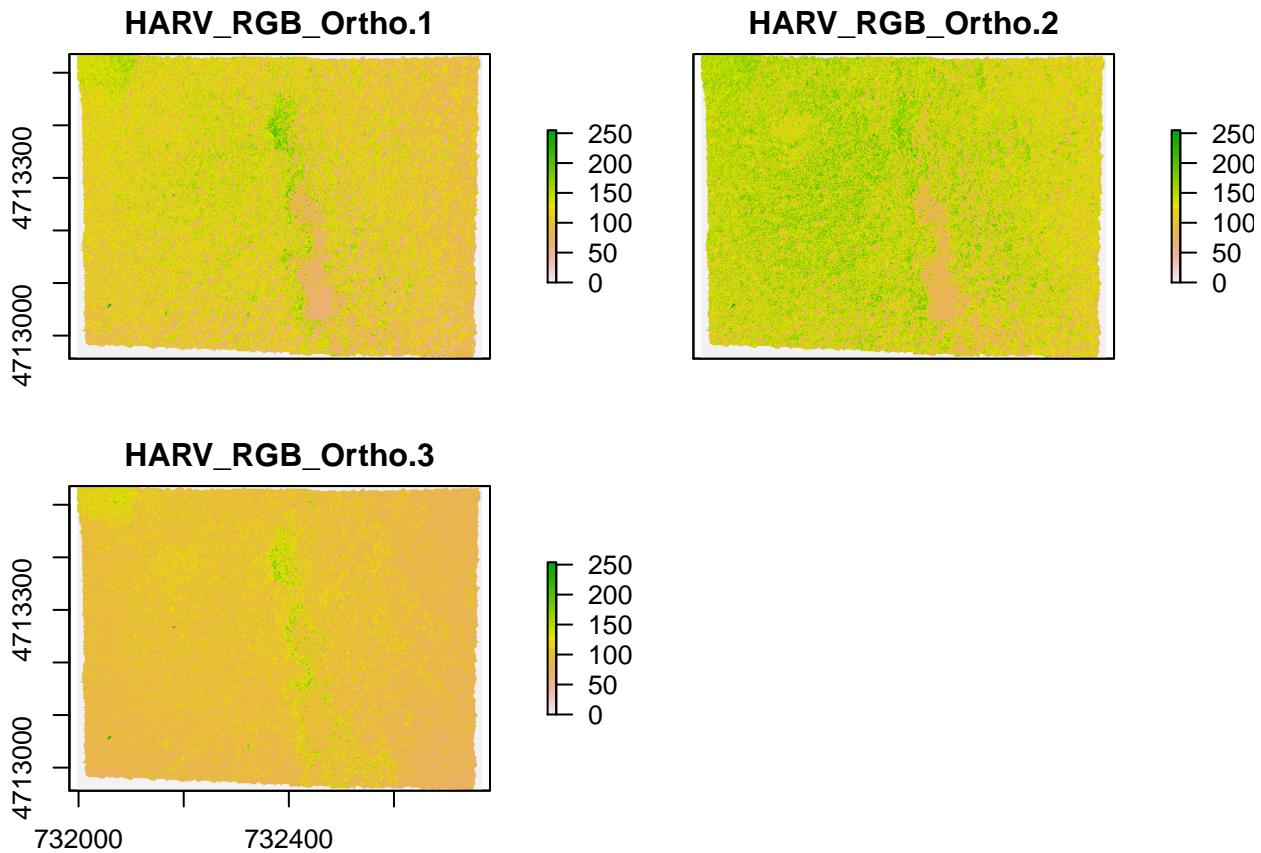
# how many layers?
nlayers(HARV_stack)
```

```
#> [1] 3
# 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

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)

#> 44248 bytes

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

#> 170897168 bytes
```

Going back to the `sp` package, 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)

#> Object of class SpatialGridDataFrame
#> Coordinates:
#>     min max
#> [1,] -0.5 4.5
#> [2,] -0.5 4.5
#> Is projected: NA
#> proj4string : [NA]
#> Grid attributes:
#>   cellcentre.offset cellsize cells.dim
#> 1           0       1      5
#> 2           0       1      5
#> Data attributes:
#>   runif.25.
#>   Min.    :0.0249
#>   1st Qu.:0.2283
#>   Median  :0.4203
#>   Mean    :0.4679
#>   3rd Qu.:0.7446
#>   Max.    :0.9867
```

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

There are a wide variety of spatial, topological, and attribute data operations you can perform with R. Lovelace et al's recent publication¹ goes into great depth about this and is highly recommended.

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

2.1 Attribute Join

An attribute join on vector data 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)².

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.

First we will 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)
```

2.1.1 How to do this in sf

If you don't have the object still loaded read the the `PhillyTotalPopHHinc` shapefile into an object named `philly_sf`. Check out the column names of `philly_sf` and of `ph_edu` to determine which one might contain the unique identifier for the join.

```
## sf ##
# if you need to read in again:
```

¹Lovelace, R., Nowosad, J., & Muenchow, J. (2019). Geocomputation with R. CRC Press.

²Per the ESRI specification a shapefile must have 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.

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

To join the ph_edu data frame with philly_sf we can use `merge` like this:

```
philly_sf_merged <- merge(philly_sf, ph_edu, by.x = "GEOID10", by.y = "GEOID")
names(philly_sf_merged)
```

```
#> [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"
```

We see the new attribute columns added, as well as the geometry column.

2.1.2 The same with `sp`

In `sp` we have a `Spatial*Dataframe` that contains the geometries and an identifying index variable for each. We combine it with a data frame, that includes the same index variable with additional variables.

The `sp` package has a `merge` command which extends the base `merge` command to work with `Spatial*` objects as arguments³.

```
## 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
```

(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.)

2.2 Topological Subsetting: 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.
```

³The `geo_join()` command from the `tigris` package also provides a convenient way to merge a data frame to a spatial data frame.

```
# 3. Create a buffer around the city center point.
# 4. Select all census tract polygons that intersect with the center buffer
```

2.2.1 Using the `sf` package

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

In addition, we need to create a `sf` Point object with the Philadelphia city center coordinates:

$$x = 1750160$$

$$y = 467499.9$$

These coordinates are in the *USA Contiguous Albers Equal Area Conic* projected CRS and the EPSG code is 102003.

With this information, we create a object that holds the coordinates of the city center. Since we don't have attributes we will just create it as a simple feature collection, `sfc`.

```
# if you need to read in again:
# philly_sf <- st_read("data/Philly/", quiet = T)

# make a simple feature point with CRS
philly_ctr_sfc <- st_sfc(st_point(c(1750160, 467499.9)), crs = 102003)
```

For the spatial operations we can recur to the suite of geometric operations that come with the `sf` package.

We create a 2km buffer around the city center point:

```
philly_buf_sf <- st_buffer(philly_ctr_sfc, 2000)
```

Ok. Now we can use that buffer to select all census tract polygons that intersect with the center buffer. In order to determine the polygons we use `st_intersects`, a geometric binary which returns a vector of logical values, which we can use for subsetting. Note the difference to `st_intersection`, which performs a geometric operation and creates a new sf object which cuts out the area of the buffer from the polygons a like cookie cutter.

Let us try this:

```
philly_buf_intersects <- st_intersects(philly_buf_sf, philly_sf)

#> Error in st_geos_binop("intersects", x, y, sparse = sparse, prepared = prepared) :
#>   st_crs(x) == st_crs(y) is not TRUE
```

Oh, what happened? Are these projections not the same?

```
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"

st_crs(philly_buf_sf)

#> Coordinate Reference System:
#>   EPSG: 102003
#>   proj4string: "+proj=aea +lat_1=29.5 +lat_2=45.5 +lat_0=37.5 +lon_0=-96 +x_0=0 +y_0=0 +datum=NAD83"
```

Ah. The difference seems to be that there is no EPSG code for `philly_sf`. Poking around the documentation we see that :

`...st_read` typically reads the coordinate reference system as `proj4string`, but not the EPSG (SRID). GDAL cannot retrieve SRID (EPSG code) from proj4string strings, and, **when needed, it has to be set by the user...**

Ok, so we need to fix this.

```
st_crs(philly_sf) <- 102003
```

```
#> Warning: st_crs<- : replacing crs does not reproject data; use st_transform
#> for that
```

This warning is ok, we know what we are doing. So now try again:

```
philly_buf_intersects <- st_intersects(philly_buf_sf, philly_sf)
class(philly_buf_intersects)
```

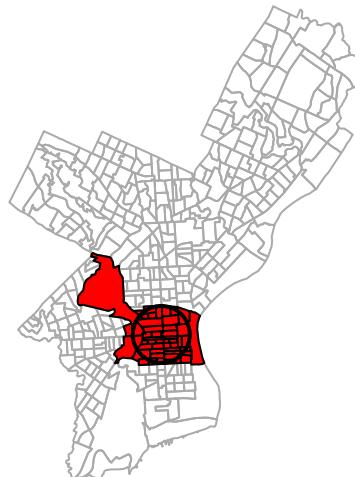
```
#> [1] "sgbp"
```

We have created a `sgbp` object, which is a “Sparse Geomtry Binary Predicate”. It is a so called sparse matrix, which is a list with integer vectors only holding the indices for each polygon that intersects. In our case we only have one vector, because we only intersect with one buffer polygon, so we can extract this first vector with `philly_buf_intersects[[1]]` and use it for subsetting:

```
philly_sel_sf <- philly_sf[philly_buf_intersects[[1]],]
```

```
# plot
plot(st_geometry(philly_sf), border="#aaaaaa", main="Census tracts that fall within 2km of city center")
plot(st_geometry(philly_sel_sf), add=T, col="red")
plot(st_geometry(philly_buf_sf), add=T, lwd = 2)
```

Census tracts that fall within 2km of city center



2.2.2 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.

Create a `SpatialPoints` object with the Philadelphia city center coordinates named `philly_ctr_sp`.

```
coords <- data.frame(x = 1750160, y = 467499.9) # set the coordinates
prj <- CRS("+proj=aea +lat_1=29.5 +lat_2=45.5 +lat_0=37.5 +lon_0=-96 +x_0=0 +y_0=0 +datum=NAD83 +units=m")
philly_ctr_sp <- 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.

```
philly_buf_sp <- gBuffer(philly_ctr_sp, width=2000) # create buffer around center
```

We will use the `gIntersects()` function from the `rgeos` package to select all census tract polygons that intersect with the center buffer. 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. Then we subset the object with the census tract polygons.

```
philly_buf_intersects <- gIntersects(philly_buf_sp, philly_sp, byid=TRUE)
```

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

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

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

2.3 Reprojecting

Occasionally you may have to change the coordinates of your spatial object into a new Coordinate Reference System (CRS). Functions to transform, or *reproject* spatial objects typically take the following two arguments:

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

You can reproject

- a `sf` object with `st_transform()`

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

Let us go back to the "PhillyHomicides" shapefile we exported earlier. Let's read it back in and reproject it so it matches the projection of the Philadelphia Census tracts.

Now let us check the CRS for both files.

```
#If you need to read the file back in:
#philly_homicides_sf <- st_read("data/PhillyHomicides/")

st_crs(philly_sf)

#> Coordinate Reference System:
#>   EPSG: 102003
#>   proj4string: "+proj=aea +lat_1=29.5 +lat_2=45.5 +lat_0=37.5 +lon_0=-96 +x_0=0 +y_0=0 +datum=NAD83"

st_crs(philly_homicides_sf)

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

We see that the CRS are different: we have `+proj=aea...` and `+proj=longlat....`. AEA refers to USA Contiguous Albers Equal Area Conic which is a projected coordinate system with numeric units. We will need this below for our spatial operations, so we will make sure both files are in that same CRS.

We use `st_transform` and assign the result to a new object. Note how we also use `st_crs` to extract the projection definition from `philly_sf`, so we don't have to type it out.

```
philly_homicides_sf_aea <- st_transform(philly_homicides_sf, st_crs(philly_sf))
```

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

```
range(st_coordinates(philly_homicides_sf))

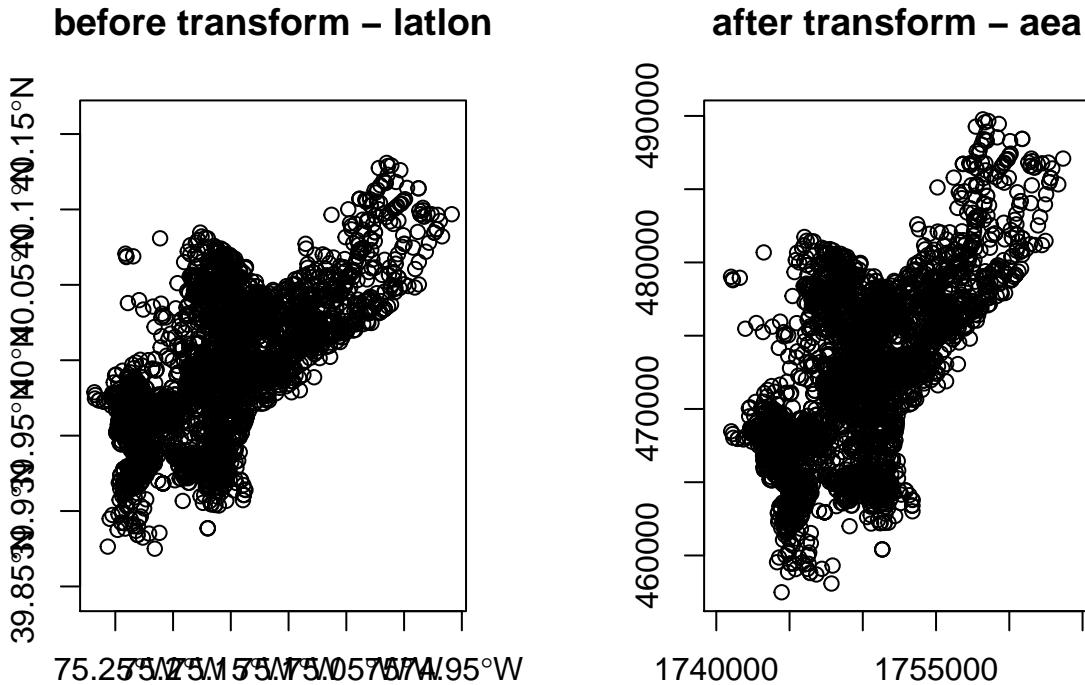
#> [1] -75.26809 40.13086

range(st_coordinates(philly_homicides_sf_aea))

#> [1] 457489.7 1763671.8
```

We can also compare them visually with:

```
par(mfrow=c(1,2))
plot(st_geometry(philly_homicides_sf), axes=TRUE, main = "before transform - latlon")
plot(st_geometry(philly_homicides_sf_aea), axes=TRUE, main = "after transform - aea")
```



Lastly, let us save the reprojected file as `PhillyHomicides_aea` shapefile, as we will use it later on.

```
st_write(philly_homicides_sf_aea, "data/PhillyHomicides_aea", driver = "ESRI Shapefile")
```

2.3.1 For sp

Below is the equivalent for `sp` objects. This is very similar, except that we wrap the `CRS` function around the result of `proj4string`, because `spTransform` requires a CRS object.

```
ph_homic_sp <- readOGR("data/PhillyHomicides/", "PhillyHomicides")
proj4string(philly_sp)
proj4string(philly_homicides_sp)
philly_homicides_sp_aea <- spTransform(philly_homicides_sp, CRS(proj4string(philly_sp)))

## check the coordinates ##
range(coordinates(ph_homic_aea_sp))
range(coordinates(ph_homic_sp))

## write out
writeOGR(philly_homicides_sp_aea, "data/PhillyHomicides_AEA", "PhillyHomicides_AEA", driver = "ESRI Shapefile")
```

2.3.2 Raster reprojection

Here is what it would look like to reproject the HARV raster used earlier to a WGS84 projection. We see that the original projection is in UTM.

```
# 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="+init=epsg:4326")
```

Let's look at the coordinates to see the effect:

```
extent(HARV)
```

```
#> class      : Extent
#> xmin       : 731998.5
#> xmax       : 732766.8
#> ymin       : 4712956
#> ymax       : 4713536
```

```
extent(HARV_WGS84)
```

```
#> class      : Extent
#> xmin       : -72.17505
#> xmax       : -72.16544
#> ymin       : 42.53393
#> ymax       : 42.5394
```

```
ncell(HARV)
```

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

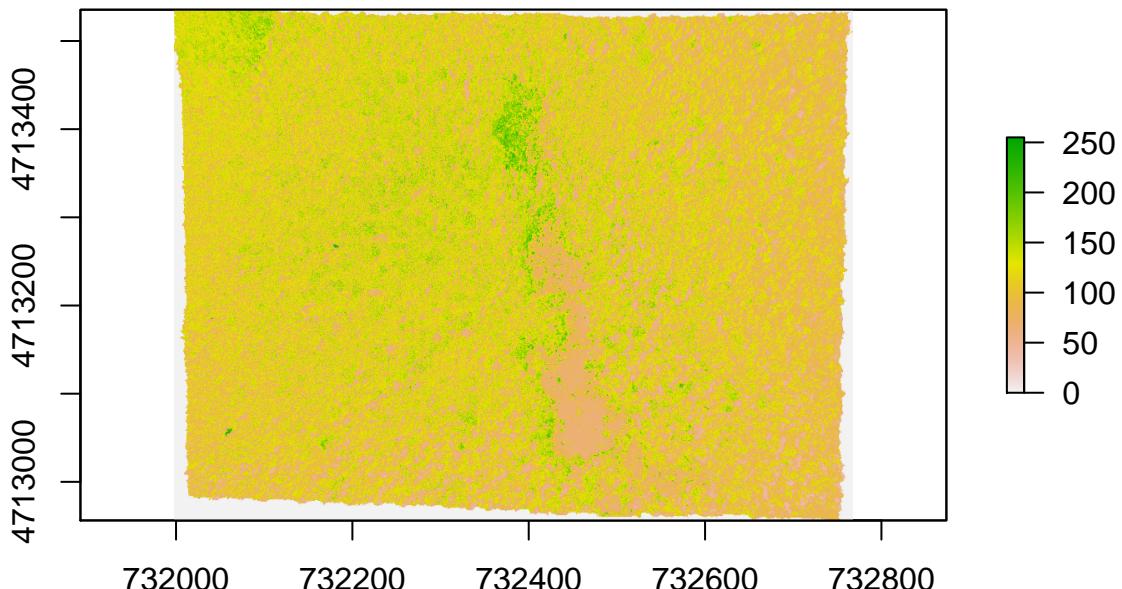
```
ncell(HARV_WGS84)
```

```
#> [1] 7687552
```

And here is the visual proof:

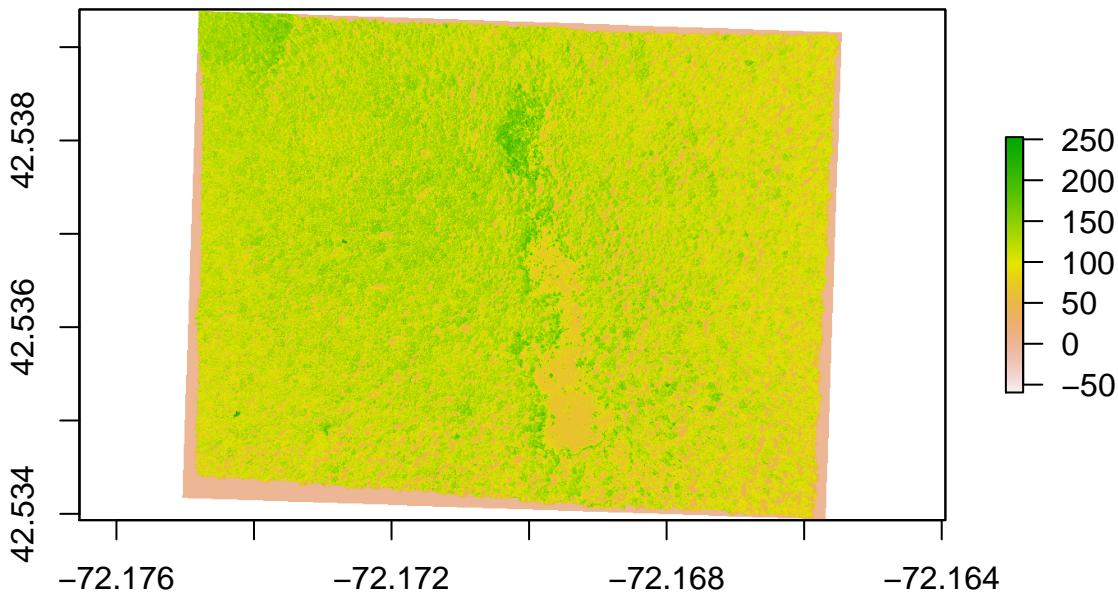
```
plot(HARV, main = "before transform - UTM")
```

before transform – UTM



```
plot(HARV_WGS84, main = "after transform - WGS84")
```

after transform – WGS84



2.4 Spatial Aggregation: Points in Polygons

Now that we have both homicides and census tracts in the same projection we will forge ahead and ask for the density of homicides for **each census tract** in Philadelphia: $\frac{\text{homicides}}{\text{area}}$

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

2.4.1 With sf

We will use piping and build up our object in the following way. First we calculate the area for each tract. We use the `st_area` function on the geometry column and add the result.

```
philly_sf %>%
  mutate(tract_area = st_area(geometry)) %>%
  head()
```

Next, we use `st_join` to perform a spatial join with the points:

```
philly_sf %>%
  mutate(tract_area = st_area(geometry)) %>%
  st_join(philly_homicides_sf_aea) %>%
  head()
```

Now we can group by a variable that uniquely identifies the census tracts, (we choose *GEOID10*) and use `summarize` to count the points for each tract and calculate the homicide rate. Since our units are in sq meter. multiply by by 1000000 to get sq km. We also need to carry over the area, which I do using `unique`.

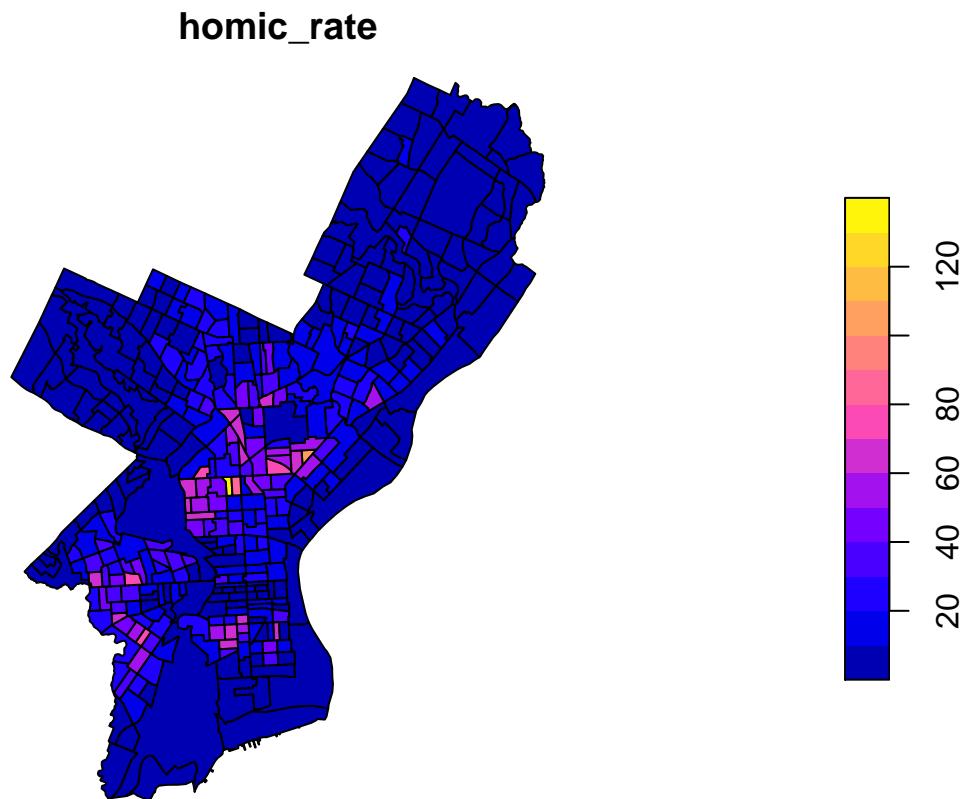
We also assign the output to a new object `crime_rate`.

```
crime_rate <- philly_sf %>%
  mutate(tract_area = st_area(geometry)) %>%
```

```
st_join(philly_homicides_sf_aea) %>%
  group_by(GEOID10) %>%
  summarize(n_homic = n(),
            tract_area = unique(tract_area),
            homic_rate = n_homic/tract_area * 1e6)
```

And here is a simple plot:

```
plot(crime_rate["homic_rate"])
```



Finally, we write this out for later:

```
st_write(crime_rate, "data/PhillyCrimerate", driver = "ESRI Shapefile")
```

2.4.2 With sp

For `sp` objects we can use the `aggregate()` function⁴. 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).

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

⁴There 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.

```
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)
```

Now let us investigate the object we created.

```
class(ph_hom_count_sp)
names(ph_hom_count_sp)
head(ph_hom_count_sp)
```

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.

2.4.3 sp - sf comparison

| how to.. | for sp objects | for sf objects |
|--------------------------|---------------------------------------------------|---------------------------------------------------------------|
| join attributes | <code>sp::merge()</code> | <code>dplyr::*_join()</code> (also <code>sf::merge()</code>) |
| reproject | <code>spTransform()</code> | <code>st_transform()</code> |
| retrieve (or assign) CRS | <code>proj4string()</code> | <code>st_crs()</code> |
| count points in polygons | <code>over()</code> | <code>st_within</code> and <code>aggregate()</code> |
| buffer | <code>rgeos::gBuffer()</code> (separate package) | <code>st_buffer()</code> |
| select by location | <code>g*</code> functions from <code>rgeos</code> | <code>st_*</code> geos functions in <code>sf</code> |

Here are some additional packages that use vector data:

- **stplanr**: Functionality and data access tools for transport planning, including origin-destination analysis, route allocation and modelling travel patterns.
- **bikedata**: Data from public hire bicycle systems, including London, New York, Chicago, Washington DC, Boston, Los Angeles, and Philadelphia

2.5 raster operations

to come

Some helpful packages that deal with raster data:

- `landscapetools` provides utility functions to complete tasks involved in common landscape analysis.
- `getlandsat`: Get Landsat 8 Data from Amazon Public Data Sets
- `MODISTsp`: automates the creation of time series of rasters derived from MODIS Land Products data
- `FedData`: Download geospatial Data from federated data sources, including the The National Elevation Dataset digital elevation models, the Global Historical Climatology Network, the National Land Cover Database, and more.

Chapter 3

Making Maps in R

Learning Objectives

- plot an `sf` object
 - 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.

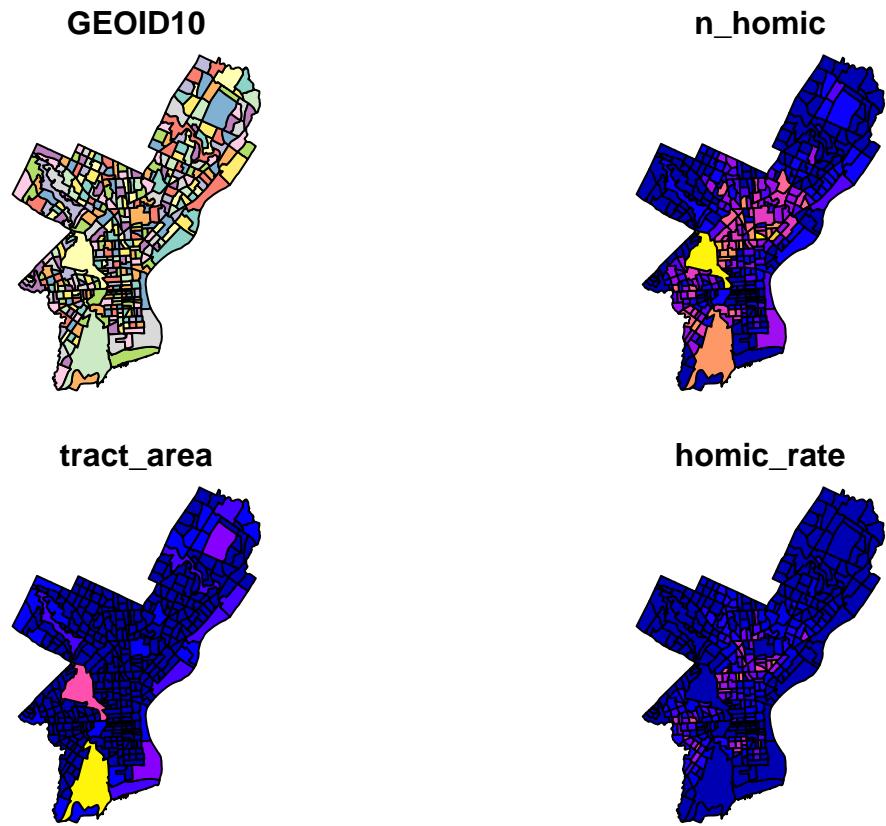
Mapping packages are in the process of keeping up with the development of the new `sf` package, so they typically accept both `sp` and `sf` objects. However, there are a few exceptions.

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 Plotting simple features (`sf`) with `plot`

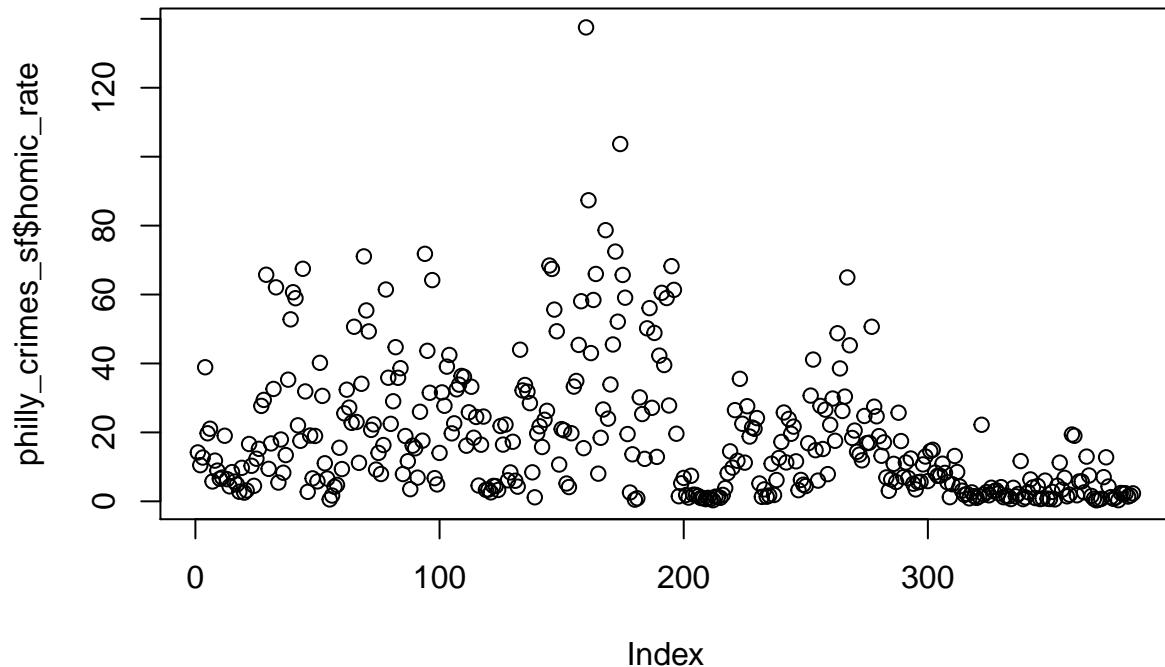
As we have already briefly seen, the `sf` package extends the base `plot` command, so it can be used on `sf` objects. If used without any arguments it will plot all the attributes.

```
philly_crimes_sf <- st_read("data/PhillyCrimerate/", quiet = TRUE)
plot(philly_crimes_sf)
```

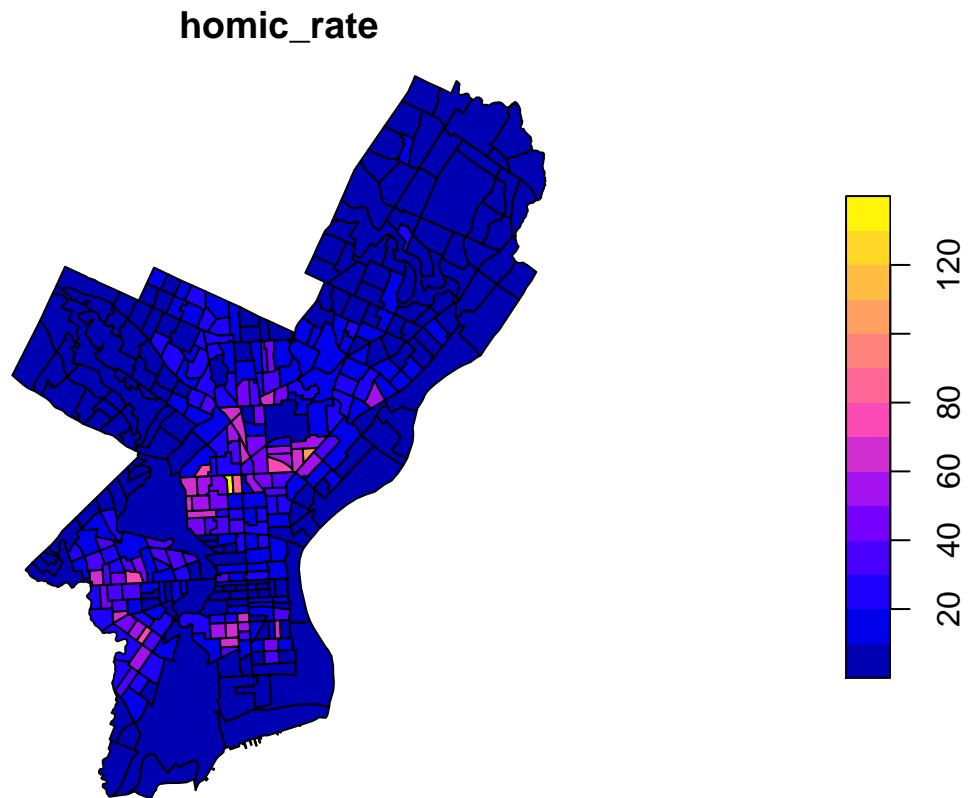


To plot a single attribute we need to provide an object of class `sf`, like so:

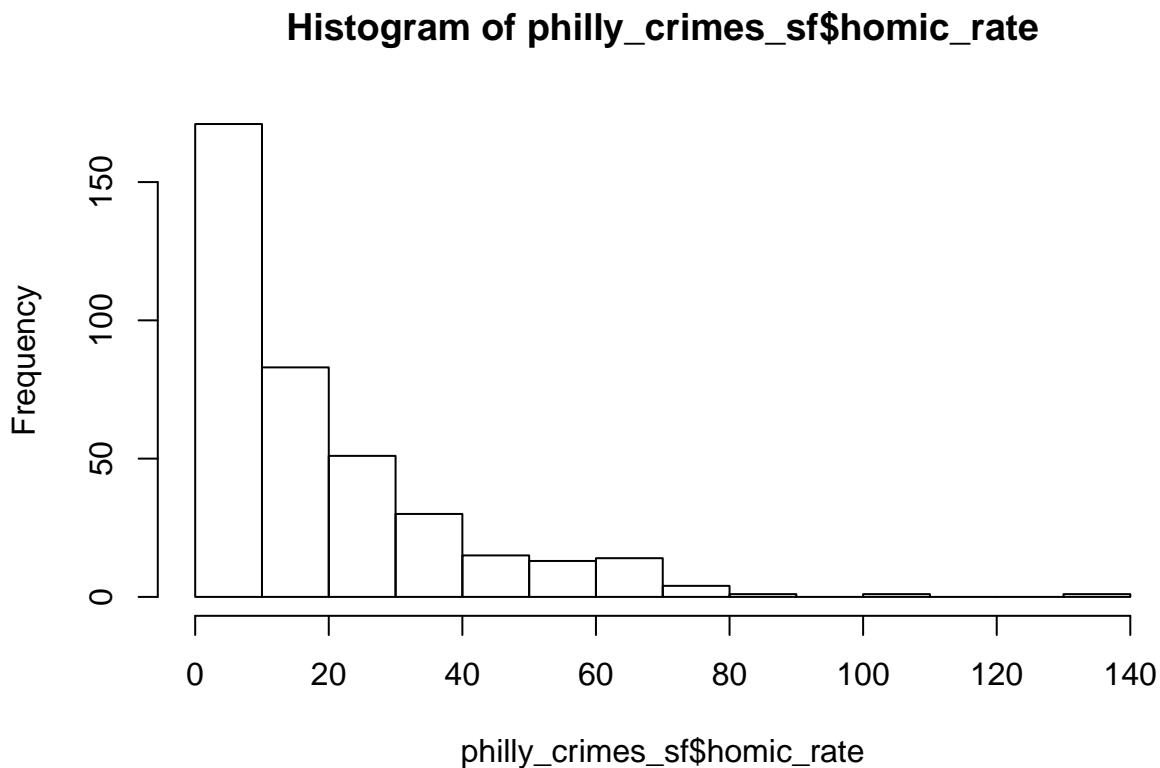
```
plot(philly_crimes_sf$homic_rate) # this is a numeric vector!
```



```
plot(philly_crimes_sf["homic_rate"])
```



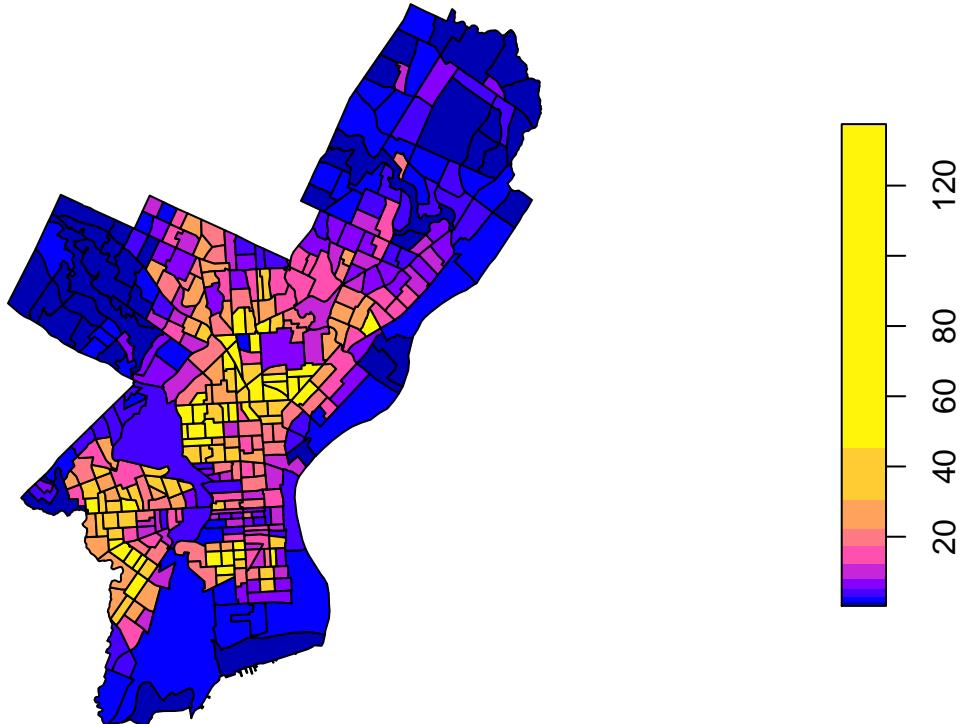
Since our values are unevenly distributed...:



...we might want to set the breaks to quantiles in order to better distinguish the census tracts with low values. This can be done by using the `breaks` argument for the `sf` plot function.

```
plot(philly_crimes_sf["homic_rate"],
     main = "Philadelphia homicide density per square km",
     breaks = "quantile")
```

Philadelphia homicide density per square km



We can change the color palette using a library called `RColorBrewer`¹. For more about ColorBrewer palettes read this.

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

We select 7 colors from the ‘Orange-Red’ palette and assign it to an object `pal`.

```
library(RColorBrewer)
pal <- brewer.pal(7, "OrRd") # we select 7 colors from the palette
class(pal)
```

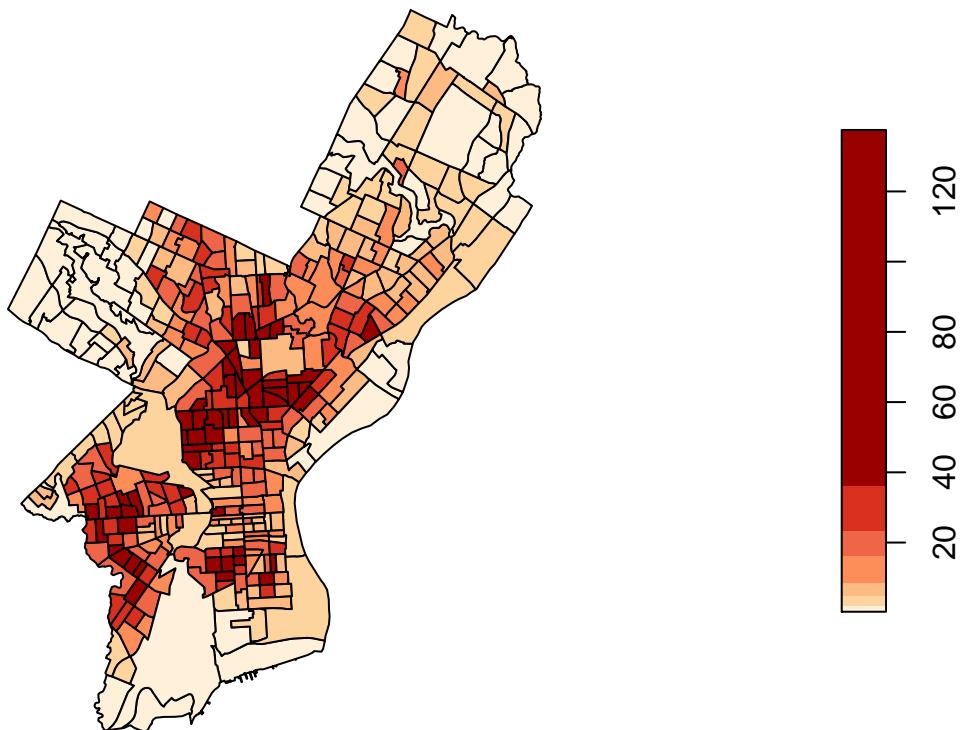
```
#> [1] "character"
```

Finally, we add this to the plot

```
plot(philly_crimes_sf["homic_rate"],
     main = "Philadelphia homicide density per square km",
     breaks = "quantile", nbreaks = 7,
     pal = pal)
```

¹This 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")`.

Philadelphia homicide density per square km



3.2 Choropleth mapping with spplot

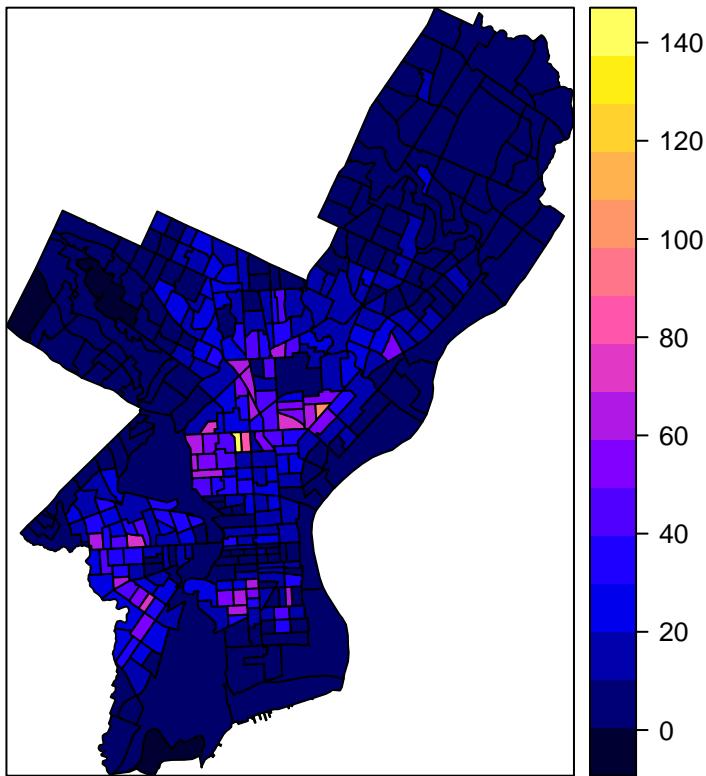
`sp` comes with a plot command `spplot()`, which takes `Spatial*` objects to plot. `spplot()` is one of the earlier functions around to plot geographic objects.

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

```
#> [1] "GEOID10"      "n_homic"      "tract_area"   "homic_rate"
```

Like `plot`, by default `spplot` maps all everything it can find in the attribute table. Sometimes this does not work, depending on the data types in the attribute table. 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_crimes_sp, "homic_rate")
```



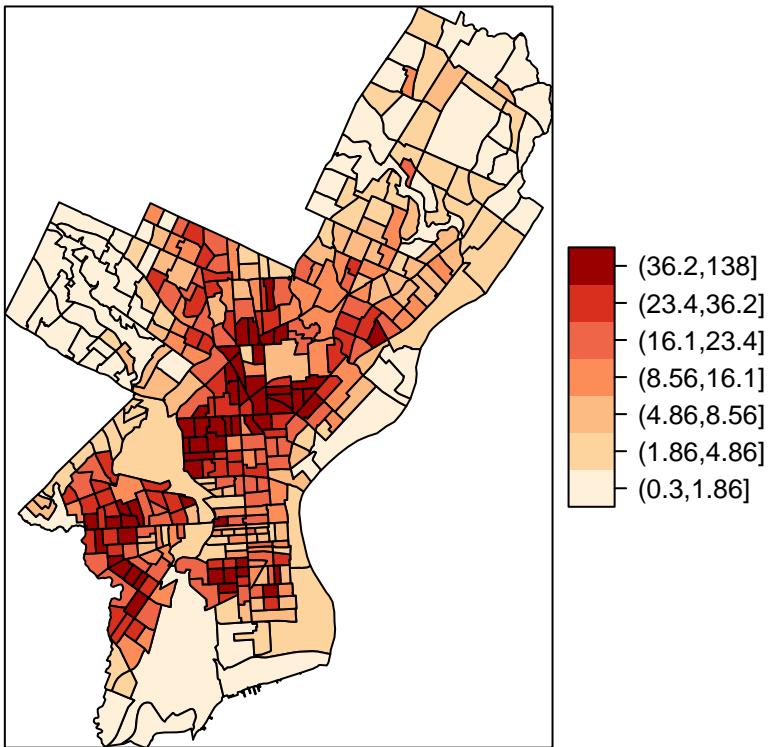
Many improvements can be made here as well, below is an example² ³.

```
# quantile breaks
breaks_qt <- classIntervals(philly_crimes_sp$homic_rate, n = 7, style = "quantile")
br <- breaks_qt$brks
offs <- 0.0000001
br[1] <- br[1] - offs
br[length(br)] <- br[length(br)] + offs
# categories for choropleth map
philly_crimes_sp$homic_rate_bracket <- cut(philly_crimes_sp$homic_rate, br)
# plot
spplot(philly_crimes_sp, "homic_rate_bracket", col.regions=pal, main = "Philadelphia homicide density p")
```

²For more details see Chaps 2 and 3 in Applied Spatial Data Analysis with R. Also, `spplot` is a wrapper for the `lattice` package, see there for more advanced options.

³For 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>

Philadelphia homicide density per square km



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)⁴.

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()
```

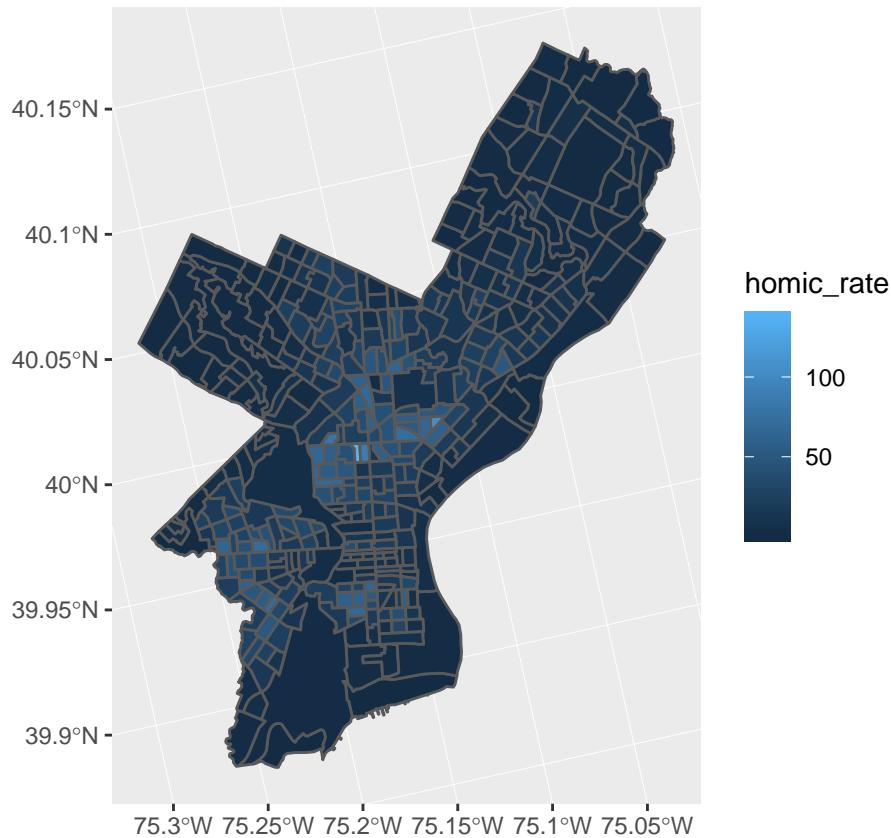
Or shorter:

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

The great news is that `ggplot` can plot **sf objects directly** by using `geom_sf`. So all we have to do is:

⁴See Wilkinson L (2005): “The grammar of graphics”. Statistics and computing, 2nd ed. Springer, New York.

```
ggplot(philly_crimes_sf) +
  geom_sf(aes(fill=homic_rate))
```



Homicide rate is a continuous variable and is plotted by `ggplot` as such. If we wanted to plot our map as a ‘true’ choropleth map we need to convert our continuous variable into a categorical one, according to whichever brackets we want to use.

This requires two steps:

- Determine the quantile breaks.
- Add a categorical variable to the object which assigns each continuous value to a bracket.

We will use the `classInt` package to explicitly determine the breaks.

```
library(classInt)

# get quantile breaks. Add .00001 offset to catch the lowest value
breaks_qt <- classIntervals(c(min(philly_crimes_sf$homic_rate) - .00001, philly_crimes_sf$homic_rate), 10)

breaks_qt

#> style: quantile
#> [0.2997339,1.856692]  [1.856692,4.814503]  [4.814503,8.495498)
#>      55                  55                  55
#> [8.495498,16.14234]  [16.14234,23.20642]  [23.20642,36.16089)
#>      55                  55                  55
#> [36.16089,137.5028]  [137.5028,137.5028]  [137.5028,137.5028]
#>      55
```

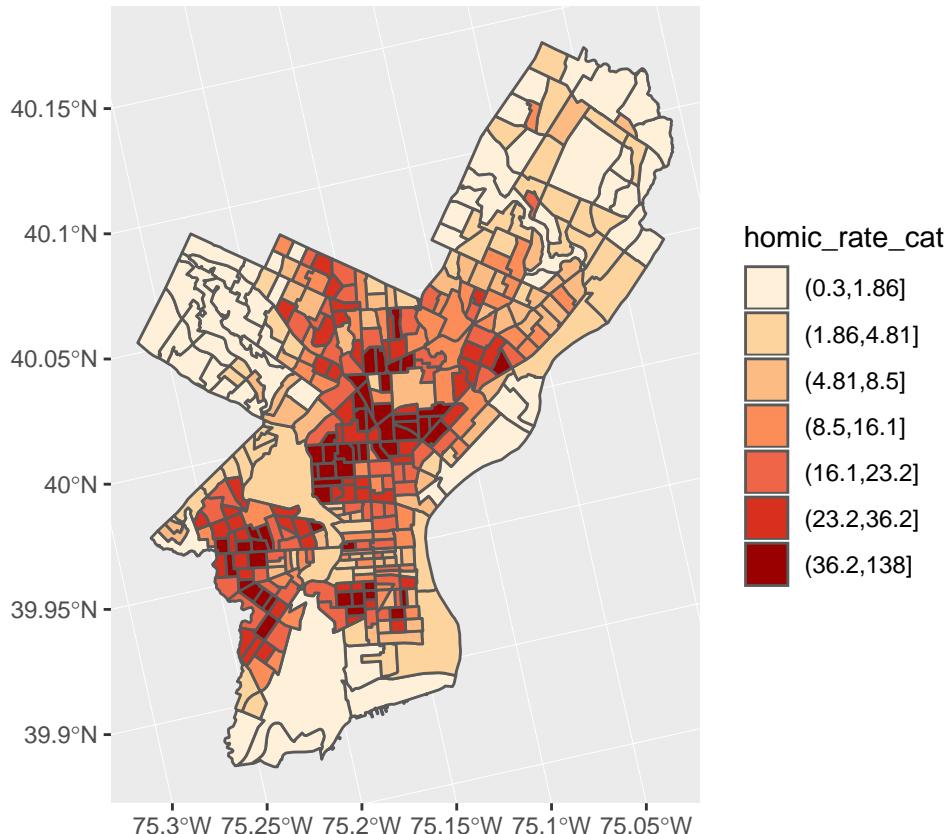
Ok. We can retrieve the breaks with `breaks$brks`.

We use `cut` to divide `homic_rate` into intervals and code them according to which interval they are in.

Lastly, we can use `scale_fill_brewer` and add our color palette.

```
philly_crimes_sf <- mutate(philly_crimes_sf, homic_rate_cat = cut(homic_rate, breaks_qt$brks))

ggplot(philly_crimes_sf) +
  geom_sf(aes(fill=homic_rate_cat)) +
  scale_fill_brewer(palette = "OrRd")
```



3.4 Adding basemaps with `ggmap`

`ggmap` builds on `ggplot` and allows to pull in tiled basemaps from different services, like Google Maps, OpenStreetMaps, or Stamen Maps⁵.

So let's overlay the map from above on a terrain map we pull from Stamen.

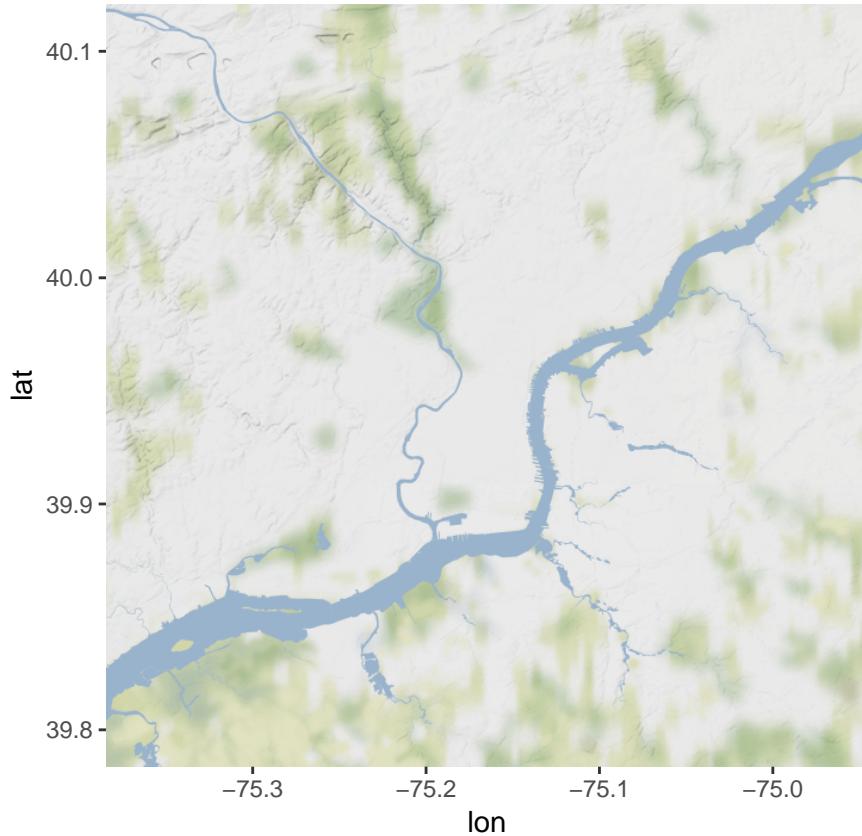
First we use the `get_map()` command from `ggmap` to pull down the basemap. We need to tell it the location or the boundaries of the map, the zoom level, and what kind of map service we like (default is Google terrain). It will actually download the tile. `get_map()` returns a `ggmap` object, name it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

```
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 = 'terrain-background')
```

⁵Google now requires an API key. 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.

```
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() +
```

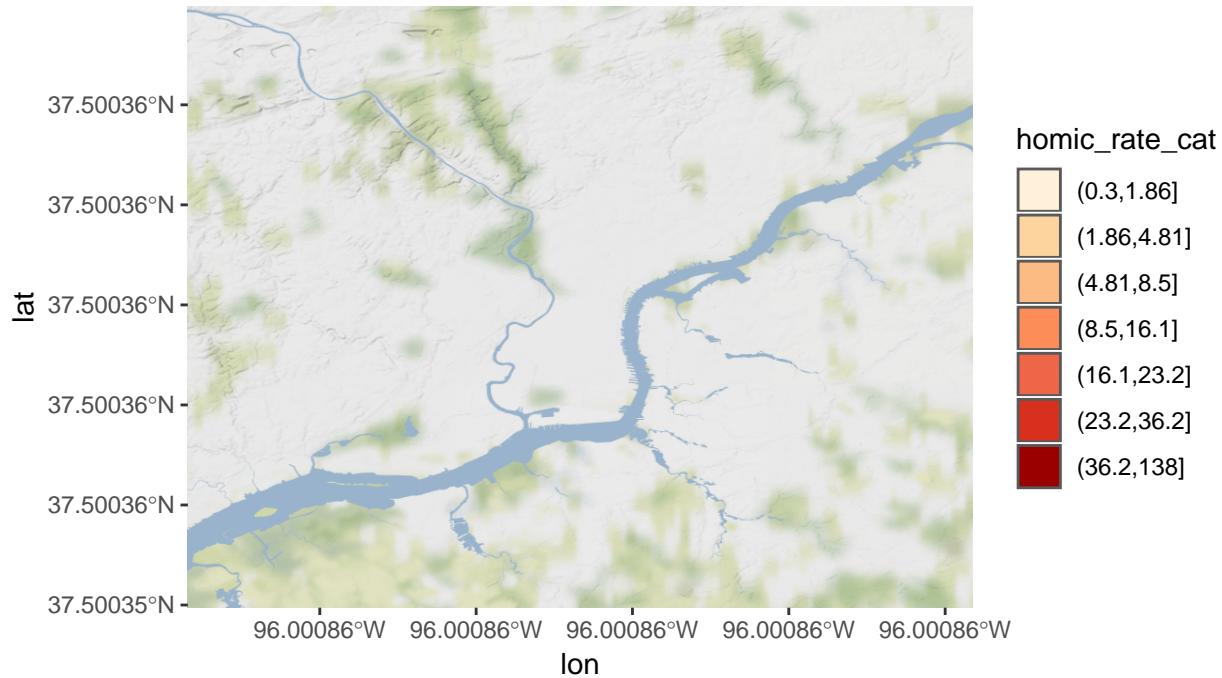
with the line to call our basemap:

```
ggmap(ph_basemap) +
```

We also have to set `inherit.aes` to FALSE, so it overrides the default aesthetics (from the ggmap object).

```
ggmap(ph_basemap) +
  geom_sf(data = philly_crimes_sf, aes(fill=homic_rate_cat), inherit.aes = FALSE) +
  scale_fill_brewer(palette = "OrRd")
```

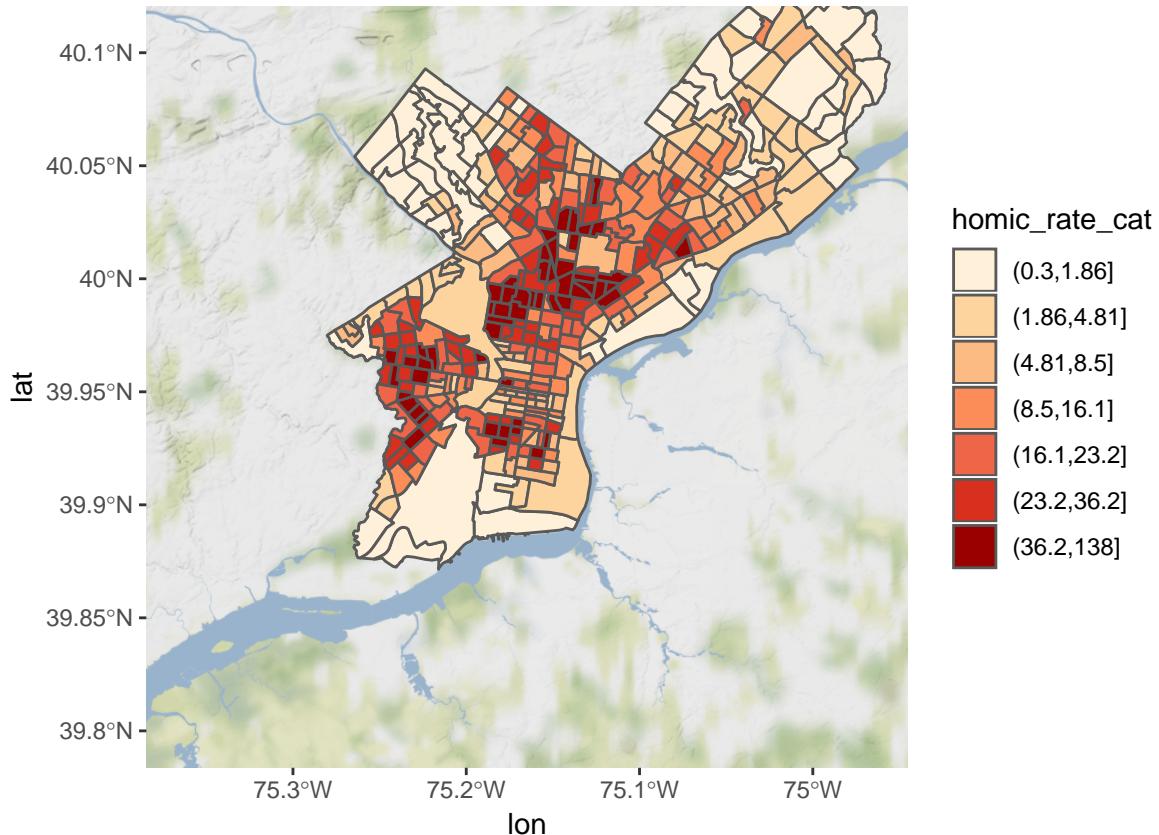
```
#> Coordinate system already present. Adding new coordinate system, which will replace the existing one
```



Oops. Any idea what might be going on?

We need to set our CRS to WGS84, which is the one the tiles are downloaded in. We can add `coord_sf` to do this:

```
gmap(ph_basemap) +
  geom_sf(data = philly_crimes_sf, aes(fill=homic_rate_cat), inherit.aes = FALSE) +
  scale_fill_brewer(palette = "OrRd") +
  coord_sf(crs = st_crs(4326))
```



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

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

#> PhantomJS not found. You can install it with `webshot::install_phantomjs()`. If it is installed, please

`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")
tmap_last()
```

Cool huh?

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

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 an `sp` or `Spatial*` 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`.

And while `tmap` was tolerant about our AEA projection of `philly_crimes_sf`, `leaflet` does require us to explicitly reproject the `sf` object.

```
library(leaflet)

# reproject
philly_WGS84 <- st_transform(philly_crimes_sf, 4326)

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_rate` 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 construct 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_rate` 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_rate)

leaflet(philly_WGS84) %>%
  addPolygons(
    stroke = FALSE, # remove polygon borders
    fillColor = ~pal_fun(homic_rate), # 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_rate),
    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_rate),
    fillOpacity = 0.8, smoothFactor = 0.5,
    popup = p_popup) %>%
  addTiles() %>%
  addLegend("bottomright", # location
            pal=pal_fun, # palette function
            values=~homic_rate, # 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 breaks⁶.

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_rate),
    fillOpacity = 0.8, smoothFactor = 0.5,
    popup = p_popup) %>%
  addTiles() %>%
  addLegend("bottomright",
            colors = brewer.pal(7, "YlOrRd"),
            labels = paste0("up to ", format(breaks_qt$brks[-1], digits = 2)),
            title = 'Philadelphia homicide density per sqkm')
```

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(
```

⁶The 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**).”

```
stroke = FALSE,
fillColor = ~pal_fun(homic_rate),
fillOpacity = 0.8, smoothFactor = 0.5,
popup = p_popup,
group = "philly") %>%
addTiles(group = "OSM") %>%
addProviderTiles("CartoDB.DarkMatter", group = "Carto") %>%
addLegend("bottomright",
  colors = brewer.pal(7, "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.